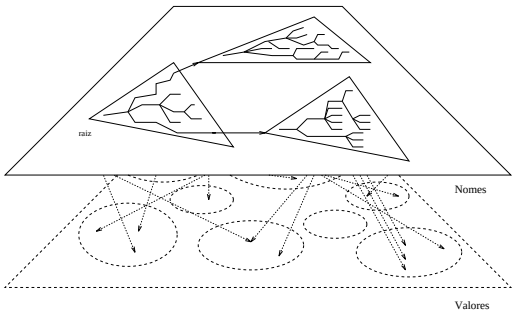
UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO



Reutilização de Componentes de Software com Base em Identificadores Hierárquicos

Pedro Manuel Guerra e Silva Reis dos Santos

(MESTRE)

Dissertação para a obtenção do Grau de Doutor em

Engenharia Informática e de Computadores

Julho, 1999

Título: Reutilização de Componentes de Software com Base em Identificadores Hierárquicos

Nome: Pedro Manuel Guerra e Silva Reis dos Santos

Doutoramento em: Engenharia Informática e de Computadores

Orientador: Rui Gustavo Nunes Pereira Crespo

Provas concluídas em: Julho de 1999

Resumo

Para aumentar a produtividade do desenvolvimento de software é necessário, além de melhores processos e técnicas de desenvolvimento, encontrar formas de escrever menos software. A reutilização surge, de momento, como uma das soluções técnicas viáveis. A institucionalização da reutilização de software deve considerar factores humanos e técnicos, bem como a harmonização entre eles. Com este objectivo identificam-se nesta dissertação três componentes: uma metodologia de reutilização, um ambiente de reutilização e um modelo unificador.

A metodologia de reutilização oferece, sem impor um processo rígido, uma variedade de modos de proceder que permitem, por composição, adaptar-se às necessidades e comportamentos individuais. O ambiente de reutilização é constituído por um conjunto de mecanismos de classificação, selecção, especialização e integração de componentes, e por uma linguagem que permite sequenciar e combinar os componentes de diversas formas. O modelo unificador oferece uma só representação de abstracções com base na organização hierárquica de identificadores. O modelo unificador permite classificar hierarquicamente e seleccionar os componentes com base em identificadores extraídos dos próprios componentes, bem como a subsequente especialização e integração dos componentes na aplicação.

A escolha de identificadores possibilita a automatização da classificação e é pouco sensível a sinónimos. A hierarquia oferece uma boa granularidade descritiva e permite uma selecção precisa do ponto de vista comparativo e evolutivo. A linguagem facilita a integração através de uma representação uniforme da hierarquia de dados e da pilha de

execução. O sistema proposto permite obter resultados conclusivos de uma forma rápida, pois o processo de extracção pode ser automatizado e a análise concentra-se num sub-conjunto da informação extraída.

Palavras chave: reutilização de software, classificação de software, componentes de software, selecção de componentes para reutilização, adaptação de componentes, linguagens de interligação de módulos.

Title: Software Components Reuse Based on Hierarchical Identifiers

Abstract

In order to increase software development productivity it is necessary to find ways to write less software, besides improving development techniques and processes. At this moment, reuse appears as the one of technically viable solutions. Institutionalizing software reuse requires technical skills and human factors, as well as their inter-relation. In this dissertation three parts were identified to achieve such goals: a reuse methodology, a reuse environment and unifying model.

A reuse methodology offers a wide variety of procedure forms to adapt, by composition, to individual needs and behaviors, without imposing a fixed process. The reuse environment is a set of component classification, selection, specialization, and integration mechanisms, and a language to sequence operations and combine components. The unifying model enables a consistent representation of abstractions based on a hierarchical organization of identifiers. The unifying model allows the hierarchical classification and selection using identifiers extracted from the components themselves, and the following specialization and integration of each component into the final application.

The use of identifiers as classifiers automates the classification process and produces a classification scheme that is almost insensitive to synonyms. The selection process is precise from a comparative and evolutive analysis due to the descriptive granularity of the hierarchical classification. The language make the integration process simpler since it uses a uniform hierarchical representation of data and an execution stack. The proposed system allows the user to obtain conclusive results quickly, since the analysis is performed on sub-set of the classified information and to the automated classification process.

Keywords: software reuse, software classification, software components, selection of reusable components, component adaptation, module interconnection languages.

Agradecimentos

Pela contribuição dada para a realização deste trabalho, pretendo expressar o meu reconhecimento muito particular:

Ao Prof. Rui Crespo na qualidade de meu orientador científico e, muito especialmente, pela sua constante disponibilidade e encorajamento. O seu acompanhamento empenhado tem sido fundamental para a minha formação científica e profissional.

Aos Prof. Carlo Ghezzi e Prof. Mansur Samadzaded pelos comentários e sugestões que contribuiram em muito para a conclusão deste trabalho.

Aos Prof. Guilherme Arroz e Prof. Cunha e Serra pelo incentivo e alento dispensados, muito em particular nos momentos mais delicados deste trabalho.

Ao Andre Zúquete e Rui Casteleiro por todo o apoio e encorajamento que me deram durante a realização deste trabalho.

À minha mãe pelos conselhos dados relativamente à escrita e pela revisão linguística de toda a dissertação.

Ao Instituto Superior Técnico pelas condições de trabalho que facultou e, em especial, à secção de sistemas digitais pelo suporte informático imprescindível à realização deste trabalho e pelo apoio financeiro concedido para deslocações a encontros científicos para publicação dos trabalhos desenvolvidos.

Por fim, uma referência muito especial àqueles que comigo padeceram e pacientemente me apoiaram e a quem dedico esta dissertação, à Isabel e ao Gonçalo.

Lisboa, Julho de 1999

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A revolução industrial teve, no século passado, uma enorme importância económica e social. A partir dessa altura as pessoas passaram de uma tradição rural de auto-suficiência para a inter-dependência da vida nas grandes cidades. As indústrias utilizavam longas linhas de montagem que exigiam grandes quantidades de mão de obra. Os indivíduos deixaram de ser auto-dependentes passando a depender do treino específico necessário para desempenhar as tarefas repetitivas requeridas para operar numa secção específica da linha de montagem. As exigências do mercado obrigavam a uma constante actualização do equipamento e o consequente treino dos operários. A actualização é necessária para garantir o mercado mas implica perda de tempo. Esta perda de tempo acrescida pelas interrupções devidas a avarias das máquinas, após longos períodos de uso, podem comprometer a competitividade. O estudo do tempo médio entre avarias das máquinas, a sua fiabilidade e eficiência deram origem à *engenharia industrial*. Hoje em dia, os robots industriais tendem a substituir as pessoas nas linhas de montagem: por um lado, os custos necessários para treinar os operários eram muito avultados, por outro, a dificuldade em treinar uma pessoa para uma nova tarefa era maior depois de a pessoa trabalhar durante muitos anos numa mesma tarefa [Rin91].

A revolução da informação pode ser considerada a segunda grande revolução da era moderna [Cox90]. O resultado imediato da revolução da informação tem sido a sucessiva automatização de diversas tarefas. Estamos cada vez mais dependentes dos computadores no trabalho, no entretenimento e em muitos outros aspectos da nossa vida

quotidiana. Além disso, estamos a confiar aos computadores tarefas cada vez mais importantes, algumas delas críticas [McF89, Pri97]. Com os benefícios, já comprovados em diversas áreas, de uma sociedade computorizada pela revolução da informação, surge também a enorme tarefa de organizar e manter actualizada a informação [Rin91, BB91].

O software moderno é constituído por milhares de aplicações desenhadas para responder a problemas concretos. À medida que os problemas evoluem, as aplicações vão sendo corrigidas sucessivamente na tentativa de as manter a funcionar com a satisfação dos utilizadores. Tais procedimentos raramente produzem software fiável, de uma forma controlada, documentada e eficiente. Em resposta a estes problemas, denominados "crise do software" [Ran96], surgiu a *engenharia de software*. É hoje geralmente aceite que a manutenção e desenvolvimento de software consistem em atacar os problemas de reutilização de software. A noção de reutilização é conhecida há muito tempo. Já durante a revolução industrial as indústrias estavam preocupadas com a reutilização quer das linhas de montagem e dos seus componentes, quer das pessoas que trabalhavam nessas linhas de montagem. Na revolução da informação o problema chave é a reutilização de sistemas de software e componentes numa gama variada de domínios de aplicação [Weg84, L.G98, Sta94].

1.1 Perspectiva histórica

De um ponto de vista histórico podemos dividir os mais de 40 anos de desenvolvimento de software em três períodos. O primeiro período decorre até 1968 e pode ser classificado de pré-histórico. O segundo período surge com a "crise do software" e o nascimento da engenharia de software. O terceiro período começa cerca de 1983 com o aparecimento dos computadores pessoais e o domínio do software sobre o hardware.

Durante o primeiro período o principal objectivo do desenvolvimento de software consistia no aproveitamento dos limitados recursos de hardware. Muito do esforço de desenvolvimento era dispendido a optimizar código, principalmente escrito em linguagem máquina. Os primeiros compiladores provaram que para a maioria das aplicações era possível gerar código com bom desempenho a partir de linguagens de alto nível,

como o **FORTRAN** e o **COBOL** [Seb93]. Seguidamente surgiu a compilação incremental, a alteração do código fonte, a depuração de programas ("*debugging*"). Verificou-se, então, que era necessário fazer manutenção dos programas e que o desenvolvimento de software exige a atenção dos projectistas noutros domínios para além da codificação.

O segundo período começa por se preocupar com os insucessos nos projectos de software e as suas causas. É introduzido o modelo do processo de queda de água e vários melhoramentos do mesmo. Tal facto resulta da compreensão que a qualidade não pode ser assegurada exclusivamente por uma análise do produto final. A qualidade do software deve compreender todo o processo de desenvolvimento, em especial as fases iniciais. Os métodos formais foram uma das formas utilizadas para aumentar a confiança no software e aumentar a produtividade [Kem90]. Estes tiveram sucesso na especificação e verificação de sistemas críticos, de pequenas dimensões. Para projectos maiores, verificou-se que nem os requisitos nem o desenho podiam ser expressos usando um modelo matemático sucinto, como, por exemplo, o desempenho, a compatibilidade, os relacionamentos com dados exteriores e outras formas de requisitos não funcionais [Hal90]. Acrescente-se ainda a dificuldade de comunicação com leigos devido à linguagem e notação utilizada. Este factor é especialmente importante quando as aplicações dependem de um forte factor humano [End96].

O terceiro período começa com o aparecimento das estações de trabalho e com a vulgarização dos terminais gráficos. O subsequente aparecimento das interfaces gráficas e das aplicações gráficas permitiu disponibilizar aos projectistas ferramentas CASE. No entanto, a tecnologia CASE teve sucesso apenas numa gama limitada de aplicações muito estudadas. A necessidade de gerar e modificar código obrigava a propagar tais alterações para o desenho. Por outro lado, a separação entre a modelação dos dados e dos processos dificultava manutenção e descrição das inter-relações entre os processos e os dados. Os princípios da orientação para objectos, de que se salienta o encapsulamento, herança e polimorfismo, resultam de uma preocupação directa com a reutilização e manutenção de software. Estes princípios foram primeiro aplicados à fase de codificação e só posteriormente às fases de desenho e de análise, facilitando a transição entre as diversas fases de desenvolvimento.

1.2 Reutilização de software

A noção elementar de reutilização de software existe há muito tempo, tendo começado com a noção de sub-rotina. Reutilização de software consiste em utilizar software existente para construir novos sistemas. Reutilização não é só aplicável a fragmentos de código fonte mas a todo o trabalho gerado durante o processo de desenvolvimento de software. A informação susceptível de ser reutilizada inclui a análise de requisitos, especificações do sistema, estruturas de desenho, e qualquer informação que seja necessária no processo de desenvolvimento. McIlroy [Ran96] propôs a criação de uma indústria de componentes normalizados com os quais pretendia construir aplicações complexas através de pequenos blocos disponíveis por catálogo. Esta ideia esteve na origem da 'fábrica de software' onde um conjunto de procedimentos permitia o desenvolvimento consistente e repetitivo de software [Weg84]. Esperava-se que os programadores reutilizassem tudo o que estivesse disponível, desde pedaços de código a experiências anteriores, mas a reutilização não fazia explicitamente parte do processo. Com a criação de aplicações de grandes dimensões, a partir do início da década de 80, a reutilização de software sofreu um impulso decisivo. Vários avanços foram conseguidos em bibliotecas, técnicas de classificação, criação e distribuição de componentes reutilizáveis, ambientes de apoio à reutilização, entre outros. Mesmo assim o grau de reutilização mantinha-se muito aquém das expectativas. Trabalhos recentes têm focado factores não exclusivamente técnicos considerando-se, actualmente, que estes problemas são tão importantes como os técnicos e talvez mais difíceis de resolver [FI94, Faf94, Lim94]. Alguns dos factores mais importantes incluem problemas comportamentais, organizacionais, culturais, económicos, judiciais e até morais. As novas aproximações procuram integrar todos estes factores numa reutilização institucionalizada.

1.2.1 Impedimentos à reutilização

A construção de aplicações com base em componentes reutilizáveis não necessita seguir uma aproximação radicalmente diferente da adoptada na construção sem reutilização [BAB+87, PDF87, Sta94]. Uma das críticas ao modelo do processo de software de

cascata tem origem no facto de cada fase deste modelo ser principalmente influenciada pelas fases anteriores. Este facto pressupõe uma abordagem de desenvolvimento em que cada fase se baseia na anterior, enquanto a existência de componentes reutilizáveis obriga a avaliar, em avanço, as possibilidades de reutilização e tirar partido delas. Tal situação resulta, principalmente, da reutilização de componentes de pequena dimensão, em geral código, e que não atravessaram as fases de análise e desenho, ou não existe documentação dessas fases. Se esta informação existir nas bibliotecas de software, em conjunto com o código, as oportunidades de reutilização podem ser detectadas e avaliadas pelos analistas, sem necessitar de inspeccionar o respectivo código. O desenvolvimento baseado em objectos utiliza uma mistura das abordagens descendentes e ascendentes, sendo considerado um veículo privilegiado na institucionalização da reutilização no desenvolvimento de software. Este desenvolvimento inclui a produção de novas classes, que podem ser especializadas e reutilizadas, numa abordagem ascendente, e o desenvolvimento da aplicação por composição dessas classes, numa abordagem descendente.

O desenvolvimento de aplicações com base na reutilização de componentes deve contrariar as razões reconhecidas como impedimentos à reutilização (ver figura 1.4). Para tal é necessário satisfazer sete condições que constituem a cadeia de sucesso na reutilização:

1. **Empenho.** A principal razão que está na base da baixa taxa de reutilização que se verifica deve-se à inexistência de qualquer esforço para reutilizar. A indiferença em reutilizar tem origem em razões como a falta de incentivo, limitações temporais, considerar os benefícios da reutilização duvidosos, falta de educação, falta de recursos técnicos, falta de suporte na organização ou falta de apoio por parte da gestão entre outros.

2. **Existência.** O facto de não existirem partes isoladas, sob a forma de componentes, quer tenham sido desenhados para reutilização ou não, representa o principal impedimento. A falta de incentivo económico para produzir tal componente ou o facto de exigir uma tecnologia mais recente para o produzir são também razões apontadas para inexistência de certos componentes.

3. **Disponibilidade.** A falta de repositórios e a limitação do seu uso são as principais razões que estão na origem da falta de disponibilidade do componente. A falta de acesso ao código fonte ou a dificuldade em analisar em profundidade o componente são também razões que conduzem a que o componente não seja considerado para reutilização.

4. **Acessibilidade.** A má ou insuficiente representação e classificação do componente ou a fraca qualidade das ferramentas de busca são as principais razões para não encontrar o componente. A dificuldade ou incapacidade para especificar o que procurar pode também conduzir a que o componente desejado nunca seja encontrado.

5. **Legibilidade.** A dificuldade em compreender o componente, seja por documentação insuficiente ou devido a uma excessiva complexidade do componente, estão na origem da sua desconsideração para reutilização.

6. **Validação.** A falta de teste do componente, o baixo desempenho, a não adesão a padrões de normalização ou a falta de suporte por parte do produtor do componente são consideradas as principais razões para a rejeição do componente na validação.

7. **Integrabilidade.** A existência de incompatibilidade de hardware e ambiente de integração são as razões mais frequentes que impedem a integração do componente. A necessidade de modificações muito extensas e trabalhosas, bem como a dependência de software exterior podem também conduzir à eliminação do componente no desenvolvimento de determinada aplicação.

1.2.2 Taxinomia de reutilização

A reutilização de software pode ser observada de diversas formas. Podem ser identificadas, pelo menos, seis perspectivas através das quais se pode compreender a reutilização de software [PD93]. Estas perspectivas, conceptualmente independentes, são úteis na análise dos problemas práticos e na identificação dos rumos de investigação:

Tabela 1.1: Perspectivas na reutilização de software.

substância	*visibilidade*	*modo*	*técnica*	*intenção*	*produto*
conceito	vertical	planeado	composição	caixa preta	código fonte
componente	horizontal	oportunista	generativo	caixa branca	desenho
processo					especificações
					objectos
					texto
					arquitecturas

Perspectiva de Substância A perspectiva de substância reflecte a aproximação seguida na resolução do problema, e que pode ser conceptual, processual ou objectiva. Os **Conceitos** formais, tais como soluções genéricas para uma classe de problemas, por exemplo, algoritmos genéricos, já atingiram a maturidade. No entanto, são necessários catálogos e informação detalhada indicando quando e como reutilizar os algoritmos. **Componentes** são pequenos grupos de objectos com uma interface bem definida que podem comunicar, sem restrições, com outros componentes. Linguagens como **Ada** [Seb93], ambientes como o **Eiffel** [Mey88] e ferramentas como o **Motif** [Ber91] são alguns exemplos. Do ponto de vista da reutilização, a adaptabilidade dos componentes bem como a sua qualidade e fiabilidade são ainda objecto de estudo. **Processos** de reutilização concentram-se não nos dados mas nos procedimentos que conduzem à reutilização. Estes processos de desenvolvimento podem ser encadeados para produzir processos mais complexos. Embora apresentem boas perspectivas de reutilização, estes processos têm sido utilizados informalmente.

Perspectiva de Visibilidade A perspectiva da visibilidade relaciona a classe de problemas a resolver com a área de aplicação dos mesmos. A reutilização diz-se **vertical** se é efectuada no mesmo domínio ou na mesma área de aplicação. O objectivo da verticalidade consiste em obter modelos genéricos que possam ser utilizados como moldes para produzir novos sistemas da mesma família. A investigação tem focado a identificação e desenvolvimento de modelos para domínios específicos [SJ85]. Embora a reutilização vertical tenha tido uma aplicação informal, a sua aplicação tem tido sucessos

em áreas como as interfaces gráficas e os sistemas operativos. O objectivo da reutilização **horizontal** é a utilização de partes genéricas em aplicações de domínios diferentes. Além do empacotamento e apresentação das partes, a atenção da reutilização horizontal tem-se virado para o desenvolvimento e acesso a bibliotecas, bem como a catalogação e classificação dos componentes nelas armazenados. Bibliotecas de rotinas científicas ou ferramentas **UNIX** [Ker84] são alguns dos exemplos mais conseguidos de reutilização horizontal.

Perspectiva de Modo A perspectiva de modo caracteriza a maturidade do processo de reutilização. A reutilização **planeada** baseia-se numa prática sistemática e planeada de orientações e procedimentos, cujo desempenho é verificado pela recolha de um conjunto de métricas. Os modelos de maturidade para reutilização estão divididos em cinco níveis de maturidade crescente: ad-hoc, repetível, portável, arquitectural e sistemático. Os níveis mais elevados exigem um considerável investimento inicial e empenho, sendo no entanto difícil de prever se o investimento será compensado [FI94]. O maior problema com esta forma de reutilização é precisamente a incerteza financeira que resulta da falta de modelos económicos fiáveis e validados. A reutilização **oportunista**, na qual os componentes são seleccionados de bibliotecas genéricas, representa a quase totalidade da reutilização de software na prática. Neste ambiente a reutilização é exercida a nível individual, sem recurso a procedimentos e utilizando bibliotecas de componentes que não foram desenhados para serem reutilizados. O estudo nesta área tem-se centrado no desenvolvimento de mais e melhores bibliotecas de componentes desenhados para reutilização, com interfaces acessíveis e mecanismos de selecção e recolha poderosos. O grande número de bibliotecas e o facto de as operações de catalogação e classificação serem manuais tem dificultado a sua utilização.

Perspectiva de Técnica A perspectiva de técnica determina o mecanismo de reutilização utilizado. Reutilização por **composição** recorre a interfaces normalizadas, bibliotecas e colecções de componentes existentes como blocos para construir novos sistemas. O estudo está orientado para a compreensão, selecção, recolha, adaptação e integração dos componentes, essencialmente ao nível do código. A utilização de ambientes integra-

dos de reutilização tem-se mostrado a forma mais promissora de atingir os objectivos. O conceito de reutilização **generativa**, reutilização ao nível da especificação através da aplicação de geradores de código, apresenta elevado potencial. No entanto, esta técnica é difícil de sistematizar para uma utilização industrial, restringindo-se a um conjunto reduzido de áreas de aplicação. Apesar disso, exemplos como as ferramentas `lex` e `yacc` do **UNIX** [SJ85], ou o **Draco** [Nei84] demonstram o potencial desta aproximação.

Perspectiva de Intenção A perspectiva de intenção estabelece a granularidade da descrição da entidade necessária ao processo de reutilização. A reutilização de **caixas pretas** assume que os componentes podem ser reutilizados sem alterações. Normalmente os componentes estão definidos por interfaces e comportamentos bem definidos oferecendo melhores garantias de qualidade e fiabilidade. Um problema importante, quando se reutilizam caixas pretas, consiste em garantir que o componente se comportará sem falhas em todas as situações possíveis. Este facto tem orientado a investigação para os métodos formais que, apesar dos seus elevados custos, podem provar a correcção dos programas e eliminar a necessidade de testes exaustivos. Como a reutilização de programas sem alterações é bastantes rara, a forma mais frequente consiste em reutilizar **caixas brancas**. A reutilização de caixas brancas pressupõe a alteração do comportamento do componente. A maioria da reutilização, em especial a oportunista, baseia-se na adaptação de componentes antes da sua utilização. Embora a adaptação seja quase sempre feita ao nível do código fonte, a tendência na utilização de caixas brancas tem vindo a englobar outros níveis do processo de desenvolvimento, como a análise e o desenho.

Perspectiva de Produto A perspectiva do produto classifica a entidade que vai ser sujeita ao processo de reutilização. Quase todas as ferramentas, ambientes e métodos estão voltados para a reutilização de **código fonte**. É natural que, à medida que as restantes formas de reutilização forem ganhando aceitação, menor será a predominância desta forma de reutilização. Eventualmente, a geração de código será automática e o desenvolvimento será feito com representações de mais alto nível. A escrita de código fonte será tão rara e específica como é hoje a escrita de código máquina. A reutilização

ao nível do **desenho** apresenta boas perspectivas, mas ainda não atingiu a maturidade. Os desenhos podem ser reutilizados parcial ou indirectamente em métodos orientados para objectos. Quando a especificação fica disponível para ser reutilizada está, quase invariavelmente, ligada com as respectivas realizações ao nível do desenho e do código. A reutilização generativa procura tirar partido da reutilização a este nível, que quando conseguida poupará grande parte do ciclo de desenvolvimento. A aproximação orientada para **objectos** tem vindo a ganhar terreno, não só por as linguagens de programação oferecerem técnicas de reutilização mas, por oferecerem o mesmo paradigma e ferramentas de suporte desde a análise à codificação. Todos os produtos, com excepção do código objecto, são para ser lidos por humanos. Desta forma, a documentação existente sob as diversas formas **textuais** toma especial importância até que o processo de reutilização possa ser automatizado em larga escala. Os principais avanços residem na utilização de hipertexto e na integração da documentação com outros produtos. As **arquitecturas** correspondem à unidade de maiores dimensões que pode ser reutilizada. Nesta aproximação, os diversos domínios de aplicação são analisados na busca de desenhos genéricos que possam ser utilizados como padrões para integrar componentes reutilizáveis ou produzir geradores de código especializados.

1.2.3 Processo de reutilização

A reutilização de conceitos, componentes ou processos segue também ela um processo. Este processo pode ser decomposto em três subprocessos distintos: o processo de desenvolvimento de software para reutilização, o processo de produção de entidades para reutilização e o processo de reutilização dessas entidades.

O processo de produção de software para reutilização não deve ser confundido com o desenvolvimento de software para reutilização. O desenvolvimento pressupõe um ciclo de desenvolvimento de software segundo o modelo usual, como o da cascata, com cuidados acrescidos para que o âmbito de aplicação do resultado obtido possa ser aplicado a várias situações. O processo de produção limita-se a preparar software, que pode ou não ter sido desenvolvido com objectivos de reutilização, para que ele possa a vir a ser reutilizado noutros projectos de desenvolvimento.

No processo de desenvolvimento (representado esquematicamente na figura 1.1) existe uma primeira fase de geração do componente, seguida da definição da interface e do armazenamento do componente. Tal como no caso do desenvolvimento de software não especificamente vocacionado para reutilização, na **geração** de componentes para reutilização segue-se a mesma sequência de operações de análise, desenho, codificação, teste e manutenção. Cada uma destas operações carece de preocupações acrescidas no que diz respeito à abstracção, configuração, parametrização entre outros, por forma a obter uma maior capacidade de reutilização. A **interface** determina as operações disponíveis para manipular o componente, estando quase exclusivamente limitadas a colecções de componentes imprecisamente designadas por bibliotecas. Embora uma mesma interface possa estar ligada a diversas formas de **armazenamento**, esta flexibilidade é raramente utilizada. O armazenamento dos componentes mais frequente é em ficheiros sequenciais [Jaz95], embora tenha sofrido alguma evolução com o advento dos sistemas distribuídos [BGMT88, Obj95].

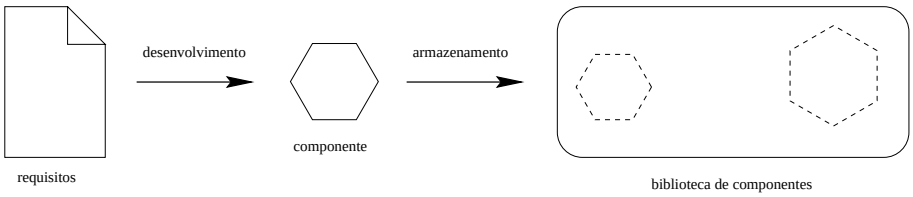


Figura 1.1: Processo de desenvolvimento de componentes para reutilização.

Na maioria dos casos os componentes não foram gerados com preocupações de reutilização, em geral, devido a limitações financeiras ou temporais. A falta de uma educação vocacionada para a reutilização pode ser, no entanto, determinante em situações em que as margens financeiras e temporais permitiam a sua aplicação, sendo ainda responsável pela ausência de um mercado para bibliotecas de componentes reutilizáveis. Assim, quer os componentes tenham sido pensados para ser frequentemente reutilizados, quer tenham sido desenhados para situações concretas são passíveis de ser reutilizados em determinado caso concreto. No processo de produção (representado esquematicamente na figura 1.2) pretende-se **identificar** as características relevantes de cada componente para reutilização. Estas características são depois atribuídas numa **representação** que permita a sua classificação. A **classificação** do componente, traduz o resultado do

guagem de interligação. Esta linguagem pode ser a mesma utilizada para descrever o componente, quer este esteja codificado ou em fase de análise ou desenho, ou recorrer a uma linguagem especialmente desenhada para o efeito.

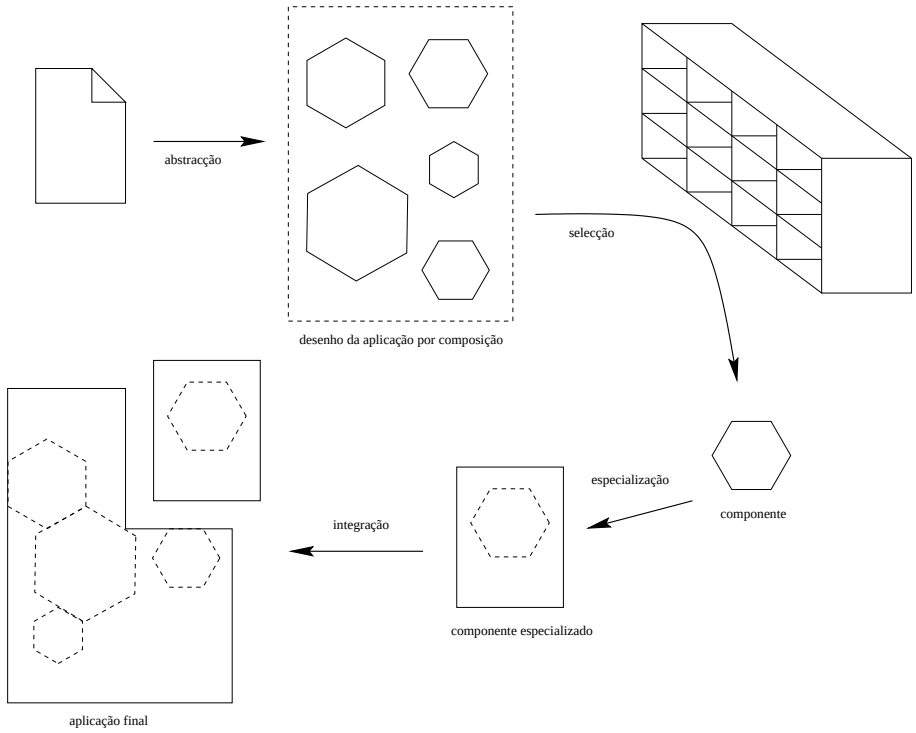


Figura 1.3: Processo de reutilização de componentes de software.

Finalmente, é conveniente referir que a utilização de um componente numa aplicação, por exemplo, uma rotina, é considerado um caso de reutilização. No entanto, as diversas invocações da mesma rotina durante a execução do programa, as diversas execuções do mesmo programa, ou a duplicação para distribuição sem alterações, não são considerados casos de reutilização.

1.2.4 Plano de reutilização

Quando se pretende definir um plano de reutilização é necessário efectuar escolhas organizacionais e tecnológicas com diferentes custos e benefícios [Lim94, BB91]. Pretende-se, nesta secção, salientar algumas das opções mais importantes existentes na literatura, bem como algumas medidas que podem influenciar directamente a escolha das opções. À adopção de um processo de reutilização deverá seguir-se um plano semelhante ao

que foi proposto por Davis [Dav93]. Numa fase inicial devem ser definidos os objectivos organizacionais, como a produtividade ou a qualidade, e analisadas as oportunidades de reutilização. Um determinado projecto, ou organização nele envolvido, pode estar activo em diversos domínios de aplicação, devendo o potencial de reutilização de cada domínio ser avaliado. Enquanto certos domínios que exibem grande estabilidade e já foram extensivamente estudados oferecem índices de reutilização potencial elevados, outros existem onde o potencial de reutilização é baixo [LG84]. Contudo, não existe uma forma simples que permita avaliar qual a reutilização potencial dentro de um determinado domínio de aplicação sem o fazer repetidamente durante alguns anos e analisar os resultados obtidos.

Com base na experiência obtida, quer directamente pela experimentação, quer indirectamente através dos resultados de terceiros, é necessário definir os domínios e subdomínios que oferecem maiores índices de reutilização potencial e menores riscos técnicos ou financeiros. Uma vez definidos os domínios, deve-se definir os objectivos de reutilização que é possível atingir dadas as limitações organizacionais técnicas e financeiras. Para tal o modelo de capacidade de reutilização ("*reuse capability model*") [Dav93] define os objectivos de reutilização em função de três medidas:

Competência A competência mede a taxa de oportunidades de reutilização exploradas face às oportunidades potenciais.

Eficiência A eficiência mede a taxa de oportunidades de reutilização exploradas face às oportunidades definidas nos objectivos da organização.

Eficácia A eficácia mede a taxa de benefícios com a reutilização face aos custos.

Note-se que todas as três medidas assumem a existência de um processo de reutilização. Contudo, estas medidas não são métricas a calcular na conclusão do projecto, mas objectivos a atingir ao iniciar o processo de reutilização. Sendo difícil determinar o potencial de reutilização de um componente num determinado domínio de aplicação, a eficácia de reutilização do componente representa uma estimativa. A eficiência de reutilização tem merecido maior interesse na literatura, embora sob a forma de percentagem de código reutilizado em novos projectos [FP94].

Gestão de componentes

A equipa de gestão dos componentes reutilizáveis é normalmente responsável pelo empacotamento dos componentes. Este empacotamento compreende a individualização dos componentes, seu armazenamento, documentação e controlo de qualidade [PDF87]. Os componentes podem ter sido especialmente produzidos para serem reutilizados, caso em que estão já individualizados; podem ser aproveitados a partir de outras aplicações, caso em que têm de sofrer algum tratamento antes de poderem ser armazenados. Este tratamento pode ser o simples desacopulamento da aplicação original, ou pode sofrer uma adaptação que melhore a sua gama de aplicações e capacidade de parametrização.

Com recurso à reutilização, a equipa de desenvolvimento pode agora ser vista segundo duas perspectivas diferentes. Por um lado a equipa pode desenvolver componentes para reutilizar, enquanto por outro pode desenvolver software com recurso a componentes reutilizáveis. Daqui podemos extrair dois modelos organizacionais [BB91]:

- **modelo produtor-consumidor puro,** em que a equipa de desenvolvimento para reutilização é totalmente distinta da equipa de desenvolvimento com reutilização. Neste caso, a equipa de desenvolvimento para reutilização será responsável pela gestão dos componentes, seu armazenamento, documentação e controlo de qualidade, pelo que coincide com a equipa de gestão dos componentes. A equipa de desenvolvimento reduz-se, consequentemente, ao desenvolvimento com componentes reutilizáveis.

- **modelo misto,** em que as equipas de desenvolvimento produzem e consomem componentes reutilizáveis. Neste caso, a equipa de gestão de componentes é distinta, podendo as equipas de desenvolvimento estar, ou não, divididas em desenvolvimento para e com reutilização.

As actividades das equipas de desenvolvimento podem ser analisadas de um ponto de vista produtivo como fábricas de software [Cus89, CB91]. Neste ponto de vista pretende-se focar não no produto – o componente – mas no processo, identificando-se dois tipos de actividades:

- **actividades síncronas** que resultam de pedidos específicos da equipa de desenvolvimento com reutilização. Estes pedidos são satisfeitos com componentes já existentes na biblioteca, ou, na sua falta, com o seu desenvolvimento de origem. As actividades síncronas são prioritárias pois dependem das restrições temporais dos projectos.

- **actividades assíncronas,** consistem na produção de componentes que podem vir a ser necessários, antecipando futuros pedidos. Estes componentes podem ser produzidos de origem ou adaptados.

Finalmente, do ponto de vista da reutilização, as tarefas de manutenção são equivalentes às tarefas de desenvolvimento, pelo que a equipa de manutenção é entendida como uma equipa de desenvolvimento. A manutenção é vista como o desenvolvimento de software em que grande parte do software existente em versões anteriores é reutilizado [KBM97, Bas90, Rin91]. Assim, na maioria dos casos, a manutenção será um caso particular de desenvolvimento com reutilização em que a quantidade de software reutilizado é, à partida, grande. O processo de selecção deste software pode ter características diferentes, uma vez que as versões anteriores são, geralmente, preferidas em detrimento de novas aproximações. Note-se, contudo, que as razões desta preferência prendem-se mais com motivos humanos, como a confiança e resistência à mudança, que com motivos técnicos, como os menores custos de especialização e integração [WES87, Faf94, Rin91].

1.3 Componentes de software

Os componentes de software [Cox90] são um conceito que permite resolver alguns problemas da crise do software. Comparado com outras engenharias onde a utilização de componentes tem sido bem sucedida, a tecnologia dos componentes de software ainda está em formação. Os aspectos técnicos e em particular a interoperabilidade constituem a maioria da pesquisa nesta área. Os aspectos sociais e organizacionais só adquirem expressão nos relatórios de grandes projectos de software executados da indústria. Muitas

das dificuldades verificadas no desenvolvimento de software resultam de uma perspectiva tecnocêntrica [Mur97]. De um ponto de vista técnico os componentes são fundamentalmente semelhantes aos objectos, quer nos objectivos quer nas soluções encontradas. A natureza específica dos componentes surge apenas no contexto da produção de software a um nível não exclusivamente técnico. Embora as alterações ao desenvolvimento de software sejam desencadeadas por inovações técnicas, o mercado dos componentes de software não se limita a problemas técnicos.

O facto de não existir uma definição consensual para "componente de software" é indicativo da falta de maturidade da tecnologia. Não existe acordo em relação ao que um componente é, excepto que os componentes são para efectuar composições. Uma definição proposta [Mur97] foca os aspectos técnicos e organizacionais: "*Um componente é uma unidade de composição com uma interface contratualmente especificada e apenas com dependências explícitas de contexto. Um componente pode ser instalado isoladamente e ser sujeito a composição por terceiros.*" Desta forma, componentes são produtos que são distribuídos com o objectivo de serem compostos por terceiros. Assim, os componentes devem ser construídos tão independentes de determinado contexto quanto possível, por forma a permitir a sua especialização e integração. Quanto mais um componente é independente de determinado contexto mais facilmente ele pode ser reutilizado em outros contextos. No entanto, um componente totalmente independente de qualquer contexto não pode ser reutilizado, uma vez que necessita de um contexto para poder ser integrado. Esta contradição constitui um dos maiores desafios no desenho de componentes reutilizáveis.

O êxito na reutilização de componentes depende de melhores técnicas de composição. Se pensarmos em composição num contexto mais geral, como poderão os utilizadores gerir a informação sobre cada componente quando existem quantidades cada vez maiores de componentes disponíveis. A forma mais frequente de descrever informação de composição é através da interface do componente e alguma informação adicional, normalmente sob uma forma informal. Esta informação não é suficientemente detalhada para permitir resolver os problemas de interoperabilidade. Descrições semânticas restringem-se, em geral, a alguns domínios de aplicação. Torna-se necessário introduzir níveis intermédios entre descrições sintácticas de interfaces e descrições semânticas

completas. Uma destas aproximações distingue os vários aspectos de cada componente tratando cada um separadamente [KLM+97]. Uma solução pragmática consiste em documentar o componente com a informação necessária à sua integração durante o processo de desenvolvimento do componente. Esta aproximação procura motivar a produção de informação contextual pela entidade encarregue do desenvolvimento do produto.

1.3.1 Caracterização de componentes

A construção de sistemas informáticos é feita cada vez mais à custa de componentes já existentes. Desta forma torna-se cada vez mais importante dispor de uma correcta caracterização de cada componente. Sem essa caracterização muito do esforço será desperdiçado na compreensão e adaptação dos componentes, cada vez que são utilizados. Além disso, a caracterização de componentes é essencial à gestão de bibliotecas de componentes, que é uma exigência da engenharia de software baseada na reutilização sistemática de componentes. Os sistemas baseados em componentes começam agora a surgir, não havendo aproximações sistemáticas com provas dadas que seja possível adoptar. A caracterização de componentes é um passo no sentido de facilitar o aparecimento de tais aproximações. A caracterização de componentes pode ser vista como uma extensão à caracterização de objectos, oferecendo uma base para o desenvolvimento, gestão e utilização de componentes [Han98].

Os elementos estruturais são partes essenciais dos componentes pois são eles que dão corpo ao componente e, seguindo a convenção dos objectos, são chamados atributos. Da mesma forma, as operações captam o comportamento dinâmico do componente, representando o serviço/funcionalidade que este oferece aos outros componentes que com ele interagem. Tal como no caso dos objectos, os atributos e as operações constituem os aspectos fundamentais de um componente.

A composição de componentes está sujeita a um conjunto de restrições. As restrições estruturais, onde certas composições de atributos não são permitidas, e as restrições operacionais, onde nem todas as sequências de invocação de componente são possíveis. Os padrões estruturais e operacionais permitem definir as composições aceitáveis para

determinado componente. O diagrama de transição de estados representa, no caso dos objectos, as sequências de invocação possíveis para cada objecto.

Além das invocações explícitas, controlo pró-activo, é ainda necessário registar, caso exista, o comportamento do sistema ao controlo reactivo, sob a forma de invocações com origem em acontecimentos. Enquanto alguns aspectos do comportamento do sistema são mais bem captados pelo controlo pró-activo, outros há que são mais bem representados pelo controlo reactivo. No controlo reactivo são gerados acontecimentos, periódicos ou não, aos quais os componentes do sistema podem optar por reagir [Ham97].

Um componente pode interagir com um certo número de outros componentes dependendo do papel, ou perspectiva, que ele representa. As interacções entre os componentes em causa podem variar de acordo com as perspectivas de cada um. Por exemplo, ao interagir com um certo tipo de componentes a partir de uma determinada perspectiva, apenas algumas operações e algumas restrições operacionais podem ser aplicadas. Este facto sugere a necessidade de definir para cada componente protocolos de interacção, ou interfaces, por cada perspectiva. Os cenários oferecem descrições de casos em que os componentes são vistos de diferentes perspectivas, o que facilita a especialização e integração do componente. O ponto de vista do componente é mais atraente nas operações de busca e selecção, não havendo consenso quanto à solução a adoptar.

Um outro aspecto de um componente diz respeito às suas propriedades não funcionais, tais como a segurança, legibilidade, desempenho e fiabilidade. No contexto da construção de sistemas a partir de componentes existentes, o impacto no resultado final é particularmente importante. Algumas propriedades não-funcionais não podem ser quantificadas, como a legibilidade.

Resumindo, os componentes são caracterizados por:

atributos
operações
restrições estruturais e operacionais
acontecimentos
multi-interfaces por cenário
propriedades não-funcionais

Contudo, é ainda difícil dispor de uma caracterização normalizada para os componentes, uma vez que mesmo os aspectos com origem na tecnologia dos objectos não estão totalmente estáveis no seu próprio domínio.

1.3.2 Catálogos de componentes

O desenvolvimento de aplicações informáticas baseado em componentes requer a existência de catálogos de componentes de software. Esta foi a visão apresentada em 1968 por McIlroy na conferência da NATO [Ran96]. No entanto, essa visão ainda não foi concretizada. Para permitir construir software essencialmente baseado em componentes com origem em catálogos é necessário que, quer os componentes quer os catálogos, obedeçam a um conjunto de requisitos fundamentais [Jaz95]:

Os componentes do catálogo devem formar uma taxinomia sistemática A taxinomia sistemática permite guiar o desenhador da aplicação na busca e selecção dos componentes do catálogo. O desenvolvimento de um conjunto de catálogos, cada um suportando uma funcionalidade bem definida, é essencial para a consolidação de um desenvolvimento de software baseado na reutilização de componentes. O sucesso de muitas linguagens de programação tem sido devido ao aparecimento de extensas bibliotecas de componentes normalizados para a linguagem. Por exemplo, **Smalltalk** [GR89] teve grande sucesso ao surgir com uma biblioteca de cerca de 40 classes. Mais recentemente, o **Java** [AG98] aumentou este número para mais de 120, cobrindo novas áreas de aplicação.

Os componentes devem ser genéricos A principal motivação para a reutilização de software é a redução do esforço necessário para o seu desenvolvimento. Quanto mais vezes um componente for reutilizado, menor é o esforço total de desenvolvimento. Por outro lado, quanto mais genéricos forem os componentes, menos componentes são necessários para cobrir um dado domínio. Tal facto facilita a sua posterior selecção e aumenta a possibilidade de ser reutilizado.

Os componentes devem ser eficientes As bibliotecas devem procurar ter realizações razoavelmente eficientes. Embora possa ser possível realizar componentes mais eficientes tal deve exigir um esforço não compensatório numa primeira aproximação. A eficiência é uma necessidade real. A eficiência de um produto pode ser essencial para o seu sucesso, por exemplo, uma base de dados. Para tal, basta ter presente as ordens dos algoritmos de ordenação ou de busca. Ignorar os requisitos de eficiência de uma forma arbitrária pode obrigar a redesenhar grande número de componentes.

Os catálogos devem ser abrangentes No primeiro requisito afirmava-se que os componentes do catálogo devem formar uma taxinomia sistemática num dado domínio. Para que o uso de bibliotecas de componentes seja difundido e vulgarizado é necessário que cada catálogo seja abrangente, ou seja, deve cobrir uma parte significativa da taxinomia do domínio. Se o catálogo for abrangente evita-se o síndroma da produção doméstica, em que o programador opta por desenvolver software de raiz, apenas por desconhecer que existe algo directamente disponível. Note-se que, embora este síndroma seja menos significativo nas grandes empresas de desenvolvimento de software, torna-se significativo em empresas de menores dimensões [FF95]. O catálogo ao ser abrangente oferece garantias mínimas de conter pelo menos um componente que obedece minimamente aos requisitos.

Os componentes devem residir em repositórios integrados A existência de um repositório comum, com interfaces e formato de dados normalizados, permite eliminar dependências de hardware, sistema operativo, bases de dados ou interfaces gráficas. Isto deve-se ao facto de o repositório escolhido ter implícito essa dependência, reduzindo o espaço físico do problema. Do ponto de vista dos componentes, um repositório de componentes define um modelo para documentar, guardar e distribuir componentes de uma forma normalizada. Os componentes podem ter associada informação que os permita classificar, e posteriormente seleccionar, tornando o processo de reutilização mais integrado e completo.

1.4 Motivação

A produtividade no desenvolvimento de software tem aumentado de forma sustentada nos últimos 30 anos. No entanto, apesar do aumento do número de analistas e programadores, não tem sido possível cobrir a procura quer para o desenvolvimento de novos sistemas quer para manutenção do software existente. Assim, para além de procurar formas de desenvolver software mais depressa devemos procurar formas de escrever menos software [Rin91, MMM95]. A programação automática, onde um sistema informático é capaz de produzir código executável a partir de requisitos informais e incompletos, não é possível devido à ambiguidade e à necessidade de intervenção criativa humana. Mesmo quando existe uma análise formal a passagem pode ser simplificada e menos sujeita a erros, mas o objectivo da automatização ainda não foi conseguido [Hal90, Kem90]. Desta forma, a reutilização de software parece ser, de momento, a única solução técnica viável [FI94, PD91b].

A reutilização permite obter ganhos consideráveis pois os componentes de software reutilizados não tiveram de ser desenvolvidos. Além disso, se for possível obter componentes de alta qualidade – eficientes, parametrizáveis e testados – também a qualidade do produto final fica melhorada [Gra93]. O elevado número de aplicações informáticas em cada área de actividade, bem como a sobreposição de sub-áreas existente entre as diversas actividades, abre boas perspectivas para uma reutilização em larga escala. Em algumas situações já foi possível atingir elevados níveis de reutilização com aumentos na produtividade e qualidade. No entanto, tais sucessos não são frequentes nem sistemáticos pelo que existe a necessidade de institucionalizar a prática de desenvolver software com base em reutilização de componentes [FF95, CBG+96, Weg84, BB91]. Tal prática obriga a ultrapassar problemas educacionais, organizacionais e técnicos [Pri97, Faf94, WES87, MMM95, Sta94, Kru92]. Um estudo, condensado na figura 1.4, releva as diversas razões apresentadas para a não reutilização de componentes na contrução de novas aplicações [FF96].

Muito trabalho tem sido desenvolvido no sentido de automatizar a reutilização de software [Kru92]. Embora importantes progressos tenham sido conseguidos, o domínio de aplicação de cada método de reutilização é reduzido. Por exemplo, a classificação de

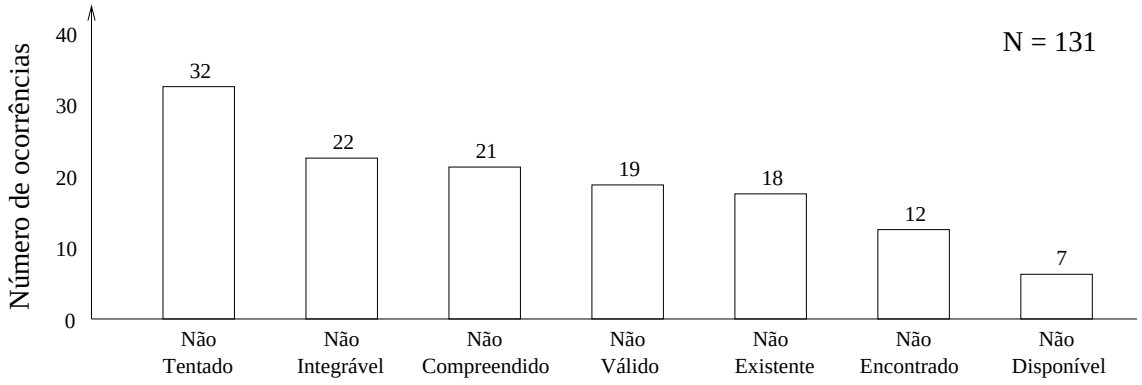


Figura 1.4: Razões na origem da não reutilização de componentes.

componentes com base em comportamento só é possível para componentes muito pequenos, normalmente funções simples [PP93]. À medida que a complexidade dos componentes aumenta torna-se mais difícil analisar o componente com base no seu comportamento [PDF87, Jr.95]. Ao nível da selecção de componentes, o problema consiste em obter os componentes que melhor respondem aos requisitos. No entanto, no mundo real não há igualdades perfeitas. Mesmo que exista um componente que corresponda exactamente ao pretendido, a imprecisão na formulação da pergunta e a sua interpretação conduzem apenas a proximidades [MMM95, MPMM98, ZW95, ZW97]. O que se obtém não é a comparação entre o componente e o requisito, mas a comparação da pergunta formulada face ao método de pesquisa e a classificação do componente [MMM95].

Também ao nível da especialização e integração automática de componentes as soluções são limitadas. As técnicas automáticas restringem-se a transformações simples, tais como a alteração da ordem dos argumentos, conversões entre tipos de dados abstractos base e parametrização com valores constantes. Quando existe mais de um componente à escolha as alterações a efectuar podem ter características diferentes. Por exemplo, num dado caso pode ser necessário um número reduzido de alterações, mas exigindo cada uma um elevado esforço intelectual. Noutro caso, as alterações podem ser muitas, mas todas elas triviais. O custo associado a cada uma das escolhas depende da capacidade e experiência da equipa envolvida, bem como das ferramentas disponíveis. É, portanto, necessário deixar a escolha sujeita a critérios subjectivos. Para tal deve ser disponibilizada toda a informação disponível, por forma a que possa ser efectuada uma escolha mais esclarecida. Contudo, esta informação deve ser extratificada por forma a

permitir o seu manuseamento em diferentes níveis de complexidade [Kru92].

1.5 Objectivos

A reutilização de software tem vários catalizadores e vários inibidores, que têm sido frequentemente discutidos na literatura da especialidade. Algumas considerações antagónicas têm sido feitas, sugerindo que a reutilização de sotfware é apenas um problema social e organizacional, enquanto outras têm reclamado ser um problema puramente técnico. Recentemente, certas figuras proeminentes na área têm afirmado tratar-se a institucionalização da reutilização de software de um problema híbrido onde se combinam as soluções técnicas e as capacidades humanas.

Para tal é necessário abordar as diversas fases do processo de reutilização de software. Este processo deve ser entendido como a construção de software com base em componentes já existentes. O problema do desenvolvimento de software para ser reutilizado é essencialmente um processo de desenvolvimento de software clássico, onde os cuidados de extensibilidade e generalidade são particularmente importantes. Desta forma, o desenvolvimento para reutilização não necessita ser considerado em simultâneo com a reutilização de software. Estes objectivos implicam que o processo de reutilização deve ser genérico, ou seja, ser capaz de reutilizar componentes que não tenham sido especialmente desenhado para efeitos de reutilização.

Numa primeira fase, os componentes candidatos a serem reutilizados devem ser reunidos e registados. Cada componente deve ser analisado e classificado de acordo com a sua funcionalidade, desempenho, interface e outros parâmetros relevantes para a sua posterior selecção. A automatização do processo de classificação permite evitar os erros de interpretação que ocorrem quando a classificação é realizada por humanos. Um processo automático permite considerar apenas a informação efectivamente disponível sobre o componente. No entanto, ao adoptar um processo de classificação automático é necessário aceitar que alguma informação semântica ou não funcional pode não estar presente, conduzindo a imprecisões, omissões ou ambiguidades. Assim, torna-se

necessário permitir a intervenção humana para identificar e corrigir estes problemas, concluindo desta forma um processo de classificação semi-automático.

A fase de selecção dos componentes candidatos a serem reutilizados em determinada situação necessita ser um processo iterativo e interactivo. Iterativo pois uma primeira selecção pode identificar vários componentes, exigindo um sucessivo refinamento, ou não encontrar nenhum candidato, como consequência de uma selecção baseada em parâmetros excessivamente restritivos e rigorosos. Interactivo pois é necessária a intervenção humana para redirigir a selecção para os requerimentos originalmente expressos. O processo de selecção pode ser automatizado e parametrizado com base num conjunto de algoritmos alternativos de busca.

O processo de especialização consiste na adaptação do componente seleccionado aos requisitos. Embora seja uma fase essencialmente dependente da intervenção humana, pela criatividade necessária, a utilização de uma linguagem de prototipagem é essencial. Uma linguagem deste tipo, mesmo que não seja a mesma a utilizar na fase de integração, permite avaliar mais rigorosamente os custos de especialização. O resultado desta fase pode inclusivamente conduzir a um novo processo de selecção, e consequente selecção de outro componente, com base na experiência adquirida. Esta experiência permitirá um refinamento do processo de selecção pela inclusão de parâmetros omissos nos requerimentos originais.

O processo de especialização está intimamente ligado com a integração dos componentes na aplicação final. O ambiente e a linguagem alvo são determinantes nos custos de especialização e, consequentemente, no processo de selecção. Se a linguagem do componente, seja esta de análise, desenho ou codificação, for distinta da linguagem e ambiente da aplicação a construir, os custos inerentes à sua integração podem orientar a escolha para outro componente.

Pelos objectivos atrás expostos conclui-se que todo o processo de reutilização necessita frequentemente da intervenção humana, embora existam tarefas que podem ser mecanizadas com o auxílio de ferramentas apropriadas. Para mais, devido à grande união existente entre as diversas fases do processo de reutilização, era útil dispor de um ambiente coerente que cubra todo o processo. Além disso, a escolha das entidades fundamentais

do ambiente deve permitir uma simples e intuitiva assimilação da sua funcionalidade por parte dos utilizadores. Finalmente, é necessária uma linguagem que permita manipular o conjunto de uma forma flexível por forma a adaptar-se facilmente às opções dos utilizadores.

1.6 Contribuições

A reutilização de software pode ser melhorada desde que para isso se tenha em conta a harmonização dos factores humanos com os factores técnicos. Com este objectivo, nesta dissertação identificam-se três componentes a desenvolver: uma metodologia de reutilização, um ambiente de reutilização e um modelo unificador. O modelo unificador oferece um esquema conceptual sobre o qual é construído o ambiente e a metodologia. A metodologia por sua vez recorre aos mecanismos oferecidos pelo ambiente para facilitar o desenvolvimento de software com base em componentes reutilizáveis.

A **metodologia de reutilização** foca os aspectos humanos, em especial a forma de ajudar o utilizador a reutilizar. Para tal é necessário ter em conta a organização do grupo de trabalho, e o método de trabalho de cada indivíduo. No entanto, não se pretende impor um processo racional de desenvolvimento, mas oferecer uma variedade de procedimentos de reutilização que permitam, por composição, adaptar-se às necessidades e ao comportamento quer do grupo quer do indivíduo.

O **ambiente de reutilização** é constituído por um conjunto de mecanismos de classificação, selecção, especialização e integração. Estes mecanismos são controlados por uma linguagem que permite sequenciá-los e combiná-los de forma a atingir os objectivos pretendidos.

O **modelo unificador** oferece uma só abstracção com base na organização hierárquica de identificadores. Este modelo permite classificar hierarquicamente e seleccionar os componentes com base em identificadores extraídos dos próprios componentes. A linguagem de especialização e integração de componentes tem por base o mesmo modelo, usando a hierarquia de identificadores para representar os tipos de dados abstractos

utilizados na classificação e selecção, bem como outras entidades utilizadas durante o processamento.

O sistema proposto promove a reutilização através de um ambiente e de uma metodologia flexíveis, facilitando a sua utilização a grupos com experiência e métodos de trabalho diferentes. Assim, embora grupos de desenvolvimento diferentes possam tomar opções distintas, estas serão as que melhor se adaptam à sua forma de ver o problema e procurar a sua solução. Tal não implica que uma solução seja melhor que a outra, quer de um ponto de vista de qualidade, custo ou esforço, embora cada grupo considere a sua solução melhor e mais adaptada à sua perspectiva.

O mecanismo de classificação proposto nesta dissertação é mais rico que os sistemas actualmente existentes, permitindo um processo de selecção mais preciso. O espaço necessário para armazenar a informação de classificação não é significativamente maior. O facto de existir muito mais informação disponível não torna a selecção muito lenta, em especial quando comparado com métodos formais.

A linguagem de integração proposta nesta dissertação, designada por **MAP**, segue uma perspectiva de carregamento dinâmico que facilita a inclusão e substituição de componentes na aplicação. A estrutura de dados proposta permite aos elementos da aplicação serem enumerados e analisados, tornando a aplicação auto-explicativa. Este facto permite aos componentes e ao utilizador disporem de uma descrição da aplicação durante a execução, facilitando a interacção e a evolução dinâmica.

1.7 Organização da dissertação

Nesta introdução apresentou-se uma primeira abordagem à reutilização de software e a motivação que desencadeou este trabalho, bem como os objectivos a atingir.

Esta dissertação encontra-se dividida em três partes: enquadramento, arquitectura e conclusões. Cada uma das partes foca uma aproximação distinta ao problema da reutilização de software - trabalho existente, trabalho desenvolvido e conclusão sobre os

resultados obtidos, respectivamente - e está organizada segundo a sequência do processo de reutilização de software: abstracção, classificação, selecção, especialização e integração.

Na parte do enquadramento, formada pelo capítulo 2, pretende-se introduzir os conceitos e a evolução da reutilização de software nas áreas mais próximas do trabalho desenvolvido. Neste capítulo, analisam-se as várias aproximações existentes para resolver os problemas de classificação e selecção de componentes. Aborda-se a especialização de componentes tendo em vista a construção de aplicações por interligação de componentes. Para tal, analisam-se várias técnicas de especialização e de integração, bem como as linguagens e métodos que dão forma a essas técnicas.

Na parte da arquitectura, formada pelos capítulos 3 a 6, define-se a metodologia seguida e descrevem-se as ferramentas de apoio desenvolvidas. No capítulo 3, explicita-se o modelo de identificadores hierárquicos escolhido e justificam-se as principais opções tomadas. No capítulo 4, definem-se os mecanismos necessários para a classificação dos componentes com base no modelo definido. No capítulo 5, explora-se o processo de selecção disponibilizado pelo modelo para a escolha dos componentes que melhor se adaptam às características desejadas para a aplicação a construir. Seguidamente, no capítulo 6, define-se a linguagem de integração utilizada, bem como a sua articulação com o modelo proposto e as suas características mais relevantes.

Na última parte, formada pelos capítulos 7 e 8, oferece-se uma visão panorâmica do ambiente de reutilização realizado e apresentam-se as conclusões obtidas. No capítulo 7, aplica-se o processo de reutilização de uma forma integrada, explorando as opções e aplicando os métodos e algoritmos identificados na arquitectura. No capítulo 8, efectua-se uma avaliação crítica do ambiente de reutilização anteriormente descrito. Finalmente, perspectiva-se a contribuição proposta e consideram-se melhoramentos e possíveis evoluções futuras.

Como nota final, a síntese de cada capítulo é apresentada no final dos mesmos.

Capítulo 2

Técnicas de reutilização

A reutilização de software está presente numa grande variedade de tecnologias de produção de software. Existem, contudo, características comuns entre as diversas técnicas utilizadas. Por exemplo, bibliotecas de componentes, geradores de aplicações e compiladores de linguagens, todos seguem um conjunto de passos comuns que podem ser condensados em abstracção, selecção, especialização e integração de componentes de software [BR87].

Abstracção Todas as aproximações de reutilização de software utilizam alguma forma de abstracção para os componentes de software. A abstracção é uma representação incompleta, ou vista parcial, de determinada entidade. A necessidade de escolher os componentes, com base num conjunto de características directamente relacionadas com os requisitos, torna a abstracção uma técnica essencial na reutilização.

Selecção A maioria das soluções de reutilização oferecem aos seus utilizadores mecanismos para localizar, comparar e seleccionar componentes de bibliotecas de software. As técnicas de selecção estão directamente ligadas às técnicas de catalogação e classificação de componentes de software, que são utilizadas para organizar a biblioteca de componentes e para guiar o utilizador na sua busca.

Especialização Em muitas das técnicas de reutilização, componentes semelhantes são fundidos num só componente mais genérico. Após a selecção desse componente para reutilização, o desenhador de software especializa-o através de parametrizações, transformações, restrições ou outra forma de aperfeiçoamento. As formas mais simples de especialização consistem na fixação de valores, por exemplo, máximos ou mínimos, restringindo a generalidade original do componente ao caso concreto onde se irá enquadrar.

Integração As tecnologias de reutilização recorrem, frequentemente, a ambientes de integração. O desenhador de software utiliza estes ambientes para combinar o conjunto de componentes seleccionados, e seguidamente especializados, num sistema de software completo. As linguagens de integração de módulos são exemplos de ambientes de integração, onde as entidades importadas e exportadas pelos componentes são combinadas através da linguagem para produzir o sistema final.

2.1 Identificação de abstracções

A abstracção e a reutilização de software estão fortemente relacionadas [Boo94, Nei84, Sta84], pois a abstracção descreve um conjunto de componentes através de elementos comuns ou relacionados. Da mesma forma, um conjunto de componentes reutilizáveis determina uma abstracção que descreve algumas das características exibidas por cada componente. Devido à importância da abstracção na reutilização de software, esta será tratada separadamente e funcionará como tema unificador desta dissertação.

2.1.1 Abstracção no desenvolvimento de software

As abstracções têm sido frequentemente utilizadas nas ciências da computação para ajudar a compreender a complexidade do software [Sha84].

Definição 2.1 (abstracção) *Uma* abstracção *de um componente de software é uma descrição sucinta que permite suprimir os pormenores que não são importantes para a sua reutilização, concentrando a atenção no que é relevante.*

A captação das características consideradas relevantes é efectuada através da ordenação, agregação ou generalização dessas características. O software é, normalmente, constituído por diversos níveis de abstracção construídos a partir do hardware de suporte.

Exemplo 2.1 *No nível mais baixo encontramos o código máquina que pode ser directamente executado pelo hardware. A linguagem assembly constitui o nível seguinte e, embora dependa do hardware em questão, está descrita textualmente e pode ser facilmente modificada. As linguagens de programação aumentam a abstracção pois deixam de estar dependentes de um hardware específico, podendo ser transportadas para um conjunto de sistemas. Em linguagens com suporte para módulos e compilação separada a programação tem por base as assinaturas - declarações - desses módulos, sem necessidade de conhecer o conteúdo dos mesmos, que podem ter sido desenvolvidos recorrendo a outras linguagens.*

Definição 2.2 (especificação e realização da abstracção) *Cada abstracção de software apresenta duas perspectivas, a* especificação da abstracção *e a* realização da abstracção*. A realização da abstracção concretiza parte da especificação pelo que representa uma especialização da abstracção. Inversamente, a especificação é uma generalização de uma ou mais realizações da mesma abstracção.*

No entanto, a realização de uma abstracção concretiza certas características mas especifica outras que constituem uma nova abstracção (ver figura 2.1).

Definição 2.3 (constituição da abstracção) *Uma abstracção é constituída por três partes: privada, parametrizável e imutável. A* parte privada *é constituída pelos pormenores da realização da abstracção, que são suprimidos na especificação da mesma. A parte visível da especificação pode ser dividida numa parte imutável e numa parte parametrizável. A* parte parametrizável *da abstracção contém as características parametrizáveis e configuráveis da realização, enquanto a* parte imutável *contém as invariantes.*

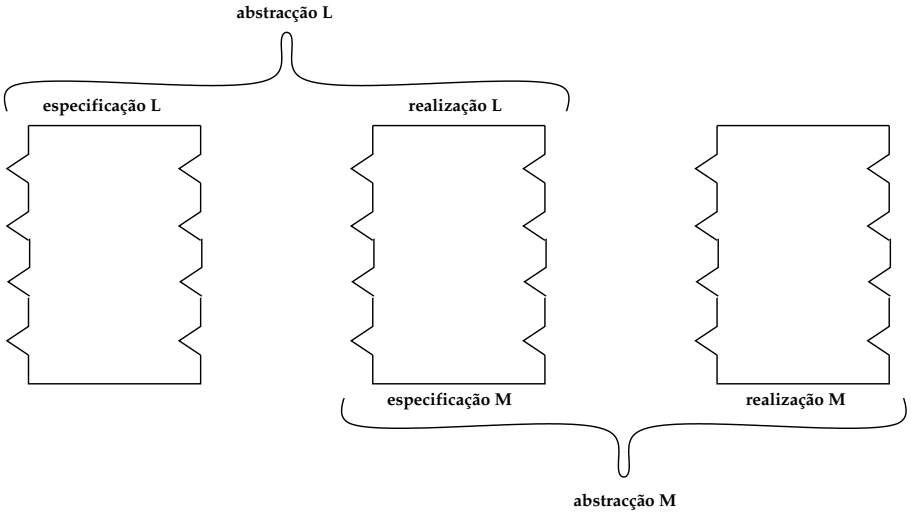


Figura 2.1: Especificação e realização de abstracções.

Desta forma, a especificação de uma abstracção com uma parte parametrizável corresponde a um conjunto de realizações alternativas (ver figura 2.2).

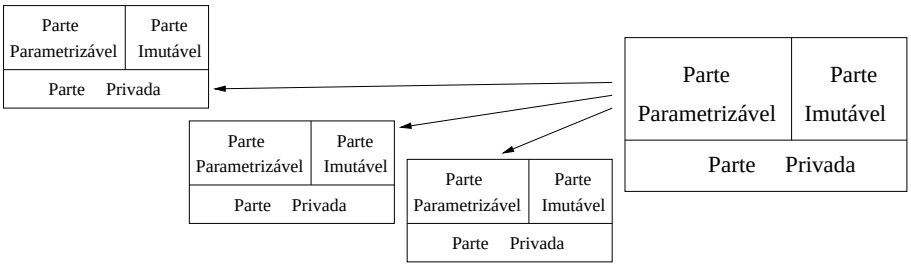


Figura 2.2: Realizações alternativas da especificação de abstracções.

Exemplo 2.2 *Na abstracção de uma pilha de dados a parte imutável exprime as características invariantes de todas as pilhas de dados, como a semântica das operações de construção, entrada e saída de elementos. O comportamento invariante da pilha de dados não depende do tipo de elementos armazenados, de opções de desempenho ou gestão de memória. A partição em zonas imutáveis, parametrizáveis e privadas não é intrínseca à abstracção mas resultado de decisões arbitrárias tomadas por quem identificou a abstracção. Na construção da abstracção é necessário decidir qual a informação que será útil aos utilizadores e que deverá, consequentemente, ser colocada na especificação da abstracção. De igual forma será necessário decidir quais as partes que podem variar, como, por exemplo, a profundidade da pilha de dados ser fixa, configurável ou dinâmica. Se a profundidade for colocada na parte imutável, o utilizador sabe qual o valor pré-definido e que não lhe é permitido alterar.*

2.1.2 Abstracção na reutilização

A abstracção desempenha um papel central, e por vezes limitativo, nos restantes passos do processo de reutilização de componentes de software. Os componentes ao serem seleccionados devem apresentar abstracções claras e concisas para que a sua classificação e posterior localização, comparação e recuperação sejam bem sucedidas.

Definição 2.4 (componente genérico) *Um* componente genérico *é um componente em que a parte parametrizável da abstracção não é vazia.*

A especialização corresponde, frequentemente, à escolha da realização da abstracção a partir da parte parametrizável da especificação da abstracção. Para integrar um componente reutilizável num sistema de software, o utilizador deve compreender totalmente a interface do componente, isto é, as propriedades dos componentes que interagem com outros componentes ou com o ambiente de integração.

Análise de variantes

A parte parametrizável de um componente está directamente ligada à generalidade do componente e, consequentemente, ao número de situações potenciais em que pode vir a ser reutilizado. A análise de variantes desempenha na reutilização de software um papel semelhante ao da análise de requisitos no desenvolvimento tradicional. O seu objectivo consiste em quantificar, à partida, o esforço de especialização necessário para a reutilização, da mesma forma que a análise de requisitos tenta quantificar o esforço de desenvolvimento estimado para cumprir os requisitos funcionais. Na sua forma mais simples, a estimativa do custo de reutilização consiste na avaliação do esforço de desenvolvimento e manutenção do componente, face ao número de vezes que é expectável o componente ser reutilizado.

A especificação de variantes é uma especificação de requisitos extendida por forma a incluir informação sobre a forma como os componentes poderão ser reutilizados. Para ajudar a transformar estas especificações de variantes em componentes reutilizáveis,

exprimem-se estas especificações em termos de produtos variantes. Os produtos variantes podem ser descritos de diversas formas, incluindo a parametrização ou requisitos genéricos. A especificação de variantes procura descrever, de uma forma heurística, um conjunto limitado de componentes, a partir de uma gama potencialmente infinita de produtos variantes, que contém os componentes com maior probabilidade de reutilização. Embora a especificação de requisitos não seja normalmente vista em termos de especificação de variantes, estes podem estar presentes sob a forma de requisitos de modularidade, portabilidade, desempenho, *etc*.

Limites de parametrização

A parametrização é uma das formas mais utilizadas para tornar os componentes mais genéricos como resultado da especificação de variantes, atrás referidos. Numa primeira aproximação, os parâmetros podem ser elementos das linguagens de programação, por exemplo, dados, funções e tipos de dados. À medida que as necessidades de configuração vão aumentando torna-se necessário dispor de mecanismos mais flexíveis de parametrização. A utilização de mini-linguagens para resolver problemas específicos permite extender os limites de parametrização. Na sua forma mais simples, as mini-linguagens podem ser simples bibliotecas que aumentam o vocabulário da linguagem original [BB91]. Na sua forma mais elaborada, as mini-linguagens são formas claras e sucintas de definir soluções para um dado problema através da composição de um pequeno número de objectos e operadores. Por exemplo, algumas ferramentas **UNIX**, como o `awk` e o `sed`, utilizam parametrizações que variam de simples constantes a pequenos programas que particularizam o seu comportamento para situações específicas.

A parametrização excessiva não conduz, necessariamente, a componentes mais reutilizáveis ou fáceis de compreender. Isto deve-se ao facto de a parametrização excessiva ser funcionalmente equivalente a desenvolver uma mini-linguagem altamente genérica. Graus muito elevados de parametrização tendem a aproximar-se do chamado limite de Turing, o ponto a partir do qual a parametrização é tal que o seu poder expressivo se compara ao da linguagem de desenvolvimento original. A partir deste ponto, a mini-linguagem permite a descrição de qualquer programa, apresentando semelhanças

Tabela 2.1: Limites de parametrização na especialização de componentes.

Grau de parametrização	Potencial de reutilização
muito baixo	não reutilizável (uma só utilização)
baixo	baixo (muito específico)
médio	alto (fácil especialização)
alto	baixo (muito genérico)
muito alto	não reutilizável (limite de Turing)

com a máquina de Turing. Na melhor das hipóteses, um esquema de parametrização que atingiu o limite de Turing é tão complexo como uma linguagem genérica (*"general-purpose language"*), podendo facilmente ficar mais complexo. Neste ponto o potencial de reutilização anulou-se pois o esforço de especialização equivale ao esforço de desenvolvimento, sendo muitas vezes mais dispendioso que desenvolver o componente de origem.

Os melhores resultados de parametrização são obtidos através de combinações criteriosas de parâmetros que cubram as aplicações mais comuns do componente, requerendo simultaneamente um esforço de especialização reduzido aos seus utilizadores. As opções que são mais prováveis de vir a ser encontradas devem ser as mais fáceis de especializar possível, enquanto que, quanto menos provável for a sua ocorrência, mais complexa pode ser a sua especialização. No limite teremos que, situações de muito baixa probabilidade de ocorrência, podem ser codificadas directamente.

2.1.3 Distância cognitiva

A eficácia das abstracções numa técnica de reutilização de software pode ser avaliada em termos do esforço intelectual exigido para as utilizar. Melhores abstracções significam que menos esforço de especialização é exigido ao utilizador. Como ajuda na avaliação, introduz-se o conceito de distância cognitiva como uma medida intuitiva para comparar abstracções.

Definição 2.5 (distância cognitiva) *A* distância cognitiva *é a quantidade de esforço intelectual que é necessário gastar para fazer um sistema de software evoluir de um estado de desenvolvimento para outro.*

Na definição 2.5 verifica-se que a distância cognitiva não pode ser quantificada.

Ao criar uma técnica de software deve-se ter em conta a minimização da distância cognitiva através das seguintes normas:

1. utilização de abstracções imutáveis e parametrizáveis que sejam simultaneamente expressivas e sucintas,
2. maximização da parte privada das abstracções,
3. utilização de mecanismos automáticos para passar de especificações de abstracções para realizações de abstracções como, por exemplo, compiladores.

Resumindo [Kru92],*"Para que uma técnica de reutilização de software seja eficaz, é necessário reduzir a distância cognitiva entre o conceito inicial do sistema e o resultado executável final."*

O conceito de distância cognitiva, embora simples, é difícil de verificar na prática. É, no entanto, possível ordenar grandes grupos de técnicas de reutilização de software dependendo da importância dos critérios considerados [Kru92]. A tecnologia de reutilização ideal reduziria a distância cognitiva, independentemente dos critérios utilizados. Ou seja, o desenhador de software utilizando esta tecnologia seria capaz de rapidamente seleccionar, especializar e integrar especificações de abstracções que obedeçam a um conjunto de requisitos informais. Para mais, seria possível obter automaticamente um sistema executável a partir da especificação das abstracções, independentemente do domínio de aplicação a considerar.

2.2 Técnicas de classificação e selecção

Na produção de software, com base em componentes reutilizáveis, a correcta identificação de abstracções decorrente da fase de análise de requisitos é extremamente importante. É com base nas abstracções obtidas que será possível procurar um conjunto de

componentes com características funcionais e não funcionais similares. Para recuperar um conjunto de componentes similares mediante a descrição das referidas abstracções é necessário dispor de um processo de selecção para efectuar as buscas e comparações pertinentes. Este conjunto de componentes deve ser o mais pequeno possível e conter os componentes que melhor respondem às necessidades. O processo de selecção ideal recuperará um só componente que corresponde à menor distância entre os requisitos apresentados e as características dos componentes.

A utilização de boas técnicas de classificação de componentes é essencial para que os componentes sejam descritos de uma forma completa e sintética, facilitando e melhorando o processo de selecção [MPMM98]. Desta forma, embora as técnicas de catalogação e classificação não façam parte do processo de reutilização, mas do processo de produção de componentes reutilizáveis, são normalmente abordadas em conjunto com as técnicas de selecção. Aliás, a qualidade das técnicas de selecção só é mensurável pelo processo de selecção.

Exemplo 2.3 *Numa biblioteca, o sucesso da recuperação depende, em grande medida, da qualidade da abstracção escolhida para descrever os livros. Os bibliotecários têm formação especial por forma a garantir uma classificação consistente dos livros arquivados. O treino permite descrever de uma forma consistente as propriedades físicas do livro: nome, autor, páginas, ISBN ou cota. No entanto, como os utilizadores procuram a informação contida nos livros e não os livros* de per si*, é preciso encontrar formas de descrever o conteúdo. Tal como no caso dos livros, também no caso dos componentes de software, o sucesso a descrever conteúdos tem sido limitado.*

Para obter os componentes com características semelhantes aos requisitos é necessário que o processo de selecção permita descrever de uma forma completa os requisitos [MN97, FP94, MMM95]. É igualmente essencial que os componentes existentes estejam também eles descritos de uma forma completa, para que os mecanismos de selecção determinem correctamente a distância mínima. O processo de selecção deve, para tal, verificar a seguinte lista de requisitos:

Riqueza da descrição Deve ser possível descrever de uma forma completa, os componentes para que a determinação das suas características seja rigorosa. Logo, deve

ser possível descrever características estáticas, dinâmicas, métricas, funcionais e não funcionais do componente, cabendo ao processo de selecção utilizar apenas aquelas que também fazem parte da pergunta;

Formulação da pergunta A descrição dos requisitos deve ser completa e flexível, isto é, deve permitir exprimir todas as facetas dos requisitos deixando alguma liberdade aos mecanismos de selecção;

Determinação da semelhança Deve ser possível calcular a menor distância entre os requisitos e as características dos componentes. A utilização de técnicas difusas ("*fuzzy*") tem sido especialmente bem sucedida pois é menos sensível a ambiguidades e imprecisões na formulação da pergunta. No entanto, nem todos os critérios podem ser simultaneamente garantidos, conduzindo a uma taxa de recuperação elevada ou à eliminação de bons candidatos.

Os métodos de classificação estão divididos quanto à forma e quanto ao método. A forma de classificação é automática se for efectuada por ferramentas, ou manual se exigir a intervenção humana. Existem também algumas situações híbridas, designadas por semi-automáticas ou assistidas, em que parte do processo de classificação é automático, outra parte é manual. Embora os processos automáticos sejam mais atractivos, por exigirem menos esforço, os resultados obtidos são menos atractivos [Jus98]. O método de classificação pode ser associativo, quando tem por base o conteúdo do componente, e atributivo quando tem por base atributos inferidos manual ou automaticamente. Embora os métodos atributivos sejam predominantemente manuais, existem casos, como por exemplo, os métodos baseados em métricas, que podem ser automáticos. Da mesma forma, os métodos associativos são predominantemente automáticos, embora existam excepções [Jus98].

Os métodos de selecção são caracterizados quanto ao emparelhamento ("*matching*") e quanto à qualidade do processo. O emparelhamento descreve a forma de comparação dos requisitos com as características dos componentes [MMM95, FP94, Jus98]. O emparelhamento exacto só devolve os componentes que obedecem rigorosamente a todos os requisitos. Esta forma de emparelhamento é utilizada em processos de integração automática, mas o seu sucesso só é aceitável sob condições controladas. A compilação

separada pode ser vista como um processo de selecção exacta com base em assinaturas, para uma integração automática sem especialização. Na quase totalidade dos casos de desenvolvimento de software o emparelhamento é aproximado, com base na distância ou na ordem parcial. O cálculo da distância tem por base sistemas baseados em métricas, uma vez que produz um valor quantitativo [JM98, MMM95, FT96, ZBS98, CB91]. A determinação da ordem parcial baseia-se na criação de estereótipos para as comparações por grupos, ou classes de soluções que são, ou não são, garantidas [MMM95, PDF87, RM95, PD91a].

A qualidade do processo de selecção é o parâmetro que permite determinar o sucesso não só do processo de selecção como dos processos de classificação e de especialização com que interage. Os critérios de qualidade são [Jus98, FP94]:

Recuperação A taxa de recuperação mede o número de componentes obtidos no processo de selecção face ao número total de candidatos que obedecem aos requisitos. Uma recuperação baixa significa que muitos componentes interessantes estão a ser ignorados.

Precisão A precisão mede o número de componentes realmente utilizáveis face ao número de componentes obtidos no processo de selecção. Uma baixa precisão significa que o método de selecção devolve muitos componentes que, na realidade, não obedecem às especificações.

Sobreposição A sobreposição oferece uma medida comparativa entre dois métodos, ou seja, não é uma característica intrínseca do método. A sobreposição mede o número de componentes recuperados por ambos os métodos face ao número de componentes diferentes que os dois métodos recuperaram em conjunto. Um valor baixo de sobreposição permite concluir que métodos diferentes recuperam componentes diferentes, mesmo que tenham valores de recuperação e precisão semelhantes.

Tempo de busca O tempo de busca mede o tempo gasto para obter os resultados pretendidos. Notar que embora estes tempos possam diferir significativamente, o custo de adaptação dos componentes obtidos pode ser suficiente para contrabalançar as diferenças de tempo em várias ordens de grandeza.

Custo de adaptação O custo de adaptação mede uma estimativa do esforço, em tempo e recursos utilizados, que se prevê necessário para adaptar o componente. Esta medida é muito subjectiva, pois equipas com métodos de trabalho diferentes podem facilmente alterar a ordem de valores. No entanto, o custo de adaptação permite verificar se o componente de adaptação mais fácil está entre os recuperados.

Além dos critérios funcionais acima apresentados, deve-se ainda ter em conta, critérios não funcionais [Jus98], tais como:

Interface utilizador A interface utilizador deve ser avaliada quanto ao seu poder descritivo, facilidade de manejo, curva de aprendizagem e adaptabilidade às preferências do utilizador.

Esquema de classificação Os esquemas de classificação devem ser avaliados quanto à correcta descrição dos componentes, atribuição de um único lugar na organização, colocação de componentes semelhantes em zonas próximas e objectividade dos critérios de classificação.

Processo de recuperação O processo de recuperação deve ser avaliado quanto à facilidade de utilização da linguagem de interrogação, flexibilidade de acesso ao repositório, adaptabilidade aos requisitos do utilizador (número máximo de componentes, ordenação, *etc.*), possibilidade de refinar buscas (por iteração ou interacção) ou capacidade de combinação de métodos.

As aproximações existentes para selecção de componentes cobrem um largo espectro de métodos de classificação e de algoritmos de busca e emparelhamento. Seguidamente apresenta-se uma visão geral sobre os resultados obtidos em diversas áreas. Os modelos de representação de componentes classificados para selecção, encontram-se agrupados em famílias com características semelhantes, seguindo-se uma aproximação de complexidade crescente para facilitar a sua compreensão.

2.2.1 Modelos textuais

Os modelos de representação baseiam-se na utilização de descrições textuais, com base na linguagem comum ou natural. A possibilidade de usar uma linguagem semanticamente rica permite descrever características funcionais e não funcionais com igual expressividade. No entanto, a ambiguidade natural neste tipo de descrições é um inconveniente, em especial se o utilizador não estiver familiarizado com a terminologia. O mecanismo de busca é, em geral, simples e as perguntas são fáceis de formular. Por outro lado, a presença do conceito *"quicksort"* em *"Ao contrário do quicksort, esta rotina..."* não implica que o componente o utilize. Da mesma forma, a ausência de um termo não implica que o componente não esteja com ele relacionado, por exemplo devido à utilização de uma terminologia diferente. Resumindo, a codificação através de texto não pode ser considerada sólida ou completa.

Os resultados, do ponto de vista da precisão e recuperação, do uso de modelos textuais não se distanciam muito dos restantes métodos disponíveis. O maior inconveniente destes métodos reside no tempo de busca. Se estivermos na presença de poucos componentes, estes métodos podem ser considerados aceitáveis e utilizáveis. Contudo, num contexto de abundância de componentes, em que existem muitas dezenas de componentes semelhantes, o tempo necessário torna-se proibitivo.

Manuais de utilização Os manuais de utilização acompanham a distribuição dos próprios componentes, em especial quando os componentes são distribuídos em caixas pretas, para permitir a compreensão das suas características funcionais ou não. Esta forma pode ser considerada aceitável em componentes de alta qualidade e que exibam uma elevada estabilidade temporal. No entanto, mesmo nestes casos, frequentes erros de interpretação, já para não falar dos de escrita, conduzem não só a uma selecção deficiente como a uma utilização incorrecta [BD97]. Mais frequente é, contudo, a discrepância entre o conteúdo do manual e as características do componente. Mesmo em produtos considerados de qualidade, a documentação representa aquilo que se pretende que o produto realize e não aquilo que ele efectivamente faz. Por outro lado, pouco software pode ser considerado completamente estável e não sujeito a alteração.

Embora existam técnicas que permitam manter a documentação sincronizada com o código [HW86, HS90], estas exigem vastos recursos humanos, o que aumenta o custo e o tempo de desenvolvimento do produto.

Hipertexto Com o objectivo de facilitar a navegação dentro da biblioteca de componentes foram adaptados mecanismos de hipertexto à selecção de componentes. A utilização de hipertexto obriga, contudo, a um ainda maior dispêndio de recursos na descrição dos componentes, que já representava a parte mais frágil dos manuais de utilização. Além da descrição do componente é necessário estabelecer associações com outros componentes, sejam estes semelhantes, mais genéricos ou mais específicos [HS90, HW86]. O principal problema técnico do hipertexto é o desenvolvimento e manutenção da rede de associações. Adicionar um componente à rede é muito complicado, pois devem considerar-se todas as possíveis associações com o novo componente. Retirar um componente da rede é igualmente uma operação complexa, uma vez que se deve garantir a destruição de todas as ligações previamente existentes com esse componente. Do ponto de vista da busca, o panorama não é igualmente simples: não é fácil decidir qual o caminho a seguir; nem sempre a rede encontrada é adequada; não existe garantia de encontrar aquilo que se procura. O utilizador pode ficar a navegar em círculos, uma vez que não é possível explorar todos os caminhos. Algumas propostas mais recentes permitem construir automaticamente a rede com base em informação sobre as associações dos componentes, mas não melhoram a navegação e consequente selecção dos componentes desejados [Jus98].

Linguagem natural Estas técnicas caracterizam-se por fazer uma análise lexicográfica, sintáctica e semântica das especificações escritas em linguagem natural dos componentes de software, sem pretender efectuar uma compreensão completa dos documentos. As técnicas de pesquisa baseadas em linguagem natural recorrem a uma base de conhecimento que armazena a informação semântica sobre um domínio específico de aplicação. O esforço concentra-se, essencialmente, na construção das bases de conhecimento, exigindo muitos recursos humanos. Estes sistemas apresentam ainda uma grande dependência de parâmetros como a linguagem natural, o domínio de aplicação

ou linguagem de programação. Os métodos de busca variam de linguagens de interrogação a interfaces gráficas, mas a sua utilização é complexa. A precisão é elevada devido à especificidade de cada método, o que permite efectuar escolhas esclarecidas entre componentes semelhantes.

Visão rápida Os métodos de visualização rápida utilizam um esquema de recuperação linear que pode ser visualizado pelo utilizador. Estes métodos funcionam, frequentemente, como auxiliares de outros métodos de selecção, permitindo avaliar os componentes recuperados apenas com o intuito de refinar a busca. Desta forma, o método de visão rápida funciona como um auxiliar no refinamento das buscas, permitindo ganhar sensibilidade em relação ao mecanismo de interrogação. Na maioria dos casos, estes métodos baseiam-se na ordenação hierárquica, por forma a permitir vários níveis de granularidade [BE96, WCG92, Jus98].

2.2.2 Modelos lexicais

Nos modelos lexicais, cada componente é descrito por palavras ou frases retiradas de um vocabulário pré-definido, em vez de ser descrito por texto livre. A qualidade de cada um destes métodos depende do conjunto de vocábulos escolhidos. Se a escolha dos vocábulos cobrir todo o espectro do domínio de aplicação em causa e se os vocábulos forem todos disjuntos, existe a possibilidade de descrever de uma forma compacta e coerente todos os componentes. Para mais, a granularidade da classificação obtida está directamente ligada com o número de vocábulos, assumindo que todos os vocábulos cobrem áreas disjuntas do domínio.

A classificação dos componentes com base nestes métodos necessita da intervenção humana. Embora existam métodos de classificação automática, estes podem ser facilmente induzidos em erro. Aliás, um dos pontos fracos destes métodos, para além de terem classificações essencialmente manuais, reside na elevada possibilidade de erro humano no processo de classificação. Tal como na classificação de livros, é muito importante a experiência e o treino da pessoa que efectua a classificação. Isto porque a classificação

não requer a existência de determinado vocábulo na sua documentação, incluindo o código, para poder ser correctamente descrito por esse mesmo vocábulo. Uma vez que o número de vocábulos é fixo e pré-definido, é necessário encontrar aquele ou aqueles que melhor descrevem o componente.

O processo de selecção é constituído por uma classificação inicial dos requisitos. Esta classificação obriga a que o processo de selecção seja feito com o mesmo cuidado e experiência que a classificação inicial. Se a classificação dos requisitos seguir outros critérios, que os utilizados na classificação dos componentes, a precisão e recuperação dos métodos ficam comprometidas. Como não se pode esperar que a selecção seja efectuada apenas por quem fez a classificação dos componentes, torna-se necessário incluir uma descrição completa dos referidos critérios.

Métodos enumerativos Nos métodos enumerativos os componentes são classificados segundo um esquema hierárquico. Em cada nível da hierarquia torna-se necessário escolher entre cada uma das opções disponíveis [Dew79, FP94, MMM95, Jus98, dSC98a, dSC98b]. Essas opções deverão ser suficientemente disjuntas, para evitar ambiguidades na sua escolha durante o processo de selecção. Estes métodos assemelham-se a um sistema hierarquizado de ficheiros, onde a procura de um ficheiro obriga a percorrer um caminho desde a raiz do sistema. Tal como no caso do sistema de ficheiros, a busca fica grandemente facilitada através de uma escolha criteriosa dos nomes – vocábulos – utilizados em cada nível da hierarquia. Se a escolha dos níveis não for criteriosa cai-se na busca sequencial que desvirtua e inutiliza todo o processo de classificação e selecção por métodos enumerativos. Por outro lado, o facto de os métodos enumerativos serem tão estruturados, facilita a sua compreensão e utilização, como se pode verificar através do paralelo com os sistemas de ficheiros. A construção de uma hierarquia bem formada ajuda os utilizadores a compreender os relacionamentos entre os vocábulos utilizados na classificação e oferece um mecanismo natural de busca com base na navegação em largura e profundidade.

A desvantagem destes métodos reside no facto de partir o domínio de aplicação em categorias mutuamente exclusivas. Este facto, não só exige um grande conhecimento do domínio em questão como limita as associações que é possível representar. O esquema

de classificação é também difícil de alterar, uma vez que introdução ou eliminação de um novo vocábulo pode obrigar à reclassificação de todos os componentes. Se atendermos, como vimos atrás, que esta classificação é essencialmente um processo manual, o esforço envolvido pode ser muito grande. O método obriga a definir à partida um esquema completo e consistente, o que não é fácil de conseguir.

Palavras-chaves Os métodos baseados em palavras-chaves recorrem a subconjuntos de vocábulos disponíveis para a descrição dos componentes [MMM95, FP94, Jus98]. O conjunto de termos, designado por perfil, descreve as características consideradas relevantes do componente. A relevância dessas características é variável, pelo que, ou se opta por um limiar a partir do qual o vocábulo é considerado irrelevante, ou existe um número máximo de vocábulos a considerar como relevantes. Alternativamente, podem ser atribuídos pesos a cada vocábulo, por forma a calibrar o posterior processo de selecção.

Alguns destes métodos recorrem a mecanismos automáticos para a classificação dos componentes. No entanto, a estereotipagem necessária, recorrente da utilização de um dicionário fixo, obriga a utilização de dicionários de sinónimos. Esta utilização permite tirar paralelos entre a linguagem utilizada para descrever o componente e os vocábulos utilizados na sua classificação. No entanto, verifica-se que nem sempre são correctos, conduzindo a classificações enganadoras.

O processo de selecção recorre a expressões lógicas, para determinar quais os componentes que obedecem a determinados requisitos. Através de uma combinação cuidada de condições, usando conectivos `ou`, `e` e `negação`, é possível obter expressões que serão avaliadas componente a componente. A refinação das expressões é essencial para o sucesso da busca. Alguns sistemas mais recentes permitem formular as perguntas em linguagem natural mas, após a sua interpretação, o mecanismo de procura é idêntico [BAB+87].

Comparativamente, os métodos de selecção por palavras-chaves oferecem uma boa recuperação embora a precisão seja inferior aos métodos enumerados. Esta falta de precisão traduz-se também numa elevada sobreposição entre interrogações distintas, com

o mesmo método. Assim, a formulação correcta da interrogação é essencial para o sucesso da pesquisa, ou seja, para o aumento da precisão do método.

Facetas Os métodos baseados em facetas combinam as duas aproximações anteriores, procurando eliminar a partição exclusiva das hierarquias que reduz a recuperação, e melhorando a precisão das palavras-chaves à custa de as dividir em grupos. Os métodos baseados em facetas definem um conjunto de perspectivas, permitindo classificar cada uma dessas perspectivas com um conjunto de palavras-chaves [PDF87, PD91a, Jr.95, Lea97]. Estas facetas, ou perspectivas, estão geralmente ordenadas com base na sua importância relativa. Os objectos são então classificados através da escolha de um só vocábulo de cada uma das facetas. O desenvolvimento das facetas é feito tendo em conta a importância do vocabulário num determinado domínio, que é depois agrupado em facetas. As classificações com base em facetas permitem uma maior liberdade no processo de classificação, permitindo exprimir associações complexas por combinação de facetas e vocábulos. É também mais fácil de modificar que a classificação hierárquica, uma vez que cada faceta pode ser alterada sem afectar as restantes.

O processo de selecção pode ser semelhante aos utilizados nos métodos anteriores, no entanto, existem algoritmos que permitem a recuperação com base em emparelhamento aproximado. Estes algoritmos recorrem a linguagens difusas ("*fuzzy.*") e distância funcional [JMM97, JM98, RM95, Jus98]. De entre os métodos lexicais este tem sido o mais explorado e o que melhores resultados tem apresentado [FP94, MMM95].

2.2.3 Modelos estruturais

Ao contrário dos modelos lexicais, os modelos estruturais não impõem limitações no número de vocábulos utilizados ou no dicionário de onde são extraídos. Quando o vocabulário não é controlado, não existem restrições quanto aos termos utilizados para classificar o componente. Os termos podem ser retirados de qualquer origem, mas, normalmente, são extraídos dos próprios componentes. Desta forma a classificação pode ser muito mais automatizada que nos modelos lexicais. Este facto reduz o custo de classificação e permite introduzir alterações no mecanismo de classificação com custos

reduzidos, facilitando o refinamento do processo de classificação. Além disso, os termos utilizados para descrever o componente já não necessitam de ser estereotipados, podendo ser muito mais específicos e expressivos. Este facto permite maior precisão nas buscas, mas reduz a recuperação.

Estes métodos recorrem, frequentemente, a mecanismos estatísticos e linguísticos para a extracção automática dos termos utilizados na classificação. Os mecanismos estatísticos baseiam-se na frequência com que as palavras aparecem na descrição, textual ou estruturada, do componente. Aquelas palavras que aparecem com maior frequência são consideradas úteis para o processo de classificação. Estes mecanismos recorrem a uma lista de palavras que indicam os termos a ser ignorados, por forma a reter apenas palavras efectivamente relevantes para o processo de selecção. Para mais, o processo de contagem pode ter em conta construções gramaticais, prefixos ou sufixos, sinónimos ou homónimos, *etc*. Inclusivamente, pode-se obter termos com raiz comum, como por exemplo, *cálculo* a partir de calcular, calculadora, calculista, *etc*. Contudo estes métodos recuperam muitos termos fora do contexto e requerem grandes amostras para que o cálculo das frequências seja apropriado. Além dos mecanismos estatísticos, a investigação recente recorre ao processamento com base no tratamento da linguagem natural para melhorar a captação dos termos relevantes existentes na documentação. No entanto, as descrições dos componentes de software não se caracterizam pela sua grande dimensão, nem pela sua clareza, nem pela utilização de linguagem natural gramaticalmente correcta [PD91a].

Pares atributo-valor Nos modelos baseados em pares atributo-valor, o componente é classificado por um conjunto de atributos e seus valores. Podem ser escolhidos, por exemplo, atributos como a acção, área funcional e linguagem. A cada um destes atributos é associado um valor. O atributo acção, por exemplo, poderá ter associado valores como enumerar, ordenar, procurar, mover e pré-processar [BAB+87, DFF97]. Este método apresenta muitas semelhanças com o método baseado em facetas. De facto, as facetas podem ser vistas como equivalentes a atributos, só que os valores possíveis para caracterizar uma faceta são fixos enquanto os valores associados a um atributo podem ser escolhidos de um conjunto que não é pré-definido. Além disso, no método das face-

tas, um domínio é descrito com número reduzido de facetas (em geral sete ou menos), enquanto nos pares atributo-valor não é imposto limite. Para mais, ao contrário das facetas, este método não impõe nenhuma ordenação entre cada um dos atributos, nem recorre a tratamento de sinónimos, pelo menos nas versões mais simples [FP94].

Assinaturas A classificação e selecção com base em assinaturas é uma aproximação mais formal, beneficiando do facto de o software, mesmo que possa ser visto como texto, é de facto bastante estruturado. A ideia baseia-se em descrever os componentes ao nível da interface. Quando efectuado de uma forma simplista, a classificação limita-se a características sintácticas do componente, um pouco à semelhança dos mecanismos usados pelas linguagens que oferecem a possibilidade de compilação separada. Na sua versão mais simples, a assinatura é constituída apenas pelo nome da entidade, enquanto soluções mais trabalhadas utilizam alguma informação que permite agrupar as entidades por tipos [MPMM98]. Na sobreposição ("*overloading*") de funções, por exemplo, o número, tipo e ordem dos argumentos são utilizados para distinguir entre funções com o mesmo nome mas comportamentos diferentes. Da mesma forma, os métodos de classificação e selecção com base em assinaturas utilizam a informação de interface para catalogar e comparar componentes. Soluções mais recentes e complexas têm procurado eliminar restrições sintácticas, tais como a ordem dos parâmetros, composição de tipos agregados ou subtipificação [ZW95]. Estas soluções têm encontrado problemas na comparação de tipos agregados definidos pelo utilizador, hierarquias de tipos, parâmetros opcionais e parâmetros genéricos, tendo tido dificuldade em chegar a conclusões correctas sob situações não controladas [Zar96].

Especificações Os modelos baseados em especificações podem ser considerados uma extensão dos modelos baseados em assinaturas. Esta técnica tem origem no enriquecimento das descrições baseadas em assinaturas com informação estrutural e semântica dos componentes [MPMM98, MMM95]. A informação adicional sobre o componente pode ser obtida de diversas formas. Uma dessas formas consiste na experimentação do componente e na recolha de amostras do seu comportamento [PP93, Atk97]. Esta solução embora utilizável em funções simples torna-se excessivamente complexa em

componentes de maiores dimensões. Outra destas formas utiliza descrições mais formais do comportamento dos componentes [ZW97, Zar96]. Esta técnica, tal como outras baseadas em princípios formais, não têm em conta critérios não funcionais. No entanto, permite uma selecção de grande precisão e boa recuperação com base em critérios exclusivamente funcionais. Posteriormente é possível efectuar uma segunda selecção para fazer uma triagem com base em critérios não funcionais. Uma outra forma consiste na recolha de informação sintáctica e semântica do componente com base em descrições não formais. Estas baseiam-se nas linguagens de programação e desenho utilizadas durante o desenvolvimento do componente, incluindo a interpretação de comentários [ED97]. Estes métodos utilizam os identificadores das linguagens de uma forma semelhante aos vocábulos dos modelos estruturais [ED97, dSC98a, dS97, dSC98b]. Estas aproximações, ao contrário das baseadas em princípios formais, necessitam em geral de uma primeira triagem, permitindo obter uma boa precisão em requisitos muito específicos e critérios não exclusivamente funcionais.

2.2.4 Outros modelos

Além dos métodos de classificação e selecção atrás descritos, que constituem a maioria, existem ainda algumas experiências em outros domínios. Alguns dos métodos que se seguem têm grande aplicação na reutilização de software mas não nas áreas de classificação e selecção de componentes de software [FT96, ALMD96, BD97, vMV95]. Qualquer dos métodos que a seguir se descreve encontra-se ainda num estado em que as conclusões úteis são limitadas [MMM95]. No entanto, estes métodos podem funcionar como ajudas complementares aos métodos anteriormente descritos, permitindo enriquecer as descrições com informação potencialmente útil no processo de selecção.

Métricas As métricas de software oferecem mecanismos de automatizar a extracção e a condensação de informação sobre os componentes, reduzindo o tempo necessário para os analisar. As métricas de software dividem-se em dois grandes grupos: estáticos e dinâmicos [ZBS98]. Uma vez que estamos preocupados em analisar código que não

nos é familiar, estamos mais interessados nas métricas estáticas que se baseiam, em geral, na análise estática do código fonte. Métricas estáticas típicas incluem a dimensão do programa, métricas da complexidade da estrutura de dados, métricas de fluxo de informação, *etc*. As métricas dinâmicas, tais como desempenho, utilização de recursos, *etc.*, podem ser úteis numa segunda fase do processo de selecção. As métricas podem ser incluídas em facetas ou em pares atributo-valor para dar uma medida da ordem de grandeza de determinada perspectiva [FT96, ZBS98, CB91, AM97]. As métricas podem ser quantitativas, representadas por grandezas numéricas, ou difusas como por exemplo: grande, médio ou pequeno. As ordens de convergência de algoritmos numéricos ou de ordenação são métricas importantes no processo de selecção, e que juntamente com exigências de recursos de memória e computacionais podem ser decisivas na escolha entre componentes funcionalmente equivalentes ou semelhantes.

Formais Os métodos formais baseiam-se na descrição das características formais dos componentes. Nestes métodos não existe propriamente um processo de classificação, uma vez que a própria descrição do componente funciona como base de classificação, não sendo possível estabelecer nenhuma ordem entre os diversos componentes, ao contrário da informação semântica dos modelos estruturais. O processo de selecção parte de uma descrição formal dos requisitos, por exemplo, sob a forma de pré e pós-condições com base em predicados de primeira ordem ou sobre representação categorial [Cre98]. A recuperação tem por base um demonstrador de teoremas para comprovar se as especificações dos componente são formalmente equivalentes aos requisitos indicados na interrogação. A grande vantagem dos métodos formais reside na sua elevada precisão, a mais alta se não se considerarem características não funcionais [MMM97, AL95a, NO95, TPW95, PA97, CC97]. De entre as desvantagens encontram-se o pequeno número de componentes descritos formalmente que existe, o tempo de processamento dos algoritmos de busca que é muito elevado; a limitação a emparelhamentos isomórficos, embora outros emparelhamentos estejam a ser estudados, como o composicional [Cre98]; o estado actual da tecnologia de demonstração de teoremas dificulta todo o processo, tornando mais atractiva a utilização de descrições textuais, léxicas ou sintácticas [PP93, Jus98, MMM95].

Padrões Os métodos baseados em padrões descrevem o componente com base na repetição de exemplos tipo ou padrões. Os padrões base incluem sequências, alternâncias (`if` ou `case`) ou iterações (`while`, `repeat` ou `for`). Cada um dos padrões encontrados é depois combinado de uma forma ascendente até descrever o componente [HPLH90, PP94]. Noutros casos, o padrão pode descrever o problema e o seu contexto, a solução e as consequências da adopção da solução. O problema pode ser descrito através de um exemplo concreto. A solução descreve as entidades que nela participam e as suas responsabilidades e colaborações [BD97, HLS97, ALMD96]. Os padrões são particularmente indicados na descrição de arquitecturas ou de componentes com diferenças estruturais subtis.

Cognitivos Os métodos cognitivos procuram melhorar a compreensão das características de um componente por forma a facilitar a sua correcta reutilização. O objectivo destes métodos é o de oferecer descrições que permitam compreender melhor o domínio de utilização, reduzir discrepâncias entre o trabalho produzido por engenheiros de boa qualidade e de qualidade deficiente [Lat97]. Estes métodos procuram obter um nível de abstracção mais elevado, por eliminação de pormenores, facilitando a sua compreensão. Tais pormenores incluem, por exemplo, questões de desenho de algoritmos de ordenação, tais como se a ordenação é feita directamente no ficheiro ou em memória, qual o tipo algoritmo usado, qual a interface oferecida, *etc*. Estes métodos são especialmente úteis em situações de abundância de componentes, onde é necessário proceder a uma primeira triagem. Consegue-se, desta forma, obter uma boa recuperação, embora com baixa precisão. Estes métodos podem ser vistos de uma forma simplista, como mecanismos de engenharia inversa, procurando inferir quais os requisitos e quais os problemas que o componente pretende resolver. A utilização de bases de conhecimento e mecanismos de assimilação formam a essência destes métodos, cujo uso em componentes pequenos tem permitido obter conclusões interessantes [Fis87, Sen97, vMV95].

2.3 Técnicas de especialização e integração

As técnicas de selecção descritas na secção 2.2 permitem pesquisar bibliotecas de componentes e obter um número de candidatos a serem reutilizados. Estes candidatos deverão ser os que melhor respondem aos requisitos pretendidos de entre os disponíveis nas bibliotecas, devendo as técnicas de selecção utilizadas exibir índices de precisão e recuperação elevados. O processo de selecção é um processo iterativo que produzirá um número reduzido de candidatos a reutilização. Este processo deverá ser repetido até se obter o melhor conjunto de componentes. Este número de componentes deverá ser também reduzido para facilitar o processo de especialização. Note-se que o processo de selecção funciona como um processo de eliminação de candidatos improváveis, pelo que a tarefa de especialização só abordará os componentes efectivamente seleccionados. Se o componente ideal – aquele que obedece exactamente aos requisitos pretendidos – não for seleccionado, então este não será considerado na especialização. Tal facto deve-se à falta de precisão do método utilizado no processo de selecção ou a um mau método de classificação. Esgotado o processo de selecção torna-se necessário avaliar as características de cada um dos componentes seleccionados por inspecção directa.

O processo de especialização tem por objecto um conjunto, supostamente reduzido, de componentes e deverá avaliar os custos de adaptação de cada componente ao ambiente final de utilização. O custo de adaptação de cada componente é função de elementos tão distintos como a linguagem e o ambiente de integração, a experiência e a organização da equipa de especialização, a necessidade de possíveis alterações aos requisitos (funcionais ou não). Desta forma, equipas com experiência e métodos de trabalho diferentes podem optar por especializar componentes diferentes, ou utilizar técnicas de especialização distintas. Como resultado, este processo produz um único componente, já alterado se houver necessidade, para a fase final de integração. As alterações podem ser tão simples como a fixação de um parâmetro a um valor constante. Se, por outro lado, forem necessárias alterações mais profundas, o processo de especialização pode corresponder à criação de um novo componente com base no seleccionado. Alternativamente, o processo de especialização pode optar pelo desenvolvimento do componente de origem, caso se verifique que os candidatos são mais difíceis de adaptar que

desenvolver um componente de origem. O desenvolvimento do componente deverá ser especialmente cuidado se for previsível a futura reutilização em outras situações.

O componente resultante do processo de especialização será posteriormente adaptado ao seu ambiente final através de um processo de integração. O processo de integração consiste na combinação de um conjunto de componentes, resultantes de processos individuais de especialização, por forma a formarem a aplicação final. O processo de composição que combina os diversos componentes num sistema final pode reduzir-se à edição das ligações ("*linker*") ou exigir uma linguagem de interligação de módulos. No caso de se utilizar uma linguagem de interligação esta pode permitir manipular a ordem dos parâmetros, fazer conversões de tipos ou alterações de ligações. Estas linguagens podem ser especialmente desenhadas para efeitos de reutilização ou fazer parte de outras linguagens mais genéricas. O seu objectivo é o de coordenarem a interligação entre os diversos componentes por forma a que as entidades importadas por certos módulos correspondam às exportadas por outros.

Uma vez que o processo de especialização produz um componente a ser composto no processo de integração, estes dois processos estão intimamente ligados e são normalmente tratados em conjunto. Aliás o processo de integração limita as possibilidades de especialização, pelo que seria incorrecto tratá-los separadamente. No entanto, os mecanismos disponíveis em cada processo são distintos e serão revistos nas secções 2.3.1 e 2.3.2 antes de analisar sistemas reais na secção 2.3.3.

2.3.1 Técnicas de especialização

Na sua maioria, os componentes não podem ser directamente reutilizados, mesmo que tenham sido especialmente desenhados para reutilização. É necessário efectuar um conjunto de adaptações, mais ou menos complexas, por forma ao componente responder rigorosamente aos requisitos e ao ambiente onde vai ser integrado. Estas adaptações podem ser divididas em adaptações funcionais (ou de caixas pretas), caso não necessitem de informação interna do componente, ou estruturais (ou caixas brancas), se alguma informação interna for necessária para realizar as modificações necessárias. As técnicas

de especialização podem ser agrupadas em cinco grandes grupos, que passamos a caracterizar, embora possam ser aplicadas de forma diferente por linguagens, ferramentas e ambientes de reutilização distintos.

Particularização

A particularização é uma técnica que permite restringir o comportamento de um componente a uma situação concreta. Esta particularização é realizada à custa da fixação de valores na parte parametrizável das abstracções. Para se poder particularizar é necessário que o componente tenha sido desenhado com vários comportamentos, e o processo de especialização consiste na escolha de um destes comportamentos ou realizações da abstracção [Kru92, MMM95]. A escolha do comportamento consiste na determinação dos valores de alguns argumentos. Os valores podem ser tipos de dados, no caso da generalidade (por exemplo, os *templates* em **C++**), ou simples valores numéricos. A particularização corresponde ao verdadeiro processo de especialização, uma vez que, desde que os valores escolhidos estejam dentro dos limites aceites, o componente continua a obedecer às especificações originais e logo aos requisitos pelos quais foi escolhido no processo de selecção. No entanto, como raramente os componentes são suficientemente genéricos para cobrir na totalidade a nova situação, torna-se necessário efectuar adaptações.

Cópia e modificação

Quando se encontra um componente que apresenta características semelhantes com aquele que se pretende integrar, uma das opções mais usadas consiste em fazer uma cópia e alterá-la de acordo com as novas necessidades. Ao efectuar a cópia garante-se que as entidades existentes não serão afectadas pelas alterações subsequentes. Por outro lado, as novas entidades podem optar pelo componente original ou pelo alterado. Do ponto de vista da reutilização, existe uma parte significativa do componente original que permanece inalterado, reduzindo o esforço necessário. No entanto, embora esta solução possa parecer atractiva numa primeira aproximação, o seu valor prático tem-se provado pouco eficaz. Este método apresenta, principalmente, dois inconvenientes: a

realização de modificações incompletas e a dificuldade de manutenção. A realização de modificações incompletas resulta da não avaliação da real consequência das alterações efectuadas, seja por descuido ou por falta de documentação. A dificuldade de manutenção traduz-se na proliferação de componentes quase iguais que, ao ser descoberto um erro numa parte comum, obriga a alteração de todos eles [Bos97].

Encapsulamento

A técnica do encapsulamento ("*wrapping*") permite o aproveitamento de muitos componentes, em especial quando já não existe documentação completa, como é geralmente o caso dos sistemas legados. O encapsulamento integra um ou mais componentes num só componente com uma nova interface. Através do redireccionamento dos pedidos, que recebe através dessa interface, para os seus sub-componentes, com um conjunto mínimo de alterações, consegue oferecer uma nova funcionalidade. Uma vez que os sub-componentes não são alterados, a sua funcionalidade só pode ser manipulada dentro dos limites da sua controlabilidade e observabilidade. Desta forma, é apenas possível realizar alterações simples. De entre estas alterações encontram-se a alteração da ordem dos argumentos, conversões de escala, conversões de tipos, *etc*. Uma importante desvantagem desta técnica resulta do considerável custo de adaptação necessário, uma vez que toda a interface dos sub-componentes necessita ser controlada pelo novo componente, mesmo aqueles elementos da interface que não carecem ser alterados. Também se verificou que a técnica de encapsulamento conduz a custos excessivos de adaptação e reduções significativas de desempenho [Hol93].

Agregação

Ao definir um componente de maiores dimensões e com um comportamento complexo, pode-se utilizar outros componentes e combiná-los por forma a reduzir o esforço total de desenvolvimento. Assim, parte do comportamento desejado é providenciado pelo sub-componente, que é desta forma reutilizado. Embora não exista uma distinção nítida entre encapsulamento e agregação, a agregação pretende produzir um novo componente à custa da acumulação de funcionalidade de componentes já existentes, enquanto o

encapsulamento é uma forma de modificar o comportamento dos componentes já existentes. A agregação é pois uma técnica construtiva, enquanto, quer a cópia e modificação quer o encapsulamento, são técnicas destrutivas, embora a última destrua apenas o comportamento e não a estrutura. A agregação pode ser vista de duas perspectivas: associação e composição [EGWZ97, BJR96, FS98]. A composição pressupõe que o sub-componente faz parte integrante do novo componente e a sua vida útil é a mesma deste último; além disso, todo e qualquer tipo de partilha é feita apenas através do novo componente, perdendo o sub-componente a sua identidade do ponto de vista exterior. A associação denota uma agregação que pode ter um carácter temporário e em que o sub-componente pode estar a ser simultaneamente partilhado por vários componentes, conservando para tal a sua identidade própria.

Herança

A herança é uma técnica característica do desenvolvimento orientado para objectos. Embora possa ser codificada em linguagens não orientadas para objectos [1], o desenvolvimento resulta da modelação por classes e subsequente especialização do seu comportamento de uma forma ordenada e controlada. Assim, a herança permite realizar em conjunto qualquer das quatro técnicas acima descritas [KA90]. Para mais, permite fazê-lo de uma forma organizada e que facilita posteriores especializações e tarefas de manutenção com custos reduzidos. A herança é uma forma de encapsulamento, pois permite redefinir e acrescentar operações ao componente original. Esta redefinição é efectuada de uma forma controlada, pelo que apenas as operações que dizem directamente respeito à parte do comportamento que se pretende alterar precisam ser especificadas, as operações responsáveis pelo restante comportamento são automaticamente herdadas. A herança é também uma forma de parametrização, uma vez que as funções redefinidas podem limitar-se a fixação de valores. Algumas linguagens, como o **C++**, permitem a definição de valores por omissão ou de tipos de dados como parâmetros (designados por *templates* em **C++** e classes parametrizadas em **UML**). A herança é ainda uma forma de agregação pois permite adicionar outros componentes ao definir o

[1] Os compiladores de **C++** mais antigos eram na realidade tradutores de **C++** para **C**.

novo componente. Finalmente, é obviamente possível criar novos componentes por cópia e modificação, mas tal torna-se desnecessário devido aos mecanismos disponíveis. Contudo, a herança é uma técnica que exige, na maioria das linguagens, a disponibilização de alguma informação interna ao componente o que a torna inutilizável quando essa informação não está disponível. Por outro lado, esta técnica só está disponível em ambientes de desenvolvimento orientados para objectos, embora seja possível utilizar esta técnica em algumas linguagens de programação não orientadas para objectos.

A herança, como mecanismo de reutilização de código, pode ser realizado sob a forma de delegação ou de concatenação, independentemente de a linguagem ser baseada em classes ou em protótipos [Ast96]. O facto de a linguagem ser baseada em classes ou em protótipos tem apenas a ver com a forma como os objectos são criados, através de uma primitiva ou por clonagem, não tendo ligação directa com a técnica de especialização. Na herança por delegação, é criado um novo objecto, contendo as características extendidas pelo processo de especialização, e este objecto delega as restantes características no objecto original de onde foi efectuada a especialização [GR89, US87]. Na herança por concatenação, o novo objecto é obtido por cópia do original, sendo depois introduzidas as novas características [Str91, Tai92].

Tabela 2.2: Técnicas de herança nas linguagens de programação.

	classes	protótipos
delegação	Smalltalk	Self
concatenação	C++	Kevo

O principal inconveniente da herança tem origem no seu uso excessivo e indescriminado. Pelo que se acabou de descrever poder-se-ia concluir que a herança é a solução para os inconvenientes da especialização: apesar da herança ter sido pensada como um mecanismo de especialização [GR89]. Ultimamente, tem-se verificado que a herança conduz a que os componentes fiquem muito mais dependentes uns dos outros, acabando por dificultar qualquer tarefa de manutenção [Lew94]. Além disso, a herança não deve ser confundida com agregação, embora possa agregar componentes como parte do processo [Rum93, BT95, Mey96].

2.3.2 Interligação de componentes

A última fase do processo de reutilização, tal como foi indicado na secção 1.2.3, consiste na integração de todos os componentes numa aplicação final. Esta integração é realizada por composição dos vários componentes e controlo da sua interoperação. Linguagens e ambientes diferentes realizam composições de componentes a níveis diferentes e suportam noções diferentes de composição. O desenvolvimento com base em componentes obriga a que exista um conjunto de componentes seleccionados que sejam compatíveis entre si. Caso os componentes já seleccionados não sejam compatíveis, o processo de especialização tem por função torná-los compatíveis. O processo de integração é essencialmente um processo de edição de ligações entre componentes [Tai98, SvD95, GC92]. Se dois componentes interagem, então um dos componentes – o emissor – inicia a interacção activando o receptor e passa-lhe o controlo. O receptor reage, realizando alguma acção e, dependendo da comunicação ser síncrona ou assíncrona, retorna o controlo para o emissor. Normalmente, alguma informação é passada entre os dois componentes ao interagirem um com o outro. Alternativamente, se a informação for extensa ou complexa, podem recorrer a um terceiro para o efeito.

O componente receptor pode ser ou não conhecido do emissor, caso em que as ligações entre os componentes são estáticas ou dinâmicas, respectivamente. Se estas ligações forem estáticas, ou seja, definidas no momento da construção da aplicação e não mais alteradas, o processo é equivalente à compilação separada de módulos. Se as ligações forem dinâmicas, então o componente de destino pode variar com o tempo, podendo nem estar definido no arranque da aplicação – equivalente ao carregamento e ligação dinâmicos de módulos. O facto de as ligações serem estáticas ou dinâmicas tem grande influência na flexibilidade que é possível atingir na aplicação final [NM95].

Uma efectiva interligação de componentes reutilizáveis implica que os componentes envolvidos possam ser compostos sem que tenham de ser conhecidos à partida. Só assim é possível alterar a aplicação sem modificar os componentes, ou seja, é possível ter um controlo dinâmico sobre a aplicação. Por exemplo, é possível substituir um algoritmo de ordenação por outro, sem alterações aos componentes que os implementam e de uma forma transparente para a restante aplicação. A programação orientada para ob-

jectos oferece, de uma forma limitada, a possibilidade de optar entre vários receptores no momento da sua invocação, sem que o receptor conheça o emissor. Desta forma, o receptor pode ser reutilizado num conjunto de situações sem alteração [Sam97].

A interoperação entre componentes depende da correcta determinação do receptor e da sua disponibilidade. Para que a interoperação resulte é necessária a aplicação de certos mecanismos de onde se salientam:

Carregamento e ligação O carregamento de componentes após o início da execução e a ligação dinâmica entre os elementos acessíveis de cada componente. A distinção entre a evolução da aplicação durante o desenvolvimento e durante a execução é artificial e limitativa. Este tipo de composição dinâmica oferece uma dimensão de reutilização adicional que pode representar um factor importante no desenvolvimento de aplicações modulares e duradouras.

Identidade A identidade é a propriedade de um componente que o distingue de todos os outros na aplicação. Em linguagens de programação, os componentes são distinguidos pela posição que ocupam na memória, enquanto em bases de dados são utilizadas chaves únicas. No entanto, para os utilizadores o nome do componente é o identificador mais utilizado.

Resolução de nomes Como a utilização de nomes é a forma mais frequente de identificar entidades que sejam acessíveis aos utilizadores, nomeadamente os componentes, é possível o aparecimento de conflitos de nomes. A resolução destes conflitos é essencial no caso de se pretender desenvolver aplicações com base em componentes reutilizáveis.

Sobreposição Uma solução particular para um certo tipo de conflitos de nomes consiste na sobreposição. A sobreposição permite que operações com o mesmo nome, mas com diferentes semânticas e realizações, sejam invocadas em função do número e tipo dos seus argumentos. Por exemplo, a operação de soma pode ser aplicada a pares de entidades do tipo inteiro ou cadeia de caracteres.

Polimorfismo Outro tipo de conflitos podem ser resolvidos à custa de várias formas de polimorfismo. Polimorfismo, do ponto de vista linguístico, afirma apenas que uma entidade pode assumir várias formas. Do ponto de vista dos nomes dos componentes, o polimorfismo reflecte a capacidade de uma mensagem produzir comportamentos diferentes com objectos de tipo diferente. Por exemplo, a representação gráfica de diferentes tipos de figuras geométricas pode apresentar a mesma interface recorrendo ao polimorfismo.

Visibilidade Uma forma de suportar em simultâneo entidades, nomeadamente componentes, com o mesmo nome consiste em definir mecanismos de visibilidade. Estes mecanismos permitem que em determinadas situações um componente com um certo nome seja visível, enquanto que noutra situação este poderá ficar inacessível, podendo ser substituído por outro. Por exemplo, variáveis globais, de programas codificados em linguagens procedimentais, podem ficar temporariamente ocultas – inacessíveis por nome – durante a execução de funções aninhadas que declarem variáveis com o mesmo nome.

2.3.3 Sistemas de interligação de componentes

Devido às cada vez maiores necessidades de especialização e integração que se exigem em todas as áreas do desenvolvimento de software, as diferenças entre os vários sistemas só se tornam visíveis de um ponto de vista cronológico. As diferenças residem mais na facilidade ou dificuldade com que se consegue obter cada um dos mecanismos pretendidos, bem como a eficiência com que são realizados.

Desenho de componentes

Ao nível da análise e desenho de componentes, as ferramentas existentes são muito semelhantes e classificam-se em métodos estruturados, orientados para objectos e formais. A este nível é importante poder exprimir de uma forma clara e concisa as características dos componentes. Torna-se, assim, mais fácil avaliar as alterações que é necessário efectuar para que o componente tenha as características desejadas bem como ter uma ideia

muito clara das interligações que é necessário vir a suportar na realização. Os critérios de interligação não se aplicam, de uma forma directa, a este tipo de métodos, surgindo em fases posteriores do processo de desenvolvimento.

Métodos estruturados Os métodos estruturados descrevem a aplicação informática com recurso a três diagramas: ERD, DFD e STD. O diagrama de entidade-relação (ERD – *entity-relationship diagram*) permite descrever os componentes que modelam o mundo real bem como os outros componentes com que se relacionam. As entidades são caracterizadas pelos dados que lhe dão forma, enquanto as relações permitem definir o carácter condicional ou mandatório das mesmas, bem como o número de entidades que participam em cada relação. O diagrama de fluxo de dados (DFD – *data flow diagram*) descreve os processos e a informação que flui entre eles. O diagrama de transição de estados (STD – *state transition diagram*) descreve os estados possíveis do sistema ou componente do sistema bem como os acontecimentos ou mensagens que produzem as transições entre os diferentes estados. Na sua versão original estes diagramas, nomeadamente o ERD, eram muito limitados permitindo apenas representar formas de agregação e particularização [You89, FK92]. As técnicas de cópia e modificação podem ser realizadas sem que para tal exista necessidade de apoio especial. No entanto, versões mais recentes permitem inclusivamente representar formas simplificadas de herança que seria expectável surgirem apenas em modelos orientados para objectos.

Métodos orientados para objectos Os métodos de análise e desenho orientados para objectos resultam da evolução natural dos anteriores, mas agora enriquecidos com notações para representar todos os mecanismos acima descritos [FK92, Weg92, ABC$^+$91, Man95]. Estes métodos permitem, através de diagramas equivalentes aos utilizados nos modelos estruturados, descrever a estrutura e o comportamento dos componentes. No entanto, o conceito de objecto que lhes está subjacente permite descrever de uma forma mais intuitiva, e simultaneamente rigorosa, o comportamento do componente [Weg92]. A tecnologia dos componentes tem sido descrita como uma extensão da tecnologia dos objectos a uma escala maior, em que o componente pode ser visto como uma colecção de objectos fortemente relacionados e com uma estrutura e comportamento bem de-

finido. Os vários métodos de análise e desenho orientados para objectos apresentam diferenças pouco significativas entre si, apresentando-se alguns mais próximos dos métodos estruturados enquanto outros apresentam uma alteração mais radical [FK92]. O aparecimento da Linguagem de Modelação Unificada (**UML** – *Unified Modeling Language*), designada como uma linguagem de modelação orientada para objectos de terceira geração, adapta e extende os trabalhos mais importantes da área até então e inclui contribuições e sugestões de muitos outros [Boo94, JCJO92, Rum91]. A **UML** oferece um conjunto mais rico e rigoroso de notações que permite captar diversas perspectivas das aplicações a desenvolver, modelando tão bem sistemas de tempo real como sistemas cliente-servidor ou sistemas de informação clássicos [BJR96, FS98, BJR97b, BJR99]. Um conjunto de oito diagramas permite descrever o sistema sob diversos pontos de vista, resultando numa descrição rica e completa do componente. A importância e quantidade de informação, disponível na fase de análise e modelação, varia de diagrama para diagrama de acordo com a área de inserção da aplicação em causa [Kru98]. Finalmente, a **UML** é, ao contrário dos restantes métodos de modelação, uma linguagem de especificação que pode ser processada para efectuar verificações sintácticas ou mesmo semânticas, permitindo inclusivamente gerar partes do código fonte de fases posteriores de desenvolvimento. Esta característica é extremamente importante do ponto de vista da especialização e da integração, e também no processo de selecção, pois permite efectuar comparações mais rigorosas que recorrendo apenas a descrições gráficas [CM98].

Métodos formais Os métodos formais apresentam a vantagem de descrever de uma forma matemática, e consequentemente rigorosa, o comportamento funcional do componente. No entanto, a sua utilização a modelar sistemas tem várias limitações, algumas das quais se agravam em situações de reutilização e de especialização em particular [Kem90]. Por outro lado, os métodos formais não permitem descrever características não funcionais dos sistemas nem são garantia de correcção da perspectiva funcional: algumas afirmações não conseguem ser provadas e podem-se cometer erros nas demonstrações [Hal90]. Do ponto de vista da especialização, os métodos formais apresentam os mesmos problemas encontrados no processo de selecção [Zar96, ZW97]. Ao nível da integração é possível descrever os componentes, não como construções sintácticas utili-

zadas para agrupar declarações e controlar a visibilidade mas, com um comportamento intrínseco, que é visível através de todo o sistema [Edw95, AL95b].

Ambientes de integração distribuídos

Um dos factores importantes para a institucionalização da reutilização e a disseminação de bibliotecas de componentes de software é a normalização e a distribuição. Por normalização entenda-se a definição de formato de dados comum de uma interface uniforme e de um serviço que permita manipular os componentes no repositório. Um formato de dados comum permite representar os componentes em si e a informação a eles associada, tal como documentação e informação de contexto, de uma forma integrada. A utilização de uma interface uniforme permite utilizar o mesmo componente a partir de diversos ambientes e linguagens, independentemente do sistema utilizado para produzir o referido componente. Finalmente, a disponibilização de um serviço de acesso ao repositório permite gerir de uma forma eficiente e integrada os componentes nele existentes, bem como permitir o acesso a ferramentas de desenvolvimento e gestão.

Estes ambientes de integração possuem repositórios de objectos nos quais é possível descrever associações entre os vários elementos de uma forma uniforme [Red97]. Os objectos do repositório podem ser acedidos através de mecanismos de comunicação e ser activados localmente por mecanismos de execução [Har87, Tho98, BGMT88, Obj95]. Estes sistemas permitem impor restrições por forma a controlar as composições aceitáveis entre os vários componentes. Embora pretendam garantir a coerência das composições, as restrições têm-se mostrado limitativas. Sistemas mais recentes permitem gerar adaptadores e fornecem um conjunto de mecanismos que permitem ao utilizador combinar e adaptar quase sem esforço os componentes existentes no repositório [Ham97, LS98].

Linguagens de programação

As linguagens de programação têm sido os meios preferenciais para a construção de componentes. Estas linguagens dividem-se em três grandes grupos, no que diz respeito ao suporte à integração: sem suporte, com suporte sintáctico, com suporte semântico. As linguagens que não suportam pelo menos o conceito de compilação separada

incluem-se no primeiro grupo e não podem ser utilizadas como mecanismos de integração, embora possam ser interligadas com auxílio do sistema operativo. No entanto, as aplicações por elas produzidas podem vir a constituir componentes reutilizáveis através, por exemplo, da execução de sub-processos. Nas linguagens com suporte sintáctico, os componentes são apenas aglomerações de funções e dados sem verdadeiro suporte de interligação por parte da linguagem. As linguagens com suporte semântico, em que o componente é uma entidade com tipo e significado, incluem as linguagens baseadas em objectos: aquelas que suportam o conceito de objecto ou módulo como entidade semântica, podendo suportar, ou não, herança e subtipificação.

Programação estruturada As modernas linguagens de programação evoluiram a partir das suas antecessoras, mas o seu principal objectivo continua a ser o de descrever instruções aos computadores. Estas linguagens não foram desenhadas para ajudar a explicar aos seus utilizadores o significado do software que eles descrevem. Este facto conduziu a que os módulos sejam apenas construções sintácticas, utilizadas para agrupar elementos relacionados do programa, mas sem uma denotação semântica como é o caso do tipo de dados, variável ou função. Embora limitadas no conceito de módulo, que em muitos casos é associado a um ficheiro, estas linguagens têm permitido encapsular de uma forma eficiente a informação neles contida e simultaneamente controlar a visibilidade dos seus elementos [Seb93]. A falta de informação semântica na própria linguagem necessita ser compensada por documentação adicional que permite compreender o comportamento do componente e facilitar a sua especialização e integração [Bas87]. As linguagens interpretadas, por efectuarem a compilação imediatamente antes da execução, permitem construções mais flexíveis que aquelas que, por serem previamente compiladas, têm de efectuar uma análise estática [Kam90, Ous94]. A utilização de bibliotecas dinâmicas e do carregamento dinâmico de componentes permite, contudo, às linguagens compiladas algum grau de liberdade que tem sido aproveitado para facilitar a composição e configuração de aplicações e, inclusivamente, de sistemas operativos.

Programação orientada para objectos As linguagens baseadas em objectos, quer suportem herança quer não, recorrem à construção de objectos que comunicam uns com os outros. Os objectos são dotados de propriedades e comportamento assumindo uma identidade própria que permite a sua especialização, partilha e integração em diferentes situações. Estas linguagens desenvolveram mecanismos de especialização que permitem poder extender o objecto com nova funcionalidade e interface sem alterar o objecto existente nem os restantes objectos que com ele interagem. Por outro lado, a determinação do receptor das mensagens no momento da invocação do mesmo permite a integração e substituição de objectos sem comprometer o funcionamento de toda a aplicação. As linguagens orientadas para objectos têm mostrado um particular cuidado com a reutilização de software pelo que oferecem um bom suporte à especialização e, em menor escala, à integração [Mey87, TFB92, Car95, Cop92].

Programação orientada para componentes Um objecto é, normalmente, uma entidade muito pequena para ser partilhada em larga escala, em especial pelo facto de possuir ligações fortes com um certo número de outros objectos. Assim, um componente inclui um conjunto de objectos, tal como são vistos ao nível das linguagens de programação, com uma interface bem definida. A programação baseada neste paradigma é designada por programação orientada para os componentes e é uma evolução em relação à programação orientada para objectos. Estas linguagens estão especialmente vocacionadas para os problemas de integração em detrimento dos problemas de especialização. Desta forma, embora a herança seja um mecanismo essencial nas linguagens orientadas para objectos como disciplina para construir objectos, ela é considerada inapropriada nas linguagens orientadas para componentes que estão mais preocupadas com os problemas de integração [Ude94]. Numa linguagem orientada para componentes deve ser possível especificar o componente sem o realizar, permitindo ao programador utilizar um componente que ainda está disponível. É óbvio que, no momento da invocação, deverá haver pelo menos um componente que obedeça à especificação. As ligações são, quando comparadas com as linguagens orientadas para objectos, ainda mais dinâmicas, oferecendo maior flexibilidade de integração [Jr.97, SLdP98, SM97]. Desta forma, estas linguagens representam um bom compromisso entre as linguagens orientadas para ob-

jectos e as linguagens de interligação de módulos. Em relação às linguagens orientadas para objectos oferecem menor capacidade de especialização mas maior capacidade de integração, apresentando as características contrárias em relação às linguagens de interligação de módulos.

Linguagens de interligação de módulos

As linguagens de interligação de módulos são restrições às linguagens de programação, uma vez que as linguagens de interligação de módulos oferecem muito pouco ou nenhum suporte para computação genérica. As linguagens de interligação utilizam a funcionalidade disponibilizada pelos componentes que interligam, mas controlam apenas entre componentes produzidos por outras linguagens [GC92, BZ91]. Assim, embora os critérios de especialização não se apliquem a este tipo de linguagens, estas revelam bons desempenhos em relação aos critérios de integração.

Várias variantes têm sido estudadas como mecanismos de interligação. Para isso, torna-se necessário descrever as obrigações de cada componente e as interacções entre os componentes. A especificação das obrigações consiste nas assinaturas das operações que o componente suporta. A utilização de obrigações permite determinar quais as composições possíveis, através da verificação da validade dos contratos de todos os componentes. O componente seleccionado é um dos que verifica as obrigações [HLS97, ALMD96]. Outras aproximações utilizam especificações algébricas que um componente pode verificar. Por exemplo, em **LIL** [Gog86] utiliza-se um conjunto de vistas para descrever o componente, não havendo necessidade para utilizar nomes no processo de verificação, pelo que a ênfase é colocada no comportamento semântico. No **MELD** [KG87], os candidatos são ordenados por forma a determinar automaticamente quais os componentes a activar. No **DRACO** [Nei84, SLdP98] os objectos e as operações são organizadas por domínios. As ligações entre os diferentes domínios representam as diferentes possibilidades de interligação.

Processos sequenciais comunicantes O mecanismo de canais de comunicação ("*pipes*") permite a um conjunto de linguagens, das quais se destacam os interpretadores

de comandos, organizar as ligações entre os diversos processos que constituem uma aplicação. O interpretador de comandos pode, através dos *pipes*, canalizar o resultado de um processo para a entrada de outro; alternativamente pode usar o resultado de um processo para parametrizar outro processo, através dos seus argumentos. Por sua vez, é possível especializar novas aplicações por encapsulamento e agregação, através de ficheiros de comandos ("*shell scripts*"). Também as regras de visibilidade podem ser alteradas em função do directório corrente ou da modificação da lista de directório que tornam um comando acessível. No entanto, não se tratando de uma linguagem de programação genérica não é possível efectuar especializações que exijam o conhecimento do conteúdo dos componentes, como a cópia e modificação ou a herança. Como a linguagem não é tipificada, também não é possível recorrer a mecanismos de sobreposição ("*overloading*") ou polimorfismo, uma vez que estes dependem do tipo dos dados. É, contudo, interessante verificar que, não sendo uma linguagem de programação, o interpretador de comandos do **UNIX** é uma linguagem de integração de módulos com capacidade para efectuar integração de componentes com considerável sucesso [Ker84]. A linguagem **NIL** oferece as funcionalidades de um sistema operativo em conjunto com uma linguagem de programação fortemente tipificada [Weg84]. Esta linguagem permite o estabelecimento de ligações dinâmicas entre os componentes, sendo cada componente manipulado como um processo. O maior problema destas linguagens reside no desempenho, uma vez que cada componente é executado por um processo. No entanto, muitas das aplicações que não exijam recursos computacionais elevados podem ser realizadas com facilidade à custa de pequenos programas – componentes – e da linguagem de comandos.

2.4 Síntese

A identificação de boas abstracções é importante no desenvolvimento de software, mas é especialmente importante quando se recorre à reutilização de componentes. A análise de variantes e o consequente nível de parametrização do componente são critérios de abstracção importantes a considerar no desenvolvimento do componente. Estes crité-

rios serão importantes na determinação das área em que o componente é aplicável, condicionando a sua reutilização a situações com requisitos concretos.

O processo de classificação deve representar o componente de uma forma completa por forma a não obter, no processo de selecção, nem componentes que não obedecem aos requisitos, nem deixar de fora componentes que efectivamente obedecem aos referidos requisitos. Uma vez que o processo de selecção se baseia na informação disponibilizada pelo processo de classificação, os modelos de representação analisados referem-se ao conjunto dos dois processos referidos. Os modelos de representação encontram-se divididos em grupos consoante o tipo de representação que utilizam, podendo cada representação ser criada, por classificação e analisada, por selecção, através de algoritmos e técnicas distintas. Os modelos textuais utilizam descrições baseadas em texto livre, escrito em linguagem natural, enquanto os modelos lexicais recorrem a um vocabulário reduzido e pré-definido. Os modelos estruturais representam os componentes através de estruturas mais complexas onde os nomes utilizados não estão limitados a vocábulos pré-definidos. Os restantes modelos, que não se incluem nos casos acima, utilizam métricas ou descrições formais, entre outros, para representar os componentes.

Uma vez que os componentes seleccionados raramente podem ser reutilizados no seu estado original, é necessário recorrer a um conjunto de técnicas de especialização. As técnicas de particularização, encapsulamento e agregação de componentes apenas acrescentam informação, pelo que se podem basear em descrições incompletas. Por outro lado, técnicas de cópia e modificação ou herança exigem uma descrição completa do componente, pelo que a sua utilização fica limitada a um conjunto mais reduzido de componentes. Depois de especializados para a aplicação concreta a desenvolver, os componentes necessitam ser integrados para darem forma à aplicação em questão. O processo de integração pode resumir-se à edição estática de ligações, como no caso da compilação separada, ou utilizar técnicas mais flexíveis e dinâmicas. A resolução dinâmica de nomes, utilizada por exemplo nas linguagens orientadas para objectos, permite que a edição de ligações tratada caso a caso dependendo das características dos componentes e dos respectivos objectos. O carregamento dinâmico de componentes permite que os componentes possam ser construídos e substituídos sem ter de reiniciar a execução da aplicação.

Capítulo 3

Modelo de identificadores

Num processo de desenvolvimento inteiramente automático seria possível, dados um conjunto de requisitos, funcionais e não funcionais, gerar a aplicação final. Da mesma forma, um processo de reutilização inteiramente automático produziria a aplicação final a partir dos requisitos, identificando os componentes necessários, procurando-os na biblioteca e integrando-os na aplicação final, fazendo as especializações que fossem necessárias. Embora as tarefas mecanizáveis devam ser automatizadas, a intervenção humana é necessária. Os sistemas de apoio à reutilização são projectados como sequências de ferramentas que resolvem problemas específicos. Embora seja muitas vezes relegada para segundo plano, a intervenção humana é essencial quer na intervenção criativa quer por servir para ligar as partes automatizadas do processo de reutilização. É necessário pensar numa forma de integrar a intervenção humana de uma maneira natural, permitindo ao mesmo tempo a utilização de ferramentas que automatizem as tarefas mecanizáveis. Para tal, deve-se escolher como entidades principais do modelo de reutilização elementos facilmente assimiláveis pelos humanos e, ao mesmo tempo, utilizáveis pelas ferramentas do sistema. Por exemplo, a linguagem natural é de difícil processamento por parte das ferramentas embora intuitiva para os utilizadores. As métricas difusas, por outro lado, são acessíveis, quer aos utilizadores quer às ferramentas, mas não disponibilizam informação suficiente para garantir uma boa precisão de selecção e uma boa recuperação de componentes para reutilização. É necessário identificar um modelo uniforme para todo o processo de reutilização e que seja igualmente assi-

milável por utilizadores e ferramentas. Neste capítulo descreve-se um modelo baseado numa organização hierárquica de identificadores. Os identificadores em questão são extraídos das descrições efectuadas com base nas linguagens utilizadas para descrever o próprio componente durante as fases do processo de desenvolvimento do componente.

3.1 Utilização de nomes

Os nomes representam o primeiro contacto com a entidade. São abstracções pois podem ser utilizadas, numa primeira abordagem, em vez da própria entidade. Um nome tem existência só por si, podendo não estar associado a uma entidade.

Definição 3.1 (nome) *Um* nome *é uma palavra ou título que serve para designar uma entidade.*

A utilização de nomes em computadores e informática tem sido, de uma forma geral, descuidada. Os nomes são introduzidos onde e quando são necessários e constituídos do comportamento estritamente indispensável para as funções que lhes são atribuídas. Tal facto é ainda mais evidente em projectos de software, devido à sua natureza exploratória. Neste caso, a criação de inúmeras versões, distribuições e testes para as mais diversas arquitecturas, processadores e ambientes, conduz a uma enorme proliferação de configurações intermédias e finais. Estas configurações diferem, nos seus componentes e forma de manipulação, umas das outras. No entanto, deve ser fácil determinar qual a configuração em questão, bem como as suas características. Esta facilidade está necessariamente ligada à expressividade dos identificadores que descrevem as características da configuração. Para que a expressividade seja total não devem ser impostas, *a priori*, condições que limitem significativamente a descrição e utilização dos identificadores.

3.1.1 Aplicações dos nomes

Nos primeiros computadores as entidades – variáveis, funções, *etc* – de um programa eram acedidos pela posição que ocupavam na memória. Os processadores de então,

tal como os de hoje, só manipulam posições de memória. Tal obrigava o programador a saber qual a entidade que ocupava determinada posição de memória, bem como as posições de memória livres. O facto de recair sobre o utilizador a gestão da memória representava uma limitação em relação ao crescimento do programas. Isto é devido, quer à grande quantidade de informação que este processo envolvia, quer pela frequência com que esta informação era alterada.

As linguagens de programação, e em outra escala os sistemas de ficheiros, pretendem automatizar a gestão de memória. Assim, permitem e, por vezes, obrigam a associar a cada recurso um identificador. Desta forma o utilizador pode referir as entidades através de nomes sugestivos. Estes nomes facilitam a utilização e o desenvolvimento de aplicações informáticas. Além disso, a utilização de nomes introduz um grau de abstracção que, ao utilizar os nomes em vez das próprias entidades, permite aos projectistas construir sistemas muito maiores do que era até então possível.

3.1.2 Determinação dos nomes

A principal função do nome é o de permitir referir uma certa entidade. Isto não implica que o nome deva ser único para uma dada entidade. Quer dizer que o nome não é uma forma de assinatura e que, por comparação de nomes, não é possível garantir se dois nomes referem a mesma entidade ou não. Se fosse esse o caso, estaríamos limitados na escolha dos nomes possíveis. Da mesma forma, não seria possível dispor de sinónimos, o que é desejável e muitas vezes importante. Finalmente, seria difícil realizar mecanismos de replicação e recombinação, indispensáveis nos sistemas distribuídos de hoje. Assim, nomes não devem ser assinaturas, isto é, não devem servir para identificar e diferenciar as entidades que referem. As entidades podem conter, quando necessário, assinaturas como forma de permitir comparar e diferenciar entidades sem exigir uma análise exaustiva do seu conteúdo.

Normalmente os nomes dão pistas sobre as entidades que referem. Por exemplo, o nome João sugere uma pessoa do sexo masculino e o nome Maria alguém do sexo feminino. No entanto, combinações destes nomes podem dar outras pistas. Por exemplo, João Maria sugere uma pessoa do sexo masculino apesar da presença do nome Maria, e

Maria João o inverso. A utilização de apelidos, por outro lado, não oferece pistas quanto ao sexo das entidades que designa, embora classifique a entidade segundo a família a que pertence.

Os nomes só representam abstracções úteis se derem pistas sobre as entidades. Por exemplo, referir um livro pelo seu ISBN, em vez do seu título, é uma abstracção mas de pouca utilidade e até um pouco confusa para o ser humano. No entanto, o ISBN é uma assinatura, permitindo a identificação do livro sem necessidade de enumerar o seu título, autor, edição, *etc*.

Nomes puros e assinaturas podem ser úteis como estruturas internas dos computadores mas de pouco servem aos humanos que as usam. Além de servirem como base de comparação de entidades, o que muitas vezes pode ser conseguido por inspecção das próprias entidades, introduzem uma enorme complexidade por obrigarem a serem únicos no universo global.

Chamar a variáveis *aux*, a programas *make*, ou a máquinas *alfa* é uma solução geralmente adoptada e até defendida [Lib89]. No entanto, estes nomes não dão qualquer indicação sobre o objecto que designam. Como ponto de partida podíamos argumentar que poderíamos substituí-los por números.

A utilização de nomes como *aux*, *make* ou *alfa* liberta-nos de manipular números. A manipulação directa de números por seres humanos é muito sujeita a erros [McD94]. Uma solução muito utilizada consiste na introdução de dígitos de verificação ("*check digits*"). O número introduzido contém mais dígitos que o número pretendido, mas nem todas as combinações estão correctas. Assim, se o utilizador se enganar a introduzir um dos dígitos, será possível verificar de uma forma simples e rápida se esse número é um dos possíveis ou não. Claro que, quanto menor for a quantidade de números válidos face às possibilidades, maior é a tolerância aos erros.

A utilização de nomes como *aux*, *make* ou *alfa* utiliza os mesmos princípios dos dígitos de verificação, mas recorre a composições de sílabas, fonemas ou palavras para representar o universo de soluções possíveis. A principal vantagem desta solução, face aos dígitos de verificação, é o facto de ser mais fácil aos humanos decorar e reproduzir tais sequências [McD94].

A escolha de nomes directamente acessíveis pelo utilizador deve, portanto, reflectir a sua funcionalidade. Desta forma é possível utilizar, sem confusão ou ambiguidade, o nome em vez da entidade e, consequentemente, abstrair de pormenores da mesma. Consegue-se, através da redução da informação que o utilizador é obrigado a manter, construir sistemas melhores e mais completos.

Embora seja possível deduzir do acima exposto, convém referir que os identificadores não devem conter informação explícita sobre a localização ou suporte. Estes e outros atributos, que não descrevam a sua função, apenas limitam a validade dos nomes podendo induzir o utilizador em erro. Tal é o caso da informação de localização que fica desactualizada cada vez que a entidade é movida. De igual forma os nomes não devem reflectir o suporte, seja ele *hardware* ou *software*, pois este é frequentemente actualizado [Lib89]. No entanto, há excepções e pode ser necessário inserir nos identificadores elementos informativos da localização da entidade. Logo, o modelo de representação dos identificadores deve aceitar excepções, que deverão ser devidamente justificadas.

3.1.3 Características dos nomes

O estudo da utilização dos nomes no dia a dia permite identificar um conjunto de características particularmente importantes na utilização de nomes em sistemas de informação. Desta forma torna-se possível aproximar, em sistemas de informação, mecanismos que reproduzam essas características. Destas características sete surgem como especialmente relevantes para o presente trabalho: individualidade, multiplicidade, mobilidade, localidade, significado, comunicação e partilha.

Individualidade Por *individualidade* entenda-se que os nomes são utilizados de forma diferente por diferentes pessoas. Utilizadores diferentes podem utilizar nomes diferentes para referir a mesma entidade, ou utilizar o mesmo nome para designar entidades diferentes. Este direito à diversidade é extremamente importante pois a necessidade de partilhar a informação aumenta, quando os sistemas de informação e número de utilizadores crescem. Esta partilha deve, contudo, permitir que se possa exprimir essa informação de uma forma compreensível e intuitiva.

Os sistemas existentes utilizam diversos mecanismos que permitem, de uma forma mais ou menos completa, a particularização da informação disponível. Formas comuns incluem a utilização de diversos apontadores para a mesma variável, em linguagens de programação, ou a utilização de sinónimos (por exemplo, directivas do tipo `define`, `parameter` ou `const`), quer pela própria linguagem quer por pré-processadores. Ao nível dos sistemas operativos podemos referir os `links` ou `search paths`. Inclusive as redes de computadores utilizam sinónimos, através de ficheiros de configuração ou serviços de nomes distribuídos. Uma situação onde a sua utilização é extensiva é a `world wide web`, onde o conceito de `link` do sistema operativo é extendido e alargado quer em capacidades disponíveis quer na forma de interpretação dos mesmos.

Multiplicidade Por *multiplicidade* entenda-se que uma entidade pode ter mais de um nome. A multiplicidade surge com a sequência lógica da individualidade. Embora os mecanismos sejam os mesmos e normalmente não imponham restrições ao seu uso, a multiplicidade permite que a mesma pessoa possa dar nomes diferentes para a mesma entidade. Nomes diferentes podem ser utilizados em situações diversas. Como vimos atrás, uma entidade pode ser atributo de uma segunda entidade mas ser também, por si só, uma entidade autónoma. Podemos inclusivamente permitir que uma entidade possa ser atributo ou componente de várias entidades distintas.

Mobilidade Por *mobilidade* entenda-se o facto de os nomes mudarem com o tempo. O mesmo nome pode referir entidades diferentes em momentos diferentes. Esta característica é importante pois garante que o conceito que o nome representa se mantém constante quando as entidades evoluem. De igual forma é possível alterar a entidade que o nome refere quando surge uma nova entidade que melhor representa o conceito. Este último caso, muito comum no projecto de software, é normalmente abordado através do controlo de versões e configurações. A relevância desta característica tem origem na dificuldade do ser humano em memorizar muitos nomes e em especial em acompanhar a sua evolução simultânea.

Localidade O significado do nome depende do contexto em que está inserido. O critério da legibilidade implica a simplicidade da forma de indicar qual o contexto a que o nome se refere. De igual forma, sempre que o tema se altera, deve ser fácil indicar qual o novo contexto onde os nomes podem ser resolvidos. Na maioria dos sistemas operativos, o directório corrente oferece um contexto local, que pode ser facilmente alterado sempre que necessário. Nas linguagens de programação, a visibilidade das variáveis é geralmente controlada por blocos ou sub-rotinas, excepção para as linguagems onde o contexto é explícito [Car95, Bro81] ou onde o contexto pode ser codificado no nome [Str91].

Significado O significado deve reflectir a informação, que o utilizador procura, e que a entidade provavelmente conterá. Embora nomes designem entidades eles possuem um *significado* próprio. É este significado que distingue os nomes dos endereços, ou mesmo das referências. A ausência de significado é precisamente o que caracteriza os nomes puros, que não são normalmente úteis para o utilizador. Os nomes atribuídos pelos seres humanos, que contêm informação sobre as entidades que referem, são títulos dessas entidades. O significado desta informação contida no título ajuda os utilizadores a memorizar e relembrar as entidades. São também importantes quando se pretende procurar determinada entidade, embora possíveis ambiguidades só possam ser esclarecidas por análise de atributos ou mesmo por inspecção do conteúdo da entidade. Em sistemas onde os nomes são constituídos por vários componentes, cada componente deve ter um significado que seja reconhecível por potenciais utilizadores da entidade.

Comunicação Nomes são utilizados principalmente como forma de *comunicação* entre utilizadores e entre programas. Nomes funcionam como abreviaturas para referir entidades. Para que haja comunicação, é necessário que os intervenientes sejam capazes de identificar as entidades a que os nomes dizem respeito. Este facto contradiz a individualidade acima descrita, uma vez que identificação de uma mesma entidade obriga a que o nome utilizado seja o mesmo. Assim, um sistema que suporte ambas as características deverá suportar sinónimos. Só através de nomes alternativos é possível

fazer coexistir nomes padrão, para comunicação, com nomes particulares que garantam a individualidade e diversidade expressiva.

Partilha Por *partilha* entenda-se não a partilha das entidades mas a partilha dos nomes em si. Por partilha de nomes entenda-se o facto de os nomes por nós utilizados já terem sido usados por outros. Não é comum a utilização de nomes originais para a designação de entidades. Mesmo a escolha de palavras-chaves secretas ("*password*") raramente é original, o que se revela um ponto fraco de muitos sistemas de segurança. A utilização de nomes previamente atribuídos por outros é um dos métodos pelos quais os utilizadores identificam as entidades relevantes. Sistemas de nomes devem permitir aos utilizadores incluir no seu espaço local de nomes, nomes já utilizados por outros. Os nomes podem ser incluídos individualmente ou em grupos relacionados. A aceitação de um nome e o seu grau de partilha resultam do número de vezes que é utilizado pelos diversos utilizadores de um sistema de nomes, ou seja, é função do número de vezes que ocorre. Os utilizadores devem ter a liberdade de, caso considerem pouco expressivos os nomes atribuídos a uma entidade, criar nomes alternativos.

3.2 Função dos nomes

Um nome é uma entidade linguística que permite distinguir uma entidade particular de grupo de entidades [vdL91]. De acordo com o nome escolhido, o utilizador associa uma determinada abstracção que lhe permite distinguir essa entidade das restantes, bem como caracterizá-la semanticamente. Os computadores, por outro lado, operam em entidades que são identificadas por nomes, normalmente atribuídos pelos utilizadores. As linguagens utilizam um conjunto de operações para manipular quer as referidas entidades quer os sistemas de nomes que as identificam. A terminologia utilizada pelas linguagens para designar os vários conceitos envolvidos é livre e ambígua, dificultando a compreensão dos mecanismos envolvidos [Bar77].

3.2.1 Nomes e identificadores

Os elementos principais num sistema de nomes são as entidades e os próprios nomes, uma vez que representam, respectivamente, o objectivo e os meios.

Definição 3.2 (entidade) *Uma* entidade *é um conjunto de dados individuais, designados por atributos.*

Muitos sistemas utilizam o termo `objecto`, mas neste trabalho não utilizaremos esse termo que tem conotações especiais nas linguagens de programação. Assim, de uma forma simplista, podemos considerar a entidade como um bloco de zero ou mais bytes, não necessariamente consecutivos, que, embora possa conter referências para outras entidades, é distinta delas.

Por vezes utiliza-se o termo nome para referir um identificador. Neste texto estabelece-se a diferença entre a existência do nome por si só e a sua associação a uma entidade - identificador.

Definição 3.3 (identificador) *Um* identificador *é a informação necessária para identificar uma entidade.*

Considera-se que um nome é promovido a identificador a partir do momento em que é associado a uma entidade. O identificador pode ser atribuído por um utilizador, ou ser gerado internamente pelo sistema informático. O identificador pode ser uma quantidade atómica, palavra ou frase, ou uma quantidade composta que identifique univocamente o objecto. Note-se que pode haver mais de um identificador, atómico ou composto, que identifique um dado objecto.

Os nomes e os identificadores necessitam apenas de referir uma e uma só entidade. Do ponto de vista matemático estão relacionados com as entidades através de aplicações que não necessitam ser injectivas ou sobrejectivas. Isto é, pode haver mais de um nome para a mesma entidade e entidades sem nome, respectivamente.

O conceito de *identificador único*, obriga a que a aplicação dos nomes nas entidades seja sobrejectiva, ou seja, todas as entidades têm nome e cada entidade tem um nome distinto das restantes. Os nomes únicos permitem distinguir as entidades por comparação dos nomes, sem necessidade de ter acesso às entidades por eles referidas.

Definição 3.4 (assinatura) *Uma* assinatura *designa um nome ou sinal que se distingue e demarca dos restantes.*

De um ponto de vista matemático, uma assinatura é uma aplicação injectiva, mas não necessariamente sobrejectiva. Ou seja, a comparação de assinaturas é suficiente para distinguir entidades, mas nem todos os objectos necessitam de possuir uma assinatura.

Definição 3.5 (sinónimo) *Um* sinónimo *é um nome alternativo utilizado para designar a mesma entidade.*

Um sinónimo não pressupõe que nenhum dos nomes tenha qualquer tipo de função que o distinga dos restantes sinónimos, por exemplo, ser o nome principal que controla a acessibilidade da entidade. Se for este o caso, então designaremos os nomes secundários por *aliases*. Frequentemente, estes nomes secundários referem a entidade através dos nomes principais. De um ponto de vista matemático é uma aplicação de um nome em outro nome, ou eventualmente o mesmo. O exemplo mais comum é o caso do *simbolic link* no sistema operativo **UNIX**.

3.2.2 Referências e endereços

Definição 3.6 (associação) *O estabelecimento de uma aplicação entre um nome a uma entidade é designado por* associação.

Definição 3.7 (resolução) *A operação de obtenção da entidade, dado o seu identificador, designa-se por* resolução.

Definição 3.8 (endereço) *O* endereço *é o local físico onde a entidade está guardada.*

Um endereço é, ele próprio, um nome que é entendido pelo sistema. Note-se que nem todos os endereços referem entidades.

Definição 3.9 (referência) *A* referência *permite identificar o local físico onde a entidade está guardada.*

Ou seja, se a entidade se mover (ou se o valor for replicado) então a entidade muda de endereço (ou passa a ter dois endereços possíveis), mas a referência mantém-se. A referência pode também ser entendida como um nome que, contudo, é gerado e existe numa forma conveniente para os computadores. Por vezes, as referências, são chamadas de nomes de sistema por oposição aos nomes dos utilizadores que aqui designamos apenas por nomes. A necessidade deste segundo nível de nomes é essencial para que a utilização dos nomes pelos utilizadores não seja restringida pelas capacidades do sistema de suporte.

3.2.3 Atributos e entidades

Comecemos por analisar a forma como os sistemas actuais usam os nomes e quais as razões por detrás das opções tomadas. O objectivo dos nomes e identificadores é o de referir, de uma forma única ou não, alguma entidade no sistema. Assim, comecemos por compreender essas entidades inferindo sobre a função que desempenham em diversas áreas dos sistemas informáticos.

O ficheiro de acesso sequencial é a forma mais simplificada de entidade sendo uma sequência de 0 ou mais bytes, sem qualquer estrutura interna. Normalmente, o ficheiro tem ainda associado a ele alguma informação auxiliar como a data da última modificação ou o dono do mesmo. Esta informação auxiliar, que pode ser mais ou menos extensa dependendo do sistema, permite caracterizar a entidade. Como de um ponto de vista conceptual esta informação auxiliar não faz parte da entidade, chamar-lhe-emos atributos da entidade.

Nas linguagens de programação imperativas as entidades são, numa primeira aproximação, variáveis e funções. Numa perspectiva muito simplificada a entidade pode ser

entendida como uma sequência de 0 ou mais bytes, tal como no caso dos ficheiros. Neste caso, contudo, não só o tipo de atributos associados às entidades varia bastante de linguagem para linguagem como a sua descrição. Por exemplo, uma variável bem definida no momento da compilação não necessita de ter associado o seu tamanho, já que o código gerado para a manipular tem isso em conta. Por outro lado, se a variável é criada dinamicamente então a sua dimensão necessita ser guardada, no mínimo, para a sua correcta eliminação. Também o tipo da variável, que define as operações a que esta pode ser sujeita, pode estar implícita no código gerado. No entanto, em linguagens em que o tipo não é conhecido à partida, como nos interpretadores, ou em que existe subtipificação, por exemplo, por herança, é necessário associar essa informação à variável.

A ausência de atributos associados às entidade permite que eles sejam incorrectamente utilizados. Por exemplo, a informação de símbolos de um ficheiro objecto inclui apenas uma descrição muito sumária das variáveis e funções que contém. Assim, é possível, ao ligar vários ficheiros objecto, produzir um resultado incorrecto. Este é muitas vezes o caso, quando não existem ficheiros de declaração que permitam ao compilador detectar, previamente, possíveis inconsistências. Uma solução consiste em codificar nos próprios nomes esses atributos. Em **C++**, por exemplo, o nome de uma função inclui, além do nome dado pelo programador, a classe a que pertence, e o número e tipo dos argumentos. Esta solução produz identificadores muito longos e obriga à alteração de várias aplicações que devem decompor o identificador nos seus componentes: nome da função, classe a que pertence, número e tipo dos argumentos. Afinal este identificador composto não é mais que uma assinatura da função e que permite identificá-la.

3.2.4 Referências e valores

Os nomes das entidades são, geralmente, representadas por sequências de caracteres assimiláveis pelos utilizadores. No entanto, as máquinas necessitam, também elas, de sequências de bytes que representem as entidades. Contudo, as sequências facilmente assimiláveis pelos seres humanos não são manipuláveis de uma forma eficiente pelas máquinas. Torna-se necessário criar representações para as máquinas que designem a

localização das entidades, ou seja, referências. Evitamos o termo `nome puro` [Mul89] para designar as referências, uma vez que um nome puro não traz consigo qualquer tipo de associação. Uma referência, pelo contrário, pode conter associações temporais ou de localização.

As referências representam um nível intermédio entre o nome, reconhecido pelo utilizador, e a entidade, representada pelo seu valor. Do ponto de vista conceptual uma referência é também um nome, uma vez que designa uma entidade. As entidades necessitam, para serem referidas pelas máquinas, de uma referência; e as referências necessitam por sua vez, para serem referidas pelo utilizador, de um nome. Assim, podemos ter entidades sem nome, se forem apenas referidas pelas máquinas, ou seja, indirectamente pelos utilizadores. No entanto, se uma entidade não possui referência não pode ser acedida pelo que, do ponto de vista do sistema, não existe.

As referências são, frequentemente, manipuladas de uma forma automática e transparente pelo sistema. Em muito sistemas, inclusivamente, o acesso a referências é negado aos utilizadores, ou pode ser apenas visualizado. Veremos que a manipulação de referências é um ponto fulcral, na medida em que influencia aquilo que pode ser referido pelo utilizador.

Convém salientar que, embora em algumas situações se utilizem endereços como referências, tal simplificação é contraproducente. Nomeadamente, as referências servem, como *referência*, para a entidade, permanecendo constantes quando esta se move ou é replicada. Um exemplo comum consiste em reservar memória dinamicamente, numa linguagem de programação como **C**, e usar o endereço devolvido como referência. Desta forma, o redimensionamento da entidade pode obrigar esta a mover-se para um novo endereço, invalidando a referência. Por outro lado, num sistema de ficheiros, tipo **UNIX**, o ficheiro pode crescer ou encolher sem que a sua referência – o número do `i-node` – sofra alteração.

3.3 Estruturas de nomes

Definição 3.10 (espaço de nomes) *O* espaço de nomes *representa o conjunto das associações entre nomes e entidades.*

Definição 3.11 (nome plano) *O* nome plano*, num espaço de nomes, é um nome sem estrutura mas que identifica univocamente uma entidade desse espaço.*

Definição 3.12 (nome composto) *O* nome composto *é constituído por uma sequência de nomes planos, designados por* componentes*, e que identifica uma entidade do espaço de nomes.*

Definição 3.13 (caminho) *Um* caminho *("*`pathname`*") é um nome constituído por uma sequência de um ou mais nomes planos.*

A diferença entre nome composto e caminho reside no facto de um caminho poder não identificar nada, caso não coincida com um nome composto.

Definição 3.14 (separador) *Os componentes de um nome composto ou de um caminho são separados entre si por um carácter especial designado por* separador*.*

Definição 3.15 (directório) *Um* directório *é uma entidade que associa um conjunto de nomes, pertencentes a um espaço de nomes, a entidades.*

A resolução de nomes num espaço de nomes é modelado por uma rede acíclica de nomes e um ponto de origem. A origem do espaço de nomes é designado por *raiz*. O *contexto* é o encadeamento ou composição de nomes que cria o ambiente onde um nome pode ser univocamente resolvido.

3.3.1 Propriedades dos sistemas

A dimensão dos sistemas de nomes tem vindo a aumentar. A forma como os referidos sistemas se comportam perante o aumento de entidades ao seu cuidado é uma medida importante da sua utilidade. O aumento da dimensão dos sistemas manifesta-se principalmente do ponto de vista numérico, geográfico e administrativo [New92].

Por escalamento numérico entenda-se o aumento do número de entidades a referir, sejam elas variáveis, rotinas ou módulos de um programa, ficheiros, dispositivos ou utilizadores de um sistema operativo. O número de entidades a que um utilizador necessita de ter acesso para desempenhar determinada tarefa tem aumentado significativamente, no entanto este crescimento não é tão grande como o número total de entidades. Assim, o número de entidades existentes, mas sem interesse para cada utilizador individualmente, tem aumentado ainda mais. Este facto obriga a mecanismos de navegação e particularização mais eficazes. A `world wide web`, pelas suas características, tem tido um papel importante na eficácia como os utilizadores localizam e obtêm a informação desejada. No entanto, a sua experiência tem sido pouco absorvida em outras áreas dos sistemas de informação, talvez por ser um sistema ainda recente.

A segunda dimensão é geográfica. As entidades são frequentemente armazenadas na proximidade de quem mais as utiliza, essencialmente por razões de desempenho. Este facto conduz a que informação relacionada seja, por vezes, armazenada separadamente em locais diversos. Daqui advém a necessidade de procurar referir ou manipular de um só local informação dispersa, através dos seus nomes. Por outro lado, copiar sistematicamente informação, que pode ser de interesse, mostra-se proibitivo quer em termos de espaço físico local, quer de largura de banda de comunicação. Assim, é necessário dispor não só de mecanismos quer de busca, quer da possibilidade de representação local de informação guardada remotamente. Também aqui a `world wide web` apresenta mecanismos e experiências interessantes embora com o objectivo diverso em vários pontos do projecto de sistemas de informação.

A terceira dimensão é administrativa e refere-se ao controlo e garantia de consistência de um todo que se pode distribuir quer geograficamente, quer por organizações administrativas distintas. O controlo baseia-se na atribuição de direitos de acesso por

parte de organizações. Assim, numa primeira aproximação, é necessário saber qual a organização que controla a entidade. As técnicas existentes resultam, normalmente, de técnicas utilizadas em sistemas centralizados. Estas técnicas começam a falhar à medida que os sistemas crescem e aumentam a sua diversidade.

Níveis de globalidade

Cedo se verificou que a introdução de nomes era limitativa, embora permitisse construir sistemas maiores e mais complexos. A limitação surge, essencialmente, do facto de o nome ter de ser único para referir uma única entidade sem ambiguidade. Como garantir tal unicidade num universo grande é uma tarefa muito trabalhosa, surgiram diversas soluções para resolver este problema. Todas estas soluções se baseiam na partição do universo de nomes em espaços mais pequenos, sendo apenas necessário garantir a unicidade dos nomes dentro de cada um destes espaços. Estes espaços de nomes são criados, quase invariavelmente, com base no tipo de entidades.

Nas linguagens de programação surgem espaços de nomes independentes onde se representam as funções, as variáveis globais, as janelas da pilha de execução ("*stack frame*") e os tipos de dados. A maioria das linguagens permite que o mesmo nome possa ser utilizado para um tipo de dados, para uma função e para uma variável sem gerar confusão ou ambiguidade [KR88]. A resolução da ambiguidade é, normalmente, feita à custa da interpretação sintáctica e semântica da linguagem. Por exemplo, em **C** um símbolo - `aa` - pode ser utilizado como variável, função, tipo estruturado e componente de tipo estruturado simultaneamente: `int aa() { struct aa { int aa; } aa; aa.aa = 1; return aa.aa; }` .

Nos sistemas operativos surgem espaços de nomes que representam os computadores ligados em rede, os ficheiros de dados e os periféricos do próprio computador. O sistema operativo **UNIX**, desenvolvido no início dos anos 70, produziu um impacto considerável ao representar quase todas as entidades do sistema como ficheiros. A uniformidade assim obtida permitia reduzir o número de operações disponíveis. Esta capacidade de aplicar o mesmo nome, por exemplo *read*, a vários tipos de entidades,

que a realizam de formas diferentes, representa uma forma de polimorfismo, tal como é entendida nas linguagens de programação [Str91, GR89].

A simples separação do universo de nomes por espaços acaba por conduzir, com aumento do número de entidades envolvidas, a uma sobrepopulação desses mesmos espaços. Têm, pois, surgido nas diversas áreas formas de subdividir os espaços de nomes através da decomposição e composição dos nomes. Os sistemas de ficheiros representam os nomes, decompondo-os numa árvore hierárquica. A entidade passa a ser referida por uma sequência de componentes, normalmente designada por *pathname*. As linguagens de programação criaram dados compostos – estruturas ou registos – que podem conter ligações entre si, podendo construir estruturas compostas complexas. Mais recentemente, as linguagens de programação associaram a estes dados compostos um conjunto de operações para os manipular. Neste último tipo de aproximação, designada por orientada para objectos, as operações são referidas por nomes compostos por nome do objecto e nomes da operação. Este nível de abstracção adicional permite descongestionar o nome das operações que assim podem surgir repetidos em objectos de tipo diferente.

3.3.2 Hierarquias de nomes

Definição 3.16 (hierarquia) *Uma* hierarquia *é uma representação estruturada por camadas, onde o espaço do problema é dividido em níveis ordenados e estratificados.*

Uma organização hierárquica permite tratar diferentes pormenores do problema a níveis de granularidade diferentes. Numa hierarquia os pormenores não são eliminados; em vez disso, os pormenores são catalogados e colocados a níveis de granularidade diferentes. Desta forma, estão sempre acessíveis quando são necessários. Assim, é possível ter um controlo sobre o grau de abstracção pretendido: generalização corresponde a aproximar da raiz e, especialização corresponde a afastar da raiz.

Do ponto de vista humano as hierarquias permitem representar a localidade, permitindo que um mesmo nome possa ser utilizado em dois contextos diferentes sem que existam conflitos. A diferenciação é feita através do contexto, ou seja, do caminho seguido

desde a origem até ao ponto em questão. Por esta razão as hierarquias são utilizadas nas linguagens de programação para controlar a visibilidade das variáveis ou funções, para descrever relações de herança entre classes, ou para representar a estrutura sintáctica dos blocos de código. O próprio código fonte é, frequentemente, organizado hierarquicamente, onde os directórios correspondem aos sub-sistemas, os sub-directórios correspondem aos módulos e, contidos nesses sub-directórios, encontram-se os ficheiros que constituem os módulos. A hierarquia pode conter muitos níveis ou muitos elementos por nível, dependendo da forma como é estruturada.

A navegação de uma hierarquia pode ser efectuada com recurso a duas dimensões – largura e profundidade – mantendo a raiz como ponto de referência. Dada a limitada capacidade de assimilação do ser humano, um número limitado de graus de liberdade facilita as decisões. Assim, em cada nível, o utilizador dispõe de um conjunto limitado de opções à escolha, à semelhança da condução de veículos onde a localização corrente mais um conjunto de indicadores de direcções possíveis limitam as decisões. Da mesma forma, os indicadores utilizados devem ser semanticamente ricos por forma a permitir uma decisão esclarecida. Enquanto nos sistemas de ficheiros apenas o nome do sub-directório fornece indicações sobre a opção tomada no caso de ser escolhido, os modelos baseados em hipertexto recorrem a figuras ou frases completas como forma de enriquecer a semântica dos indicadores. Para mais, qualquer decisão, que se verifique incorrecta, pode ser facilmente anulada através da navegação pelo percurso inverso até à raiz [BE96, WCG92].

3.4 Separação de responsabilidades

Começamos por introduzir qual a função dos nomes nos projectos de software e nos sistemas resultantes desses projectos. Vimos, de uma forma geral, como são utilizados bem como a sua escolha pode influenciar na abstracção das entidades que designam. Procuramos agora definir objectivos genéricos para o seu estudo pormenorizado.

Para melhor compreendermos qual a importância e influência dos nomes no projecto de sistemas informáticos bem como da sua utilização, necessitamos definir uma abor-

dagem sistemática. Como os nomes são utilizados em quase todos os tipos de projectos de software, aplicaremos essa abordagem às mais significativas áreas de aplicação do software.

A primeira fase na análise dos nomes em projectos de software consiste na identificação dos vários componentes necessários ao seu funcionamento. Para tal, torna-se necessário compreender qual a função desempenhada por cada componente do sistema de nomes. Assim, deve ser possível descrever de uma forma simples e concisa a sua função no referido sistema. Uma vez identificados os componentes e as suas funções isoladamente, torna-se necessário descrever as suas interacções, isto é, a forma como colaboram entre si por forma a atingir o objectivo pretendido – identificação de entidades por nomes.

Plenitude: Como primeira validação torna-se necessário verificar em que medida os componentes identificados e suas interacções se encontram presentes nos vários sistemas. Deve também ser analisado em que grau a presença ou ausência de alguns destes componentes compromete as características dos sistemas em observação.

Simplicidade do modelo: Desta observação deve ser possível isolar um modelo simples e minimalista. Pretende-se que o modelo a obter integre apenas os componentes essenciais ao desempenho genérico esperado do sistema de nomes. No entanto, deverá ser possível, por composição dos componentes ou definição de mecanismos sobre esses componentes, obter a funcionalidade da maioria dos sistemas observados. Só assim, poderá efectuar-se com sucesso a substituição dos sistemas de nomes sem perda de funcionalidade.

Eficiência: Finalmente, pretende-se concluir em que medida o modelo obtido consegue melhorar o comportamento dos sistemas onde é integrado. Uma vez que o modelo aplicado nas diversas áreas é o mesmo, conseguimos pelo menos obter um elevado grau de integração e uniformidade. Este facto é extremamente importante numa altura em que os sistemas tendem a ser cada vez mais interdisciplinares. Assim, podem-se obter ganhos globais importantes, mesmo que os ganhos em situações específicas possam ser nulos ou mesmo negativos.

O objectivo deste trabalho consiste em avaliar as possibilidades de utilizar o sistema de nomes como entidade aglutinadora e integradora das entidades específicas de cada área. Pretende-se centralizar num sistema uniforme de nomes simples a estrutura e organização de diversas áreas da tecnologia da informação. Embora sistemas reais possam recorrer a soluções híbridas, essencialmente por condicionalismos de desempenho, o modelo deverá permitir desenhar sistemas maiores e mais complexos. Daqui se conclui que embora se trate de um empreendimento multidisciplinar, a base e motivação têm origem nos princípios de engenharia de software.

3.4.1 Exemplo ilustrativo: minibanco

Com o objectivo de avaliar a capacidade descritiva de um modelo baseado numa hierarquia de identificadores, utiliza-se um exemplo simples inspirado no sistema bancário, designado por `minibanco`. Neste momento, o exemplo é apenas introduzido, sendo explorado ao longo dos próximos capítulos. Para apresentar o problema e discutir as soluções de uma forma concisa utilizaremos uma versão reduzida do `minibanco`.

Exemplo 3.1 (minibanco) *O* `minibanco` *é constituído por contas bancárias à ordem e a prazo, com informação respeitante ao respectivo cliente. As contas dispõem das operações de depósito, levantamento e informação sobre o saldo disponível e o cliente respectivo. A informação sobre o cliente inclui o nome, a morada e o telefone. As contas à ordem admitem um limite de crédito e não oferecem juro, enquanto as contas a prazo oferecem juro mas não permitem descoberto.*

O desenvolvimento do `minibanco` com recurso a componentes previamente construídos obriga a procurar elementos com as características acima enunciadas. Desta forma, uma descrição de tais componentes incluirá identificadores semelhantes aos utilizados no exemplo 3.1. Será expectável encontrar identificadores como `conta`, `ordem`, `prazo`, `levantar`, `depositar`, `saldo`, `juro`, `crédito`, `cliente`, `nome`, `morada`, *etc.*, podendo apresentar sinónimos dos identificadores ou exibir variações no tempo dos verbos. Estes identificadores podem modelar na linguagem utilizada para os desenvolver funções, variáveis, tipos de dados, classes ou módulos. Por exemplo, um verbo pode

sugerir uma operação, enquanto um substantivo pode denotar um dado. Se for encontrado um número significativo de identificadores semelhantes aos acima indicados, o componente em questão é um bom candidato a integrar o `minibanco`. O problema de determinar se dois identificadores são ou não semelhantes, e qual o grau de semelhança que permite considerar o componente como um bom candidato a reutilizar, será abordado no processo de selecção.

A utilização de hierarquias permite, utilizando o mesmo número de identificadores para descrever o componente, obter uma representação mais rigorosa do mesmo. De acordo com o exemplo 3.1, o identificador `juro` deverá surgir no contexto das contas a `prazo`, enquanto o identificador `crédito` deverá surgir no contexto das contas à `ordem`. Desta forma, será possível rejeitar componentes que ofereçam `juro` em contas à `ordem`, ou de igual forma rejeitar componentes que não apresentem `cédito` para contas à `ordem`. Através da ordenação e estratificação dos identificadores torna-se a representação mais rigorosa e expressiva, melhorando as possibilidades de obter um componente que obedecendo aos requisitos do exemplo 3.1 exija um menor número de adaptações para ser integrado no desenvolvimento do `minibanco`.

3.4.2 Implicações do uso de identificadores

Sendo os identificadores componentes privilegiados na estruturação e organização, por serem a parte visível do projecto e utilização de software devem ter maior relevância e expressividade. O problema é essencialmente um problema de engenharia de software. A eficiência leva a que, frequentemente, se utilizem referências, endereços e mesmo valores, onde do ponto de vista de desenho e abstracção se deveriam utilizar nomes. Para tal deve-se centrar o projecto e os sistemas de informação em nomes atribuídos pelos utilizadores. O mesmo sistema de nomes deve suportar nomes de entidades secundárias que funcionam como atributos das outras, ditas principais. Num sistema deste tipo deve-se, por razões de eficiência, recorrer à resolução de nomes previamente ou na primeira referência. Ou seja, em vez de ser o utilizador a fazer a optimização da conversão de nomes em referências ou valores, esta conversão deve ser deixada a ferramentas e ambientes que a realizam como um mecanismo de tampão ("*caching*").

Do parágrafo anterior conclui-se que os nomes do utilizador são utilizados para nomear tudo a que este tem acesso. As entidades constituídas por estruturas complexas resultam de processos de compilação dos nomes utilizados. Assim, teremos um sistema de nomes uniforme que refere qualquer tipo de entidade. Este facto exige primeiro uma forma de identificar univocamente cada objecto e segundo uma forma de separar claramente os nomes do utilizador dos restantes componentes do sistema. No primeiro caso, estuda-se um tipo de referência híbrido entre a concatenação hierárquica e os mecanismos de *proxis* que permite referir qualquer entidade num sistema global sem as principais desvantagens de ambas as aproximações. No segundo caso, estabelece-se uma clara separação entre os espaços de nomes e os espaços de entidades; por exemplo, operações de `renomear` ("*rename*") e `mover` ("*move*") actuam separadamente em nomes e entidades, respectivamente, permitindo uma maior uniformidade na utilização dos nomes, nomeadamente na interligação de sistemas de ficheiros e no suporte à persistência em linguagens de programação e bases de dados.

Como o objectivo se centra na utilização dos nomes dos utilizadores para a representação dos sistemas e da sua estrutura, é necessário dispor de mecanismos mais flexíveis e poderosos. O primeiro consiste em assumir que os directórios, por exemplo num sistema de ficheiros, não são entidades mas sim elementos internos e invisíveis de uma estrutura mais complexa que são os espaços de nomes. Esta aproximação permite que se associe informação, não apenas às folhas da árvore, mas também aos nós intermédios da árvore de nomes. Este facto é importante do ponto de vista da expressividade do sistema pois permite melhor descrever conceitos base da engenharia de software como a composição, decomposição, continuidade e abstracção. Do ponto de vista de protecção das entidades implica que deixa de ser possível proteger individualmente directórios ou níveis da hierarquia de nomes. No entanto, de um ponto de vista prático, é comum ter uma sub-árvore totalmente constituída por ficheiros pertencentes ao mesmo utilizador e com a mesma protecção. Em bases de dados relacionais, e não só, o panorama é idêntico mas a entidade é a tabela e não o ficheiro. No que diz respeito às linguagens de programação nem existe actualmente mecanismos de protecção sem recurso a operações de `abrir` ("*open*") explícitas. Exceptuam-se algumas linguagens orientadas para objectos em que a protecção é utilizada como forma de disciplina de programação e não

de controlo de acesso. Caso se pretenda proteger uma entidade individualmente, será necessário criar um espaço de nomes para a conter; no entanto, tais situações são raras.

Com o aumento da dimensão da hierarquia, quer em largura quer em profundidade, torna-se mais difícil a busca e acesso aos elementos que a constituem. Para simplificar a sua utilização e permitir a manipulação de hierarquias de maiores dimensões é necessário dispor de mecanismos de busca poderosos. Estes mecanismos devem ser genéricos e flexíveis para poderem permitir buscas com base em critérios que não foram antecipados. A utilização de uma linguagem de composição facilitará a combinação desses mecanismos por forma a obter a informação armazenada na hierarquia. Os mecanismos de busca deverão ter por base as caraterísticas dos nomes e da sua organização hierárquica expostos em 3.1 e 3.3.

3.4.3 Decomposição da complexidade

A decomposição da complexidade, também designado por "*dividir para conquistar*", é o problema crucial na criação de uma abstracção que permita construir um modelo para um sistema de reutilização de componentes. O objectivo é organizar o sistema de tal forma que permita aos utilizadores compreender os componentes com um esforço que seja proporcional à sua dimensão. Existem três formas de decompor a complexidade para atingir o objectivo: abstracção, partição e projecção.

A utilização da abstracção permite suprimir pormenores e concentrar nas propriedades essenciais. Assim, ao referir uma abstracção esta pode representar vários objectos, sem estar associado a nenhum desses objectos em particular. A utilização de abstracções produz hierarquias de uma forma natural, permitindo a elaboração de pormenores cada vez maiores, oferecendo um controlo intelectual sobre o processo de representação usado.

A partição designa a representação de um todo pela soma das suas partes. Desta forma, é possível concentrar separadamente em cada um dos sub-sistemas. A partição torna um sistema modular. Notar que, se cada sub-sistema possuir uma abstracção hierárquica, então este fica decomposto vertical e horizontalmente.

A terceira forma de decomposição – projecção – permite compreender o sistema a partir de diferentes perspectivas. A projecção de um sistema representa o sistema na íntegra, mas apenas no que diz respeito a um sub-conjunto das suas propriedades. Uma analogia de fácil assimilação consiste na representação a duas dimensões de uma figura tri-dimensional.

Identificadores hierárquicos

Os identificadores representam a camada intermédia entre o homem e o computador. A utilização de identificadores é desnecessária, de um ponto de vista estritamente computacional, uma vez que os computadores manipulam apenas valores. Os nomes são uma necessidade humana com a qual os computadores têm de trabalhar para conseguir comunicar com os utilizadores. Do lado contrário, os utilizadores trabalham com conceitos a que dão nomes. A utilização de valores permite apenas resolver situações concretas e não construir soluções. Desta forma, identificam-se três camadas distintas: conceitos, nomes e valores computáveis. Enquanto os utilizadores se preocupam com os conceitos e os nomes que lhes atribuiram, os computadores trabalham com os nomes e os valores que lhe estão associados.

Os nomes são a forma utilizada pelos humanos para comunicar conceitos entre si, e, por consequência, são a forma utilizada para dialogar com os computadores. Para facilitar o processamento da informação que lhes é fornecida os computadores utilizam linguagens simplificadas. Estas linguagens utilizam identificadores para estabelecer paralelos entre os conceitos e as estruturas computacionais que modelam esses mesmos conceitos. Desta forma, o modelo proposto baseia-se na escolha de identificadores, nomes associados a alguma entidade, como forma privilegiada de abstracção. Os identificadores têm um significado semântico essencial no desenvolvimento de software, seja para designar tipos de dados ou entidades isoladas [Boo94, ED97].

O nome utilizado para designar um identificador pode não ser, só por si, suficientemente esclarecedor. Torna-se necessário complementar o nome com um contexto que remova qualquer tipo de ambiguidade na descrição da abstracção que se pretende representar. Os identificadores necessitam, desta forma, ser designados por um nome e

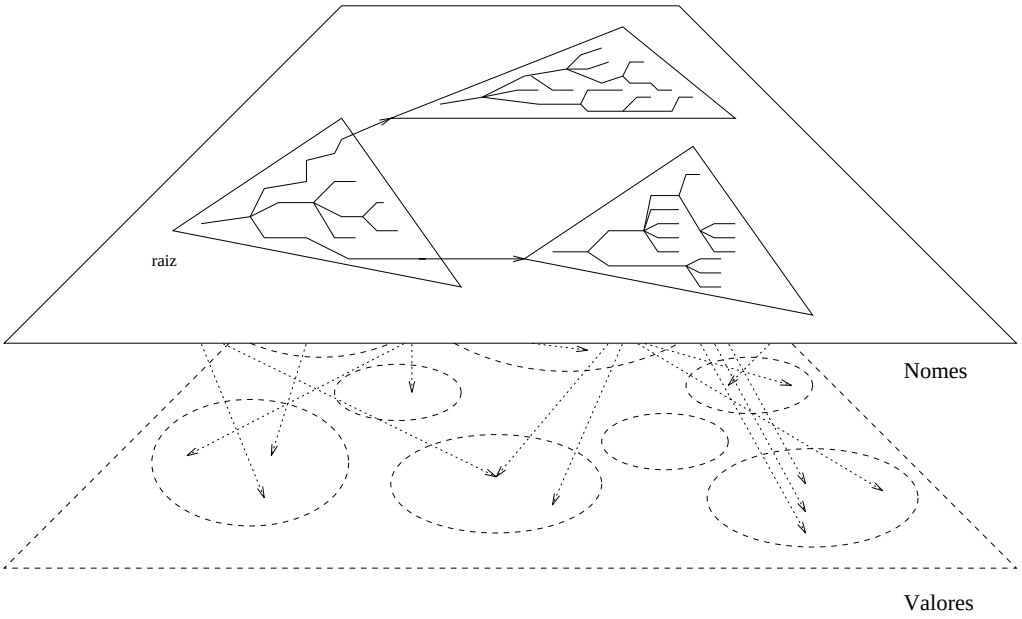


Figura 3.1: Descrição dos componentes por uma hierarquia de identificadores.

um contexto. As hierarquias, como foi visto atrás, surgem como forma natural de ordenar e estratificar a informação. Os identificadores são catalogados e colocados a níveis da hierarquia, exibindo a localidade desejada, sendo a sua posição na hierarquia representativa do contexto em que se inserem. Uma hierarquia de nomes surge como um bom modelo para representar os sistemas por decomposição da complexidade.

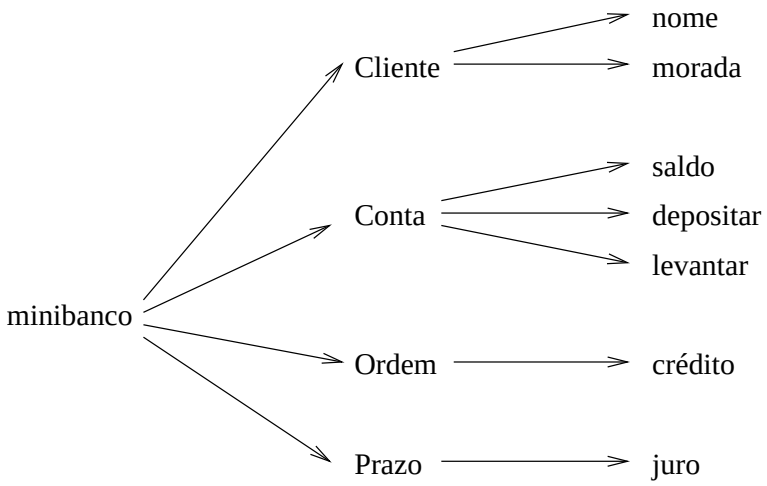


Figura 3.2: Simplificação da descrição hierárquica do `minibanco`.

Numa primeira aproximação, a disposição dos nomes do exemplo 3.1 pode utilizar três níveis que representam o sistema, as estruturas/classes e as funções/variáveis. Nesta perspectiva, a hierarquia de nomes da figura 3.2 apresenta uma solução muito simplifi-

cada do `minibanco`.

3.5 Síntese

Os nomes são utilizados pelos utilizadores para referirem entidades identificáveis nos componentes, apresentando um conjunto de características que evidenciam, de uma forma expressiva, os elementos constituintes do componente. A sua escolha criteriosa fornece informação importante sobre o componente. As funções desempenhadas pelos nomes na descrição dos componentes torna-os elementos essenciais na sua caracterização. A organização dos nomes em estruturas complexas permite descrever conceitos de uma forma mais completa. A utilização de hierarquias de nomes permite representar a informação em níveis sucessivos, que correspondem a descrições de granularidade crescente. Uma estrutura hierárquica de identificadores permite reter apenas a organização dos nomes, suprimindo a restante informação, obtendo-se uma descrição rica e compacta do componente. Desta forma, obtém-se um modelo que agrega e ordena as características consideradas relevantes dos componentes de uma forma sucinta, tal como foi identificado em 2.1. A estrutura regular da hierarquia facilita o desenvolvimento de uma linguagem para seleccionar componentes com base nos nomes classificados.

Capítulo 4

Processo de classificação

Para que um componente seja reutilizado é necessário, ao projectista, ter mentalmente presente a sua existência e ter meios para o indagar. O primeiro caso só é realizável pelos humanos se estivermos em presença de um pequeno número de componentes, devido às limitadas capacidades de memorização. Tal é possível nalgumas linguagens em que o número de primitivas disponíveis é reduzido, tornando a consulta de documentação ocasional. Desta forma, a documentação pode existir apenas em papel e sem necessitar de mecanismos de busca complexos. Com o aumento da complexidade das linguagens e das respectivas bibliotecas de componentes, o número de componentes disponíveis torna-se muito numeroso impossibilitando a sua presença mental permanente. Assim, é necessário dispor de mecanismos de busca que permitam indagar sobre a existência de possíveis candidatos a reutilização.

O processo de busca mais simples recorre à análise dos componentes um por um. No entanto, a complexidade da busca é linear com a complexidade de cada componente. Além disso, as diversas buscas baseiam-se, na sua grande maioria, nas mesmas características do componente. Desta forma surge como aceitável extrair previamente essas mesmas características, através de um processo de classificação, e basear as buscas numa representação compacta e optimizada dessas mesmas características. A identificação das características do componente e a escolha das características consideradas relevantes determina a qualidade do processo de classificação. O processo de selecção irá efectuar buscas nessas características, previamente classificadas, e determinar o grau

de semelhança de cada componente face aos requisitos formulados. Como resultado, o utilizador do sistema obterá um número, preferencialmente reduzido, de componentes candidatos ao cumprimento dos requisitos. A qualidade do processo de selecção é medido em função das taxas de recuperação, precisão, sobreposição, e outros critérios estudados na página 39. Estes critérios de qualidade são directamente dependentes do processo de selecção, na medida em que a flexibilidade na formulação dos requisitos e a determinação de semelhança influenciam a escolha de certos componentes em detrimento de outros. Contudo, o processo de classificação influencia indirectamente o processo de selecção e, consequentemente, os referidos critérios de qualidade. De facto, a determinação das características a utilizar na classificação condicionam o processo de selecção, pois a determinação incorrecta de determinada característica ou a sua omissão no processo de classificação influenciam o cálculo da semelhança. Desta forma, certos componentes serão preteridos em favor de piores candidatos devido a erros ou imprecisões de classificação.

Os processos de classificação e selecção necessitam ser tratados em conjunto, uma vez que o grau de sucesso só é mensurável em função dos componentes existentes no sistema face aos obtidos na selecção. A atribuição de responsabilidades devidas às falhas do sistema, entendido como o processo conjunto de classificação e selecção, pode não ser exclusivamente imputável a uma das partes mas ao sistema no seu conjunto. Assim, embora em termos de reutilização, a classificação e a selecção de componentes pertençam a processos de desenvolvimento distintos (ver página 12), a sua avaliação deve ser realizada em conjunto.

4.1 Mecanismos de classificação

A classificação de um componente é efectuada quando o componente é introduzido no sistema de reutilização. O processo de classificação deverá permitir a introdução de componentes no sistema, sem que os já existentes sejam afectados por essa introdução. Da mesma forma a remoção de componentes, embora menos frequente, deverá ser igualmente transparente. Como esta introdução ocorre uma única vez, ao contrário

da selecção que deverá ocorrer frequentemente, este processo tem sido essencialmente manual. No entanto, o número cada vez mais elevado de componentes disponíveis tem conduzido à necessidade de automatizar, mesmo que parcialmente, o processo de classificação. Numa primeira aproximação torna-se necessário classificar os novos componentes, ou seja, aqueles que nunca foram classificados por nenhum processo e cuja taxa de geração é cada vez maior. Mesmo que se considere como reduzido o número de novos componentes, é necessário ter em conta que não existe um sistema de classificação ideal, pelo que os componentes podem ter de ser classificados segundo diversos sistemas. Estes sistemas, quer pelas opções de classificação e selecção tomadas, quer pela área em que se inserem, permitem alargar o espectro de reutilização de um dado componente, melhorando a sua rentabilidade. Além disso, em especial em sistemas de classificação com base em vocabulários pré-definidos, o aumento do número de componentes obriga a alargar o respectivo vocabulário de classificação. Desta forma consegue-se melhorar a distinção entre os vários componentes, e, consequentemente, diminuir significativamente a sobreposição e melhorar a precisão e recuperação. No entanto, tal processo de reclassificação torna-se cada vez mais dispendioso quanto maior for o número de componentes existentes na biblioteca. O grau de automatização é a medida de qualidade mais importante do processo de classificação que pode ser avaliada *per si*, sem recurso ao processo de selecção.

4.1.1 Extracção de identificadores

No capítulo 3 apresentou-se um modelo baseado em identificadores, como forma de descrever as abstracções que cada componente realiza. Logo, o processo de classificação, necessário para construir o referido modelo, fundamenta-se no reconhecimento de identificadores. Os identificadores são extraídos da documentação existente sobre os componentes. Esta documentação pode existir sob a forma de texto livre, caso em que são necessários métodos baseados em linguagem natural (ver secção 2.2.1), métodos estatísticos (ver secção 2.2.4) ou a intervenção manual. Outra solução consiste em recorrer a documentação que obedeça a linguagens rigorosas, como, por exemplo, métodos formais (ver secção 2.2.4) ou linguagens de desenho ou codificação (ver secção 2.2.3).

Os métodos formais representam de uma forma muito rigorosa a funcionalidade do componente, mas não permitem concluir sobre a sua estrutura interna ou desempenho. Por outro lado, os métodos estruturais permitem recolher informação sobre a estrutura do componente. Contudo, nem sempre é simples inferir, a partir dessa informação estrutural, a funcionalidade e o desempenho do componente.

O modelo proposto adapta-se de uma forma mais natural a técnicas de classificação baseadas em modelos estruturais, embora os identificadores possam ser igualmente extraídos com recurso a técnicas baseadas em modelos textuais. A utilização de técnicas baseadas em modelos formais é também possível, muito embora a representação com base em identificadores suprima a informação funcional essencial. Desta forma, o posterior processo de selecção utilizará apenas um subconjunto da informação formal disponível o que influenciará certamente a sua precisão e taxa de recuperação. O modelo de identificadores apresenta-se especialmente atractivo como forma de sintetizar informação esparsa, essencialmente originária de linguagens de arquitectura ("design") e programação ou linguagem natural.

Os nomes encontram-se, frequentemente, relacionados com propriedades e operações das entidades que modelam. Tal facto verifica-se na linguagem natural e é transportada para as linguagens de programação sob a forma de identificadores. Por exemplo, Booch [Boo94] sugere a identificação de atributos, a serem modelados como variáveis, através da detecção de substantivos utilizados na linguagem natural. Da mesma forma, sugere a identificação de operações, a serem modeladas como rotinas, através da detecção de verbos no texto. Também, Rumbaugh [Rum91] estabelece relações entre substantivos e atributos, além de sugerir que a modelação de associações entre entidades possa ter também por base verbos de estado.

Numa primeira aproximação o componente pode ser caracterizado pelos seus atributos — propriedades que são geralmente modeladas como variáveis —, operações — funções desempenhadas que são geralmente modeladas como rotinas — e associações — relações com outros componentes ou entidades exteriores. Do ponto de vista linguístico, as propriedades do componente estão relacionadas com substantivos, verbos transitivos ou adjectivos. As associações estão relacionadas com verbos de estado e verbos intransitivos; as operações estão relacionadas com os restantes verbos que não são

abrangidos por nenhuma das classificações anteriores. Um estudo recente [ED97], que recorre a uma classificação semelhante à atrás descrita, apresenta taxas de sucesso de 82% para as operações e de 91% para os atributos. Os valores são especialmente bons se atendermos a que o estudo não se baseou apenas em linguagem natural mas em comentários que acompanham o código, e que nem sempre recorrem a frases completas ou sintacticamente correctas. Aliás, os autores do estudo referem que grande parte dos 12% de incorrecções obtidas na determinação das operações se deve à má construção das referidas frases.

As linguagens de arquitectura e programação, por outro lado, têm muito deste trabalho já feito pois muitos dos pormenores linguísticos, irrelevantes do ponto de vista da classificação do componente, como as preposições, já foram removidos ou substituídos por símbolos especiais das linguagens de desenvolvimento escolhidas. Para mais, já foi feita uma primeira selecção dos termos relevantes e efectuada a sua classificação em termos de identificadores principais: atributos, associações e operações. As técnicas de classificação com base em pares atributo-valor e com base em assinaturas utilizam partes da informação disponível nos identificadores principais, como suporte à determinação de semelhanças, necessárias no processo de selecção (ver secção 2.2.3).

Os identificadores principais oferecem uma boa taxa de recuperação, ou seja, num processo de selecção, permitem obter quase todos os componentes que obedecem aos requisitos. No entanto, o recurso exclusivo a identificadores principais traduz-se numa baixa precisão, ou seja, muitos dos componentes seleccionados na realidade não obedecem aos requisitos (ver secção 2.2.3). Torna-se necessário complementar os identificadores principais com informação mais detalhada que permita distinguir pormenores que possibilitem a rejeição de componentes que não obedecem aos requisitos e, consequentemente, aumentar a precisão do processo de selecção. Note-se que, se a informação adicional não existir, por muito bom que seja o processo de selecção, os componentes podem ter sido classificados pelos mesmos identificadores principais, tornando a sua distinção impossível. De acordo com o modelo apresentado no capítulo 3, a informação detalhada é adicionada ao modelo através de uma decomposição hierárquica em árvore.

Tratamento de sinónimos

A riqueza do vocabulário existente nas linguagens permite uma escolha vasta de nomes para identificadores. Tal facto conduz, por um lado, a classificações diferentes e, por outro, à não determinação de semelhanças correctas no processo de selecção. A utilização de dicionários de sinónimos permite reduzir o espectro de vocábulos utilizados para definir uma mesma abstracção. As técnicas baseadas em vocabulários controlados utilizam um conjunto de sinónimos para cada um dos vocábulos do dicionário pré-definido. A escolha dos sinónimos tem de ser cuidadosa para minimizar erros de classificação devido à substituição de vocábulos em certos contextos. A necessidade de escolher entre um dos vocábulos pré-definidos conduz, por vezes, a imprecisões na classificação. Por outro lado, o alargamento do dicionário pré-definido obriga à redefinição dos sinónimos com vista a minimizar ambiguidades, por exemplo, um determinado vocábulo poder ser classificado por dois sinónimos distintos [DFF97].

Em técnicas baseadas em vocabulários não controlados, não existe necessidade para um completo tratamento de sinónimos. Tal facto restringe a expressividade dos vocábulos utilizados e pode ser considerado inclusivamente nocivo se se converter verbos em substantivos, ou vice-versa. De qualquer forma, o processamento de sinónimos pode ser efectuado no processo de selecção de uma forma controlada. Consegue-se, assim, gerir a precisão do processo de selecção, através da utilização de um maior ou menor número de sinónimos, mantendo a diversidade da informação de classificação. No entanto, mesmo com técnicas baseadas em vocabulários não controlados, a utilização de sinónimos permite uniformizar o género — masculino ou feminino — ou número — plural ou singular — dos vocábulos ou o tempo dos verbos. Verifica-se, contudo, que tais variações são pouco frequentes, estando 83% dos verbos no tempo presente [ED97].

4.1.2 Construção da hierarquia

A necessidade de distinguir entre componentes com características semelhantes obriga a representar um número elevado de pormenores. As técnicas baseadas em vocabulários pré-definidos têm grande dificuldade em representar pormenores de complexidade

crescente, pelo que, em situações em que existem vários componentes semelhantes, exibem uma baixa precisão. A sua elevada taxa de recuperação torna-os, contudo, úteis para efectuar uma primeira selecção. As técnicas que não recorrem a vocabulários pré-definidos, pelo contrário, devido à sua mais baixa recuperação e boa precisão são especialmente úteis para distinguir pequenos pormenores entre dois ou mais componentes. Para tal, é necessário registar tais pormenores em graus de granularidade crescente. No modelo apresentado no capítulo 3 tais pormenores são registados numa hierarquia.

A hierarquia permite classificar os identificadores em diversos níveis de granularidade. Uma vez que os níveis mais interiores da hierarquia resultam da partição dos níveis mais próximos da origem, pode-se associar identificadores que representem maior pormenor e complexidade aos extremos da hierarquia. Assim, os níveis mais próximos da origem representaram abstracções mais genéricas que funcionam como base para uma primeira selecção, por forma a permitir uma boa taxa de recuperação ao processo de selecção. Os níveis mais interiores são analisados quando o número de componentes seleccionados é elevado e existe a necessidade de aumentar a precisão através da eliminação de candidatos menos próximos dos requisitos. A fixação da profundidade do algoritmo de busca permite considerar um maior ou menor número de pormenores, embora à custa de um aumento do tempo necessário para analisar cada componente. É assim possível, de uma forma simples, controlar a precisão do método de selecção. Note-se, contudo, que o controlo da precisão pode ser efectuado em cada busca individual, permitindo inclusive refinar buscas sucessivas, e não à custa da fixação de um determinado grau de granularidade *a priori*, como acontece em muitos processos de classificação (ver secção 2.2.2).

A grande capacidade de escalamento das hierarquias quer em largura, quer em profundidade, permite adquirir novas características sobre os componentes e complementar as descrições existentes com informação adicional. Esta informação adicional é importante para melhor descrever os componentes, mas essencialmente para permitir uma descriminação efectiva entre os componentes. Ao verificar-se que dois ou mais produtos são descritos pelo mesmo conjunto de características, ou que são sistematicamente seleccionados em conjunto, reduzindo a precisão das buscas, é possível corrigir a sua classificação. As diferenças entre os referidos componentes podem ser apenas de pormenor,

bastando introduzir mais um ou outro identificador aumentando a profundidade da hierarquia. Contudo, se as diferenças, embora significativas, não tiverem sido captadas pelo método de classificação, é possível aumentar a hierarquia em largura.

No processo de classificação, cada componente é considerado individualmente e é construída uma árvore individual. Posteriormente, as diversas árvores podem ser combinadas numa só para permitir a associação de informação existente nas diversas fases do processo de desenvolvimento, ou nas diversas versões de uma mesma fase. O processo de construção da hierarquia, quando efectuado a partir de linguagens que exibam uma estrutura clara, como as linguagens de desenho ou programação, é bastante simples. Tal é o caso dos paradigmas de programação imperativa e, em menor escala, da programação funcional. O objectivo é manter as estruturas das descrições dessas linguagens, uma vez que essas descrições já resultam de uma análise que conduziu a um processo de classificação dos requisitos em função das características da linguagem. Pode-se, assim, afirmar que parte do processo de classificação já se encontra efectuado, nomeadamente em termos de identificadores principais e secundários. Os identificadores principais surgem nas assinaturas das estruturas e rotinas da linguagem, enquanto os secundários resultam da decomposição dos tipos de dados das estruturas e da decomposição em instruções cada vez mais simples das rotinas. A classificação, embora retire pormenores lexicais e sintácticos da linguagem, retém a estrutura dos componentes nela representada, limitando-se a converter uma descrição textual linear numa estrutura hierárquica. Como o processo de classificação assume uma estrutura lexical, sintáctica e semanticamente correcta, uma vez que se assume que o componente está apto a ser reutilizado e não se encontra em desenvolvimento, a árvore a construir reflectirá os aspectos estáticos do componente.

4.1.3 Enriquecimento da descrição

Uma descrição estrutural é essencialmente estática e, consequentemente, muitas vezes insuficiente para caracterizar o componente de uma forma minimamente completa. Muito embora seja possível, através de analisadores estáticos ou simulações, obter con-

clusões sobre algum comportamento dinâmico e funcional do componente, estes métodos são geralmente dispendiosos e pouco rigorosos.

Como foi visto atrás, a hierarquia permite enriquecer sucessivamente a descrição do componente através da adição de novas propriedades. Desta forma é possível aumentar a informação disponível sobre o componente, permitindo uma maior precisão do processo de selecção. As propriedades a adicionar à descrição estrutural existente, que resulta da extracção de identificadores das linguagens utilizadas na realização do componente, podem ser obtidas através de outras técnicas de classificação. É, assim, possível criar ramos da hierarquia, com nomes pré-definidos que contêm informação utilizada por outros métodos de classificação. Os novos identificadores e os seus valores podem resultar de processos de classificação automática, ou serem introduzidos manualmente pelo utilizador após a análise dos componentes ou como forma de corrigir comportamentos particulares dos algoritmos de selecção utilizados [TV97]. Por exemplo, o cálculo de métricas de complexidade ou desempenho podem ser determinantes na escolha entre algoritmos de ordenação. A utilização de facetas permite, mediante a utilização de vocabulários pré-definidos, uma primeira selecção com uma taxa de recuperação elevada. No entanto, devido à facilidade de escalamento da estrutura hierárquica é possível representar vários níveis de facetas, obtendo-se classificações semelhantes às observadas no Dewey Decimal System [Dew79].

Como veremos no capítulo 5, o sistema proposto permite efectuar buscas utilizando apenas determinados ramos da hierarquia, pelo que é possível, desde que exista informação disponível, integrar várias técnicas de representação distintas na mesma hierarquia. No processo de selecção é, assim, possível recorrer a cada uma dessas técnicas de classificação individualmente ou combiná-las através de buscas mais complexas. A utilização de uma linguagem de busca permite construir buscas arbitrariamente complexas por forma a permitir a selecção do componente que mais se aproxima dos requisitos, ou seja, será idealmente possível obter a precisão e recuperação máximas.

Critérios funcionais

Ao pesquisar uma biblioteca de componentes reutilizáveis, o primeiro objectivo do projectista é indagar a presença de componentes que realizem a função pretendida. Embora critérios de desempenho ou disponibilidade em determinado sistema ou linguagem possam ser importantes, estes critérios podem, geralmente, ser passados para segundo plano devido ao suporte de interoperabilidade existente entre muitos sistemas e as ferramentas de transformação e adaptação disponíveis.

Nem sempre o projectista realiza no componente toda a semântica da especificação, por vezes alguns aspectos são omitidos devido a opções no processo de desenvolvimento, a restrições do ambiente utilizado ou a esquecimentos. É necessário captar o comportamento do componente através de descrições que permitam objectivamente determinar se obedecem a determinados requisitos, ou qual o grau de semelhança entre o comportamento do componente e os requisitos. A utilização de restrições aos dados é uma solução frequente quer através do recurso a tipos de dados, quer através de expressões invariantes. Uma outra solução consiste na imposição de pré e pós-condições às operações que as rotinas realizam sobre os dados do componente. No entanto, é especialmente complexo determinar o grau de semelhança, e mesmo de emparelhamento exacto entre duas representações formais (ver secção 2.2.4).

Desta forma, a utilização de identificadores obriga a uma classificação com base em vocábulos com origem na linguagem natural, e não com base em expressões matemáticas. Esta classificação com base em vocábulos facilita, contudo, a posterior determinação do grau de semelhança, em geral com recurso a dicionários de sinónimos. Uma classificação deste tipo obriga a uma escolha extremamente cuidada dos vocábulos utilizados. Resumindo, a introdução de critérios funcionais numa descrição baseada em identificadores exige uma classificação manual, de preferência por pessoas experimentadas.

Critérios não funcionais

Os critérios não funcionais permitem, geralmente, a escolha entre opções funcionalmente equivalentes. Embora a presença, ou não, destes critérios possa ser razão bastante para a sua rejeição, o seu rigor não é tão crítico como na descrição funcional. Assim, os critérios não funcionais são em geral mais fáceis de automatizar, uma vez que a determinação de semelhança no processo de selecção, obedece a técnicas mais difusas. Além de métricas directamente extraídas do componente, os identificadores com características não funcionais podem representar conclusões ou medidas da experiência anterior com o componente. A história das reutilizações bem sucedidas do componente pode dar indicações valiosas sobre as áreas de aplicação do componente e funcionar como indicativo qualitativo do custo de adaptação a outras áreas, por exemplo. Estes critérios não funcionais são essenciais em fases finais do processo de selecção, em que a precisão é um factor crítico, e que exigem intervenção humana para a eleição do candidato a reutilizar [TV97].

4.2 Classificação de componentes

No processo de classificação pretende-se dispor de uma forma organizada os nomes utilizados no desenvolvimento dos componentes a reutilizar. A descrição que se obtém do componente deverá reflectir informação nele contida que possa ser relevante analisar no momento da sua selecção para reutilização. Assim, a disposição de nomes a obter deverá fornecer informação útil sobre a estrutura do componente. Por forma a manter a estrutura original do componente, o modelo de classificação proposto procura manter a hierarquia exibida pela linguagem de descrição utilizada. As diversas linguagens de modelação e programação podem apresentar diferentes graus de profundidade na descrição do componente, a informação comum está presente e surge em zonas semelhantes da descrição hierárquica. De igual forma, dois componentes que resolvem o mesmo problema apresentam bastantes semelhanças estruturais, quer sejam descritos em linguagens distintas quer sejam realizadas, na mesma linguagem, por pessoas diferentes.

O processo de classificação proposto procura traduzir a hierarquia resultante da decomposição do problema com vista à obtenção da solução. Este processo pode ter por base linguagens de modelação, quer ao nível da análise quer ao nível da arquitectura, ou linguagens de programação. A construção da descrição pode ter origem em descrições informais ou textuais, embora neste caso o processo de classificação seja manual. No entanto, quer o processo de classificação seja manual ou automático a metodologia proposta baseia-se sempre na representação da decomposição hierárquica do componente. Para exemplificar, em casos concretos, a aplicação da metodologia escolheu-se uma linguagem de modelação – **UML** –, uma linguagem de programação – **C++** – e a introdução manual de anotações.

4.2.1 Desenho de componentes

A Linguagem de Modelação Unificada – **UML** – extende e adapta os muitos dos trabalhos anteriormente efectuados na representação de aplicações e componentes ao nível da análise e desenho, sendo designada por linguagem de terceira geração. No **UML**, os diversos tipos de sistemas são descritos através de um modelo que pode ser representado através de um conjunto de diagramas. Estes diagramas permitem descrever informação estrutural, evolutiva e organizativa. A **UML** permite, ao contrário de outras aproximações referidas em 2.3.3, efectuar comparações simples e rigorosas entre representações alternativas, o que torna a sua utilização especialmente atractiva na classificação e selecção de componentes. Como o modelo de classificação proposto no capítulo 3 se integra nos modelos estruturais descritos em 2.2.3, a informação estrutural contida no diagrama de classes **UML** é aquela que melhor se traduz para o modelo proposto nesta dissertação. No entanto, informação evolutiva e organizativa pode igualmente ser integrada de uma forma condensada, o que obriga à intervenção humana e será tratada como introdução manual de anotações.

Em **UML**, o diagrama de classes é constituído por um conjunto de classes e suas interrelações que modelam o sistema em causa. As classes são representadas graficamente por um rectângulo onde se incluem o nome da classe, os seus atributos e as operações

que suporta. As associações são representadas por linhas quebradas que unem as classes envolvidas na associação (ver figura 4.1).

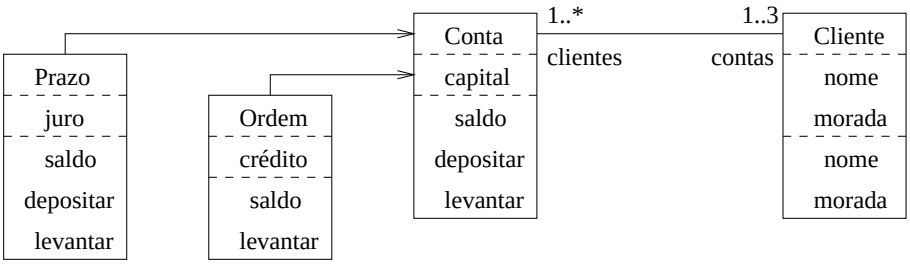


Figura 4.1: Diagrama de classes **UML** do `minibanco`.

A hierarquização de uma representação **UML** começa por descrever os elementos mais genéricos, introduzindo um grau crescente de pormenor nos níveis subsequentes da hierarquia. O nível mais elevado da hierarquia é constituído pelo sistema em causa, descrito pelo seu nome simbólico, sendo decomposto nas classes que o compõem, representadas pelo seu identificador. Cada classe é, seguidamente, subdividida em três componentes:

operações O sub-nível das operações é constituído pelos métodos a que o objecto responde, sendo frequentemente modelados como rotinas que manipulam os atributos. Cada operação é descrita pelo nome que a identifica, podendo ser decomposta em nome do tipo de dados de retorno e dos argumentos que recebe. Estes argumentos podem, por sua vez, ser caracterizados por um tipo de dados.

atributos Os atributos são compostos pelos elementos que constituem o estado do objecto, e que serão modelados, em geral, como variáveis de instância. Cada variável pode ser caracterizada por um tipo de dados que descreve o seu comportamento e um valor de inicialização. Do ponto de vista do modelo proposto, apenas o valor de inicialização não necessita ser considerado na classificação pois o processo de selecção não o utiliza.

associações As associações que cada classe estabelece com algumas das outras classes do sistema são de grande importância estrutural pois escondem dependências que não podem ser ignoradas no processo de selecção. Cada associação, além de ser descrita por um nome, é decomposta em cardinalidade das classes envolvidas e

função desempenhada por cada uma dessas classes na associação. As associações podem ser caracterizadas por indicadores de dependência e ser classificadas como agregações e composições, de acordo com a forma como se relacionam com a classe de que dependem.

Embora nem toda a informação acima descrita esteja sempre presente, em especial em fases de análise não detalhada, a representação obtida pelo processo de classificação deve reflectir toda a informação disponível. Muita da informação relativa a tipos de dados só estará disponível em fase de desenho detalhado, conforme desenvolvido em 7.1.1. Se bem que a não existência de certo tipo de informação possa impedir a recuperação do componente no processo de selecção, a taxa de recuperação da selecção não é afectada pois a informação em causa também não existia no documento original.

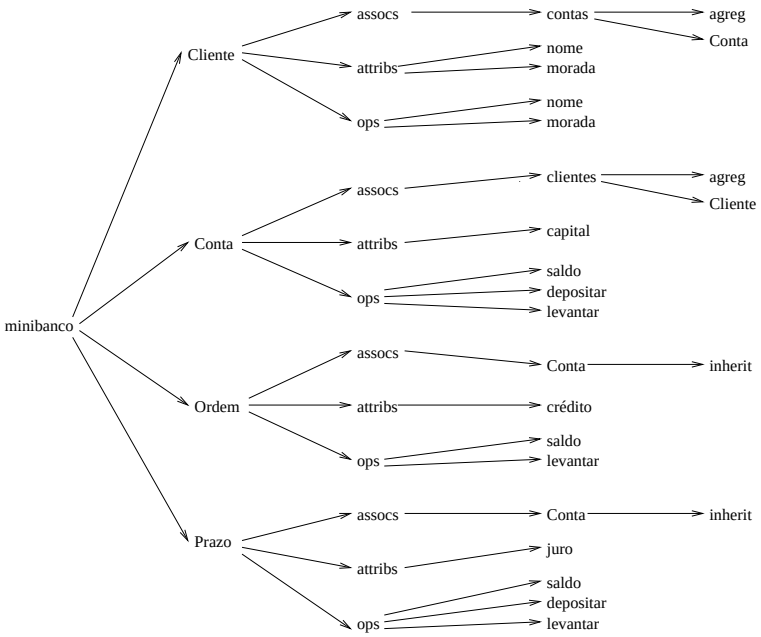


Figura 4.2: Hierarquia de identificadores obtida do diagrama de classes da figura 4.1.

O algoritmo de construção da hierarquia de identificadores a partir de uma descrição **UML** é:

1. A hierarquia tem o nome do componente para identificação no catálogo. A raiz da hierarquia identifica o documento original que descreve o componente.

2. Os vários ramos do primeiro nível da hierarquia são acedidos através dos nomes das classes encontradas nas descrição **UML**, e os respectivos vértices indicam o local no documento onde se declaram cada uma dessas classes.
3. O segundo nível da hierarquia é fixo e descreve os três grupos – associações, atributos e operações – e o ponto no documento que os descreve.
4. O terceiro nível inclui, na sub-hierarquia do respectivo grupo, os identificadores dos atributos e associações (variáveis) e operações (métodos), bem como o nome das classes de quem a classe em extracção tem associações de herança.
5. O quarto nível descreve a informação de tipo de cada um dos respectivos identificadores no nível anterior. A informação de tipo inclui não só o tipo de dados como os identificadores que os caracterizam como, por exemplo, herança, agregação ou composição para as associações.

Exemplo 4.1 *Recorrendo a uma representação simplificada do exemplo 3.1, obtemos um diagrama de classes descrito na figura 4.1 que produz a representação hierárquica de identificadores da figura 4.2. O primeiro nível contém os nomes das classes identificadas no diagrama, a que se seguem os conjuntos de associações, atributos e operações que caracterizam um digrama de classes em* **UML**. *Os últimos níveis da hierarquia descrevem os elementos que dão corpo a cada um dos três conjuntos referidos, bem como informação que os caracteriza. Por exemplo, as associações são identificadas pelo nome e caracterizadas pela classe com que se relacionam, bem como pelo tipo de associação que definem.*

Como os modelos de análise e de desenho são descrições condensadas e simplificadas do sistema informático que as realiza, quase toda a informação presente é representada na classificação proposta. No entanto, como determinadas características dos modelos, como as cardinalidades, não são extraídas pelo processo de classificação, não é possível reconstruir o modelo a partir da informação utilizada na classificação.

4.2.2 Codificação de componentes

As linguagens de programação incluem muito mais informação que os modelos de análise e desenho em **UML** do mesmo sistema. A descrição de níveis de representação

mais elevados apresentam muitas semelhanças com os modelos de análise e desenho, embora atributos possam ser codificados como rotinas e as associações possam ser desempenhadas por variáveis ou rotinas. O principal contributo estrutural da fase de codificação reside no conteúdo das rotinas, uma vez que o diagrama de classes em **UML** apenas descreve a interface. A estrutura interna das rotinas está omisso no diagrama de classes **UML**, sendo coberto por outros diagramas que não foram considerados para efeitos de classificação.

As rotinas são conjuntos de instruções, normalmente executadas sequencialmente, onde se salientam instruções condicionais e de ciclo. Estas instruções influenciam o processo de selecção do componente do ponto de vista de desempenho e de dimensão do código, constituindo atributos de classificação a inserir manualmente. Tal é o caso, por exemplo, de rotinas de ordenação, onde as diferenças de desempenho só podem ser inferidas após uma cuidadosa análise. No entanto, a rigorosa classificação dessas instruções não se traduz em informação útil ao processo de selecção, na grande maioria das situações de selecção.

Por outro lado, a chamada a outras rotinas contém associações que, na sua grande maioria, não existiam nas fases de análise e desenho em **UML**. Por exemplo, o cálculo do factorial pode ser recursivo ou iterativo, sendo tal facto visível da existência, ou não, de uma chamada à própria rotina. Também diversas realizações de pilhas de dados podem ser distinguidas através das respectivas chamadas a rotinas de reserva dinâmica de memória, como se exemplifica na tabela 4.1.

Tabela 4.1: Distinção entre tipos de pilhas de dados através de chamadas a rotinas de reserva de memória.

stack	fixed	variable	dynamic	list
create		malloc	malloc	
push			realloc	malloc
pop				free
delete		free	free	

As linguagens de codificação dispõem de muita informação que não é relevante para

a selecção dos componentes. A informação estrutural classificada nas descrições na fase de desenho representa o essencial da informação útil para a selecção do componente para reutilização. Desta forma, a representação de componentes com origem em linguagens de codificação assemelha-se às representações obtidas para descrições de desenho detalhado por linguagens de modelação. As descrições com origem em linguagens de codificação apresentam, em geral, descrições com mais pormenor pois resultam de uma fase posterior do processo de desenvolvimento de software.

```
class Conta {
public:
  virtual int saldo() { return capital; }
  virtual int depositar(int valor) { capital += valor; return saldo(); }
  virtual int levantar(int valor) { capital -= valor; return valor; }
  Cliente *cliente(int i) { if (i > 0 && i < 4) return cli[i-1]; return 0; }
private:
  int capital;
  Cliente cli[3];
};
```

Figura 4.3: Codificação em **C++** da classe `Conta` do `minibanco`.

O algoritmo de construção da hierarquia de identificadores a partir de uma classe **C++** consiste na identificação de blocos sintácticos que se assemelham a regras de um analisador sintáctico:

1. A hierarquia tem o nome do componente seguido dos ramos respeitantes a cada uma das classes descritas, como no caso da extracção da **UML**.

2. O segundo nível da hierarquia é fixo e descreve três grupos de protecções: privado, protegido e público.

3. O terceiro nível inclui, na sub-hierarquia do respectivo grupo, os identificadores das variáveis e métodos, bem como o nome das classes de quem a classe herda.

4. O quarto nível descreve, para as variáveis, a informação de tipo de cada um dos respectivos identificadores no nível anterior. No caso das funções, existem quatro grupos fixos: retorno, chamadas, argumentos e estrutura.

5. Os níveis seguintes incluem a informação de tipo dos identificadores do nível anterior e a informação estrutural. A estrutura corresponde às condições, ciclos e blocos de **C++** onde os identificadores em cada um são descritos sem ordem. O algoritmo termina na descrição do tipo dos identificadores do bloco mais interior da linguagem.

Exemplo 4.2 *Recorrendo a uma descrição simplificada da classe* `Conta` *do exemplo 3.1, cuja interface é descrita na figura 4.3, obtemos a representação hierárquica de identificadores da figura 4.4. O primeiro nível da hierarquia, após o nome da classe em causa, define os ramos que contêm métodos e variáveis com o mesmo grau de acesso. Cada método ou variável contido nesses ramos é identificado pelo seu nome e decomposto nos elementos que o caracterizam, como, por exemplo, o tipo de dados.*

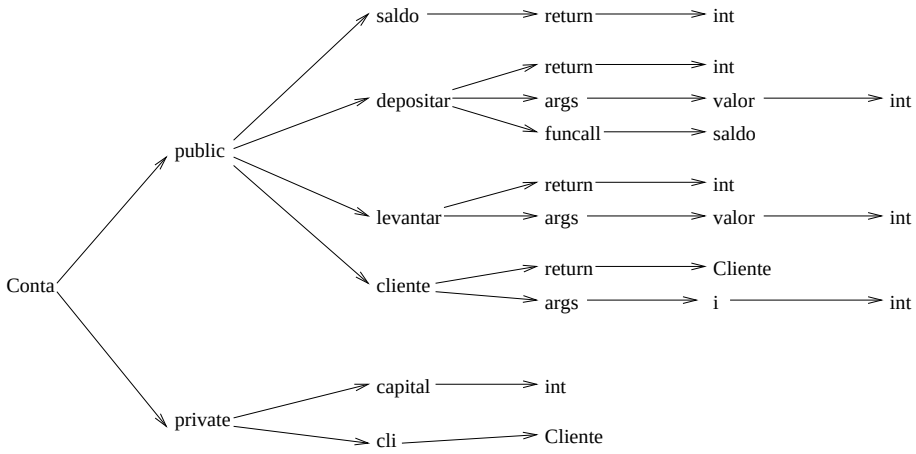


Figura 4.4: Hierarquia de identificadores simplificada obtida da classe da figura 4.3.

4.2.3 Introdução de anotações

A informação obtida a partir de descrições estruturais permite obter classificações diferentes para componentes semelhantes contribuindo para a redução da sobreposição no processo de selecção. No entanto, a inexistência de esterotipos degrada a taxa de recuperação se não for possível dispor de um mecanismo de sinónimos poderoso. A introdução manual de esterotipos, obtidos através de métodos de classificação baseados

num número limitado de vocábulos, permite aumentar a taxa de recuperação. A introdução de anotações com base em métricas, tais como complexidade, desempenho ou dimensão, permite aumentar a precisão do processo de selecção. Este tipo de anotações dificilmente podem ser inferidas de uma forma mais ou menos automática a partir de descrições do componente sem a intervenção humana.

O processo de introdução manual de anotações consiste na introdução, eliminação ou alteração do nome de identificadores na hierarquia, ou das entidades por eles referidas. Por exemplo, o método de classificação por facetas pode coexistir na hierarquia através da introdução de um ramo, geralmente designado por "`facets`". Este ramo contém os nomes das facetas escolhidas para classificar os componentes de uma dada biblioteca, e cujos valores representam grandezas e escolhas de um conjunto reduzido.

4.3 Descrição de componentes

Os identificadores usados na descrição dos objectos são modelados numa árvore, retendo uma referência para a sua posição original no objecto. Estes identificadores, depois de caracterizados, continuam a exibir as associações dos objectos originais. As equivalências representam, numa segunda fase, essas mesmas associações na árvore que se transforma num grafo [SvD95]. São as equivalências que permitem descrever relações de partilha bem como definir objectos como extensões de outros [BD96, Ast96]. Desta forma, enquanto os identificadores designam as entidades originais e os seus valores, as equivalências permitem a um identificador referir outro identificador.

A aproximação descrita não usa, em princípio, identificadores ou vistas pré-definidas. Na realidade, alguns identificadores, utilizados como pontos de entradas, são necessários. Desta forma, os identificadores utilizados, não necessitam ser aqueles que melhor se aproximam de entre os disponíveis, mas os identificadores efectivamente utilizados. A ausência de estereótipos permite maior realismo e expressividade, facilitando a selecção manual mas dificultando a utilização de métodos automáticos ou semi-automáticos. Como o processo de classificação tem em conta o contexto em que o identificador é utilizado, mesmo que se utilizem termos diferentes para descrever o mesmo conceito, estes

serão armazenados em zonas adjacentes. Este facto facilita a inspecção manual pois existe o conceito de localidade.

4.3.1 Propriedades da descrição

A descrição obtida, como resultado do processo de classificação, apresenta um conjunto de propriedades que a tornam especialmente flexível e abrangente, quando comparada com a maioria das descrições obtidas pelas técnicas analisadas na secção 2.2.

Independência da linguagem: os identificadores não são característica de nenhuma linguagem particular, tornando a descrição abstracta e livre dos aspectos léxicos e sintácticos. Esta propriedade permite que o processo de selecção possa encontrar componentes que obedeçam aos requisitos independentemente da fase do processo de desenvolvimento em que se encontram especificados ou da linguagem escolhida para a sua realização. É, assim, possível aproveitar os algoritmos ou o raciocínio caso o componente não exista num dos formatos desejados, e não existam ferramentas ou técnicas automáticas de conversão. Caso se pretenda seleccionar apenas componentes descritos em determinada linguagem, ou conjunto de linguagens, é sempre possível instruir o mecanismo de busca para rejeitar os componentes que não exibam esse requisito, apesar da descrição ser independente da linguagem.

Nominalismo: os nomes dos identificadores representam o comportamento real das entidades que modelam, assumindo uma escolha criteriosa dos mesmos. Esta propriedade resulta das propriedades dos nomes do modelo escolhido (ver secção 3.1). A não utilização de esterótipos permite reter a riqueza descritiva sem comprometer a taxa de recuperação do processo de selecção. Esta propriedade torna possível o controlo da taxa de recuperação, caso a caso, dependendo do uso de sinónimos mais ou menos abrangentes (ver secção 4.1.1).

Diferenciação: diferença entre as características escolhidas para descrever o componente. Esta propriedade é importante pois nem todos os comportamentos têm a mesma importância. A técnica de classificação pode controlar a importância de

cada identificador através da determinação do nível da hierarquia que lhe é atribuído. Esta possibilidade de controlar a granularidade permite reter abstracção e detalhe em simultâneo, podendo cada busca fixar a profundidade máxima a analisar, controlando o grau de abstracção a utilizar e, consequentemente, as taxas de sobreposição e recuperação obtidas.

Perspectivação: capacidade de avaliar separadamente comportamentos diferentes em contextos diferentes. Enquanto a profundidade da hierarquia influencia a diferenciação entre as características do componente, a perspectivação resulta da independência entre os diferentes ramos da hierarquia. Cada ramo da hierarquia descreve uma determinada perspectiva que pode ser caracterizada por decomposição com um detalhe crescente. A perspectivação agrupa informação relacionada explorando a localidade, o que facilita a distinção entre componentes semelhantes no processo de selecção.

Uniformidade: semelhança nos elementos utilizados para descrever os componentes e na estrutura para os representar. O facto de não existirem casos especiais simplifica, quer a manipulação automática pelas ferramentas de classificação e selecção, quer a inspecção manual pelo utilizador. A utilização de uma estrutura uniforme e flexível também oferece boas perspectivas de evolução, pois deverá permitir assimilar mais facilmente futuras extensões.

Embora a classificação seja uma técnica de sintetização e ordenação da informação, é importante reter grande riqueza descritiva. A estereotipagem pode ser obtida de uma forma controlada no processo de selecção. Além disso, é possível parametrizar os algoritmos de busca para efectuarem uma maior ou menor estereotipagem das descrições disponíveis. A principal desvantagem reside no facto de as descrições obtidas serem muito extensas, mas a ordenação dos identificadores e a regularidade da estrutura hierárquica permitem desenvolver ferramentas eficientes.

4.3.2 Representação formal

Para poder definir de uma forma rigorosa as operações de selecção sobre os dados, começaremos por definir rigorosamente a estrutura de dados que se obtém após o processo de classificação. Os elementos constituintes dos componentes são classificados com base em identificadores extraídos dos próprios componentes que são colocados numa hierarquia de nomes. A hierarquia de nomes é modelada por árvore dirigida, acíclica e etiquetada.

Definição 4.1 (hierarquia de nomes) *A* hierarquia de nomes *é um grafo* $G = (V, \Sigma, E)$*, onde cada vértice* $v \in V$ *representa uma referência para o identificador no documento original de onde o nome foi extraído, e cada arco dirigido em* $E \subseteq V \times V \times \Sigma$*, designado por* $v \xrightarrow{\sigma} w$*, é etiquetado. As etiquetas dos arcos são nomes, utilizados no documento original para designar o identificador, retirados de um alfabeto* Σ *que contém todos os nomes no sistema.*

Definição 4.2 (antecessor, sucessor) *Num arco* $e \in E$ *dirigido* $v \xrightarrow{\sigma} w$*, designa-se por* antecessor(e) *o vértice* v *e por* sucessor(e) *o vértice* w*.*

Definição 4.3 (caminho) *Um* caminho *do vértice* v_1 *para o vértice* v_k*, designado por* $c(v_1, v_k)$*, numa hierarquia é uma sequência de vértices* $v_1, v_2, \ldots, v_k, k \geq 1$*, tal que* $\exists_{\sigma \in \Sigma} \bullet(v_i, v_{i+1}, \sigma) \in E$ *para todo o* $i, 1 \leq i < k$*. Se não existirem arcos dirigidos que unam sucessivamente os vértices que separam um vértice* v_a *de um vértice* v_b*, então o caminho,* $c(v_a, v_b)$*, é definido como uma sequência vazia*

Definição 4.4 (comprimento) *O* comprimento *de um caminho* $c(v_1, v_k)$*, é uma função total* $\#c : V^2 \to (\mathbb{N} \cup \{\perp\})$ *que devolve o número de arcos que unem* v_1 *a* v_k*, na sequência de arcos que constituem o caminho* $c(v_1, v_k)$*. O comprimento de uma sequência vazia é* $\perp$*. O caminho de um vértice para ele próprio,* $c(v_a, v_a)$*, é uma sequência contendo apenas o vértice* v_a*, a que corresponde um comprimento* 0*.*

Definição 4.5 (raiz) *Existe um vértice, a* raiz $q_0 \in V$*, que não tem antecessores e a partir do qual existe um caminho para todos os restantes vértices, ou seja, cada vértice tem apenas um antecessor, com a excepção da raiz.*

Note-se que, caso se ignore as direcções dos arcos, considerando um grafo não dirigido, existe sempre uma ligação entre quaisquer dois vértices, a que corresponde o caminho da raiz para um vértice mais o caminho inverso do outro vértice para a raiz.

Resolução de nomes

Em alguma fase do processo de selecção dos componentes classificados será necessário aceder ao documento no ponto em que foi classificado na hierarquia. Este ponto é identificado pela etiqueta do arco que antecede o vértice que contém a referência ao documento. Para modelar a completa resolução dos nomes que conduzem a um dado identificador define-se um autómato de resolução de nomes:

Definição 4.6 (autómato de resolução de nomes) *Um* autómato de resolução de nomes *é um tuplo* $AN = (V, \Sigma, \Delta, \delta, q_0, \alpha)$. *Cada vértice na hierarquia é representado por um contexto* $q \in V$. *O alfabeto,* Σ, *é o conjunto de arcos dirigidos e etiquetados da hierarquia. A função de transição,* $\delta : V \times \Sigma \rightarrow Q$, *converte um contexto e um nome num novo contexto. A sua definição é baseada no conjunto de vértices da hierarquia tal que* $\delta(q, \sigma)$ *é definido como o vértice de destino em E a partir de q com o nome* σ, *e está indefinido se tal vértice não existir. O estado inicial,* q_0, *identifica a raiz da hierarquia a partir de onde os nomes serão resolvidos. O alfabeto de saída,* Δ, *designa as localizações dos documentos a obter através do autómato de resolução de nomes e contêm o endereço dos documentos e, por vezes, informação sobre o tipo do documento. A função de acesso,* $\alpha : Q \rightarrow \Delta$, *que converte vértices na informação necessária para aceder aos documentos, está definida na hierarquia. Esta informação consiste num endereço e em informação que indica como o endereço deve ser utilizado pelo sistema.*

Exemplo 4.3 *Na hierarquia de nomes da figura 4.5, considerando* V_1 *como o estado incial* q_0, *a sucessiva aplicação da função de transição à sequência de nomes* "quickSort", "compare" *produz o vértice* V_3. *Por aplicação da função de acesso obtém-se a localização do documento associado à referida sequência (ver figura 4.6).*

Os autómatos de resolução de nomes [New92] estão relacionados com os autómatos finitos [HU79]. A diferença reside no facto de um autómato de resolução de nomes devolver o resultado de aplicar a função de acesso ao contexto final, enquanto a aplicação

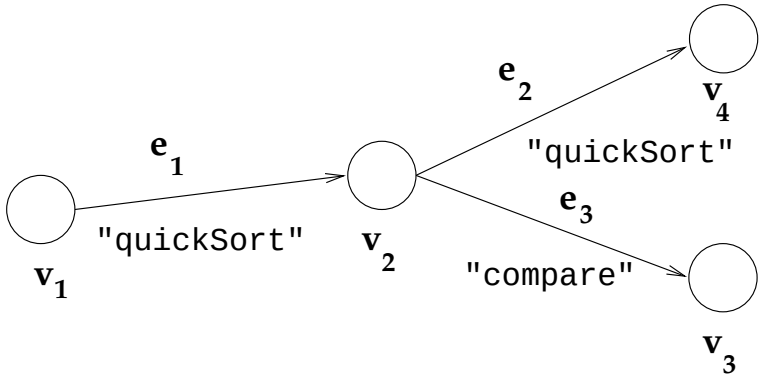


Figura 4.5: Exemplo de uma hierarquia de nomes.

de autómatos finitos produzir resultados booleanos, que indicam se o contexto final é ou não aceite. No caso das máquinas de Mealy e de Moore o resultado é retirado de um alfabeto de saída, mas a saída está associada ou com um determinado estado (máquina de Moore) ou com uma transição (máquina de Mealy) e não à aplicação da função de acesso ao contexto final.

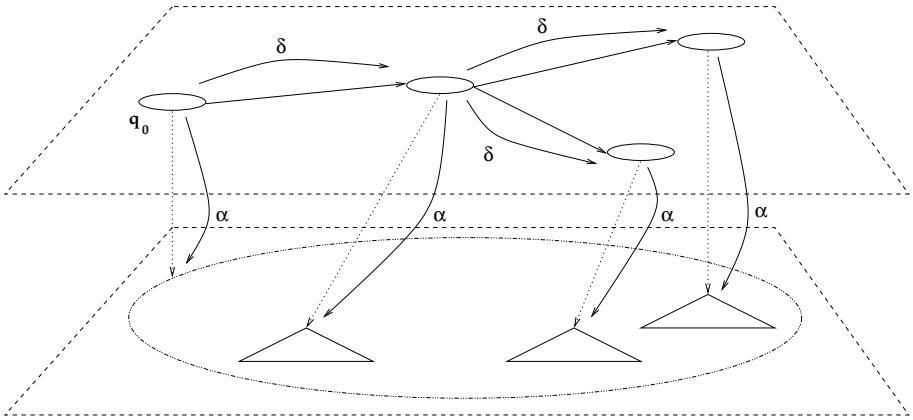


Figura 4.6: Exemplo de um autómato de resolução de nomes.

4.4 Síntese

Uma estrutura hierárquica de identificadores permite reter apenas a organização dos nomes, suprimindo a restante informação. A construção da hierarquia que representa o componente é efectuada reproduzindo as relações de agregação e decomposição existentes na linguagem utilizada na descrição do componente. A descrição do componente obtida é rica, compacta e independente dos pormenores lexicais e sintácticos

da linguagem utilizada para o desenvolvimento do componente. A hierarquia pode ser complementada com informação introduzida manualmente ou com recurso a outras ferramentas de classificação. Esta informação pode conter valores quantitativos ou qualitativos de desempenho, complexidade, ou outras métricas consideradas relevantes. O ambiente proposto não utiliza, ao contrário de outros sistemas, um tratamento de sinónimos no processo de classificação. A informação classificada não é estereotipada nem adulterada, transferindo esse tratamento para o processo de selecção onde o grau de sinonímia pode ser ajustado caso a caso.

Capítulo 5

Processo de selecção

Se a produção de software num ambiente, ou linguagem, com reduzido número de componentes reutilizáveis obrigava a desenvolver quase todo o software, a abundância de componentes levanta novos problemas. A procura de componentes torna-se, por vezes, tão fastidiosa como o seu desenvolvimento. A classificação permite ordenar as características dos componentes por forma a facilitar a sua procura. No entanto, se o número de características a classificar for pequeno resultarão muitos candidatos de cada procura, pois vários componentes serão classificados pelas mesmas características. Por outro lado, se o número de características a classificar for grande, a procura será ineficiente.

5.1 Mecanismos de busca

O processo de selecção procura obter o componente que melhor satisfaça os requisitos através do refinamento de sucessivas procuras. Para tal, é necessário dispor de mecanismos de busca flexíveis que possam efectuar procuras de diversas formas. De uma forma geral, o processo de selecção consiste numa primeira procura com base num conjunto inicial de critérios, em geral pouco exigentes, e posterior análise dos resultados obtidos. As procuras subsequentes podem resultar de um refinamento dos critérios por forma a excluir os componentes menos interessantes face aos requisitos. Se o número de componentes obtidos for muito elevado, pode-se optar por efectuar as buscas subsequentes

apenas no subconjunto de componentes previamente seleccionados, reduzindo o número de componentes a analisar. Assim, a selecção consiste em sucessivas operações de procura em que, após a análise dos resultados, se refinam os critérios de selecção e se limita o número de componentes a re-analisar.

5.1.1 Navegação

A determinação da validade dos resultados de uma procura é essencial para avaliar a necessidade de proceder a novas buscas e quais as alterações aos critérios de busca que é necessário efectuar. Esta avaliação pode ser efectuada por inspecção directa dos resultados ou basear-se num conjunto de métricas. Enquanto as métricas podem ser atractivas em fases iniciais do processo de selecção, a inspecção exaustiva dos resultados da busca, e até dos próprios componentes, é imprescindível nas fases terminais do processo.

A utilização de uma representação hierárquica permite descrições complexas mas torna possível percorrer toda a estrutura com recurso a apenas dois graus de liberdade: largura e profundidade. A utilização de um ponto de referência, a raiz, permite determinar o contexto actual e contribui para opções de navegação esclarecidas. Estes mecanismos simples são intuitivamente assimilados pela mente humana e são análogos a operações semelhantes efectuadas pelo sistema visual humano [WCG92]. A navegação recorre a sucessivas decisões, sendo cada uma delas simples. Tal como na condução de um veículo, cada decisão tem por base a posição corrente e um conjunto reduzido de opções possíveis. Mais uma vez, é muito importante que cada opção esteja clara e correctamente descrita, pelo que a escolha dos nomes para os identificadores é decisiva no tempo necessário para atingir o objectivo. Sistemas hierárquicos semelhantes têm sido utilizados em sistemas de ficheiros, controlo de versões, gestão de interdependências com considerável sucesso [BE96, WCG92].

Representação gráfica

A capacidade humana de apreender imagens complicadas baseia-se na aptidão em procurar objectos em paralelo, tornando possível procurar um objecto com características

pré-definidas rapidamente, e na facilidade de retenção das localizações desse objecto. Ao nível da selecção e classificação de componentes, diversos sistemas baseados em hipertexto têm sido propostos como formas privilegiadas de interacção com os utilizadores (ver secção 2.2.1).

Para tirar partido da capacidade humana de apreender uma imagem e concentrar a atenção em determinado objecto, várias ferramentas têm optado por ferramentas gráficas como forma de apresentar estruturas hierárquicas aos utilizadores [BE96, WCG92]. As hierarquias apresentam uma estrutura regular e bem organizada que facilita a navegação com base em interfaces gráficas. O utilizador começa com um conjunto de características gerais, que representam o maior nível de abstracção, e vai caminhando para situações cada vez mais concretas. Tal facto permite a fácil identificação de opções de navegação incorrectas, pois o utilizador é conduzido a contextos que não correspondem às suas expectativas. A utilização de ferramentas gráficas facilita a memorização visual dos caminhos, através de referências visuais como a posição, a cor ou o comprimento dos nomes. O desenvolvimento de interfaces expeditas torna o processo mais rápido e eficiente.

Embora a representação gráfica de hierarquias seja computacionalmente e cognitivamente eficiente não é necessariamente um subtituto para formas de diálogo textuais com o utilizador. As representações gráficas são certamente atractivas mas os resultados da sua utilização devem ser correctamente avaliados. Estudos recentes [Pet95] têm verificado que diferenças de notação e o facto de pessoas diferentes interpretarem uma mesma imagem de forma diversa conduz a capacidades distintas de assimilação de informação. Nomeadamente, a informação que um utilizador experimentado e treinado obtém de uma representação gráfica é dramaticamente diferente, em termos de velocidade e conteúdo, quando comparada com alguém que acaba de tomar contacto com o sistema. Mais importante parece ser o facto de, embora representações gráficas possam transmitir uma ideia de uma forma mais rápida e fácil que representações textuais, estas últimas surgem mais eficientes na comunicação de conceitos rigorosos e precisos.

5.1.2 Determinação de igualdades

O processo de selecção proposto tem por base a determinação de igualdades entre nomes de identificadores nos requisitos e nos nomes dos componentes dos caminhos que compõem as árvores. O objectivo consiste em determinar uma medida da igualdade entre os requisitos e as características dos componentes por forma a poder obter uma ordenação linear. O componente seleccionado é recolhido entre os primeiros elementos da lista, ou seja, o melhor candidato após a inspecção manual. A obtenção da lista ordenada tem por base critérios de igualdade que procuram medir o grau de semelhança entre os requisitos e cada um dos componentes. Este grau de semelhança é extremamente importante pois representa uma medida do esforço de adaptação do componente seleccionado para obedecer aos requisitos definidos na aplicação a que se destina [JM98].

A determinação de igualdades é o processo através do qual se compara o espaço do problema com o espaço das soluções. Enquanto o processo de classificação parte do espaço das soluções e produz uma representação intermédia, o processo de selecção parte do espaço do problema e produz uma pergunta que modela os requisitos. A classificação baseia-se na extracção de algumas características do componente, consideradas relevantes, produzindo uma representação simplificada mas incompleta. Por outro lado, a construção da pergunta baseia-se nas capacidades da linguagem utilizada para a descrever e no entendimento que o utilizador tem do problema. A determinação de igualdades tem por função verificar se a representação intermédia satisfaz a pergunta. Para tal, a linguagem em que a pergunta é expressa e a representação intermédia do componente devem ser compatíveis.

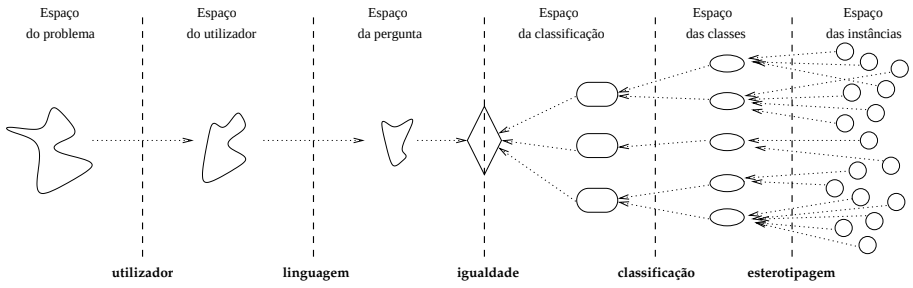


Figura 5.1: Determinação de igualdades entre componentes e requisitos.

A pergunta do utilizador pode ser vista como um predicado $Q(x)$ que recupera o componente c quando $Q(c)$ é verdadeiro. Os componentes que obedecem aos requisitos formulados na pergunta são dados por $\mathit{igualdade}(Q, C) = \{c \in C : Q(c)\}$. Esta igualdade é considerada exacta uma vez que os componentes obedecem integralmente ao predicado.

Igualdades ponderadas

Na ciência da computação têm sido estudados algoritmos de verificação automática de igualdades utilizadas na programação em lógica [Hog90]. No entanto, dado o número de transformações, a sua complexidade, e o facto de haver perda de informação em cada uma delas, existe uma grande margem para imprecisões e ambiguidades. Assim, os algoritmos de determinação de igualdades necessitam ter em conta a existência de tais imprecisões e ambiguidades. Como igualdades exactas são raras, ou mesmo inexistentes, é necessário considerar situações de semelhança. Esta semelhança pode dever-se ao facto de o componente ser mais genérico que os requisitos, sendo designada por igualdade genérica. Neste caso, é necessário especializar o componente, em geral por particularização ou herança. Por outro lado, se o componente não cumprir todos os requisitos ele pode ser especializado por agregação, encapsulamento ou cópia e modificação, que será designada por igualdade restrita [OHDB92, ZW95]. Com o objectivo de minimizar o esforço de especialização torna-se necessário determinar uma medida desse esforço para encontrar o componente que mais se aproxima dos requisitos. Para tal define-se uma função $D : C \to \mathbb{N}$ que calcula a distância dos requisitos a cada um dos componentes. O componente, c, a seleccionar é $\mathit{semelhante}(D, C) = \{c \in C : \forall_{d \in C} D(d) \geq D(c)\}$. Assim, a pergunta à biblioteca de componentes é formulada através de uma função que é avaliada para cada componente da biblioteca pelo processo de selecção.

A função que determina a distância entre requisitos e descrições dos componentes é definida a partir do predicado. Uma vez que a selecção é feita com base em identificadores, o predicado é constituído por comparações entre nomes de identificadores. Neste ponto assumiremos que cada comparação é exacta, ou seja, só é considerada verdadeira se o nome expresso no requisito for igual ao nome descrito na classificação do componente.

Assim, cada comparação produz um resultado booleano. A comparação não é efectuada ao componente na sua globalidade mas tem por base um conjunto de características designadas por $f \in \Psi$, onde Ψ é o espaço das características consideradas relevantes. Uma função, $D_f : \Psi \rightarrow \mathbb{N}$, para calcular a proximidade pode ser, no caso mais simples, a soma das igualdades dos nomes pretendidos,

$$D(c) = \sum_{f \in \Psi} D_f(c.f)$$

Assim, quanto maior for o número de igualdades encontradas, mais próximo está o componente dos requisitos em questão, ou seja, menor será a distância. Esta aproximação simplista tem sido usada com sucesso em situações em que não se exige elevada precisão em alguns métodos léxicos e estruturais.

Para refinar as buscas e, consequentemente, aumentar a precisão da selecção é necessário ter em conta que a existência de certos nomes, na descrição do componente, tem maior importância que a presença de outros nomes. Nesse sentido, diversos métodos têm sugerido a introdução de pesos que permitem aumentar ou reduzir a importância de determinada igualdade face às restantes [PD91a, MMM97]. A função distância passa a ser

$$D(c) = \sum_{f \in \Psi} P_f \times D_f(c.f), \tag{5.1}$$

onde $P_f : \Psi \rightarrow \mathbb{N}$ é peso da característica $f \in \Psi$. A utilização de pesos permite influenciar o cálculo da distância da mesma forma, ou seja, aumentar ou reduzir a importância de cada comparação. A utilização de factores negativos (correspondendo a valores elevados) permite, inclusivamente, influenciar a rejeição de determinados componentes, nos quais se considere que a presença de determinado nome dificulta a sua adaptação a determinadas situações. Exemplos imediatos deste tipo de situações ligam-se com a portabilidade dos componentes, como, por exemplo, a presença do nome "`fork`" num componente a ser reutilizado numa aplicação de um sistema operativo monoprocesso como o **MS-DOS**.

Como estamos a minimizar a distância entre os requisitos e as descrições de cada um dos componentes da biblioteca, os melhores candidatos encontram-se para valores próximos de zero. A função distância pode ser escolhida de tal forma que produza valores positivos quando o componente é mais genérico que os requisitos e valores negativos no

caso de o componente ser mais restrito. Desta forma, é possível ordenar os componentes segundo o tipo de igualdade, genérica ou restrita, dependendo do tipo de especialização que se pretende realizar. Restrições quanto ao tipo de especialização surgem quando a linguagem a utilizar não suporta certos tipos, ou por limitações de espaço ou por eficiência. Contudo, o facto da distância ser maior ou menor não reflecte um maior ou menor esforço. Por exemplo, se a especialização pode ser efectuada com recurso a técnicas de agregação com outros componentes já disponíveis, para distâncias de valores negativos, o custo pode ser o mesmo que através da parametrização de um componente mais genérico, para valores positivos da distância.

Até ao momento, as considerações feitas sobre o mecanismo de selecção não tiram partido do facto de os nomes se encontrarem organizados em hierarquias. A importância relativa de um mesmo nome existir próximo da base ou dos extremos da hierarquia não é tido em conta. Na realidade, mecanismos com as mesmas características podem ser utilizados sobre classificações efectuadas com base em pares nome-valor ou mesmo, com algumas restrições em facetas. Na secção 5.1.3 é apresentada a forma de combinar as igualdades descritas nesta secção com determinadas partes da hierarquia de nomes.

Comparações difusas

Até ao momento consideraram-se todas as comparações como devolvendo resultados booleanos. No entanto, uma lógica de dois valores não fornece uma granularidade suficiente para captar a natureza imprecisa, e até ambígua, do processo de selecção de componentes. Embora seja desejável recorrer a representações quantitativas rigorosas, quer para classificar os componentes, quer para descrever os requisitos, tal não é geralmente possível. Para mais, nos casos em que se utilizam apenas características funcionais descritas em termos matemáticos, a determinação de igualdades não só não é trivial como é impossível de verificar em muitas situações reais [ZW97, Zar96]. Para mais, a determinação de igualdades incompletas e o cálculo de proximidades é ainda mais complexo.

A utilização de valores qualitativos permite avaliar com maior facilidade situações de semelhança. Por outro lado a percepção humana é difusa, recorrendo a representações

simplificadas obtidas através da remoção de certos pormenores [SM95]. A percepção humana é importante na especificação dos requisitos, onde conceitos como "`rápido`" são conceitos difusos. Assim, os custos de adaptação dependem directamente da percepção humana. Se houver um componente cuja distância se apresente superior a outro, mas para o qual o utilizador tem presente a forma de o alterar, então o custo real da adaptação poderá ser inferior ao do outro componente. As comparações difusas são especialmente importantes por considerarem as características qualitativas da percepção humana que representam a metade esquerda do processo de selecção tal como foi apresentada na figura 5.1.

Do ponto de vista do sistema proposto, a característica determinante na consideração de igualdades difusas provém do facto de os nomes serem menos precisos que os números. As vantagens e desvantagens desta característica foram já analisadas no capítulo 3. No caso dos nomes, as comparações difusas têm por base um conjunto de nomes considerados mais ou menos equivalentes, ou seja, um dicionário de sinónimos.

Dicionários de sinónimos A existência de sinónimos obriga a considerar igualdades difusas. Estas igualdades devem ter em conta a semelhança entre os termos, uma vez que poucos são os casos de sinónimos perfeitos. Por sinónimos perfeitos entendam-se dois, ou mais, nomes que podem ser utilizados apenas nas mesmas situações, não podendo nenhum deles ser utilizado num contexto em que os restantes também não possam. Na maioria dos casos existem dois ou mais nomes que podem ser utilizados numa mesma situação embora cada um deles possa ser utilizado noutros contextos. Se considerarmos a gama de situações a que cada nome se aplica verificamos que existem sobreposições de situações e situações disjuntas. É a existência de situações de sobreposições que os transforma em sinónimos.

A determinação de igualdades de nomes no processo de selecção é feita entre dois nomes, o nome proposto nos requisitos e o nome existente na descrição do componente. Se representarmos cada um desses dois nomes por um conjunto de situações em que se aplicam, podemos ter sobreposições que vão desde intersecções vazias até à inclusão ou coincidência total. Para poder descrever as diversas situações de semelhança torna-se necessário construir um dicionário de sinónimos. A cada sinónimo associa-se um peso

que quantifica o grau de semelhança entre os nomes em questão. Uma solução muito utilizada consiste na determinação de um limiar de igualdade, valor a partir do qual os dois nomes são considerados sinónimos [MMM97, PD91a, PDF87].

No sistema proposto nesta dissertação, devido à forma como a distância é calculada, as igualdades entre os nomes não necessitam ser booleanas, pelo que os pesos podem ser tidos em conta na expressão de selecção. Estes pesos, cujos valores variam de zero (disjunção total) até um (inclusão ou coincidência total), são depois afectados do factor mutiplicativo que determina a importância do nome original nos requisitos. Assim, um nome que não seja especialmente importante na selecção do componente a reutilizar só terá alguma relevância, no cálculo da distância final, se o componente analisado contiver o próprio nome ou um sinónimo muito semelhante. Por outro lado, um nome que seja determinante na selecção do componente a reutilizar deverá ter um factor multiplicativo elevado, por forma a que possam ser considerados componentes que utilizam sinónimos aproximados.

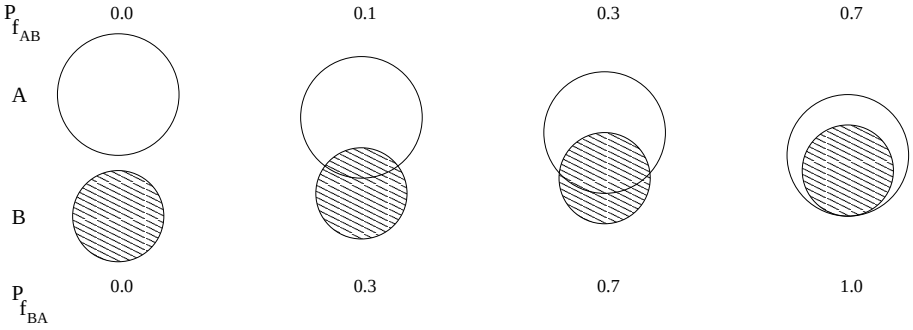


Figura 5.2: Pesos de semelhança entre sinónimos entre A e B.

Como a selecção com base em nomes de identificadores parte do pressuposto que a sua escolha foi esclarecida, criteriosa e rigorosa, a inclusão de sinónimos é importante nos casos em que a escolha dos nomes não obedece a esses mesmos critérios. Infelizmente, tais casos surgem devido a limitações impostas pela linguagem na escolha dos nomes ou no seu comprimento máximo, pela existência de conflitos com outros nomes já existentes na linguagem ou em componentes com os quais o componente em questão interagiu. Menos defensáveis são os casos que resultam da inexperiência do programador ou analista, mas que não podem deixar de ser considerados.

Expressões regulares A utilização de dicionários de sinónimos permite associar graus de semelhança entre palavras graficamente distintas. No entanto, em muitas línguas as palavras utilizadas como identificadores apresentam pequenas diferenças gráficas entre si. Estas diferenças têm origem na escolha de um género – feminino ou masculino –, número – plural ou singular – ou tempo do verbo distinto. Estas pequenas diferenças gráficas, pelo seu elevado número e diferentes combinações possíveis, não se encontram cobertas nos dicionários. De um ponto de vista do processo de selecção estas diferenças podem ser tratadas como igualdades por não traduzirem alterações significativas nos componentes que descrevem. Além disso, a escolha de palavras não é, geralmente [ED97, DFF97], suficientemente cuidadosa para reflectir no componente as diferenças semânticas da sua utilização.

A utilização de expressões regulares permite tratar de uma forma acessível e compacta este tipo de igualdades.

Exemplo 5.1 *A escolha de termos como* `cliente` *ou* `clientes` *na descrição dos titulares de uma* `Conta`*, no exemplo 3.1, pode ter origem na opção entre tipos de dados vector ou lista. Já a equivalência entre* `titular` *e* `cliente` *é específica do exemplo em causa e é tratada como um sinónimo aproximado, cujo grau de sinonímia pode ser considerado elevado neste exemplo. A utilização da expressão regular* `"deposit*"` *permite seleccionar, quer tempos de verbo como* `depositar` *ou* `deposite`*, quer substantivos como* `depositante`*. Neste caso, admitindo uma correcta escolha de termos, a selecção pode ser feita com base no atributo ou na operação, o que se pode traduzir numa precisão mais baixa. Se for o caso, será necessário refinar a busca com outros critérios para eliminar os componentes com* `"deposit*"` *que não sejam pretendidos.*

5.1.3 Operações de selecção

As hierarquias resultantes da classificação dos componentes, em especial se descritas com alguma precisão, produzem estruturas manifestamente grandes para permitirem uma análise manual e exaustiva, cuidada e esclarecida. A utilização de ferramentas gráficas permite apreender alguma informação com maior facilidade, mas a capacidade

humana de absorção e processamento dessa mesma informação permanece limitada. Por outro lado, a informação relevante para determinada selecção restringe-se, normalmente, a um subconjunto de dados que se relacionam com os critérios de busca. Assim, é possível, com base nesses critérios produzir hierarquias de menores dimensões, contendo apenas informação relacionada com os critérios em análise, facilitando a avaliação dos resultados de cada procura.

As operações de selecção permitem gerar sub-árvores que podem ser utilizadas para efectuar comparações de nomes em determinadas regiões da hierarquia. Inversamente, as comparações de nomes podem ser utilizadas para gerar sub-árvores contendo apenas os identificadores escolhidos na sua posição original na hierarquia. Como resultado das operações de selecção obtém-se uma sub-árvore que, em conjunto com o resultado da função distância, vai permitir avaliar os componentes. Se, por um lado, a distância entre requisitos e componentes constitui uma boa base para a ordenação dos mesmos, por outro lado, a sub-árvore resultante é essencial na avaliação dos custos de especialização necessários. Desta forma, a escolha do componente a reutilizar pode recair num componente que não tenha, segundo a função distância escolhida, a maior semelhança com os requisitos.

Granulação

A grande quantidade de informação que pode ser representada na hierarquia traduz-se no registo de muitos pormenores que se revelam irrelevantes em muitas buscas. Numa fase inicial do processo de selecção pretende-se obter uma primeira escolha que contenha todos os bons candidatos, ou seja, uma elevada taxa de recuperação. Embora o número de candidatos iniciais possa ser elevado, o importante nesta fase é não rejeitar nenhum dos componentes que apresente um conjunto de características mínimas concordantes com os requisitos. Nesta fase inicial, poderia recorrer-se a um outro método de selecção, mas tal solução obriga quase invariavelmente a segundo processo de classificação. Esta solução gastaria mais espaço por componente, para guardar a informação de classificação adicional e poderia recorrer a métodos de classificação não automáticos.

Uma solução possível para efectuar uma primeira selecção, com uma elevada taxa de recuperação, consiste em utilizar apenas os elementos mais genéricos e abstractos da descrição do componente. Na representação da hierarquia, construída pelo processo de classificação, tais elementos encontram-se nos primeiros níveis da hierarquia. Assim, uma forma simples de generalizar a representação de um componente, correndo o risco de representar vários componentes com a mesma descrição, consiste na eliminação dos níveis mais específicos da hierarquia. Desta forma, a operação de granulação produz uma hierarquia igual à original, mas contendo apenas os primeiros níveis. O número de níveis é parametrizável por forma a permitir controlar a granularidade das descrições obtidas e, consequentemente, o número de componentes recuperados na primeira selecção. Se este número for muito elevado então o número de níveis escolhido é tão reduzido que produz descrições quase iguais para todos os componentes. No limite, se a granularidade escolhida for nula, todos os componentes são descritos pela mesma hierarquia, que contém apenas a raiz.

A operação de granulação consiste em seleccionar todos os vértices e arcos contidos nos caminhos de comprimento não superior a N, onde N representa o nível de granularidade pretendido.

Definição 5.1 (granulação) *A operação de* `granulação:` $\mathbb{N} \times \mathcal{G} \rightarrow \mathcal{G}$*, definida como* `granulação`$(N, (V, \Sigma, E)) = (V', \Sigma, E')$*, onde*

$$\begin{aligned} V' &= \{v \in V : \ \#c(q_0, v) \leq N\} \\ E' &= \{e \in E : \ sucessor(c) = v \wedge v \in V'\} \ . \end{aligned}$$

Exemplo 5.2 *Considerando a hierarquia apresentada na figura 4.2 uma operação de granulação para comprimentos unitários permite obter o nome das classes do* `minibanco`*. Por outro lado, uma operação de granulação que seleccione os caminhos de comprimento não superior a 2, na figura 4.4, permite obter os nomes das variáveis e funções da classe* `Conta`*.*

Particularização

Uma vista parcial é uma representação incompleta, ou abstracção, de determinada entidade, captando apenas as características consideradas relevantes nessa vista. O conceito

de vista é extremamente importante na selecção de componentes pois permite focar a atenção em características consideradas importantes em determinada situação. Uma vista isola um conjunto de características associadas da restante descrição do componente. A constatação que determinado nome existe numa determinada vista fornece mais informação que o simples facto de esse mesmo nome aparecer referido algures no componente.

A utilização de hierarquias como formas de descrever os componentes oferece uma forma natural de descrever e combinar as diversas vistas de um componente. Cada ramo da hierarquia, tal como foi descrita em 4.3.2, é disjunto dos restantes e só pode ser acedido através do arco antecessor do vértice que lhe dá origem. Desta forma, cada ramo pode ser univocamente identificado pelo caminho que termina no vértice onde o ramo tem origem. O caminho, além de servir de identificador do ramo, faculta informação sobre o contexto em que os termos do ramo estão a ser utilizados, através dos nomes que o constituem. No limite, se o vértice for a raiz então a vista obtida é a completa descrição do componente e a árvore obtida é a mesma.

A operação de particularização consiste em seleccionar todos os vértices e arcos contidos nos caminhos que partem de um dado vértice v.

Definição 5.2 (particularização) *A operação de* `particularização`: $\mathcal{V} \times \mathcal{G} \rightarrow \mathcal{G}$, *definida como* `particularização`$(v_1, (V, \Sigma, E)) = (V', \Sigma, E')$, *onde*

$$\begin{aligned} V' &= \{v \in V : \ v \in c(q_0, v_1) \vee v \in c(v_1, v)\} \\ E' &= \{e \in E : \ sucessor(c) = v \wedge v \in V'\} \ . \end{aligned}$$

Exemplo 5.3 *A particularização da hierarquia da figura 4.2, com base no vértice definido pela sequência* `"minibanco", "Conta"`, *permite analisar isoladamente a classe* `Conta` *e, nomeadamente, comparar com a figura 4.4.*

Localização

Quando o número de identificadores que nos conduzem desde a origem do sistema até um dado identificador é elevado, ou seja o caminho até esse identificador é bastante

grande, temos uma informação detalhada sobre o seu contexto de utilização. Por outro lado, se esse caminho for curto a informação disponibilizada pode ser manifestamente insuficiente. Neste último caso poderá ser necessário utilizar operações adicionais para determinar o contexto de utilização do identificador. Uma forma de determinar a informação de contexto recorre a uma operação de localização para obter o conjunto de identificadores que são usados no mesmo contexto. Estes identificadores podem permitir inferir, através do ambiente, qual a utilização dada ao identificador em questão. Se nenhuma desta informação for suficiente para determinar a funcionalidade do identificador, então será necessário recorrer directamente ao documento. Este documento pode ser o código ou uma fase anterior de desenvolvimento, como a análise ou o desenho.

Quando a selecção tem por base nomes com uma grande gama de aplicações, é natural que estes nomes surjam em contextos diferentes com significados diferentes. Um dos problemas de efectuar classificações com base no número de ocorrências (ver secção 2.2.3) reside no facto de certos nomes serem suficientemente abrangentes para não poderem ser utilizados como base de classificação. A localização deste tipo de nomes pode fornecer informação suficiente quanto à forma como são utilizados, devido ao contexto onde surgem, e daí inferir da sua real importância na selecção do componente em questão.

A operação de localização consiste em seleccionar todos os vértices e arcos contidos nos caminhos cujo último arco tem um dado nome.

Definição 5.3 (localização) *A operação de* `localização:` $\Sigma \times \mathcal{G} \rightarrow \mathcal{G}$*, definida como* `localização`$(s, (V, \Sigma, E)) = (V', \Sigma, E')$*, onde*

$$\begin{aligned} V' &= \{v \in V : \forall_{w \in V}, v \in c(q_0, w) \wedge \mathit{antecessor}(e) = w \wedge \mathit{nome}(e) = s\} \\ E' &= \{e \in E : \mathit{sucessor}(c) = v \wedge v \in V'\} \ . \end{aligned}$$

Exemplo 5.4 *A localização do nome* `Conta` *na hierarquia da figura 4.2 permite verificar que este nome designa uma classe, uma associação simples nas classes* `Ordem` *e* `Prazo`*, e uma associação de* `contas` *na classe* `Cliente`*.*

Qualificação

Além da informação de contexto poderemos obter informação mais complexa, nomeadamente recorrendo a comparações entre conjuntos de identificadores. A comparação dos atributos de um determinado identificador em dois contextos, por exemplo duas vistas, permite determinar se alguma informação está a ser perdida ou adicionada. Neste último caso, é necessário verificar se o aumento do número de atributos contraria os requisitos ou não. É natural que com a evolução do sistema, e à medida que opções vão sendo tomadas, o número de atributos vá reflectindo esse enriquecimento por um aumento significativo do seu número. Uma operação que pode ser efectuada é a operação de qualificação, que permite obter um conjunto de atributos para esse identificador. Pode-se desta forma determinar através dos seus componentes qual a sua utilização.

A informação de localização de um determinado identificador é importante para a sua compreensão. Igualmente de grande importância são os identificadores que com ele se relacionam mais de perto. Neste conjunto incluem-se os identificadores que têm origem no mesmo vértice que o identificador dado, os seus irmãos, e os identificadores que têm origem no vértice de destino do identificador dado, os seus filhos. Os filhos caracterizam directamente o identificador quanto ao seu tipo e comportamento, enquanto os irmãos definem as associações do componente e o ambiente onde se inserem (o contexto de utilização).

A operação de qualificação consiste em seleccionar todos os vértices e arcos contidos nos caminhos em que apenas o último arco difere de um caminho até um vértice dado.

Definição 5.4 (qualificação) *A operação de* `qualificação:` $V \times \mathcal{G} \rightarrow \mathcal{G}$*, definda como* `qualificação`$(v_1, (V, \Sigma, E)) = (V', \Sigma, E')$*, onde*

$$\begin{aligned} V' &= \{v \in V : \ v \in c(q_0, v_1) \vee \#c(v_1, v) = 1\} \\ E' &= \{e \in E : \ sucessor(c) = v \wedge v \in V'\} \ . \end{aligned}$$

Exemplo 5.5 *Aproveitando o resultado do exemplo de localização anterior, é possível, através de uma operação de qualificação, verificar que a utilização de* `Conta` *na associação de* `contas` *na classe* `Cliente` *é uma agregração.*

União e Diferenciação

Muitas outras operações podem ser definidas por forma a auxiliar a identificação de nomes nas hierarquias e o processo de selecção que nelas se baseia. No entanto, as operações acima descritas surgem como as mais importantes na redução da hierarquia a dimensões que possam ser directamente analisadas pelo utilizador, sempre que necessário. A futura experiência com a utilização do sistema proposto poderá evidenciar a necessidade de criar novas operações, ou mesmo de descontinuar alguma das acima descritas. Porém, é igualmente importante poder relacionar duas ou mais árvores para poder dispor de uma visão de conjunto. De entre as operações possíveis duas apresentam-se como essenciais no processo de selecção: a união e a diferenciação.

Uma vez que uma árvore pode ser definida como a soma dos caminhos para todos os seus nós (ver secção 4.3.2), as operações de união e diferenciação podem ser definidas em termos de caminhos. Por união entenda-se uma árvore que contém todos os caminhos existentes em ambas as árvores que lhe deram origem. Se pensarmos em termos de função de transição, deve ser possível aplicá-la na árvore obtida, desde a raiz, se essa aplicação for possível em pelo menos uma das árvores que lhe deu origem. Esta união, ou soma de árvores, permite recombinar um conjunto de vistas obtidas por operações de particularização, ou juntar as descrições de dois ou mais componentes para verificar se a sua agregação obedece aos requisitos.

Definição 5.5 (união) *A operação de* `união:` $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$, *definida como* $\texttt{união}((V_1, \Sigma, E_1), (V_2, \Sigma, E_2)) = (V_1 \cup V_2, \Sigma, E_1 \cup E_2)$.

A operação de diferenciação é muito importante quando se pretende efectuar uma análise comparativa ou evolutiva. A diferenciação entre duas árvores permite evidenciar os elementos que existem numa árvore mas não existem na outra. A árvore obtida por diferenciação contém apenas os caminhos que existem numa árvore mas não existem na outra. Tal como a subtracção de valores numéricos, a operação de diferenciação não é comutativa, e uma análise de ambas as hipóteses pode revelar dados importantes. A diferença simétrica, entendida como a soma entre os resultados das duas operações de

diferenciação, pode ser conseguida por composição das operações com o auxílio de uma linguagem de interacção.

Uma outra aplicação da operação de diferenciação consiste em descrever os requisitos como uma árvore de nomes, tal como se os requisitos fossem também eles classificados segundo o mesmo processo. Nesta hipótese, os requisitos seriam representados por um componente imaginário, permitindo determinar as diferenças entre componentes e requisitos de uma forma alternativa. Esta forma alternativa tem a vantagem de comparar os nomes nos respectivos contextos, mas pequenas alterações no processo de classificação podem colocar nomes em posições diferentes, podendo comprometer o resultado do processo de selecção.

Definição 5.6 (diferenciação) *A operação de* `diferenciação`: $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$, *definida como* `diferenciação`$((V_1, \Sigma, E_1), (V_2, \Sigma, E_2)) = (V', \Sigma, E')$, *onde*

$$\begin{aligned} V' &= \{v \in V_1 : \forall_{w \in V_1}, w \notin V_2 \wedge v \in c(q_0, w)\} \\ E' &= \{e \in E : \mathit{sucessor}(c) = v \wedge v \in V'\} \ . \end{aligned}$$

Exemplo 5.6 *Considere-se uma hierarquia A com apenas dois ramos designados por X e Y, e uma segunda hierarquia B com apenas dois ramos designados por X e Z. A operação de diferenciação entre A e B permite obter o ramo Y, enquanto a diferenciação entre B e A permite obter o ramo Z.*

5.1.4 Linguagem de interacção

Enquanto alguns modelos optam por oferecer um elevado número de construções e de operações, no sistema proposto optou-se por oferecer um conjunto de primitivas que podem ser encadeadas. É preferível oferecer boas primitivas, a partir das quais se possam sintetizar as soluções pretendidas, que disponibilizar um conjunto limitado de soluções acabadas. Desta forma, embora parte do trabalho possa recair sobre o utilizador, permite-se uma maior flexibilidade. Uma pequena linguagem de apoio – **MAP** – permite definir sequências – pequenas rotinas – que podem ser posteriormente invocadas.

Exemplo 5.7 *A determinação dos filhos de um dado vértice, referido na apresentação da operação de qualificação, pode ser obtida pela aplicação de uma operação de particularização do referido vértice seguida de uma operação de granulação unitária.*

A linguagem permite combinar as operações apresentadas nesta secção para obter descrições contendo apenas a informação relevante para o caso a seleccionar. A árvore obtida pode depois ser analisada através de ferramentas de navegação que permitem, sempre que necessário, visualizar o documento original na posição onde o identificador em causa foi extraído. Assim, além de ser possível construir perguntas complexas para estabelecer uma ordenação de componentes, é ainda disponibilizado acesso aos componentes originais. Desta forma, é possível efectuar uma escolha esclarecida, minimizando as tendências particulares de cada método de selecção com uma inspecção manual. Esta inspecção manual, para ser viável deve-se basear num número reduzido de componentes, devendo ser tipicamente inferior a cinco. Notar que a inspecção manual não é utilizada como substituição do método de selecção mas pretende permitir ao utilizador escolher um componente que ele reconheça ser de mais fácil integração.

Em vez de definir uma linguagem específica para controlar o processo de selecção optou-se por associar os tipos de dados e operações necessárias à linguagem de integração que se descreve no capítulo 6 e exemplifica no capítulo 7. Esta linguagem recorre a uma estrutura de dados igualmente hierárquica pelo que a integração é simples. Como principal vantagem resulta o facto de se dispor de uma linguagem mais rica que permite fazer buscas mais complexas, que podem incluir o teste dos próprios componentes se estes estiverem já integrados no sistema.

5.2 Apreciação do processo de selecção

Para responder a uma determinada dúvida do utilizador é necessário extrair um subconjunto de identificadores que permita comparações claras e conclusões esclarecidas. Para tal tornam-se necessárias várias iterações até que se atinjam conclusões fiáveis. No

entanto, a análise do código propriamente dito é, frequentemente, a solução apresentada por diversos métodos de selecção. Evita-se, assim, ter de conhecer, no processo de selecção, diversas linguagens de especificação e de programação, bem como os estilos de cada utilizador.

Nem toda a informação do documento original é convertida na representação de identificadores. Em primeiro lugar, parte da informação não está directamente associada aos identificadores, como é o caso de constantes. Por outro lado, o conversor utilizado pode não extrair a totalidade da informação, quer por limitações do conversor, quer por opções de análise. Assim, poderá, por vezes, ser necessário recorrer ao código, mas apenas em situações extremas.

A hierarquia de identificadores funciona como uma assinatura complexa do componente. Aliás esta técnica é uma extensão a métodos de assinaturas, que usam técnicas mais simples com base em menos informação. Contudo, estas assinaturas complexas conseguem reflectir muitas das características, quer estáticas quer dinâmicas, do componente que modelam.

O sistema proposto nesta dissertação não foi desenhado para oferecer resultados quantitativos. Conclusões que sejam tiradas com base na dimensão da árvore gerada ou no comprimento do caminho de um identificador não representam métricas válidas. Estas observações podem depender de factores como estilo de programação ou a linguagem utilizada. A aproximação proposta é baseada numa análise manual sendo precedida de um certo automatismo para seleccionar a informação relevante, evitando a consulta exaustiva dos documentos. Como qualquer sistema que seja dependente da experiência humana, o seu sucesso está directamente ligado com o treino e a capacidade de interpretação do engenheiro de software que o utiliza [McF89]. Por outro lado, oferece um grande manancial de informação de uma forma compacta e que pode ser manipulada de forma semi-automática.

Os resultados do processo de selecção podem ser avaliados segundo três eixos principais: descritivo, comparativo e evolutivo. Cada um destes eixos reflecte a preocupação principal do utilizador na escolha do componente a reutilizar.

Eixo descritivo O eixo descritivo está directamente ligado à procura de uma solução para o problema em questão, aceitando-se numa primeira aproximação um componente que esteja conforme um conjunto de requisitos mínimos.

Eixo comparativo O eixo comparativo procura, por comparação, obter melhores soluções para um dado problema, que cumpram um conjunto de requisitos adicionais, que sejam considerados de menor importância ou mesmo facultativos.

Eixo evolutivo O eixo evolutivo prende-se com a análise das diversas fases do processo de desenvolvimento do componente, podendo tornar mais atractiva a reutilização de um componente na fase de desenho que o equivalente na fase de codificação. Tal opção pode dever-se a dependências da linguagem de codificação ou de opções de codificação que não se adaptem às necessidades.

5.2.1 Análise descritiva

Na escrita de documentos em linguagem natural procura-se diversificar a linguagem para não a tornar repetitiva. Tal facto obriga quem escreve à procura de sinónimos, não só por razões estéticas e de correcção da língua, mas também como resultado de escolhas subjectivas de termos por parte do autor. A escolha de identificadores como elemento de classificação tem a principal vantagem de já estar estereotipado. Devido às regras de visibilidade das linguagens de especificação e programação, o recurso a sinónimos torna-se menos frequente. A boa programação, que inclui uma escolha criteriosa de nomes para os identificadores ou utilização de poucas variáveis globais, reduz a necessidade da utilização de sinónimos na produção do componente. Desta forma, a necessidade de fazer as conversões inversas na classificação ou selecção dos mesmos documentos torna-se menos crítica.

A utilização de hierarquias de nomes permite reter, no processo de classificação, muita informação que pode depois ser analisada no processo de selecção. Quanto maior for a quantidade de informação disponível maior é o tempo necessário para analisar cada componente. O recurso a estruturas hierárquicas torna possível efectuar buscas a diversos níveis de profundidade, e consequentemente de pormenor da descrição, permitindo

um balanceamento entre velocidade e precisão. Numa primeira selecção, optar-se-á por uma baixa profundidade por forma a rejeitar a grande maioria dos componentes. Nas escolhas seguintes, à medida que o número de componentes que subsiste vai diminuindo, aumenta-se a profundidade dos algoritmos de busca. Consegue-se, assim, com um mesmo método de classificação efectuar selecções a diversos níveis de detalhe. Caso se justifique, e se o número de componentes candidatos à selecção for pequeno, o algoritmo de busca pode inclusivamente aceder ao conteúdo dos próprios componentes que serviram de base à classificação.

A escolha de uma estrutura hierárquica de identificadores surge, do ponto de vista descritivo, atractiva face a outros métodos de classificação e selecção. Por exemplo a utilização de facetas, um dos métodos mais divulgados e de maior sucesso, além de ser geralmente baseado em classificações manuais, tem uma precisão limitada pela dimensão do vocabulário utilizado. Como consequência, na abundância de componentes surgem vários componentes distintos com a mesma classificação. O seu comportamente é contudo muito satisfatório em bibliotecas com um número médio de componentes, todos eles com funções bem distintas dos restantes. Notar que a hierarquia proposta pode ser enriquecida com facetas, que correspondem a ramos com nomes pré-definidos, permitindo superar qualquer insuficiência do método proposto em relação a esquemas baseados em facetas.

5.2.2 Análise comparativa

De um ponto de vista comparativo procura-se determinar pequenas diferenças entre dois ou mais componentes que apresentam as mesmas características genéricas. Refira-se que apenas os métodos formais e alguns métodos estruturais permitem efectuar este tipo de análise. Na maioria dos métodos de selecção, o utilizador é remetido para uma análise manual, por inspecção directa, dos componentes recuperados pelo método de selecção utilizado.

A utilização de um vocabulário não controlado é essencial neste tipo de análise pois a descrição do componente é efectuada com os próprios nomes utilizados no seu desenvolvimento. O maior problema desta aproximação recai sobre uma má escolha dos

nomes que, no entanto, está associada a componentes de má qualidade e cujo interesse em reutilizar é mais reduzido.

Também a utilização de hierarquias de nomes facilita, através de operações de particularização, concentrar a análise em determinadas vistas do componente que possam surgir mais relevantes para a comparação. A diferenciação, como seria de esperar, tem um papel importante ao evidenciar as diferenças entre vistas diferentes, embora a utilização de vocabulários não controlados possa dificultar este tipo de análise. A comparação entre diferenças cruzadas (A-B e B-A) possibilita, por vezes, a identificação de certas discrepâncias entre os componentes (ver exemplo 5.6).

5.2.3 Análise evolutiva

Uma análise evolutiva tem a particularidade de não ser dependente do vocabulário, uma vez que este é escolhido nas fases iniciais do desenvolvimento e, em princípio, mantido constante ao longo da sua evolução. A evolução do componente pode ser vista de duas perspectivas: uma perspectiva de desenvolvimento e uma perspectiva de alterações. Na perspectiva de desenvolvimento, o componente, e a respectiva descrição utilizada na classificação, vão variando desde a fase de especificação dos requisitos até à codificação e ao teste do componente. Na perspectiva de alterações, o componente vai sofrendo modificações resultantes da identificação de erros ou da alteração dos objectivos a atingir.

Numa análise evolutiva qualquer das duas perspectivas traduz-se em pequenas alterações da sua descrição pelo que a sua análise é semelhante à análise comparativa. A principal diferença, além do vocabulário ser aproximadamente constante, consiste no aumento da informação associada à descrição do componente que vai surgindo com a evolução deste. Este aumento de informação resulta de uma maior concretização, no caso da perspectiva de desenvolvimento, e do facto de as alterações serem quase sempre efectuadas por acréscimo. Desta forma, nestes casos, as diferenças cruzadas têm significados específicos. Nomeadamente, enquanto a diferenciação entre uma descrição posterior e uma anterior apresenta grandes diferenças dificultando uma análise objectiva, a diferenciação contrária evidencia a supressão de determinados identificadores em

descrições posteriores. A supressão de informação, como consequência quer do desenvolvimento, quer das alterações, resulta de erros ou de opções que transformam a gama de aplicações do componente e, consequentemente, as situações em que este pode ser reutilizado.

5.3 Recuperação de componentes

Tendo como base as operações oferecidas, a linguagem permite formular perguntas que escolhem os componentes com base em determinações de semelhança. O processo de recuperação de componentes analisa a relação entre os componentes escolhidos e as perguntas efectuadas. Desta análise surge uma metodologia de aplicação do método de selecção que permite explorar de uma forma óptima os componentes classificados.

A maioria dos métodos apresentados em 2.2.2 efectua uma única busca, apresentando como resultado um número de componentes recuperados tipicamente inferior a dez, que podem ser escolhidos manualmente por inspecção directa. Tal pressuposto baseia-se no facto de a biblioteca ter por base componentes com funcionalidades disjuntas e classificados segundo uma taxinomia sistemática, como descrito em 1.3.2. A utilização de métodos de classificação automática raramente produz uma taxinomia sistemática. Além disso, a rápida distribuição de componentes, em especial via internet, gera uma abundância de componentes com características muito semelhantes. Por estas duas razões torna-se cada vez mais necessário um processo de refinamento de escolha. Para mais, a utilização de nomes de identificadores para classificar componentes é sensível à escolha dos nomes utilizados para descrever os identificadores, agravando o problema.

A classificação hierárquica de componentes com base em identificadores produz árvores de nomes que servem de base ao processo de selecção. Consequentemente, os componentes a recuperar dependem largamente de uma correcta escolha de nomes. Embora uma substituição dos nomes dos identificadores não altere a estrutura da árvore, o grau de semelhança entre o nome a procurar e o nome do identificador classificado pode variar significativamente. Este facto pode levar a que sejam rejeitados componentes que, obedecendo aos requisitos formulados, não tenham tido uma escolha cuidadosa dos

nomes dos seus identificadores. Desta forma, o número de componentes recuperados face aos disponíveis diminui, ou seja, reduz a taxa de recuperação do método. Menos importante é o facto de serem escolhidos componentes que, não obedecendo aos critérios, possuam nomes semelhantes. Nesta situação, a recuperação não é afectada pois todos os componentes desejáveis são recuperados, mas diminui a precisão do processo de recuperação pois são também recuperados componentes indesejáveis. Contudo, estes componentes indesejáveis podem ser eliminados num segundo processo de recuperação que tem apenas por base os componentes previamente recuperados.

O processo de recuperação de componentes é pois um processo iterativo em que basta ser garantida uma elevada taxa de recuperação na primeira escolha. As escolhas seguintes deverão aumentar gradualmente a precisão, sem afectar a taxa recuperação, até que um só componente seja seleccionado. Devido às necessidades distintas da primeira busca face às buscas seguintes, a primeira busca será tratada separadamente.

5.3.1 Primeira escolha

A principal preocupação ao efectuar uma primeira escolha consiste em não ignorar nenhum componente que apresente alguma característica que o possa vir a eleger como o componente seleccionado. Desta forma, a primeira escolha deverá rejeitar apenas os componentes que não apresentem nenhuma característica que obedeça, mesmo que remotamente, aos critérios de selecção formulados pelo utilizador. Deverão ser rejeitados a quase totalidade dos componentes classificados e disponíveis para selecção. A primeira escolha deverá reduzir, pelo menos, em uma ou duas ordens de grandeza o número de componentes a analisar. Ao contrário, o processo iterativo de refinamento que se segue conseguirá reduções apenas próximas de 1:2, ou seja, uma redução para metade em cada iteração.

Um método de escolha por grosso, como é o caso da primeira escolha, deverá ser insensível à grande maioria das opções de desenvolvimento do componente, reflectindo apenas características como a funcionalidade. Entre elas encontra-se a linguagem utilizada no desenvolvimento do componente e em especial pormenores sintácticos da

mesma. Note-se que o método de classificação proposto é pouco sensível à linguagem escolhida apresentando uma boa uniformidade de representação (ver secção 4.3.1).

Da mesma forma, a influência de uma escolha de nomes menos adequada deverá ser minimizada. Este facto conduz à necessidade da introdução de sinónimos, ou outros métodos de determinação de igualdades descritos em 5.1.2, com elevado grau de permissividade. O elevado grau de permissividade garante que componentes com nomes com baixo grau de semelhança com os formulados na busca sejam seleccionados. Nesta situação convém referir o bom comportamento das expressões regulares, em especial quando combinadas com sinónimos, por permitir recuperar componentes com um grau de semelhança baixo. Um efeito semelhante é obtido na utilização de ferramentas genéricas de procura em ficheiros de texto como, por exemplo, o comando `grep` do **UNIX**.

Sinónimos

A utilização de dicionários de sinónimos obriga à introdução para cada nome de todos os seus sinónimos considerados aceitáveis e respectivos pesos. Por outro lado, a não utilização de um dicionário de sinónimos previamente construído obriga à enumeração, caso a caso, dos sinónimos a considerar e seus pesos. Esta aproximação evita a construção de um dicionário de sinónimos completo apenas para avaliar de uma forma sumária o sistema proposto. Por outro lado, permite determinar, através da inclusão de um maior ou menor número de sinónimos, qual a influência dos sinónimos nas taxas de recuperação, precisão e sobreposição do processo de selecção. A não utilização de um dicionário de sinónimos facilita, para efeitos de avaliação do método proposto, a atribuição de pesos diferentes a cada sinónimo dependendo da situação. Desta forma, é possível controlar o grau de semelhança entre dois nomes, caso a caso, o que torna especialmente útil para a primeira escolha. Do acima exposto conclui-se que o método proposto necessita de afectar os pesos associados a cada sinónimo de um factor de escala multiplicativo durante a primeira escolha. É possível, desta forma, dar maior ênfase a sinónimos fracos, garantindo a elevada taxa de recuperação que se torna necessário garantir.

Métodos auxiliares

Os métodos de classificação manual, embora dispendiosos do ponto de vista de recursos humanos, surgem como bons candidatos. Os métodos descritos em 2.2.2, em especial, devido à elevada capacidade de abstracção que exigem como consequência do limitado vocabulário disponível num dicionário fixo. O processo de classificação proposto no capítulo 4 permite a inserção manual de nomes na árvore através do processo descrito em 4.2.3. Desta forma, a primeira escolha pode-se basear num método auxiliar que ofereça as características enunciadas, ou seja, uma elevada taxa de recuperação podendo ter uma baixa precisão.

Os métodos lexicais são uma opção atractiva, embora se baseiem essencialmente em processos de classificação manual. Um segundo ponto negativo, já referido em 2.2.2, reside na necessidade de alargar o dicionário e, consequentemente, reclassificar todos os componentes sempre que um número elevado de componentes sofre igual classificação. Embora o conceito de elevado varie de método para método, o problema subsiste. Nomeadamente, estudos efectuados por terceiros sugerem que métodos baseados em facetas conseguem minimizar os problemas de reclassificação [FP94, MMM95].

5.3.2 Refinamento

Após a primeira escolha torna-se necessário melhorar a precisão do processo de recuperação, por forma a poder eleger o componente que melhor se adapta aos requisitos. O processo iterativo de refinamento da busca consiste na redução da taxa de sobreposição de uma iteração para a iteração seguinte. Este processo baseia-se no aumento do número de critérios na formulação da pergunta. A informação que permite formular a nova pergunta tem por base a análise da busca anterior. Caso contrário, não haveria razão para não introduzir os novos critérios na primeira busca, pois a informação em que se baseavam já estava disponível. Assim, o processo de refinamento divide-se em análise dos resultados da recuperação anterior e formulação da nova pergunta com base nesses resultados.

A análise dos resultados obtidos num processo de recuperação baseiam-se, no sistema proposto, na observação das árvores de nomes dos componentes recuperados em consequência da pergunta formulada. Como a complexidade dos componentes tende a ser grande, bem como o número de componentes recuperados nas primeiras iterações tende a ser grande, torna-se necessário condensar a informação disponível por forma a ser rapidamente analisável. Como as árvores de nomes tendem a ter dimensões que tornam uma inspecção manual directa muito trabalhosa, torna-se necessário recorrer a um conjunto de operações, referidas em 5.1.3, que permita isolar as características relevantes.

O recurso às operações de selecção permite reduzir a dimensão das árvores, permitindo a sua análise manual. O objectivo da análise consiste em ordenar os componentes com base na distância cognitiva de cada componente aos requisitos. Os critérios de ordenação baseiam-se na exploração das características dos componentes segundo os três eixos referidos em 5.2: descritivo, comparativo e evolutivo. A análise segundo estes três eixos deverá permitir aumentar o número de critérios utilizados na formulação da pergunta e, consequentemente, aumentar a precisão do método na iteração seguinte.

Finalmente, quando o número de componentes recuperados tende a ser baixo, e consequentemente analisável manualmente, o processo iterativo termina. Este critério de paragem é importante pois, tal como em outros métodos de recuperação, não se pode garantir uma precisão infinita. O facto da precisão ser limitada implica que à medida que os critérios de selecção se tornam mais exigentes a probabilidade de erro aumenta.

Custo de adaptação

Na escolha do componente não deve ser esquecido o custo de adaptação necessário para o tornar utilizável na aplicação a desenvolver. Na prática, todos os componentes necessitam de alguma adaptação. Esta adaptação pode ser tão simples como a fixação de certos parâmetros a determinados valores, ou obrigar a uma reescrita quase total. A escolha de componentes mais genéricos em detrimento de outros mais específicos surge geralmente como mais atractiva. Embora a gama de aplicação de tais componentes seja

maior não se deve esquecer o custo de parametrização e o facto de serem, frequentemente, menos ineficientes.

A escolha ideal recai sobre o componente mais específico que cubra todas as situações de utilização numa dada aplicação. Estes componentes, por serem mais específicos, utilizam uma terminologia também mais específica. Este facto permite que, no método proposto, uma redução quer do número de sinónimos quer do grau de semelhança dos mesmos permita dar preferência a componentes mais específicos. Notar que é precisamente no controlo sobre o número de sinónimos e o respectivo grau de sinonímia face ao requisito que o método proposto adquire vantagem face a outros métodos.

5.3.3 Avaliação da recuperação

A construção de aplicações a partir de catálogos de componentes obedecendo aos critérios enunciados em 1.3.2 pode ser conseguida com recurso a métodos mais simples que o proposto. No entanto, a abundância de componentes com características funcionais semelhantes requer processos de selecção mais complexos e que permitam controlar a granularidade dos critérios e, consequentemente, a precisão do processo de recuperação. A abundância surge da utilização de vários catálogos de componentes com origens diferentes e, mais recentemente com o uso generalizado da *internet* como forma da divulgação da informação, de milhares de componentes não catalogados.

A procura de um componente que realize uma tabela de dispersão ("*hash table*") com redimensionamento dinâmico pode oferecer, numa primeira busca, várias centenas de candidatos. A menos que o processo de classificação utilizado seja muito extenso, a escolha do componente final obriga à inspecção manual de centenas de componentes. A utilização de outros métodos de classificação pode permitir restringir esse número. Por exemplo, a utilização de métricas de complexidade pode permitir eliminar os componentes que apresentem métricas baixas, assumindo que, se a complexidade é baixa, a tabela de dispersão não tem redimensionamento dinâmico. Mesmo assim, podem restar largas dezenas de componentes, que tornam o processo moroso ou incentivam uma escolha menos esclarecida, ou mesmo aleatória. O método proposto adapta-se especialmente bem a estas situações de abundância de componentes, quer por se basear

num processo de classificação automático, quer por permitir níveis de granularidade crescente.

A procura de características indicativas de um redimensionamento dinâmico de uma tabela de dispersão incluem chamadas a operações de redimensionamento de memória, ou pelo menos um maior número de chamadas a operações de reserva e libertação de memória. Embora estas chamadas sejam dependentes da linguagem, o que dificulta uma busca mais alargada, permitem uma selecção implícita da linguagem a utilizar. Se a linguagem a utilizar não deve ser um dos primeiros critérios a ter em conta, também é necessário considerar que os custos de adaptação entre linguagens diferentes podem ser significativos.

Exemplo 5.8 *Um exemplo muito simples e quase totalmente independente da linguagem reside na escolha de uma função de cálculo do valor do* `factorial` *de um número inteiro. A opção por uma função iterativa, em detrimento de soluções recursivas, pode ter origem em limitações na dimensão da pilha de dados disponível. Uma forma simples de identificar a função iterativa consiste em rejeitar todos os componentes que apresentem o nome da função –* `factorial` *– mais de uma vez num mesmo caminho.*

Do ponto de vista da análise estrutural este facto é indicativo da presença de recursividade directa. Notar que linguagens como o **Forth** recorrem a palavras-chaves específicas para indicar a recursividade, invalidando o método de busca indicado mas facilitando a rejeição de todos os componentes que contenham a referida palavra-chave.

A distinção, por exemplo, dos diversos algoritmos de ordenação pode não ser tão fácil, com a excepção do `quickSort` que também é recursivo. É claro que o essencial consiste em saber o que procurar, mas só sabendo o que se procura é que se sabe que se encontrou. Sendo assim, é difícil apresentar critérios de busca objectivos e com largo espectro de aplicação. Os critérios a utilizar baseiam-se principalmente na busca de palavras específicas, ou seus sinónimos, isoladamente ou determinados contextos. O método de classificação proposto no capítulo 4 permite descrever o contexto com base nos nomes que ocorrem no caminho para um dado nó da hierarquia. É essencialmente neste ponto que o processo de selecção se distingue dos anteriores, com a excepção dos métodos

formais, pois permite estebelecer um contexto com correspondência com o componente real sem necessidade de intervenção no processo de classificação. Finalmente, a utilização de sinónimos apenas no processo de selecção e não no processo de classificação permite um controlo do nível de estereotipagem caso a caso, permitindo um controlo sobre a precisão do método essencial em situações de abundância de componentes.

5.4 Síntese

A informação classificada não é estereotipada nem adulterada, transferindo esse tratamento para o processo de selecção onde pode ser ajustado caso a caso. A importância de cada uma das igualdades entre nomes que contribui para a selecção de um dado componente é afectada de um factor de ponderação que determina a relevância do nome. O conceito de igualdade entre dois nomes, que constitui a base do processo de selecção, pode ser calibrado por ponderação e através da determinação dos pesos de semelhança para cada um dos sinónimos.

A determinação de igualdades constitui o critério de extracção utilizado pelas operações de selecção para construir hierarquias de menores dimensões. A combinação de operações de granulação, particularização, localização, qualificação, união e diferenciação extraem apenas a informação relevante para determinada selecção. Estas hierarquias, normalizadas pelo processo de selecção, podem depois ser comparadas para determinar mais rigorosamente qual o componente que melhor se adapta à resolução do problema. A escolha final deverá ter em conta as capacidades técnicas da equipa de reutilização e as possibilidades de especialização do componente.

Capítulo 6

Linguagem de integração MAP

O desenvolvimento de aplicações com recurso à reutilização de componentes pressupõe três fases. A primeira fase consiste na selecção de componentes existentes que obedeçam aos requisitos, mesmo que parcialmente, ou ao desenvolvimento de origem se não existir nenhum componente que obedeça minimamente aos requisitos. Cada um dos componentes a reutilizar é, se necessário, adaptado de forma a cumprir totalmente os requisitos. A fase final do processo de reutilização consiste na integração dos vários componentes a reutilizar, com ou sem adaptações e desenvolvidos de origem, numa aplicação final.

O processo de integração é realizado, na sua forma mais simples, por um editor de ligações ("*linker*"), produzindo uma aplicação monolítica. Este tipo de aplicações, embora constituam a maioria, são de difícil manutenção, além de ocuparem muito espaço quer em disco quer em execução, devido à fraca partilha de código. Muito embora a utilização de bibliotecas dinâmicas tenha vindo aliviar o problema do espaço, veio introduzir um conjunto de possíveis inconsistências. Estas inconsistências devem-se ao facto de, apesar do aspecto repartido, os vários componentes não poderem ser separados ou substituídos pois a edição de ligações já foi efectuada.

A utilização de comunicações entre processos, na mesma máquina ou em máquinas diferentes, permite criar aplicações distribuídas constituídas por componentes, mas os custos de comunicação entre processos limitam o desempenho da aplicação. O recurso

a linguagens interpretadas permite o carregamento durante a execução de componentes, descritos nessas linguagens, mas penalizando significativamente o desempenho da aplicação. As linguagens que controlam o desenvolvimento e execução deste tipo de aplicações, sejam elas monoprocesso ou multiprocesso, designam-se, tal como foi definido na secção 2.3.2, por linguagens de interligação de módulos. A linguagem proposta nesta dissertação é denominada **MAP**.

6.1 Arquitectura da linguagem

A interligação de componentes baseia-se no pressuposto de que a aplicação pode ser modificada sem que os componentes tenham de ser alterados. Além disso, deve ser possível construir a aplicação com base na especificação das abstracções dos componentes, sem que exista uma realização. Claro que no momento da activação de um dado componente deverá existir pelo menos uma realização utilizável. A flexibilidade exigida por aplicação desenvolvida com base na integração de componentes obriga a considerar mecanismos de suporte. O primeiro consiste num mecanismo de activação dos componentes quer estes residam em máquinas diferentes, em processos diferentes na mesma máquina, em tarefas diferentes do mesmo processo, ou venham a ser executados sequencialmente na mesma tarefa. A resolução de referências entre os componentes – edição de ligações – deverá ser dinâmica para permitir o carregamento e a substituição interactiva dos componentes. E, finalmente, deve existir um mecanismo de suporte que permita resolver as referências entre os componentes e transferir informação para a activação do componente.

6.1.1 Definição de expectativas

A capacidade de adicionar, retirar ou substituir componentes oferece grande flexibilidade à aplicação, permitindo que a sua funcionalidade seja melhorada e modificada sem ter que reconstruir e reiniciar a aplicação. A funcionalidade da aplicação pode ser alterada de acordo com as interacções, com o ambiente que a rodeia, ou como resposta a estímulos específicos. Com a vulgarização da edição dinâmica de ligações torna-se

possível compor e modificar uma aplicação, a partir de componentes reutilizáveis de uma forma fácil. A edição dinâmica de ligações permite combinar a velocidade de execução de código máquina nativo com a flexibilidade das aplicações modificáveis.

O actual desempenho das máquinas permite desenvolver aplicações baseadas na edição dinâmica de ligações entre componentes. Embora o desempenho da aplicação final possa ser afectado, dependendo da granularidade dos componentes e da sua interdependência, passa a existir um controlo dinâmico sobre a aplicação. Inclusivamente, se os componentes que exijam maiores recursos computacionais forem executados em código máquina nativo, o desempenho total da aplicação, devido à utilização da edição dinâmica de ligações, será pouco afectado.

Interoperação

A organização de aplicações constituídas por muitos componentes, e onde podem existir diversos candidatos a determinada tarefa, levanta um conjunto adicional de problemas.

- O suporte simultâneo de várias variantes do mesmo componente obriga à compartimentação dos componentes, não só por questões organizacionais, como para evitar conflitos de nomes.
- O mecanismo de resolução de nomes deverá distinguir os diversos componentes e activar a entidade associada ao nome escolhido no componente desejado.

Definição 6.1 (integração) *A* integração *é um processo de carregamento dinâmico de componentes seguido de um processo de edição de ligações interactivo.*

Neste sentido é útil distinguir e identificar a funcionalidade de cada um destes processos individualmente. Assim, é necessário um sistema de gestão de memória flexível, mas que pode ter por base uma linguagem com reserva e libertação dinâmica de memória como, por exemplo, o **C** através das rotinas `malloc` e `free`. É, igualmente, necessário um sistema de gestão de nomes, que, para garantir a flexibilidade exigida,

obriga a definir um sistema distinto das linguagens de programação clássicas compiladas ou interpretadas. O sistema de nomes deverá descrever de uma forma expressiva a interface dos componentes que gere, por forma a ser facilmente manipulado quer pelos utilizadores, quer por ferramentas automáticas. Finalmente, é necessário um modelo de execução bem definido, preferencialmente baseado numa máquina abstracta de uma linguagem interpretada, para facilitar a especialização dos componentes com vista à sua interoperação.

Modularidade

Os módulos e os objectos são hoje os principais componentes de estruturação de muitas linguagens de programação. Na prática, eles representam organizações criadas por agregação e composição de variáveis e rotinas das linguagens de programação mais antigas. Embora a complexidade, organização e estrutura interna dos componentes possa diferir, a funcionalidade oferecida é semelhante. Uma interface de fácil compreensão, baseada numa organização simples, facilita a utilização de sistemas complexos.

O encapsulamento e abstracção têm sido aplicados com significativo sucesso em diversas linguagens e ambientes de programação. Os dados e as operações oferecidos pelas interfaces são acedidos através dos nomes que lhes são atribuídos, sendo activados pela invocação desses mesmos nomes e, se necessário, alguns argumentos adicionais. Os nomes oferecem ao utilizador uma interface amigável para referir as partes visíveis e acessíveis das entidades do sistema. Como tal, os nomes devem ser tratados pelos engenheiros de software, como os principais elementos estruturais dos ambientes e linguagens de programação. Os nomes podem ser utilizados para referir valores, operações, estruturas ou tipos de dados, sendo designados por identificadores.

Definição 6.2 (linguagem de integração) *Uma* linguagem *diz-se* de integração *se definir um modelo de cooperação a que os vários componentes devem obedecer.*

O modelo de cooperação é realizado por um conjunto de mecanismos que garantem a correcta aplicação desse modelo. O sucesso do sistema operativo **UNIX** [Ker84] baseia-se num sistema de ficheiros para dar suporte aos dados e de um modelo de processos

comunicantes através de filas FIFO. A utilização de um modelo simples com uma clara separação entre o modelo de dados e o modelo de execução facilita a especialização e a posterior integração dos componentes numa aplicação flexível e dinâmica.

A linguagem de integração proposta nesta dissertação segue o modelo descrito no capítulo 3 e pode ser entendida como uma extensão das operações de manipulação definidas no processo de selecção. De facto, a linguagem de integração efectua as operações de selecção por integração de um componente onde essas operações estão definidas. A integração do modelo de dados é extremamente simples pois a estrutura utilizada para representar os elementos da linguagem é a mesma que é utilizada para classificar os componentes para as operações de selecção. Aliás, se o modelo apresenta boas características para representar, classificar e comparar muitos componentes, também deve ser bom para organizar e integrar os componentes das aplicações. Para além do modelo de dados descrito no capítulo 3, é também necessário definir um modelo de execução para manipular e transferir os valores associados aos identificadores entre os componentes.

Para além da linguagem proposta nesta dissertação ser uma linguagem de integração, oferece algumas possibilidades de especialização. A especialização só é possível entendendo os componentes como caixas pretas, pelo que mecanismos de herança ou cópia e modificação não são possíveis. De notar que continua a ser possível efectuar especializações com base em componentes representados como caixas brancas nas linguagens originais de desenvolvimento desses componentes.

6.1.2 Modelo de dados

O modelo de dados da linguagem **MAP** baseia-se na organização dos nomes dos identificadores para formar estruturas arbitrariamente complexas. A utilização dos nomes dos identificadores como elemento privilegiado deve-se ao facto de estes serem essencialmente úteis para interface com o utilizador. A computação em si tem por base apenas os valores e não os nomes que os referem. Nas primeiras fases do desenvolvimento de software, como o nível de abstracção é ainda elevado, o projectista preocupa-se principalmente com os algoritmos e a organização da aplicação sendo quase isentas de

valores concretos. As limitações impostas por certos tipos de dados e por determinados valores concretos vão sendo introduzidos com o desenvolvimento. Muito embora alguns desses valores possam existir na especificação dos requisitos, a maioria pode ser ignorada nas primeiras fases do projecto. A separação entre nomes e valores subsiste até ao nível dos programas executáveis, onde as tabelas de símbolos existem para edição de ligações ou depuração da aplicação pelo utilizador.

Segundo o princípio da reciprocidade, toda a entidade acessível ao utilizador deve ser-lhe mostrada para observação e alteração. Várias linguagens de programação baseiam-se na utilização de um conjunto de elementos – designados por `slots` – que contêm um valor e o nome que lhe está associado [US87, Tai92, DLCP96]. Os `slots` são agregados para formar estruturas mais complexas e objectos manipuláveis pelo utilizador. Esta filosofia pressupõe que, na grande maioria, os elementos necessitam de ter um nome para serem manipulados. Embora em linguagens de programação genéricas tal posição possa ser defensável, em linguagens de integração tal filosofia não se aplica. Na aproximação proposta em **MAP**, a filosofia seguida utiliza princípios opostos à anterior, separando os nomes dos valores.

Assim, estabelece-se ao nível da programação uma clara separação entre os elementos estritamente computacionais e os elementos de interface com o utilizador. Desta forma, podem-se identificar dois domínios distintos: o domínio dos nomes e o domínio dos valores. O domínio dos nomes contém um conjunto de nomes com uma estrutura hierárquica. Alguns dos nomes referem entidades do domínio dos valores, enquanto outros existem por razões estritamente estruturais e organizativas. O domínio dos valores contém as entidades computacionais cujo formato e comportamento são descritos por um tipo representado no domínio dos nomes.

Associações principais

Uma vez que foram identificados dois domínios distintos, a linguagem **MAP** disponibiliza quatro associações:

Identificador, ou associação de nome para valor, permite designar um valor por um nome. O valor torna-se visível através da resolução do nome. O nome pode ser

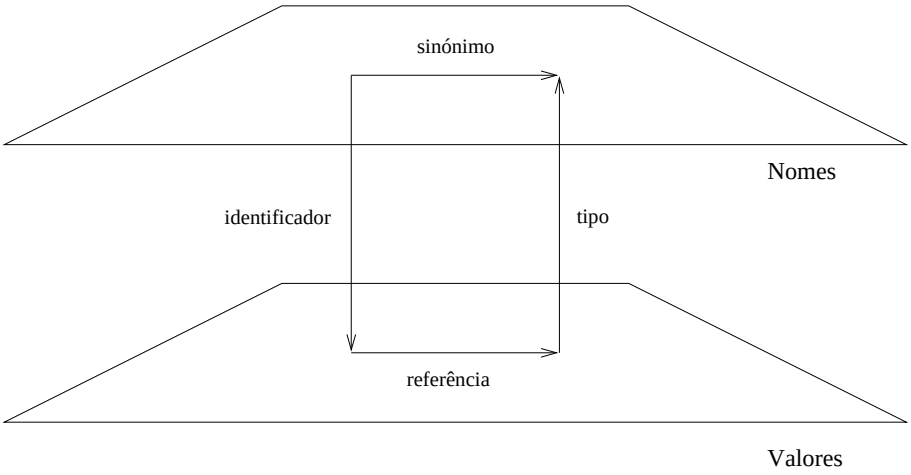


Figura 6.1: As quatro possíveis associações entre os domínios dos nomes e dos valores.

associado a variáveis, constantes, rotinas, ou tipos de dados, de forma indiferenciada do ponto de vista da estrutura do domínio dos nomes.

Referência, ou associação de valor para valor, desempenha um papel idêntico ao dos apontadores nas linguagens de programação comuns. Do ponto de vista da integração, as referências são elementos internos dos componentes e das linguagens utilizadas para os descrever. No entanto, as referências podem ser úteis como auxiliares para o processo de especialização ser suportado por um dos componentes integrados.

Sinónimo, ou associação de nome para nome, permite criar nomes alternativos que podem ser úteis para resolver conflitos de nomes ou alterar as designações de elementos que constituem a interface do componente. Desta forma, vários componentes podem aceder a uma mesma entidade quando cada um conhece essa por nomes diferentes.

Tipo, ou referência de valor para nome, determinam o formato dos dados e descrevem o comportamento de uma dada entidade. A linguagem de integração fornece um conjunto muito limitado de tipos e de operações sobre esses tipos, pelo que, na maioria dos casos, os tipos de dados são definidos pelos vários componentes integrados.

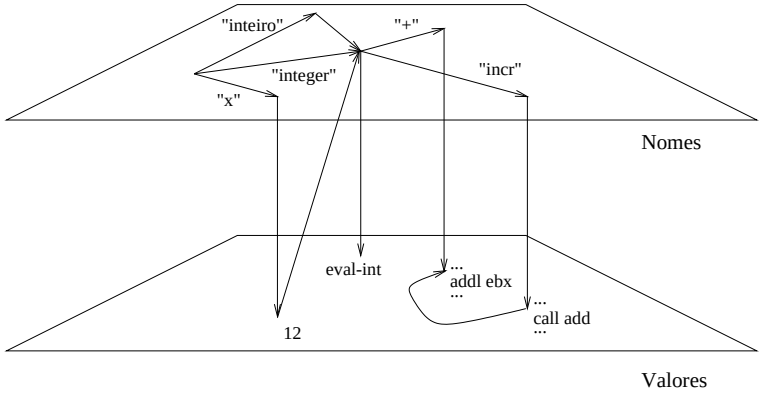


Figura 6.2: Exemplo das quatro associações possíveis em **MAP**.

As quatro associações descritas oferecem grandes possibilidades expressivas. No entanto, a falta de restrições permite a criação de estruturas arbitrariamente complexas que na falta de disciplina na sua utilização se podem tornar de difícil gestão. O mesmo tipo de problema surge na utilização indisciplinada de apontadores nas linguagens de programação comuns. No entanto, ao contrário dos endereços dos apontadores, os nomes oferecem informação útil. Segundo Boundy [Bou91], metade da documentação de um programa reside nos seus nomes.

Organizações hierárquicas

A abstracção é o mecanismo mais eficaz para manipular sistemas complexos. As hierarquias são estruturas onde as abstracções podem ser tratadas a diversos níveis. Numa hierarquia, o problema é decomposto em níveis ordenados onde o pormenor não desaparece, mas é disponibilizado apenas nos níveis de maior detalhe. O facto mais importante reside no facto de a utilização de hierarquias surgir de forma natural à maioria das pessoas.

O primeiro passo para a integração dos componentes consiste em construir, de uma forma automática ou manual, uma hierarquia de nomes que descreva os elementos que constituem o componente. O componente em si não necessita de ser alterado, apenas uma camada contendo referências para as suas entidades acessíveis necessita ser construída. Embora esta camada exiba uma estrutura hierárquica, o componente por ela

referida não necessita de o ter, basta que se construa uma vista que interpreta e organiza os elementos visíveis hierarquicamente.

Os nomes que constituem os elementos da hierarquia podem ser referidos através de caminhos quer a partir da origem da hierarquia, quer a partir de um outro ponto, tal como na maioria dos sistemas de ficheiros. Esta característica permite utilizar sequências de nomes curtos para designar nomes locais enquanto sequências maiores designam nomes exteriores ao componente. Desta forma, conflitos de nomes entre componentes são evitados, uma vez que cada componente define a sua estrutura de nomes a partir de um ponto distinto da hierarquia. Linguagens como o **C++** [Str91] ou **Ada** [Kru92] oferecem mecanismos limitados para o controlo de conflitos de nomes através de operadores de visibilidade. O **Obliq** [Car95] utiliza caminhos absolutos para permitir a distribuição dos objectos, mas utilizando as designações das suas posições originais para os identificar. O **Tcl/Tk** [Ous94] ou o **Java** [AG98] utilizam nomes hierárquicos para designar as entidades evitando conflitos entre os nomes definidos por cada uma.

Composição de hierarquias

A utilização de hierarquias como estruturas de representação dos dados permite que o sistema cresça sem que exista um limite prévio, mas o crescimento não implica que deixe de existir um controlo sobre toda a estrutura. De facto, cada sub-hierarquia encontra-se sob a administração do componente a que se refere, e o sistema de integração limita-se a interligar as diversas sub-hierarquias num única árvore de nomes. O escalamento das aplicações pode ser obtido sem dificuldade, uma vez que o seu crescimento não obriga a redesenhar o sistema de integração. Aliás, o estudo de sistemas de ficheiros distribuídos e de aplicações baseado na "`world wide web`" tem verificado que o crescimento dos sistemas não afecta directamente cada componente pois este apenas interage com um número limitado dos restantes componentes.

Uma vez que cada componente contribui com uma sub-hierarquia para a árvore global, a primeira fase do processo de integração procura organizar as sub-hierarquias dos componentes de uma forma consistente. A integração de cada sub-hierarquia a partir da raiz da árvore, embora seja uma solução possível, não explora as interdependências

entre os componentes e a localidade de certos tipos de dados envolvidos. A colocação das sub-hierarquias de vários componentes a partir de um mesmo vértice da árvore surge como desejável em situações de composição ou de escolha múltipla. As hierarquias construídas no capítulo 4 e os resultados das operações definidas no capítulo 5 são exemplos de sub-hierarquias que devem ser mantidas próximas para uma mais fácil e acessível manipulação das mesmas. De facto, se se agrupar em componentes relacionados em determinadas zonas da hierarquia, os caminhos cruzados ficam mais curtos e facilita-se a gestão do processo de integração. Mais importante seja talvez o facto de os componentes poderem ser substituídos sem existir necessidade de afectar outros componentes além daqueles com quem se relaciona directamente. No caso de certos componentes funcionarem apenas como sub-componentes, ou seja, só dialogam com o resto da aplicação através de determinados componentes – por exemplo, diálogo com o utilizador, outras aplicações ou periféricos –, as suas sub-hierarquias podem ser colocadas nos extremos da árvore do componente com quem dialogam. A utilização de uma estrutura hierárquica consegue, assim, um encapsulamento e modularidade que simplifica a gestão dos diversos componentes da aplicação.

6.1.3 Modelo de execução

Os elementos básicos do modelo de execução de um programa são apresentados na figura 6.3. O modelo de execução é baseado numa pilha ("`stack`") de dados, designada por pilha de argumentos, para armazenar os argumentos e aceder aos resultados de rotinas. A utilização de pilha de argumentos como mecanismos de execução é comum a muitas linguagens de programação interpretadas. A manipulação directa de uma pilha de argumentos pressupõe a utilização de uma notação em que os operandos precedem os operadores, designada por *postfixada*. No entanto, a maioria dessas linguagens oferece ao utilizador uma interface em que muitos dos operadores matemáticos são *infixados*, caso em que o operador surge entre os seus operandos. Além disso, muitas outras construções dessas mesmas linguagens utilizam uma notação *prefixada*, em que o operador precede os seus operandos. Embora algumas linguagens, como o Lisp [Seb93], utilizem apenas uma notação prefixada, a maioria utiliza uma mistura de notações infixadas e

prefixadas. A utilização deste tipo de notações implica que o utilizador necessita de pensar nas construções a utilizar e, subsequentemente, o processador da linguagem necessita fazer o raciocínio inverso para obter uma sequência postfixada que possa ser calculada. A decisão de oferecer ao utilizador o acesso directo à pilha de argumentos, e por consequência, à necessidade de utilizar uma notação postfixada, procura simplificar a linguagem. Contudo, como se trata de uma linguagem de integração, é possível disponibilizar um componente que funcione como conversor da linguagem de formato semelhante à notação matemática ("*infix*") para texto de formato postfixado.

No modelo de execução proposto nesta dissertação uma segunda pilha de dados, designada por pilha de retorno, armazena variáveis locais e endereços de retorno das rotinas. A maioria das linguagens compiladas dispõe as duas funcionalidades numa única pilha. A utilização de duas pilhas permite desacopular os argumentos das rotinas, tornando-se mais fácil partilhar argumentos comuns a diversas rotinas, sendo os elementos não partilhados mantidos na pilha de retorno. Fica também facilitada a alteração dos dados por interposição de rotinas de transformação que consomem, produzem ou substituem argumentos. Neste aspecto, a linguagem proposta assemelha-se ao **Forth** [Bro94, Kna93] que também recorre a duas pilhas com semânticas muito semelhantes, ou o **PostScript** [Inc85] onde a segunda pilha está completamente escondida sendo utilizada exclusivamente pelo processador da linguagem e por componentes compilados que dela necessitem.

Mecanismos baseados em objectos

A utilização de muitos dos conceitos associados às linguagens baseadas em objectos surge de uma forma natural. De facto, a notação postfixada permite descrever os procedimentos como frases constituídas por sujeito seguido do verbo, apenas sendo eliminadas as preposições. A linguagem **Smalltalk** [GR89] utiliza uma notação mais rigorosa, do ponto de vista linguístico, em que o sujeito e o verbo são seguidos por um conjunto de selectores e argumentos que correspondem às preposições e complementos directos, indirectos ou circunstanciais. No entanto, o processamento da linguagem já exige um

tratamento elaborado e demorado, razão pela qual a linguagem é pré-compilada para uma máquina abstracta baseada numa pilha de dados.

Na linguagem **MAP**, devido à notação postfixada, o objecto colocado no topo da pilha é utilizado para determinar o receptor da mensagem. Assim, o topo da pilha funciona como o `this` ou o `self` nas linguagens orientadas para objectos. O resultado do envio da referida mensagem produzirá um conjunto de resultados em que o elemento colocado no topo da pilha é utilizado como receptor para a mensagem seguinte, se nenhum outro objecto for colocado na pilha antes do envio da nova mensagem.

Variáveis lineares

Nas linguagens funcionais as variáveis são entendidas como entidades que podem ser criadas, cujos valores podem ser lidos mas não alterados, e que podem ser destruídas. Na lógica linear, a leitura dos valores das variáveis implica a sua destruição [Bak95]. Assim, as variáveis lineares podem apenas ser criadas com um dado valor, e serem destruídas com a devolução do mesmo valor. Desta forma, a utilização de um valor implica a destruição da variável que o suporta, obrigando à posterior criação de uma nova variável com o mesmo valor, e possivelmente o mesmo nome, para que a leitura não seja considerada destrutiva de um ponto de vista macroscópico. Da mesma forma, a utilização da mesma variável não pode ocorrer duas vezes, pelo que é necessário proceder à duplicação explícita.

A utilização de uma pilha de argumentos permite a criação de variáveis lineares. De facto, as operações são obrigadas a consumir os valores e a recolocá-los de volta, se necessário, tal é o caso da duplicação do elemento do topo da pilha – "`dup`" – que é retirado e depois colocado de volta em duplicado. As principais vantagens consistem no facto de não ser necessário efectuar gestão de memória sobre as variáveis que se encontram na pilha, o que é um ganho significativo se considerarmos que estas variáveis têm um tempo de vida curto e existem em grande número. Por outro lado, não é necessário efectuar nenhum tipo de sincronização uma vez que existe um só caminho para a variável, o que facilita a interligação de módulos que podem executar, ou comunicar com entidades que executem, em paralelo [Bak94].

Note-se que a utilização de variáveis não lineares é ainda possível através da sua colocação na hierarquia, obrigando contudo à atribuição de um nome. As variáveis não lineares são aquelas que podem ser reescritas e lidas um número ilimitado de vezes, sendo utilizadas como elementos de partilha entre os diversos componentes da aplicação. As variáveis lineares são utilizadas para passar valores não partilháveis entre os referidos componentes. O modelo de execução é constituído por duas pilhas para variáveis lineares e uma árvore para variáveis não lineares.

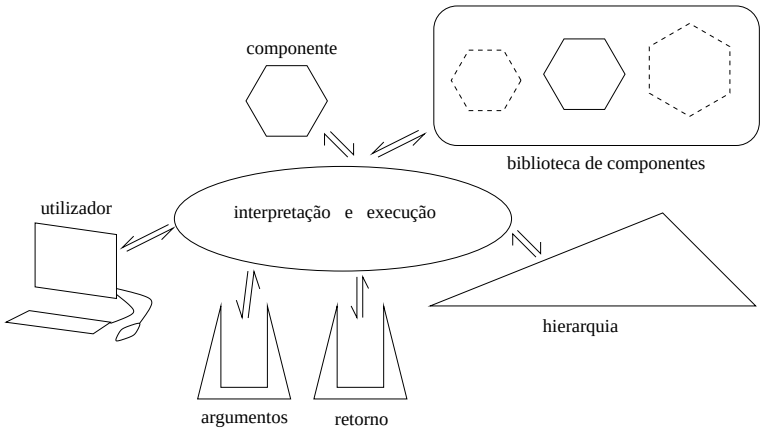


Figura 6.3: Modelo de execução de um programa **MAP**.

6.2 Mecanismos de activação de identificadores

A linguagem proposta baseia-se na activação de identificadores representados na hierarquia de dados. A execução de um programa distingue apenas constantes, que são empurradas para a pilha de argumentos, e identificadores, que são activados. A activação de um identificador desencadeia a execução do elemento por ele referido: variável ou rotina. A execução de uma variável empurra o valor a ela associado para a pilha de dados, enquanto a execução de uma rotina consiste na activação sequencial dos identificadores nela definidos. As rotinas pré-definidas pela linguagem oferecem operações base de manipulação da pilha de dados e da árvore de nomes. A pilha de retorno não é visível através das rotinas disponibilizadas, mas pode ser acedida por componentes que executem em código máquina nativo.

Como se trata de uma linguagem de integração não é possível definir à partida quais os tipos de dados a disponibilizar. A linguagem limita-se a definir os tipos que necessita para efectuar a integração de componentes e a execução das rotinas neles definidas. Assim, apenas os tipos `name` e `code` estão pré-definidos pela linguagem, qualquer outro tipo de dados básico, como números inteiros, é suportado por um componente que pode ou não ser incluído. Inclusivamente, a definição e activação de rotinas interpretadas existe num componente à parte, e só necessita de ser incluído se houver interface com o utilizador através da linguagem e não de um dos restantes componentes. O próprio processador da linguagem em notação postfixada é um componente que funciona em conjunto com o interpretador, e que define um conjunto de tipos de dados adicional, como inteiros e cadeias de caracteres. No entanto, o processador da linguagem apenas coloca na pilha de argumentos os elementos obtidos do utilizador, devendo as operações que operam sobre esses tipos de dados ser fornecidas por outros componentes. Esta aproximação, tipo construções de `Lego` ou `Mecano`, define apenas a estrutura e a semântica do mecanismo de integração, sendo a quase totalidade das operações fornecidas por componentes a integrar caso a caso.

6.2.1 Resolução dos nomes

A resolução dos nomes consiste na obtenção da entidade que se encontra associada a um nome que é identificado por um caminho. O caminho que conduz a um dado nome diferencia-o de outros nomes iguais que possam existir na aplicação, e determina o componente que é responsável pela sua manutenção. A resolução do caminho é feita pela linguagem de integração que também verifica a existência do nome, a partir do qual obtém uma referência para o componente. Com a referência, assim obtida, a linguagem interage com o componente para activar a rotina ou manipular a variável, dependendo do tipo de referência que se encontrava associada ao nome. Como a linguagem não fornece informação quanto ao comportamento do componente, assume-se que a referência pode indicar elementos distintos em instantes de tempo diferentes. Assumindo este comportamento dinâmico, a resolução dos nomes é novamente executada em cada pedido de resolução. Para mais, nada impede o componente de alterar a sua hierarquia

de nomes dinamicamente, por exemplo, por forma a reflectir o seu estado. Assim, a existência de um nome a partir de um dado caminho em certo momento, não implica a existência desse nome, ou do caminho que lhe conduz, instantes depois.

Visibilidade dos nomes

Uma vez que todos os nomes existentes na aplicação se encontram registados na hierarquia de nomes, é sempre possível aceder a qualquer dos nomes a partir de caminhos que partem da raiz. No entanto, é possível a utilização de caminhos abreviados, da mesma forma que nos sistemas de ficheiros se utilizam os caminhos relativos. Por exemplo, as palavras-chaves do interpretador são nomes com um caminho pré-definido e que é procurado antes de qualquer outro. Garante-se, desta forma, a precedência destas palavras-chaves e impede-se que estas possam ser redefinidas por engano. Como não existem protecções dentro da aplicação, caso se pretenda retirar determinada palavra-chave, pode-se eliminar o nome do referido caminho.

A utilização de caminhos a partir da raiz para aceder a todas as entidades do sistema, excepto as que estão nas pilhas de dados, permite criar uma organização estruturada. As entidades globais deixam de existir num único local, sujeitas a conflitos de nomes e a excesso de população, e passam a poder ser classificadas por importância ou por áreas temáticas. As entidades de maior importância estarão, em princípio, mais perto da raiz do sistema e, consequentemente, de acesso mais fácil e através de caminhos mais curtos. Da mesma forma, grupos de entidades relacionadas podem ser agrupadas a partir de um mesmo caminho ou mesmo estabelecer importâncias relativas através da utilização de um maior número de níveis da hierarquia.

Reencaminhamento de nomes

A utilização de nomes alternativos, ou sinónimos, permite alterar a resolução dos nomes. Note-se que, do ponto de vista estrutural, continua a existir apenas um caminho para cada nome. No entanto, ao encontrar um determinado nome cujo valor seja um sinónimo, o sistema de resolução de nomes recomeça a busca a partir do caminho indicado no valor desse nome. Os sinónimos permitem, como o seu significado linguístico

sugere, criar significados alternativos para a mesma entidade. Desta forma, é possível criar classificações e associações baseadas em entidades com origem em diversos componentes, que, de outra forma, estariam colocadas em ramos distintos da hierarquia em posições determinadas por cada componente.

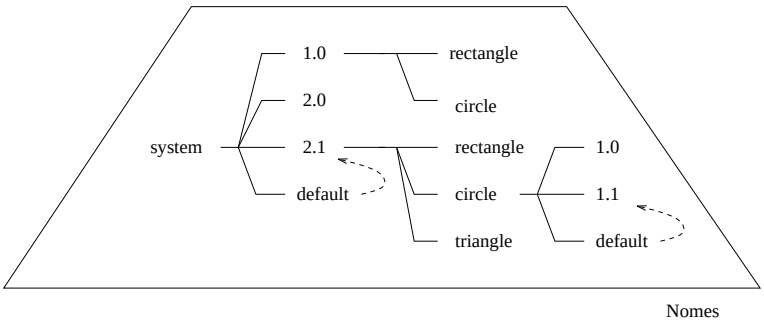


Figura 6.4: Sinónimos chamados `default` são utilizados como mecanismo de selecção de versões.

A resolução de nomes utiliza ainda um sinónimo com o nome "`default`" para permitir a continuação da busca em caminhos alternativos. Se o nome pretendido não existir no caminho indicado, a busca continua no caminho referido pelo valor do sinónimo `default`. Estes sinónimos de características especiais têm especial importância no processo de especialização. A sua utilização como mecanismo de especialização será discutido nas secções 6.2.3 e 6.2.4. Além disso, verificou-se que estes sinónimos são úteis como auxiliares na selecção de versões e configurações, ou seja, em situações em que mais de uma versão de determinado componente existe simultaneamente na execução de uma aplicação. Esta situação vulgar em projectos de desenvolvimento, uma vez que muitos dos componentes existentes recorrem a versões de certo componente que são incompatíveis com as versões exigidas por outros.

6.2.2 Esquema de tipos

Definição 6.3 (tipo) *Um* tipo *é um conjunto de entidades caracterizadas por uma representação estrutural e um conjunto de operações que lhes conferem um comportamento comum.*

Em programação, os tipos de dados são usados na detecção de certos tipos de erros. Quando os tipos de dados são utilizados para verificações de consistência, estes são

introduzidos, quando necessário, pelas entidades que necessitam das referidas verificações [MM90]. No caso da integração de componentes, cada componente define um conjunto de tipos de dados que necessita para dialogar com outros componentes. É responsabilidade da linguagem de integração e do processo de especialização garantirem as equivalências entre os diversos tipos exportados por cada componente. Só desta forma é possível trocar informação utilizando a mesma representação, e substituir um componente por outro que apresente o mesmo comportamento para o tipo em questão. A menos que a linguagem de integração obrigue a uma especificação muito rigorosa dos tipos de dados, a informação disponível sobre cada tipo de dados é limitada. Assim, as verificações de consistência que é possível efectuar são também reduzidas, pelo que os tipos de dados funcionam mais como identificadores da entidade. Os componentes, que pretenderem, podem verificar o tipo dos argumentos que recebem.

A identificação de um tipo de dados é feito através de um caminho na hierarquia. Ao caminho que identifica o tipo está associada a rotina de tratamento que é utilizada pelo interpretador para activar as entidades do referido tipo, uma vez que cada tipo de dados pode ser activado de forma diferente. Embora as variáveis possam ter o mesmo método de activação, as rotinas interpretadas são processadas por código especial do interpretador e as rotinas compiladas em código máquina nativo são processadas por outro activador. Desta forma, é possível emular código máquina de outras arquitecturas de forma transparente.

Os nomes com origem no caminho que identifica o tipo de dados definem os atributos e o comportamento desse tipo. No caso de os nomes representarem atributos, a sua activação calcula o deslocamento na estrutura da entidade e coloca o respectivo endereço na pilha de dados para posteriores operações de leitura ou escrita. Se os nomes representam rotinas, a sua activação corresponde à execução do código que lhes está associado. O processo de activação de um nome, depois de excluir as palavras-chaves e os caminhos com origem na raiz, utiliza o caminho definido pelo tipo do objecto no topo da pilha de dados e activa a entidade com esse nome com origem nesse caminho. Se o referido nome não existir a partir desse caminho, é apresentada uma mensagem de erro "`message not understood`", ou seja, a tipificação é dinâmica.

Consistência de tipos

Se o nome utilizado representar um caminho desde a raiz, esse nome é activado e pode modificar o conteúdo da pilha de argumentos e das entidades nela contidos. Desta forma, é possível utilizar rotinas definidas para outros tipos de objectos sobre os objectos que se encontram na pilha de argumentos sem gerar qualquer tipo de erro, subvertendo qualquer tipo de verificação de consistência. Embora este género de acções esteja em desacordo com as leis básicas de programação tornando fraco o sistema de tipos, permite efectuar sem restrições conversões entre tipos de dados de componentes distintos. Em muitos casos as entidades têm representações equivalentes em dois ou mais componentes, pelo que o processo de conversão resume-se a uma metamorfose das entidades, ou seja, a substituição do identificador de tipo associado às referidas entidades.

6.2.3 Sobreposição de operadores e polimorfismo

Um dos mecanismos de abstracção mais importantes ao nível da programação consiste na sobreposição de operadores. A sobreposição de operadores consiste na utilização de um mesmo nome para activar realizações semanticamente distintas em entidades de tipos diferentes. Uma vez que a realização a activar depende da entidade no topo da pilha de argumentos, o nome de qualquer operador pode ter várias realizações sobrepostas. No entanto, apenas a entidade do topo da pilha de argumentos é utilizada para decidir a realização a utilizar, não sendo considerados os restantes elementos da pilha, o que limita as possibilidades de sobreposição.

Enquanto na sobreposição de operadores o mesmo nome é associado a realizações semanticamente distintas, no polimorfismo uma operação com a mesma semântica suporta um conjunto de tipos distintos.

O polimorfismo de inclusão está geralmente relacionado com a subtipificação, que no caso das linguagens orientadas para objectos é obtido à custa de mecanismos de herança. Na linguagem **MAP** não existe herança nem subtipificação mas é ainda possível permitir que a mesma operação seja aplicável a vários tipos de dados, conforme descrito na secção 6.2.4. A possibilidade de utilizar sinónimos `default` para extender as

operações que são aplicáveis a um determinado tipo de dados com outras operações, possivelmente provenientes de outro tipo de dados. Os sinónimos `default` estabelecem relações de dependência entre os vários tipos de dados, mas estas dependências não estão sujeitas a um critério ou classificação como no caso das hierarquias de herança. A inexistência de regras no estabelecimento das dependências permite maior flexibilidade à linguagem de integração, mas necessita ser utilizada de uma forma controlada e consistente.

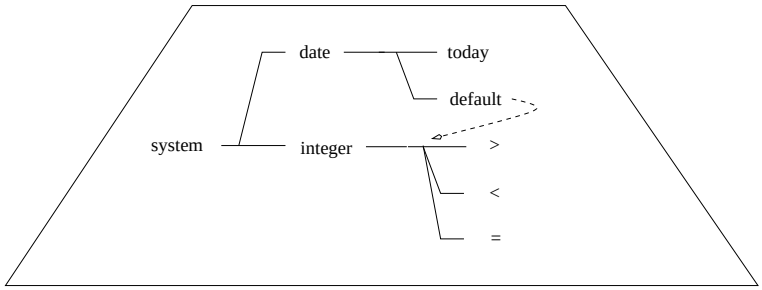


Figura 6.5: Exemplo de utilização do polimorfismo com base em sinónimos `default`.

O polimorfismo paramétrico utiliza tipos de dados como argumentos para rotinas. Como a identificação de um tipo é feita através de um caminho, a passagem de um tipo para uma rotina consiste na passagem do caminho que a identifica. Para a utilização desse mesmo tipo e das suas operações pela rotina, é necessário que o próprio tipo de dados possa ser manipulado como um dado. Na linguagem proposta o caminho é descrito por uma cadeia de caracteres, pelo que a manipulação dos caminhos, e nomeadamente a sua composição, é simples de realizar. A activação de operações com base em argumentos que denotam tipos de dados resume-se a concatenar o identificador do tipo com o nome desejado.

6.2.4 Modificação incremental

A modificação incremental denota a capacidade de alterar e adicionar características aos componentes sem ter de recorrer à edição do componente. Das técnicas de especialização abordadas em 2.3.1, a especialização não implica a modificação do componente. A modificação e cópia necessita ser efectuada na linguagem de desenvolvimento em que

o componente está descrito, pelo que, além de não ser um caso de modificação incremental só pode ser executado na linguagem em que o componente foi originalmente descrito.

A utilização de técnicas de encapsulamento permite modificações incrementais distintas do mesmo componente. Isto é, podem existir encapsulamentos distintos, cada um orientado para uma situação concreta, para um determinado componente. A linguagem **MAP** consegue, através do reencaminhamento de nomes (ver secção 6.2.1), substituir e acrescentar identificadores à descrição do componente. Sob este ponto de vista, o encapsulamento pode ser entendido como uma forma restrita de herança.

A linguagem **MAP** não suporta o mecanismo de herança nem sob a forma de delegação nem sob a forma de concatenação. A herança por delegação obriga a actualizar o receptor da mensagem no momento da delegação. Como os sinónimos não efectuam a substituição do elemento no topo da pilha de argumentos pelo novo receptor, a rotina a executar não tem o contexto correcto. Na herança por concatenação, o receptor deverá ser um sub-tipo do objecto que recebe a mensagem. Na linguagem **MAP** não existe nenhum tipo de verificação que obrige os sinónimos a ter origem apenas em sub-tipo, logo o mecanismo só funciona se o utilizador ou ferramentas específicas assim o façam cumprir. No caso de os componentes a integrar terem origem em linguagens de desenvolvimento orientadas para objectos, a linguagem proposta é capaz de continuar a garantir a correcta partilha de código.

Embora a distinção entre agregação e encapsulamento não seja sempre nítida, a agregação preocupa-se com a associação e composição de componentes. Frequentemente, os componentes a agregar, já foram previamente sujeitos a alguma forma de especialização. No entanto, como a agregação produz um componente modificado que altera o comportamento dos componentes originais, tal perspectiva é por vezes confundida com encapsulamento. Tal como no encapsulamento, a agregação pode combinar os diversos componentes de diversas formas.

6.3 Breve caracterização da linguagem

A integração de componentes pressupõe, numa primeira fase, a possibilidade de cada um desses componentes poder operar com os outros. Para que esta interoperação seja possível, é necessário definir um modelo comum a que todos os componentes obedeçam. O sistema proposto consiste numa árvore de nomes para representar os dados, por forma a que os componentes possam exportar as entidades que se pretendem acessíveis, e simultaneamente ter acesso à informação disponibilizada pelos outros componentes. Para tal é disponibilizado um conjunto de operações que permite introduzir ou retirar um componente, introduzir ou retirar um nome e o respectivo valor no sistema, bem como obter o valor a partir do nome e enumerar os nomes da árvore.

Para desencadear as operações desejadas, os componentes activam os nomes desejados. A activação consiste na determinação do valor da entidade associada ao nome. A determinação dos valores é feita através da execução de código compilado. Para tal, é utilizada uma pilha de dados para a transferência dos valores intermédios. Mesmo que a entidade seja constante, é executado o código que copia esse valor para a pilha de dados. O processo de activação é constituído por uma operação que determina o código a executar para determinada entidade e por um conjunto de operações para introduzir, retirar e inspeccionar os elementos da pilha de dados, bem como criar e destruir a própria pilha.

A partir deste momento já é possível, dado um conjunto de componentes, fazer chamadas cruzadas e transferir informação nessas chamadas. A integração fica concluída com a determinação da primeira chamada, que desencadeia o processo. O componente que recebe esta primeira chamada é designado por interpretador. Este componente só é efectivamente um interpretador se for necessária a intervenção humana ou a execução de código não compilado. O componente que desempenha as tarefas de interpretador pode ser alterado ou mesmo substituído durante uma mesma execução da aplicação. Desta forma, não existe um único interpretador e, consequentemente, uma linguagem pré-definida, mas uma interface que permite a interoperabilidade dos componentes.

No entanto, e numa primeira fase, disponibiliza-se uma linguagem muito simples que oferece um acesso quase directo à interface de interoperabilidade. Esta linguagem ofe-

rece ainda a capacidade de definir rotinas que retêm um conjunto de símbolos para posterior interpretação, e tipos de dados e funções básicas comuns à maioria das linguagens de programação imperativa.

6.3.1 Aspectos lexicais e sintácticos

Nesta secção, abordaremos o interpretador no ponto de vista lexical e sintáctico, apresentando-se em A.1 uma representação exaustiva e rigorosa. As características semânticas das rotinas do interpretador serão tratadas como elementos da interface de interoperabilidade devido à correspondência directa que existe.

O interpretador permite a introdução de três tipos de dados que correspondem a valores inteiros, reais e cadeias de caracteres. Todos estes valores podem ser introduzidos de formas alternativas e depois convertidos entre si. A sua identificação segue as regras da linguagem **C** [KR88] para valores inteiros, reais e cadeias de caracteres com caracteres no código ASCII. Além disso, permite-se a identificação de números em qualquer base de 2 a 36, bem como cadeias de caracteres com caracteres em código hexadecimal, nomes executáveis e nomes literais segundo as regras da linguagem **PostScript** [Inc85]. Contudo, como a identificação dos nomes literais substitui o caracter '/' por ':', uma vez que a separação dos componentes de um nome é feita como no sistema operativo **UNIX** [Ker84].

Note-se que o analisador sintáctico do interpretador constrói o valor de acordo com a sequência de caracteres lidos e atribui a esse valor um caminho que determina o local na hierarquia onde deverá existir a descrição do tipo e das suas operações. Embora se pressuponha a existência de operações para manipular estes tipos, estas podem ser substituídas e retiradas sem conhecimento do interpretador.

O interpretador não utiliza palavras reservadas, embora comece por procurar os nomes numa directoria específica, que pode ser directamente manipulada pelo utilizador para controlar a visibilidade de um nome. No entanto, pelo seu uso intensivo justifica-se referir a forma de contrução de matrizes e de rotinas, que são delimitadas por parêntesis rectos e chavetas, respectivamente. Estes dois tipos de dados compostos são construídos por rotinas exteriores ao interpretador, exibindo um funcionamento igual ao da

linguagem **PostScript**. Apenas a construção das rotinas exige acesso aos lexemas do analisador lexical antes de serem avaliados pelo interpretador, pelo que o analisador lexical e o interpretador devem poder ser acedidos separadamente.

6.3.2 Descrição semântica

Como quase todas as rotinas da linguagem se encontram em componentes que podem ser inseridos ou retirados, o núcleo da linguagem reduz-se a funções de manipulação da pilha de dados, da árvore de nomes e da avaliação dos nomes existentes na referida árvore. Nesta secção são introduzidas as funções disponibilizadas aos componentes para construção, manipulação e activação de nomes na árvore. Funcionando como um componente, o interpretador tem acesso a essas mesmas funções, embora nem todas sejam disponibilizadas ao utilizador. Uma definição mais rigorosa dessas funções é expressa em A.2 e as assinaturas das funções de interface apresentadas em A.3.

As operações que permitem aceder à pilha de dados consistem em colocar e retirar elementos no topo da pilha. Na realidade, a operação de colocação ("`push`") não existe no interpretador, sendo realizada automaticamente quando é identificado um lexema. Existem também diversas operações que colocam elementos na pilha de dados e que apresentam a mesma semântica das funções homónimas em **Forth** [Kna93] como, por exemplo, `dup` ou `over`. Os elementos ao serem retirados da pilha, através da operação `drop`, são imediatamente apagados pois são lineares. Caso o valor de algum desses elementos necessite ser guardado na árvore deverá ser primeiro copiado, através da operação `fetch`. Esta operação permite obter cópias de qualquer elemento na pilha de dados para facilitar a escrita das rotinas. No entanto, a operação `fetch` não se encontra disponível para o utilizador que apenas poderá retirar um valor da pilha, retendo o seu valor, se o colocar na árvore.

As operações de acesso à árvore consistem na inserção de um elemento na árvore dado o seu nome, a sua procura ou eliminação com base no nome que lhe foi atribuído na inserção. A função `bind` insere um elemento na árvore dado um nome que representa o caminho completo desde a raiz e um valor que é copiado do elemento no topo da pilha de dados. A enumeração dos nomes de todos os elementos na árvore, efectuada

pela função `tree` ou através do comando `dir` do interpretador. Estes elementos são retirados da árvore através da indicação do caminho completo desde a raiz através da função `forget`. A procura de um elemento na árvore é feita automaticamente pelo interpretador quando encontra um nome executável colocando uma cópia do seu valor no topo da pilha de dados. No entanto, os componentes têm acesso, através da função `lookup`, aos valores dos nomes. Esta busca pode representar uma busca directa na árvore, se for indicado o caminho completo desde a raiz, ou ser influenciada pelas regras de visibilidade e pelo tipo do elemento no topo da pilha.

O interpretador utiliza as funções `lexeme` para identificar um único lexema e colocar o elemento correspondente no topo da pilha de dados e `eval` para determinar o valor do elemento no topo da pilha. O próprio interpretador pode ser invocado pelos componentes, através da função `interp`, sendo localizado pelo caminho que identifica a sua posição na árvore. Assim, podem existir simultaneamente diversos tipos interpretadores distintos ou várias ocorrências de um mesmo interpretador. Para tal é frequentemente necesssário criar uma pilha de dados independente e uma zona da árvore distinta para cada interpretador guardar os elementos que necessita. Esta operação é conseguida através da função `task` que fica associada a um sub-directorio de `/task/` com o número da tarefa que a designa. Finalmente, é possível colocar o sistema a funcionar através de uma única função – `main` – e existem funções para iniciar e concluir o sistema antes de chamar as funções acima – `init` e `end`, respectivamente.

Das funções que não fazendo parte do núcleo da linguagem representam extensões quase sempre presentes salientam-se as instruções condicionais – `if` e `ifelse` – e os ciclos – `loop` e `while`. Existem ainda as funções usuais para a manipulação das cadeias de caracteres e de números reais e inteiros. A introdução de um novo tipo de dados resulta de uma operação de sintetização de um ou mais elementos dos tipos base e da colocação num ramo da árvore de uma função de avaliação do tipo e nos seus ramos das operações que disponibiliza. Este comportamento é efectuado por numerosos componentes, como parte do processo de integração, antes de iniciar a interacção com os restantes componentes.

Exemplo 6.1 *Cálculo em* **MAP** *do factorial de valores inteiros positivos, no exemplo, para o*

valor 5:

```
{ dup 1 > { dup 1 - factorial * } if } :factorial bind
5 factorial print
```

6.4 Apreciação global

Além da modularidade inerente à utilização de componentes independentes, o sistema proposto decompõe cada componente em rotinas e variáveis com nomes distintos. A árvore de nomes funciona como uma representação aglutinadora dos vários nomes, respectivas rotinas e variáveis associadas, criando composições que podem ser distintas das existentes no componente original. O encapsulamento de determinadas partes dos componentes é conseguida quer através de atribuir um nome apenas às partes acessíveis, quer tornando alguns nomes menos acessíveis através da sua colocação em extremos da hierarquia. A hierarquia permite, igualmente, criar mais de uma perspectiva – ou vista – do mesmo componente, o que pode ser útil quando dois ou mais componentes necessitam de informação distinta de um mesmo componente. Ao nível da linguagem é de referir a continuidade da sintaxe que permite activar rotinas ou variáveis da mesma forma e, consequentemente, substituir umas por outras sem necessidade de alterar os programas.

Exemplo 6.2 *As sub-hierarquias* `public` *e* `private` *da figura 4.4 apresentam duas perspectivas, das quais apenas a pública tem interesse para integração. A perspectiva privada é, contudo, relevante para a selecção do componente. Da mesma forma, a perspectiva protegida (não representada na figura) pode ser utilizada conjuntamente com a perspectiva pública para efeitos de especialização.*

6.4.1 Manipulação de componentes

A manipulação dos componentes, quer pela linguagem de integração, quer pelos próprios componentes requer a correcta identificação de cada entidade. Assim, no momento do registo cada componente insere na árvore de nomes um conjunto de elementos

através dos quais aceita partilhar a sua informação. Cada componente deverá ocupar um ramo da árvore existente e povoá-lo com os nomes de acesso. Por exemplo, o interpretador e os componentes mais básicos registam os seus nomes em `/sys`. Ao povoar um único ramo da árvore, em vez de distribuir os nomes por toda a árvore, fica facilitada a localização e identificação do contexto em que se inserem, da mesma forma que no processo de classificação e selecção.

Muitos dos componentes pretendem substituir ou extender a funcionalidade de outros elementos já existentes. Para tal, é necessário substituir a associação do nome pelo novo elemento fornecido pelo componente. A utilização de sinónimos permite que esta substituição seja feita de uma forma controlada e indicando, de forma explícita, o novo elemento e o seu contexto. A identificação do contexto permitirá determinar o componente responsável pela análise do valor do sinónimo.

Quando os elementos de substituição não apresentam a mesma interface, quer do ponto de vista de ordem ou número de argumentos, quer do tipo de cada um desses argumentos, torna-se necessário escrever uma rotina de interposição. Pela sua simplicidade, estas rotinas podem ser frequentemente escritas numa linguagem interpretada, como a que se propõe nesta dissertação. As rotinas de interposição são também especialmente úteis para fazer uma triagem das invocações. A triagem é necessária quando se pretende que o mesmo nome seja utilizado em várias situações diferentes e as técnicas usuais, como o polimorfismo, não são suficientes. Uma vez que os critérios de triagem podem não se limitar aos tipos dos objectos mas a instâncias com características particulares, torna-se necessário escrever código específico para cada caso.

Todas estas situações de excepção, muito comuns em ambientes de integração, podem ser descritos por pequenas rotinas escritas na linguagem proposta. A linguagem apresenta, além das capacidades de integração dos módulos, um conjunto de facilidades das linguagens genéricas para permitir realizar operações de especialização de complexidade reduzida.

6.4.2 Desempenho

Como se trata de uma linguagem de integração, o desempenho da ferramenta depende mais do desempenho dos componentes que da sua interligação. Com efeito, quanto maior for a complexidade de cada componente menos relevante será a contribuição da linguagem para o desempenho da ferramenta. Nesta secção pretende-se dar uma ordem de grandeza do desempenho da ferramenta desenvolvida, que representará o pior caso, sempre que existam componentes compilados em código máquina nativo.

Quando comparadas com linguagens compiladas para código máquina nativo, as linguagens interpretadas equivalentes apresentam elevados tempos de execução. Este facto deve-se, principalmente, à necessidade de determinar qual o código nativo a executar na sequência de um pedido do utilizador. No entanto, uma linguagem interpretada com características essencialmente estáticas pode ser executada de duas a mais de cem vezes mais devagar que o equivalente compilado. A codificação cuidada do interpretador e escolhas esclarecidas, como técnicas de passagem directa de instruções ("*direct threading*"), permitem velocidades de execução que se aproximam das linguagens compiladas [Bro81].

A resolução dinâmica de nomes ("*dynamic binding*") penaliza igualmente as ferramentas das linguagens que a utilizam. Este mecanismo oferece grande flexibilidade à linguagem e permite um conjunto de técnicas que se têm mostrado especialmente úteis no desenvolvimento de software. O recurso a uma forma limitada de resolução dinâmica de nomes permite a alguns compiladores de linguagens atingir desempenhos aceitáveis. No entanto, nestes casos, a resolução dos nomes é feita pelo compilador. Desta forma, a resolução tem de ter por base um número limitado de possíveis candidatos conhecidos no momento da compilação. Como uma linguagem de integração com características dinâmicas não conhece quais os componentes com que vai interagir, nem mesmo no início da execução da aplicação, torna-se necessário dispor de uma resolução dinâmica de nomes.

Os testes de desempenho efectuados com a ferramenta desenvolvida cobrem precisamente estas duas situações que se descreveram acima. Para os testes escolheu-se um

programa utilizado frequentemente na comparação de desempenhos de linguagens interpretadas e que consiste na execução de um ciclo duplo cujo corpo executa operações que se anulam. Ambos os contadores de ciclo utilizam o valor inicial de 1000 e o mais interior executa a instrução $x = 1 - x$. Todos os testes foram efectuados numa máquina Toshiba 4010 com um processador Pentium II a 266 MHz e 96MB de memória RAM, com o sistema operativo **Linux**. No primeiro teste comparam-se os tempos de execução do programa totalmente escrito em **MAP** com a utilização de dois componentes que executam o corpo do ciclo duplo e o ciclo interior, respectivamente.

Tabela 6.1: Desempenho do ciclo duplo para diferentes relações entre compilação e interpretação.

Código MAP	tempo
0 1000 0 { 1000 0 { 1 swap - } loop } loop	48 seg
1000 0 { 1000 0 { incycle } loop } loop	15 seg
1000 0 { outcycle } loop	0,07 seg

A título de comparação indicam-se os tempos equivalentes de linguagens compiladas e com características estáticas. No entanto, estes tempos são apenas indicativos das penalizações sofridas em utilizar interpretação e resolução dinâmica de nomes. Aliás, a realização da linguagem é muito simples e não recorre a técnicas elaboradas, como a passagem directa de instruções e *hashing* ou *caching* de nomes, pelo que os valores são apenas indicativos.

Tabela 6.2: Desempenho do ciclo duplo para algumas linguagens.

C	Java	Forth	Smalltalk	Map
0,05 seg	0,7 seg	1,46 seg	13 seg	48 seg

Os valores relativos desses tempos, apresentados na tabela 6.2 permitem concluir sobre a necessidade de recorrer, sempre que o desempenho seja um factor importante, a componentes compilados ou pelo menos desenvolvidos em linguagens interpretadas sem resolução dinâmica de nomes. Assim, para todas as partes da aplicação que não necessitem de resolução dinâmica de nomes ou de ser interpretadas, devem ser seleccionados

componentes compilados. No entanto, a sua interligação através de uma linguagem com características dinâmicas, como a que se propõe nesta dissertação, permite reter muita da flexibilidade necessária. Com uma linguagem de integração dinâmica é possível, ao verificar-se uma deficiência de desempenho, desenvolver um novo componente e substituí-lo na aplicação, sem que esta tenha de ser reiniciada.

6.4.3 Especialização

A reutilização de um componente no seu estado original é pouco frequente, em especial se este não foi desenvolvido com o objectivo de vir a ser reutilizado. Assim, alterações ao componente original, por forma a cumprir os requisitos, são frequentes.

Das técnicas de especialização referidas em 2.3.1, a particularização consiste na fixação de determinados valores ou limites aos parâmetros de determinados componentes. A flexibilidade do sistema de nomes e da gestão de tipos permitem a utilização de valores constantes ou com origem em outro componente, com grande facilidade. A interposição de rotinas de verificação de parâmetros é simplificada pela utilização da pilha de dados, que permite à rotina retirar, modificar, substituir ou eliminar qualquer dos argumentos antes da invocação do componente. A escrita dessas rotinas de verificação não necessita ser lenta uma vez que pode ser escrita à custa de funcionalidade fornecida por componentes compilados em código máquina nativo.

A utilização de uma técnica de modificação e cópia requer o acesso à descrição do componente na sua linguagem de desenvolvimento, pelo que a especialização é efectuada nessa linguagem. Mesmo que parte do componente tenha sido realizado na linguagem proposta, esta técnica deve ser evitada, por criar várias cópias semelhantes, dificultando a detecção e correcção de erros bem como a introdução de alterações.

A aplicação das restantes técnicas já foi abordada em 6.2.4. Os mecanismos oferecidos pela linguagem permitem descrever diversas formas de especialização, incluindo herança, embora as capacidades da linguagem sejam limitadas. Aliás, como linguagem de integração, muitas das capacidades são fornecidas pelos componentes que a integram. Assim, é desejável que os componentes ofereçam extensões à linguagem, permi-

tam executar as tarefas de especialização de uma forma simples e com um mínimo de verificações.

6.4.4 Integração

As aplicações construídas à custa da integração de componentes podem ser estáticas ou dinâmicas. As aplicações são consideradas estáticas se todos os componentes são conhecidos no início da execução, quer estes componentes estejam em um só ou em diversos ficheiros. As aplicações designadas por dinâmicas permitem, pelo menos, adicionar novos componentes à aplicação, sem que esta tenha conhecimento deles no início da execução. Uma das formas mais utilizadas de criar aplicações dinâmicas consiste na utilização de técnicas de comunicação com outros processos que executam alguns dos componentes. O recurso a outros processos delega no sistema operativo a resolução do nome do ficheiro a executar e seu carregamento.

O sistema proposto e a linguagem **MAP** permitem a integração dinâmica dos componentes, dentro do mesmo processo. Uma vez que a criação de processos e a troca de informação entre os mesmos são mecanismos lentos, quando comparados com o carregamento dinâmico, consegue-se um melhor desempenho que na maioria das ferramentas para linguagens semelhantes. Embora o sistema proposto nesta dissertação possa ser mais eficiente, em especial quando existe muita comunicação entre os componentes, não existe uma separação clara entre os diversos componentes. A falta de separação entre os componentes permite que estes partilhem informação, de uma forma difícil de controlar, complicando a sua eliminação e substituição. A linguagem **MAP** efectua a eliminação e substituição de componentes usando uma política optimista, ou seja, sem garantias de consistência. Desta forma, admite-se que possam ficar residentes, na árvore de nomes, referências para entidades que já não existem. Contudo, se for o caso, é possível detectar a ausência do componente e assumir que o nome não está associado a nenhuma entidade.

Se todas as entidades de uma dado componente estiverem referidas por nomes existentes num mesmo ramo da árvore, a eliminação do referido componente permite a supressão desse mesmo ramo. A utilização de sinónimos a partir de outros ramos da

árvore, embora não sejam também suprimidos como consequência da eliminação do componente, deixam de referir nomes válidos da árvore pelo que a sua resolução gera um erro. A utilização da pilha de dados para transferir valores, mas não referências, é a forma mais segura de garantir a consistência dos dados. Note-se que a utilização, na pilha de dados, de referências para componentes eliminados não é sujeita a qualquer tipo de verificação produzindo um erro de execução e a possível terminação de toda a aplicação.

A possibilidade de os diversos componentes poderem corromper a informação dos restantes componentes da aplicação é a maior desvantagem desta forma de carregamento dinâmico. No entanto, a utilização de componentes de alta qualidade e extensivamente testados torna a solução viável em soluções comerciais. Aliás, existem vários exemplos onde, embora os componentes não sejam carregados dinamicamente, a possibilidade de um componente corromper a informação de outro é idêntica à solução proposta. A utilização de bibliotecas de rotinas matemáticas ou gráficas e de acesso a base de dados são alguns exemplos de integração onde a corrupção da informação alheia é rara e, geralmente, não imputável ao componente. Por fim, é interessante referir que o carregamento de componentes começa a vulgarizar-se, sendo exemplo as últimas versões do navegador da internet `netscape` e do sistema operativo `linux`.

6.5 Síntese

A linguagem de integração utiliza quatro tipos de entidades – identificador, referência, tipo e sinónimo – que estabelecem as relações possíveis entre um espaço de nomes e um espaço de dados. A representação dos tipos de dados como caminhos na hierarquia de nomes facilita a descrição e conversão entre os tipos de dados definidos por cada componente. Tal representação permite à linguagem oferecer mecanismos de sobreposição de operadores e a utilização de técnicas de polimorfismo paramétrico e de inclusão.

A representação dos dados partilháveis numa hierarquia de identificadores, permite a qualquer componente obter a informação que necessita e disponibilizar toda a informação que possa ser relevante para outros. Aplicações, onde os componentes recorram

a protocolos e interfaces distintas para a sua comunicação, comprometem a sua evolução e a sua manutenção. O ambiente proposto recorre à passagem de argumentos numa pilha de dados onde os componentes recolhem e depositam a informação a trocar nas suas interacções.

A linguagem oferece igualmente mecanismos de modificação incremental necessários às técnicas de especialização. Estes mecanismos incluem a resolução dinâmica de nomes, o controlo da visibilidade dos nomes e o reencaminhamento de nomes através de sinónimos. O carregamento dinâmico de componentes oferece possibilidades adicionais de controlo e configuração dinâmica da aplicação.

Capítulo 7

Ambiente de reutilização

Neste capítulo é descrito o ambiente de reutilização desenvolvido e as ferramentas que lhe dão forma, para realizar a metodologia de reutilização descrita nos capítulos anteriores. O ambiente de reutilização proposto reflecte algumas opções e decisões de ordem prática tomadas na realização das ferramentas que lhe dão forma. As decisões resultam de soluções que facilitam o desenvolvimento das ferramentas ou a sua utilização.

Para descrever o ambiente realizado seguiu-se a sequência descrita na secção 1.2.3 na apresentação do processo de reutilização, que corresponde à perspectiva seguida na organização da dissertação. Assim, começa-se por produzir árvores de classificação a partir de descrições **UML** e **C++** de componentes previamente produzidos. A classificação é um conceito central às aproximações orientadas para objectos, o que dirigiu a escolha para as linguagens **UML** e **C++**. Seguidamente, procede-se à selecção de componentes com base nas árvores de classificação por forma a identificar os melhores candidatos para a situação de reutilização em causa. A adaptação do componente seleccionado é quase inevitável, mesmo que trivial, pelo que se aborda as técnicas de especialização permitidas pela linguagem **MAP**. Finalmente, integra-se o componente adaptado na ferramenta em desenvolvimento e identificam-se formas de coordenar os diversos componentes por forma a obter a aplicação desejada.

7.1 Construção de catálogos

A maioria dos catálogos de componentes existentes resultam da aplicação dos critérios enunciados na secção 1.3.2. Consequentemente, a quase totalidade dos métodos de selecção baseia-se em catálogos formados por uma taxinomia sistemática e abrangente, residindo num repositório integrado. Logo, a maioria dos métodos de selecção pressupõe uma taxa de sobreposição (ver secção 2.2) muito baixa, ou seja, existe um só componente a cobrir uma dada área do domínio – taxinomia sistemática – e existe pelo menos um componente a cobrir cada área do domínio – abrangência. Além disso, a informação provém quase invariavelmente de uma só linguagem de especificação ou de programação, facilitando a construção dos catálogos.

No capítulo 4 definiu-se um processo de classificação, baseado no modelo enunciado no capítulo 3, que permite classificar componentes de diferentes origens com níveis de granularidade variáveis. Os mecanismos de classificação apresentados em 4.1 permitem registar com pormenor cada componente – riqueza da descrição (ver 2.2) – permitindo ao algoritmo de selecção tratar componentes com taxas de sobreposição elevadas. Para tal definiram-se, em 4.2, algoritmos que permitem, a partir de descrições **UML** ou **C++** dos componentes, obter a árvore de classificação de cada componente.

As árvores de classificação de cada componente – **CCT** [1] – utilizam a mesma estrutura do modelo de dados da linguagem **MAP** (ver 6.1.2), mas onde os valores representam o número da linha do ficheiro **UML** ou **C++** onde o identificador classificado se encontra. Uma vez que a linguagem **MAP** já possuía mecanismos para ler e escrever as árvores do modelo de dados, utilizou-se o mesmo formato de dados para introduzir as **CCT**s para o processo de selecção. Assim, as ferramentas de classificação geram **CCT**s com o formato

$$número - de - linha \qquad caminho/identificador$$

onde o *número-de-linha* representa a linha do ficheiro onde o *identificador* se encontra, e o *caminho* representa a sequência de identificadores que define o contexto do *identificador* classificado de acordo com o algoritmo enunciado em 4.2.

[1] Component Classification Tree.

7.1.1 Classificação de componentes UML

A Linguagem de Modelação Unificada – **UML** – é uma linguagem gráfica [BJR97a, BJR99] Desta forma, as ferramentas existentes guardam os seus dados num formato próprio e não documentado, produzindo apenas ficheiros de impressão (geralmente em **PostScript**) e ficheiros de codificação (em **C++** ou outras linguagens de programação). No entanto, para se proceder à classificação de componentes na fase de desenho torna-se necessário dispor de descrições textuais de **UML** ou ter acesso às estruturas de dados internas de uma ferramenta gráfica **UML**.

Embora desenvolver uma ferramenta gráfica **UML** seja uma solução atractiva optou-se pela solução mais simples, que consiste em exprimir textualmente a informação existente na representação gráfica. A sintaxe da representação textual da informação estrutural do diagrama de classes **UML** é apresentada no apêndice B.

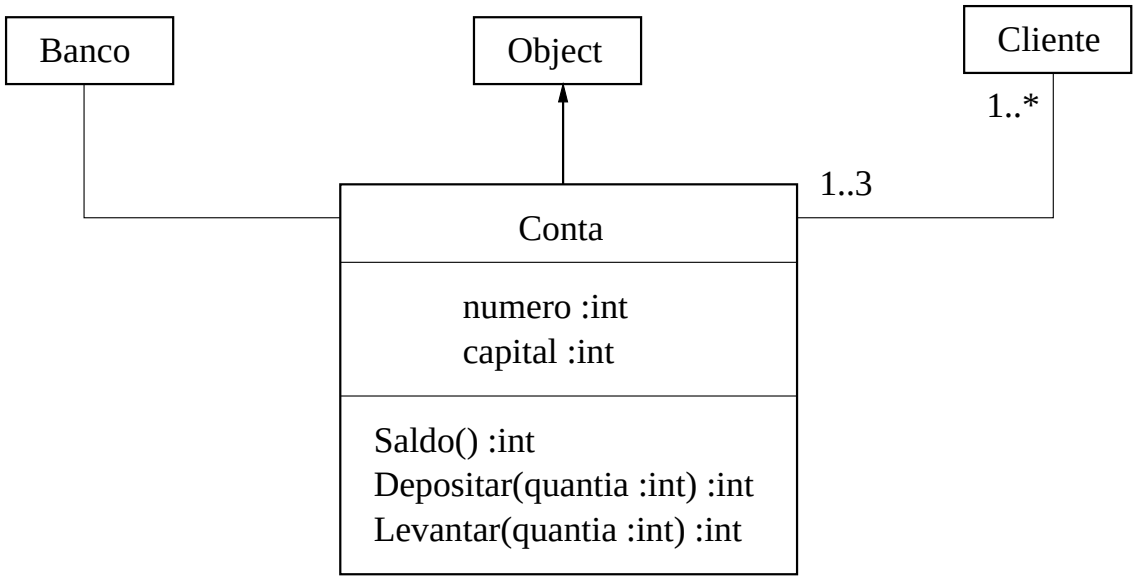


Figura 7.1: Diagrama de classes **UML** da classe *Conta* do `minibanco`.

Para exemplificar o processo de classificação utiliza-se uma descrição mais detalhada da classe *Conta* com base no exemplo 4.1. Assim, a classe *Conta* da figura 7.1 é descrita textualmente por:

class Conta
- - heranca
:Object;

- - atributos
numero:int;
capital:int;

- - operacoes
Saldo():int;
Depositar(quantia:int):int;
Levantar(quantia:int):int;

- - associacoes
banco^Banco;
clientes[1..3]^Cliente[1..*];

A conversão do formato textual para a **CCT** é executada pela ferramenta **uml2tree** desenvolvida no âmbito do trabalho de doutoramento que, a partir de uma ou mais descrições textuais **UML**, produz uma representação com o formato do modelo de dados da linguagem **MAP**.

Exemplo 7.1 *O ficheiro com a representação da árvore obtida da descrição textual do diagrama* **UML** *é dada por:*

```
16 /Conta/assocs/clientes/Cliente
15 /Conta/assocs/banco/Banco
12 /Conta/ops/Levantar/int
12 /Conta/ops/Levantar/args/quantia/int
11 /Conta/ops/Depositar/int
11 /Conta/ops/Depositar/args/quantia/int
10 /Conta/ops/Saldo/int
7 /Conta/attribs/capital/int
```

```
6 /Conta/attribs/numero/int
3 /Conta/assocs/Object/inherit
```

A representação gráfica que lhe corresponde, obtida à semelhança da figura 4.1, e que será utilizada pela linguagem **MAP** no processo de selecção, é apresentada na figura 7.2.

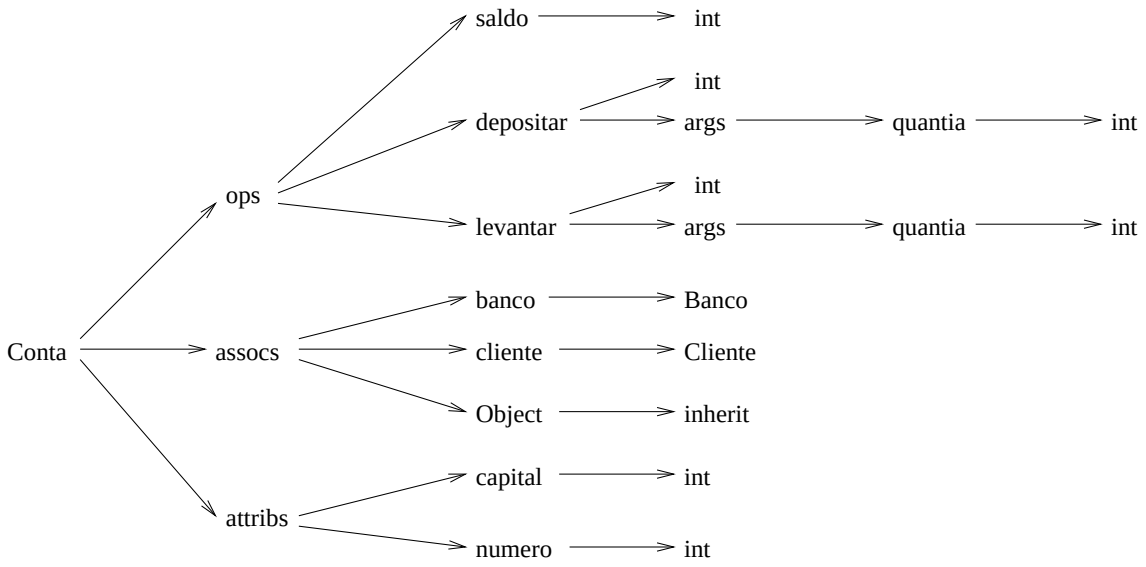


Figura 7.2: Hierarquia de identificadores obtida da figura 7.1.

7.1.2 Classificação de componentes C++

A produção de **CCT**s a partir de componentes descritos na linguagem **C++** segue o mesmo processo que no caso anterior. Assim, construiu-se uma ferramenta – **cpp2tree** –, com a mesma interface que a ferramenta **uml2tree**, e que produz um **CCT** a partir de um ou mais ficheiros de código ou de declarações.

A leitura dos ficheiros **C++** utiliza um processador de linguagem desenvolvido por terceiros e baseado exclusivamente nas ferramentas de gestão de analisadores lexicais e sintácticos **lex** e **yacc** [RD95, SJ85]. Esta realização tem a vantagem de ser bastante simples atendendo à complexidade sintáctica da linguagem **C++**, embora tenha algumas limitações que não comprometem o algoritmo de classificação. As primeiras limitações prendem-se com o processamento de padrões ("*templates*") e o tratamento de condições

de excepção, não sendo essenciais no estudo que se pretendeu efectuar sobre a classificação com base em identificadores. A restante limitação, necessária para reduzir o número de conflitos do processador de linguagem utilizado, estabelece a distinção entre identificadores e tipo de dados definidos pelo utilizador. Assim, todos os tipos de dados definidos pelo utilizador deverão ter a primeira letra maiúscula, e todos os restantes identificadores deverão ter a primeira letra minúscula. Esta limitação, que obrigou a algumas alterações nos ficheiros utilizados para teste, pode ser eliminada através da construção de uma tabela de tipos de dados. A construção de uma tabela de tipos de dados obriga a uma interpretação, mesmo que elementar, da semântica do componente.

No entanto, verificou-se que a quantidade de informação disponibilizada nas **CCT**s com origem em **C++** dispunha de informação em excesso para o processo de selecção. Logo, a selecção acaba por ser efectuada por uma **CCT** reduzida pelas operações de selecção e que contém aproximadamente a mesma informação disponibilizada nas **CCT**s de **UML**. A principal informação adicional fornecida pelas **CCT**s de **C++** consiste na lista de chamadas que determinada função realiza, embora esta informação se traduza, em descrições **UML** completas, em **associações**. Além disso, a informação disponibilizada pelas chamadas a rotinas exteriores é difícil de interpretar em **C++** devido ao polimorfismo e à resolução dinâmica de nomes.

De um ponto de vista comparativo, a classificação com base em **UML** apresentou um grau de sucesso muito superior ao das descrições com base **C++** o que desmotivou a continuação do trabalho na classificação de componentes em **C++**. Convém referir que existem ferramentas comerciais que produzem representações **UML** a partir de descrições **C++**, embora estas ferramentas não tenham sido utilizadas neste trabalho devido ao seu elevado custo.

7.1.3 Manipulação da árvore de classificação

O processo de classificação é avaliado segundo os critérios enunciados em 2.2, nomeadamente quanto ao esquema de classificação (ver página 40) e à riqueza da descrição (ver página 37). No esquema de classificação proposto a *correcta descrição do componente* é garantida, pois a informação tem por base elementos directamente extraídos

do componente e não se baseia em nenhum esquema subjectivo. A *objectividade dos critérios de classificação* é assegurada através dos algoritmos de classificação descritos em 4.2. A *atribuição de um lugar único na organização* não é aplicável a esquemas de classificação que suportem taxas de sobreposição elevadas pois, nestes casos, existirão diversos componentes para cada área do domínio de aplicação. A existência de um lugar único pressupõe que os catálogos são formados por uma taxinomia sistemática e abrangente, o que não é o caso presente tal como foi afirmado em 7.1. Finalmente, a riqueza da descrição depende da capacidade em determinar de uma forma rigorosa as características estáticas, dinâmicas, métricas, funcionais e não funcionais. Uma vez que o esquema de classificação proposto se baseia na construção de hierarquias de identificadores, a informação estática está subjacente à estrutura dos identificadores e o nome dos identificadores pode dar indicações quanto à sua utilização funcional. As restantes características não são abrangidas pelo esquema de classificação automática proposto. Assim, torna-se necessário enriquecer a descrição segundo os critérios estabelecidos em 4.1.3 através da introdução manual de anotações à **CCT** como apresentado em 4.2.3.

Como o sistema proposto se encontra numa fase experimental não se desenvolveu uma ferramenta para a introdução manual de anotações. No entanto, esta operação foi realizada de duas formas: através da utilização de um editor de texto para manipular os **CCT**s e através da utilização da linguagem **MAP**. Embora a linguagem **MAP** só surja no processo de reutilização, ou seja, nas fases de selecção, especialização e integração, é possível ler o **CCT**, fazer a manipulação da árvore e reescrever o **CCT**. As funções de manipulação de árvores relevantes para processo de edição manual resumem-se a *bind*, *dir* e *lookup* e *forget* descritas em 6.3.2, e cujas assinaturas se apresenta em A.4. Qualquer que seja o método seguido convencionou-se utilizar valores negativos para distinguir as anotações manuais dos elementos de classificação automática que utilizam o número de linha do ficheiro original e o valor 0 (zero) para os elementos intermédios da árvore sem correspondência no ficheiro original.

7.2 Recuperação de componentes

O processo de selecção pretende escolher, de entre os componentes disponíveis no catálogo, o que melhor se adapta às necessidades de reutilização. O número de componentes recuperados deverá diminuir nas sucessivas iterações do processo de selecção até que apenas um componente seja considerado para o processo subsequente de especialização e integração. Este componente deverá ser aquele que melhor corresponde aos requisitos enunciados na formulação da pergunta.

A utilização da linguagem **MAP** no processo de selecção pressupõe um conjunto de passos a desenvolver nesta secção e que se resumem no seguinte método:

1. definir dicionário de sinónimos que se adapta aos termos da área em estudo.
2. restringir cada **CCT** do catálogo utilizando uma combinação das 6 operações de selecção definidas em 5.1.3..
3. formular a pergunta, através da escolha dos nomes e respectivos pesos, que melhor modela os requisitos segundo os critérios apresentados em 5.3.
4. calcular a distância de cada componente através da aplicação de cada **CCT** transformado à pergunta, de acordo com a expressão 5.1 da página 126.
5. ordenar as distâncias obtidas para identificar os componentes considerados mais próximos dos requisitos enunciados na formulação da pergunta.
6. se nenhum componente se distinguir dos restantes ou, se através de sucessivas interações não existir um componente que persista nos primeiros lugares da ordenação, repetir o processo desde 2 modificando a transformação sobre as **CCT**s ou os pesos da pergunta.

7.2.1 Operações de selecção

As operações de selecção definidas em 5.1.3 manipulam uma ou mais árvores que são inicialmente construídas a partir das **CCT**s. Estas árvores são manipuladas pelas operações base do tipo de dados `/sys/tree`, a que se acrescentam as 6 operações definidas

em 5.1.3. Todas as 6 operações baseiam-se na função `/sys/tree/all` que corresponde à função de interface `mapAll` definida em A.3.

A função `/sys/tree/all` percorre todos os vértices da sub-hierarquia com origem num dado vértice – `origem` – até um comprimento indicado – `nível` –, segundo a terminologia definida em 4.3.2. Para cada vértice encontrado que tenha informação associada, ou seja, um identificador segundo 6.1.2, é chamada uma função de tratamento fornecida como argumento. As diversas operações de selecção fundamentam-se na manipulação dos três argumentos apresentados. Cada uma destas operações é apresentada com a notação de efeito na pilha de dados ("*stack effect*") segundo [Kna93] e descrita em A.4.

Operação 7.1 (Granulação)

Pilha anterior	*nível:/sys/integer*	*árvore:/sys/tree*
Operação	`/sys/tree/gran`	
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*

A operação de *granulação*, enunciada na definição 5.1, determina como `origem` a raiz e como `nível` o valor do argumento. A função de tratamento reduz-se à cópia incondicional para a árvore resultante de todos os vértices, para as quais é invocada. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.2 *Visualização em* **MAP** *da operação de granulação a partir do* **CCT** *do exemplo 7.1 através da sequência:*

```
"conta.cct" /sys/tree/open 2 swap gran drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
0 Conta/assocs
0 Conta/ops
0 Conta/attribs
```

Os número de linha são 0 (zero) pois não identificam um elemento específico da descrição, representam apenas sub-caminhos para um ou mais elementos da descrição.

Operação 7.2 (Particularização)

Pilha anterior	*caminho:/sys/text*	*árvore:/sys/tree*
Operação	`/sys/tree/part`	
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*

A operação de *particularização*, enunciada na definição 5.2, determina como `origem` o vértice correspondente ao extremo do caminho indicado no argumento, como `nível` o valor -1 (todos os níveis). A função de tratamento é a mesma da operação granulação e efectua a cópia incondicional para a árvore resultante de todos os vértices, para as quais é invocada. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.3 *Visualização em* **MAP** *da operação de particularização a partir do* **CCT** *do exemplo 7.1 através da sequência:*

```
"conta.cct" /sys/tree/open
"/Conta/assocs" swap part drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
0 Conta/assocs
0 Conta/assocs/clientes
16 Conta/assocs/clientes/Cliente
0 Conta/assocs/banco
15 Conta/assocs/banco/Banco
0 Conta/assocs/Object
3 Conta/assocs/Object/inherit
```

Operação 7.3 (Localização)

Pilha anterior	*nome:/sys/text*	*árvore:/sys/tree*
Operação	`/sys/tree/local`	
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*

A operação de *localização*, enunciada na definição 5.3, determina como `origem` a raiz, como `nível` o valor -1 (todos os níveis). A função de tratamento adiciona à árvore

resultante os caminhos onde o nome associado ao último arco for igual ao *nome* do argumento. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.4 *Visualização em* **MAP** *da operação de granulação a partir do* **CCT** *do exemplo 7.1 através da sequência:*

```
"conta.cct" /sys/tree/open
"args" swap local drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
0 Conta/ops
0 Conta/ops/Levantar
0 Conta/ops/Levantar/args
0 Conta/ops/Depositar
0 Conta/ops/Depositar/args
```

Operação 7.4 (Qualificação)

Pilha anterior	*nome:/sys/text*	*árvore:/sys/tree*	
Operação	`/sys/tree/qualif`		
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*	

A operação de *qualificação*, enunciada na definição 5.4, determina como `origem` o vértice anterior ao vértice correspondente ao extremo do caminho indicado no argumento, como `nível` o valor 1. A função de tratamento reduz-se à cópia incondicional para a árvore resultante de todos os vértices, para as quais é invocada. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.5 *Visualização em* **MAP** *da operação de qualificação a partir do* **CCT** *do exemplo 7.1 através da sequência:*

```
"conta.cct" /sys/tree/open
"/Conta/ops/Saldo" swap qualif drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
```

```
0 Conta/ops
0 Conta/ops/Levantar
0 Conta/ops/Depositar
0 Conta/ops/Saldo
```

Operação 7.5 (União)

Pilha anterior	*árvore1:/sys/tree*	*árvore2:/sys/tree*
Operação	`/sys/tree/uniao`	
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*

A operação de *união*, enunciada na definição 5.5, executa duas vezes a função `/sys/tree/all` para cada um dos argumentos determinando, em ambos os casos, como `origem` a raiz e como `nível` o valor -1 (todos os níveis). A função de tratamento reduz-se à cópia incondicional para a árvore resultante de todos os vértices, para as quais é invocada. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.6 *Visualização em* **MAP** *da operação de união a partir do resultado dos exemplos 7.3 e 7.5 através da sequência:*

```
"conta.cct" /sys/tree/open dup
"/Conta/ops/Saldo" swap qualif drop swap
"/Conta/assocs" swap part drop
uniao drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
0 Conta/assocs
0 Conta/assocs/clientes
16 Conta/assocs/clientes/Cliente
0 Conta/assocs/banco
15 Conta/assocs/banco/Banco
0 Conta/assocs/Object
3 Conta/assocs/Object/inherit
0 Conta/ops
```

```
0 Conta/ops/Levantar
0 Conta/ops/Depositar
0 Conta/ops/Saldo
```

Operação 7.6 (Diferenciação)

Pilha anterior	*árvore1:/sys/tree*	*árvore2:/sys/tree*
Operação	`/sys/tree/difer`	
Pilha posterior	*resultado:/sys/tree*	*contagem:/sys/integer*

A operação de *diferenciação*, enunciada na definição 5.6, determina como `origem` a raiz, como `nível` o valor -1 (todos os níveis). A função de tratamento copia para a árvore resultante os caminhos correspondentes aos vértices, para as quais é invocada, se estes não existirem na segunda árvore. A operação devolve ainda o número de cópias efectuadas.

Exemplo 7.7 *Visualização em* **MAP** *da operação de diferenciação a partir do resultado dos exemplos 7.4 e 7.5 através da sequência:*

```
"conta.cct" /sys/tree/open dup
"args" swap local drop swap
"/Conta/ops/Saldo" swap qualif drop
difer drop "/" swap print
```

produz o resultado seguinte:

```
0 Conta
0 Conta/ops
0 Conta/ops/Saldo
```

Para se obter a diferença contrária basta preceder a operação `difer` *de uma operação* `swap`*.*

As operações acima exemplificadas permitem obter árvores de dimensão e profundidade adaptadas ao tipo de pergunta a efectuar, em especial no que diz respeito à sobreposição dos vários componentes. Assim, inicialmente começa-se com árvores de pouca profundidade e geralmente limitadas às operações (sem argumentos) e associações. Caso se verifique uma elevada sobreposição, aumenta-se a dimensão e profundidade das

árvores por forma a incluir mais informação que permita distinguir componentes semelhantes, num processo de refinamento iterativo como descrito em 5.3.2.

7.2.2 Dicionário de sinónimos

Uma vez que o método de selecção proposto se baseia num esquema de classificação de vocabulário não controlado é necessário recorrer a um dicionário de sinónimos. O dicionário de sinónimos pesados assimétrico, conforme descrito em 5.1.2, define que o peso entre os sinónimos **A** e **B** pode não ser o mesmo que entre **B** e **A**, como apresentado na figura 5.2. Tal é o caso de um termo mais geral poder ser considerado um sinónimo total de um termo mais específico, mas não o contrário.

O dicionário de sinónimos a utilizar no processo de selecção estabelece, para um dado termo, quais os seus sinónimos e respectivos pesos. Cada termo corresponde a uma linha do ficheiro que estabelece os sinónimos, ou *aliases*, a utilizar numa dada área de aplicação.

```
termo      peso1 alias1      peso2 alias2      ...      pesoN aliasN
```

A utilização de expressões regulares é possível, tal como indicado em 5.1.2 e 5.3.1, e apresentado no exemplo 5.1. No entanto, apenas os sinónimos podem conter expressões regulares, pois a formulação da pergunta recorre também a expressões regulares. Esta limitação prende-se com o facto de a função de tratamento de expressões regulares não permitir comparar uma expressão regular com outra. A utilização de expressões regulares facilita a determinação de equivalências sem depender do tempo do verbo, género ou número do adjectivo, substantivo, *etc.*, como é especialmente comum em línguas latinas como a língua portuguesa. Contudo, deixa de ser possível utilizar funções de dispersão, ou outros mecanismos equivalentes para acelerar a procura, sendo necessário iterar sistematicamente pelo dicionário.

Exemplo 7.8 *Um dicionário de sinónimos permite determinar o grau de sinonímia entre dois termos, podendo servir inclusivamente para estabelecer equivalências entre termos de línguas*

distintas.

```
valor     .5 quantidade  .7 quantia
saldo     .3 capital
capital   .5 saldo
morada    .8 endereco    .5 local*
calcular  .8 avalia*     .7 determina*
obtem     .8 saber       .7 retorna     .6 saber  .4 da  .2 fazer
juro      .9 interest
```

Operação 7.7 (Dicionário)

Pilha anterior		*nome:/sys/text*
Operação	`/sys/select/thesaurus`	**Pilha posterior**

O carregamento do *dicionário* é executado uma só vez pois na presente realização existe um único dicionário de sinónimos acessível em cada momento, facto que, dado o problema a resolver não se torna limitativo.

Operação 7.8 (Sinónimo)

Pilha anterior		*nome:/sys/text* *alias:/sys/text*
Operação	`/sys/select/alias`	
Pilha posterior		*peso:/sys/real*

O acesso ao dicionário é efectuado através da operação *sinónimo* que permite determinar o peso entre os nomes passados como argumento. Caso não exista equivalência entre os dois nomes será devolvido o valor **0.0**.

Operação 7.9 (Expressão regular)

Pilha anterior		*correcto:/sys/text* *expregular:/sys/text*
Operação	`/sys/select/wildcard`	
Pilha posterior		*igual:/sys/bool*

A operação sinónimo recorre à operação *expressão regular* para determinar se um nome obedece a uma expressão regular. Esta operação pode ser acedida genericamente e, nomeadamente, utilizada em outras fases do processo de selecção.

7.2.3 Selecção dos componentes

O processo de selecção consiste em obter o componente que implica menores custos de adaptação de entre os disponíveis no catálogo. Minimizar o custo de adaptação consiste em minimizar a função que determina a distância entre os requisitos expressos e a descrição do componente. A distância cognitiva, tal como foi definida em 2.5, não pode ser quantificada pelo que se recorre a uma quantificação com base na comparação pesada de nomes de identificadores. A função distância quantificada, tal como foi definida em 5.1, calcula uma estimativa do custo de adaptação de cada componente face aos requisitos expressos. A ordenação das distâncias calculadas permite seleccionar o componente (ou componentes) que apresenta maiores semelhanças com os requisitos expressos. Consequentemente, minimizar a função distância consiste em maximizar a função semelhança.

Função semelhança

A utilização de uma função semelhança, em vez de uma função distância, facilita a computação e simplifica a aplicação, pois:

- a não existência de igualdades não contribui para o valor final da função, pelo que se contabiliza apenas os requisitos verificados no componente.
- os pesos da característica tomam valores negativos para rejeição e valores positivos tanto maiores quanto a sua importância relativa face às restantes características.
- os pesos da semelhança e os pesos da característica podem ser directamente multiplicados para obter cada uma das parcelas da função semelhança.

O cálculo da função semelhança consiste em comparar um termo dado com os diversos nomes dos identificadores que constituem a hierarquia do **CCT**, tendo em conta os pesos dos sinónimos existentes no dicionário. Tal como nas operações de selecção descritas em 7.2.1, também o cálculo da função semelhança se baseia na função `/sys/tree/all`.

Neste caso, cada nome da árvore é comparado com o termo dado e, caso não exista semelhança, recorre-se ao dicionário de sinónimos. Sempre que existe uma igualdade o valor do peso associado a esse sinónimo, ou o valor unitário no caso da semelhança directa, é adicionado ao resultado da função semelhança. Assim, quanto mais ocorrências de um dado nome existir maior o valor da função. Neste ponto, o método proposto assemelha-se a alguns métodos baseados em métricas apresentados em 2.2.4.

Formulação da pergunta

A formulação da pergunta consiste no encadeamento pesado dos diversos termos que constituem a modelação dos requisitos. Para cada um desses termos é cálculada a função semelhança e afectada do peso resultante do encadeamento. O valor final utilizado para a ordenação, com base na distância, é obtido pela soma dos resultados das sucessivas invocações da função semelhança para os termos existentes na pergunta.

A pergunta é constituída por um peso e uma lista de nomes que são afectados desse peso. Cada nome da lista pode ser substituído por uma nova pergunta, sendo o seu peso multiplicado pelo peso da pergunta inicial. Recorrendo à linguagem **MAP** a pergunta é formulada através de uma matriz, do tipo `/sys/array`, com a forma:

```
[ peso1 nome1 ... [ peso2 [ peso3 nome3 ... ] nome2 ... ] nome1b ... ]
```

onde o `nome3` é afectado de um peso igual a $peso1 \times peso2 \times peso3$. Qualquer dos nomes pode ser substituído por uma expressão regular que será utilizada pela função semelhança. Na realização presente os pesos são valores reais, do tipo `/sys/real`, e não números naturais, como se descreve no capítulo 5, para facilitar o processo de refinamento. Este processo, descrito na secção 5.3.2, necessita de pequenas alterações nos valores dos pesos por forma a melhor exprimir na pergunta as restrições dos requisitos.

Exemplo 7.9 *Uma sequência completa, muito simples, que permite obter um valor para distância, carrega o dicionário, constrói a matriz da pergunta e utiliza uma* **CCT** *não modificada:*

```
"sinon" /sys/select/thesaurus
```

```
[ 0.12 "int" ] "g52uml.cct" /sys/tree/open /sys/select/pergunta print
```

produz como resultado o valor de **1.92**, *que corresponde à soma aritmética das 16 ocorrências directas do nome* `int` *na árvore.*

7.2.4 Apreciação do método de selecção

Os métodos de selecção devem ser avaliados tendo em conta os critérios expostos em 2.2, nomeadamente em relação à determinação da semelhança, formulação da pergunta, interface com o utilizador e processo de recuperação. A determinação da semelhança pretende produzir uma estimativa do custo de adaptação, ou seja, converter a percepção da distância cognitiva numa medida. Tal como foi referido na página 37, as técnicas difusas são menos sensíveis a ambiguidades e imprecisões da pergunta, pelo que a utilização de palavras de um vocabulário não controlado flexibiliza o trabalho de selecção do utilizador. De igual forma, a formulação da pergunta é maleável e garante bastante liberdade aos mecanismos de selecção. A interface com o utilizador apresenta um bom poder descritivo e adapta-se bem às preferências do utilizador, mas a notação postfixada da linguagem **MAP** dificulta a facilidade de manejo e a curva de aprendizagem da maioria dos utilizadores. Os critérios de apreciação do processo de recuperação – facilidade de utilização da linguagem de interrogação, flexibilidade de acesso ao repositório, adaptabilidade aos requisitos do utilizador, possibilidade de refinar, buscar e de combinar métodos – são na sua maioria garantidos.

Análise de desempenho

O desempenho do método de busca não contituiu uma prioridade da realização presente. De facto, considerou-se que primeiro era necessário avaliar a utilidade do método e só numa segunda fase se deverá determinar quais os mecanismos do método cuja optimização contribui significativamente para o desempenho global do método.

Na ferramenta desenvolvida no âmbito do trabalho de doutoramento o tempo de busca é proporcional a $Q \times V \times C \times \frac{D}{2}$ onde Q é o número de termos existentes na pergunta,

V é o número médio de vértices dos **CCT**s do catálogo, C é o número de componentes resultantes da iteração anterior e, D é o número de sinónimos do dicionário.

O reduzido desempenho do método de busca deriva, numa primeira aproximação, da utilização de expressões regulares. A utilização destas expressões obriga a que a busca no dicionário de sinónimos seja sequencial. Contudo, a elevada dispersão de nomes utilizados na caracterização dos identificadores exige a sua aplicação para o correcto funcionamento do método proposto.

Exemplo de aplicação

Foi efectuado um estudo preliminar do método de selecção, recorrendo aos trabalhos dos alunos da disciplina de *Programação com Objectos* da *Licenciatura em Engenharia Informática e Computadores* do IST, no ano lectivo 1996/97. O trabalho da disciplina consistia num sistema bancário, semelhante ao exemplo 3.1, do qual se entregava uma descrição **UML** em papel e uma realização em **C++**. Os trabalhos **UML** de alguns dos mais de 30 grupos foram convertidos para a notação descrita em B e o código **C++** corrigido nos pontos que entravam em conflito com as restrições do processador descrito em 7.1.2.

Estes trabalhos, por terem origem na mesma especificação, eram necessariamente semelhantes pelo que permitiam testar a capacidade da ferramenta em distinguir pormenores e avaliar a sua sensibilidade aos nomes escolhidos pelos alunos. Um primeiro estudo com base na dimensão das árvores produzidas a partir de código **C++**, embora não seja uma métrica aceitável, permite especular quanto à complexidade da solução. Assim, árvores com muitos elementos estão frequentemente associadas a código de fraca qualidade, enquanto árvores pequenas representam soluções incompletas ou de boa qualidade.

A localização de identificadores específicos foi extremamente importante para ter uma ideia da funcionalidade. No entanto, poucos foram os identificadores que tiveram uma aceitação geral, impedindo uma amostragem uniforme. A escolha dos nomes verificou-se ser determinante no sucesso da metodologia. Algumas soluções optaram por uma escolha especialmente infeliz de nomes, tornando a sua busca e emparelhamento através

do dicionário de sinónimos quase impossível. Contudo, verificou-se que estas soluções estavam, quase invariavelmente, associadas a trabalhos de fraca qualidade técnica.

A metodologia proposta apresentou resultados bem mais satisfatórios para as soluções bem comportadas, que representavam a maioria dos trabalhos de boa qualidade. É pois aceitável especular que grupos de trabalho experientes e com bons conhecimentos de engenharia de software, em especial na perspectiva de reutilização, produzam componentes de boa qualidade e que recorram a nomes coerentes e espectáveis facilitando o processo de selecção proposto [dSC98b, dSC98a].

7.3 Desenvolvimento de adaptações

Após o processo de selecção, é escolhido um componente que terá, quase invariavelmente, de ser adaptado para poder ser integrado na aplicação a desenvolver. As adaptações podem ser efectuadas de três formas distintas: na linguagem original do componente, como parte do processo de especialização ou, através da linguagem **MAP** após o processo de integração. Se as alterações forem efectuadas na linguagem original do componente recuperado deverão ser utilizadas as técnicas disponibilizadas pela linguagem em questão tal como exposto em 2.3.1. Caso se utilize a linguagem **MAP** para efectuar as alterações, estas podem ser consideradas como a conclusão do processo de integração e serão tratadas em 7.4.3. Nesta secção trataremos do processo que o componente sofre para poder ser sujeito ao processo de integração, neste caso para ser integrado na ferramenta que realiza a linguagem **MAP**.

7.3.1 Construção de rotinas de interface

Para que uma rotina desenvolvida noutra linguagem possa interagir com a pilha de dados descrita em 6.1.3, e assim trocar informação com outros componentes que integrem a aplicação, é necessário construir rotinas de interface. Estas rotinas de interface poderão, no futuro, ser geradas, total ou parcialmente, por uma ferramenta especializada, libertando o programador de um trabalho frequentemente repetitivo. A informação disponível na declaração da rotina é, para a maioria das outras linguagens a utilizar e

para grande parte das rotinas a integrar, suficiente para construir a rotina de interface. Contudo, na presente realização, o desenvolvimento de rotinas de interface é executado manualmente.

As rotinas de interface podem ser agrupadas num ficheiro que as regista em conjunto, no momento do carregamento descrito no processo de integração, na árvore de nomes descrita em 6.1.2. A função de registo deverá ter o mesmo nome do ficheiro objecto a carregar (ver 7.4.1) e deverá registar todas as funções de interface que existam nesse ficheiro objecto. Uma vez registadas, as rotinas de interface podem ser invocadas quer através de programas escritos na linguagem **MAP** quer através de outros componentes da aplicação. Ao ser invocada, a rotina de interface, deverá retirar da pilha de argumentos os elementos necessários para a invocação da rotina desenvolvida noutra linguagem, colocando os elementos devolvidos por esta última na mesma pilha de argumentos. As funções de interface disponíveis para interagir com a ferramenta que realiza a linguagem **MAP** e nomeadamente com a sua pilha de argumentos encontram-se descritas no apêndice A.

Exemplo 7.10 *A função* **C** *que permite efectuar a verificação de expressões regulares simples utilizada em 7.2.2:*

```
int wildcard(char *correct, char *wildcarded);
```

pode ser adaptada para integração na ferramenta **MAP** *através da rotina de interface:*

```
static int Wildcard(Map *map, Entity *ent, void *user)
{
  Entity *wild;
  int ret;

  if ((wild = mapFetch(map, 2)) == 0) return mapError(map, MAP_UNDFL);
  if (wild->path != map_text) return -1;
  if ((ent = mapFetch(map, 1))->path != map_text) return -1;
  mapDrop(map, 2);
  ret = wildcard(wild->data.text, ent->data.  text);
  mapPush(map, map_integer)->data.integer = ret ? 1 : 0;
  return 1;
```

```
}
```

sendo o registo da rotina de interface efectuada por:

```
int selec(Map *map)
{
  Entity *e = mapName(map, "/sys/select/wildcard", map_code);
  e->data.func = Wildcard;
  return 1;
}
```

7.3.2 Definição de novos tipos de dados

A definição de um novo tipo de dados, tal como descrito em 6.2.2 consiste na escolha de uma representação estrutural e de um conjunto de operações que conferem um comportamento comum aos elementos desse tipo. A definição das operações é realizada, quer seja sobre um tipo de dados existente ou novo, da mesma forma que foi desenvolvido em 7.3.1. O facto de estas operações se agruparem num mesmo nó da árvore de dados permite tirar partido dos mecanismos de sobreposição de operadores e polimorfismo descritos em 6.2.3, uma vez que o tipo de dados é identificado pelo caminho para esse nó (ver 6.2). O problema que ficou por tratar na definição de um novo tipo de dados consiste na representação estrutural e sua manipulação. Este pode ser dividido em estruturas simples, estruturas lineares e estruturas não lineares.

Quando um tipo de dados é representado por uma estrutura simples, ou seja, uma estrutura que ocupa o espaço disponível para os dados na entidade (ver A.3), a gestão é realizada pela ferramenta **MAP**. Neste caso, o problema da linearidade (ver 6.1.3) não é da responsabilidade do componente que define o tipo. Assim, não existe necessidade de definir uma rotina de tratamento do tipo, e esta, caso exista, deverá devolver o valor inteiro 1 (um). A rotina de tratamento do tipo reside no nó da árvore que identifica o tipo de dados.

Exemplo 7.11 *Para definir o tipo* `Ponto` *para desenho de figuras geométricas em sistemas gráficos de mapas de pontos (“*bitmap*”) constituído por dois inteiros de pequena capacidade (por exemplo, o tipo* `short` *da linguagem* **C***), pode-se utilizar a estrutura*

```
short small[2];
```

definida na estrutura **Entity** *descrita em A.3. Se escolhermos o caminho* `/sys/ponto` *para colocar o novo tipo de dados, as operações que manipulam este tipo deverão ser colocadas imediatamente após este nó, por exemplo,* `/sys/ponto/desenha`*. Embora não seja necessário incluir uma rotina de tratamento do tipo, esta, a existir, deverá residir em* `/sys/ponto` *e ter um aspecto conforme a*

```
static int Eval(Map *, Entity *, void *) { return 1; }
```

No caso de estruturas mais extensas e complexas, o espaço disponível para dados na estrutura **Entity** revela-se insuficiente, tornando-se necessária a presença de uma rotina de tratamento para o tipo a definir. Neste caso torna-se necessário distinguir a linearidade das entidades descritas pelo novo tipo de dados. Se o tipo descreve entidades lineares é necessário oferecer os mecanismos de duplicação e eliminação de entidades exigidos por uma máquina baseada numa pilha de dados como é o caso da linguagem **MAP** e da ferramenta homónima. Estas operações são uma consequência directa das operações de colocar `push` e retirar `drop` elementos na pilha de argumentos. Assim, para tipo de dados lineares, a rotina de tratamento do respectivo tipo deverá responder às mensagens MAP_COPY e MAP_DROP, efectuando as correspondentes operações de cópia e eliminação de entidades. Uma operação de cópia deverá colocar na pilha de argumentos um novo elemento com as mesmas características do que lhe é passado. Uma operação de eliminação deverá efectuar as operações necessárias à remoção da entidade indicada no argumento, tais como libertação de memória ou outros recursos, por exemplo, ficheiros de dados ou canais de comunicação.

Exemplo 7.12 *Para definir o tipo* `array` *para manipulação de matrizes utiliza-se a estrutura*

```
struct entity **array;
```

definida na estrutura **Entity** *descrita em A.3. A rotina de tratamento do tipo reside em* `/sys/array` *e tem aspecto conforme a*

```
static int Handle(Map *map, Entity *ent, void *user)
{
  switch (MAP_ADDR(user)) {
    case MAP_VALE: case MAP_EVAL: return 1; /* evaluates to itself */
    case MAP_COPY:
      mapPush(map, map_array)->data.array =
       copia_matriz(ent->data.array);
      break;
    case MAP_DROP:
      liberta_matriz(ent->data.array);
      break;
    default: return -1; /* not an expected call */
  }
  return 1;
}
```

Nos casos em que o tipo de dados manipule entidades não lineares é responsabilidade da rotina de tratamento impor as restrições necessárias, tendo presente que a informação existente no espaço disponível para dados na estrutura **Entity** é duplicada e eliminada pela ferramenta **MAP** na sequência das operações de cópia e eliminação. A passagem do controlo para o utilizador ou a utilização de métodos de reciclagem de memória como, por exemplo, a contagem de referências são utilizadas por alguns tipos de dados.

Finalmente, convém referir um tipo de dados `/sys/routine` que recorre a características especificamente desenhadas para ele, embora o mesmo mecanismo possa ser utilizado por outros tipos de dados. O tipo `routine` é utilizado para descrever operações na linguagem **MAP**, mas também para definir grupos de instruções que podem ser executadas por instruções de controlo de fluxo, como `if`, `ifelse`, `loop` ou outras. A distinção que, por exemplo o `Smalltalk-80` [GR83], faz entre blocos e métodos obvia este mecanismo. Em **MAP**, quando um elemento do tipo é calculado na sequência da resolução do nome a que estava associado, a rotina de tratamento do tipo é invocada com a constante MAP_EVAL. Quando se trata de um cálculo na sequência de uma ocorrência directa do elemento do tipo, a rotina de tratamento do tipo é invocada com a

constante MAP_VALE. Para definir o tipo `routine` utilizou-se a mesma estrutura que no caso das matrizes

```
struct entity **array;
```

mas a rotina de tratamento do tipo trata de forma distinta as ocorrências directas e as invocações por nome. Desta forma a rotina de tratamento tem aspecto conforme a

```
static int Handle(Map *map, Entity *ent, void *user)
{
  switch (MAP_ADDR(user)) {
    case MAP_VALE: return 1; /* evaluates to itself */
    case MAP_EVAL: return eval_routine(map, ent);
    case MAP_COPY:
      mapPush(map, map_array)->data.array =
       copia_matriz(ent->data.array);
      break;
    case MAP_DROP:
      liberta_matriz(ent->data.array);
      break;
    default: return -1; /* not an expected call */
  }
  return 1;
}
```

Exemplo 7.13 *Uma consequência deste tipo de tratamento é o facto de não ser indistinta a utilização de nomes ou a referência directa a entidades do tipo* `/sys/routine`*, como, por exemplo, em*

```
12 1 { print } if
12
{ print } :my_print bind
12 1 my_print if
1
Error: Stack underflow.
```

7.3.3 Utilização das técnicas de especialização

De entre as técnicas de especialização apresentadas em 2.3.1 nem todas são realizáveis em **MAP**. Por exemplo, a cópia e modificação só é possível na linguagem de origem do componente, embora seja de um ponto de vista de engenharia de software a solução menos aconselhável. A fixação de argumentos a valores constantes – particularização – é possível quer a nível do processo de especialização propriamente dito quer já na fase de coordenação entre os componentes a executar no fim do processo de integração (ver 7.4.3).

A técnicas de encapsulamento e agregação são as que mais se identificam com o processo de especialização tal como é realizado no âmbito da linguagem **MAP** e da ferramenta homónima descritas nesta dissertação. O encapsulamento é obtido quer através da utilização da pilha de argumentos para aceder às operações registadas, quer ao nível da estrutura de dados que é escondida dos restantes componentes e acedida apenas através das operações registadas. A agregação é possível através do agrupamento em torno do caminho que descreve o tipo. Tal é o caso da associação das 6 operações de selecção ao tipo `/sys/tree` já existente previamente.

Finalmente, embora a linguagem **MAP** não suporte herança no sentido genérico como é definido nas linguagens orientadas para objectos, pois não permite a extensão das estruturas de dados, permite acrescentar e alterar operações a um tipo já definido extendendo a sua funcionalidade. Esta possibilidade, através da delegação de espaços de nomes permite obter algumas das vantagens organizativas da herança, tal como referido em 6.2. A extensão de estruturas de dados não é possível pois estas são definidas pelos componentes, podendo estes ser desenvolvidos em linguagens distintas e com filosofias diversas [Ude94], tornando o processo de integração muito complexo para a aproximação desenvolvida nesta dissertação.

7.4 Incorporação de componentes na aplicação

A fase final do processo de reutilização consiste na integração dos diversos componentes, entretanto adaptados, na aplicação final e no estabelecimento das interligações de

uma forma coordenada. Para um componente poder ser utilizado no âmbito da aplicação deverá ser carregado, estática ou dinamicamente, e efectuadas as ligações das funções de interface utilizadas pelo componente e o registo das operações disponibilizadas por este. Seguidamente, com base nos mecanismos de resolução de nomes o utilizador deve desenvolver as rotinas que permite interligar os diversos componentes numa aplicação coerente e que realize os objectivos inicialmente especificados.

7.4.1 Carregamento e ligação

A operação de carregamento quer de ficheiros **MAP** quer de componentes é efectuada pela operação `/sys/load`. A distinção é obtida através da análise da extensão do ficheiro a carregar: a extensão `.map` corresponde a ficheiros de texto descritos segundo a linguagem **MAP**, sendo todas as restantes extensões consideradas ficheiros binários contendo código objecto para não depender de imposições e limitações de arquitecturas de máquinas, sistemas operativos ou software específico de carregamento.

O caso mais simples de carregamento de componentes corresponde ao carregamento estático, onde os componentes se encontram já agrupados na aplicação. O facto de os componentes estarem agregados na mesma aplicação só implica a existência de um só ficheiro se os componentes fizerem já parte do ficheiro executável. No caso de executáveis com edição de ligações dinâmicas, será responsabilidade do sistema operativo construir em memória uma imagem da aplicação contendo os componentes antes da invocação da função inicial, geralmente **main**. Em ambos os casos, como os componentes estão imediatamente acessíveis na iniciação da aplicação o carregamento é considerado estático, do ponto de vista da integração de módulos da ferramenta **MAP**. Devido à impossibilidade de existir mais de um símbolo definido globalmente, as rotinas de registo de operações devem possuir o nome do ficheiro onde residem, que é igualmente único numa mesma directoria. Os componentes a carregar estaticamente devem ter os endereços de cada uma das suas funções de registo colocadas sequencialmente por ordem de iniciação numa lista de ponteiros para a função a passar à rotina de interface `mapMain` no momento da iniciação da ferramenta **MAP**. A criação desta lista é realizada automaticamente, na presente versão, pela ferramenta de gestão de configurações

make. Assim, a partir de uma lista de componentes indicados no ficheiro **Makefile** gera-se a referida lista de ponteiros e a própria função `main` para produzir um ficheiro executável – **map** – da ferramenta com os componentes carregados estaticamente.

O carregamento dinâmico de componentes pressupõe que a aplicação já se encontra em execução quando um componente é a ela ligado. Na presente realização, apenas é possível efectuar o carregamento dinâmico através da biblioteca de programação da interface com o carregador dinâmico ("*Programming interface to dynamic linking loader*") existente na maioria das versões do sistema operativo `UNIX` e de alguns outros sistemas operativos. Esta interface, vulgarmente conhecida por `dlopen`, permite efectuar a colocação em memória do componente e resolver os seus símbolos indefinidos face aos existentes na aplicação em execução de uma forma automática. Assim, quer o carregamento propriamente dito quer a resolução dos endereços das funções de interface da ferramenta **MAP** são executados de uma forma transparente por estas rotinas de biblioteca. A interface `dlopen` ainda invoca a rotina **_init** do componente carregado antes de retornar. Esta rotina poderia ser utilizada para efectuar o registo das operações disponibilizadas à aplicação. Contudo, optou-se por manter o registo das operações numa rotina com o nome do ficheiro para compatibilidade com o modo de carregamento estático, evitando-se a compilação distinta do ficheiro fonte consoante o carregamento seja estático ou dinâmico. A produção de componentes para carregamento dinâmico pode ser efectuada independentemente da linguagem de desenvolvimento dos mesmos ou das rotinas de interface.

Exemplo 7.14 *Por exemplo, em* `Linux`*, o componente que realiza as matrizes da linguagem* **MAP** *pode ser produzido na versão dinâmica a partir da versão estática por*

```
ld -E -Bsymbolic -shared array.o -o array.so ./map.so -lm -lc
```

Em alguns sistemas operativos, como o `Solaris` ou `Linux`, a própria ferramenta **MAP** pode ser gerada como um componente, ficando com o nome **map.so**. Desta forma a própria pode ser dinamicamente inserida em outras aplicações e que, por sua vez, carrega os outros módulos da forma habitual.

7.4.2 Resolução de nomes

Tal como foi exposto em 7.4.1, cada componente deverá possuir uma função de registo com um nome único para possibilitar o seu carregamento estático. Assim, este nome único surge como o melhor candidato para controlar a identidade dos componentes existentes na aplicação, tal como exposto em 2.3.2.

Uma vez identificado o componente, através da sua função de registo de operações, torna-se possível efectuar o registo dessas mesmas operações de uma forma controlada. O processo de registo de nomes, realizado pela função de interface `mapInit`, começa por criar uma tarefa constituída por uma pilha de dados própria e uma árvore de dados comum com as restantes tarefas. Cada tarefa é, ela própria, registada no caminho `/task` da árvore de dados, começando com a tarefa `/task/1`. Deverá ser debaixo do caminho que identifica cada tarefa que esta última deverá registar dados que não sejam comuns às restantes tarefas. O processo de registo de nomes prossegue com o registo das operações do interpretador da linguagem e, seguidamente, invoca sequencialmente as funções de registo de cada componente constantes da lista de ponteiros para a função descrita em 7.4.1.

Desta forma, cada componente pode redefinir qualquer das operações dos componentes carregados antes dele ou mesmo das operações do interpretador da linguagem. Caso se pretenda vir a repor a operação original, o componente deverá, previamente, registar a operação a redefinir num local conhecido. Assim, o componente pode acrescentar sinónimos para operações já existentes ou alterar a sua visibilidade. Notar que as mesmas acções podem ser realizadas pelo utilizador após o processo de registo ou dinamicamente por qualquer operação que seja invocada. Neste ponto, o sistema proposto apresenta a mesma falta de protecção do ambiente `Smalltalk`, mas como se trata de uma só aplicação as consequências da manipulação indevida por um componente estão mais limitadas.

7.4.3 Coordenação entre componentes

A fase final do processo de integração e, de uma forma mais geral, do processo de reutilização que constitui a motivação desta dissertação é a coordenação entre os diversos componentes que constituem a aplicação. O sistema proposto não estabelece nem promove nenhuma política de coordenação entre os componentes. Ao contrário de outras aproximações apresentadas em 2.3.3 não existem obrigações a determinar as composições possíveis ou um conjunto fixo de vistas a considerar. A livre organização hierárquica dos dados permite que sejam definidas as vistas consideradas necessárias, embora esteja implícito, tal como descrito em 6.2, que os nomes mais próximos da raiz sejam mais globais que os restantes. Para mais, existe a possibilidade de manipular informação anónima nas pilhas de dados, em especial na pilha de argumentos.

Como toda a interacção entre componentes é realizada através do núcleo da ferramenta esta oferece um conjunto de mecanismos, dos quais se destaca a linguagem **MAP**. Esta linguagem permite, quer aos componentes, quer ao programador que constrói a aplicação, quer como forma de extensão aos próprios utilizadores da aplicação a definição e alteração das interacções dos elementos que compõem a aplicação desenvolvida. A extensão de um exemplo completo sai fora da dimensão desta dissertação, mas os exemplos de aplicação apresentados ao longo da mesma, e em particular neste capítulo, oferecem uma primeira aproximação à forma de coordenar as funcionalidades oferecidas pelos diversos componentes.

7.5 Síntese

Neste capítulo procurou-se descrever as ferramentas que realizam o processo de reutilização proposto, nomeadamente nas perspectivas de classificação, selecção, especialização e integração. A classificação de componentes em árvores de classificação, ou **CCT**s, é realizada pelas ferramentas `uml2tree` e `cpp2tree` a partir das linguagens **UML** e **C++** respectivamente. O catálogo de componentes disponíveis é constituído por estas árvores de classificação que são utilizadas no processo de selecção para escolher o componente que melhor se adapta aos requisitos estabelecidos para o desenvolvimento

da aplicação a construir. A selecção necessita de definir um dicionário de sinónimos ajustado ao domínio da aplicação a desenvolver, de estabelecer quais os nomes que melhor descrevem o componente a recuperar e os seus pesos relativos, e determinar a granularidade da informação disponível nas **CCT**s a utilizar na selecção. O cálculo das distâncias, com base nos elementos de selecção definidos, permite a ordenação dos componentes existentes no catálogo e a escolha dos mais aptos a serem recuperados para o processo de especialização. O processo de especialização prepara o componente para ser integrado na aplicação final, nomeadamente através da construção de rotinas de interface e da definição de novos tipos de dados de acordo com o modelo da linguagem **MAP**. A construção da aplicação final usa a ferramenta **MAP** que realiza a linguagem homónima e incorpora os diversos componentes de uma forma coordenada.

Capítulo 8

Conclusões

Reutilização consiste apenas na capacidade de um componente ser utilizado mais de uma vez. Na área do desenvolvimento de software existe um conjunto de componentes que se apresentam como bons candidatos a reutilizar. Estes componentes incluem especificações, desenhos e módulos de código. No entanto, apesar das elevadas somas que são gastas em desenvolvimento de código redundante, a reutilização de software é ainda uma actividade rara. Nos casos em que existe alguma reutilização de software, esta é efectuada com poucas bases técnicas e, frequentemente, auxiliada pela intuição e sensibilidade de alguns membros da equipa de desenvolvimento. A institucionalização da reutilização de software necessita do domínio de um conjunto de problemas técnicos, humanos e organizacionais.

A reutilização apresenta-se atractiva pois os objectivos são atingidos apenas com um reduzido esforço de adaptação. Além disso, se os componentes utilizados forem de alta qualidade, a qualidade da aplicação final será também reflexo dessa qualidade. Para tal, é necessário dispor de um conjunto de componentes que obedeçam aos requisitos do problema a resolver. Contudo, é difícil obter bons níveis de reutilização se os componentes não forem desenhados com o objectivo de cobrir um espectro de aplicação mais alargado que a situação concreta para que estão a ser desenvolvidos. Assim, é necessário o desenvolvimento de bibliotecas constituídas por componentes de qualidade e que cubram as áreas de aplicação.

Presentemente, já são disponibilizados comercialmente significativas quantidades de componentes de software que cobrem um vasto conjunto de áreas de aplicação. No entanto, tal disponibilidade não se reflecte numa reutilização frequente desses componentes. Para mais, muitas das vezes tais componentes são escolhidos em detrimento de outros que apresentavam melhores características funcionais, de desempenho ou de integração no ambiente em questão [FF96]. Daí a necessidade de dispor de um ambiente que permita auxiliar o projectista na escolha dos componentes apropriados, na sua adaptação ao caso concreto e na sua integração na aplicação final.

8.1 Um sistema integrado

O sistema de integração proposto nesta dissertação cobre as actividades que vão desde a classificação de um componente até à sua integração na aplicação final. O processo de classificação introduz o componente no sistema, catalogando-o com base em informação considerada relevante para o processo de selecção. O processo de selecção procura obter o componente que melhor se adapta às características dos requisitos fornecidos em cada caso, por forma a minimizar o esforço de adaptação. A adaptação do componente à situação concreta é efectuada pelo processo de especialização que produz um componente pronto a ser integrado na aplicação. Finalmente, o processo de integração reune todos os componentes resultantes do processo de especialização e junta-os, de uma forma articulada, na aplicação final.

Para desenvolver as referidas actividades, o ambiente proposto nesta dissertação recorre a uma estrutura hierárquica de identificadores, extraídos dos componentes de uma forma automática. A escolha de identificadores deve-se ao elevado conteúdo semântico dos nomes que os designam. Os nomes empregues indicam conceitos utilizados no desenvolvimento do componente. Os identificadores efectuam a ligação entre esses conceitos e as entidades do componente a que o identificador tem acesso. Por outro lado, os identificadores são fáceis de distinguir e catalogar de uma forma automática. Além disso, se os componentes forem descritos por caixas brancas, grande parte do seu conteúdo será igualmente utilizado na classificação, o que permite uma selecção mais rigorosa.

A utilização de hierarquias de identificadores permite registar de uma forma organizada a informação extraída dos componentes. A hierarquia dispõe os identificadores em níveis de pormenor crescente, agrupando identificadores relacionados em conjuntos distintos. A posição de dado nome, que designa o identificador, fornece informação estrutural quanto ao seu contexto. Este contexto é descrito não só pelo caminho que o liga à raiz da hierarquia como pelos nomes que o circundam. Uma estrutura regular deste tipo simplifica o desenvolvimento de ferramentas para a manipular. A inspecção manual, pelos utilizadores, fica também facilitada, pois desenvolve-se segundo apenas duas dimensões, largura e profundidade.

Uma estrutura hierárquica de identificadores permite reter apenas a organização dos nomes, suprimindo a restante informação. A construção da hierarquia que representa o componente é efectuada reproduzindo as relações de agregação e decomposição existentes na linguagem utilizada na descrição do componente. A descrição do componente obtida é rica, compacta e independente dos pormenores lexicais e sintácticos da linguagem utilizada para o desenvolvimento do componente. A hierarquia pode ser complementada com informação introduzida manualmente ou com recurso a outras ferramentas de classificação. Esta informação pode conter valores quantitativos ou qualitativos de desempenho, complexidade, ou outras métricas consideradas relevantes.

O ambiente proposto nesta dissertação não utiliza, ao contrário de outros sistemas, um tratamento de sinónimos no processo de classificação. A informação classificada não é estereotipada nem adulterada, transferindo esse tratamento para o processo de selecção onde pode ser ajustado caso a caso. A importância de cada uma das igualdades entre nomes que contribui para a selecção de um dado componente é afectada de um factor de ponderação que determina a relevância do nome. O conceito de igualdade entre dois nomes, que constitui a base do processo de selecção, pode ser calibrado por ponderação e através da determinação dos pesos de semelhança para cada um dos sinónimos.

A determinação de igualdades constitui o critério de extracção utilizado pelas operações de selecção para construir hierarquias de menores dimensões. A combinação de operações de granulação, particularização, localização, qualificação, união e diferenciação extraem apenas a informação relevante para determinada selecção. Estas hierarquias, normalizadas pelo processo de selecção, podem depois ser comparadas para determinar

mais rigorosamente qual o componente que melhor se adapta à resolução do problema. A escolha final deverá ter em conta as capacidades técnicas da equipa de reutilização e as possibilidades de especialização do componente.

A linguagem de integração utiliza quatro tipos de entidades – identificador, referência, tipo e sinónimo – que estabelecem as relações possíveis entre um espaço de nomes e um espaço de dados. A representação dos tipos de dados como caminhos na hierarquia de nomes facilita a descrição e conversão entre os tipos de dados definidos por cada componente. Tal representação permite à linguagem oferecer mecanismos de sobreposição de operadores e a utilização de técnicas de polimorfismo paramétrico e de inclusão.

A representação dos dados partilháveis numa hierarquia de identificadores, permite a qualquer componente obter a informação que necessita e disponibilizar toda a informação que possa ser relevante para outros. Aplicações, onde os componentes recorram a protocolos e interfaces distintas para a sua comunicação, comprometem a sua evolução e a sua manutenção. O ambiente proposto recorre à passagem de argumentos numa pilha de dados onde os componentes recolhem e depositam a informação a trocar nas suas interacções.

A linguagem oferece igualmente mecanismos de modificação incremental necessários às técnicas de especialização. Estes mecanismos incluem a resolução dinâmica de nomes, o controlo da visibilidade dos nomes e o reencaminhamento de nomes através de sinónimos. O carregamento dinâmico de componentes oferece possibilidades adicionais de controlo e configuração dinâmica da aplicação.

O ambiente proposto nesta dissertação é constituído por uma estrutura hierárquica de nomes que pode ser directamente inspeccionada e alterada. Com base nessa estrutura é identificada uma linguagem que permite construir um conjunto de ferramentas para auxiliar na selecção dos componentes a reutilizar. A linguagem é igualmente utilizada para efectuar adaptações simples e controlar a integração dos componentes na aplicação final. Finalmente, pode-se tirar vantagem da linguagem para estabelecer métodos de reutilização para classes específicas de problemas.

8.2 Evolução do trabalho

O espaço ocupado pela informação de classificação é grande quando comparado com alguns dos outros métodos de classificação. Embora na presença de um número pequeno a médio de componentes este espaço não seja relevante, a classificação de grande número de componentes pode gerar muita informação. Na prática, cada projecto utiliza um número muito reduzido de componentes face aos disponíveis nas bibliotecas. Mesmo a primeira escolha recupera uma pequena percentagem do total de componentes inspeccionados.

O sistema proposto nesta dissertação mostrou ser especialmente eficaz na diferenciação de pormenores, indispensável na fase final do processo de selecção. Assim, o método proposto pode ser aplicado, caso a caso, nos componentes seleccionados na primeira escolha através de um método mais simples de classificação. Nesta hipótese, em vez de manter descrições completas e rigorosas, que se desactualizam com facilidade, recorre-se, por exemplo, ao método de facetas que descreve o componente de uma forma muito geral e improvável de se desactualizar.

Desta forma, os componentes recuperados pela primeira escolha são sujeitos a um processo que corresponde à ligação dos processos de classificação e selecção num só. A construção da hierarquia normalizada é feita não a partir de uma classificação prévia, mas a partir da informação disponível do próprio componente. A desvantagem mais evidente deste processo consiste na impossibilidade de efectuar anotações à classificação, tal como é possível no ambiente proposto.

Do ponto de vista da linguagem é óbvia a necessidade de uma interface com o utilizador mais convencional, em especial do ponto de vista sintáctico. A introdução de mecanismos de *hashing* e *caching* dos nomes mais frequentemente acedidos, que incluem pelo menos os tipos de dados, deverá melhorar significativamente o desempenho da linguagem. A realização da linguagem ficará certamente muito mais complexa por forma a garantir o correcto funcionamento destes mecanismos. O suporte, por parte da linguagem, de mecanismos de herança, quer com base em classes, quer com base em protótipos, poderá contribuir para uma especialização mais controlada e organizada.

8.3 Considerações finais

A resolução de problemas apresenta muitas das características encontradas nos metais preciosos. Esta deve ser usada com sensatez e parcimónia, substituída sempre que possível por recursos mais económicos e, recuperado para uso futuro sempre que possível. Dados os elevados custos técnicos e humanos envolvidos no desenvolvimento de software, a reutilização de partes desse software é essencial. O sucesso da reutilização de componentes de software depende de factores técnicos, humanos e organizacionais.

A existência de muitos componentes de software facilmente acessíveis torna-os, no futuro próximo, bons candidatos a integrar no desenvolvimento de aplicações, com subsequente aumento da produtividade do processo de desenvolvimento. Embora a clara abundância de componentes como caixas pretas os torne como bons candidatos a reutilização, a sua utilização real representa apenas uma pequena parte do total de componentes reutilizados. Nas aplicações desenvolvidas com recurso à reutilização de componentes verifica-se que menos de 15% do código provém de componentes com descrições baseadas em caixas pretas. O custo de adaptação destes componentes está estimado em cerca de 20% do custo de desenvolvimento, o que se traduz num ganho de cerca de 80%. Por outro lado, cerca de 60% a 65% do código, das mesmas aplicações, é constituído por componentes com descrições baseadas em caixas brancas. Estes componentes exigem maior esforço na compreensão das suas descrições, por serem mais completas, mas apresentam custos de adaptação iguais ou superiores aos das caixas pretas [JM98].

A utilização de caixas brancas apresenta, contudo, um potencial de adaptação maior, face às caixas pretas. Embora exista um custo adicional de compreensão das descrições, este custo pode ser amortizado através de uma escolha mais precisa do componente a adaptar e da determinação mais rigorosa das adaptações a efectuar. Como mais de metade do código das aplicações construídas com base em técnicas de reutilização tem origem em caixas brancas, o potencial aumento de produtividade é significativo. Daí que, disponibilizar ambientes de classificação e selecção com base em descrições que incluem a estrutura interna, como a descrita nesta dissertação, possibilitem um potencial aumento da reutilização de componentes.

O conceito de boa reutilização não está na reutilização dos componentes em si, mas na reutilização das soluções para resolver problemas pelos projectistas. A decisão de construir, comprar ou reutilizar software compete ao gestor da actividade de desenvolvimento. No entanto, também lhe cabe, frequentemente a tarefa de encorajar, pedir, implorar ou mesmo obrigar os membros da sua equipa técnica e reutilizar componentes disponíveis. Por esta razão, os melhores analistas e programadores não são apenas um pouco melhores que a maioria dos programadores, eles são substancialmente melhores. Daí que seja essencial o ensino e sensibilização para os problemas e soluções envolvidos na reutilização de software. Além disso, projectistas não treinados na reutilização de software tendem a ser influenciados por critérios menos significativos em detrimento de outros mais relevantes [WES87]. A utilização de processos e ferramentas que permitam guiar cientificamente os analistas e programadores, consegue que estes obtenham bons resultados com uma preparação e experiência inferior.

Os sucessos de reutilização em áreas de engenharia, como a produção automóvel, encorajaram a reutilização ao nível do software e, nomeadamente, dos componentes com base nos quais as aplicações são construídas. De facto, muitas instituições afirmam conseguir níveis de reutilização apreciáveis, que sugerem um elevado domínio de certas áreas. Por exemplo, a NASA apresenta taxas de reutilização de software de 70% a 95% em determinada área de aplicação.

No entanto, diversos insucessos têm vindo a demonstrar que a reutilização de software necessita de um estudo mais cuidado e aprofundado. Alguns fiascos [JM97, Lio96, MN97] evidenciaram alguns dos erros cometidos na reutilização de software. Estes exemplos, largamente divulgados, apresentam apenas a parte visível de um problema mais vasto. Especialmente preocupante é o facto de surgirem em áreas críticas e sujeitas a um elevado controlo de qualidade.

A fábrica de software imaginada por McIlroy em 1969 exige um trabalho de preparação que ainda está longe de estar concluído. O autor tem esperança que o seu modesto trabalho, descrito nesta dissertação, contribua para uma maior visibilidade dos problemas envolvidos na reutilização de componentes de software e o reforço de competências e avanços científicos no sistema universitário.

Apêndice A

Descrição da linguagem MAP

A.1 Descrição sintáctica

A gramática apresentada nesta secção está escrita em **BNF** extendido, na qual as seguintes extensões são utilizadas para tornar a descrição sintáctica mais concisa e legível: $\langle\alpha\rangle*$ representa zero ou mais ocorrências do símbolo $\langle\alpha\rangle$; e $\langle\alpha\rangle+$ representa uma ou mais ocorrências de $\langle\alpha\rangle$. A cadeia de caracteres vazia é representada por ϵ.

⟨**programa**⟩ → (⟨lexema⟩ | ⟨brancos⟩ | ⟨comentário⟩) ∗

⟨**lexema**⟩ → ⟨nome⟩ | ⟨literal⟩ | ⟨caracteres⟩ | ⟨inteiro⟩ | ⟨real⟩ | ⟨rotina⟩

⟨**nome**⟩ → ⟨letra⟩⟨palavra⟩

⟨**literal**⟩ → : ⟨palavra⟩

⟨**caracteres**⟩ → " ⟨elemento⟩∗ " | < ⟨hexdígito⟩+ > | ' ⟨palavra⟩

⟨**inteiro**⟩ → ⟨sinal⟩ ⟨dígito⟩+

⟨**real**⟩ → ⟨sinal⟩ ⟨dígito⟩ ∗ . ⟨dígito⟩ + ⟨sufixo⟩

⟨**rotina**⟩ → { ⟨lexema⟩∗ }

⟨**comentário**⟩ → # (⟨palavra⟩ | '␣' | \t | " | \ ) ∗

⟨**palavra**⟩ → (⟨dígito⟩ | ⟨letra⟩ | ⟨outros⟩) +

⟨**elemento**⟩ → (⟨dígito⟩ | ⟨letra⟩ | ⟨outros⟩ | ⟨brancos⟩) | \ (⟨dígito⟩ | ⟨letra⟩ | ⟨outros⟩ | ⟨brancos⟩ | " | \ )

⟨**sufixo**⟩ → ϵ | ⟨expoente⟩ ⟨sinal⟩ ⟨dígito⟩+

⟨**sinal**⟩ → ϵ | + | −

⟨**expoente**⟩ → *E* | *e*

⟨**brancos**⟩ → ' ' | \n | \t

⟨**hexdígito**⟩ → ⟨dígito⟩ | *A* | *B* | *C* | *D* | *E* | *F* | *a* | *b* | *c* | *d* | *e* | *f*

⟨**dígito**⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨**letra**⟩ → *A* | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z* | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z*

⟨**outros**⟩ → ! | # | $ | % | & | ' |) | (| * | + | , | − | . | / | : | ; | < | = | > | ? | @ |] | [| ^ | _ | ‘ | } | | | { | ~

A.2 Descrição semântica

A semântica da linguagem **MAP** é descrita de forma axiomática, através da listagem das condições verificadas antes e após as operações. Para exprimir a semântica axiomática adopta-se a linguagem de especificação **Z** [Dil94].

Cada um dos cinco tipo de lexemas – `nome`, `literal`, `caracteres`, `inteiro` e `real` – vai dar origem a entidades contendo o valor da sequência identificada e um caminho que descreve o tipo de lexema encontrado. Além desses lexemas existem ainda os tipos compostos `matriz`, `rotina` e `código` que são construídos a partir dos tipos básicos.

Do ponto de vista da descrição da funcionalidade base da linguagem não é relevante qual o tipo de valor associado a cada entidade pois o processamento é efectuado apenas

Tabela A.1: Caminho associado a cada um dos lexemas identificados.

lexema	caminho
nome	/sys/name
literal	/sys/literal
caracteres	/sys/string
inteiro	/sys/integer
real	/sys/real
matriz	/sys/array
rotina	/sys/routine
código	/sys/code

em função do caminho declarado em cada entidade. Contudo, o processamento específico de cada tipo de entidade pressupõe a existência de uma representação específica. Assim, na descrição semântica da funcionalidade base da linguagem faz-se distinção apenas entre caminhos e valores:

$$[\, NOME, VALOR \,]$$

Uma entidade é descrita por um nome que representa o caminho e um dado:

$$entidade == \mathrm{seq}\, NOME \times VALOR$$

Embora a localização de uma entidade seja realizada por um caminho descrito por uma sequência não nula de `nomes`, é igualmente possível obter descrições de caminhos a partir dos valores associados aos tipos `literal` e `caracteres` pois utilizam a mesma representação de dados. Para tal define-se uma função de conversão exterior *caminho* : $VALOR \nrightarrow \mathrm{seq}\, NOME$

Das duas pilhas de dados disponíveis para os componentes compilados apenas a pilha de argumentos está disponível na linguagem. Assim, designaremos a pilha de argumentos por `pilha`. O `dicionário` define as entidades com `nome`, enquanto os `caminhos` definem todas as sequências de nomes que podem ser utilizados para identificar as entidades.

MAP

$$
\begin{array}{l}
pilha : seq\ ((\mathrm{seq}\, NOME \cup \{\bot\}) \times VALOR) \\
dicionário : \mathrm{seq}\, NOME \pfun (\mathrm{seq}\, NOME \times VALOR) \\
caminhos : \mathbb{P}\, \mathrm{seq}\, NOME \\
\hline
\mathrm{dom}\, dicionário \subseteq caminhos
\end{array}
$$

$$\Xi MAP == [\Delta MAP \mid \Theta MAP = \Theta MAP']$$

Ξnomes

$$
\begin{array}{l}
\Delta MAP \\
\hline
dicionário' = dicionário \\
caminhos' = caminhos
\end{array}
$$

init

$$
\begin{array}{l}
\Delta MAP \\
\hline
pilha' = \langle\, \rangle \\
dicionário' = \{\, \} \\
caminhos' = \{\, \}
\end{array}
$$

Existem cinco operações básicas de manipulação da pilha de argumentos, podendo as restantes ser definidas como sequências destas. A operação aqui designada por `push` corresponde à rotina de interface com o mesmo nome, mas na linguagem **MAP** esta operação é executada pelas funções `lexeme` e `read` dependendo do valor ser lido do código ou do terminal.

push

$$
\begin{array}{l}
\Xi nomes \\
lexema? : VALOR \\
caminho? : \mathrm{seq}\, NOME \\
\hline
pilha' = pilha \frown \langle caminho?, lexema? \rangle
\end{array}
$$

drop

$$
\begin{array}{l}
\Xi nomes \\
\hline
\# pilha \geq 1 \\
pilha' = tail\ pilha
\end{array}
$$

dup

$$\Xi nomes$$

$$\#\, pilha \geq 1$$
$$pilha' = pilha \frown \langle head\ pilha \rangle$$

swap

$$\Xi nomes$$

$$\#\, pilha \geq 2$$
$$pilha' = \langle head\ tail\ pilha \rangle \frown \langle head\ pilha \rangle \frown tail\ tail\ pilha$$

size

$$\Xi nomes$$

$$pilha' = \langle \#\, pilha \rangle \frown pilha$$

A operações de manipulação da árvore interagem preferencialmente com a pilha de argumentos e permitem extender as capacidades da linguagem.

$$subcaminhos : \mathrm{seq}\, NOME \times \mathbb{P}\, \mathrm{seq}\, NOME \pfun \mathbb{P}\, \mathrm{seq}\, NOME$$

$$\forall a : \mathrm{seq}\, NOME;\ c : \mathbb{P}\, \mathrm{seq}\, NOME \exists d : \mathbb{P}\, \mathrm{seq}\, NOME \bullet$$
$$subcaminhos(a, c) = d \land (\ \#a = 0 \Rightarrow d = \{\} \land$$
$$(\#a > 0 \land a \in c) \Rightarrow d = subcaminhos(front\ a, c) \land$$
$$(\#a > 0 \land a \notin c) \Rightarrow d = \{a\} \bigcup subcaminhos(front\ a, c)\ )$$

bind

$$\Delta MAP$$

$$\#\, pilha \geq 2$$
$$head\ pilha \in \mathrm{dom}\ dicionário \Rightarrow$$
$$dicionário' = dicionário \oplus \{caminho(head\ pilha) \mapsto head\ tail\ pilha\}$$
$$head\ pilha \notin \mathrm{dom}\ dicionário \Rightarrow$$
$$dicionário' = dicionário \bigcup \{caminho(head\ pilha) \mapsto head\ tail\ pilha\}$$
$$\exists a : \mathrm{seq}\, NOME \bigcup \{\bot\};\ b : VALOR \bullet \langle a, b \rangle = head\ pilha \land$$
$$caminhos' = subcaminhos(a, caminhos) \bigcup caminhos$$
$$pilha' = tail\ tail\ pilha$$

look

$\Xi nomes$

$\#\ pilha \geq 1$
$head\ pilha \in \mathrm{dom}\ dicionário \Rightarrow$
$\quad pilha' = \langle dicionário(head\ pilha)\rangle \frown tail\ pilha$
$head\ pilha \notin \mathrm{dom}\ dicionário \Rightarrow$
$\quad pilha' = \langle \bot \rangle \frown tail\ pilha$

forget

ΔMAP

$\#\ pilha \geq 1$
$head\ pilha \in \mathrm{dom}\ dicionário \Rightarrow$
$\quad dicionário' = \{head\ pilha\} \lhd dicionário$
$caminhos' = caminhos$
$pilha' = pilha$

find

$\Xi nomes$
$encontrou! : \mathbb{B}$

$\#\ pilha \geq 1$
$encontrou! = head\ pilha \in caminhos$
$pilha' = tail\ pilha$

tree

$\Xi nomes$
$nomes! : \mathbb{P}\,\mathrm{seq}\,NOME$

$\#\ pilha \geq 1$
$nomes! = \{c : \mathrm{seq}\,NOME \mid head\ pilha \frown c \in caminhos\}$
$pilha' = tail\ pilha$

interp

ΔMAP

$\#\ pilha \geq 1$
$\forall a : \mathrm{seq}\,NOME \cup \{\bot\};\ x : VALOR \bullet \langle a, x\rangle = head\ pilha \wedge$
$\quad (\ caminho(x) \in \mathrm{dom}\,dicionário \Rightarrow call(dicionário(caminho(x))) \wedge$
$\quad\ (caminho(x) \notin \mathrm{dom}\,dicionário \wedge a \in \mathrm{dom}\,dicionário) \Rightarrow call(dicionário(a)) \wedge$
$\quad\ (caminho(x) \notin \mathrm{dom}\,dicionário \wedge a \notin \mathrm{dom}\,dicionário) \Rightarrow$
$\quad\quad call(dicionário(caminho(/sys/error)))\ )$

As operações de interpretação de cada tipo de dados encontram-se associadas a entida-

des designadas pelos caminhos indicados na tabela A.1. A função *call* : seq $NOME \times VALOR$ recebe como argumento uma entidade, normalmente do tipo `código` ou `rotina`, responsável pelo tratamento da entidade no topo da pilha. Uma vez que a rotina de tratamento de cada tipo pode alterar quer o dicionário quer a pilha arbitrariamente não é possível descrever os respectivos esquemas. No entanto, a maioria dos tipos de dados não possui operação de interpretação, com a excepção do `código`, `rotina` e certos tipos exteriores que venham a ser definidos. A interpretação de rotinas baseia-se na sucessiva colocação das entidades que definem a sequência de instruções e a sua interpretação.

A.3 Descrição da interface

A interface disponível para os diversos componentes actuarem sobre a ferramenta **MAP** inclui os três tipos de dados e as 19 funções que seguidamente se descrevem e que constituem o ficheiro de declarações da ferramenta `map.h`.

O interpretador é referido por um ponteiro para uma estrutura de dados `Map`, definida como uma estrutura opaca para evitar acessos a campos privados do interpretador e limitar dependências de futuras versões.

```
typedef struct map Map;
```

A estrutura de dados `Entity` suporta a maioria dos tipos de dados base da linguagem **C**, utilizada para escrever a ferramenta, e as estruturas mínimas para colocar em funcionamento o interpretador e facilitar a sua depuração. Todas as outras estruturas de dados que seja necessário realizar, tais como árvores, utilizam o ponteiro genérico não tipificado `user`.

```
typedef struct entity {
  char  *path;
  union {
    long   integer;
    short   small[2];
    float   real;
```

```
    char  *text;
    int (*func)(Map *, struct entity *ent, void *user);
    struct entity  *ent;
    struct entity **array;
    Map  *task;
    void  *user;
  } data;
} Entity; /* The object descriptor */
```

A assinatura da funcção de retro-chamada `mapFunc` utilizada na função `mapAll`.

```
typedef long (*mapFunc)(Map *map, char *name, Entity *e, void *user);
```

int mapMain(int argc, char *argv[, int (*funcs[])(Map*))] é a função principal da ferramenta que inicia o sistema `mapInit`, coloca os argumentos `argv[]` na pilha de argumentos sob a forma de `/sys/text` e invoca o `mapEval` do `argv[1]` sob a forma de `/sys/name`, seguidamente chama o `mapLoop` para o processamento normal.

Map *mapInit(char *path, int (*funcs[ )(Map*), char *parser, void *user)] função de iniciação do sistema que regista as operações do interpretador da linguagem e dos componentes ligados estaticamente.

Map *mapTask(Map *parent, char *parser, void *user) cria uma tarefa a partir de outra, apenas a pilha de argumentos é distinta da tarefa original,

int mapLoop(Map *map, char *parser) função que invoca sucessivamente o parser indicado, ou na sua omissão `/sys/interp`, até receber um código de terminação.

void *mapEnd(Map *map) efectua a terminação do sistema libertando a memória reservada pelo núcleo e os recursos utilizados.

int mapError(Map *map, int errno) função de impressão de mensagens de erro com base nos códigos de error devolvidos pelas operações.

void *mapUser(Map *map, char *local, char *parser) função que permite obter os parâmetros do núcleo, ou seja, o identificador da tarefa e do parser em uso, bem como o ponteiro para a área de utilizador.

Entity *mapBind(Map *map, char *path, Entity *ent) função que permite associar uma entidade a um caminho na árvore, quer seja constante, variável, função, tipo de dados, *etc*.

Entity *mapName(Map *map, char *path, char *name) cria um objecto do tipo `name` e associa-o ao caminho `path`; é a função mais utilizada pelos componentes para registar as operações.

int mapErase(Map *map, char *name) retira a entidade associada ao caminho `name` da árvore de nomes; o espaço ocupado pela entidade não é reclamado nem a rotina de tratamento do tipo invocada.

long mapAll(Map *map, char *name, int level, mapFunc func, void *user) função que itera na sub-árvore da árvore de dados cuja raiz é definida pelo caminho `name`, até uma profundidade `level`; para cada nó em que exista uma entidade associada invoca a função `func` com o caminho da entidade encontrada, um ponteiro para a referida entidade e o ponteiro com informação do utilizador `user`.

Entity *mapLook(Map *map, char *name) função que permite obter uma entidade dado o seu caminho; notar que a função devolve o ponteiro nulo, caso não exista uma entidade associada ao nó indicado pelo caminho `name`, mesmo que este exista.

int mapFind(Map *map, char *name) função que indica se existe um indicado pelo caminho `name`.

Entity *mapPush(Map *map, char *name) função de acesso à pilha de argumentos e coloca no seu topo uma nova entidade – reservando espaço apenas para a área comum de caminho e dados – com o tipo `name`, devolvendo um ponteiro para a mesma para posterior preenchimento pelo programador.

Entity *mapFetch(Map *map, int pos) devolve a entidade na posição `pos` da pilha de argumentos a contar do topo – onde `pos = 1` representa o topo da pilha – e devolve um ponteiro para essa entidade; não existe duplicação ou libertação da entidade, nem mesmo a sua remoção da pilha.

int mapCopy(Map *map, Entity *ent) função que copia uma entidade `ent`; para tal cria uma nova entidade e copia a área comum e, caso exista uma função de tratamento de tipo, invoca-a para que esta última proceda à cópia das partes comum e específica.

int mapDrop(Map *map, int pos, int delete) função que elimina `pos` elementos da pilha de argumentos a contar do seu topo – `pos = 1` remove apenas o elemento do topo – chamando a respectiva rotina de tratamento do tipo, caso exista. (O argumento `delete` existe temporariamente até serem eliminadas as fugas de memória e permite impedir a libertação da área comum da entidade.)

int mapEval(Map *map, Entity *ent, int exec) função que determina o valor dos objectos, invocando as respectivas funções de tratamento de tipo, caso existam, ou copiando as entidades que não as possuam.

int mapInterp(Map *map, char *parser) função que invoca o `parser` – ou o parser de omissão caso o argumento seja nulo – e avalia a entidade que ficou no topo da pilha de argumentos. É expectável que em cada invocação o parser processe um lexema do ficheiro de instruções que está a executar e o coloque no topo da pilha. Assim, esta função executa a análise de um lexema (ou *token*).

A.4 Operações para o utilizador

Convém distinguir as funções de interface, disponíveis para os componentes que interagem com o núcleo da ferramenta (em **C**), das operações acessíveis ao utilizador para a composição de programas com base nas facilidades oferecidas pelos componentes (em **MAP**). Para cada uma destas operações é apresentada uma assinatura com base na notação de efeito na pilha de dados ("*stack effect*") muito utilizada em linguagens postfixadas, nomeadamente na variante tipificada apresentada em [Kna93].

Com o objectivo de tornar a assinatura parecida com a utilização prática, cada operação é identificada por uma notação da forma:
antes `operação` *depois*

onde *antes* representa as entidades que devem estar presentes na pilha de dados no momento da invocação da `operação` que, por sua vez, produz na pilha de dados a sequência *depois*. Cada elemento, quer da sequência *antes* quer da sequência *depois*, é identificado por um nome e um tipo, por exemplo:

número:/sys/integer

onde a parte referente ao tipo pode ser omitida se a operação não se restringir a um tipo específico. Referir, ainda, que a sequência é apresentada por ordem ascendente na pilha, por exemplo:

fundo:/sys/text teceiro:/sys/logical segundo:/sys/real primeiro:/sys/integer

onde o último elemento da sequência *primeiro:/sys/integer* representa o topo da pilha.

Uma formulação mais rigorosa desta notação, em **BNF** extendido, com base na gramática apresentada em A.1 permite expor com maior rigor a notação de efeito na pilha de dados atrás apresentada.

⟨**assinatura**⟩ → ⟨sequência⟩ ⟨operação⟩ ⟨sequência⟩

⟨**sequência**⟩ → − − − | (⟨parcela⟩) +

⟨**parcela**⟩ → ⟨nome⟩ | ⟨nome⟩ : ⟨tipo⟩

⟨**tipo**⟩ → / ⟨nome⟩

⟨**operação**⟩ → / ⟨nome⟩

Seguidamente apresentam-se algumas das operações disponíveis na presente versão da linguagem (0r12), assumindo que todos os componentes se encontram ligados, estática ou dinamicamente. Mesmo que nenhum dos componentes se encontre ligado, com a excepção do interpretador da linguagem que está sempre presente, as funções de interface do núcleo da ferramenta apresentadas em A.3 estão sempre presentes para permitir a integração de novos componentes e a definição das suas funcionalidades.

interpretador da linguagem (/sys/interp)

```
                                        ---   /sys/interp/quit   ---
                                        ---  /sys/interp/debug  ---
                                filename:/sys/text   /sys/interp/load   ---
                                        ---   /sys/interp/tree   ---
                                        ---    /sys/interp/lsr   ---
        pathname:/sys/text      level:/sys/integer    /sys/interp/dir   ---
                                pathname:/sys/text    /sys/interp/cd    ---
                                        ---    /sys/interp/ls    ---
```

pilha de dados (/sys/interp)

```
                entity    /sys/interp/dup    entity    entity
        entity1 entity2   /sys/interp/over   entity1   entity2   entity1
        entity1 entity2   /sys/interp/swap   entity2   entity1
                entity    /sys/interp/drop   ---
                  ---   /sys/interp/count   ---
```

rotinas (/sys/rotine)

```
                                                  ---     /sys/interp/{     ---
                                condition:/sys/logical    entity     /sys/interp/if    ---
        condition:/sys/logical                  entity1   entity2   /sys/interp/ifelse  ---
           initial:/sys/integer      final:/sys/integer     entity    /sys/interp/loop   ---
                                                  entity    /sys/rotine/eval   ---
```

nomes (/sys/literal)

```
                entity   name:/sys/literal    /sys/literal/bind   ---
                         name:/sys/literal   /sys/literal/forget  ---
                         name:/sys/literal    /sys/literal/text   ---
        path:/sys/literal name:/sys/literal   /sys/literal/morph  ---
```

valores lógicos (/sys/logical)

```
        logical1:/sys/logical   logical2:/sys/logical     /sys/logical/and     logical:/sys/logical
        logical1:/sys/logical   logical2:/sys/logical      /sys/logical/or     logical:/sys/logical
                                logical1:/sys/logical     /sys/logical/not     logical:/sys/logical
                                 logical:/sys/logical  /sys/logical/integer  value:/sys/integer
                                 logical:/sys/logical    /sys/logical/text    description:/sys/text
                                 logical:/sys/logical    /sys/logical/print   ---
```

matrizes (/sys/array)

	---	/sys/array/[	mark:/sys/mark
mark:/sys/mark	entities	/sys/array/]	array:/sys/array
index:/sys/integer	array:/sys/array	/sys/array/nth	entity
	array:/sys/array	/sys/array/count	count:/sys/integer
	array:/sys/array	/sys/array/first	entity
	array:/sys/array	/sys/array/last	entity
array1:/sys/array	array2:/sys/array	/sys/array/append	array:/sys/array
	array:/sys/array	/sys/array/reverse	array2:/sys/array
	array:/sys/array	/sys/array/split	entities
	array:/sys/array	/sys/array/print	---

árvores de nomes (/sys/tree)

	filename:/sys/text	/sys/tree/open	tree:/sys/tree
	tree:/sys/tree	/sys/tree/close	---
filename:/sys/text	/sys/tree	/sys/tree/load	---
filename:/sys/text	/sys/tree	/sys/tree/save	---
name:/sys/text	tree:/sys/tree	/sys/tree/get	value:/sys/integer
name:/sys/text	tree:/sys/tree	/sys/tree/del	---
name:/sys/text	tree:/sys/tree	/sys/tree/find	pathname:/sys/text
	tree:/sys/tree	/sys/tree/print	---

name:/sys/text	value:/sys/integer	tree:/sys/tree	/sys/tree/put	---
iter:/sys/routine	name:/sys/text	level:/sys/integer	tree:/sys/tree	
			/sys/tree/all	count:/sys/integer

cadeias de caracteres (/sys/text)

string1:/sys/text	string2:/sys/text	/sys/text/concat	string:/sys/text
	string:/sys/text	/sys/text/real	value:/sys/real
	string:/sys/text	/sys/text/integer	value:/sys/integer
	string:/sys/text	/sys/text/literal	name:/sys/literal
string1:/sys/text	string2:/sys/text	/sys/text/==	value:/sys/logical
string1:/sys/text	string2:/sys/text	/sys/text/!=	value:/sys/logical
string1:/sys/text	string2:/sys/text	/sys/text/>	value:/sys/logical
string1:/sys/text	string2:/sys/text	/sys/text/<	value:/sys/logical
string1:/sys/text	string2:/sys/text	/sys/text/>=	value:/sys/logical
string1:/sys/text	string2:/sys/text	/sys/text/<=	value:/sys/logical
	string:/sys/text	/sys/text/length	length:/sys/integer
	string:/sys/text	/sys/text/depth	count:/sys/integer
	string:/sys/text	/sys/text/print	---

valores inteiros (/sys/integer)

num1:/sys/integer	num2:/sys/integer	`/sys/integer/+`	num:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/-`	num:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/*`	num:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/\`	num:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/%`	num:/sys/integer
	num:/sys/integer	`/sys/integer/abs`	value:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/max`	num:/sys/integer
num1:/sys/integer	num2:/sys/integer	`/sys/integer/min`	num:/sys/integer
	num:/sys/integer	`/sys/integer/sign`	value:/sys/integer
	num:/sys/integer	`/sys/integer/real`	value:/sys/real
	num:/sys/integer	`/sys/integer/string`	description:/sys/text
num1:/sys/integer	num2:/sys/integer	`/sys/integer/==`	value:/sys/logical
num1:/sys/integer	num2:/sys/integer	`/sys/integer/!=`	value:/sys/logical
num1:/sys/integer	num2:/sys/integer	`/sys/integer/>`	value:/sys/logical
num1:/sys/integer	num2:/sys/integer	`/sys/integer/<`	value:/sys/logical
num1:/sys/integer	num2:/sys/integer	`/sys/integer/>=`	value:/sys/logical
num1:/sys/integer	num2:/sys/integer	`/sys/integer/<=`	value:/sys/logical
	num:/sys/integer	`/sys/integer/print`	`---`

valores reais (/sys/real)

as mesmas operações existentes para os valores inteiros, mas onde o caminho `/sys/integer` é substituído por `/sys/real`, eliminando a operação `/sys/integer/%` e substituindo a operação `/sys/integer/real` por

num:/sys/real	`/sys/real/integer`	value:/sys/integer

Além das operações acima enunciadas existem as especificamente desenvolvidas para o processo de selecção, e cujo significado se desenvolve no capítulo 7.

Apêndice B

Representação textual UML

A representação textual descrita neste capítulo refere-se apenas a um sub-conjunto do diagrama de classes da **UML** que contém a informação relevante para a construção das árvores de identificadores. A gramática apresentada está escrita em **BNF** extendido, na qual as seguintes extensões são utilizadas para tornar a descrição sintáctica mais concisa e legível: $\langle\alpha\rangle*$ representa zero ou mais ocorrências do símbolo $\langle\alpha\rangle$; e $\langle\alpha\rangle+$ representa uma ou mais ocorrências de $\langle\alpha\rangle$. A cadeia de caracteres vazia é representada por ϵ.

⟨**diagrama**⟩ → ⟨descrição⟩∗

⟨**descrição**⟩ → *class* ⟨nome⟩⟨declaração⟩∗ | ⟨comentário⟩

⟨**declaração**⟩ → ⟨herança⟩ | ⟨atributo⟩ | ⟨operação⟩ | ⟨associação⟩ | ⟨comentário⟩

⟨**herança**⟩ → : ⟨nome⟩

⟨**atributo**⟩ → ⟨nome⟩ ⟨tipo⟩ ? (= ⟨valor⟩) ?

⟨**operação**⟩ → ⟨nome⟩ $\big($⟨argumentos⟩$\big)$ ⟨tipo⟩ ?

⟨**associação**⟩ → ⟨nome⟩⟨relação⟩ˆ⟨nome⟩⟨relação⟩ (ϵ | *agregation* | *composition*)

⟨**comentário**⟩ → − − (⟨palavra⟩ | ′␣′ | \t | ′ | \ ) ∗

⟨**argumentos**⟩ → (ϵ | ⟨argumento⟩ | , ⟨argumento⟩)

⟨**argumento**⟩ → (⟨nome⟩ | ⟨tipo⟩ | ⟨nome⟩⟨tipo⟩)

⟨**tipo**⟩ → : ⟨nome⟩

⟨**relação**⟩ → ϵ | ⟨número⟩ (. . (⟨número⟩ | $*$)) ?

⟨**nome**⟩ → ⟨letra⟩ (⟨letra⟩ | ⟨dígito⟩) $*$

⟨**valor**⟩ → ⟨caracteres⟩ | ⟨real⟩ | ⟨inteiro⟩ | ⟨número⟩

⟨**caracteres**⟩ → ′ ⟨elemento⟩$*$ ′

⟨**elemento**⟩ → (⟨dígito⟩ | ⟨letra⟩ | ⟨outros⟩ | ⟨brancos⟩) | \ (⟨dígito⟩ | ⟨letra⟩ | ⟨outros⟩ | ⟨brancos⟩ | ′ | \ )

⟨**real**⟩ → ⟨sinal⟩ ⟨dígito⟩ $*$. ⟨dígito⟩ + ⟨sufixo⟩

⟨**sufixo**⟩ → ϵ | ⟨expoente⟩ ⟨sinal⟩ ⟨dígito⟩+

⟨**sinal**⟩ → ϵ | + | −

⟨**expoente**⟩ → E | e

⟨**brancos**⟩ → ′␣′ | \$n$ | \$t$

⟨**inteiro**⟩ → (+ | −) ⟨número⟩

⟨**número**⟩ → (⟨dígito⟩) +

⟨**dígito**⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨**letra**⟩ → A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨**outros**⟩ → ! | # | $ | % | & | ” |) | (| $*$ | + | , | − | . | / | : | ; | < | = | > | ? | @ |] | [| ˆ | _ | ‘ | } | | | { | ˜

Bibliografia

[ABC+91] Patrick Arnold, Stephanie Bodoff, Derek Coleman, Helena Gilchrist, e Fiona Hayes. An evaluation of five object-oriented development methods. Relatório técnico, Information Management Laboratory, Hewlett Packard, HP Laboratories Bristol, Jun. 1991.

[AG98] Ken Arnold e James Gosling, editores. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, 2ª edição, 1998.

[AL95a] Dean Allemang e Beat Liver. Functional representation for reusable components. In *Workshop on Institutionalizing Software Reuse*, 1995.

[AL95b] Dean Allemang e Beat Liver. Trade-off between flexibility and efficiency in recombining reusable components. In *Workshop on Institutionalizing Software Reuse*, 1995.

[ALMD96] Patrick Ateyaert, Carine Lucas, Kim Mens, e Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA*, pp. 268–285, 1996.

[AM97] Alain Abran e Marcela Maya. Measurement of functional reuse. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Ast96] Hernán Astudillo. Reorganizing split objects. In *OOPSLA*, 1996.

[Atk97] Steven Atkinson. Examining behavioural retrieval. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[BAB+87] Bruce A. Burton, Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, e Lauren A. Mayes. The reusable software library. *IEEE Software*, 4(4):25–33, Jul. 1987.

[Bak94] Henry G. Baker. Linear logic and permutation stack – the forth shall be first. *ACM Sigarch Computer News*, 22(1):34–43, Mar. 1994.

[Bak95] Henry G. Baker. 'use-once' variables and linear objects – storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, Jan. 1995.

[Bar77] D. W. Baron. *An Introduction to the Study of Programming Languages*, volume 7 de *Cambridge Computer Science Texts*. Cambridge University Press, The Pitt Building, Trumpington Street, Cambridge CB2 1RP, 1977.

[Bas87] Paul G. Bassett. Frame-based software engineering. *IEEE Software*, 4(4):9–16, Jul. 1987.

[Bas90] Victor R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, Jan. 1990.

[BB91] Bruce H. Barnes e Terry B. Bollinger. Making reuse cost-effective. *IEEE Software*, 8(1):13–24, Jan. 1991.

[BD96] Daniel Bardou e Christophe Dony. Split objects: a disciplined use of delegation within objects. In *OOPSLA*, 1996.

[BD97] Greg Butler e Pierr Denommée. Documenting frameworks. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[BE96] Thomas Ball e Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, Abr. 1996.

[Ber91] Thomas Berlage. *OSF/Motif: Concepts and Programming*. Addison-Wesley, Reading, MA, USA, 1991.

[BGMT88] Gerald Boudier, Ferdinando Gallo, Regis Minot, e Ian M. Thomas. An overview of PCTE and PCTE+. *ACM SIGSOFT/SIGPLAN Software Engineering*

Symposium on Practical Software Development Environments, 13(5):248–257, Nov. 1988.

[BJR96] Grady Booch, Ivar Jacobson, e James Rumbaugh. *The Unified Modeling Language for Object-Oriented Development*. Rational Software Coporation, Set. 1996.

[BJR97a] Grady Booch, Ivar Jacobson, e James Rumbaugh. *Unified Modeling Language Notation Guide*. Rational Software Coporation, Jan. 1997.

[BJR97b] Grady Booch, Ivar Jacobson, e James Rumbaugh. *Unified Modeling Language Sematics*. Rational Software Coporation, Jan. 1997.

[BJR99] Grady Booch, Ivar Jacobson, e James Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, USA, 1999.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 2ª edição, 1994.

[Bos97] Jan Bosch. Adapting object-oriented components. In *ECOOP, 2nd International Workshop on Component-Oriented Programming*, pp. 13–21, 1997.

[Bou91] David Boundy. A taxonomy of programmers. *Software Engineering Notes*, 16(4):23–30, Out. 1991.

[BR87] T. J. Biggerstaff e C. Richter. Reusability framework, assessment and directions. *IEEE Software*, 4(2):41–49, Mar. 1987.

[Bro81] Leo Brodie. *Starting FORTH*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1981.

[Bro94] Leo Brodie. *Thinking FORTH*. Forth Interest Group, Inc., Oakland, California, 2ª edição, 1994.

[BT95] Robert Biddle e Ewan Tempero. Understanding OOP language support for reusability. In *Workshop on Institutionalizing Software Reuse*, 1995.

[BZ91] Paul Butcher e Hussein Zedan. Lucinda - an overview. *ACM SIGPLAN Notices*, 26(8):90–100, Ago. 1991.

[Car95] Luca Cardelli. A language with distributed scope. In *Conf. on Principles of Programming Languages*, pp. 286–297, 1995.

[CB91] Gianluigi Caldiera e Victor R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, Fev. 1991.

[CBG+96] Lorette Cameron, Charles Berman, Sanjiv Gossain, BrianHenderson-Sellers, Laura Hill, e Randall Smith. Perspectives on reuse. In *OOPSLA*, pp. 101–103, 1996.

[CC97] Yonghao Chen e Betty H. C. Cheng. Formally specifying and analysing architectural and functional properties of components for reuse. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[CM98] Siobhan Clarke e John Murphy. Verifying components under development at the design stage: A tool to support the composition of component design models. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[Cop92] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, USA, 1992.

[Cox90] B. J. Cox. Planning the software revolution. *IEEE Software*, 7(6):25–35, Nov. 1990.

[Cre98] Rui G. Crespo. Matching single-sort algebraic specifications for software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 8(3), Set. 1998.

[Cus89] M. A. Cusumano. The software factory: A historical interpretation. *IEEE Software*, 5(2):23–30, Mar. 1989.

[Dav93] T. Davis. The reuse capability model: A basis for improving an organization's reuse capability. In *International Workshop on Software Reusability Advances in Software Reuse*, pp. 123–133, Lucca, Italy, Mar. 1993.

[Dew79] M Dewey. *Decimal Classification and Relative Index*. Forest Press Inc., Albany, N.Y., 1979.

[DFF97] Ernesto Damiani, Maria Grazia Fugini, e Enrico Fusaschi. A description-based approach to OO code reuse. *IEEE Computer*, 30(10):73–80, Out. 1997.

[Dil94] Antoni Diller. *Z: An introduction to formal methods*. John Wiley and Sons, New York, NY, USA, 1994.

[DLCP96] Allen H. Dutoit, Sean Levy, Douglas Cunningham, e Robert Patrik. The basic object system: Supporting a spectrum from prototypes to hardened code. In *OOPSLA*, pp. 104–121, 1996.

[dS97] Pedro Reis dos Santos. Identifier based representation and management of software components. In *ECOOP'97 workshop on Modeling Software Processes and Artifacts*, pp. 33–36, Jyvaskyla, Finland, Jun. 1997.

[dSC98a] Pedro Reis dos Santos e Rui Gustavo Crespo. Assisted selection of components using classified identifiers. In *7th Conference on Information Processing and Management of Uncertainty in Knowledged-based Systems*, pp. 740–747, Paris, France, Jul. 1998.

[dSC98b] Pedro Reis dos Santos e Rui Gustavo Crespo. Classificação e selecção de componentes em projectos de software. In *3º Encontro Nacional para a Qualidade nas Tecnologias da Informação e Comunicações*, Guimarães, Portugal, Nov. 1998.

[ED97] Letha H. Etzkorn e Carl G. Davis. Automatically identifying reusable OO legacy code. *IEEE Computer*, 30(10):66–71, Out. 1997.

[Edw95] Stephen H. Edwards. Good mental models are necessary for understandable software. In *Workshop on Institutionalizing Software Reuse*, 1995.

[EGWZ97] Stephen H. Edwards, David S. Gibson, Bruce W. Weide, e Sergey Zhupanov. Software component relationships. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[End96] Albert Endres. A synopsis of software engineering history: The industrial perspective. In *History of Software Engineering*, Ago. 1996.

[Faf94] Danielle Fafchamps. Organizational factors and reuse. *IEEE Software*, 11(5):31–41, Set. 1994.

[FF95] William B. Frakes e Christopher J. Fox. Sixteen questions about software reuse. *Communications of the ACM*, 38(6):75–87, Jun. 1995.

[FF96] William B. Frakes e Christopher J. Fox. Quality improvement using a software reuse failure modes model. *IEEE Transactions on Software Engineering*, 22(4):274–279, Abr. 1996.

[FI94] William B. Frakes e Sadahiro Isolda. Success factors of systematic reuse. *IEEE Software*, 11(5):15–19, Set. 1994.

[Fis87] Gerhard Fischer. Cognitive view of reuse and redesign. *IEEE Computer*, 4(4):60–72, Jul. 1987.

[FK92] Robert G. Fichman e Chris F Kemerer. Object-oriented and convetional analysis and design methodologies. *IEEE Computer*, 25(10):22–39, Out. 1992.

[FP94] William B. Frakes e Thomas P. Pole. An empirical study of representation models for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, Ago. 1994.

[FS98] Martin Fowler e Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, USA, 1998.

[FT96] William Frakes e Carol Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, Jun. 1996.

[GC92] David Gelernter e Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Fev. 1992.

[Gog86] Joseph A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, Fev. 1986.

[GR83] Adele Goldberg e David Robson. *SmallTalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, USA, 1983.

[GR89] Adele Goldberg e David Robson. *Smalltalk-80: the language and its implememtation*. Addison-Wesley, Reading, MA, USA, 1989.

[Gra93] Robert B. Grady. Practical results from measuring software quality. *Communications of the ACM*, 36(11):62–68, Nov. 1993.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(9):11–19, Set. 1990.

[Ham97] Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, 1997.

[Han98] Jun Han. Characterization of components. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[Har87] William Harrison. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4(11):46–56, Nov. 1987.

[HLS97] Koen De Hondt, Carine Lucas, e Patrick Steyaert. Reuse contracts as component interface descriptions. In *ECOOP, 2nd International Workshop on Component-Oriented Programming*, pp. 43–49, 1997.

[Hog90] Christopher Hoggan. *Essentials of Logic Programming*. Graduate Texts in Computer Science. Oxford University Press, 1990.

[Hol93] U. Holzle. Integrating independentely-developed components in object-oriented languages. In *7th ECOOP*, pp. 36–56, 1993.

[HPLH90] Philip A. Hausler, Mark G. Pleszkosh, Richard C Linger, e Alan R. Hevner. Using function abstraction to understand program behavior. *IEEE Software*, 7(1):55–63, Jan. 1990.

[HS90] C. Frederick Hart e John J. Shiling. An environment for documenting software features. *ACM SIGPLAN Notices*, 25(6):120–132, 1990.

[HU79] John E. Hopcroft e Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA, USA, 1979.

[HW86] Ellis Horowitz e Ronald C. Williamson. Sodos: A software documentation support environment - its definition. *IEEE Transactions on Software Engineering*, 12(8):849–859, Ago. 1986.

[Inc85] Adobe Systems Inc. *PostScript: Tutorial and Cookbook*. Addison-Wesley, Reading, MA, USA, Dez. 1985.

[Jaz95] Mehdi Jazayeri. Component programming - a fresh look at software components. In *European Software Engineering Conf.*, Set. 1995.

[JCJO92] Ivar Jacobson, Magnus Chisterson, Patrik Jonsson, e Gunnar Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, Reading, MA, USA, 1992.

[JM97] Jean-Marc Jézéquel e Bertrand Mayer. Design by contract: The lessons of ariane. *IEEE Computer*, 30(1):129–130, Jan. 1997.

[JM98] Lamia Labed Jilani e Ali Mili. Correlating adaptation effort with functional distance. In *7th IPMU*, pp. 733–739, Paris, France, Jul. 1998.

[JMM97] Lamia Labed Jilani, Rym Mili, e Ali Mili. Approximate component retrieval: An academic exercise or a practical concern? In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Jr.95] Gordon S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, Digital Equipment Corporation 1995.

[Jr.97] Gordon S. Novak Jr. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering*, 23(7):401–417, Jul. 1997.

[Jus98] José Luís Barros Justo. Técnicas para la classificación/recuperación de componentes software reutilizables y su impacto en la calidad. In *3º Encontro Nacional para a Qualidade nas Tecnologias da Informação e Comunicações*, Guimarães, Portugal, Nov. 1998.

[KA90] Setrag Khoshafian e Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley and Sons, New York, NY, USA, 1990.

[Kam90] Samuel N. Kamin. *Programming Languages: An Interpreter Based Approach*. Addison-Wesley, Reading, MA, USA, 1990.

[KBM97] Oh Cheon Kwon, Cornelia Boldyreff, e Malcolm Munro. Integration of a reuse process and a maintenance process within a software configuration management environment. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Kem90] Richard A. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, 7(9):37–50, Set. 1990.

[Ker84] Brian W. Kerninghan. The unix system and software reusability. *IEEE Transactions on Software Engineering*, 10(5):513–518, Set. 1984.

[KG87] Gail E. Kaiser e David Garlan. Melding software systems from reusable building blocks. *IEEE Software*, 4(4):17–24, Jul. 1987.

[KLM+97] Gregor Kiczales, John Lamping, Amurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, e John Irwin. Aspect-oriented programming. In *11th ECOOP*, pp. 220–242, Jun. 1997.

[Kna93] Peter J. Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. Tese de doutoramento, University of Teesside, Mar. 1993.

[KR88] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2ª edição, 1988.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, Jun. 1992.

[Kru98] Philippe Kruchten. Modeling component systems with the unified modeling language. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[Lat97] Larry Latour. The need for a cognitive viewpoint on software component understanding. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Lea97] Gary T. Leavens. Specification facets for more precise, focused documentation. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Lew94] Ted Lewis. The dark side of objects. *IEEE Computer*, 27(12):6–7, Dez. 1994.

[LG84] R. G. Lanergan e C. A. Grasso. Software engineering with reusable designs and code. *IEEE Transactions on Software Engineering*, 10(5):498–501, Set. 1984.

[L.G98] Robert L.Glass. Reuse: What's wrong with this picture ? *IEEE Software*, 15(2):57–59, Mar. 1998.

[Lib89] Don Libes. Choosing a name for your computer. *Communications of the ACM*, 32(11):1289, Nov. 1989.

[Lim94] Wayne C. Lim. Effects on reuse on quality, produtivity and economics. *IEEE Software*, 11(5):23–30, Set. 1994.

[Lio96] J. L. Lions. Flight 501 failure: Report by the inquiry board. Relatório técnico, European Space Agency – Centre National D'Etudes Spaciales, Jul. 1996.

[LS98] Fred Long e Robert C. Seacord. A comparison of component integration between JavaBeans and PCTE. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[Man95] Frank Manola. X3H7 object model features matrix. Relatório técnico, GTE Laboratories, Waltham, MA, Fev. 1995.

[McD94] D. McDonald. A convention for human-readable 128-bit keys. Relatório técnico, RFC 1751, Digital Equipment Corporation 1994.

[McF89] Michael C. McFarland. The social implications of computarization: Making the technology more humane. In *26th ACM/IEEE Design Automation Conf.*, pp. 129–134, 1989.

[Mey87] Betrand Meyer. Eiffel: Programming for reusability and extensibility. *ACM SIGPLAN notices*, 22(2):85–94, Fev. 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1988.

[Mey96] Bertrand Meyer. The many facets of inheritance: A taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, Maio 1996.

[MM90] Ole Lehrmann Madsen e Boris Magnusson. Strong typing of object-oriented languages revised. In *OOPSLA*, pp. 140–149, 1990.

[MMM95] Hafedh Mili, Fatma Mili, e Ali Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, Jun. 1995.

[MMM97] Rym Mili, Ali Mili, e Roland T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, Jul. 1997.

[MN97] Neil A. Maiden e Cornelius Ncube. Acquiring COTS software selection requirements. *IEEE Software*, 14(2):46–56, Mar. 1997.

[MPMM98] R. T. Mittermeir, H. Pozewaunig, Ali Mili, e Rym Mili. Uncertainty aspects in component retrieval. In *7th IPMU*, pp. 564–571, Paris, France, Jul. 1998.

[Mul89] Sape Mullender, editor. *Distributed Systems*, capítulo 5, pp. 89–101. ACM Press, 1989.

[Mur97] Tobias Murer. The challenge of the global software process. In *ECOOP, 2nd International Workshop on Component-Oriented Programming*, pp. 69–76, 1997.

[Nei84] James M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, Set. 1984.

[New92] Barry Clifford Newman. *The Virtual System Model: A Scalable Approach to Organizing Large Systems*. Tese de doutoramento, University of Washington, 1992.

[NM95] Oscar Nierstrasz e Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, Jul. 1995.

[NO95] F. Luís Neves e José N. Oliveira. Software reuse by model reification. In *Workshop on Institutionalizing Software Reuse*, 1995.

[Obj95] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2ª edição, 1995.

[OHDB92] Eduardo Ostertag, James Hendler, Rubén Prieto Díaz, e Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Software Engineering and Methodology*, 1(3):205–228, Jul. 1992.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, USA, 1994.

[PA97] John Penix e Perry Alexander. Component reuse and adaptation at the specification level. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[PD91a] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, Maio 1991.

[PD91b] Rubén Prieto-Díaz. Making software reuse work: An implementation model. In *International Workshop on Software Reusability*, Jul. 1991.

[PD93] Rubén Prieto-Díaz. Status report: Software reusability. *IEEE Software*, 10(3):61–66, Maio 1993.

[PDF87] Rubén Prieto-Díaz e Peter Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, Jan. 1987.

[Pet95] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, Jul. 1995.

[PP93] Andy Podgurski e Lynn Pierce. Retrieving reusable software by sampling behaviour. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, Jul. 1993.

[PP94] Santanu Paul e Atul Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–474, Jun. 1994.

[Pri97] Eric V. Price. Organizational culture and behavioral issues affecting software reuse. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Ran96] Brian Randell. The 1968/69 nato software engineering reports. In *History of Software Engineering*, Ago. 1996.

[RD95] Steven P. Reiss e Tony Davis. Experiences writing object-oriented compiler front ends. In *OOPSLA*, 1995.

[Red97] A. L. N. Reddy. Storage considerations in reusable interface design. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Rin91] David C. Rine. A short overview of a history of software maintenance: as it pertains to reuse. *Software Engineering Notes*, 16(4):60–63, Out. 1991.

[RM95] António Nestor Ribeiro e Fernando Mário Martins. A fuzzy query language for a software reuse environment. In *Workshop on Institutionalizing Software Reuse*, 1995.

[Rum91] James Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1991.

[Rum93] James Rumbaugh. Desinherited! examples of misuse of inheritance. *Journal of Object Oriented Programming*, 5(9):22–24, Fev. 1993.

[Sam97] Johannes Sametinger. Component interoperation. In *8th Annual Workshop on Software Reuse*, Mar. 1997.

[Seb93] Robert W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1993.

[Sen97] Arun Sen. The role of opportunism in software design reuse process. *IEEE Transactions on Software Engineering*, 23(7):418–417, Jul. 1997.

[Sha84] M. Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(4):10–26, Out. 1984.

[SJ85] Axel T. Schreiner e H. George Friedman Jr. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1985.

[SLdP98] Marcelo Sant'Anna, Júlio Prado Leite, e António Francisco do Prado. A generative approach to componentware. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[SM95] Webb Stacy e Jean MacMillian. Cognitive bias in software engineering. *Communications of the ACM*, 38(6):57–63, Jun. 1995.

[SM97] Jorgen Steensgaard-Madsen. A generator for composition interpreter. In *ECOOP, 2nd International Workshop on Component-Oriented Programming*, pp. 87–93, 1997.

[Sta84] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, 10(5):494–497, Set. 1984.

[Sta94] Werner Staringer. Constructing applications from reusable components. *IEEE Software*, 11(5):61–68, Set. 1994.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 2ª edição, 1991.

[SvD95] Karen R. Sollins e Jeffrey R. van Dyke. Linking in a global information architecture. In *4th International WWW Conf.*, Boston, MA, USA, Digital Equipment Corporation 1995.

[Tai92] Antero Taivalsaari. Why should object-oriented forths be based on prototypes rather than classes. In *FORML'92*, 1992.

[Tai98] Stefan Tai. A connector model for object-oriented component integration. In *International Workshop on Component-Based Software Engineering*. Software Engineering Institute, 1998.

[TFB92] Antero Taivalsaari e Bjorn Freeman-Benson. Towards fine-grained reusability with self-sufficient objects. In *OOPSLA'92 workshop on Object-Oriented Programing Languages - the Next Generation*, pp. 239–245, 1992.

[Tho98] Craig Thompson, editor. *Workshop on Compositional Software Architectures*, Monterey, California, Fev. 1998.

[TPW95] Robert Haddon Terry, Margaretha Price, e Louann Welton. Standardized software classification in the world wide web. In *Workshop on Institutionalizing Software Reuse*, 1995.

[TV97] Jose M. Troya e Antonio Vallecillo. On the addition of properties to components. In *ECOOP, 2nd International Workshop on Component-Oriented Programming*, pp. 95–103, 1997.

[Ude94] Jon Udell. Componentware. *BYTE*, pp. 46–56, Maio 1994.

[US87] David Ungar e Randall B. Smith. Self: the power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–241, 1987.

[vdL91] R. J. van der Linden. Federated naming model. Relatório técnico, ESPRIT Project 5279, 1991.

[vMV95] Anneliese von Mayrhauser e A. Marie Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, Ago. 1995.

[WCG92] M. Wein, W. Cowan, e W. M. Gentleman. Visual support for version management. In *Symposium on Applied Computing ACM/SIGAPP*, pp. 1217–1233, Mar. 1992.

[Weg84] Peter Wegner. Reusability of software components. *IEEE Software*, 1(4):12–22, Jul. 1984.

[Weg92] Peter Wegner. Dimentions of object-oriented modeling. *IEEE Computer*, 25(10):12–20, Out. 1992.

[WES87] Scott N. Woodfield, David W. Embley, e Del T. Scott. Can programmers reuse software? *IEEE Software*, 4(4):52–59, Jul. 1987.

[You89] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1989.

[Zar96] Amy Morrmann Zaremski. *Signature and Specification Matching*. Tese de doutoramento, Carnegie Mellon University, Jan. 1996.

[ZBS98] Mansour K. Zand, Bradley J. Balentine, e Mansur H Samadzadeh. Fuzzy metrics and code understandability: A study. In *7th IPMU*, pp. 718–725, Paris, France, Jul. 1998.

[ZW95] Amy Moormann Zaremski e Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, Abr. 1995.

[ZW97] Amy Moormann Zaremski e Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, Out. 1997.