Ensinar objectos: o ciclo de vida completo

Ana Moreira e Pedro Guerreiro

Departamento Informática
Faculdade Ciências Tecnologia, Universidade Nova Lisboa
2825 Monte da Caparica PORTUGAL
TEL: + 351-1-2948536; FAX: + 351-1-2948541
{amm|pg}@di.fct.unl.pt

Abstract

Ao sair de um curso superior de Engenharia Universitária, os recém-licenciados, devem, entre outras capacidades, ter um domínio completo sobre o desenvolvimento de *software* orientado pelos objectos. Esse domínio, não se pode restrigir às actividades de programação, deve sim abranger todo o ciclo de vida. Como a tecnologia dos objectos não vem substituir as tecnologias mais antigas, mas sim complementá-las, contribuindo para que novos problemas possam ser resolvidos, há que arranjar mais espaço nos currículos das licenciaturas para as matérias respeitantes ao desenvolvimento de *software*. Para um curso de cinco anos, pensamos que deve haver quatro semestres de programação: o primeiro para a programação elementar convencional, o segundo para a programação com classes, o terceiro para a programação genérica, para a programação de bibliotecas de classes e para a programação com concorrência, e o quarto para a programação visual e para a programação distribuída. Esta fileira de programação deve ser complementada com dois semestres de metodologias de desenvolvimento de *software*: o primeiro para as metodologias de análise e de desenho estruturado, e o segundo para as metodologias de desenvolvimento orientadas pelos obejctos. Propomos ainda um semestre de métodos formais onde os alunos podem aprender conceitos e técnicas que dificilmente terão oportunidade de aprender fora da universidade: técnicas de descrição formal, validação e verificação de especificações, produção de uma especificação formal a partir de um conjunto de requisitos informais. E, finalmente, outro semestre de Engenharia de *Software* para fechar o ciclo.

1 Introdução

Os métodos de análise e desenho orientados pelos objectos e as linguagens de programação orientadas pelos objectos estão a tornar-se a forma preferida de construir *software*. A indústria já oferece acções de formação em várias áreas da tecnologia dos objectos e as universidades estão a reformular os seus cursos em Engenharia Informática de modo a que vários aspectos da orientação pelos objectos façam parte dos currículos. Por isso, ensinar a tecnologia dos objectos é uma actividade em expansão, mas, em muitos aspectos, ainda uma experiência pedagógica. Com efeito, se as ideias básicas — objectos, classes, herança, polimorfismo — estão estabelecidas, se já sabemos que a orientação pelos objectos diz respeito a todo o processo de desenvolvimento, se a indústria cada vez mais pede à universidade para lhe fornecer quadros informáticos com preparação em tecnologia dos objectos, também é verdade que a didática dos objectos ainda vai na sua infância.

Para ajudar a perceber onde estamos é útil recordar de onde viemos. Há vinte anos, depois do advento da programação estruturada, o ensino da programação estava estabilizado. Dispúnhamos de uma linguagem, o Pascal, que nos ajudava verdadeiramente na nossa tarefa, e que nos permitia concentrar no que era fundamental. Tínhamos até alguns livros que se tornaram clássicos, como o *Structured Programming*, de Dijkstra, Hoare e Dahl [Dahl 72], o *Algorithms + Data Structures = Programs*, de Wirth [Wirth 76], o *The Discipline of Programming*, de Dijkstra [Dijkstra 76], o *The Science of Programming*, de Gries [Gries 81], por exemplo. E havia um conjunto variado de livros didáticos que inspiravam confiança, alguns com traduções razoáveis para Espanhol e para Português, nos quais podíamos basear o nosso curso, e os quais podíamos recomendar sem receio aos nossos alunos. Finalmente, a componente prática era realizada em

terminais alfanuméricos ligados a um computador central multi-utilizador. Todos os alunos usavam o mesmo compilador para escrever programas que faziam *input-output* para o ecrã ou para ficheiros.

Para as metodologias, podíamos contar com o clássico *Structured Analysis and System Specification* de Tom DeMarco [DeMarco 78], ou com o *Structured Systems Analysis* de Gane e Sarson [Gane 79], para modelar a componente dinâmica de um sistema, ou com o *The Entity-Relationship Model* de Chen, ou o *Data Analysis for Data Base Design* de Howe [Howe 89], para a componente estática, por exemplo. Já para a fase de desenho podíamos basear-nos no *Structured Design* de Yourdon e Constantine [Yourdon 79] ou no *The Practical Guide to Structured Systems Design* de Page-Jones [Page-Jones 80]. Claro que também havia livros que integravam ambas as fases do ciclo de vida. Mais recentemente, podíamos ainda combinar a nova visão de fazer análise estruturada proposta no *Modern Structured Analysis* de Yourdon [Yourdon 89] com qualquer outro método que suportasse o modelo de Chen. A parte prática, essa era levada a cabo usando apenas papel e lápis. Mais tarde, nos finais da década de 80 apareceram as ferramentas CASE (*computer aided sofwtare engineering*) que vieram revulcionar o modo de usar métodos de desenvolvimento estruturados.

Actualmente, depois da "revolução" dos objectos, a situação complicou-se para nós, professores de programação e de análise de sistemas. Não só temos agora duas tecnologias para ensinar, como no caso da mais recente as coisas estão longe de ter acalmado. Por isso, para além de termos nós próprios de aprender uma série de coisas novas (ainda bem!), temos muito menos ajuda na selecção das matérias a ensinar depois. Por um lado, não existem ainda livros "clássicos" para a tecnologia dos objectos (com a possível excepção do *Object Oriented Software Construction*, de Meyer [Meyer 97]), e, por outro, a abundância de manuais e livros de texto faz-nos desconfiar da qualidade. Para mais, algumas das linguagens estão em evolução (ainda que com a promessa de rapidamente estabilizarem). E, ainda por cima, os programas que agora queremos escrever, mesmo os mais simples, devem ter interfaces gráficas, com caixas de diálogo, menus e botões, e isso coloca problemas de plataforma de desenvolvimento e de execução.

Quanto às metodologias, os problemas são semelhantes, mas talvez menos difíceis de ultrapassar. Apesar de a tecnologia dos objectos ainda estar numa fase de evolução, existem várias dezenas de métodos e de ferramentas CASE, fáceis de obter. Localizar entre eles o método ideal e a ferramenta perfeita é uma tarefa complicada, mas se não formos muito exigentes e se estivermos preparados para integrar propostas de dois ou três autores, podemos nós próprios criar com uma abordagem interessante e que colmata as fraquezas de uns e outros métodos. Quanto aos CASEs, o melhor é não esperar demasiado: encaremo-los apenas como ferramentas de apoio ao desenho dos modelos para não termos grandes desilusões [Oliveira 98].

Este trabalho é uma reflexão sobre as questões que se colocam aos professores de programação e de análise de sistemas nesta mudança de tecnologia. Começamos, na Secção 2, por reconhecer que a tecnologia dos objectos diz respeito ao ciclo de vida completo do *software*, desde a análise (ou mesmo antes) até à programação (ou mesmo depois). Depois, na Secção 3, discutimos a questão de por onde começar: pela programação? pela análise? pelas funções? pelas classes? A seguir, na Secção 4, mostramos o modelo de ensino destas matérias na licenciatura em Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, em que participamos. Este modelo é, quanto a nós, muito insuficiente, mas é o resultado de equilíbrios internos que tivemos que aceitar. Na Secção 5, apresentamos, para um plano de curso de cinco anos, a sequência de disciplinas que, quanto a nós, deveriam ser dedicadas ao desenvolvimento de *software*, de forma a proporcionar aos alunos uma formação completa na tecnologia dos objectos.

2 Tecnologia dos objectos para o ciclo de vida do *software*

É verdade que a tecnologia dos objectos começou pela programação, inicialmente com o Simula-67, e depois, já nos tempos modernos, com o Smalltalk e o C++. No entanto, tal como acontecera com a programação estruturada uma geração antes, depressa alastrou para abranger todo o ciclo de vida, primeiro o desenho e depois a análise. Curiosamente, o primeiro método de desenho orientado pelos objectos produzia especificações de pacotes am Ada e foi proposto por Booch, no seu livro *Software Enginnering with Ada*. Logo a seguir, nos finais da década de 80, apareceu o primeiro método de análise, para ser integrado com o de Booch, o *Object-Oriented Requirements Analysis*, proposto pela companhia americana EVB Software Engineering Inc [EVB 89]. A partir daqui floresceram muitos métodos orientados pelos objectos. Para suportar estes métodos também foram sendo construídas ferramentas CASE. Todavia, estas ferramentas não têm ainda uma robustez comparável com as dos CASEs para desenvolvimento estruturado [Oliveira 98]. Esperamos que a recente linguagem de modelação UML (*Unified Modeling Language*) [Rational 97] possa

vir ajudar a estabilizar tantos os métodos como as ferramentas. Sendo um *standard*, tanto os criadores de métodos como os construtores de ferramentas podem inpirar-se nele e preocuparem-se apenas com a qualidade dos seus produtos.

Actualmente aceita-se que para tirar todo o partido da tecnologia dos objectos não basta começar com ela na fase da programação: é preciso integrá-la em todo o processo de desenvolvimento. Mas nem sempre foi assim. É verdade que durante algum tempo houve quem sustentasse que era possível fazer programação orientada pelos objectos a partir de uma análise estruturada [Shumate 91, Constantine 89a, Constantine 89b, Ward 89]. No entanto, os conceitos utilizados pelas abordagens estruturadas são muito diferentes dos utilizados na tecnologia dos objectos. Integrar uma especificação funcional com uma implementação orientada pelos objectos cria-nos dificuldades que preferimos evitar. Em particular, a transição e a integração entre as várias fases fica ainda mais problemática.

Nós acreditamos que a dificuldade em combinar as duas abordagens é desnecessária. Na nossa opinião, a atitude correcta para o desenvolvimento de *software* orientado pelos objectos, é usar os seus conceitos desde a análise (ou mesmo antes) até à programação (ou mesmo depois). Idealmente, todo o desenvolvimento terá sido suportado por ferramentas elas também orientadas pelos objectos, onde os conceitos da análise e do desenho têm uma representação explícita.

3 Por onde começar

Porque o desenvolvimento de um projecto começa na análise, isto não quer dizer que a sequência da leccionação tenha que ser primeiro a análise, depois o desenho, e finalmente a programação. Aliás, uma questão recorrente a respeito do ensino da programação no contexto de uma licenciatura em Engenharia Informática, em que se pretende formar profissionais capazes de intervir no processo de desenvolvimento de *software* em todas as suas fases, é saber se se deve começar pela programação, ou, se, em vez disso, e imitando o ciclo de vida tradicional, é melhor estudar primeiro os métodos de análise. Esta questão é antiga e nos casos que conhecemos, tem sido resolvida dando primazia à programação. Em abstracto, pode parecer estranho proceder assim, pois qualquer problema de programação envolve tarefas de análise, e, portanto, um programador conhecedor dos métodos de análise estará mais bem preparado para levar a cabo a sua missão. No entanto, a programação nos cursos iniciais é elementar e os problemas que aparecem são sobretudo de natureza algorítmica. Os objectivos desta programação elementar são fazer os estudandes conhecer os mecanismos básicos de programação (programas, funções, variáveis, passagem de argumentos, tipos, *arrays*, estruturas de controlo, *input-output*, gestão de memória, etc.), os algoritmos de uso geral (busca linear, busca dicotómica, ordenação, leitura de um ficheiro, etc.), e ainda algumas estruturas de dados mais comuns (vectores, ficheiros, listas, arvores binárias, pilhas, filas, tabelas de *hash*, etc.). Como efeito lateral deste ensino preliminar, os estudantes adquirem um conhecimento "intuitivo" sobre os limites dos computadores, e têm um primeiro contacto com a arquitectura das máquinas *(bits*, *bytes*, código ASCII, base 2, precisão das operações aritméticas, velocidade dos cálculos, modelo de execução.)

Estes conhecimentos não podem ser menosprezados, e temos que encontrar um lugar para os discutir no início do curso. Se não houver uma cadeira introdutória vocacionada para estas questões, caberá à cadeira inicial de programação apresentar estes assuntos.

3.1 Programação estruturada *versus* orientada pelos objectos

Quais são então os objectivos da cadeira inicial de programação, para além dos de promoverem a familiarização dos estudantes com a utilização técnica dos computadores? Se nos colocarmos num contexto de ingresso apressado na tecnologia dos objectos, os objectivos seriam tornar os alunos conhecedores dos fundamentos da programação orientada pelos objectos. É esta a atitude adoptada por muitos colegas nossos, com êxito, queremos acreditar, mas não é isso que nós fazemos.

Vejamos: qual é o conceito central da tecnologia dos objectos? É o conceito de classe. As classes encapsulam dados e operações, normalmente os dados são privados e as operações públicas. Um programa orientado pelos objectos (em C++, em Java, em Eiffel) é um amontoado de classes, relacionadas umas com as outras. A questão fundamental do desenvolvimento orientado pelos objectos é, portanto, identificar as classes que fazem falta. Dessas, umas até já estão prontas, em bibliotecas de classes, outras seremos nós a programar.

Depois, no programa em execução, surgem os objectos, que são instâncias das classes, e os objectos enviam mensagens uns aos outros. Como apresentar estas ideias, que não são simples, mas que podem fazer sentido para quem já tenha alguma experiência de desenvolvimento de *software*, a estudantes sem preparação prévia em programação? Como nós não sabemos, optamos por uma atitude mais prudente, *bottom-up*, em que começamos por conceitos mais elementares.

Que conceitos mais elementares são esses? Bom, são afinal os ingredientes básicos da programação estruturada: as funções e os procedimentos. Quer dizer, ao tentar resolver problemas de programação simples, como o do cálculo da área de um triângulo dadas as coordenadas do vértices [McCracken 65], ou o do posto de câmbios, ou o preço a pagar num parque de estacionamento com tarifas horárias crescentes, o que começamos por fazer é identificar as funções que fazem falta. No caso da área do triângulo, temos a função para medida do lado dadas as coordenadas dos vértices, e a função para a fórmula da área dadas as medidas dos lados; para o posto de câmbio, temos as funções para achar a taxa de câmbio, para calcular os contravalores, para fazer os arredondamentos, para aplicar a comissão de serviço; para o parque de estacionamento, temos o cálculo da duração de um intervalo de tempo, a aplicação de tolerâncias ou bónus, e a fórmula que representa o tarifário, o cálculo do troco.

Esta abordagem pode ser conduzida sem grandes delongas, pois os alunos trazem da Matemática do ensino secundário o conceito de função, como transformação determinística de um conjunto noutro, e não é preciso mais do que isso. A programação inicial baseia-se nisso apenas, e numa linguagem de programação com a qual se descrevem inequivocamente as funções.

Todos estes programas aceitam perfeitamente uma implementação unimodular. É verdade que em todos eles há vários tipos em jogo, e podíamos decompor o sistema em vários módulos para arrumar melhor as coisas. Por exemplo, no caso do parque de estacionamento, poderíamos ter um módulo para os cálculos relacionados com o tempo, e outro para as operações de pagamento. No entanto, esta discussão sobre a melhor maneira de decompor o sistema, nesta fase inicial, só serviria para distrair os alunos do objectivo central.

Note-se que se é verdade que o conceito de função que vem da Matemática é importante, a sua concretização em Programação requer bastantes explicações técnicas: qual o mecanismo de passagem de parâmetros? para que servem as variáveis locais? como se pode aceder às variáveis locais? e as funções recursivas, como funcionam? e quando é que devemos usar um procedimento em vez de uma função?

Se considerarmos que estas questões devem ser tratadas logo no início, temos que ter um ambiente onde elas possam ser analisadas "experimentalmente". Esse ambiente é o da programação Pascal. É verdade que o Pascal está a sair de moda, mas, em nossa opinião, os candidatos a substitutos — Smalltalk, Eiffel, Ada 95, C++ — não ajudam verdadeiramente a discutir os temas da programação básica com a mesma objectividade que o Pascal permite. Existe, é verdade, um outro candidato, curiosamente da mesma estirpe que o Pascal, a linguagem Oberon [Wirth 92], mas não tem tido divulgação alargada.

Esta discussão tem-se centrado na problemática do esclarecimento do conceito de função, mas com as devidas adaptações, aplica-se igualmente a outros temas elementares, por exemplo: como fazer uma busca num vector, tendo o cuidado de não cair em "*index out of bounds*" se não encontrarmos o que procuramos? como carregar um ficheiro para um vector, tendo o cuidado de não ler para além do fim do ficheiro? como ordenar um vector, usando o *bubblesort* ou o *quicksort*?

Um ponto de vista mais radical em favor de uma abordagem inicial logo com objectos reclamaria que todos estes temas se podem tratar com vantagem no contexto de uma programação com classes: por exemplo, no problema do triângulo, teríamos classes Ponto e Triângulo; no problema do parque de estacionamento, teríamos classes Tempo, Duração, Dinheiro; ordenar um vector seria uma operação de uma classe Vector; etc.

Sem dúvida. No entanto, a infraestrutura que teríamos que montar para permitir discutir com seriedade os problemas nesse contexto só serviria, nesta fase inicial, para desviar as atenções. Teríamos que fazer uma antevisão da relação de herança, teríamos que discutir os modos de acesso (numa altura em que não há propriamente nada a esconder), teríamos que mencionar os construtores e os destrutores e a recuperação de memória, teríamos que esclarecer a distinção entre os argumentos de uma função e o objecto da função, teríamos que gerir o programa multimodular. E, para programar as classes devidamente, deveríamos preocupar-nos com a sua reutilização, o que nos faria incluir nas classes funções que não intervêm na

resolução do problema que as suscitou. Tudo isto são questões interessantes, mas não básicas, e, na nossa opinião discuti-las preliminarmente seria contraprodutivo.

3.2 E na análise de sistemas, por onde começar?

E entre a análise e o desenho, qual se ensina primeiro? Cronologicamente, como já mencionámos, o desenho, quer o estruturado quer o orientado pelos objectos, surgiu antes da análise. No entanto, é vantajoso aprender primeiro a análise. Os métodos de análise e de desenho são apresentados com base em exercícios de média dimensão. Um dos exemplos que temos usado é o sistema da "via verde", para pagamento automático das portagens em pontes e auto-estradas. A análise deste problema prolonga-se por várias aulas. Depois, as técnicas de desenho são alimentadas com os resultados da análise. Fazer de outro modo seria irrealista. Por exemplo, como se explicava as regras da cardinalidade para derivar um esquema de tabelas (desenho) a partir de um diagrama de entidades e associações (análise), se ainda não se tivesse estudado os métodos de análise? Ou, para o caso dos métodos orientados pelos objectos, que faríamos no desenho sem o diagrama de classes e os diagramas de sequência de mensagens? Para o exercício da via verde, identificar classes como Veículo, Passagem, Identificador de via verde, Preçário, Portagem de entrada, Portagem de saída e Portagem de ponto único, etc., tornava-se muito menos óbvio. Para além disso, como explicaríamos a necessidade de objectos de controlo e de interface? Para evitar confusões, o melhor mesmo é começar por ensinar a análise e depois ensinar o desenho. Aliás, não há vantagem em separar as técnicas e na prática os assuntos são tratado em sequência, dentro da mesma cadeira.

Tendo decidido isto, por qual tecnologia devemos começar? Pela tecnologia convencional, estruturada, ou pela dos objectos? Aqui as razões nossa opção inicial pelas metodologias estruturadas são idênticas às do ensino da programação: conhecimentos prévios dessas metodologias permitem aligeirar e simplificar o ensino e a aprendizagem da tecnologia dos objectos. Apesar de tudo, um diagrama de classes ainda contém muitas ideias tiradas do um diagrama de entidades e associações. Não há vantagens em ignorar isso.

Note-se que para além de facilitarem o estudo posterior do métodos orientados pelos objectos, os métodos estruturados são necessários por si só na cadeira de Bases de Dados, que ocorre pouco depois. É verdade que em relação às bases de dados se podia colocar a mesma questão: estuda-se primeiro as bases de dados "tradicionais" ou as bases de dados orientadas pelos objectos? No entanto, temos que admitir, o problema não é tão importante como com a programação, uma vez que as bases de dados orientadas pelos objectos estão ainda numa fase embrionária. As bases de dados orientadas pelos objectos (melhor seria chamar-lhes "bases de objectos") ainda não nos inspiram a confiança com que olhamos para as bases de dados relacionais. Por isso, aí a primazia vai para as bases de dados relacionais e as de objectos são depois ensinadas juntamente com a implementação de bases de dados, com o controlo de acesso concorrente e com as bases de dados distribuídas.

4 Um plano curricular minimalista

O elenco das disciplinas que formam um curso universitário resulta de contributos variados, dos professores que vão estar envolvidos directamente no ensino, dos seus colegas de outras escolas a quem foi pedido o conselho, das associações profissionais, que querem garantir a boa formação dos seus futuros membros, da indústria, que precisa dos licenciados para prosseguir os seus objectivos… Os próprios estudantes, ainda que não intervindo nos conteúdos das cadeiras, acabam por ter uma grande influência pois podem vetar na prática mudanças curriculares a meio do curso, se desconfiarem delas.

Todos estes parceiros têm visões diferentes acerca do que é importante. Se uns estão convencidos de que a tecnologia dos objectos é fulcral e que deve permear todas as matérias, outros há que acham que se trata apenas de mais uma maneira de desenvolver *software*, e que o seu âmbito se circunscreve a algumas cadeiras de programação e análise de sistemas. É esta a situação que temos actualmente na nossa faculdade, a Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa.

No actual plano curricular da licenciatura em Engenharia Informática, o ensino da tecnologia dos objectos ocorre em apenas dois dos dez semestres do curso: o segundo e, estranhamente, o décimo.

No segundo semestre estuda-se os fundamentos da programação orientada pelos objectos. O objectivo é ajudar os alunos a aderirem naturalmente aos conceitos de classe, a perceberem que é normal e desejável que certas informações estejam escondidas, enquanto outras ficam visíveis, a relacionarem as classes entre si, sem inibições. Tal como foi argumentado anteriormente, esta cadeira vem a seguir a uma outra onde se tratou da programação estruturada. Note-se que tal como nessa primeira cadeira não se teorizou sobre o ser programação estruturada ou não, também na segunda não se insiste no facto se agora as coisas serem orientadas pelos objectos. Aliás, o ênfase principal é colocada nas classes, na identificação das classes e na maneira de as relacionar umas com as outras.

No décimo semestre, e em paralelo com a disciplina de projecto que geralmente é levada a cabo fora da universidade, estuda-se modelação orientada pelos objectos. Quer dizer, os estudantes, depois de aprenderem programação orientada pelos objectos com C++ no segundo semestre, passam virtualmente todo o curso sem aprenderem mais nada sobre objectos. O que era preciso era que esta disciplina fosse leccionada mais cedo. Até porque, muitos dos projectos de fim de curso são para ser ser desenvolvidos usando a tecnologia dos objectos! (Os métodos estruturados, por sua vez, são ensinados no sétimo semestre.)

Em termos da matéria a ensinar, o problema principal para o professor é escolher uma abordagem coerente entre a análise e o desenho, mas que seja capaz de integrar técnicas que suportem bem a modelação das características comportamentais e estáticas de um sistema. Na nossa opinião, não existe um método único que suporte igualmente bem estas duas componentes de um sistema. Uns dão primazia ao comportamento, como é o caso do OBA [Rubin 92] e OOSE [Jacobson 1991], outros preocupam-se mais com a identificacão de objectos entidade, como é o caso do OOA [Coad 1991] e OMT [Rumbaugh 91], apesar deste último incluir um bom modelo dinâmico. A solução usada por nós tem sido fazer um *cocktail* de métodos, guardando de vários deles aquilo que consideramos os seus pontos fortes.

5 O plano curricular recomendado

Em nossa opinião, o estado actual é pobre. A nossa recomendação para os debates internos e que aqui expomos, consiste em criar uma fileira permanente de tecnologia de objectos ao longo do curso, com uma cadeira em cada semestre. Propomos assim quatro cadeiras de programação, duas de metodologias de desenvolvimento, uma de métodos formais, uma de engenharia de *software*, uma de ambientes de desenvolvimento, e uma de projecto.

5.1 Programação geral

Quatro cadeiras de programação pode parecer excessivo, sobretudo para aqueles que estão agarrados à ideia de que o que interessa é o "raciocínio", e que quem programa numa linguagem programa depois noutra linguagem qualquer. Tomado à letra, isto é falso, em nossa opinião. Quanto muito, quererá dizer que quem sabe programar numa linguagem, terá, em princípio, menos dificuldade em aprender uma segunda, mas isto é uma banalidade. O C++, o Eiffel, o Ada 95 têm características próprias, e modos de abordar, que têm que ser aprendidos. Quem sabe C não aprende C++ nas calmas, quem sabe Pascal não fica um perito em Eiffel ou Ada 95 com uma rápida leitura dos manuais. Inversamente, se a aprendizagem for conduzida por professores com experiência, o esforço será muito mais efectivo.

Acresce que a programação moderna não se resume a manipular com à-vontade funções, procedimentos e as três figuras da programação estruturada. Por um lado, há que incorporar desde logo as preocupações da Engenharia de *Software* — modificabilidade, fiabilidade, eficiência, compreensibilidade [Ross 75]. Por outro lado, há que tirar partido das bibliotecas de classes disponíveis, usando-as numa primeira fase apenas através da sua especificação, tal como Meyer recomenda [Meyer 97].

Para além disso, o âmbito das modernas linguagens (C++, Eiffel, Ada 95) é muito mais vasto que o das linguagens clássicas (FORTRAN, COBOL, Pascal, C), e se para estas antigamente era preciso um ano de estudos, dois anos agora certamente não é demais. Esclareçamos que os quatro semestres que propomos não esgotam o estudo da programação. Fazendo uma analogia com cadeiras de outros domínios chamadas Matemáticas Gerais ou Física Geral, o que temos aqui é Programação Geral. Há que prever também as cadeiras de Programação em Lógica, Algoritmos e Estruturas de Dados, Linguagens de Programação, Computação Gráfica, Compilação, Bases de Dados, pelo menos.

Vejamos então como organizar os quatro semestres de programação geral.

Os dois primeiros semestres correspondem às ideias que defendemos na Secção 3: começa-se, no primeiro semestre, pela programação elementar convencional, com funções, procedimentos, variáveis, tipos, estruturas de dados simples, usando Pascal. No segundo semestre, entra-se na programação orientada pelos objectos, com classes, e todo o restante arsenal. Se a escolha da linguagem para a programação elementar era relativamente pacífica, agora já é mais controversa.

Para ensinar programação com classes, precisamos de uma linguagem que suporte classes, é óbvio. Felizmente, há muito por onde escolher. Se quiséssemos rentabilizar ainda mais o esforço que os estudantes dispensaram no primeiro semestra aprendendo a sintaxe e a semântica do Pascal, podíamos pensar em usar o ambiente Delphi. No entanto, isso teria o inconveniente de atar o ensino a uma determinada plataforma, o que não é desejável numa fase tão inicial do curso. Se preferíssemos uma abordagem "pura", podíamos optar pelo Smalltalk. Mas então, grande parte do esforço do primeiro semestre era desperdiçado, pois o Smalltalk é uma linguagem *sui generis*. Alternativamente, existe o Eiffel, que também é uma linguagem orientada pelos objectos "pura". É porventura uma hipótese a considerar no futuro, mas nesta altura, para este efeito, cremos que seria incompreendida. Uma escolha que teria certamente muitos adeptos seria o Java, que irrompeu no mundo da programação com um fulgor nunca visto. No entanto, a linguagem Java é apenas uma parte da tecnologia Java, a qual merece ser tratada à parte, não havendo grande vantagem em isolar, antecipando-a, a componente de programação. A linguagem menos recomendável de todas, tanto em termos pedagógicos, como em termos de Engenharia de *Software* é o C++, mas acaba por ser essa a nossa escolha. Justifiquemo-la então um pouco mais.

O C++ tem todos os ingredientes necessários, ou quase todos, para fazer uma programação com classes, orientada pelos objectos. Os mais cépticos dirão até que tem ingredientes demais, e que alguns deles estão mal acabados. Mas recordemos que os objectivos do segundo semestre de programação geral são a programação com classes. O C++ tem classes, as classes C++ são um bom mecanismo de encapsulação, com a declaração separada da implementação, o que é interessante (na ausência de um ambiente de desenvolvimento que permitisse várias vistas sobre o código), há dois níveis de acesso principais — público e privado — que servem perfeitamente, o qualificador *const* nas funções permite distinguir claramente aquelas que apenas consultam o estado do objecto (as funções *const*) e as que o modificam, há construtores e também há destrutores, tem herança múltipla, tem classes genéricas através do mecanismos das *templates*, e permite programar excepções. Tem também muitas coisas que não fazem falta, nesta fase de aprendizagem, e que podem meter-se no caminho, tornando-o mais difícil: funções *friend*, variáveis globais, funções *inline*, herança privada, membros protegidos. Compete aos professores chamar a atenção para o que é importante, e proibir, se for caso disso, a utilização de construções mais complicativas.

Tem sido esta a nossa abordagem desde há alguns anos. As nossas observações indicam que no fim do semestre os alunos começam a não estranhar programar com classes, a achar natural organizar as hierarquias de herança, a espontanemante desenhar classes em que alguns membros são objectos de outras classes. Tal como no fim do primeiro semestre os alunos devem ter ganho uma certa "intuição" para a decomposição funcional, também aqui pretendemos que eles tenham ganho a "intuição" para as classes. Isto não significa que o desenvolvimento de *software* se baseie apenas na intuição. Pelo contrário. As disciplinas sobre metodologias de desenvolvimento aparecem depois para permitir ir mais longe, para permitir resolver problemas que não se compadecem com uma aproximação tão simplista. Mas o fundamental é que os estudantes encarem os conceitos de classe, objecto, herança, polimorfismo, não como umas coisas avançadas que só se usam em problemas complicadas, mas como uma ferramenta do dia-a-dia da programação.

Como exemplos da utilização das classes, apresentam-se durante este semestre aquelas estruturas de dados clássicas — listas, pilhas, filas de espera, tabelas de *hash* — que não houve necessidade de introduzir no semestre anterior. Aqui estas estruturas têm a programação certa, com a devida encapsulação, com a implementação separada da especificação, e aparecem integradas numa hierarquia de classificação (e herança). Na prática, no entanto, verifica-se que não há tempo, nem disponibilidade mental da maioria dos estudantes para chegar à programação genérica. A programação genérica em C++ tem as suas manhas, e é melhor entrar nela só quando os estudantes já têm as bases bem assimiladas. O que faz mais falta aqui é um mecanismo de asserções no estilo do Eiffel, com o qual pudéssemos exprimir as pré-condições e as pós-condições das funções e os invariantes das classes.

A programação genérica é um dos temas do terceiro semestre, e surge com duas facetas: utilização de classes genéricas, nomeadamente a biblioteca STL do C++, e programação de classes genéricas, para criação de bibliotecas de classes. É verdade que o modelo de genericidade usado no C++ é inferior ao do Ada ou ao do Eiffel, e por isso é altura de introduzir estas duas linguagens. A apresentação pode ser relativamente apressada, sobretudo no caso do Eiffel, pois os conceitos básicos já estão adquiridos. Quanto ao Ada, a presença de pacotes em vez das classes pode ser perturbadora, mas é importante distinguir os dois tipos de programação, com classes e com pacotes.

Uma vez introduzida a linguagem Ada, com o pretexto dos pacotes e da programação genérica, aproveita-se a oportunidade para estudar a programação concorrente, usando os mecanismos que o Ada oferece para esse tipo de programação. Provavelmente não é preciso ir muito longe, pois os estudantes reencontrarão essa problemática nas cadeiras de Sistemas Operativos, mas o contacto com a concorrência tem grande valor formativo.

Finalmente, o objectivo do quarto semestre de programação é tornar os estudantes aptos, por um lado a programar interfaces gráficas sofisticadas, usando as bibliotecas de classes disponíveis para isso, e por outro a saber usar os sistemas de programação visual, exemplificados pela família VisualAge, da IBM. A linguagem a usar nesta cadeira é o Java, finalmente. Uma vez que os conceitos básicos da tecnologia dos objectos já estão assentes, e que os estudantes estão à-vontade com as bibliotecas de classes, a aprendizagem pode concentrar-se em tirar partido dos aspectos inovadores da tecnologia Java, nomeadamente a possibilidade de escrever programas que podem correr, sem recompilação, em qualquer plataforma onde haja uma máquina virtual Java. Isto envolve programar na arquitectura cliente-servidor, com o cliente a residir numa máquina e o servidor noutra, usando o mecanismo *remote method invocation*.

No espaço de dois anos lectivos os estudantes terão aprendido cinco linguagens de programação — Pascal, C++, Ada, Eiffel e Java — umas com mais profundidade outras com menos. Pode parecer muito, e é, mas não é demais. Aliás, destas linguagens todas, a única que pode ser dominada completamente, sem recurso a auxiliares de memória (*help on line*, livros, manuais) é o Pascal. As outras são linguagens "de manual": como são muito mais vastas, como incluem biblitecas de classes (de pacotes no caso do Ada), não faz muito sentido conhecê-las de uma ponta à outra. O que é preciso são conhecimentos básicos muito firmes e agilidade para procurar o que faz falta.

5.2 Modelação

A seguir às cadeiras de Programação vêm as de metodologias. Aqui estudaríamos, desde o quinto ao sétimo semestres, três tipos de métodos: métodos estruturados, primeiro, métodos orientados pelos objectos, logo a seguir, e métodos formais, finalmente. Todos estes tipos de métodos devem ser ensinados usando ferramentas de apoio: CASEs para os dois primeiros casos, e simuladores para o último caso.

De entre os métodos estruturados existentes, a nossa preferência, para a fase de análise, vai para análise essencial, proposta pelo método *Modern Structured Analysis* de Yourdon [Yourdon 89]. Claro que se utilizássemos o método SSADM [Downs 88], por exemplo, ficaríamos logo com cobertura também para a fase de desenho. No entanto, a análise essencial permite-nos: ganhar tempo, pela construção de menos modelos; dar uma visão dinâmica integrada do sistema; facilitar a identificação das funções que compõem o sistema, através da identificação de eventos; facilitar a decomposição funcional lógica, sem correrrmos o risco de sermos influenciados pela estrutura física da organização. Além disso, o modo como os vários modelos estão divididos ajuda a separar ideias, e a notação usada pelo dicionário de dados é pequena e simples, permitindo no entanto descrever as entradas de uma forma curta e sem ambiguidades.

Para a fase de desenho inspiramo-nos nos métodos de Page-Jones [Page-Jones 80] e de Howe [Howe 89]. Nesta fase, é preciso ensinar as formas de normalização, a derivar um esquema de tabelas para uma base de dados relacional, e a desenhar diagramas de estrutura para facilitar a definição do número de aplicações a construir e, para cada uma, a identificação dos módulos que constituem os programas a implementar.

Quanto a ferramentas CASE, podemos escolher algo simples e barato, como o EasyCase, ou algo mais robusto e de maior dimensão, como o ADW ou o Obsydian.

No sexto semestre, a escolha do método é mais problemática. Ainda não temos clássicos, como nos métodos estruturados, e o advento do UML veio dificultar, nesta fase inicial, a decisão.

Cada método propõe o seu próprio processo, as suas próprias técnicas e modelos para construir. Por exemplo, o OMT [Rumbaugh 91] propõe a construção de um modelo de objectos, de um modelo dinâmico e de um modelo funcional. Nós gostamos do modelo dinâmico, mas pensamos que o modelo de objectos pode ser melhorado e que o modelo funcional tem pouca utilidade. Por isso, propomos um modelo de objectos melhorado e integrado com um modelo dinâmico. O modelo de objectos é dado por um diagrama de classes e o modelo dinâmico é dado por um conjunto de diagramas de sequência de mensagens e diagramas de estado.

O diagrama de classes deve incluir serviços oferecidos e diferenciar associações (que implicam conhecimento) e comunicação. (Isto segue as ideias do modelo de objectos proposto por Coad e Yourdon [Coad 91].) O método de Rumbaugh, e também o de Coad e Yourdon, são limitados, por focarem quase exclusivamente objectos entidade; eles concentram-se fundamentalmente em problemas virados para as bases de dados. Nós acreditamos que devemos começar a pensar em objectos interface ainda no início do desenvolvimento, se possível usando prototipagem rápida ainda na fase de análise. Nisto preferimos a visão de Jacobson [Jacobson 92] onde objectos entidade, objectos interface e objectos de controlo são todos necessários. Os objectos de controlo aparecem sempre em problemas com comportamento dinâmico interessante. Nós tratamo-los como entidades transientes que são criadas dinamicamente para levar a cabo uma função e depois terminam. Do método de Jacobson também achamos que os *use cases* são particularmente relevantes, para nos ajudar a descobrir e completar os requisitos. O nosso curso baseia-se principalmente nos três métodos referidos acima, mas usa ainda algumas ideias provenientes do método VMT (*Visual Modeling Technique*) [Tkach 96] da IBM.

Colegas noutras universidades estão a basear o seu ensino no UML. Nós temos algumas desconfianças relativamente a essa atitude, já que, na nossa opinião, é importante usar um método coerente para ensinar os conceitos da tecnologia dos objectos. Ora o UML é uma notação grande, ainda em fase de desenvolvimento e com muitas inconsistências para serem resolvidas. (No que diz respeito a isto, nós próprios temos vindo a trabalhar para formalizar um subconjunto do UML com o objectivo de produzir uma ferramenta coerente para modelação.) O UML pode vir a tornar-se muito importante para os construtores de ferramentas, mas precisa ainda de ser integrado num processo, ou num método, para ser útil num curso como o nosso. Dito isto, nós adoptamos a notação e a terminologia do UML, mas não tornamos o UML o instrumento básico do ensino que fazemos.

O nosso objectivo final é especificar o problema como um conjunto de objectos concorrentes, cujas classes estão representadas num diagrama de classes (o qual constitui o modelo de objectos). Começamos com um conjunto de requisitos informais, onde identificamos objectos "óbvios" (geralmente objectos entidade e interface) e iniciamos a construção do diagrama de classes. Para problemas com comportamento dinâmico forte, os restantes objectos podem ser difíceis de encontrar. Neste caso concentramo-nos no comportamento, identificamos *use cases* e, por cada um, construímos um diagrama de sequência de mensagens. A primeira mensagem no diagrama corresponde à motivação inicial que originou o *use case*. (Os nossos diagramas de sequência de mensagens estendem os diagramas de eventos propostos pelo OMT, incorporando alternativas e ciclos de modo a minimizar o número de diagramas a desenhar.) À medida que construímos estes diagramas, identificamos objectos que faltam (especialmente objectos de controlo) e a passagem de mensagens pode levar-nos a identificar novos serviços. Quanto a ferramentas CASE, podemos usar qualquer uma de domínio público, ou então o OMTool da Martin e Martietta, ou o Rose da Rational.

Quanto aos métodos formais, é a única cadeira do género que os alunos têm durante o seu curso. O nosso objectivo é abrir-lhes o apetite para o assunto, mostrar-lhes a sua utilidade, e não assustá-los com estranhos formalismos cheios de letras gregas e outros símbolos ilegíveis. Por isso preferimos técnicas de descrição formal capazes de produzir especificações executáveis, para podermos usar prototipagem. Ora isto pode ser feito de duas formas distintas: escolher uma linguagem de especificação e utilizá-la, ou então, ensinar uma linguagem de especificação, mas integrada num processo, que efectivamente guie o desenvolvimento. Nós preferimos a última alternativa, até porque estamos interessados em mostrar a futuros engenheiros informáticos o valor de uma especificação formal. Note-se que o nosso objectivo é sermos capazes de especificar formalmente sistemas com complexidade apreciável, pelos menos como o da via verde, citado acima, e não ficarmos pela tradicional formalização do comportamento da pilha… Claro que nisto tudo vamos preferir continuar com a abordagem orientada pelos obejctos.

Nos últimos anos temos vindo a desenvolver um método, o método ROOA (*Rigorous Object-Oriented Analysis*) [Moreira 94, Moreira 96, Clark 98], que tanto pode utilizar LOTOS [Brinksma 88] como SDL'92 [Ellsberger 97] para produzir especificações executáveis a partir de um conjunto de requisitos informais. Escolhemos este método, na falta de outro, e preferimos o SDL, por ser orientado pelos objectos e por incorporar mecanismos para a estruturação de sistemas grandes e complexos, como os pacotes e os blocos.

Note-se que o SDL tem uma forma gráfica e uma textual, sendo a gráfica a preferida. Por isso podemos descrever as nossas especificações formais graficamente, o que é mais simples e mais rápido, mantendo a possibilidade de as executar.

5.3 Cadeiras complementares

Nesta altura, em que os alunos já têm os conhecimentos fundamentais sobre programação e análise de sistemas, precisamos de uma cadeira onde possamos dar aos alunos uma visão integrada e genérica de todo o processo de desenvolvimento, reunindo os conhecimentos obtidos nas cadeiras que temos vindo a descrever e também noutras cadeiras do curso. É essa a missão da cadeira de Engenharia de *Software*, que surge no oitavo semestre.

É interessante que os estudantes tenham oportunidade de experimentar ao longo do curso vários ambientes de desenvolvimento, desde o ciclo editar–compilar–testar–editar–… na linha de comando, até aos sistemas CASE estilo Obsydian. No entanto, os ambientes de desenvolvimento estão em evolução rapidíssima, e, portanto, se queremos que os estudantes conheçam alguns deles de maneira a que isso suavize o seu ingresso nas empresas, a aprendizagem deve ser feita no último ano da licenciatura. Assim, para o penúltimo semestre propomos uma cadeira de ambientes de desenvolvimento, na qual os estudantes experimentariam, em regime de quase auto-estudo, um *cocktail* de ferramentas dessas. Provavelmente, o elenco dos ambientes variaria de ano para ano, mas a título de exemplo, se fosse este ano teríamos, entre outros, o Visual Studio (Microsoft), o Delphi (Borland), o VisualAge (IBM). Nesta altura poderíamos usar as ferramentas do VisualAge em dois níveis diferentes: primeiro, juntamente com as técnicas de modelação, ainda no domínio do problema, para fazer prototipagem rápida; depois, no domínio da solução, para obter a aplicação desejada.

O último ano da licenciatura é, na prática, uma ano de transição da universidade para o mundo do trabalho. No último semestre os estudantes realizam um projecto, ou junto de professores do Departamento de Informática, ou em empresas, e nesse projecto colocarão em jogo as capacidades de desenvolvimento de *software* que adquiriram durante o curso. Se as coisas continuarem a evoluir como o têm feito recentemente, muitos desses projectos serão orientados pelos objectos, e utilizarão as ferramentas de desenvolvimento mais modernas.

6 Conclusão

Mais linguagens, mais técnicas de programação, mais metodologias de desenvolvimento, significam mais trabalho para os estudantes de engenharia informática. Daqui não há escapatória. A questão é saber se vamos redesenhar e reequilibrar os currículos das nossas licenciaturas para reflectir isso (assim arranjando mais trabalho também para os professores…), ou se vamos apenas encaixar os novos assuntos no quadro existente. Estamos convencidos de que o volume de matéria a tratar, tanto no domínio da programação e das linguagens como no dos métodos de desenvolvimento, exige um estudo mais prolongado. Para criar condições para isso, a nossa proposta inclui quatro cadeiras de programação, três de metodologias e ainda uma de síntese.

Em nossa opinião, a tecnologia dos objectos trouxe para o desenvolvimento de *software* um novo corpo de matérias que permitem não só uma abordagem cientificamente válida e pedagogicamente interessante, como também profissionalmente muito útil. Esse corpo de matérias cobre todo o ciclo de vida do software, e, curiosamente, constitui, segundo a nossa proposta, também uma fileira homogénea a ser seguida pelos estudantes ao longo de todo o seu ciclo universitário.

7 Referências

[Booch 86] Booch, G.: *Software Engineering with Ada*, 2ª edição, Benjamin-Cummings, Menlo-Park, 1986.

[Brinksma 88] Brinksma E. (ed).: *Information Processing Systems — Open Systems LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1988.
[Clark 98] Clark, R. e Moreira, A.: "Formal Specification of User Rquirements", Automated Software Engineering Journal, 1998, para aparecer.
[Coad 91] Coad, P. e Yourdon, E.: *Object-Oriented Analysis*, 2ª edição, Yourdon Press, Prentice-Hall, 1991.
[Constantine 89a] Constantine, L.: "Beyond the madness of Methods: Systems Structure Modeling and Convergent Design", em actas da Software Development'89, Miller-Freeman Publishing Co., 1989.
[Constantine 89b] Constantine, L.: "Object-Oriented and Structured Methods: Towards Integration", American Programmer, 2(7/8):34-40, Agosto 1989.
[Dahl 72] Dahl, O.-J., Dijkstra, E.W., e Hoare, C.A.R.: *Structured Programming*, Academic Press, Londres, 1972.
[DeMarco 78] DeMarco, T.: *Structured Analysis and System Specification*, Yourdon Press, 1978.
[Dijkstra 76] Dijkstra, E.W.: *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, 1976.
[Downs 88] Downs, E, Coad, P. e Coe I.: *Structured Systems Analysis and Design Method — Application and Context*, Prentice-Hall, 1988.
[Ellsberger 97] Ellsberger, J., Hogrefe, D. e Sarma: A.: *SDL: Formal Object-Oriented Language for Communication Systems*, Prentice Hall, 1997.
[EVB 89] EVB sofwtare Engineering: "Object-Oriented requirements Analysis", transparentes do curso, 1989.
[Firesmith 91] Firesmith, D.G.: "Structured Analysis and Object-Oriented Design are not Compatible", Ada Letters, 11(9):56-66, Novembro/Dezembro, 1991.
[Gane 79] Gane, C. e Sarson, T.: *Structured System Analysis: Tools and Techniques*, Prentice Hall, 1979.
[Gries 81] Gries, D.: *The Science of Programming*, Springer-Verlag, Berlin-New York, 1981.
[Howe 89] Howe, D.R.: *Data Analysis for Data Base Design*, Edward Arnold, 2ª edição, 1989.
[Jacobson 92] Jacobson, I.: *Object-Oriented Software Engineering — a use case driven approach* , Addison-Wesley, Reading Massachusetts, 1992.
[McCracken 65] McCracken, D.D.: *A Guide to FORTRAN IV Programming*, John Wiley & Sons, New York, 1965.
[Meyer 97] Meyer, B.: *Object-Oriented Software Construction*, 2ª edição, Prentice Hall, Upper Saddle River, 1997.
[Moreira 94] Moreira, A.: "Rigorous Object-Oriented Analysis", PhD Thesis, Universidade de Stirling, Escócia, 1994.
[Moreira 96] Moreira, A. e Clark, R.G.: "Adding Rigour to Object-Oriented Analysis", Software Engineering Journal, 11 (5), 270-280, 1996.
[Oliveira 1998] Oliveida, G., Guerreiro, P. e Moreira, A.: "Adaptability of CASE Tools for Object-Oriented Software Development", World Multiconference on Systemics, Cybernetics and Informatics (SCI'98) e 4th International Conference on Information Systems Analysis and Synthesis (ISAS'98), Orlando, USA, 1998. Para aparecer.
[Page-Jones 80] Page-Jones, M.: *The Practical Guide to Structured Systems Design*, McGraw-Hill, 1980.
[Rational 97] Rational: Unified Modeling Language, http://www.rational.com, 1997.
[Ross 75] Ross, D.T., Goodenough, J.B. e Irvine, C.A.: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
[Rumbaugh 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. e Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
[Rubin 92] Rubin, K.S. e Goldberg, A.: "Software Engineering: process Principles and Goals", Computer, May, 1975.
[Shumate 91] Shumate, K.: "Structured Anlysis and Object-Oriented Design are Compatible", Ada Letters, 11(4):78-90, Maio/Junho, 1991.
[Tkach 96] Tkach, D, Fang, W. e So, A.: *Visual Modeling Technique — Object Tecnology Using Visual Programming*, Addison-Wesley, 1996.
[Ward 89] Ward, P.: "How to Integrate Object-Orientation with Structured Analysis and Design", IEEE Sofwtare, 6(2):74-82, Março 1989.
[Wirth 76] Wirth, N.: *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, 1976.
[Wirth 92] Wirth, N. e Reiser, M.: *Programming in Oberon — Steps Beyond Pascal and Modula*, Addison-Wesley, Reading, 1992.
[Yourdon 89] Yourdon, E.: *Modern Structured Analysis*, Prentice-Hall, 1989.
[Yourdon 79] Yourdon, E. e Constantine, L.: *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, Prentice-Hall, 1979.