Desenvolvimento de Software Orientado a Objetos

Prof. Nicolas Anquetil

Capítulo 1

Desenvolvimento de software orientado a objetos

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Esse curso tem por objetivo apresentar a modelagem de sistemas orientados a objetos utilizando UML. O curso pretende fazer uma pequena introdução ao modelo orientado a objetos também.

1.1 Referências

O curso é baseado sobre os seguintes livros:

- Martin Fowler, Kendall Scott, *Uml Essencial : Um Breve Guia Para a Linguagem Padrão de Modelagem de Objetos*, Bookman.
- Grady Booch, Ivar Jacobson, James Rumbaugh, *UML Guia do Usuário*, Editora Campus,1999.
- Martin Fowler, Kendall Scott, *UML Distilled – Applying the Standard Object Modeling Language*, Addison Wesley, 1997.
- Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1998.
- Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Developement Process*, Addison Wesley, 1998.
- Geri Schneider, Jason P. Winters, *Applying Use Cases – A Practical Guide*, Addison Wesley, 1998.

Tem também muitas informações sobre a UML na internet:

- Minha página.

- A página da Rational, a empresa onde os autores da UML trabalham: http://www.rational.com.
- Alguns modelos de documentos do processo de desenvolvimento do livro "Applying Use Cases": http://books.txt.com.
- A página do OMG (Object Modeling Glossary): http://uml.systemhouse.mci.com/ .

1.2 Concepção de um sistema

Sistemas de computação existem para resolver alguns problemas:

- Gestão de uma universidade (professores, aulas, salas, alunos, etc.)
- Criação de um editor de textos.
- Controle de tráfego aéreo.
- Gestão de reservas para um show.
- etc.

Conceber um pequeno programa é fácil, mas grande sistemas tem várias dificuldades:

- Desenvolvidos por muitas pessoas (dezenas ou mais);
- Pessoas entram e saiam do projeto;
- O sistema é grande demais para ser entendido por uma pessoa só.

Isso resulte em problemas de comunicação entre todos os participantes do projeto, de coordenação deles, de planificação dos riscos, etc.

Para desenvolver esses sistemas, é necessário ter um bom método de trabalho. Tais métodos são projetados por pessoas com muito experiência e permitem evitar erros que poderiam atrasar o projeto ou mesmo o fazer falhar (se estima que mais de 60% dos projetos de desenvolvimento de software são entregados com atraso ou não implementam o que era previsto no início).

Esses métodos são compostos de várias fases, que são tip[icamente:

Levantamento dos requisitos: Qual é o problema a ser resolvido ? Quais são as limitações (tempo, dinheiro, máquinas, etc.) ?

Análise: Início da definição informática, mas sem pensar numa implementação precisa (qual será a linguagem usada, o banco de dados, etc.)

Projeto: Definição detalhada da futura implementação.

Implementação: Implementação na realidade.

Teste: Teste se o sistema implementado faz o que era preciso e sem erros.

As fases mais abstratas (as primeiras) são as mais importantes e as mais difíceis. Um erro nessas etapas pode ter conseqüências muito importantes depois.

> **Nota** : As coisas não são tão bem separadas. Na realidade, se torna difícil de marcar exatamente o limite entre qualquer das etapas acima. O processo de desenvolvimento também não é tão simples, se precisa sempre retornos para corrigir alguns erros. O importante é descobrir os erros o mais rápido possível para não perder tempo demais refazendo as coisas erradas.

Nesse curso vamos estudar o RUP, que é um dos métodos de desenvolvimento de software existentes. Este método se baseia fortemente sobre o uso de uma linguagem de modelização: a UML.

1.3 UML

UML significa Unified Modeling Language (linguagem de modelagem unificada). Ela deriva principalmente da unificação das linguagens de modelização de três métodos: o "OMT" de Rumbauch, o "método de Booch" e os "casos de uso" de Jacobson. Cada um foi bem sucedido como método de análise orientada a objeto, e o objetivo da UML é tirar vantagem das principais caraterísticas de cada um deles.

Como todas as linguagens, a linguagem UML é usada para transmitir a outros, e receber de outros, algumas informações. Nesse caso as informações são a definição detalhada de como um sistema deveria ser implementado para realizar o que o usuário deseja.

É importante conhecer a linguagem para entender os modelos que outras pessoas desenvolveram com muito esforço ou para explicar a outras pessoas (por exemplo programadores) como um sistema deve ser implementado. Cada conceito a ser modelado (sub-sistema, classe, relações, etc.) tem uma representação gráfica específica na linguagem. É muito importante sempre respeitar as convenções definidas na UML para que outras pessoas possam entender os diagramas gerados.

A principio, a linguagem UML pode ser utilizada com qualquer processo de desenvolvimento de software. Mas ela se adequa muito bem ao processo RUP.

1.4 Modelo de objetos

A UML é baseada sobre o modelo a objetos. Esse modelo foi introduzido inicialmente como modelo de programação (Simula-67 e Smalltalk-80). Agora, ele é usado como modelo de banco de dados, de representação de conhecimento ou de modelagem de software.

O objetivo do curso não é de ensinar a programação objeto. Nós estudáremos apenas as bases do modelo de objetos. No entanto, essas bases deveriam nos ajudar programar com qualquer linguagem orientada a objeto.

O modelo de objetos possui várias vantagens para o desenvolvimento de software (ver capítulo sobre o modelo a objeto), inclusive:

- O mesmo modelo pode ser usado desde o início enquanto os conceitos são muito abstratos, até a implementação.

- O modelo permite dar mais importância à estrutura de um sistema do que às funções. Isso é importante porque as funções que um sistema deve realizar mudam com o tempo (sempre, mesmo durante o desenvolvimento do sistema) enquanto são muito raras as mudanças da estrutura (uma universidade sempre terá professores, alunos e cursos).

Capítulo 2

O modelo a objetos

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Antes de estudar UML, vamos primeiro (re)ver as noções básicas do modelo a objetos. Neste capítulo, vamos definir as seguintes noções: Objeto, classe, instância, modelo, propriedade, atributo, operação, método, mensagem, herança, hierarquia de herança, polimorfismo e interface [1].

Vamos ver também as vantagens que possui o modelo a objetos sobre outros modelos como o modelo "procedural" (ou funcional). Essas vantagens são muito importantes quando se tratar de fazer a modelização de um software. É por isso que a UML é baseada no modelo a objetos.

Este capítulo é apenas uma pequena introdução ao modelo a objetos. Vamos aproveitar os próximos capítulos sobre a UML para aprofundar também nosso conhecimento do modelo a objetos.

2.1 Introdução

Vamos primeiro estudar um exemplo simples para introduzir algumas noções básicas.

Em linguagens como C ou Pascal, representamos conceitos complexos com estruturas. Por exemplo, um endereço pode ser representado assim em C:

```
typedef struct {
  char nome[50];
  int  numeroRua;
  char rua[100];
  char cidade[50];
  int  CEP;
  char estado[2];
} Endereco;
```

[1] Atenção, no modelo a objetos, a palavra "interface" pode ter vários sentidos levemente diferentes, inclusive: *interface* de uma classe e *interface* como um tipo especial de classe. Vamos ver o primeiro sentido neste capítulo e o segundo num próximo copítulo

Esta estrutura pode ser manipulada por funções especiais:

```
Endereco *criarEndereco(char *n,int nr,char *r,char *c,int cep,char *e);
int compararEndereco(Endereco *e1, Endereco *e2);
void lerEndereco(Endereco *e);
void imprimirEndereco(Endereco *e);
...
```

A estrutura implementa um conceito do mundo real na linguagem de programação. "Nome", "numeroRua" são *atributos* do conceito. As funções implementam algumas *operações* que podem ser efetuadas sobre o conceito.

As coisas são um pouco diferentes no modelo a objetos. Nesse modelo, a representação do conceito (a estrutura) e as operações são reunidas em apenas uma construção, chamada de *classe*. Por exemplo em C++, o mesmo conceito com as mesmas operações poderia ser implementado assim:

```
class Endereco {
  char nome[50];
  int  numeroRua;
  char rua[100];
  char cidade[50];
  int  CEP;
  char estado[2];

  Endereco *criarEndereco(char *n,int nr,char *r,char *c,int cep,char *e);
  int compararEndereco(Endereco *e);
  void lerEndereco();
  void imprimirEndereco();
  ...
};
```

> **Nota** : Os parâmetros das funções mudaram um pouco. Vamos ver o porquê, depois de introduzir os objetos.

> **Nota** : Em C, é preciso acrescentar aos nomes das funções (criar, comparar, ...) o nome da estrutura (Endereco) para diferenciar de outras funções que trabalham com outras estruturas (ex.: criarCliente, compararCliente, ...). Com o modelo a objetos, as funções são definidas dentro da classe "Endereco" e não podem ser confundidas com outras funções. Assim seria melhor (mais simples) tirar o "Endereco" dos nomes das funções.

Uma primeira vantagem da classe é a proximidade física dos dados e das funções que pertencem ao mesmo conceito. Porque todas as partes do conceito são agrupadas no mesmo lugar, se torna mais fácil de modificar a implementação (ex.: acrescentar um atributo) sem esquecer nenhuma parte (ex.: acrescentar um parâmetro à função "criar" e modificar a função "imprimir").

Nas linguagens procedurais, a estrutura é um tipo que pode ser usado para criar variáveis que vão ser usadas depois com parâmetros das funções já definidas. Essas funções vão executar algumas operações sobre as variáveis.

```
Endereco *criarEndereco(char *n,int nr,char *r,char *c,int cep,char *e) {
  Endereco *retorno;
  retorno = (Endereco*)malloc(sizeof(Endereco));
  retorno->nome = n;
  ...
  imprimirEndereco(retorno);
  return(retorno);
}
```

No modelo a objetos podemos fazer a mesma coisa, a classe é um tipo que pode ser usado para criar *objetos* (ou *instâncias*).

```
Endereco *criarEndereco(char *n,int nr,char *r,char *c,int cep,char *e) {
  Endereco *retorno;
  retorno = new(Endereco);
  retorno->nome = n;
  ...
  retorno->imprimir ();
  return(retorno);
}
```

Nota : O exemplo acima é para a linguagem C++. Outras linguagens, como Java, não usam apontadores explícitos: `Endereco retorno; retorno=new(Endereco); retorno.nome = n; ...`

Com instâncias, é possível usar as operações da classe. A instrução "`retorno->imprimir();`" significa: execute a operação (a função) "imprimir" para o objeto "retorno". Outra maneira de ver é que estamos pedindo ao objeto "retorno" de executar ele mesmo a operação "imprimir", de se imprimir ele mesmo. Essa característica é chamada de mandar uma *mensagem* ao objeto. Mandamos ao objeto a mensagem: "se imprima". Já que o objeto é do tipo (da classe) "Endereco", ele sabe como fazer para se imprimir.

Podem reparar que tiramos o primeiro parâmetro (de tipo Endereco) das operações, no modelo a objetos, todas as operações são chamadas com um parâmetro por default: o objeto ao qual a mensagem está sendo mandada.

A vantagem do modelo a objetos é que cada objeto "sabe" quais funções ele deve executar. Na programação procedural, quando tivermos várias estruturas, cada uma com uma função imprimir, é preciso saber qual função chamar com qual estrutura. No modelo a objetos, basta mandar ao objeto a mensagem desejada.

Isso concorda mais com o entendimento comum: tem uma operação abstrata "impressão" e sabemos o que imprimir um objeto quer dizer. Mas, essa operação abstrata tem que ser

implementada de maneira diferente para cada classe. Esse fato é uma limitação da programação, o modelo a objetos permite fazer abstração desse fato e só considerar a operação abstrata. Isso é chamado de *polimorfismo*.

Notem que na implementação, tem poucas diferenças entre o modelo a objetos e o modelo procedural. O primeiro reune a estrutura e as operações na mesma unidade sintática (a classe), o segundo as guardam separadas. Mas esta pequena diferença na implementação (junto com alguns outros mecanismos simples) fazem uma grande diferença na concepção de um sistema porque as coisas se tornam mais simples. No modelo procedural, as funções agem sobre os seus próprios parâmetros. Quando usamos uma função, temos que pensar como ela age sobre os parâmetros, e relacionar a função correta com os parâmetros corretos. A função e os parâmetros que passamos para ela são duas coisas separadas. No modelo a objetos, podemos pensar que os próprios objetos executam as funções e que eles mesmos se modificam. Assim, a utilização se torna mais fácil, é só pensar no que queremos que o objeto faça e "pedir" a ele para fazer.

O modelo a objetos possui uma outra vantagem. Podemos usar as classes já definidas para definir novas classes mais facilmente. Se queremos implementar agora o conceito "EnderecoComTelefone", no modelo procedural, temos que criar uma nova estrutura com os mesmos atributos mais o número de telefone, e temos também que recriar todas as funções com parâmetro desse novo tipo. No modelo a objetos podemos tirar vantagem de que o conceito EnderecoComTelefone é um sub-conceito de Endereco que já conhecemos. Este novo conceito pode ser implementado com uma *sub-classe* de Endereco:

```
classe EnderecoComTelefone: Endereco
{
  int numeroTelefone;
}
```

Isso se chama *herança*: a (sub)-classe EnderecoComTelefone herda da (super-)classe Endereco. Podemos também dizer que EnderecoComTelefone é uma *especialização* de Endereco (e Endereco é uma *generalização* de EnderecoComTelefone).

Uma especialização de um conceito, possui as mesmas propriedades que o super-conceito (uma bananeira possui todas as propriedades de uma árvore em geral) mais algumas propriedades particulares. O modelo a objetos implementa as mesmas noções: uma sub-classe possui, por default, todos os atributos e todas as operações da sua super-classe. A sub-classe funciona como se a definição da super-classe estivesse copiada integralmente nela.

Mas:

- A sub-classe pode acrescentar novos atributos ou novas operações às da super-classe. Por exemplo, EnderecoComTelefone acrescenta o atributo numeroTelefone.
- A sub-classe pode redefinir algumas operações da super-classe. Por exemplo, EnderecoComTelefone pode redefinir a operação imprimir, para imprimir também o numero de telefone.

A herança combinada com o polimorfismo são ferramentas muito poderosas. Uma instância de uma sub-classe pode ser usada em qualquer momento como uma instância da super-classe. Isso significa que podemos fazer o seguinte:

```
void main
{
  Endereco *E;
  E = new(EnderecoComTelefone);
  E->imprimir();
}
```

Por que EnderecoComTelefone é uma sub-classe de Endereco, podemos usar as instâncias da primeira como se fossem instâncias da segunda. Objetos que são "endereços com telefone" são também "endereços" simples. E graças ao polimorfismo, a instrução "`E->imprimir();`" vai chamar a função adequada para o objeto E, seja:

- a função "imprimir" da super-classe, caso EnderecoComTelefone não redefinir esta operação;
- a função "imprimir" da sub-classe, caso EnderecoComTelefone redefinir ela.

Isso facilita a definição de biblioteca. Uma biblioteca define (e usa) algumas classes (por exemplo, janelas para construir interfaces), e o usuário pode definir classes mais específicas (sub-classes) com atributos ou métodos específicos (por exemplo janelas redondas). Estas sub-classes podem funcionar como classes da própria biblioteca e serem utilizadas por ela.

> **Nota** : O inverso não é possível, não se pode colocar uma instância de Endereco dentro de uma variável para um EnderecoComTelefone, por que um endereço simples não é um "endereço com telefone". A classe EnderecoComTelefone poderia ter algumas operações que não são definidas para Endereco (simples).

2.2 Definições

Vamos agora definir mais formalmente as principais noções do modelo a objetos.

Classe: A modelagem de um conceito do mundo real. As propriedades associadas ao conceito são representadas por atributos (variáveis) e operações (i.e. funções). Uma classe descreve os atributos que seus objetos vão ter e as funções que podem executar.

Por exemplo, no domínio universitário os conceitos de professores, aulas, alunos, poderiam ser representados por classes.

Objeto: A modelagem de um objeto do mundo real. Cada objeto do mundo real (no domínio universitário, um professor particular, uma aula especifica) pode ser representado por um objeto.

Cada objeto "pertence" a uma classe. A estrutura do objeto como as operações que ele pode executar são descritas pela classe.

Objetos são entidades independentes uns dos outros. Dois objetos com exatamente os mesmos atributos são dois objetos diferentes. Na informática, se pode pensar na memória de um computador: dois objetos com os mesmos atributos, estão em diferentes partes da memória, na verdade eles tem um atributo implícito (o endereço na memória onde eles ficam) que é diferente.

Instância: Um objeto. A noção de instância é quase igual à de objeto. Instância se refere implicitamente a uma classe, é sempre uma instância de uma classe específica. Objetos podem ser considerados independentemente das suas classes.

A noção de instância é também mais geral. Qualquer conceito abstrato pode ter instâncias (aplicações concretas do conceito). Por exemplo, um método pode ser considerado uma instância de uma operação; um atributo particular de uma classe pode ser considerado uma instância do conceito geral de atributos, etc.

Propriedade: Um conceito é definido por um conjunto de *propriedades*. Por exemplo, uma pedra tem as seguintes propriedades: dura, inanimada, sólida, e muito mais.

Uma classe modela um conceito listando algumas das suas propriedades interessantes, considerando o problema a resolver. Por exemplo, a data de nascimento de um aluno vai ser uma propriedade interessante. A cor dos cabelos é também uma propriedade do conceito aluno, mas provavelmente, não é importante no modelo e não será modelada.

Atributo: Uma propriedade importante de uma classe que pode ser representada com uma variável. Por exemplo, a data de nascimento de um aluno.

Método: Uma propriedade importante de uma classe que pode ser representada com uma função. Por exemplo, a idade de um aluno pode ser calculada a partir da data de nascimento e da data do dia corrente.

Operação: Um método mais abstrato. O método se refere implicitamente a uma função. A operação é mais abstrata, ela se refere à idéia do que a função faz, mas não se preocupa com os detalhes de implementação. Um método implementa uma operação.

Mensagem: Se manda uma mensagem a um objeto para pedir a ele executar uma operação particular. Novamente, isso é uma noção muito parecida ao método, mas aqui, se presta mais atenção ao aspecto dinâmico: pedimos a um objeto que execute uma operação mandando uma mensagem a ele.

Herança: A modelagem da noção de especialização/generalização. No mundo real, conceitos são especializações uns dos outros: um professor visitante é um caso especial de professor; um professor, um aluno são seres humanos; um ser humano é um animal, ... Os conceitos mais especiais têm todas as propriedades dos conceitos mais gerais, mais algumas propriedades em si próprio.

No modelo a objetos, uma sub-classe possui todos os atributos e todas as operações da super-classe. A sub-classe pode acrescentar alguns atributos e métodos. Ela pode também redefinir alguns métodos da super-classe (dentro de alguns limites que estudaremos). Ela não pode tirar nenhuma propriedade da super-classe.

Hierarquia de herança: Todas as relações de herança entre todas as classes formam uma árvore chamada de hierarquia de herança.

Polimorfismo: Várias classes podem implementar a mesma operação de maneira diferente. A mesma operação (com o mesmo nome) tem várias ("poli") formas ("morfismo") em

cada classe. Uma característica poderosa do modelo a objetos é que todas as formas de uma operação podem ter o mesmo nome. Assim se torna mais fácil de reconhecer qual operação um método particular implementa. Isso é possível porque cada objeto sabe qual é sua própria classe, e pode executar os métodos apropriados.

Interface: Conjunto das propriedades da classe que pertencem ao conceito implementado pela classe.

Algumas operações são complicadas demais para serem implementadas com uma função só. É preciso criar pequenas funções utilitárias que vão simplificar a implementação das operações. Mas essas pequenas funções utilitárias não pertencem ao conceito que a classe implementa. Elas são apenas detalhes de implementação.

A interface especifica quais funções podem ser chamadas por outros objetos e quais não podem. As funcões que não pertencem à interface são funções utilitarias que só podem ser chamadas pelo próprio objeto.

Suponhemos que as propriedades do conceito (propriedades da interface) não vão mudar enquanto os detalhes de implementação podem mudar. A interface permite proteger as o exterior das mudanças dentro da classe. Enquanto a interface não muda, para o exterior, a classe não muda.

Interface tem também um outro sentido um pouco diferente deste, que estudaremos mais tarde.

2.3 Vantagens do modelo a objetos

O modelo a objetos tem várias vantagens para a concepção e a programação de software. Já vimos informalmente a maioria delas.

Abstração: As noções de operação e interface permitem conceber classes de maneira abstrata sem se preocupar em como serão implementadas. Elas diminuem a quantidade de coisas em que temos que pensar (uma operação para vários métodos), o que simplifica a concepção.

Também, graças ao alto nível de abstração que pode conseguir, o modelo a objetos pode ser utilizado desde o início da concepção (análise) até a programação. Assim, se torna mais fácil relacionar qualquer parte do programa com os conceitos mais abstratos que ele implementa ("rastreamento"). Se pode relacionar a implementação com as decisões de projeto. Isso é muito importante para a boa manutenção do programa, tanto durante, quanto após o seu desenvolvimento.

Encapsulamento: Graças à interface, o exterior da classe é protegido das modificações que podem acontecer na implementação da classe (interior). Enquanto a interface de uma classe não mudar, a classe não muda para o exterior. Ao mesmo tempo, a implementação pode mudar muito (métodos se tornarem atributos, implementação dos métodos mudando).

Isso já era possível com modelos "não objetos". Mas como as funções e a estrutura são agrupadas, se torna mais fácil respeitar a mesma interface enquanto se muda a implementação. (Existem também "receitas" para aproveitar mais essa qualidade como por exemplo nunca acessar direitamente os atributos de uma classe, mas sempre usar métodos "get" e "set".)

Agrupamento dados/funções: Essa característica facilita o polimorfismo. Como uma classe contem ambos dados e funções, uma operação pode ter várias implementações em várias classes. Cada implementação é diferente das outras, mas todas pertencem à mesma operação.

Reutilização: O modelo a objetos permite duas formas de reutilização:

- A herança permite definir uma nova classe, reutilizando outras classes já definidas. Esta reutilização facilita a definição da nova classe, basta se concentrar no que a nova classe acrescenta.
- Cada classe é um pequeno módulo que contem dados e funções. Desta maneira, este pequeno módulo pode ser reutilizado em outros sistemas.

Estrutura e função: O modelo a objeto permite dar mais importância à estrutura de um sistema do que às funções. Isso é importante porque as funções que um sistema deve realizar sempre mudam com o tempo enquanto são muito raras as mudanças da estrutura.

2.4 Exercícios

Listem algumas classes com atributos e operações para modelar:

- Um jogo de carta
- Um sistema de reserva para uma empresa aérea
- Um sistema de faturamento para uma empresa

(Procurem utilizar a herança quando possível.)

Capítulo 3

Diagrama de classe (básico)

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos iniciar o estudo da UML/linguagem de representação de modelos de objetos. Já vimos que UML define também um processo para criar modelos, esta parte será estudada mais tarde num outro capítulo. Neste capítulo, vamos primeiro introduzir a UML como linguagem. Um modelo UML é composto de vários diagramas, vamos ver rapidamente quais. Vamos iniciar também o estudo destes diagramas com um dos mais importantes deles: o diagrama de classe que descreve as classes necessárias e as relações entre elas.

Finalmente, vamos aproveitar para melhorar nosso conhecimento do modelo a objetos e estudar algumas outras noções importantes deste.

Este capítulo é baseado nos capítulos 2, 4 e 5 do "UML User Guide" (c.f. referência no capítulo "Introdução" do curso).

3.1 Introdução

UML significa "Unified Modeling Language" (linguagem de modelagem unificada). É uma linguagem unificada por que deriva de três outros métodos (cf. introdução ao disciplina). Em parte por causa dessa unificação, a UML é muito rica, tanto na "largura" (se pode modelar um grande variedade de sistemas como sistemas de tratamento de dados, sistemas de tempo real ou sistemas distribuídos), como na "profundidade" (se pode modelar muitos detalhes, cada noção tem variações). Isso pode se tornar também um problema porque a UML é complexa a aprender. Nesse disciplina, vamos tentar concentrar nas partes mais genéricas do modelo, por exemplo, não veremos tudo o que trata de tempo real ou de sistemas distribuídos.

A UML se compõe de 9 diagramas:

Classe Mostra as classes que compõem o sistema e as relações entre elas (por exemplo a herança). Trata de um aspecto estático e estrutural do sistema.

Objeto Mostra alguns objetos (instâncias das classes) e as relações entre eles (qual objeto cria ou usa quais outros). É muito parecido ao primeiro (objetos são instâncias das classes), mas é um pouco mais aplicado. As classes definem a estrutura abstrata do

sistema, os objetos já supõem que um programa está sendo executado e que instâncias são criadas para enviar e receber mensagens.

Caso de uso Mostra como o sistema vai interagir com os usuários (pessoas ou outros sistemas).

Seqüência Mostra interações entre vários componentes do sistema.

Colaboração Mostra, também, interações entre vários componentes do sistema. É muito parecido com o precedente e os dois serão estudados juntos.

Atividade Mostra como um sub-sistema ou um objeto realizam uma operação. Também conhecido como diagrama de fluxo de dados.

Estados Mostra como um sub-sistema ou um objeto realizam uma operação. Parecido com o precedente mas se concentra mais sobre eventos e ações que eles produzem.

Componentes Mostra a organização dos componentes. Trata da implementação do sistema.

Implantação Mostra a configuração física ("hardware") sobre qual o sistema será instalado.

Vamos estudar o diagrama de classe e brevemente o de objetos. O diagrama de classe pode ser considerado um dos dois mais importantes. Vamos estudar também o diagrama de casos de uso. Este seria o segundo diagrama mais importante, ele será estudado quando vermos o processo de análise da UML. Os outros diagramas (seqüência, colaboração, atividade, estados, componentes e implantação) serão estudados mais rapidamente.

Os diagramas de classes e de objetos são os dois diagramas que definem a estrutura do sistema, por isso eles são bastante importantes. Outros diagramas (uso de caso, seqüência, colaboração, atividade e estados) definem o comportamento do sistema: o que ele deve fazer e como ele vai fazer. Finalmente os diagramas de componentes e implantação são diagramas de arquitetura, eles são úteis principalmente para os sistemas muito grandes (por exemplo sistemas distribuidos sobre vários computadores).

3.2 Diagrama de classe

Este diagrama lista todos os conceitos do domínio que serão implementados no sistema. Por exemplo, no domínio de reservas de assentos, os conceitos importantes seriam: lugar, data, e cliente; no domínio de edição de textos, os conceitos importantes seriam: caracteres, linhas, paginas, palavras, fonte de caracteres, etc. O diagrama lista também as relações entre os conceitos: um cliente "reserva" x lugar(es) para a data y. Em geral, conceitos são substantivos e relações são verbos.

O diagrama de classe é muito importante porque define a estrutura do sistema a desenvolver. A maioria dos outros diagramas vão usar as classes definidas no diagrama de classe. Por exemplo, o diagrama de colaboração, mostra as interações entre classes do sistema.

3.3 Classes e Objetos

Podemos buscar os conceitos importantes do domínio (na universidade tem professores e alunos) ou os objetos importantes (a sala de show número 1 tem 250 lugares). A partir dos objetos, é fácil descobrir as classes.

Na UML o nome de uma classe é um texto contendo letras e dígitos e algumas marcas de pontuação. Na realidade, é melhor guardar os nomes curtos com apenas letras e dígitos. UML sugere capitalizar todas as primeiras letras de cada palavra no nome (ex.: "Lugar", "DataReserva"). É melhor também manter nomes de classe no singular, classes por default "contem" mais de um objeto, o plural é implícito.

Em geral, se deve prestar muito atenção aos nomes em modelos. Isso se verifica para as classes como para os objetos. Por exemplo, na edição de texto, dissemos que "palavra" é um conceito importante, linhas são compostas de palavras, mas linhas podem também conter números, seja palavra e número o mesmo conceito ou não?

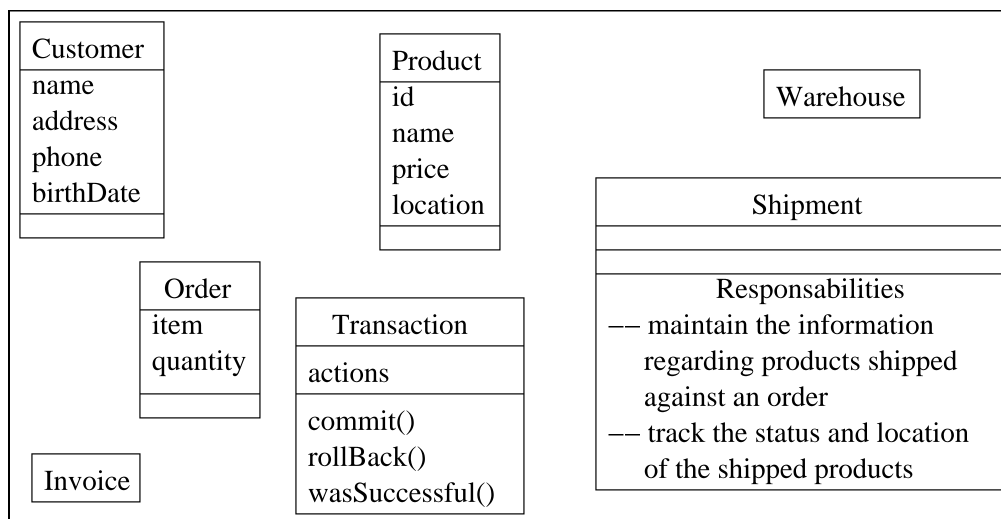

Figura 3.1: Representação das classes em UML (copiado de "*UML User Guide*", p. 52).

Normalmente, classes são representadas por caixas com três compartimentos: nome da classe, atributos e operações. Por exemplo, na figura 3.1, a classe "Transaction" tem um atributo (actions) e três operações.

A representação de uma classe num modelo pode ser mais ou menos abstrata. Por exemplo, podemos "esquecer" algumas propriedades da classe (ex.: a classe "Order" não tem operações). Podemos até apresentar unicamente o nome da classe, sem precisar as propriedades dela (ex.: classe "Invoice" na figura). Isso não significa que a classe não tem propriedades, mas apenas que não queremos mostrar essas propriedades no modelo. Estamos apresentando uma abstração da classe, a mesma classe num outro modelo poderia ser apresentada com propriedades completamente diferentes.

Quando uma classe tiver propriedades que não queremos apresentar num modelo, UML sugere acrescentar três pontos (...) ao fim das propriedades apresentadas para indicar que existem mais.

Ao contrário, podemos especificar mais a classe e mostrar os tipos dos atributos e os parâmetros das operações (ex.: classe "Product").

Num modelo é importante apresentar toda a informação necessária, senão o modelo fica impreciso, mas somente a informação necessária, senão o modelo fica confuso.

Os objetos são parecidos às classes (c.f. a figura 3.2). Um objeto pode ser anônimo (ex.: instância de "Product" na figura) ou teem r um nome (ex.: "ZéCarlos"). Se for importante, é possível especificar os valores dos atributos.

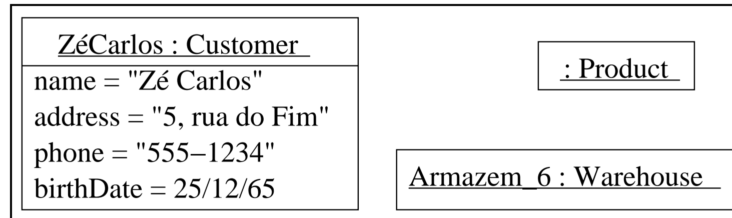

Figura 3.2: Representação dos objetos em UML

3.3.1 Responsabilidades

No modelo a objetos, classes têm propriedades (atributos e operações). Mas no início da concepção, todas as propriedades não são claras ainda, as classes têm apenas *responsabilidades*.

Uma responsabilidade é um contrato entre a classe e o resto do sistema. Ela especifica o que as instâncias da classe vão fazer. Essa idéia é muito parecida à idéia de operação, mas responsabilidades são mais abstratas. Responsabilidades são representadas por um texto curto (uma frase). Por exemplo, as responsabilidades de um aluno na universidade seriam de se registrar na universidade para um curso e fazer as disciplinas necessárias para este curso. A responsabilidade de uma sala seria de ter aulas nela.

Uma classe pode ter qualquer número de responsabilidades, mas na realidade, ela deveria ter pelo menos uma responsabilidade e somente algumas. Todas as responsabilidades deveriam ser repartidas com eqüidade entre as classes.

Responsabilidades são representadas dentro de um quarto compartimento depois das operações (c.f. "Shipment" na figura 3.1).

3.3.2 Atributos

Depois de identificar os objetos (ou conceitos) interessantes, temos que definir as propriedades deles que queremos representar. Essas propriedades podem ser dados (atributos) ou operações.

Um atributo representa uma propriedade que todos os objetos da classe têm (por exemplo, todas as mesas têm uma altura, número de pernas, posição na sala, etc.). Mas cada objeto terá valores particulares para seus atributos (algumas mesas são muito baixas, outras são altas, etc.). O valor de um atributo pode mudar com o tempo (ex.: posição da mesa na sala), a lista de atributos geralmente não muda.

Uma classe pode ter qualquer número de atributos. Na UML, o nome de um atributo é um texto contendo letras e dígitos e algumas marcas de pontuação. Na realidade, é melhor guardar os nomes curtos com apenas letras e dígitos. UML sugere de capitalizar todas as primeiras letras de cada palavra no nome menos a primeira palavra (ex.: "nome", "codigoPessoaFisica").

Num modelo, os atributos devem ser de um tipo simples (inteiro, texto, talvez data), não podem conter outros objetos. Por exemplo o nome de um aluno pode ser um atributo dele, mas o curso que ele faz não pode porque dizemos que cursos são classes.

Em bancos de dados, se dá geralmente um atributo "identificador" aos objetos (chave primaria). Já vimos que no modelo a objetos, cada objeto é único, não se precisa, e não se deve colocar, identificadores aos objetos se não existirem na realidade. Por exemplo, palavras num texto não têm identificador, então a classe "Palavra" não deveria ter um tal atributo, mas um aluno na universidade tem um número único (matricula), este número tem que ser um atributo da classe "Aluno". Na realidade, a chave primaria dos bancos de dados é uma das implementações possíveis da idéia do que cada objeto tem uma identidade própria. O modelo a objeto quer representar a realidade e não cria de maneira artificial identificadores. Esses identificadores serão criados mais tarde no momento que precisaremos implementar a noção de identidade única dos objetos.

Novamente, se deve escolher os nomes dos atributos com cuidado. Dentro de uma classe, todos os atributos devem ter nomes diferentes. Classes diferentes podem possuir atributos com mesmo nome. Contudo, num modelo, dois atributos com o mesmo nome (em duas classes diferentes) devem identificar a mesma coisa. Por exemplo, um professor e uma sala ambos podem ter um atributo "numero" por que ele significa a mesma coisa nas duas classes, mas a nota que um aluno consegue numa disciplina e um atributo para gravar anotações sobre uma sala (ex.: "luz não funciona nessa sala"), não podem ambos ser chamados "nota" porque as duas coisas são diferentes.

Na UML, o nome de um atributo é um texto contendo letras e dígitos e algumas marcas de pontuação. Na realidade, é melhor guardar os nomes curtos com apenas letras e dígitos. UML sugere de capitalizar todas as primeiras letras de cada palavra no nome menos a primeira palavra (ex.: "nome", "codigoPessoaFisica").

3.3.3 Operações

O segundo tipo de propriedade é as operações ou métodos. A palavra método se refere mais à implementação, operação é mais abstrata e designa um serviço que os objetos devem cumprir. Operações podem mudar os valores dos atributos do objeto que a efetua, ou não. Por exemplo podemos abrir, fechar, ou deslocar uma janela, ou podemos contratar um empregado, etc.

Uma classe pode ter qualquer número de operações.

Como os atributos, duas operações em duas classes podem ter o mesmo nome. Isso se chama de polimorfismo, a mesma operação (o que significa realmente, o mesmo serviço abstrato a ser efetuado) está implementada com dois métodos diferentes (versões concretas do serviço abstrato) nas duas classes.

Ao início, a operação vai ser definida de maneira muito abstrata.Quando afinar o modelo, precisaremos definir a *assinatura* da operação, o que significa: seu nome (já definimos isso),

número de parâmetros, tipos e ordem deles e eventualmente, seu tipo de retorno. Por exemplo, uma operação para copiar algum texto num editor de texto poderia ter a assinatura seguinte (usando a sintaxe do C++): booleano copiar(posição início-texto, posição fim-texto).

Todos os métodos que vão implementar a operação tem que respeitar exatamente a assinatura dela (mesmo nome, mesmo número de atributo, com os mesmo tipos e o mesmo ordem). Um método não pode acrescentar ou cortar um parâmetro. Isso seria um violação do polimorfismo. Para mandar a mensagem corretamente, teríamos que saber qual é a classe do objeto (cada classe tendo método com assinatura diferente). O que é possível, no caso de cortar um parâmetro, é simplesmente ignorá-lo na implementação.

Já vimos que a escolha dos nomes é muito importante num modelo. Isso é ainda mais importante nesse caso. Dois métodos com o mesmo nome deveriam efetivamente pertencer à mesma operação, o que significa que os dois métodos deveriam fazer a mesma coisa (de maneira abstrata, é claro que concretamente, as implementações vão ser diferentes). Por exemplo, uma classe pintor não poderia ter uma operação "copiar" (fazer cópias de pinturas famosas) porque o serviço fornecido seria muito diferente da operação copiar do editor de texto acima.

3.3.4 "Truques"

Para classes/objetos: Discutir com os usuários, os objetos são muito naturais para eles porque correspondem aos objetos reais que eles são acostumados a manipular.

Definam responsabilidade para cada classe, o que a classe vai fazer no sistema. As responsabilidades deveriam ser repartidas com eqüidade entre todas as classes.

Para conseguir isso: Identifique as classes que colaboram para cumprir algum trabalho (um pequeno sub-sistema). Considere esse sub-sistema como um todo. Se alguma classe tiver responsabilidades demais, procure dividir ela em várias classes menos abstratas (menores). Ao contrário, se algumas classes tiverem poucas responsabilidades (ou responsabilidades muito simples), procure juntar elas numa classe só.

Define para cada classe os atributos e as operações necessários para cumprir as responsabilidades da classe.

3.4 Relações

Os objetos tem relações entre eles: um professor *ministra* uma disciplina *para* alunos *numa* sala, um cliente *faz* uma reserva *de* alguns lugares *para* uma data, etc. Essas relações são representadas também no diagrama de classe.

A UML reconhece três tipos mais importantes de relações: dependência, associação e generalização (ou herança).

A dependência não é exatamente uma relação do modelo a objetos, não pode ser representada nesse modelo, mas ela é útil no desenvolvimento de software e por isso a UML inclui ela no diagrama de classe. As duas outras relações existem no modelo a objetos. Por outro lado, tanto a dependência, quanto a generalização, existem somente entre classes, enquanto a associação existe entre classes como entre objetos.

Exemplos das três relações são apresentados na figura 3.3, vamos explicar os detalhes de cada relação nas seções a seguir.

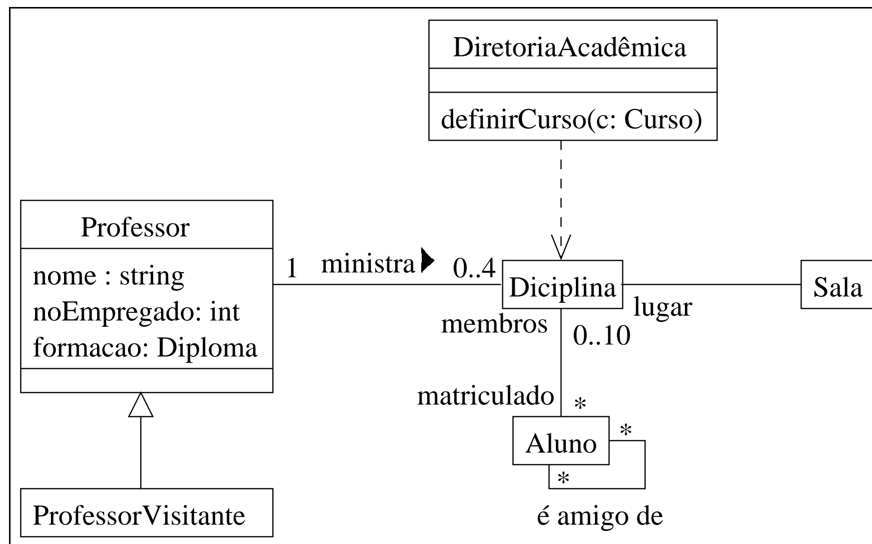

Figura 3.3: Representação das relações em UML

Existe uma dependência entre duas classes quando uma mudança na especificação de uma classe (a classe sobre qual se depende) pode ter conseqüências para a outra classe (aquela que depende). Dependência entre classes significa normalmente que a classe que depende faz alguma referência à outra classe, por exemplo como tipo de um parâmetro de uma operação dela. Se a estrutura da classe tipo do parâmetro mudar, é provável que o método vai mudar também. Isso é o significado que a relação de dependência quer trazer.

Uma relação de dependência é representada por uma seta tracejada da classe que depende para a classe sobre qual ela depende. Por exemplo, na figura, a DiretoriaAcademica depende de Disciplina.

3.4.1 Associação

A associação pode existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

Vimos que um atributo de uma classe normalmente deveria ser de um tipo simples. A razão é que se uma classe precisar de um atributo de tipo de uma outra classe (uma disciplina precisa de um atributo "ministradoPor" de tipo "Professor"), isso não se modela normalmente com um atributo, mas com uma associação (tem uma associação entre "Disciplina" e "Professor").

Uma associação tem dois "sentidos": (1) um professor ministra uma disciplina e (2) a disciplina é ministrado por um professor. É comum interessar-se por apenas um dos dois

sentidos, por exemplo num editor de textos, somos interessados em saber se um documento contem tal figura, mas talvez não em saber qual documento contem uma figura particular.

O problema de representar uma associação por atributos numa classe é que perdemos essa noção dos dois sentidos. Se a classe Professor contem um atributo de tipo Disciplina, não podemos adivinhar, olhando a classe Disciplina que ela tem uma associação com Professor. Mesmo quando estivermos interessados por apenas um dos dois sentidos de uma associação, é melhor representá-la no modelo com uma associação mesmo.

O diagrama de classe tentar modelar a realidade (a associação tem dois sentidos). Isso não impede que no momento da implementação podemos decidir nos aproveitar de somente um dos dois sentidos.

> **Nota** : Contudo é permitido ter um atributo de tipo não simples (o tipo é uma classe) numa classe por causa de abstração. Já vimos que é possível representar uma classe com apenas o nome dela, fazendo abstração dos atributos e das operações dela. Da mesma maneira, é possível representar uma classe fazendo abstração de outras classes com que ela é associada, por exemplo pelo fato de estas classes serem menos importantes. Neste caso, se pode sugerir as associações com atributos na classe principal.

Associações existem normalmente entre duas classes, tem alguns casos (raros) quando tiver uma associação entre mais de dois classes (associação n-ária).

É comum ter uma associação entre uma classe e ela mesma. Isso significa normalmente que a associação é entre duas instâncias diferentes da mesma classe. Por exemplo, a associação "pai de" existe entre duas instâncias da classe SerHumano. É uma associação da classe SerHumano com ela mesma.

Uma associação é representada por uma linha simples entre duas classes. A figura 3.3 apresenta várias associações.

Associações podem ter vários *adornos* (i.e. notações adicionais para precisar alguns aspectos):

Nome: É comum dar um nome a uma associação (um Professor "ministra" uma Disciplina). Este nome é geralmente um verbo. É permitido acrescentar uma seta ao nome para indicar o sentido dele (não é a Disciplina que ministra o Professor!)

Nomes se tornam úteis quando tiver várias associações entre as mesmas classes (ou entre a classe e ela mesma). Por exemplo, SerHumano pode ter várias associações com ele mesmo: "é pai de", "é mãe de", "é casado com", etc.

O nome da associação se coloca acima da linha que a representa.

Papel: Os dois pontos de uma associação têm um *papel* diferente (um SerHumano é o pai, o outro é o filho, o Professor ministra a Disciplina, a Disciplina é ministrada, etc.) Um papel pode ser comparado a um atributo.

Papéis são importantes quando a associação for entre uma classe e ela mesma. Eles permitem diferenciar os dois objetos a cada ponto da associação (quem é o pai e quem é o filho).

Se precisar nomes de papéis para uma associação, não é necessário dar um nome a ela (mas não é proibido também). É possível também dar só um dos dois nomes de papel a uma associação, para indicar que um sentido é mais importante que o outro.

Os nomes de papéis se colocam abaixo da associação, cada um do lado da classe que tem o papel (ex.: Sala tem o papel "lugar" na associação dela com Disciplina).

Multiplicidade: Um objeto pode ser associado a mais de um outro objeto. Por exemplo, um Cliente pode reservar 5 Lugares (ou mais) numa sala de show, mas um Lugar só pode ser reservado por um Cliente (senão, não é reserva!). Essas limitações são chamadas de *multiplicidade*. É possível indicar qualquer multiplicidade em um ponto de uma associação, mas algumas são mais comum: 0..1 (zero ou um), 1 (exatamente um), 1..* (um ou mais), * (zero ou mais). Multiplicidade pode ser até uma lista: 0..1,3,5,10..* (zero ou um ou três ou cinco ou 10 ou mais de 10).

Sejam prudentes com as multiplicidades. Por exemplo, qual é a multiplicidade da associação "é pai de"? Do lado do papel filho, a multiplicidade poderia ser *, uma pessoa pode ter qualquer número de filhos (na verdade tem uma limite, poderíamos escolher 0..20, mas há sempre exceções!). Do lado do papel pai, a multiplicidade parece evidente, é 2, todo o mundo tem dois pais. Mas na implementação, isso pode ser um problema. Se toda instância da classe SerHumano tem que ter dois pais, vamos ter uma sucessão infinita de objetos SerHumanos!

Multiplicidades são representadas em cada ponto da associação, acima dela.

3.4.2 Generalização

A relação de generalização também se chama de herança no modelo a objetos. Como a relação de dependência, ela existe só entre as classes. Um objeto particular não é um caso geral de um outro objeto, só conceitos (classes no modelo a objetos) são generalização de outros conceitos.

Esta é uma relação entre uma classe geral (a super-classe) e algumas classes mais específicas (as sub-classes), por exemplo entre um Professor geral e um ProfessorVisitante que é um tipo específico de professor. Uma sub-classe possui, por default, todos os atributos, todas as operações e todas as associações da super-classe dela. A sub-classe funciona como se a definição da super-classe estivesse copiada integralmente nela.

É comum a sub-classe acrescentar novos atributos ou novas operações aos da super-classe, ou redefinir uma operação com a mesma assinatura (mesmo nome, número de parâmetro, ...). Este segundo caso é uma aplicação do polimorfismo.

De forma geral, a sub-classe não deve cortar (ou ignorar) qualquer parte (atributo, método ou associação) da definição da super-classe. Se isso acontecer, pode indicar que a sub-classe não é realmente uma especialização da super-classe.

Um objeto da sub-classe pode ser utilizado em qualquer lugar onde se espera um objeto da super-classe. O contrário não é possível.

A generalização entre classes de um modelo deveria sempre ser usada quanto existe uma generalização semelhante no mundo real. Existe uma tentação muita forte de usar a

generalização como uma ferramenta de reutilização de código. Por que a definição da super-classe é "copiada" na sub-classe, a herança pode ser usada quando quisermos reutilizar alguns métodos complexos da super-classe.

Por exemplo, implementamos um método complexo para reservar uma sala para uma disciplina. O método verifica que não tem conflitos de horário com outras disciplinas, que a sala é suficiente grande para acomodar todos os alunos previstos, e que tem todas as facilidades para dar a disciplina (quadro, projetor, etc.)

Queremos agora implementar um método semelhante para reservar uma sala de show para um show. A solução mais fácil é criar SalaDeShow como sub-classe de Sala (de disciplina) e Show como sub-classe de Disciplina. Assim podemos reutilizar diretamente o método de reserva que vai verificar que não tem conflito de horário com outros shows, que a sala é suficiente grande para acomodar o público esperado, e que tem as facilidades para o show (luzes, som, etc.)

Isso é perfeitamente possível no modelo a objetos, mas é uma péssima idéia. Podemos considerar que Sala é uma classe geral para qualquer tipo de sala. Então SalaDeShow é realmente um tipo particular de Sala. Mas Show não pode herdar de Disciplina porque não respeitaria a realidade.

A utilização da herança nessas condições complica muito o trabalho de manutenção dos programas e deve ser proibida.

Em tal caso, tem duas soluções:

- Abstração da parte comum às classes Disciplina e Show e criação de uma super-classe para elas. A nova super-classe conterá o(s) método(s) a serem reutilizado(s) e as duas classes herdarão esses métodos.

- Delegação do trabalho, a classe Show define um método reserva que chama o método complexo da outra classe com os mesmo atributos que ele próprio recebeu.

A escolha da melhor solução depende das condições. Aqui a primeira solução parece a melhor. Mas não é sempre possível usá-la, por exemplo, quando o método complexo é definido numa biblioteca que não podemos mudar.

No mundo real, cada conceito é especialização de vários outros conceitos (um professor é também, um pai, um marido, um condutor de carro, um cliente de lojas, etc.). Isso se chama de herança *múltipla*. Em informática, a herança múltipla apresenta alguns problemas. Por exemplo, a classe Professor tem um atributo telefone que é o número de telefone do professor na universidade para os alunos puderem ligar para ele. A classe Cliente tem também um atributo telefone para a loja ligar para o cliente em casa em caso de problema. Se quisermos criar uma sub-classe de Professor e Cliente, temos um problema, as definições das duas super-classes vão ser "copiadas" na sub-classe, mas ela só pode ter um atributo telefone. Ainda não tem solução satisfatória para esse problema. Ao invés disso, várias linguagens, como Smalltalk ou Java por exemplo, utilizam a herança simples em qual uma classe só pode ter uma super-classe.

3.5 Exercícios

Mesmo exercício que no capítulo sobre o modelo a objetos da disciplina: modelem as classes e as relações com UML.

Capítulo 4

Diagrama de classes (avançado)

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos aprofundar nosso conhecimento do diagrama de classe na UML. Vamos estudar novos conceitos do modelo a objetos (ex.: interface, classe abstrata, visibilidade, etc.) e da propria UML (ex.: propriedade de atributos, pacotes, etc.).

Este capítulo é baseado nos capítulos 6, 9, 11 e 12 do "UML User Guide" (c.f. referência no capítulo "Introdução" da disciplina).

4.1 Vários adornos

Na UML, *adornos* são notações adicionais para detalhar alguns aspetos de uma classe, uma relação, ...

Existem muitos adornos que não vimos. Alguns são muito gerais, outros aplicam-se só às classes, ou só às associações, etc. Nessa seção vamos estudar os adornos seguintes: estereótipo, visibilidade, multiplicidade de classe, restrições, propriedades de atributos/métodos, notas e compartimentos com nome.

4.1.1 Estereótipo

O primeiro adorno que vamos ver é o *estereótipo*. Na UML se pode usar estereótipos para acrescentar novas construções à linguagem. Por exemplo, se pode usar estereótipos para criar várias categorias de atributo e/ou método numa classe que tem muitas propriedades (c.f. figura 4.1). Estereótipos podem também ser usados para criar um novo tipo de classe ou de relação. Por exemplo, as "exceções" da linguagem Java poderiam ser representadas como classes com um estereótipo especial ≪Exceção≫.

Estereótipos são representado entre ≪ e ≫. Tem exemplos de esteréotipos de classes e relações nas figuras 4.3 (≪subsystem≫, ≪signal≫, ≪import≫) e 4.2 (≪instanceOf≫).

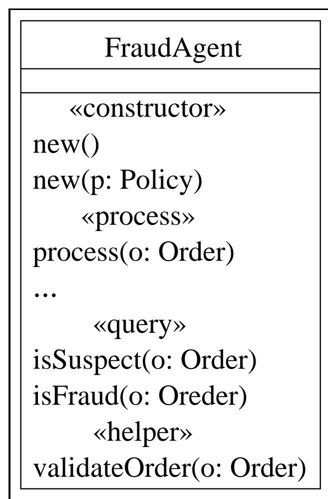

Figura 4.1: Uso de estereótipo para organizar as operações de uma classe. (copiado de "*UML User Guide*", p. 55)

4.1.2 Visibilidade

Já vimos que dentro de uma classe, alguns métodos implementam operações do conceito representado pela classe (operações da interface), e alguns métodos são apenas funções utilitárias. Os métodos que implementam propriedade do conceito tem que ser "visivel" do exterior da classe (são as operações que a classe possui oficialmente). As funções utilitárias são escondidas do exterior, só a classe mesma pode utilizá-las.

No modelo de objetos como na UML, uma operação visível ao exterior se chama de "*public*" (pública), se não estiver, ela é "*protected*" (protegida). Existe um terceiro tipo ("*private*", i.e. privada) que indica que o método não é visivel ao exterior (é "protected") e não é visivel às sub-classes. A UML usa os três símbolos (+, #, -) para indicar qual é a visibilidade de uma propriedade (public, protected, private). Veremos mais tarde nessa aula que classes dentro de pacotes podem também ter indicador de visibilidade, a figura 4.3 apresenta um pacote "WindowsGUI" com classes públicas (ex.: Form) e privadas (EventHandler).

4.1.3 Multiplicidade de classe

Normalmente, uma classe pode ter qualquer número de instâncias. Portanto, é possível indicar que uma classe só pode ter um número definido de instâncias (uma *multiplicidade*). O número é indicado no canto superior direito da classe.

4.1.4 Restrições

Freqüentemente, se deve exprimir *restrições* num modelo. Um número definido de instâncias para uma classe é um exemplo de um tal limitação. UML não tem uma notação especial para todas as restrições mas fornece uma notação geral para exprimir qualquer restrição.

Por isso, basta escrever um texto curto entre chaves (ex.: `{peso professor<100Kg}`) perto da classe ou relação que se deseja restringir.

4.1.5 Notas

No diagrama de classe, também como em outros diagramas, se pode criar *notas*. Uma nota é um comentário, que não pertence realmente ao modelo mas que o conceptor acha importante lembrar. Notas são representadas com uma caixa simples com o canto superior direito dobrado. Podem conter qualquer texto ou desenho ou fórmula matemática, etc. (p.78 do "UML User Guide").

4.1.6 Compartimentos adicionais das classes

Podemos acrescentar às classes outros compartimentos que os três básicos (nome da classe, atributos e operações). Esses novos compartimentos vão ter nome para explicar o que contêm. O compartimento das responsabilidades que já vimos é um tal compartimento com nome. Um outro exemplo possíveis seria acresentar a uma classe um compartimento para listar as exceções que ela pode atirar (v. a linguagem Java).

4.1.7 Outros adornos

Finalmente a UML propõe algumas propriedades que se podem usar para determinar a natureza dos atributos ou dos métodos. Por exemplo, um atributo pode ser "changeable" (modificavel). Tem três propriedades possivel para os atributos e sete para os métodos (p.128 e 129 do "UML User Guide"). Não vamos ver os detalhes dessas propriedades.

4.2 Classes abstratas, interfaces e "templates"

Nessa seção, vamos estudar novos tipos de classes: as classes abstrata, as interfaces e as classes "templates".

4.2.1 Classe abstrata

Já vimos que uma sub-classe tem todas as operações da super-classe dela. Isso pode ser usado para reutilização de código (mas não pode ser o único incentivo a uma relação de generalização). Vamos ver agora que a generalização pode, também, ser usada para especificar um contrato que todas as instâncias devem respeitar.

Por exemplo, podemos definir uma classe PeçaXadrez geral para todas as peças, cada peça vai ser um objeto e teremos também sub-classes de PeçaXadrez para cada tipo de peça (Rei, Rainha, Cavalheiro, Torre, Peão, etc.). Já sabemos quais operações cada sub-classe deveria implementar, por exemplo, queremos que todas tenham uma operação movimento(Casa Destino), que vai verificar se o movimento é possível e fazé-lo se for. Mas cada tipo de peça tem movimentos diferentes, não podemos implementar a operação na super-classe PeçaXadrez. Nessa super-classe só sabemos que queremos este método.

Para fazer isso, vamos definir um método *abstrato* na super-classe PeçaXadrez. Vamos apenas estabelecer que todas as instâncias de PeçaXadrez têm que ter uma operação movimento. Mas a operação mesmo (o método na verdade) não será implementada na super-classe porque não é possível. Cabe às sub-classes implementar efetivamente a operação.

Uma classe que contem uma operação abstrata é chamada de *classe abstrata*. Uma tal classe não pode criar instâncias porque algumas das operações dela não são implementadas. Classes abstratas têm que ter sub-classes que vão herdar os métodos (concretos) que ela define e vão definir as operações abstratas dela. Classes que têm instâncias podem ser chamadas de *concretas*.

Em UML, classes e operações abstratas são representadas com o nome em itálico (c.f. classe PeçaXadrez na figura 4.2).

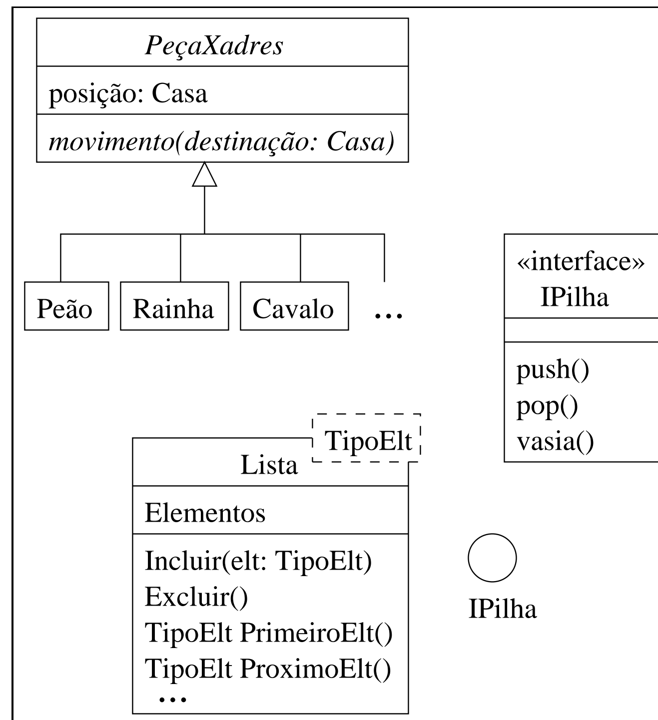

Figura 4.2: Representação de classes abstratas, interfaces e templates em UML

4.2.2 Interface

Existe um tipo de classe abstrata ainda um pouco mais abstrata: as *interfaces*. Uma interface é uma classe abstrata que tem unicamente operações abstratas (nenhum método implementado), e nenhum atributo. Uma interface especifica alguns operações para providenciar um serviço (cumprir uma responsabilidade).

Já vimos uma outra definição da palavra interface no modelo de objetos: a interface de uma classe designa todas as operações (e os atributos) da classe que são visiveis do exterior. Se pode considerar que a interface como classe abstrata defini operações que serão

visiveis do exterior das classes que vão implementá-la. Assim, as operações definidas por a interface (classe abstrata) vão ser na interface das classes que implementaram (*realizaram*) essa interface.

Interfaces podem ser representadas como classes, com um estereótipo ≪interface≫, ou mais simplesmente, apenas por um círculo com o nome da interface. Esta segunda solução suponha que a interface já foi introduzida num outro modelo.

A UML sugere acrescentar um “I” no inicio do nome das interfaces.

4.2.3 Template

Finalmente, existe um terceiro tipo de classes abstratas que são os *templates*. Nessa disciplina, a noção de template não é considerada importante, basta intender a idéia geral.

Queremos implementar uma classe geral de lista que pode aceitar qualquer tipo de dados. Uma solução é de ter uma super-classe de todos os tipos possível. Java, por exemplo, fornece a super-classe “Object” (objeto) de que todas as outras classes herdem. Nesse caso, basta dizer que nossa lista geral vai conter instâncias da class Object. Como instâncias de um sub-classe podem ser usadas em vezes de instâncias da super classe, podemos colocar qualquer instância na lista.

Mas outras linguagens (C++ por exemplo) não têm essa classe generica. Nesse caso vamos usar uma outra solução, a classe Lista mesma vai aceitar um “parâmetro” que é o tipo de seus elementos. Tais classes são chamadas de *templates*. Esse parâmetro sera uma outra classe. Por exemplo se queremos fazer uma lista de Professores, o parâmetro do template Lista vai ser a própria classe Professor.

Um template tendo uma classe no seu(s) “parâmetro(s)” pode ser considerado como uma classe normal. Essa classe normal “instância” o template. A UML representa isso com uma relação de depêndencia com um estereótipo ≪bind≫. Se o template tiver métodos abstratos a classe que o instância será uma classe abstrata. Caso contrario, ela pode ter instâncias.

Templates são representados com classes com uma caixa tracejada no canto superior direito (c.f. a figura 4.2).

4.3 Classificadores e pacotes

4.3.1 Classificador

A UML introduze a noção de *classificador* (“classifiers”). A noção não é bem clara, mas geralmente um classificador vai ter uma estrutura (ex.: atributos), um comportamento (ex.: operações) e vai “conter” outras coisas menos abstratas. Por exemplo, uma classe é um tipo particular de classificador, ela tem estrutura e comportamento, e “contem” instâncias.

Para essa disciplina, basta saber que classificadores existem e podem ser um das noções seguintes:

Classe: O tipo de classificador mais importante. Já estudimos ele.

Interface: Nós descrevemos ela com um tipo particular de classe abstrata, a UML considera que seja diferente. De qualquer forma, é um tipo de classificador.

Sinal: Uma comunicação assîncrona entre objetos. Vamos estudar ele mais tarde, na aula sobre o diagrama de atividade.

Caso de uso: A descricão de uma interação entre o sistema e os usuários. Vamos estudar casos de uso na proxima aula.

Sub-sistema: Um agrupamento de elementos que cumpre uma função abstrata.

etc.: "Tipo de data", "componente" e "nodo" são os outros classificadores em UML (p.121 "UML User Guide").

A figura 4.3 apresenta vários tipos de classificadores: classe ("Shape"), interface ("IUnknown"), sub-sistema ("Custom Service subsystem"), caso de uso ("Process loan") e sinal ("OffHook").

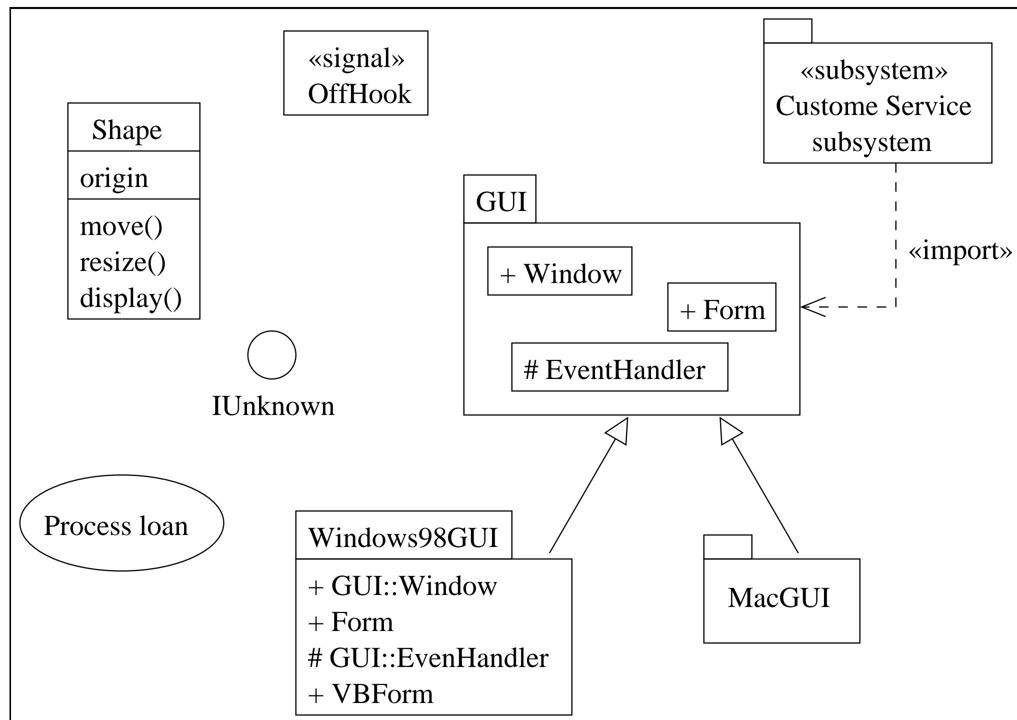

Figura 4.3: Representacao de alguns classificadores e pacotes em UML (copiado de "*UML User Guide*", p.122 e 176).

4.3.2 Pacote

Uma outra noção, mais conhecida, da UML é a de *pacote* ("packages"). Um pacote é simplesmente um agrupamento de vários elementos. O objetivo do pacote é organizar um conjunto de elementos em sub-conjuntos. Por exemplo, um sub-sistema é um tipo de pacote que permete organizar os membros de um sistema em sub-conjuntos. Já vimos que um sub-sistema é também um classificador. Todos os pacotes não são classificadores.

Alguns tipos de pacotes são apresentados na figura 4.3: um sub-sistema ("Custom Service subsystem") e outros pacotes não especificados.

Algumas propriedades dos pacotes:

- Pacotes podem conter vários tipos de elementos da UML: classes, interfaces, diagramas, outros pacotes, etc.

- Pacotes não podem conter dois elementos com o mesmo nome. Se pode acrescentar o nome do pacote ao nome de um elemento dentro desse pacote para ficar claro onde o elemento se encontra. Em tal caso, o nome do pacote é separado por dois dois pontos (::) do nome da classe. Por exemplo, a classe "Form" dentro do pacote "GUI" (figura 4.3) pode ser nomeada "GUI::Form". Quando o pacote for também dentro de um outro pacote, o nome dele pode ser acrescendo de novo: "IO::GUI::Form".

- Elementos dentro de um pacote tem uma visibilidade (ver acima) que define se eles são accesíveis ao exterior do pacote. Por exemplo se um pacote contem uma classe privada ("private"), outros pacotes exteriores não têm acesso essa classe (por exemplo não podem criar objetos dela). Visibilidade dos elementos é especificada como explicado acima.

- Pacotes não são classes, eles não têm instâncias.

- Pacotes podem *importar* outros pacotes. Quando um pacote importar um outro pacote, ele (seus elementos) ganha(m) acesso aos elementos públicos do pacote importado. Se pode considerar que declarar um elemento do pacote público já é uma exportação desse elemento.

- Pacotes podem também herdar de outros pacotes. Herança entre pacotes significa que o sub-pacote herda todos os elementos do super-pacote e pode acrescentar seus próprios elementos (ou subsituir nova definição para elementos herdados).

  Na figura 4.3, o pacote "MacGUI" herda de "GUI".

- Pacotes não são abstrações de objetos do domínio. Tais abstação são classes. Pacotes são criados para organizar os elementos do sistema.

Nessa disciplina vamos mencionar três tipos especificos de pacote: sistema, sub-sistema e quadros ("framework").

4.4 Relações

Vamos estudar nessa seção novos tipos de relações (aggregação/composição, e realização) e adornos para relações (estereóptipos de dependência, restrição sobre generalização, espicificação de interface, qualificação de associação e classe de associação).

Exemplos de todos esses conceitos são apresentados na figura 4.4.

4.4.1 Realização

Começamos com um novo tipo de relação: a *realização*. A realização é uma relação entre um classificador que especifica um contrato (por exemplo uma interface) e um classificador que cumpre o contrato (por exemplo uma classe).

Tem duas representações possíveis para a realização de acordo com as duas representações das interfaces (v. figura 4.2).

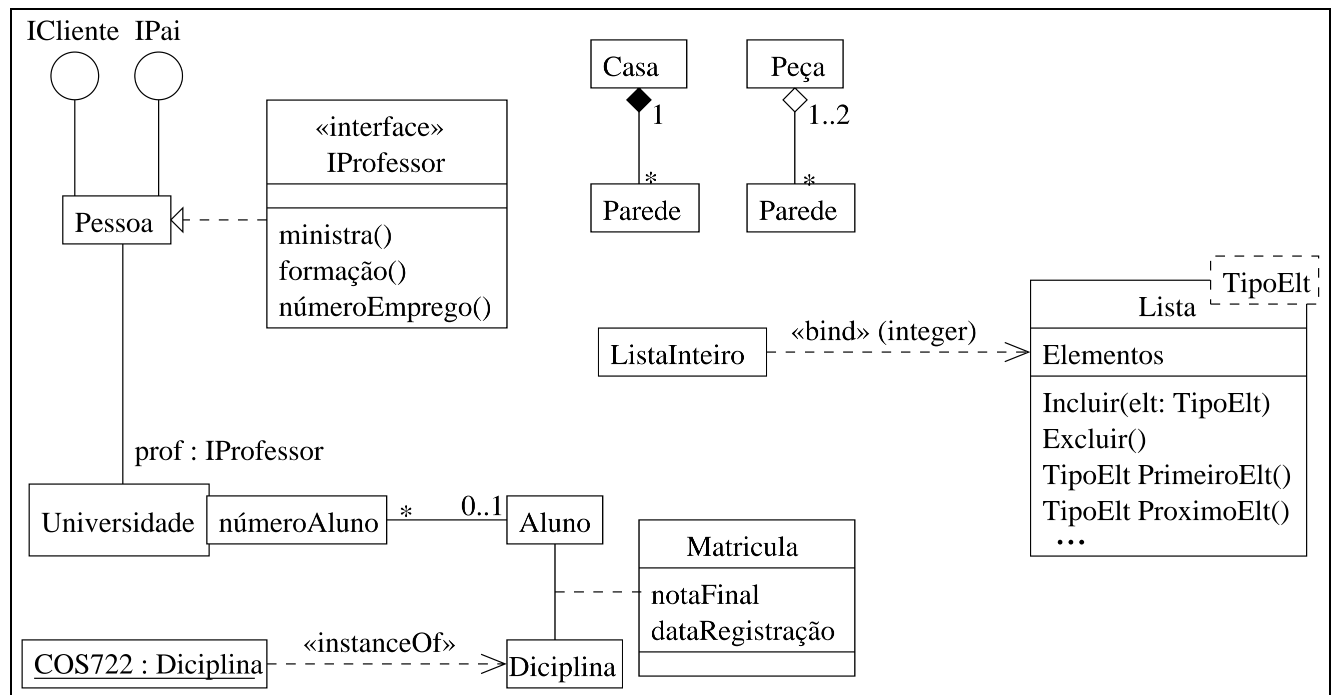

Figura 4.4: Representação de algumas relações avançadas em UML

4.4.2 Agregação e composição

A associação tem duas formas particulares: a *agregação* e a *composição*. As duas são muito parecidas, elas relacionam um objeto composto com suas partes. Por exemplo, a associação entre uma universidade e seus departamentos é uma agregação, a associação entre um carro e suas partes é uma composição. A diferença entre os dois é que a composição é mais "física", com a composição, uma parte não pode pertencer a dois objetos compostos ao mesmo tempo (um motor não pode pertencer a dois carros ao mesmo tempo), e uma parte não pode existir sem o objeto composto (mas o composto pode "sobre-viver" às suas partes).

A diferença entre os dois não é sempre clara. Por exemplo, a associação entre uma casa e os suas paredes é uma composição, mas entre uma peça da casa e um parede, é uma agregação (a parede pertence às duas peças e não desaparece com a destrução de uma das duas peças). Em caso de dúvidas, é melhor usar a agregação que é menos restrita, ou até uma associação simples.

Agregação e composicão aceitam multiplicidade, só do lado das partes para a composição (um carro é composto de quatro rodas) e dos dois lados com a agregação (uma palavra pode pertencer a várias frases e uma frase possui várias palavras).

Geralmente, a composição implica uma forma de propagação de algumas propriedades. Por exemplo, quando o objeto composto morrer, as partes morrem também (definição da composição), ou quando um carro se mover, as partes se movem também. Essas propagação não é a herança, ela não é automática, você tem que especificá-la e implementá-la. Todas as propriedades não são propagadas, por exemplo, um carro pode ser vermelho e o motor não.

Agregaçao e composição são representadas com associações acrescentadas de um losango do lado do objeto composto: losango vazio para a agregação e losango preto para a composição. A composição pode também ser representada com uma classe com as partes dentro do retângulo (um pouco como o pacote "GUI" na figura 4.3).

4.4.3 Especificação de interface

Vamos agora ver vários adornos às relações. O primeiro é para a associação: a *especificação de interface*. Cada classe pode realizar várias interfaces, por exemplo, já vimos que uma pessoa pode ser um professor, um cliente, um pai, etc. Dentro de uma associação com uma outra classe, Pessoa pode mostrar só uma faceta da sua "personalidade". Por exemplo, na associação com a classe Universidade, Pessoa pode mostrar só a faceta Professor, para a universidade, a classe Pessoa só realiza a interface IProfessor. Isso se chama de especificação de interface, a classe Pessoa especifica que ela deseja aperacer apenas como um IProfessor para a Universidade. A especificação se representa com um papel da associação (v. primeira aula sobre o diagrama de classe) acrescentado do nome da interface (c.f. a figura 4.4).

4.4.4 Qualificação

Um outro adorno de associação, é a *qualificação* de associação. Em uma universidade, cada aluno tem um número (identificador). A universidade usa esse número para "aceder" ao aluno. É a finalidade de um identificador. Essa situação (associação entre duas classes e uma usa um identificador para aceder os objetos da outra) se representada com uma qualificação (ver a figura). A qualificador é um atributo que se ata à classe que usa o indentificador (no exemplo acima, a universidade) e não aquela que "possui" o identificador (o aluno).

A qualificação muda tambem a multiplicidade da associação. Na figura 4.4, a multiplicidade de aluno na associação é zero ou um porque a cada numero de aluno pode corresponder só um aluno (ou zero se o número não ser usado ainda). É como se o aluno for associado ao número e não a universidade. Na figura, a multiplicidade "*" indiqua que um aluno pode registrar em várias universidades.

4.4.5 Classe de associação

Associações podem ter atributos também. Por exemplo, um aluno tem uma nota final a uma disciplina. A nota não é um atributo da Disciplina (cada aluno tem uma), e não é do aluno

(ele é matriculado em várias disciplinas). É um atributo da associação mesmo, o aluno tem uma nota final porque ele matriculou na disciplina.

Esses atributos são modelados com uma *classe de associação*. Cada instância da associação (entre um aluno particular e uma disciplina particular) vai ter uma instância da classe de associação. Não pode ter instância dessa classe fora da associação (e não pode ter associação sem uma instância dessa classe).

4.4.6 Estereótipos de dependência

A relação de dependência tem também adornos. No caso dessa relação são 17 estereótipos especificos de dependência. Não vamos ver eles aqui, você pode encontrar eles na pagina 137 e seguintes do "UML User Guide". Vamos mencionar só dois aqui: ≪instanceOf≫ e ≪import≫. A dependência ≪instanceOf≫ existe entre um classificador (ex.: uma classe) e uma instância dele (ex.: um objeto). Esta apresentada na figura 4.4.

Já vimos a dependência ≪import≫ no inicio dessa aula. Pode existir por exemplo entre dois pacotes (c.f. exemplo na figura 4.3).

4.4.7 Restrição

Finalmente, a UML define restrições sobre as relações de generalização (4 restrições) e a associação (5 restrições). Para a relação de generalização, esses restrições exprimem que todas as sub-classes são listadas no modelo ou não ({complete}/{incomplete}) ou que as sub-classes se excluem mutualmente ou não ({disjoint}/{overlapping}). Para associações, as restrições podem exprimir que os objetos de um lado da associação são em ordem (ordered) ou que a relação entre dois objetos pode mudar ou não ({changeable}/{addOnly}/{frozen}).

Capítulo 5

Diagrama de casos de uso

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos estudar o primeiro diagrama para modelar o comportamento de um sistema: os casos de uso. Vamos discutir várias noções, incluindo: caso de uso, atores, extensão e inclusão de casos de uso, cenário, ...

Este capítulo é baseado nos capítulos 16 e 17 do "UML User Guide" e o livro "Applying Use Cases – A Practical Guide" (c.f. referência no capítulo "Introdução" da disciplina).

5.1 Descrição geral

Este diagrama mostra como o sistema a ser desenvolvido vai interagir com seu ambiente (usuários ou outros sistemas). Ele é bastante importante porque vai ser a base do processo de desenvolvimento do sistema. O diagrama de classes especifica a estrutura do domínio e do sistema, os casos de usos vão ser a entrada para formalizar as funcionalidades que o sistema deve cumprir.

Um caso de uso descreve as operações que o sistema deve cumprir para cada usuário. Ele vai ajudar a formalizar as funções que o sistema precisa fazer. Vamos definir um caso de uso para cada tarefa que o sistema deve cumprir para um usuário.

Por exemplo, para o sistema de reservas, vamos ter um caso de uso para fazer uma reserva, consultar as reservas, suprimir uma reserva, imprimir relatórios, etc.

Um *caso de uso* se apresenta como uma lista completa das interações entre um usuário e o sistema para cumprir uma tarefa. Lista completa significa que o caso de uso descreve as interações desde o início da tarefa, até o fim.

Um exemplo de caso de uso é proposto na figura 5.1.

Casos de uso descrevem interações entre o sistema e *atores*. Um ator é "alguma coisa" (usuário, outro sistema, ...) que não faz parte do sistema e interage com ele. Por exemplo para um sistema de regulagem da temperatura, o termostato vai ser um ator, ele interage com o sistema e não faz parte dele.

Um usuário real pode cumprir vários papeis para o sistema (i.e. ser representado por vários atores), por exemplo num banco, o gerente pode ser um ator quando ele coloca dinheiro no caixa (ator: operador) e um outro ator quando ele tira dinheiro de sua própria conta (ator:

Tirar dinheiro de um caixa eletrônico

1. O usuário introduz o cartão no caixa eletrônico
2. O caixa eletrônico propõe varias operações
3. O usuário aperta o botão “saque”
4. O usuário escolhe a conta (ex.: “conta corrente”)
5. O usuário entra a valor do saque
6. O usuário entra sua senha
7. O caixa eletrônico verifica a senha com o banco e o saldo da conta
8. O caixa eletrônico da o dinheiro para o usuário
9. O caixa eletrônico imprime um recibo

Figura 5.1: Um exemplo de caso de uso

cliente). Um ator pode também representar várias pessoas, quando falamos de um “cliente” tirando dinheiro do caixa eletrônico, é claro que esse ator representa qualquer pessoa.

É importante identificar todos os atores que vão interagir com o sistema. E para cada ator, é importante identificar que operações ele vai precisar. A pesquisa de caso de usos se faz a partir dos atores, não a partir das operações. Isso quer dizer que não vamos procurar qualquer funcionalidade abstrata que seria interessante para o sistema cumprir, mas vamos procurar funcionalidades concretas que um usuário realmente precisa.

Geralmente, um caso de uso é iniciado por um ator, não pelo sistema. Por exemplo, na figura 5.1, o primeiro passo é “O usuário introduz o cartão”.

Um caso de uso só descreve as interações entre o ator e o sistema, não descreve como essas funcionalidades vão ser implementadas. Por exemplo na figura 5.1 não descrevemos como o sistema vai verificar a senha ou o saldo da conta. Isso será explicitado mais tarde, com outros diagramas.

> **Nota** : No desenvolvimento de um sistema, é muito importante fazer a distinção entre o “que” e o “como”.
>
> Pensar no “como” cedo demais poderia influenciar o “que” de uma maneira útil para o programador mas não para o usuário. É importante no início do desenvolvimento tentar não pensar na implementação e só concentrar nos desejos dos usuários. Poderemos modificar esses desejos depois se tiver um problema com o projeto, mas desse modo, a modificação será explicita. O que significa que depois de alguns anos, saberemos porque tomamos esta decisão.

> **Nota** : De maneira geral, no desenvolvimento de um software, é importante sempre gravar as razões de todas as decisões para poder reutilizá-las depois. A ausência de tal informação é um problema muito comum e importante na manutenção de sistemas.

> Também é muito difícil gravar todas as decisões porque muitas delas são feitas de maneira inconsciente.

Casos de uso têm vários objetivos importantes:

- Ser compreensíveis para usuários que provavelmente não entendem informática.
- Incentivar a análise do sistema especificando as funcionalidades necessárias.
- Delimitar o sistema.
- Servir de base para os casos de teste.

Casos de uso têm que ser compreensíveis por usuários por que só eles sabem o que o sistema precisa fazer. Os casos de uso permitem verificar se o desenvolvedor e o usuário concordam sobre o que o sistema deve fazer. Isso é um problema importante no desenvolvimento de software. No mesmo tempo, casos de uso podem servir de "contratos" entre os usúarios e a equipe de desenvolvimento.

No "Unified Software Developement Process", os casos de uso vão, também, incentivar o desenvolvimento do sistema porque eles descrevem as funcionalidades necessárias. Tudo o que o sistema precisa fazer vai ser descrito em um dos casos de uso.

Já falamos que um ator é uma "coisa" que não faz parte do sistema, nesse sentido, quando identificarmos atores e o que eles devem fazer, nós estamos definindo os limites do sistema. Para especificar uma interação, devemos definir o que o sistema e os atores fazem, por exemplo quando entrar um número de identificação, quem vai verificar que o número é correto, o sistema ou o ator? Pode, também, ser difícil identificar atores quando eles foram outros sistemas (ou sub-sistema).

Não é sempre claro o que pertence ao sistema e o que é fora dele. Especificar o caso de uso nos obriga a tomar essa decisão explicitamente. (De novo, o fato da decisão ser explícita é muito importante).

Finalmente, os casos de uso podem também ser usados como base para criar casos de teste. Testar o sistema para verificar se ele faz o que precisa e sem bugs é uma tarefa fundamental do desenvolvimento de software. Mas ao mesmo tempo, é difícil estabelecer testes bons. Os casos de uso são muito úteis nesse sentido, já que um caso de uso descreve uma interação completa e real com o sistema.

Os casos de uso são descrições muito abstrata das funcionalidades necessárias. Quando vamos afinar mais a especificação do sistema, cada caso de uso poderá ser formalizado com diagramas de seqüência e/ou de colaboração. Esses diagramas são mais formais, e então mais úteis para especificar precisamente (sem ambigüidades) o funcionamento do sistema. Mas, ao mesmo tempo, esses diagramas podem ser mais difíceis de entender pelo usuário. Normalmente, tem vários desses diagramas para um só caso de uso.

5.2 O diagrama

O nome de um caso de uso pode ser qualquer sentencia, mas a UML recomenda usar uma frase ativa curta (verbo + substantivo), por exemplo: "entrar reserva", "tirar dinheiro", ...

Esse diagrama é diferente dos outros porque contem poucos gráficos. Um caso de uso é principalmente composto de texto livre, ou pseudo-código que descreve cada interação. Os elementos principais do diagrama são uma elipse para representar um caso de uso e um pequeno boneco para representar um ator (figura 5.2). O nome do caso de uso pode ser dentro da elipse ou abaixo dele.

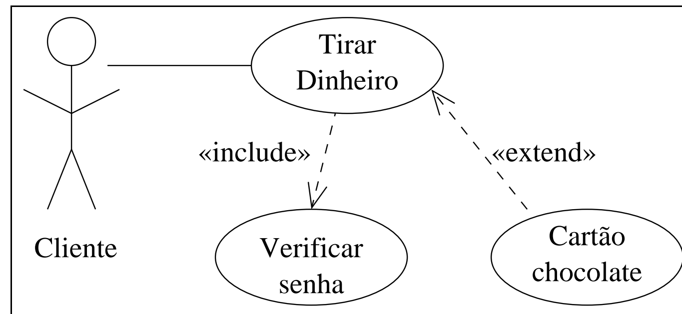

Figura 5.2: A representação do diagrama de caso de uso em UML

O diagrama não especifica os casos de uso, mas só apresenta qual ator interage com qual caso e organiza os casos entre eles (ex.: qual herda de um outro). Os casos de uso vão ser espificados em outros documentos. A UML não define precisamente a forma dessa descrição. Ela é um texto livre.

Casos de usos podem ser organizados em pacotes, ligados com a relação de generalização ou a relação de dependência. Tal organização de casos de uso será usada só para o desenvolvimento de sistemas amplos. Ela não é considerada importante nessa disciplina.

O livro "Applying Use Cases" (c.f. referências no capítulo "Introdução" da disciplina) propõe uma estrutura um pouco mais formal de especificação dos casos de uso (p.84 e anexo B). Vocês podem também procurar essa especificação na internet: http://books.txt.com.

Vamos estudar aqui uma versão simplificada dessa estrutura (figura 5.3). Notem que essa estrutura introduz várias noções que serão descritas na próxima seção.

Esta simplificação é ainda muito rica e não será usada inteiramente na maioria dos casos. Um caso mínimo vai ter um nome e o(s) fluxo(s) de eventos (principal e possivelmente alternativos). De maneira geral, a descrição abstrata ao inicio é uma boa idéia. Pontos de extensão e casos de uso incluídos (ver próxima seção) são opcionais sendo utilizando quando necessário.

5.3 Noções avançadas

As noções mais avançadas que vamos discutir nessa seção são: fluxo de eventos e cenário, pre- e pós-condições, interfaces, a relação de generalização no diagrama de caso de uso, inclusão e extensão de casos de uso, atores de um sub-sistema.

Nome do caso de uso
Descrição do caso de uso (um parágrafo).
Atores
Lista dos nomes dos atores com descrição curta.
Prioridade
Seja este caso de uso muito importante no projeto ou acessório?
Estado
Qual é o estado desse caso de uso (ainda muito abstrato, bem especificado, fechado, ...) ?
Pre-Condições
Lista de condições que têm que ser verificadas antes que o caso de uso começa.
Fluxo de eventos
Fluxo principal
1. Primeiro passo no caso de uso.
2. Segundo passo no caso de uso.
3. ...

Fluxos alternativos
Descrever os fluxos alternativos.
Pós-Condições
Lista de condições que têm que ser verificadas depois do fim do caso de uso.
Pontos de extensão
Lista dos pontos de extensão no caso de uso.
Casos de uso incluídos
Lista dos nomes dos casos de usos incluídos.
Outros requisitos
Lista de outros requisitos que afetam o caso de uso.

Figura 5.3: Especificação de um caso de uso

5.3.1 Fluxo de eventos, cenário

Um caso de uso descreve um fluxo de eventos. Normalmente, para cumprir uma operação, pode ter vários fluxos de eventos alternativos. Por exemplo, para o caso de uso "tirar dinheiro", tem o fluxo de eventos principal descrito na figura 5.1. Mas podem existir também outros fluxos que se referem ao usuário errar na entrada da sua senha, querer cancelar a operação, ou não ter dinheiro suficiente na sua conta, etc.

Cada fluxo de eventos, cada caminho de interação desde o início da tarefa até o fim é chamado de *cenário*.

A maioria das operações são amplas demais para nós descrevermos todos os cenários simultaneamente. Para simplificar o trabalho, vamos procurar definir independentemente cada cenário. Isto deve também nos ajudar achar todos os cenário possíveis.

Vamos sempre definir primeiro o "fluxo principal" de eventos que é o cenário "normal", sem erros, nem cancelamento, etc. É o cenário que descrevemos no caso de uso exemplo da

figura 5.1. Em geral, esse caso será aquele que um usuário vai descrever em primeiro lugar também. É o caso básico a partir do qual poderemos acrescentar variações, nuanças e casos "anormais" (erros, cancelamentos, etc.)

Todas essas variações vão ser descritas depois nos fluxos alternativos. Tem vários tipos de cenários alternativos:

- Caminhos alternativos (menos freqüentes) para chegar ao mesmo ponto (cliente pode tirar dinheiro da poupança em vez da conta corrente).
- Caso de erro (por exemplo a senha do cliente é errada).
- Caso de cancelamento (o cliente desistiu de tirar dinheiro).

5.3.2 Pré-condições, pós-condições

A descrição do caso de uso pode conter condições que tem que ser verificadas (verdadeiras) antes o caso ser executado (pré-condições) ou depois que for executado (pós-condições).

As pré-condições especificam qual é o estado do sistema antes do caso começar. As pós-condições indicam em qual estado o caso de uso vai deixar o sistema. Essas condições tem que ser verdadeiras independentemente do cenário que for executado (incluindo um possível cancelamaneto).

Por exemplo uma pre-condição para nosso caso de uso seria que o caixa eletrônico estivesse ligado e não seja usado por outra pessoa nesse momento. A pós-condição, nesse caso, poderia ser a mesma. Depois do cliente tirar dinheiro, a maquina tem que estiver de novo pronta para atender um novo cliente (ou o mesmo para uma outra operação). Não importa qual cenário o cliente executou exatamente —se ele tirou dinheiro, se ele cancelou ou se ele não tinha dinheiro suficiente na conta— a pós-condição tem que ser verificada.

Pré-condições não podem depender de informações que vão ser entradas no caso de uso mesmo. Por exemplo, no caso Tirar dinheiro, uma pré-condição não pode ser que o cliente tiver conta nesse banco, porque antes do caso (é uma PRÉ-condição), não sabemos ainda qual é o banco dele, ele ainda não introduziu o cartão.

5.3.3 Interfaces

O livro "Applying Use Cases" propõe de associar interfaces aos casos de uso e atores. Como já vimos, as interfaces vão especificar quais operações o ator (ou o caso de uso) realizam. No caso de um ator usúario, é claro que as operações são virtuais e só servem para descrever as ações que ele pode fazer. Essas interfaces vão permitir especificar um pouco mais as interações. Interfaces podem também ser úteis para reutilização de casos de uso (ou parte de casos).

Com nosso caso de uso, a interface do ator cliente vai conter as operações "entra operação escolhida", "entra valor do saque" e "entra senha".

A representação das interfaces é indicada na figura 5.4. A linha simples indica quem realiza a interface, a seta tracejada indica quem usa a interface.

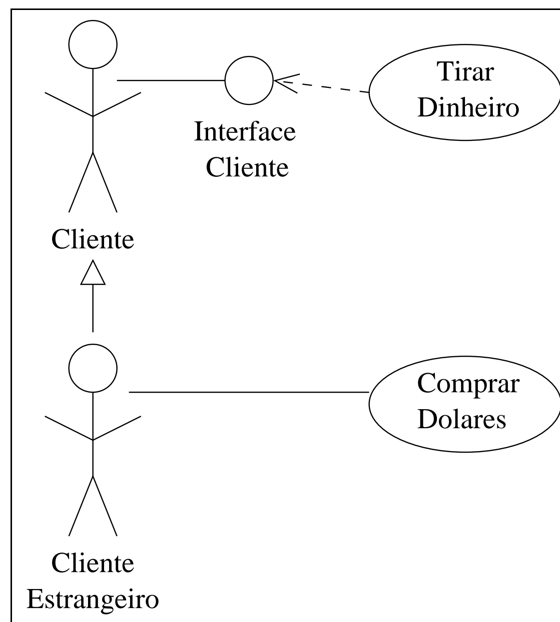

Figura 5.4: Interfaces e generalização para atores e casos de uso.

5.3.4 Generalização

O livro "Applying Use Cases" diz que existe generalização só entre atores, não entre caso de usos. O "UML User Guide" diz que existe generalização entre caso de usos. Essa segunda interpretação parece a melhor interpretação.

Entre atores, a generalização significa que o ator mais especifico pode ser usado em vez de o ator mais geral. O ator que herda interage com os mesmos casos de uso que o ator mais geral, e pode também interagir com outros casos de uso que o mais geral não tem.

Entre casos de uso, a generalização significa a mesma coisa: o caso de uso mais especifico pode ser usado em vez de o mais geral. Por exemplo, o caso de uso "Verificar senha" pode ser usado em vez de um caso de uso mais geral "Identificar usuário". Outros casos de uso específicos podem também ser usados em vez de "Identificar usuário".

Essas relações de generalização talvez sejam mais fácil de entender quando pensarmos nas interfaces: um ator ou um caso de uso "realizam" uma interface e o "sub-ator" ou o "sub-caso" têm que realizar a mesma interface.

Na figura 5.4, podemos ver uma especialização do ator Cliente que pode efetuar o caso de uso "tirar dinheiro" mas também um outro caso especifico a esse tipo de Cliente.

5.3.5 Inclusão

A UML propõe também duas relações de dependência entre casos de uso: ≪estende≫ e ≪inclui≫.

A dependência ≪inclui≫ é usada para decompor um caso de uso complexo em sub-partes. O caso de uso complexo inclui um outro que cumpre uma sub-parte dele (exemplo). Esse mecanismo permite não repetir a mesma sub-parte várias vezes. Nesse caso, os dois casos de

uso (o complexo e a sub-parte) não podem ser encontrados só porque o caso complexo não é completo sem a sub-parte, e a sub-parte não é um caso completo e não significa nada fora do contexto do caso complexo.

5.3.6 Extensão

A dependência ≪estende≫ indica uma extensão possível de um caso de uso básico. No caso de uso básico, tem um ponto de extensão onde se pode usar o caso de uso extensão. Isso quer dizer que nos casos normais, apenas o caso de uso básico será usado, mas em alguns casos, ele será estendido pelo caso de uso extensão que vai acrescentar alguns interações. Ao contrário da inclusão, o caso de uso que é extendido não sabe da existência do caso de uso extensão.

Por exemplo, o caso de uso descrito na figura 5.1 poderia conter um "ponto de extensão" antes do ponto 8 para os clientes com cartão "chocolate" que indique que o cliente deve deixar pelo menos 500$Reais na conta. O caso de uso extensão é proposto na figura 5.5.

Caso de uso extensão "Cartão chocolate"
Se o cartão for um cartão chocolate, então extensão "cliente especial":

1. Caixa verifica com o banco que o saldo da conta será superior a 500$Reais depois do saque

Figura 5.5: Um exemplo de caso de uso

O caso de uso estendido (figura 5.1) deve especificar quais são seus pontos de extensão, por exemplo assim: Ponto de extensão "Cliente especial" antes do passo 8.

Num diagrama, os pontos de extensão podem ser especificados também: a elipse do caso de uso é separada em dois, na parte superior se coloca o nome do caso de uso, na parte inferior se coloca os pontos de extensão.

O caso extensão deve conter uma condição ao inicio. Ele é "executado" somente se a condição for verdade. Esse caso de uso tem que especificar também qual ponto de extensão ele usa, porque um caso de uso pode ter mais de um.

Um ponto de extensão único pode ser usado por vários casos extensão.

5.3.7 Sub-sistema

Em muitos casos, o sistema é amplo demais para ser entendido inteiramente. Por isso, decompomos tais sistemas em sub-sistemas de tamanho menor que podemos entender completamente até um nível suficiente de detalhes para implementá-los.

Um exemplo possível de decomposição em sub-sistemas seria de ter um sub-sistema gerando uma base de dados, um gerando as interações com os usúarios (telas, impressões, etc.), e um intermédio fazendo computações. Esses sub-sistemas vão ter que comunicar entre

eles, por exemplo o sub-sistema de computação pede dados ao sub-sistema do banco de dados, faz algumas computações sobre esses dados e passa os resultados ao terceiro sub-sistema para imprimir.

Para cada sub-sistema, os outros sub-sistemas são atores com quem ele vai interagir. A noção de interface nesse caso é mais clara, ela corresponde à interface do sub-sistema: quais operações ele exporta para outros sub-sistema.

5.4 "Truques"

Procurem primeiro os atores que interagem com o sistema. Para isso, procurem responder perguntas como: Quem usa o sistema? Quem o instala? Quem o inicia ou o pára? Quem faz a manutenção? Quem entra dados no sistema? E quem precisa/usa dados do sistema?

É bom definir um pequeno dicionário dos atores, que descreve sucintamente cada ator. Isso permite verificar que temos uma idéia clara de o que o ator representa e permite também verificar com os usúarios as definições dos atores.

Depois, para cada ator, procurem definir quais operações ele precisa fazer. Uma interação é uma troca de dados: Quem fornece dados para quem (o sistema para o ator ou o ator para o sistema ou os dois)? Quem cria, consulta, modifica ou destrui os dados?

Sempre definam em primeiro o fluxo de eventos principal de um caso de uso sem se preocupar com os fluxos alternativos.

5.5 Exercícios

Especifiquem um caso de uso completo para um sistema de telefone quando:

- O ramal XXX tenta ligar o ramal YYY.
- Mesmo, mas o ramal YYY rederecionou os telefonemas para um outro ramal (ZZZ).

Quais são os casos de uso para o sistema de reserva da nossa empresa aérea ? Especifiquem cada um deles.

Capítulo 6

Diagramas de interação

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos estudar os dois diagramas de interação que são o diagrama de seqüência e o diagrama de colaboração. Esses diagramas vão especificar mais em detalhes interações como descritas nos casos de uso do capítulo precedente. Vamos introduzir os componentes desses diagramas como: objetos, conexões, os vários tipos de mensagens, etc.

Este capítulo é baseado nos capítulos 15 e 18 do "UML User Guide" (c.f. referência na introdução).

6.1 Introdução

Tem dois diagramas de interação, o diagrama de seqüência e o de colaboração. Os dois representem a mesma coisa mas com um foco um pouco diferente, é fácil passar de um ao outro. Os dois especificam como os objetos do sistema vão interagir para cumprir alguma função.

Vamos primeiro estudar os elementos básicos desses dois diagramas: mensagens, seqüência, objetos, fluxo de controle.

Diagramas de interação mostram as comunicações entre objetos (do sistema ou do lado de fora) para cumprir uma tarefa complexa demais para ser implementada com uma só operação dentro de uma só classe.

Esses dois diagramas podem modelar comunicações entre ator (fora do sistema) e objetos dentro do sistema quando foram usados para formalizar um caso de uso. Eles podem também formalizar a realização de uma operação complexa por vários objetos dentro do sistema. Finalmente, podem formalizar os dois ao mesmo tempo: eles especificam uma interação entre objetos dentro do sistema e atores, ao mesmo tempo que eles mostram as comunicações dentro do sistema para cumprir as operações complexas desejadas pelos usuários.

6.1.1 Conexões

Esses diagramas mostram objetos (instâncias de classes e atores) trocando mensagens. Objetos são ligados por conexões ("links"). As mensagens vão "caminhar" sobre essas conexões.

Uma *conexão* é uma instância entre dois objetos de uma associação entre as duas classes desses objetos. Isso mostra como a dinâmica do sistema (troca de mensagens) é baseada sobre sua estrutura (associações). Conexões parecem muito parecidas às associações, mas elas são instâncias das associações. Uma diferencia importante é que as conexões não tem multiplicidade porque elas existem só entre dois objetos particulares.

Nesses diagramas, os objetos podem ser instâncias de classes abstratas ou de interfaces das quais, falamos que não podem ter instâncias. A idéia que as instâncias representadas nesses diagramas podem ser objetos reais ou protótipos de objetos, objetos que não existem realmente mas representam outros objetos. No caso de classes abstratas, uma instância será o protótipo de instâncias das sub-classes dessa classe abstrata que podem ter instâncias. Da mesma maneira, uma instância de uma interface nesses diagramas vai representar instâncias de classes que realizam essa interface.

6.1.2 Mensagens

Na apresentação do modelo a objetos, dizemos que enviar uma mensagem a um objeto era chamar uma operação desse objeto. Nesses diagramas, as mensagens têm uma significação mais larga. Uma mensagem pode corresponder a cinco tipos de ação possíveis: chamada, retorno, envio, criação e destruição.

O primeiro tipo de ação (*chamada*) é aquele que vimos na apresentação do modelo a objetos.

Uma mensagem de *criação* (ou *destruição*) enviada a um objeto significa que este objeto é criado (ou destruído).

Uma mensagem de tipo *retorno*, é usada para mostrar a valor retornada por uma operação (retorno de uma mensagem chamada).

A mensagem *envio* é usada para sinais (o que quer dizer coisas como exceções em Java). Sinais não são operações e não pertencem ao fluxo de execução normal. Eles são marcas que um evento determinado aconteceu. Por exemplo a exceção "EndOfFileException" é produzida quando tentar ler depois do fim de um arquivo. A operação que estava lendo pode decidir de tratar a exceção (e mudar o fluxo de execução) ou não. Um sinal não tem retorno, ele é enviando e o controle do fluxo e execução passa por o objeto que recebe o sinal.

6.1.3 Fluxo de controle

A troca de mensagens entre os objetos vai definir um fluxo de controle: um objeto que executa uma operação tem o *controle*. Um objeto que tem o controle pode:

- continuar executar sua operação e guardar o controle,
- delegar parte da operação a um outro objeto e passar o controle a esse,
- terminar a operação e retornar o controle ao objeto que lhe passou (chamando sua operação).

Assim, o controle passa de objeto a objetos quando eles trocam mensagens. A passada do controle entre os objetos se chama de *fluxo de controle*.

Um fluxo de controle tem um inicio quando o sistema (ou processo ou "thread") é sendo iniciado e continua enquanto o sistema (ou processo ou thread) é sendo rodeando.

A um momento, o fluxo de controle pertence a um objeto só. O fluxo de controle só muda de objeto com mensagens. Mensagens são trocadas uma depois da outra em uma *seqüência*.

A posição de uma mensagem na seqüência é modelada com um número: primeira mensagem=número um, segunda=número dois, ... A seqüência pode ter sub-seqüências quando uma operação complexa chama sub-operações para se executar. Elas são indicadas assim (sub-seqüência da segunda operação na seqüência principal): primeira mensagem da sub-seqüência=2.1, segunda=2.2, ...

Em teoria, não tem limite ao número de sub-seqüência.

Um sistema pode ter vários fluxos de controle quando conter processos e/ou threads. Nesse caso é preciso indicar com cada mensagem a qual fluxo ela pertence. Não vamos estudar isso.

6.2 Diagramas

Existem dois diagramas para apresentar as interações entre objetos: o de seqüência e o de colaboração. O primeiro insiste sobre a ordem das mensagens no tempo, o segundo sobre a organização entre os objetos, qual objeto interage com qual outro objeto.

6.2.1 Diagrama de seqüência

A figura 6.1 apresenta um exemplo de diagrama de seqüência.

Num diagrama de seqüência, cada objeto é apresentado com uma linha vertical que representada a "vida" do objeto. A cima dessa linha tem uma caixa representando o objeto. Enquanto o objeto tem o controle (ou esta esperando para uma operação retornar o controle a ele), a linha de vida é uma caixa vertical. Caso contrário, ela é apresentada com uma linha tracejada.

Nesse diagrama, o tempo vai de cima para baixo. não é necessário especificar os números de seqüência das mensagens porque eles são implícitos com o eixo vertical do tempo.

Mensagens são trocadas entre as linhas de vida dos objetos. O diagrama não apresenta explicitamente as conexões entre os objetos. A figura mostra várias mensagens: envio de um sinal (ex.: "introduzCartão"); chamada de operações (ex.: "escolherOperação()", "entregarDinheiro()", etc.); retorno de valor (ex.: "senha") e criação e destruição (da instância de Operação).

Nessa figura, especificamos apenas o nome das operações, mas poderíamos também especificar o valor dos seus parâmetros.

Um objeto pode enviar uma mensagem a ele mesmo. Nesse caso a seta sai da linha de vida do objeto e retorna para essa mesma linha.

O diagrama permite especificar outras informações:

loop: acrescentar antes do nome da mensagem uma estrela e opcionalmente o teste de controle da loop. Por exemplo para especificar uma loop enviando a mensagem "setValue"

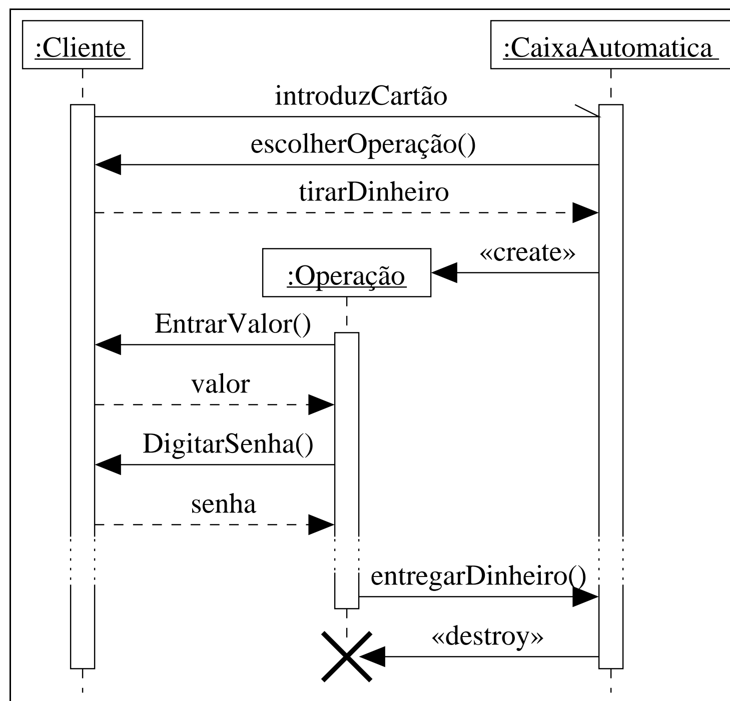

Figura 6.1: Um exemplo de diagrama de seqüência na UML

se escreve: `* : setValue()`, se quisermos fazer a loop 10 vezes: `*[i:=1..10] : setValue()`.

condição: acrescentar a condição antes do nome da mensagem. Por exemplo: `[i > 0] : setValue()`. Várias mensagens podem ser enviadas com condições diferentes a partir do mesmo ponto. Nesse caso, cada condição tem que ser mutuamente exclusiva com as outras.

recursividade: desenhar uma outra caixa na linha de vida do objeto, um pouco deslocada na direita da caixa principal.

6.2.2 Diagrama de colaboração

A figura 6.2 apresenta um exemplo de diagrama de colaboração.

Ao contrario do diagrama de seqüência, nesse diagrama o foco não é sobre o tempo, mas a organização dos objetos. Por isso, esse diagrama mostra explicitamente as conexões entre os objetos (o que o diagrama de seqüência não faz), enquanto ele deve acrescentar números de seqüência às mensagens para indicar a ordem de chamada.

Uma outra diferencia é que no diagrama de seqüência se mostra explicitamente o retorno das mensagens enquanto o diagrama de colaboração não o faz.

A figura mostra várias noções interessantes desse diagrama:

- Conexões podem ter estereótipos para clarificar como ou porque o objeto emissor da mensagem tem acesso ao objeto receptor. Tem cinco estereótipos possíveis: ≪associação≫

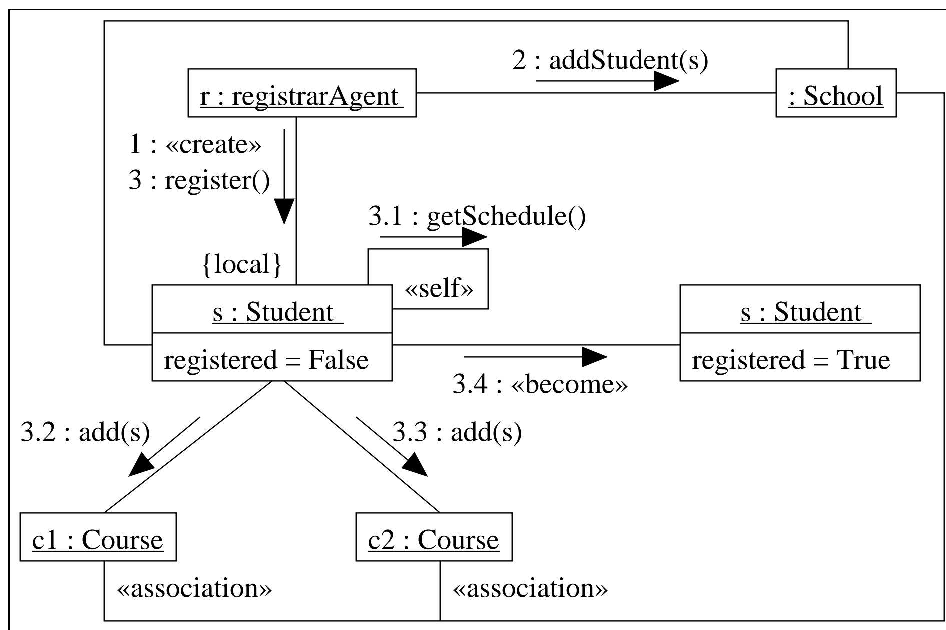

Figura 6.2: Um exemplo de diagrama de colaboração na UML (copiado de "*UML User Guide*", p. 254).

que é uma conexão normal, por causa de uma associação entre as classes dos objetos; ≪self≫ a conexão não existe realmente, o emissor esta enviando uma mensagem para ele mesmo; ≪global≫ a conexão não existe realmente, mas o receptor é disponível globalmente no contexto do emissor; ≪local≫ o receptor é disponível localmente (por exemplo, dentro de uma variável local à operação do emissor); ≪parameter≫ o receptor é disponível localmente por que foi recebido como parâmetro da operação do emissor.

- Tem duas instâncias da classe Course, isso mostra a diferencia entre diagrama de objeto e diagrama de classe, e associação e conexão.
- A instância "s" da classe Student, é apresentada duas vezes porque ela muda de estado (o atributo registered mudou de valor). Isso é representado junto com uma mensagem de estereótipo ≪become≫. Vamos discutir os estados no capítulo sobre diagrama de estados.
- Três conexões de School até Course e Student não carregam mensagens. Elas são apresentadas apenas para explicitar como um Student tem acesso aos Course usando as associações entre, primeiro Student e School, e segundo School e Course.

6.3 Exercícios

Usando um caso de uso do capítulo precedente, criar os diagramas de seqüência e colaboração correspondentes.

Capítulo 7

Diagrama de atividade, diagrama de estado

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos estudar os diagramas de atividade e de estado. Como os dois diagramas do capítulo precedente, esses diagramas vão permitir especificar com mais detalhes o comportamento do sistema. Mas enquanto os diagramas de interação especificam o comportamento de vários objetos juntos, esses dois vão especificar o comportamento de um objeto só.

Este capítulo é baseado nos capítulos 19, 20 e 21 do "UML User Guide" (c.f. referência na introdução do curso).

7.1 Introdução

Os dois diagramas de interação do capítulo precedente, como os diagramas que vamos estudar nesse capítulo, são usados para especificar em detalhes como o sistema vai cumprir as funcionalidades identificadas para os casos de uso. Os casos de uso são os mais abstratos.

Os dois diagramas de interação (diagrama de seqüência e diagrama de colaboração) especificam os interações entre objetos do sistema e o exterior (ou entre apenas objetos do sistema).

Ao contrario, os diagramas de atividade e de estado, que vamos estudar aqui, especificam o comportamento de uma entidade só, seja um objeto instância de uma classe, uma operação, um sistema, etc.

A figura 7.1 apresenta exemples muito simples dos diagramas de atividade e de estado. O diagrama de atividade apresentado contem estados de atividades (Ligar televisão, Assistir programa, etc.), um estado inicial (o círculo preto), um estado final (o círculo preto dentro de um outro círculo), transições entre esses estados e ramificações ("branch") com expressões de guarda ("guard": Programa interessante). O diagrama de estado apresentado contem dois estados (luzApagada, luzAcendeda), e dois transições com eventos (ligar, desligar).

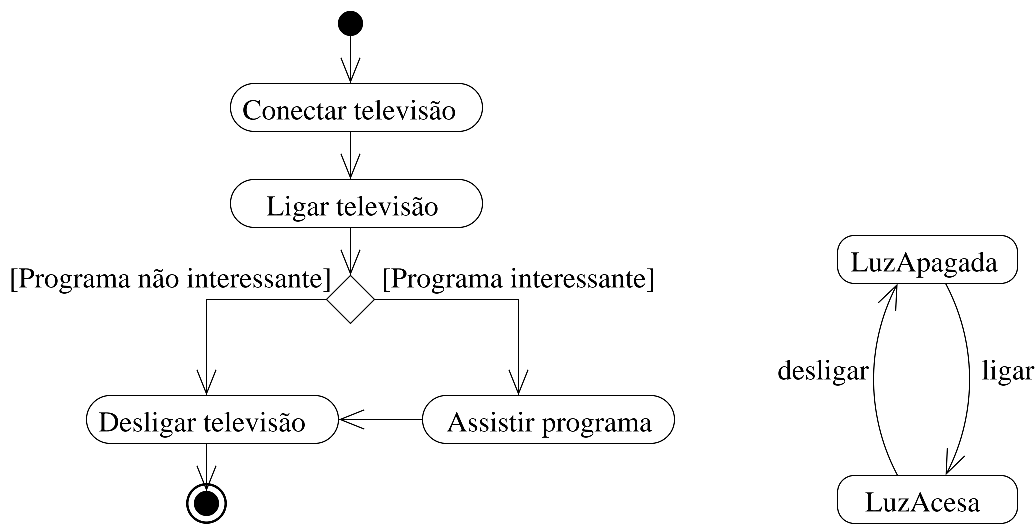

Figura 7.1: Exemplos de diagrama de atividade (a esquerda) e diagrama de estado (a direita) na UML

7.2 Componentes básicos

Vamos definir agora os componentes básicos dos dois diagramas:

Estado: A descrição de uma situação na vida do sistema ou de um objeto a um momento definido. O estado pode ser descrito como o conjunto de valores dos atributos do objeto, com um evento que ele esta esperando, ou para uma operação que ele esta executando. Nesse último caso, existem dois tipos de estados: estado de ação —enquanto o sistema ou o objeto estão executando uma ação— e estado de atividade —enquanto o sistema ou o objeto estão executando uma atividade. Um objeto permanece num estado por um tempo finito.

A UML sugere capitalizar todas as primeiras letras de cada palavra no nome de um estado (ex.: "LuzApagada") mas o nome pode conter qualquer caracter (ex.: "Ligar televizão").

Estado inicial: Um estado virtual que marca o ponto de entrada do diagrama.

Estado final: Um estado virtual que marca o(s) ponto(s) de saída do diagrama.

Ação: Uma execução atômica. Ela não pode ser interrompida e se considera que ela dura um tempo não significativo.

Os tipos de ações são: chamada de uma operação, envio de um sinal, retorno de um valor (avaliação de uma expressão, execução de um calculo), criação de um objeto, destruição de um objeto, ou modificação do valor de um atributo. Já vimos que os cinco primeiros corespondem ao envio de uma mensagem. Uma outra classificação seria de dizer que uma ação pode modificar o estado do sistema ou de um objeto ou retornar um valor.

Nota : Dizemos que um estado descreve o conjunto de valores dos atributos de um objeto, portanto a modificação do valor de um atributo muda o estado desse objeto e a criação ou destruição de objetos muda o estado do sistema. Dizemos, também, que um estado descreve uma operação que o sistema ou um objeto esta executando, portanto chamar uma operação ou enviar um sinal vão, também, mudar o estado do sistema e de alguns objetos.

Atividade: Uma execução não atômica composta de ações ou de outras atividades. Por exemplo, a execução de uma operação para um objeto é uma atividade.

Atividades podem ser interrompidas e se considera que suas execuções duram alguns tempos.

Evento: Alguma coisa que acontece. Tem quatro tipos de eventos: envio de um sinal, chamada de uma operação, passo do tempo e mudança de estado. Já vimos dois desses no capítulo precedente como tipos possíveis de mensagens: chamada de uma operação e envio de um sinal.

Uma chamada de operação é um evento sîncrono, o emissor pára sua execução e espera a resposta (retorno da operação) do outro objeto.

Sinal: Um objeto nomeado enviado ("thrown") de maneira assîncrona por um objeto e recebido ("caught") por um outro.

Sinais são eventos assîncronos —ao contrário das chamadas de operações que são eventos sîncronos— o emissor envia o sinal mas continua sua execução sem esperar resposta, nem saber se o outro objeto esta pronto para receber o sinal.

Exceções (em Java ou outras linguagens) são sinais.

Transição: A passagem de um estado para um outro. As transições mostram o fluxo de controle de uma atividade (ou ação) para uma outra.

A passagem de uma atividade para uma outra pode ser automática ("trigerless") enquanto a atividade terminou e o fluxo de controle passa para a atividade seguinte, ou pode ser provocada por um evento. Por exemplo a chamada de uma operação (um dos vários eventos possíveis) vai provocar uma mudança do fluxo de controle e vai iniciar uma nova atividade (execução da operação chamada).

7.3 Funcionamento

Os diagramas de atividade e de estado descrevem o ciclo de vida de um objeto, um sistema ou uma operação. Um diagrama começa ao estado inicial que marca o começo da operação ou do sistema, a criação do objeto, etc. A "execução" vai seguir as transições de estados em estados (de atividade ou de ação). Um diagrama pode não ter de estado final enquanto não a fim prevista à vida do objeto. Nesse caso, o diagrama vai ser uma "loop" como no exemplo de diagrama de estado da figura 7.1.

Algumas transições são automáticas ("triggerless transitions") enquanto a atividade ou a ação do estado corrente termina. Por exemplo no exemplo de diagrama de atividade (figura 7.1), imediatamente depois do começo, o fluxo de controle passa ao primeiro estado de atividade (Conectar televisão). Assim que esta atividade termina, o fluxo segue a segunda transição automática para o segundo estado de atividade.

Outras transições têm eventos. O fluxo de controle segue essas transições só quando o evento acontecer. Caso contrario, o objeto permanece no estado fonte da transição. Por exemplo, no diagrama de ação, passamos do estado LuzAscendeda ao estado LuzApagada só quando o evento desligar acontecer. Esse evento é chamado de evento acionador ("event trigger").

Transações automáticas ou com eventos podem, a mais, ter uma condição ("guard expression"). Isso significa que quando estarmos pronto para atravessar esta transição (a atividade do estado fonte terminou ou o evento aconteceu), vamos testar a condição. Se a condição for verdadeira, atravessamos a transição, se não, procuramos por outra transição que pode ser atravessada. Por exemplo, no diagrama de atividade, passamos do estado Ligar televisão ao estado Assistir programa só se a condição [Programa interessante] seja verdadeira (condições são marcadas com colchetes).

Se acontecer um evento enquanto o objeto esta num estado que não espera este evento (não tem transição de saída deste estado com este evento), o evento esta perdido, ele não vai produzir qualquer ação.

Enquanto um objeto executa uma atividade (ele esta num estado de atividade), esta pode ser interrompida se acontecer um evento que aciona alguma transição de saída desse estado. Isso não pode acontecer com estados de ação (ações não podem ser interrompidas porque duram um tempo não significativo).

7.4 Transições, eventos e sinais: noções avançadas

Ação: Podemos associar uma ação a uma transição. Quando atravessarmos a transição a ação será executada. Este mecanismo permite especificar comportamento mais complexos com menos estados e então com um diagrama mais simples.

Não é possível associar uma atividade a uma transição, só pode ser uma ação.

A ação é representada por seu nome, depois do nome do evento separada dele por uma barra (evento / ação). Se não tiver evento (transição automática, o nome da ação vem depois da barra só.

Parâmetro: Uma chamada de operação é um dos vários eventos possíveis. Nessa chamada, a operação pode ter parâmetros. Esse(s) parâmetro(s) não faz(em) parte do evento. O evento é só a chamada da operação, independentemente da parâmetro.

Um sinal enviado, pode ter atributos. Esses serão representados como atributos do sinal no diagrama de estados. Não tem diferencia gráfica entre a representação da chamada de uma operação com alguns atributos e o envio de um sinal com atributos.

Tempo: A UML reconhece um tipo especial de evento ("time event"): after(3 secundas).

Esse evento significa que depois de 3 secundas, o evento vai ser automaticamente produzido e a transição será acionada.

Nesse caso, o tempo vai começar enquanto entramos no estado fonte da transição.

Mudança: Existe um outro tipo especial de evento ("change event") representado assim: when(condição). Nesse caso, a transição será acionada logo que a condição for realizada. Esse evento é diferente de uma transição com condição (seção funcionamento) por que a segunda vem a mais do evento, enquanto a primeira é o evento (enquanto for verdadeira). O que significa que se o evento acontecer, a segunda condição é testada. A primeira esta testada continuamente logo que entramos no estado fonte da transição.

Declaração: Sinais podem ter atributos, eles são parecidos aos objetos. Se pode definir uma classe de sinais com sub-classes, etc. Sinais são definidos como classes com um estereótipo ≪sinal≫ e exceções (tipos especial de sinais) com estereótipo ≪exception≫.

No diagrama de classe, se pode indicar qual classe e/ou método envia qual sinal com uma dependência com estereótipo ≪send≫.

7.5 Estados: noções avançadas

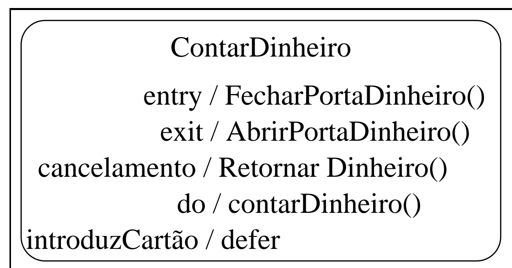

Figura 7.2: Algumas noções avançadas com estados

Um estado tem outras partes que só seu nome:

Entrada: Uma ação executado na entrada do estado. Isso é útil quando se quer executar a ação independentemente da transição com que entramos no estado.

Ações de entrada são marcadas com um evento especial: "entry" (v. FecharPortaDinheiro(), figura 7.2).

Saída: Uma ação executado antes de sair do estado, independentemente da transição com que saimos.

Ações de saída são marcadas com um evento especial: "exit" (v. AbrirPortaDinheiro(), figura 7.2).

Transição para se: ("self-transition") Uma transição em que o objeto fonte e o objeto destinação são os mesmos. Nessa transição, vamos sair do estado e re-entrar. Isso é diferente da transição interna (v. abaixo) em que não saimos do estado.

Transição interna: Podemos tratar um evento sem sair de um estado. Isso se chama de transição interna. Normalmente, um tal evento será acompanhado de uma ação (v. próxima seção). Aquela ação será executando quando receber o evento, mas não saíramos nem entraremos no estado, o que significa que não vamos executar possíveis ações de saída e entrada.

Transições internas são marcadas simplesmente com a lista dos eventos e das ações associadas (v. evento cancelamento, figura 7.2).

Eventos adiantados: Uma lista de eventos que não vão ser tratados por o estado. Se esses eventos acontecerem, serão guardados para ser tratados num outro estado.

Eventos adiantados são marcados com uma ação especial: "defer" (v. introduzCartão, figura 7.2).

Sub-estados: Um estado (de atividade) pode ser composto, o que quer dizer que ele vai conter um outro diagrama de ação ou de atividade (v. figura 7.3).

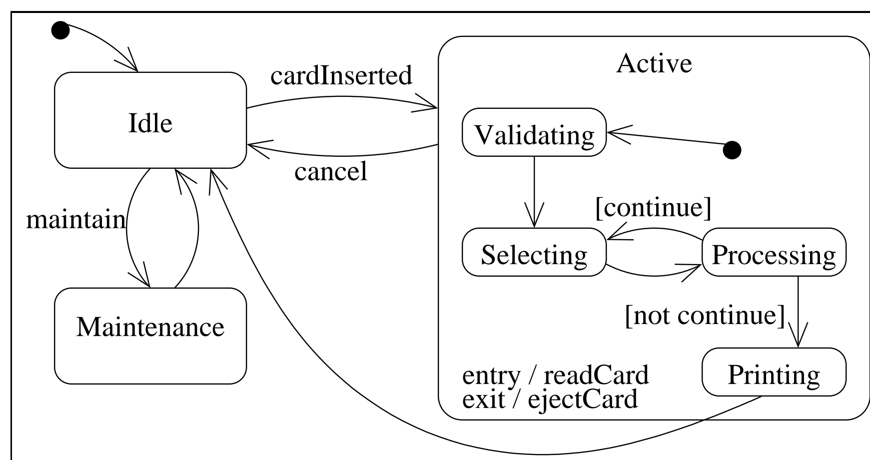

Figura 7.3: Representação de sub-estados na UML (copiado de "*UML User Guide*", p. 299)

O sub-diagrama vai conter um estado inicial que marca onde ele começa enquanto entramos no estado. O sub-diagrama pode também conter um estado final (um só).

Qualquer evento sobre uma transição com estado fonte o estado composto nos (por exemplo evento cancel de Active para Idle) nos faz sair deste estado composto e no mesmo tempo termina o sub-diagrama, nos fazendo também sair de qualquer sub-estado em que estávamos. Um exemplo típico de tal transição seria um evento de cancelamento que pode interromper qualquer atividade dentro do estado composto.

Pode também ter transição de um sub-estado até um estado do lado de fora do estado composto. Tal transição nos faz sair do sub-estado e do estado composto.

Sub-estados paralelos: Um estado (de atividade) pode também ser composto de vários diagramas (v. figura 7.4). Nesse caso, os diagramas se "executam" em paralelo. A

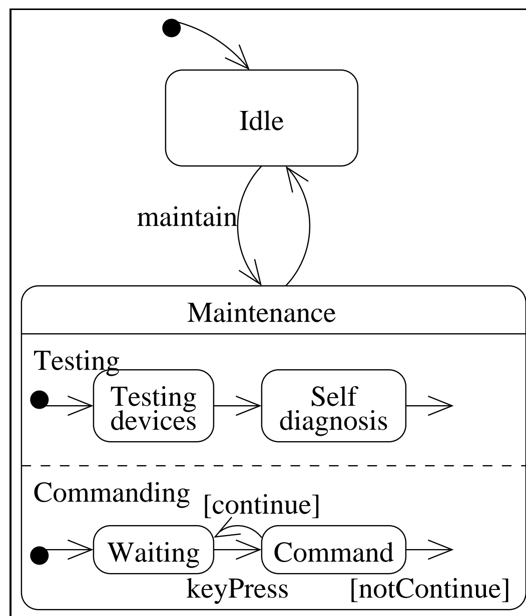

Figura 7.4: Representação de sub-estados paralelos na UML (copiado de "*UML User Guide*", p. 303)

atividade do estado composto termina enquanto todos os sub-diagramas são terminados.

Historia: Normalmente, cada vez que entramos um estado composto, seu diagrama de estado recomeça ao estado inicial. Portanto, existem casos em que queremos lembrar em qual sub-estado estivemos quando saimos a última vez do estado composto para voltar a esse sub estado quando entrarmos de novo.

Podemos especificar isso com um estado de historia (um H dentro de um círculo) como primeiro estado do sub-diagrama e uma transação que entra no estado composto, diretamente para o estado de historia (v. figura 7.5).

Quando a "execução" chegar ao o último estado do sub-diagrama, o estado de historia é reinicializado e a próxima entrada no estado composto vai recomeçar com o primeiro sub-estado.

7.6 Diagrama de atividade: noções avançadas

Decisão: ("Branch") Isso é um estado especial, —representado por um losango, cf. figura 7.1— que aceita uma transição de entrada e duas ou mais transições de saída, cada um com uma condição ("guard expression"). Todas essas condições serão testadas quando entrarmos no estado de decisão. O fluxo de controle vai seguir a única transição com a condição verdadeira. Por isso é importante que todas as condições sejam mutuamente exclusivas e que todas as situações possíveis sejam cobertas. É possível definir uma condição "[else]" (senão) que será verdadeira se nenhuma outra condição for verdadeira.

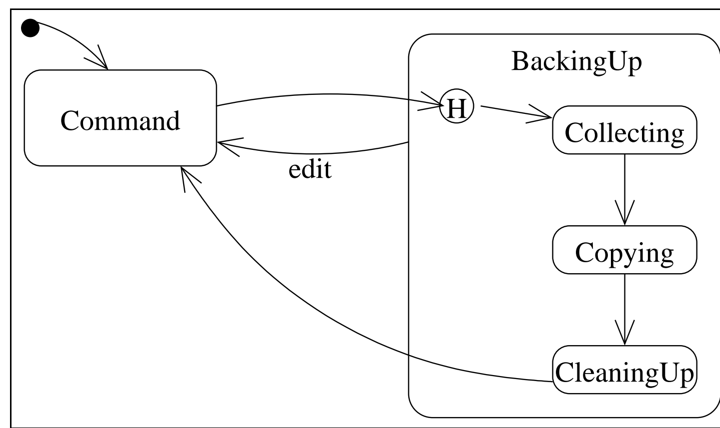

Figura 7.5: Representação de sub-estados com historia na UML (copiado de "*UML User Guide*", p. 301)

Divisão/Sincronização: ("fork/join") Os diagramas podem apresentar fluxos de controle em paralelo com as linhas de divisão e sincronização. A primeira divide um fluxo de controle em vários fluxos paralelos (ex. a linha a baixo do estado Request product na figura 7.6); a segunda reune vários fluxos paralelos em um só (ex. a linha a cima do estado Pay bill na figura 7.6).

Divisão e sincronização funcionam como parênteses, uma sincronização deve sincronizar exatamente os mesmos fluxos que foram divididos por uma divisão.

Notem que no caso de sub-estados paralelos (cf. seção precedente) a divisão é implícita quando entrar no estado composto e a sincronização é implícita quando sair do estado composto.

Raias: ("swimlane") As atividades são conectadas com entidades ou organizações no mundo real (como repartições numa empresa). Essas organizações normalmente são implementadas com classes. Se pode representar essas organizações e as atividades que elas executam com raias (ex. raia Warehouse na figura 7.6).

Fluxo de objeto: Finalmente, as atividades podem criar, modificar ou destruir objetos. Se pode representar essas operações no diagrama como relações de dependência entre a atividade e um objeto. O objeto pode ser apresentado com alguns atributos importantes ou seu estado (cf. a figura 7.6).

7.7 Exercícios

Escolha uma atividade no exemplo da empresa aérea e a modele.

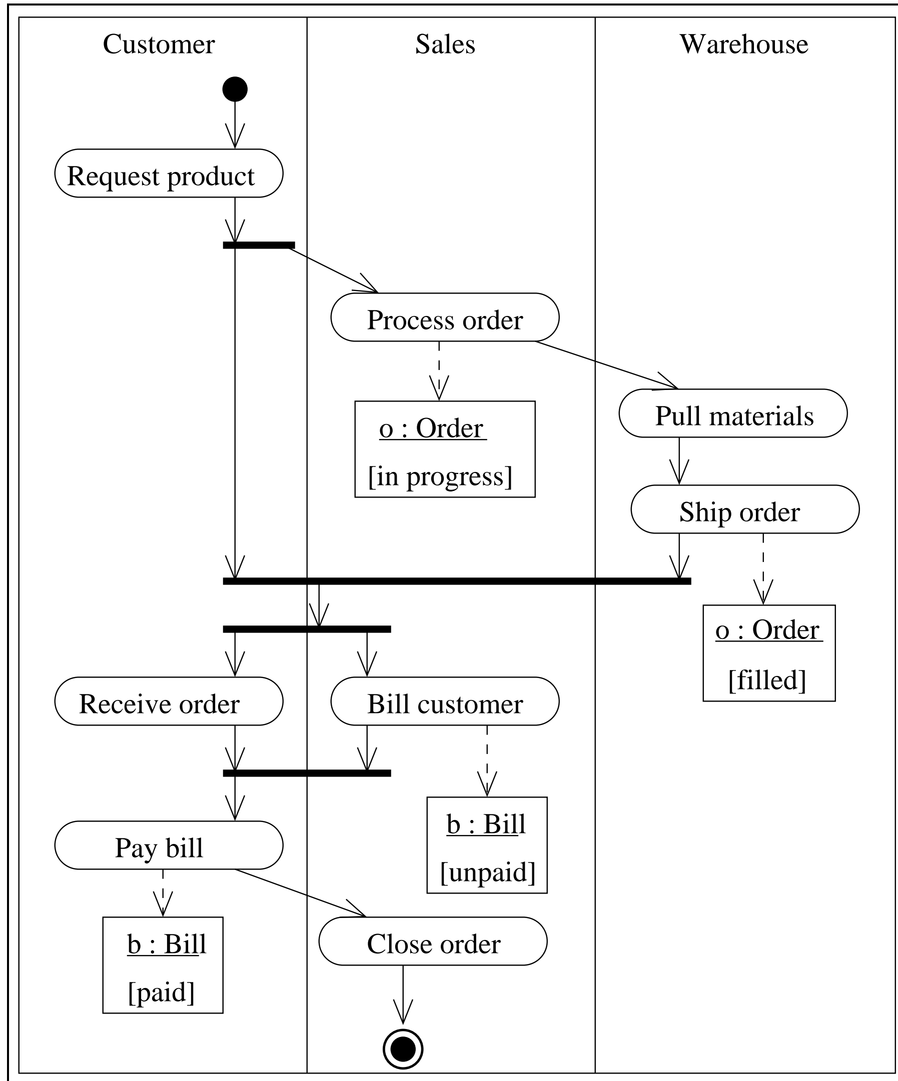

Figura 7.6: Noções avançadas do diagrama de atividade (copiado de “*UML User Guide*”, p. 267)

Capítulo 8

Diagramas de componentes e implementação

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos estudar os diagramas de componentes e de distribuição. Esses diagramas vão permitir especificar a implementação de um sistema (ex: de quais arquivos se compõe um executável) e a sua distribuição sobre várias máquinas. Esses diagramas são usados para ter uma visão completa de um sistema, não somente da parte software, mas também da parte material.

Este capítulo é baseado nos capítulos , 25, 26, 29 e 30 do "UML User Guide" (c.f. referência na introdução do curso).

8.1 Diagrama de Componentes

O diagrama de componentes mostra a organização entre arquivos de código fonte, bibliotecas, tabelas de banco de dados, etc. A relação a mais usada é a de dependência, mostrando como um arquivo de código fonte depende de um outro que ele incluí, o como um executável depende de uma biblioteca por exemplo.

Um componente é um parte física do sistema. Muitas vezes um componente mostra um arquivo específico do sistema.

A UML reconhece cinco estereótipos de componentes:

executável: Um componente que pode ser executado (um programa).

biblioteca: Uma biblioteca de classes ou funções, dinâmica ou estática.

tabela: Uma tabela de um banco de dados.

documento: Uma parte da documentação (texto livre, diagramas, documentos de ajuda, etc.)

arquivo: Outros arquivos, geralmente, se trata de um arquivo de código fonte, mas pode ser tambem um arquivo de dados, um "script" ou outros arquivos.

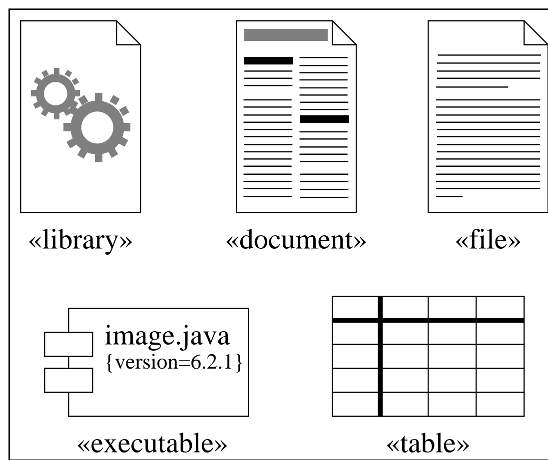

Figura 8.1: Os cinco tipos de componentes com os respectivos ícones na UML

Cada estereótipo tem um ícone definido na UML (ver a figura 8.1). Esses ícones podem ser difícil de desenhar sem ferramentas especializadas. Nessa situação, se pode pensar em apresentar cada componente com um a caixa simples e o estereótipo apropriado.

O livro "UML User Guide" apresenta também sugestões de ícones para outros estereótipos de componentes: Banco de dados, pagina web.

Componentes realizam e/ou dependem de interfaces. Por exemplo, na figura 8.2, o componente "component.java", que corresponde a um arquivo de código fonte Java, implementa (realiza) a interface "ImageObserver". Podemos dizer também que "component.java" *exporta* a interface "ImageObserver". Do outro lado, o componente "image.java" depende da mesma interface, ele *importa* essa interface.

Graças a especificação das interfaces exportadas, componentes são substituíeis: uma biblioteca que realiza uma interface pode ser substituída por uma outra biblioteca realizando a mesma interface.

8.2 Usos comuns de diagrama de componentes

Alguns usos mais comuns do diagrama de componentes são:

- Organizar o código fonte. Por exemplo mostrar todas as dependências entre os arquivos de código fonte. Isso pode ajudar a definir as dependências de compilação (se modificarmos um arquivo, quais outros arquivos precisam ser recompilados).

- Organizar os produtos executáveis. A partir de um conjunto de arquivos (código fonte), é possível criar vários produtos diferentes, com configurações um diferentes. Além disso um sistema não se compõe só de um executável, mas também de tabelas, "script" (por exemplo de inicialização ou de manutenção), etc. O diagrama de componente pode apresentar todas as informações necessárias para configurar um sistema especifico (quais bibliotecas são necessárias, com quais tabelas etc).

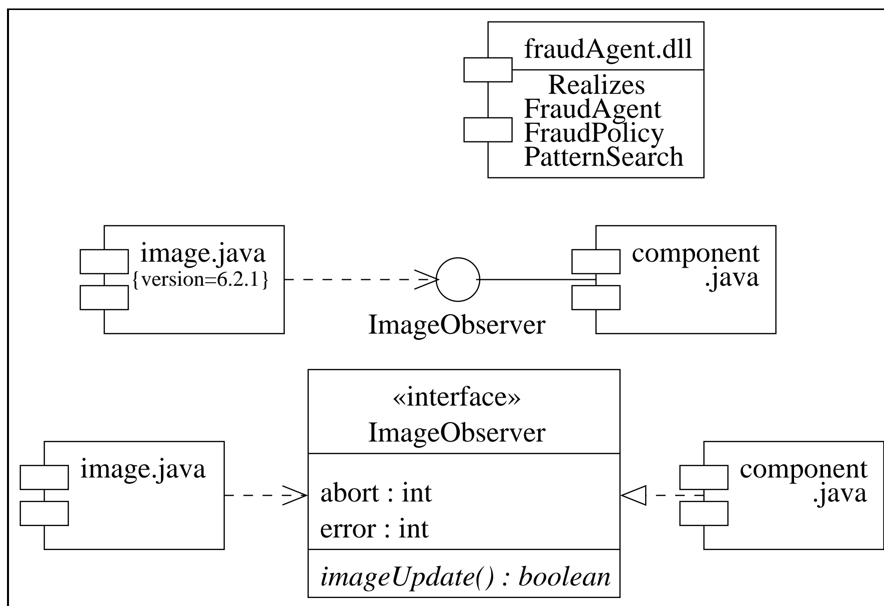

Figura 8.2: Componentes exportando e importando interface (copiado de "*UML User Guide*", p. 348)

Nessa visão, a especificação de qual interface cada componente importa ou exporta é muito importante. Ela permite averiguar que toda interface importada por um componente do produto é exportada por um outro componente.

A figura 8.3 propõe um exemplo dessa utilização do diagrama de componentes.

- Modelar um banco de dados. Um banco de dados se compõe de várias tabelas. Também com a chegada dos "data warehouse" (armazém de dados), pode ser nécessario de reorganizar tabelas de vários banco de dados dentro de um armazém virtual. Um diagrama de componente pode apresentar a agregação de várias tabelas dentro de um banco de dados.

8.3 Diagrama de implantação

O diagrama de implantação é usado para sistema distribuídos. Ele permite apresentar a topologia de uma rede de "máquinas" e qual processo (um componente executável) cada "máquina" vai rodar.

As "maquinas" são chamadas de *nos*. Um no apresenta uma fonte computacional, sendo normalmente um processador com alguma memória. Um no é representado por um cubo (ver figura 8.4). A relação entre um no e um componente que "roda" sobre esse no é apresentada como uma relação de dependência.

Alguns usos mais comuns do diagrama de distribuição são:

- Modelar um sistema "embarcado" ("embeded system"). Esse tipo de sistema inclui sensores que vão receber informações do mundo exterior e partes agentes que vão agir

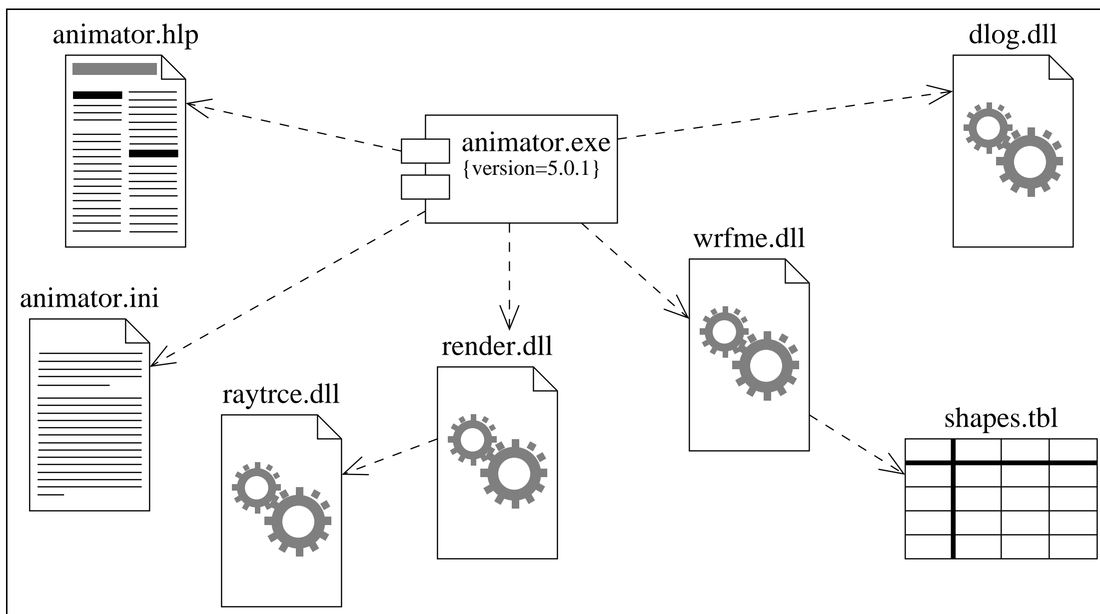

Figura 8.3: Exemplo de um diagrama de componentes (copiado de "*UML User Guide*", p. 354)

sobre o mundo exterior. Cada parte precisa de "drivers" e software especializado para funcionar com o resto.

- Modelar sistemas distribuídos como no exemplo da figura 8.4, onde vários nos são inter-conectados numa rede de máquinas.

- Modelar um sistema cliente/servidor. Esse tipo de sistema é um tipo particular de sistema distribuído onde as máquinas são classificadas em clientes, que interagem com o exterior (normalmente, os usuários), e os servidores que realizam os cálculos e o armazenamento dos dados e interagem com os clientes.

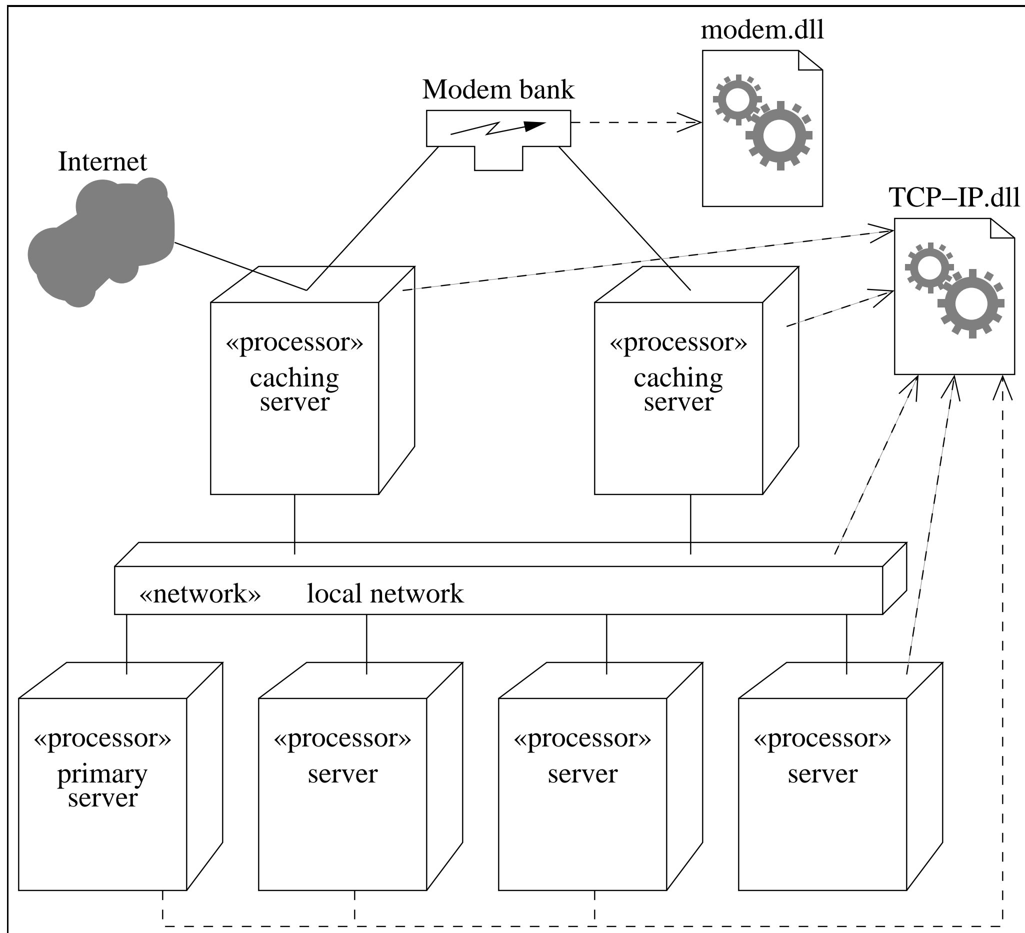

Figura 8.4: Exemplo de um diagrama de distribuição (copiado de "*UML User Guide*", p. 408)

Capítulo 9

O processo de desenvolvimento de software

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos introduzir o processo de desenvolvimento de software RUP. Este processo é dirigido pelos casos de uso que já vimos num capítulo precedente.

Vamos descrever as principais características do processo de desenvolvimento RUP: ser dirigido por casos de uso, ser baseado na arquitetura, ser iterativo e incremental. Vamos apresentar as atividades desse processo (levantamento dos requisitos, analise, projeto, implementação e teste) e as fases (concepção, elaboração, construção e transição).

Este capítulo é baseado nos capítulos 1 a 5 do livro "The Unified Software Development Process" (c.f.referência na introdução do curso).

9.1 Introdução aos processos de desenvolvimento

Um processo de desenvolvimento de software, começa com um problema que um usuário quer resolver e acaba com um sistema que resolve este problema. O sistema é composto de: programas, documentação, bancos de dados, etc.

Um processo de desenvolvimento de software apresenta bons e maus hábitos de trabalho para desenvolver um sistema. Ele descreve em detalhes, quem deveria fazer quais operações, quando e como.

Com o tempo, os hábitos de trabalho evoluem e novas soluções são descobertas para fazer o trabalho mais rápido ou com mais segurança. Processos de desenvolvimento têm que se adaptar e se renovar para sempre disponibilizar as melhores soluções conhecidas e contribuir difundir e generalizá-las a toda a comunidade.

Um processo de desenvolvimento de software define um *ciclo de vida* do software, o que quer dizer todas as atividades durante a vida do software, desde o início, quando tivermos apenas a idéia do que queremos desenvolver, até o fim, quando o sistema não for mais útil e for abandonado. Por exemplo o primeiro modelo de ciclo de vida a ser formalizado foi o *modelo em cascata*. Ele propõe as seguintes atividades:

Análise: Cujo objetivo principal é entender o problema dos usuários e as restrições que podem influir o desenvolvimento (tempo, dinheiro, maquinas, etc.).

Um outro objetivo é de planificar o projeto de desenvolvimento, avaliar os custos, etc.

Projeto: Cujo objetivo é definir o sistema: algoritmos, interfaces, procedimentos de utilização, etc.

Implementação: Cujo objetivo é implementar (em geral: por programação) o sistema definido na atividade precedente.

Um outro objetivo aqui é de verificar o bom funcionamento dos programas (testes).

Manutenção: Cujo objetivo é de realizar a evolução do sistema para corrigir bugs, acrescentar novas funcionalidades, adaptá-lo às novas tecnologias, etc.

As características desse modelo são que:

- Ele define sub-objetivos para atingir o objetivo final.
- Ele é seqüencial, as atividades se desenvolvem uma depois da outra.
- Ele tem o conceito de *produto entregavel* ("deliverable"). Um produto entregavel é um documento, um programa ou um outro produto de software. Ele é o (um dos) resulto(s) de uma atividade. Produtos entregáveis são verificados e avaliados antes de passar à atividade seguinte.

A idéia dos sub-objetivos é de facilitar o processo de desenvolvimento, já que esse é muito difícil.

A idéia de produtos entregáveis é de não deixar o cliente esperar até o fim do processo para lhe dar o sistema final sem deixar ele intervir. Ao fim de cada atividade, o cliente pode avaliar os produtos entregáveis e decidir de continuar o projeto na mesma direção ou não. Os coordenadores da equipe de desenvolvimento podem, também, usar esses produtos entregáveis para verificar se o trabalho estar progredindo normalmente e dentro dos prazos definidos.

Os vários processos diferenciam-se pela decomposição em atividades que eles propõem (eles podem detalhar mais ou menos cada atividade) e o fluxo de atividade de uma atividade para outra.

Os maiores defeitos do modelo em cascatas são:

- A seqüêncialidade das atividades que não permite voltar para corrigir erros: erros são propagados nas atividades seguintes.
- A "unicidade" do processo que deixa os problemas aparecer só ao fim do processo quando for tarde demais para fazer qualquer coisa.

  Se acontecer um problema importante, o projeto inteiro corre o risco de ser cancelado.

Alguns processos mais recentes (ex.: modelo em espiral ou o processo da UML), tentam resolver esses problemas com modelos em que o sistema não é desenvolvido em uma vez só, mas é decomposto em sub-partes com repetição das várias atividades para cada sub-parte. Passamos de um modelo linear a modelos iterativos, "em duas dimensões": a dimensão das atividades, semelhante ao modelo em cascatas, e a dimensão das "sub-partes" (que vamos chamar de iterações).

Esses processos usam uma extensão dos princípios de sub-objetivos e produtos entregáveis (decomposição do trabalho e interação com os usuários):

- Porque tem mais produtos entregáveis (um para cada atividade de cada sub-parte), tem mais possibilidades de interação entre os usuários e o desenvolvedor.
- Porque as sub-partes são projetos mais pequenos, se torna mais fácil desenvolvé-las.

Esses processos têm outras vantagens que vamos discutir mais tarde nesse capítulo: diminuição dos riscos, possibilidade de corrigir erros nas atividades iniciais, etc.

9.2 O processo da UML

O processo proposto na UML tem as seguintes atividades:

Levantamento dos requisitos: Descrição do problema do ponto de vista do usuário.

Análise Descrição do problema do ponto de vista do desenvolvedor.

Projeto: Concepção da solução.

Implementação: Programação da solução.

Teste: Verificação do programa.

Mas o processo tem, também, uma segunda dimensão de fases: concepção, elaboração, construção e transição (c.f. a figura 9.1). As quatro fases são detalhadas na seção "Processo iterativo".

O processo é iterativo, o que significa que vamos "executar" essas cinco atividades várias vezes para um mesmo sistema. Cada repetição será considerada um mini-projeto de desenvolvimento (mas os mini-projetos não são independentes um do outro). Dentro de um determinado mini-projeto, as atividades são seqüenciais como no modelo em cascata.

Notem que o processo não tem fase de manutenção. Cada manutenção é considerada como um projeto de desenvolvimento e comporta as cinco atividades (com várias iterações).

O processo tem as seguintes características: dirigido pelos casos de uso, baseado na arquitetura, iterativo e incremental.

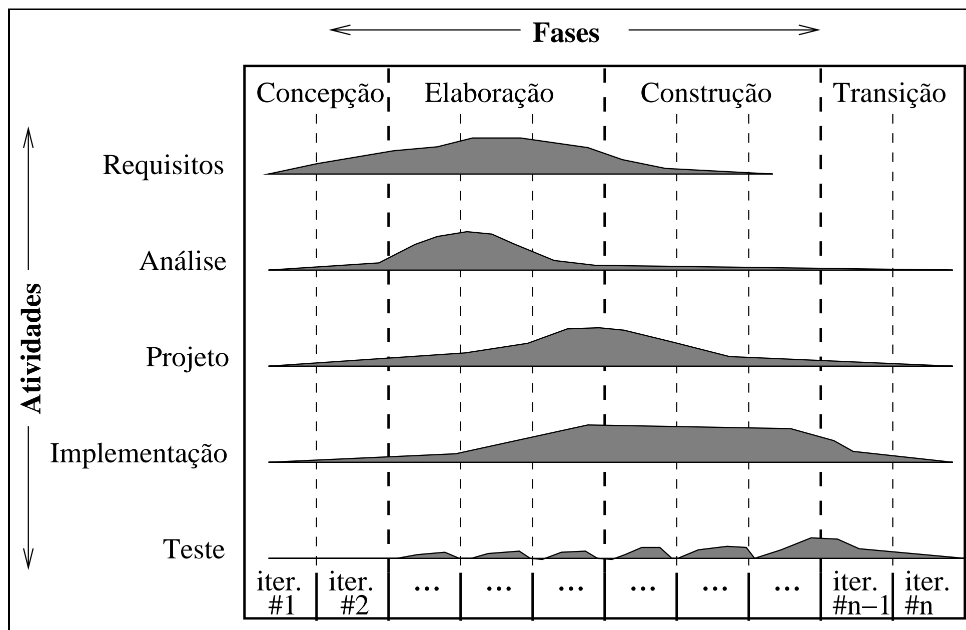

Figura 9.1: As atividades, fases e iterações do processo de desenvolvimento da UML (copiado de "*Unified Software Development Process*", p. 11)

9.3 Processo dirigido pelos casos de uso

Os diagramas de interação, podem ser rastreados até os casos de uso que eles realizam ("traceability"), classes e operações podem ser rastreadas a esses diagramas ou aos casos de uso. Assim, o código pode ser rastreado a um requisito (caso de uso).

Com a decomposição do projeto inteiro em casos de uso (nos passos de análise, projeto, implementação e teste), é possível, desde o início, avaliar o trabalho a ser feito pelo menos em número de tarefas: cada caso de uso representa várias tarefas (análise, projeto, implementação, teste). É possível também decompor o projeto inteiro em partes a assinar para desenvolvedores.

Casos de uso explicitam os requisitos funcionais, mas podem, também, ser usados para exprimir requisitos não funcionais. Por exemplo, podemos especificar que um caso de uso tem que se "executar" em menos de 2 minutos.

Levantamento dos requisitos: Casos de uso ajudam achar requisitos verdadeiros e permitem representá-los numa forma que os usuários podem entender.

Uma das inovações do processo com caso de uso é que em vez de perguntar "Qual operação o sistema deve cumprir?", ele pergunta "Qual operação o sistema deve cumprir para um usuário?". A primeira questão é o que uma análise "tradicional" tenta responder. Como já vimos várias vezes, a diferença entre a nova questão e a velha não parece importante, apenas três palavras a mais. Mas essa pequena diferença na forma marca uma diferencia maior na concepção da tarefa. Essas três palavras significam que não vamos procurar qualquer funcionalidade abstrata que seria interessante para o sis-

tema cumprir, mas vamos procurar funcionalidades concretas que um usuário precisa realmente. Isso significa que o processo vai ser mais focalizado, e então, mais eficiente.

Casos de uso são, também, intuitivos e podem ser entendidos diretamente por usuários. Isso lhes da acesso ao processo de desenvolvimento.

Análise, projeto e implementação: Vamos estudar cada caso de uso independentemente para achar classes que vão ajudar realizar os casos. Vamos assinar responsabilidades definidas nos casos de uso às classes. Casos de uso vão ajudar decidir em qual ordem desenvolver e implementar cada componente.

Teste: Podemos verificar diretamente que o sistema pode efetuar normalmente todos os casos de uso. Testar um sistema sempre foi testar casos de usos (mesmo se o nome não existia). A novidade é que podemos planejar os testes desde o início do desenvolvimento e acertar isso com os usuários se for necessário. Esse tipo de teste (baseando só sobre o que o sistema tem que fazer, sem saber como ele faz) se chama de teste caixa-preta ("black-box test"). Depois da atividade de projeto, podemos definir testes caixa-branca ("white-box test") que vão testar algumas particularidade da implementação.

9.4 Processo baseado na arquitetura

A arquitetura é uma descrição *global* do sistema (a ser desenvolvido) que deve ajudar intendé-lo inteiramente. Ela vai especificar a estrutura do sistema sem detalhes. A arquitetura é uma descrição do sistema que não vai mudar durante a vida dele (ou pelo menos, mudara pouco e muito devagar). Por isso, é importante definir uma arquitetura que poderá aceitar e mesmo facilitar as várias modificações que o sistema vai sofrer durante sua vida.

A arquitetura é composta de visões dos diferentes modelos (modelo de caso de uso, modelo de análise, modelo de projeto, ...). Uma visão de um modelo é uma simplificação desse modelo que contem só os elementos mais importantes como classes, sub-sistemas, dependências, colaborações, ...

Os objetivos da arquitetura são:

- Ajudar todos os participantes a entender o sistema *integralmente*.
- Definir as interfaces dos sub-sistemas para permitir desenvolvé-los independentemente um do outro.
- Ajudar os desenvolvedores a achar os componentes (por exemplo para reutilização).
- Resistir às várias mudanças que o sistema sofrera.

A arquitetura deve promover princípios gerais como:

Modularidade: O sistema é decomposto em módulos independentes, isso permite de configurar o sistema ativando ou desativando vários módulos.

Separação dos objetivos: A interface de um sistema pode ser implementada em elementos diferentes do que as funcionalidades básicas (os serviços). Esse permite modificar partes importantes do sistema (por exemplo: re-implementar a interface) sem re-escrever tudo, ou de substituir uma parte por uma outra (com mesma funcionalidade) sem modificar o comportamento.

Localização dos serviços: Cada funcionalidade do sistema deve ser implementada com um componente só. Isso facilita a distribuição do sistema sobre várias maquinas.

etc.

Normalmente não é possível conseguir uma arquitetura com todas essas qualidades, os desenvolvedores (junto com os usuários) devem escolher as mais desejadas.

A arquitetura e os casos de uso são desenvolvidos ao mesmo tempo. Primeiro, se define uma arquitetura simples baseada sobre o que sabemos do domínio (talvez um “framework”), depois vamos estudar se os casos de uso são compatíveis com essa arquitetura. Caso contrario, devemos ou mudar a arquitetura ou redefinir o caso de uso.

Por exemplo, podemos decidir por uma implementação em camadas (cada camada implementa funcionalidade mais básicas que as camadas superiores e mais abstratas que as camadas inferiores, e uma camada usa os serviços só das camadas inferiores) mas descobrir que isso não é compatível com um caso de uso (talvez porque esse tipo de arquitetura é geralmente lenta).

Os casos de uso serão testados começando com os mais arriscados e mais importantes pelo cliente.

Alguns padrões de análise:

“Broker”: Usado quando tiver objetos distribuidos. Um objeto local é usado como interface para pedir serviços a um objeto distante. Para o objeto usuários do objeto local, a comunicação com o objeto distante é transparente.

Cliente/Servidor: O servidor fornece um serviço usado por vários clientes. Também útil quando tiver um sistema distribuido.

Camadas: O sistema é implementado em camadas de níveis de abstração diferentes. As camadas mais baixas implementam pequenos utilitários independentes da aplicação. As camadas mais altas implementam funções mais abstratas. Uma camada só pode usar os serviços das camadas mais baixas. Esta decomposição facilita a implementação de funções complexas, e a reutilização das camadas baixas. Pode ser usada, também, quando uma camada é implementada com um software comprado. Em geral, essa arquitetura vai prejudicar a rapidez do sistema.

Decomposição vertical: Os sub-sistemas correspondem a funcionalidades bem definidas (por exemplo, interface, tratamento, persistência, ...). A diferença das camadas é que um único sub-sistema implementa funcionalidades de níveis de abstração muito diferentes. Por exemplo, o sub-sistema de interface, vai gerar o teclado e a tela, também como fornecer funcionalidades mais abstratas (sistema com janelas, telas do sistema, ...).

etc.

É, também, juntar dois (ou mais) padrões. Por exemplo, o padrão em camadas é o mais comum, ele pode ser usado para implementar as camadas baixas enquanto um outro padrão é usado em vez das camadas mais altas.

9.5 Processo iterativo e incremental

O processo da UML é iterativo, o que significa que vamos repetir as cinco atividades do ciclo de vida (ver seção "Processo da UML") várias vezes. Cada seqüência das cinco atividades é chamada de *iteração*. O desenvolvimento de um sistema será composto de várias iterações, cada uma sendo um pequeno projeto, mas esses pequenos projetos não são independentes um do outro.

Dentro de uma iteração, as atividades são seqüenciais como no modelo em cascata. Ao fim da iteração vamos avaliar seus resultados para definir se novos requisitos foram descobertos, se requisitos já conhecidos foram modificados, ou se a lista dos riscos é ainda a mesma. Essa avaliação vai nos ajudar a decidir se podemos seguir com a próxima iteração ou se devemos replanejar as iterações futuras.

A organização iterativa, permite tratar os riscos do desenvolvimento melhor. Por isso, temos primeiro que identificar os riscos e ordenar eles por nível de perigo (início do desenvolvimento). Os riscos mais importantes serão tratado ao início do desenvolvimento. Assim se não pudermos resolver eles, poderíamos reorientar o projeto antes de ter gastado muito tempo e dinheiro. O assunto dos riscos é muito importante para esse processo, e voltaremos a ele numa outra aula.

Um problema de um processo iterativo é que depois de cada iteração temos que avaliar se a última iteração não introduziu erros no que já existia antes (iterações anteriores). Isso se chama de "regression testing". Quanto mais vamos avançar no desenvolvimento, mais essa avaliação será importante.

As iterações pertencem a diferentes *fases* (c.f. a figura 9.1): concepção ("inception"), elaboração, construção e transição. Cada fase tem uns objetivos diferentes:

Concepção: Compreensão do problema, e definição precisa do que o sistema deve fazer. Nesta fase, vamos, também, avaliar a pertinência do projeto: será que o sistema é realmente necessário? Os riscos mais importantes do projeto deveram ser identificados e resolvidos.

Elaboração: Criação da arquitetura e levantamento da maioria dos requisitos. Vamos, também, resolver os riscos os segundos mais importantes. No final da fase, podemos estimar o custo total do projeto e planejar o desenvolvimento.

Construção: Desenvolvimento do sistema e começo da transição para os usuários. Eles devem ter acesso às funcionalidades mais importantes.

Transição: Entrega do produto aos usuários (corrigir os bugs, formação dos usuários)

Essas quatro fases não são as mesmas coisas que as cinco atividades (requisitos, análise, projeto, implementação e teste). Mas existe uma forte correlação entre atividades e fases.

- Na fase de concepção, vamos estudar a possibilidade de fazer o projeto, a parte principal será de levantamento dos requisitos para delimitar o projeto.
- Na fase de elaboração, vamos continuar ter atividades de levantamento dos requisitos e vamos ter, também, as partes de analise (mais forte nas iterações do início da fase) e de projeto (mais forte nas iterações do fim da fase).
- Na fase de construção, as atividades as mais importantes vão ser as de implementação e de teste (ao fim de cada iteração).
- Finalmente, na fase de transição, vamos ter, de novo, importantes partes de implementação e de teste, mas a implementação já não será tão grande como na fase precedente.

Mas cada fase vai ter as cinco atividades (c.f. a figura 9.1). Por exemplo, depois da elaboração, a maioria dos casos de uso serão definidos, mas não todos, alguns menos importantes podem ser ainda especificados na fase de construção, o que quer dizer que vai existir uma pequena atividade de levantamento dos requisitos nessa fase.

Concepção e elaboração são fases mais curtas, feitas com uma equipa pequena, elas vão custar menos dinheiro. O projeto pode ser "facilmente" abandonado durante ou depois dessas fases se for julgado perigoso demais ou não suficiente interessante. Nessas duas fases vamos apenas definir o projeto, avaliar os riscos mais importantes e definir o calendário de desenvolvimento.

A partir da fase de construção, a equipe de desenvolvimento vai ser maior e a fase mesma vai demorar mais tempo. As pessoas da equipa de base (duas primeiras fases) podem ajudar muito em integrar novos membros. Elas já conhecem a metodologia (eles fizeram algumas iterações nas duas primeiras fases) e, também, conhecem o problema por ter trabalhado sobre ele nas duas primeiras fases.

9.6 Pessoas

Os processos de desenvolvimento, propõem atividades, fases, documentos, etc. a ser executados ou criados para o bom desenvolvimento do sistema. Mas esses processos não ensinam como tratar uma parte muito importante do desenvolvimento: as pessoas.

Esses processos só apresentam uma situação ideal onde todo mundo trabalha para o sucesso do projeto. Muitas vezes, a realidade é diferente.

Problemas de personalidades não podem ser tratado nos processos, mas vocês devem saber que são muito freqüentes. Podem ser pequenos problemas (você acha seu chefe um idiota, mas o trabalho está feito) ou podem ser problemas muito difícil a resolver (um usuário ou um desenvolvedor tenta sabotar o projeto).

Os problemas podem surgir na equipe de desenvolvimento mesma, por exemplo conflito entre duas pessoas, ou falta de motivação. Eles podem acontecer entre os desenvolvedores e

os usuários ou podem surgir dos próprios usuários. Por exemplo um chefe resolveu ter um novo sistema, mas os usuários verdadeiros não o querem. Um problema normal, que pode ser importante ou não, é uma resistência à mudança, um medo de tudo o que é novo (novo sistema vai significar novos processos de trabalho, novas máquinas desconhecidas, talvez mudança de escritório e, então, novos colegas, novo contexto de trabalho, ...). Um outro problema, ainda, pode ser um "excesso de colaboração", quando um usuário pensar que ele conhece informática e quer "ajudar" os desenvolvedores.

É importante ter consciência disso para descobrir os problemas o mais cedo que for possível enquanto é ainda possível solucioná-los.

Não tem solução miraculosa para esse tipo de problema, mas existem alguns problemas na equipe de desenvolvedores que o processo da UML pretende resolver:

- Pessoas não gostam de trabalhar sobre um projeto que parece impossível. O processo da UML pretende avaliar a viabilidade de um projeto cedo (depois das fases de concepção e elaboração).

  Um projeto viável supõe também que os prazos são razoáveis. De novo, o processo da UML pretende facilitar essa parte.

- Pessoas gostam de ver o trabalho progredir. No processo da UML, cada iteração resulta em alguns produtos entregáveis que ajudam nesse sentido.

- Pessoas trabalham melhor em pequenos grupos (menos de 6 ou 8 pessoas). A decomposição em casos de uso pode ajudar definir tarefas suficiente pequenas para isso.

A abordagem com casos de uso pretende também dar mais importância aos usuários, um caso de uso é definido por e para um usuário. Isso deve permitir facilitar a integração deles no processo de desenvolvimento e portanto facilitar a aceitação do sistema.

Capítulo 10

As atividades do processo

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos apresentar as atividades (levantamento dos requisitos, análise, projeto, implementação e teste) do processo de desenvolvimento de software RUP que formam uma iteração. Vamos, também, listar os artefatos que elas produzem.

A descrição de cada atividade pode não ser completa, o objetivo desse capítulo é de dar uma visão global de uma iteração do processo de desenvolvimento RUP, e não de apresentar todos os detalhes das atividades. Vamos, também, passar mais rapidamente sobre as últimas atividades (implementação e teste) que apresentam poucas novidades.

Neste capítulo, vamos falar várias vezes do desenvolvimento de um *programa*. Usamos essa palavra para lembrar que estamos descrevendo aqui uma iteração só do processo de desenvolvimento de um *sistema*. O processo inteiro vai conter várias iterações e conseqüentemente várias vezes cada atividade.

Este capítulo é baseado nos capítulos 6 a 11 do livro "The Unified Software Development Process" (c.f. referência no capítulo "Introdução" do curso).

10.1 Atividade de levantamento dos requisitos

O objetivo da atividade de levantamento dos requisitos é de definir qual programa é preciso desenvolver. Por isso vamos definir os requisitos do programa, quer dizer o que ele deve fazer e com quais restrições.

Esta atividade vai resultar em quatro artefatos:

- Um modelo do negócio ou um modelo do domínio para representar o contexto do sistema.
- Um diagrama de casos de uso para representar os requisitos que dependem de um caso de uso só.
- Protótipos das interfaces com os usuários (por exemplo telas).
- Uma lista dos requisitos mais gerais, que não dependem de um caso de uso só.

Uma dificuldade dessa atividade é que ela deve resultar numa especificação dos requisitos que seja compreensível por os próprios usuários. Isso significa que temos que usar a linguagem deles. Seremos limitados na possibilidade de formalizar essa especificação porque de maneira geral, a linguagem do usuário não é adaptada ao tipo de formalização que engenheiros de computação precisam.

O levantamento dos requisitos é difícil por que deve ser feita pelos usuários que são os únicos quem sabem o que precisam. Mas eles não são especialistas em informática, eles não sabem qual parte dos seus trabalhos pode ser implementada ou como o computador pode lhes ajudar. Outro problema é que cada usuário tem uma visão restrita do domínio e do que o programa precisa fazer, vamos precisar unificar essas visões parciais em uma grande visão do programa. Finalmente os requisitos mudam: as maquinas evolvam, novos métodos de trabalho são introduzidos, leias mudam, etc.

A atividade de levantamento dos requisitos é realizada principalmente nas iterações das fases de concepção, elaboração e construção.

- Na(s) iteração(ões) da fase de concepção, a atividade de levantamento dos requisitos vai principalmente identificar a maioria dos casos de uso e detalhar os mais importantes (menos de 10% do total). Identificamos (quase) todos os casos de uso para delimitar o projeto e o programa, e para puder identificar os mais importantes.

- Nas iterações da fase de elaboração, a atividade de levantamento dos requisitos vai identificar o resto dos requisitos (a partir dos 90% de casos de uso menos importantes). Ao fim dessa fase, deveríamos ter descrevido quase todos os casos de uso, ter identificado uma grande parte de todos os requisitos (80%), e ter implementado na arquitetura uma pequena parte deles (5% ou 10%).

- Nas iterações da fase de construção, a atividade de levantamento dos requisitos vai identificar e implementar na arquitetura o resto dos requisitos.

- Nas iterações da fase de transição, não tem a atividade de levantamento dos requisitos.

10.1.1 Requisitos

Os requisitos são a definição de o que os programa precisa fazer dentro de quais restrições. Isso inclui as funcionalidades que o programa precisa cumprir (ex: fazer uma reserva, procurar um vôo), que se chamam de requisitos funcionais. Requisitos não funcionais são outras restrições introduzidas pelos usuários, por exemplo, sobre o tempo de execução, o formato das interfaces, introduzidas pelo material, por exemplo, o tamanho máximo do código ou dos dados, ou que são derivadas de vários casos de uso, por exemplo, necessidade de criar um banco de dados central para armazenar todas as reservas e todos os vôos.

Os requisitos funcionais descrevem o que o programa precisa fazer. Os requisitos não funcionais ponham restrições sobre como o programa deve funcionar.

Claramente, requisitos funcionais podem ser levantados com ajuda de casos de uso porque cada caso descreve como realizar uma operação com o programa.

As coisas são mais difíceis com os requisitos não funcionais. Alguns deles são específicos para um caso de uso só. Por exemplo, poderíamos especificar que um caso de uso tem que

se “executar” em menos que 3 minutos. Isso é um requisito não funcional (ele não especifica uma função que o programa deve cumprir) que pertence a um caso de uso só. Especificando o caso de uso, será possível especificar também esse requisito não funcional (ver a categoria “Outros requisitos” no padrão proposto no capítulo sobre casos de uso), por que pensar na execução da tarefa do caso de uso, o usuário vai provalmente lembrar, também, deste requisito.

Para os requisitos que não pertencem a um caso de uso só, vamos manter uma lista de requisitos candidatos. Em cada iteração, vamos rever a lista e decidir se podemos tratar alguns requisitos ou cortar alguns. Essa maneira é o modo “tradicional” de identificar requisitos.

10.1.2 Contexto do sistema

Para achar os requisitos corretos, precisamos conhecer o contexto do (futuro) sistema. Temos dois soluções por isso: Modelagem do domínio (“domain modeling”) e modelagem do negócio (“business modeling”).

O modelo do negócio descreve as operações dentro da impressa e quem é responsável para efetuar elas. O modelo de negócio é definido com casos de uso e um diagrama de classe.

O modelo do domínio descreve os conceitos importantes do domínio e as relações entre eles. Ele é definido com um diagrama de classe. Esse modelo pode ser considerado uma sub-parte do modelo do negócio.

É importante lembrar que o objetivo de ambos modelos é entender o domínio, o contexto do sistema e não especificar uma solução possível ao problema. Esses modelos devem ser de um nível alto de abstração e relativemente independentes do problema considerado.

O modelo de domínio descreve os conceitos mais importantes. Ele deve ser pequeno (10 a 50 classes). Outros conceitos achados mas não representados no modelo podem ser listados num pequeno glossário do domínio.

O glossário e o modelo vão ajudar todos os participantes (desenvolvedores e usuários) a falar a mesma linguagem e assim se entenderem. Isso vai, também, identificar e cortar as ambigüidades que podem existir entre vários sub-domínios (que normalmente não se “encontram”, mas serão representados juntos no sistema).

A segunda parte do modelo do negócio (casos de uso) descreve as interações entre os atores do domínio.

10.1.3 Desempenho do levantamento dos requisitos

A atividade de levantamento dos requisitos produz os artefatos seguintes:

Lista dos atores: Obrigatório. Cada ator deve ter uma pequena descrição.

Casos de uso: Obrigatório.

Interface com usuário: Facultativo. A interface aqui é um protótipo (por exemplo, telas só) para ajudar o usuário entender como o programa vai funcionar.

Descrição da arquitetura: Obrigatório. A arquitetura não é completa ainda, mas ela inclui uma descrição dos casos de uso mais importantes. É essa parte da arquitetura que será criada aqui.

Glossário: Fortemente recomendado. Vai ajudar todos os participantes (desenvolveres como usuários) a falar a mesma linguagem. Já falamos da importância de bem escolher os nomes em qualquer modelo. Uma outra coisa importante é ter uma definição precisa, explicita e comum de cada conceito. Projetos já falharam por falta desse artefato.

Esses artefatos serão desenvolvidos com as operações seguintes:

- Achar atores e casos de uso, com as sub-operações:
  - Achar atores. Cada trabalhador da organização será um ator potencial do programa. Atores devem ser tanto independentes quanto for possível. Podem ser procurados a partir de um modelo de negócio.
  - Achar casos de uso. Para cada ator, vamos procurar o que o programa precisa fazer com entrevistas com ele, baseado sobre as suas tarefas.
  - Descrever casos de uso.
  - Criar o diagrama de casos de uso. Essa operação inclui definir o glossário.

  As sub-operações não são necessariamente seqüenciais, podemos começar com um ator, procurar seus casos de uso, descrever eles, e depois voltar a achar novos atores. Ou podemos construir o diagrama de caso de uso ao menos que achamos os casos de uso.

- Identificar prioridade de cada caso de uso. Os casos de uso importantes serão integrados na arquitetura. As prioridades serão estabelecidas considerando vários aspectos: técnicos, econômicos, de negócio, etc.

- Descrever os casos de uso mais importantes para essa iteração.

- Criar protótipos das interfaces com os usuários. Podemos começar com uma definição lógica da interface: quais informações ela deve conter e como elas são relacionadas. Depois vamos transformar isso numa interface física.

- Re-estruturar o diagrama de casos de uso. Vamos rever o modelo já definido (ver acima) e procurar reestruturar ele, talvez com introdução de novos casos de uso compartilhado por vários outros casos (criação de inclusões), ou vamos integrar o novo diagrama de casos de uso com um já existente. Mas é, também, importante de não decompor os casos de uso demais, de maneira que eles permanecem coomprensíveis para os usuários. Um caso de uso deveria descrever uma operação “natural” para eles.

10.2 Atividade de análise

O objetivo da atividade de análise é de criar uma descrição conceitual do problema do ponto de vista dos desenvolvedores. Depois da primeira atividade, requisitos são descritos na linguagem dos usuários, que não permite resolver algumas questões importantes para os desenvolvedores. Por exemplo:

- Os casos de uso devem ser independentes um do outro, não tratamos ainda os problemas da integração deles num programa, o que vai acontecer quando dois casos de uso tentaram acessar a recursos que não podem ser compartilhados, ou se um usuário tenta executar dois casos de uso que não são compatíveis, um depois do outro?
- A linguagem do usuário não permite descrever as interações com um nível suficiente de detalhes para implementá-las.
- Um caso de uso descreve uma operação que seja natural e compreensível pelos usuários, mas para a implementação, é melhor decompor em unidades menores (funções por exemplo).

A análise vai permitir começar a colocar o foco sobre o desenvolvimento guardando a rastreabilidade até os casos de uso iniciais.

Esta atividade vai resultar em quatro artefatos:

- Pacotes de análise. Serão úteis para decompor o programa em sub-sistemas. Esses sub-sistemas (pacotes) devem constituir uma decomposição funcional do programa (baseadas sobre as funções que queremos disponibilizar).

  Alguns desses pacotes serão pacotes de serviços, o que quer dizer que eles vão especificar funcionalidades úteis por várias interações (casos de uso). A diferencia com os requisitos e que a decomposição pode ser mais fina, um pacote de serviços representa uma operação que não é natural pelos usuários porque eles sempre fazem isso dentro de algumas outras operações mais abstratas.
- Classes de análise. Ver descrição a baixo.
- Realização de casos de uso—análise. Cada caso de uso será especificado com mais detalhes, o que o processo chama de “realizar” o caso de uso. Na análise, isso significa que vamos especificar diagramas de classes (com classes de análise, por exemplo, ver a figura 10.1) e/ou diagramas de interação e/ou uma descrição textual mais completa (e com a linguagem dos desenvolvedores desta vez, por exemplo, com referências às classes de análise) e/ou novos requisitos não funcionais.
- Descrição da arquitetura—análise. A descrição da arquitetura se compõe das partes mais importantes dos vários modelos. Para o modelo de análise, as partes importantes vão incluir: decomposição em pacotes de análise, realização de casos de usos e classes de análise mais importantes (por exemplo as classes centrais que tem associações com muitas outras, ou classes associadas com casos de uso julgados importantes) e finalmente realizações (ex.: diagrama de interação) dos casos de uso importantes.

O modelo de análise (o resultado dessa atividade) pode ser considerado temporário ou não, o que significa que ela vai ajudar melhorar nossa compreensão do problema, mas depois pode ser abandonado e começaremos a atividade de projeto sem usar os modelos criados aqui

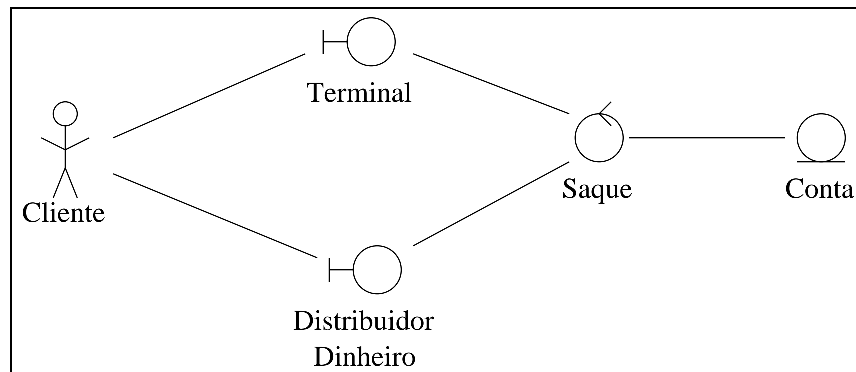

Figura 10.1: Um diagrama de classe (de análise) para a realização de o caso de uso Tirar dinheiro (copiado de "*Unified Software Development Process*", p. 47)

10.2.1 Classes de análise

Classes de análise são uma etapa intermediária entre os atores e casos de uso e as classes do projeto. Elas representam conceitos mais detalhados do que só os atores e o programa, mas elas, provalmente, não serão implementadas diretamente.

Classes de análise vão se concentrar nos requisitos funcionais, os não funcionais serão tratados na próxima atividade.

Vamos especificar só as responsabilidades das classes, não as operações. De maneira semelhante, os atributos serão muito abstratos.

Podemos classificar as classes de análise em três tipos gerais:

Classe de fronteira: Representa uma interação com um ator, suas responsabilidades serão de apresentar algumas informações ao ator e receber informações dele. Por exemplo, para o caso de uso Tirar saque, poderíamos ter duas classes de fronteira, uma para o terminal e uma para o mecanismo que entrega o dinheiro.

Cada classe de fronteira deveria interagir com pelo menos um ator e vice-versa.

Classe de entidades: Representa um conceito do domínio, por exemplo a conta do cliente ou uma fatura. Normalmente essas classes vão conter informações que queremos arquivar (persistência).

Classe de controle: Representa o controle de uma operação, o tratamento associado com uma operação, a ordem dos passos, etc.

O livro propõe uma representação especial para cada tipo de classe (c.f. a figura 10.1).

10.2.2 Desempenho da análise

Para criar os quatro artefatos dessa atividade, podemos efetuar as seguintes operações:

- Primeiro, estabelecer os pacotes, os casos de uso não serão locais a um pacote, mas distribuídos.
- Identificar as classes de entidades.
- Análise de cada caso de uso, para identificar as outras classes de análise. Especificar a realização dos casos de uso.

  Para cada classe de análise, vamos especificar suas responsabilidades.
- Identificar os requisitos especiais (requisitos não funcionais): persistência (quantos objetos, quanto tempo, com qual freqüência de modificação, etc.), seguridade, tolerância às falhas. Esses requisitos podem ser descobertos analisando as classes e as realizações de casos de uso.

Normalmente, os primeiros casos de uso vão criar (muitas) novas classes, mas depois de algum tempo, outros casos de uso vão reutilizar classes já criadas, principalmente as classes de entidades e as de fronteira.

Normalmente, se deve considerar em criar:

- Uma classe de fronteira central para cada ator. Se um ator já tiver uma tal classe fronteira "central", podemos ou aumentar a definição dela ou criar uma segunda classe de fronteira menos importante.

  Essas classes serão úteis para definir interfaces com os usuários.
- Uma classe de fronteira para cada classe de entidade já identificada. Essas classes de fronteira podem ser usadas para criar as interfaces das classes de entidade com os usuários.
- Uma classe de controle para cada caso de uso. Mas, essa classe não é sempre necessária, por exemplo, se todo o controle é assumido pelo ator e o programa só "reage".

Os casos de uso são realizados com diagramas de colaboração. Algumas regras gerais para fazer os diagramas de colaboração:

- O diagrama deveria começar com uma instância de uma classe de fronteira (caso de uso geralmente iniciado por um ator).
- Todas classes identificadas deveriam aparecer em pelo menos um diagrama de colaboração, caso contrario tem que tirar esta classe.
- Nesse modelo, as mensagens não são operações (das classes de análise) porque as classes ainda só têm responsabilidades.
- A seqüência exata das mensagens não é importante aqui. O importante é: as conexões entre as instâncias de classes de análise e as intenções de cada mensagem.

As responsabilidades das classes vão seguir dos diagramas de colaboração, cada mensagem que a classe pode receber será provalmente uma nova responsabilidade.

Os atributos das classes vão seguir das responsabilidades. Na análise é bom usar tipos de atributos conceitual em vez de tipos de programação. Por exemplo, montante seria melhor do que real. É bom tentar reutilizar tipos conceituais já definidos para outros atributos. Mas as classes de controle têm poucos atributos em geral.

Atributos de classes de fronteira que interagem com pessoas representam informações que vamos dar ao usuário ou que ele vai entrar.

10.3 Atividade de projeto

O objetivo da atividade de projeto é de começar a definir o programa, preparando sua implementação:

- Tratar todos os problemas relacionados com requisitos não funcionais, implementação com uma linguagem definida, reutilização de componentes, restrições materiais, outros softwares usados, ...
- Criar a documentação adequada para facilitar a manutenção no futuro.
- Definir interfaces entre os sub-sistemas.

Esta atividade vai resultar em cinco artefatos:

- Sub-sistemas de projeto.
- Classes de projeto com seus atributos, operações, associações. A introdução de novos elementos (essas classes) significa que podem surgir, ao mesmo tempo, novos requisitos não funcionais para esses elementos. Por exemplo, sobre o tempo mínimo ou máximo de execução de uma operação, ou o número de instâncias que um classe pode ter.
- Realização de casos de uso—projeto.
- Descrição da arquitetura—projeto. As partes importantes do modelo de projeto que serão incluídas na arquitetura são as mesmas que para a análise, mas ao nível do projeto (decomposição em sub-sistemas, com definição das interfaces dos sub-sistemas, classes importantes, e realização dos casos de uso.
- Modelo de distribuição (se tiver). Não estudemos o diagrama de distribuição, ele vai especificar a distribuição física (sobre várias máquinas) dos componentes implementados (sub-sistemas ou classes). Esse diagrama,também, pode ter partes incluídas na descrição da arquitetura.

10.3.1 Artefatos

Propusemos aqui algumas precisões sobre os vários artefatos produzidos nessa fase:

- As classes de projeto são exprimidas na linguagem de programação e de maneira geral, podem ser diretamente implementadas, por exemplo, os atributos e métodos podem ter marcas de visibilidade se existirem na linguagem de programação. O modelo é de um nível de abstração pouco mais alto que o código, por exemplo, os métodos, nesse modelo, podem ser exprimidos em pseudo-código.

- A realização de caso de uso—projeto é parecida a aquela que fizemos na análise. Ela é uma especificação mais detalhada, por exemplo, usando classes de projeto.

- Os diagramas de interação podem ser definidos com sub-sistemas em vezes de objetos. Esses diagramas de interação de alto nível de abstração permitem de descobrir as interfaces do sub-sistemas antes de começar a especificar os componentes deles.

  Normalmente, as interfaces dos sub-sistemas são consideradas pertinentes para ser incluídas na arquitetura.

- Diagramas de interação entre classes de projeto podem se tornar difíceis a entender por que já estamos a um nível muito baixo de abstração. Para ajudar, eles podem ser acompanhado de uma descrição em texto livre da interação, semelhante ao fluxo de eventos de um caso de uso, mas expresso em termos de classes do projeto.

  Nesses textos se deve evitar mencionar diretamente atributos, operações ou associações das classes, porque esses podem mudar freqüentemente o que nos obrigaria a reformular as descrições. Com descrições textuais mais abstratas, o trabalho de manutenção sobre elas será menor.

10.3.2 Desempenho do projeto

Para criar os artefatos da atividade, podemos efetuar as seguintes operações:

Sub-sistemas: Eles podem ser deduzidos dos sub-sistemas definidos na análise. Isso é uma decomposição do sistema da cima para baixo ("top-down").

Mas eles podem, também, ser definidos por agrupamento das classes de projeto (de baixo para cima, "bottom-up").

Um sub-sistema na análise pode ser decomposto em vários sub-sistemas no projeto:

- Uma parte do sub-sistema de análise pode ser reutilizada para outro sub-sistema de análise. Os dois serão decomposto em partes menores, a parte comum sendo criada uma vez só.
- Uma parte do sub-sistema da análise será "implementada" por software comprado, ou por software reutilizado. O sub-sistema de análise será decomposto em vários sub-sistemas no projeto, um deles sendo a parte "implementada" por um software existente.

- O sub-sistema de análise não pode ser implementado facilmente e tem que ser decomposto em sub-partes.

Dependências e interfaces de sub-sistemas: Dependências entre sub-sistemas podem ser deduzidas das dependências entre os componentes deles (por exemplo, entre duas classes em dois sub-sistemas diferentes), ou das dependências entre os sub-sistemas correspondentes na análise.

Com essas dependências, vamos deduzir qual sub-sistema precisa ter interface e o que essa interface deve ser. Cada sub-sistema sobre qual um outro depende deve ter uma interface.

A definição das operações que cada interface deve disponibilizar será deduzida da realização dos casos de uso.

Realização de caso de uso: Cada cenário de um casos de uso pode ser detalhado com um diagrama de sequencia. Ao inicio, o diagrama de sequencia pode ser deduzido de um diagrama de colaboração na realização do caso de uso na análise. Depois ele é detalhado com a introdução de classes de projeto.

Classes: As classes de projeto, vão ser deduzidas do estudo das classes de análise (como implementá-las?) e dos requisitos não funcionais.

As operações das classes são especificadas. Operações podem seguir das responsabilidades das classes de análise (uma responsabilidade freqüentemente necessita várias operações) ou de requisitos não funcionais.

10.4 Atividade de implementação e teste

Depois do projeto, vamos implementar e testar o programa. A natureza interativa do processo impõe de prestar atenção à integração dos novos componentes dentro do sistema.

A atividade de implementação pode ser decomposta em vários "builds". Cada "build" deve acrescentar novas funcionalidade (um cenário ou um caso de uso) e/ou refinar algumas funcionalidades já implementadas. Um "build" não pode ser muito grande porque o trabalho de integração seria complicado demais. "Builds" devem fazer crescer o sistema para cima, o que significa que vamos começar com funcionalidades básicas e construir funcionalidades mais abstratas sobre elas depois.

Cada componente deve ser testado só e depois integrado ao sistema. Depois de cada integração, temos que testar os componentes integrados (para verificar que funcionem) e o sistema inteiro (para verificar que não introduzem erro no que já estava funcionando).

Depois de cada integração, vamos repetir os mesmos testes já feito sobre o sistema inteiro ("regression testing") e acrescentar novos testes para os novos componentes. Isso significa que a atividade de teste vai ser cada vez maior para verificar que não introduzimos erros.

Capítulo 11

As fases do processo

Este material existe em versão eletronica, na página web:
`http://www.mestradoinfo.ucb.br/prof/anquetil/disciplinas.html`

Neste capítulo vamos apresentar com mais detalhes as quatro fases do processo de desenvolvimento RUP: Concepção, elaboração, construção e transição. Vamos definir os objetivos de cada fase e como o processo propõe de cumprí-los.

Este capítulo é baseado nos capítulos 12 a 16 do livro “The Unified Software Development Process” (c.f. referência no capítulo “Introdução” do curso).

> **Nota** : Nesse capitulo, apresentamos várias porcentagens (ex.: a fase de concepção representa 5% dos gastos totais do projeto). Eles são unicamente indicativos e não devem ser considerados como números absolutos. Cada projeto pode ter repartições diferentes.

11.1 Iteração e critério de sucesso

Planejar uma iteração:

- sobre quais sub-sistemas vamos trabalhar,
- quais casos de uso vamos tratar (completamente ou não),
- quais riscos vamos tratar (por exemplo, criando um caso de uso que suprimirá o risco),
- quais são os requisitos que mudaram.

Porque uma iteração é um mini-projeto, tem um risco de demorar demais nela para conseguir um resultado “perfeito”. Para evitar isso, se deve especificar claramente, antes de começar a iteração, quais são seus objetivos. Esses objetivos devem ser especificados de maneira que seja possível medí-los. As medidas vão definir os critérios de sucesso da iteração

Os critérios são requisitos (funcionais ou não) a ser especificados, ou realizados.

Exemplo de um mau critério: Especificar mais o caso de uso XXX. Não é claro até qual ponto temos que especificar o caso de uso. Um bom critério aqui poderia ser: Especificar o fluxo principal do caso de uso XXX.

Outros critérios podem ser ao respeito de requisitos não funcionais (ex.: Especificar a realização de um caso de uso tendo conta de um requisito não funcional) ou riscos (ex.: identificar os riscos associados que podem ter um impacto sobre um sub-sistema).

11.2 Concepção

O objetivo principal desta fase é de avaliar se o sistema valer a pena de ser desenvolvido, o que quer dizer se o custo de desenvolvé-lo será menor que os benefícios que ele pretende trazer. Uma conseqüência é que nessa fase vamos minimizar os gastos (em tempo e pessoal) por que não sabemos ainda se podemos desenvolver esse sistema com gastos razoáveis.

Não vamos realizar uma análise muito detalhada do problema nem implementar nada de muito elaborado. Vamos apenas estudar os casos de uso suficientes para entender o problema e estabelecer que podemos solvá-lo.

Nessa fase vamos:

Delimitar o sistema: Para delimitar a pesquisa dos riscos, a definição da arquitetura, etc.

Iniciar a arquitetura: Principalmente ao respeito das partes novas, ou mais difíceis. Queremos apenas estabelecer que podemos definir uma arquitetura para o sistema (inteiro).

Identificar os riscos: Procuramos os riscos que poderiam fazer falhar o projeto e tentar achar uma solução para eles (ver abaixo). Outros riscos, menos graves, serão identificado e tratado nas fases a seguir.

Criar um protótipo: Queremos mostrar aos usuários como o sistema vai resolver o problema deles. Por exemplo, podemos criar uma seqüência de telas para mostrar como o sistema vai funcionar, ou podemos criar um protótipo mesmo.

É importante lembrar de guardar essa fase pequena, porque ainda não sabemos se o sistema vale a pena de ser desenvolvido. Tipicamente essa fase representa 5% dos gastos totais do projeto. Essa pequena participação aos gastos não significa que a fase pode ser suprimida. Ela tem sua importância no processo de desenvolvimento.

Critérios possíveis para essa fase são:

Delimitar o sistema: O que o sistema deve fazer é claro? Identificamos todos os atores? As interfaces são especificadas (ainda de maneira muito abstrata)?

Iniciar a arquitetura: A arquitetura permite responder aos desejos dos usuários? Ela parece funcionar? Ela será eficiente, capaz de evoluir com o sistema, etc. ?

Identificar os riscos: (ver discussão sobre os riscos abaixo) Todos os riscos (críticos) foram identificados? Para cada um, temos um plano para tratar ele? Ter um plano para tratar o risco não significa que sabemos exatamente como o tratar, podemos só ter uma boa idéia para reduzir a probabilidade do risco ou sua importância, ou que o cliente aceito rever seus requisitos para eliminar um risco.

A maioria do trabalho se faz na atividade de levantamento dos requisitos com menos trabalho nas atividades de análise e projeto. Talvez vamos implementar um protótipo, mas a atividade de teste deveria ser quase inexistente (não precisa testar um protótipo).

11.3 Elaboração

Os objetivos principais desta fase são de estabelecer um planejamento para a construção do sistema e de definir sua arquitetura.

Nessa fase vamos:

Arquitetura: Criar as bases da arquitetura tendo conta dos principais casos de uso e requisitos não funcionais dos usuários.

Riscos: Identificar outros riscos que poderiam atrasar o projeto ou aumentar os gastos. Temos, também, que planejar como vamos tratar esses riscos.

Requisitos: Especificar a maioria dos casos de uso (80%) e de maneira geral, levantar a maioria dos requisitos.

Submissão: Fazer uma submissão para o desenvolvimento do sistema.

Vamos apenas construir uma visão geral sem detalhes do sistema, mas podemos estudar em detalhes, também, algumas partes às quais são associados riscos importantes.

Esta fase é mais importante (em investimento) que a primeira, tipicamente ela representa 20% dos gastos totais do projeto.

Critérios possíveis para essa fase são:

Arquitetura: Mesmos critérios que na fase precedente mas desta vez, temos mais informações para julgá-los.

Riscos: Os risco críticos são tratados? Todos os riscos importantes são identificados? Temos um método rápido e eficiente de acesso à lista de riscos (para riscos rotineiros)?

Requisitos: Identificamos todos os atores e casos de uso para procurar os riscos e definir a arquitetura? Eles são detalhados suficientemente?

Submissão: O projeto esta suficientemente definido para definir o custo global, um calendário de desenvolvimento e critérios de qualidade?

A maioria do trabalho se faz nas atividades de levantamento dos requisitos, análise e projeto. Depois desta fase, a atividade de levantamento dos requisitos será quase finita, deveríamos tendo explicitado 80% dos casos de uso, incluido os mais importantes. Em paralelo, começamos a atividade de análise —vamos realizar em torno de 50% dos casos de uso— que será continuada na seguinte fase. Vamos, também, começar a atividade de projeto (apenas 10%) para identificar os sub-sistemas e as classes importante para a arquitetura e para especificar as interfaces dos sub-sistemas. Vamos começar, também, a implementar o sistema para implementar a base da arquitetura (sub-sistemas, suas interfaces e as classes mais importantes) e puder testar.

11.4 Construção

O objetivo principal desta fase é de construir uma versão beta do sistema, o que quer dizer um sistema com quase todas as funcionalidades (e todas as importantes), mas que vai provavelmente conter bugs. Esta primeira versão do sistema será testada por um conjunto restrito de usuários.

Esta fase será mais longa (terá mais iterações) que as outras e com mais pessoal. Seu bom desempenho depende muito da qualidade do trabalho das fases precedentes.

Também, nesta fase, vamos trocar o foco, da pesquisa de informações (concepção e elaboração) vamos passar a construção do sistema.

Nesta fase vamos:

Casos de uso: Identificar, especificar e realizar todos os casos de uso.

Riscos: Manter o controle dos riscos identificados. Quando um risco se realizar, temos que tratá-lo.

Documentação: Começar a redação da documentação

Esta fase é a mais importante (em investimento), tipicamente ela representa 65% dos gastos totais do projeto. Isso porque implementação é uma atividade longa onde se deve cuidar de muitos detalhes.

Critérios possíveis para essa fase são:

Implementação: Os requisitos ligados com cada caso de uso (funcionais ou não) são os próprios critérios: será que o sistema realiza um caso de uso com os requisitos não funcionais especificados?

Documentação: A documentação (manual do usuário, de manutenção, etc.) é suficientemente clara para ajudar os usuários?

As atividades iniciais (levantamento dos requisitos e análise) serão terminadas nas primeiras iteração desta fase. Vamos terminar de especificar os últimos casos de uso e realizar todos. A maioria (90%) de todo o trabalho de projeto se faz nesta fase. As atividades de implementação e teste serão muito importantes, vamos implementar o sistema inteiro, testar cada classe, sub-sistema, etc. e testar que eles se integram bem para fornecer o sistema completo. Cada parte é testada individualmente e depois integrada e o sistema "inteiro" (o que já esta implementado) é testado. De uma iteração para a outra, vamos guardar e recomeçar os mesmos testes, para verificar que os novos componentes não introduzem erros no que já existia antes ("regression testing").

11.5 Transição

O objetivo principal desta fase é de finalizar o sistema e entregar a todos os usuários.

Nessa fase, vamos também fazer a análise post-mortem do projeto: lições aprendidas, o que funcionou bem, o que não funcionou (e por que). Vamos, finalmente, gravar informações

que poderiam ser interessante para um futuro desenvolvimento (melhoramentos que não foram implementados, problemas não previstos, etc.)

Nesta fase vamos:

Documentação: Terminar a documentação (manuais e outra documentação), entregá-la aos usuários.

Bugs: Corrigir os problemas identificados pelos beta testadores, problemas que não apareceram nos testes e foram descobertos no ambiente do cliente.

Ambiente: Ajustar o sistema ao ambiente do cliente e o ambiente ao sistema: treinamento dos usuários, mudança de ambiente (troca de máquina, de sistema de operações, etc.)

Esta fase é pequena (em investimento), tipicamente ela representa 10% dos gastos totais do projeto. Mas não se deve cortá-la. Algumas pessoas podem ter a tendência de deixar passar essa fase:

- Para respeitar um prazo curto demais. Essa fase é tão importante como as outras, não tem mais razões de cortar ela que de cortar a fase de construção, por exemplo.
- Porque pensam que depois da versão beta sobra muito pouco trabalho. O fato do sistema ser completo na versão beta, não significa que o trabalho acabou, ainda tem bugs no sistema, tem que formar os usuários, completar a documentação, etc. Esta fase é curta, mas é importante também.
- Porque pensam que bugs nessa fase mostram a falha do projeto. É normal depois da versão beta ter ainda vários bugs para corrigir.

Critérios possíveis para essa fase são:

- Os usuários da versão beta testaram todas as funcionalidades?
- Os critérios estabelecidos no contrato são respeitados?
- A documentação é completa?
- Os usuários estão satisfeitos com o produto?

O trabalho esta concentrado nas duas últimas atividades (implementação e teste), principalmente para implementar correções de bugs.

11.6 Riscos

O tratamento dos riscos é uma parte importante dentro do processo de desenvolvimento. Desde a concepção, vamos manter uma lista dos riscos identificados. O que precisamos saber sobre um risco:

- Descrição curta

- Prioridade. Em geral, três categorias são suficiente (grave ou crítico, importante, rotineiro).
- Impacto: Qual parte do sistema apresenta o risco?
- Revisor: Quem (qual equipe) vai controlar se o risco acontecer?
- Responsabilidade: Quem (qual equipe) vai suprimir o risco?
- Tratamento: O que fazer se o risco se tornar verdadeiro?

Ao longo do projeto, a lista vai crescer enquanto identificarmos novos riscos, e diminuir enquanto eliminarmos um risco ou ele não aparecer.

Tipos de riscos:

Tecnologia: Por exemplo, problemas de sincronização com processos distribuídos, ou problemas ao usar novas técnicas (ex. reconhecimento da palavra).

Eficiência: Por exemplo, o sistema não cumpre alguns requisitos não funcionais de rapidez.

Arquitetura: Por exemplo, a arquitetura usada não permite evolução do sistema numa direção desejada ou não permite de encontrar alguns requisitos.

Adequação: Por exemplo, o sistema não responde às esperanças dos usuários, ou a solução não é ótima. Não construímos o bom sistema. Pode acontecer quando tiver erros ou esquecimentos no levantamento dos requisitos.

A lista vai conter riscos dos dois primeiro tipo. Os outros são tratado diretamente pelo processo nas fases de concepção e de elaboração.

Tem quatro possibilidades de tratar um risco:

Eliminar: Podemos tentar eliminar um risco pelo replanejamento do processo, ou pela redefinição dos requisitos (supõe uma negociação com os usuários).

Limitar: Podemos tentar limitar um risco a uma pequena parte do sistema.

Testar: Podemos testar o risco e ver se ele se realiza. Se ele realmente se realizar, vamos ter um pouco mais informação sobre ele para tentar um outro tratamento.

Controlar: Se as outras soluções não forem possíveis, temos que controlar ele durante o desenvolvimento. Se ele se realizar, vamos ter que reavaliar o processo: será que podemos desenvolver os sistema?