Engenharia de Software: Uma Visão Prática

2a Edição

Ricardo R. Gudwin
DCA-FEEC-UNICAMP

Agosto de 2015

Índice

1. Introdução

Muitas vezes podemos nos perguntar qual é a importância da engenharia de software no desenvolvimento de sistemas de software. Parece ser uma idéia natural que o desenvolvimento de software deva utilizar algum tipo de engenharia para sua concretização, mas muitas vezes nos deparamos com bons programadores questionando sua utilização. Dizem eles:

- Bah, ... , para que tanta documentação, tantos diagramas, tanta perda de tempo. Eu programo direto mesmo ... abro meu editorzinho de texto e desenvolvo meu sistema direto no código. É muito mais rápido, muito mais eficiente e rapidinho eu saio com o sistema funcionando, sem documentação, sem diagramas, sem metodologia nenhuma ! Isso não me serve para nada na prática. Na prática, o que interessa é o código. Se me pressionarem, ... , primeiro faço o código e depois faço essa documentação que me pedem ... só para não me encherem a paciência !

Imagino que muitos de vocês podem ter uma idéia semelhante, fruto de alguma experiência mal sucedida com as diversas metodologias de software que existem no mercado (muitas que são ruins mesmo !). Entretanto, resolvi gastar algumas linhas aqui para tentar convencê-los do contrário, tentando ressaltar as vantagens da engenharia de software no desenvolvimento de sistemas de software. Para isso, vou usar uma analogia que é muito interessante, e que vai nos permitir entender POR QUE a engenharia de software é importante, e POR QUE deve ser utilizada.

Imaginemos a tarefa de construir uma casa. É claro que existem diversos tipos de casa … desde as casas de pau-a-pique até grandes mansões ou edifícios de apartamento. É óbvio que é necessário sabermos que casa iremos construir. Imaginemos a princípio que seja uma casa muito simples, de pau-a-pique. Para construir uma casa dessas, o que é necessário ? Algum conhecimento de construção (do tipo que um bom pedreiro deve ter) e eventualmente experiência na construção de casas. Para uma casa de pau-a-pique, sem eletricidade, sem encanamento, basta escolher o lugar, ter uma idéia na cabeça do que quer construir e sair construindo. Realmente, é possível construir boas casas de pau-a-pique sem um projeto associado, sem uma **engenharia** associada.

Vamos tentar deixar a coisa um pouco mais complexa. Não é mais uma casa de pau-a-pique, mas uma casa de tijolos - simples, mas de tijolos. Com eletricidade, encanamento, piso, forro, portas e janelas. Será possível construirmos uma casa desse tipo sem um projeto associado ? Sim, é ! E de fato, existem diversas construções clandestinas que são feitas dessa maneira, principalmente na periferia. Na melhor das hipóteses, o pedreiro faz um rascunho do "jeitão" que a casa deve ter, posiciona as paredes, portas e janelas e sai construindo. As partes elétrica e hidráulica são minimalistas e feitas de improviso: uma tomada aqui, um ponto de luz ali, uma privada aqui e um chuveiro lá. Sim, … , a casa sai e é possível viver razoavelmente bem dentro de uma delas. Entretanto, alguns problemas começam a aparecer quando, por exemplo, após a construção toda feita, descobre-se que esqueceu-se de colocar um ponto para uma antena de televisão, por exemplo. Ou que o número de tomadas na sala não permite colocarmos os abajures em pontos convenientes à decoração. Ou que (o que é muito comum), o disjuntor cai quando ligamos juntos o chuveiro e a máquina de lavar roupa. O que é isso ? São os primeiros indícios de que: ou não houve projeto para a construção da casa, ou o projeto foi muito mal feito. Estendendo esse raciocínio de "pequenos defeitos" associados a uma construção sem projeto (ou projeto deficiente), podemos dizer que seria praticamente impossível construirmos por exemplo, um edifício de apartamentos, sem um projeto associado, com uma engenharia e uma metodologia por trás que garantisse que esse projeto foi bem

executado.

Por quê ? Porque à medida que a complexidade de nossa construção aumenta, milhares de detalhes passam a tornar-se importantes, e a desconsideração desses detalhes, (ou sua má consideração) faz com que a construção termine cheia de problemas, podendo até (em alguns casos) desmoronar !

O mesmo acontece com o desenvolvimento de um software hoje em dia ! Nos primórdios da computação, os programas que eram desenvolvidos eram muito sumários - equivalentes às casas de pau-a-pique na construção. Um bom programador, podia desenvolver esses programas diretamente codificando sobre a linguagem desejada e provavelmente sairia realmente com programas de uma qualidade até que aceitável, sem nenhum planejamento ou metodologia de desenvolvimento. Entretanto, não devemos negligenciar a complexidade dos sistemas de software da atualidade. Com a introdução de sistemas distribuídos, rodando sobre múltiplos threads de controle, tecnologias de componentes (e.g. COM, DCOM, CORBA, ActiveX, etc…), servlets e applets, agentes, etc… a complexidade de um sistema de software cresceu a níveis que dificilmente poderiam comportar um desenvolvimeno direto-no-código ! É óbvio que para o desenvolvimento de programinhas simples, pode-se fazê-lo, mas tentar desenvolver um sistema de software grande e complexo sem um planejamento e sem uma metodologia só nos levará a problemas. Existe um custo para a engenharia de software ? Sim, existe ! Mas é um custo que cada vez mais vale a pena ser pago, para que o desenvolvimento de nossos projetos de software possam resultar programas confiáveis e de bom design.

Outro ponto importante: a padronização na elaboração do projeto. É óbvio que quando o nosso pedreiro fez um rascunho da planta da casa que iria construir, de alguma forma esse rascunho era uma forma de projeto. Da mesma maneira, vemos diversos tipos de uso de diagramas e rascunhos na elaboração de um projeto de software. Muitas vezes, esses diagramas até são feitos com fragmentos de diagramas e metodologias de engenharia de software. Entretanto, são na maioria das vezes úteis somente para quem os fez, pois a maioria dos detalhes está na cabeça de quem fez o desenho. Quero aqui ressaltar a importância de adotarmos uma padronização para a linguagem que utilizamos nos nossos planos. Cada vez mais, não é somente uma pessoa que participa da construção de um software, mas uma equipe de pessoas. Se cada membro da equipe usa sua notação individual, essa documentação acaba por ser confusa e fonte de problemas. Da mesma maneira, havendo uma equipe é importante que haja um conhecimento da metodologia de desenvolvimento utilizada, e que cada membro saiba o que deve ser feito e que parte lhe cabe. O que aconteceria na construção de uma casa, se o instalador de carpetes instala o piso antes do pintor pintar as paredes ? Chão sujo de tinta! O que acontece se o pintor inicia a pintura antes do pedreiro abrir a parede para colocar os conduites de eletricidade e os canos ? É obrigado a fazer tudo de novo ! Assim o mesmo acontece com os elementos de uma equipe de desenvolvimento de software. É necessário respeitarmos a metodologia de desenvolvimento para que não sejamos obrigados a fazer tudo de novo alguma coisa que atropelemos no desenvolvimento.

Espero que tenha ficado clara a necessidade de adotarmos uma engenharia de software para a construção de grandes sistemas de software. Assim como quando construímos uma casa, é necessária uma planta, um projeto, que nos permita visualizar o sistema, antes que ele seja construído. Da mesma forma, é necessária uma linguagem de modelagem padronizada para que qualquer engenheiro, de posse da documentação, possa ter uma idéia de como o programa será construído. E é óbvio que é necessária uma metodologia de desenvolvimento para que o esforço seja utilizado de maneira racional, fazendo com que a construção de nosso sistema seja feita de

maneira mais eficiente.

Para pequenos projetos de software, é necessária toda essa metodologia ? Talvez não ! Mas uma vez que nos eduquemos em adotar uma metodologia de desenvolvimento (o que é o mais difícil), talvez não custe nada adotarmos a engenharia de software mesmo em projetos de pequeno porte. É o tipo de cuidado que nos faz engenheiros ao invés de pedreiros, … , que nos faz engenheiros de computação ao invés de programadores. E que nos faz vermos nossa eficiência aumentar, tanto no tempo de desenvolvimento quanto na qualidade do software que disponibilizamos para uso.

Pensem bem ! Reflitam nesses pontos que colocamos acima. Tornem-se engenheiros conscientes da importância de seu papel. Vocês só têm a ganhar !

2. Sistemas de Software

Sistemas de software podem ser dos mais variados tipos possíveis. Podem ser, por exemplo, **sistemas embutidos** (ou sistemas embarcados - *embedded systems*), que são sistemas dedicados ao controle de dispositivos (normalmente rodando em micro-controladores) e agregados a um produto eletrônico. Exemplos de sistemas embutidos são sistemas de controle de video-cassetes, fornos de micro-ondas, aparelhos de ar-condicionado, sistemas de injeção eletrônica de automóveis e muitos outros que passam desapercebidos nos artefatos tecnológicos que utilizamos em nosso dia a dia. Algumas características de sistemas embutidos são as restrições de memória (limitadíssima) e o fato de não utilizar dispositivos periféricos tais como monitor de vídeo, discos rígidos, drives, etc

Da mesma maneira, outro tipo de sistema de software radicalmente diferentes são as **aplicações de bancos de dados**. Bancos de dados são aplicativos que tem por finalidade o armazenamento de dados de diversas espécies (comerciais ou industriais) e seu acesso e manipulação. As aplicações de bancos de dados permitem que esses dados sejam criados, alterados, substituídos e apagados, de tal forma a dar suporte a algum negócio comercial ou industrial. São o tipo de sistema de software que talvez mais tenham mercado em ambientes comerciais e também em ambientes industriais.

Outro tipo de sistema de software são os **sistemas de simulação**. Nesse tipo de sistema, têm-se normalmente algum processo físico/químico que se deseja analisar, mas cuja análise do processo real ou é muito custosa, ou muito demorada ou de muito risco. Nestes casos, antes de se fazer experimentos com o processo real, efetuam-se exercícios de simulação em computadores. Após se chegar a algum resultado que pareça promissor, passa-se então aos testes reais. Sistemas de simulação são muito comuns no ambiente industrial, principalmente em sistemas de produção da indústria química e de alimentos. Podem, entretanto encampar diversas outras finalidades como os sistemas de simulação de pilotagem ou de ambientes de realidade virtual.

Um tipo de sistema de software que muitas vezes é desconsiderado é o de **ferramentas de produtividade**. Desenvolver um sistema de software desse tipo muitas vezes nem envolve diretamente a tarefa de programação, mas sim a integração de diversos tipos diferentes de programas que já têm uma finalidade por si só. Um exemplo do desenvolvimento de um sistema desse tipo poderia ser a integração de ferramentas do tipo Office com e-mail e acesso a web para o uso de um banco de dados que já existe em algum tipo de escritório ou divisão de uma companhia. Apesar de não envolver diretamente programação, o desenvolvimento de sistemas desse tipo exigem grande conhecimento técnico e muitas vezes as mesmas fases de desenvolvimento que um sistema de outro tipo.

Alguns tipos de sistemas são **aplicações de propósito especial**. Esses sistemas são sistemas que têm uma finalidade muito particular, mas que em alguns casos pode servir de elemento atômico em sistemas de integração como os ressaltados anteriormente. Um exemplo de uma aplicação de propósito especial seriam aplicativos de manipulação de arquivos ZIP, de arquivos de imagens, etc

Podemos chegar até ao caso (muito comum na atual indústria do software) do desenvolvimento de meros **componentes de software**. Nesse caso, explicitamente a intenção não é desenvolver um sistema completo, mas simplesmente um componente que deverá ser integrado a um sistema maior para que tenha sua funcionalidade verificada.

Outro tipo de sistema são os **sistemas de controle**. Diversos tipos de sistemas de controle podem existir. Controles de processos industriais, controle de recursos humanos, etc. Esse tipo de sistema envolve fases de monitoração por meio de sensores e atuação por meio de atuadores.

Diversos outros tipos de sistemas de software podem existir. Cada um deles demandará um tipo diferente de metodologia de engenharia de software para sua construção. Entretanto, um corpo de metodologia pode ser utilizado para a maioria desses tipos de sistemas.

3. Processos e Linguagens de Desenvolvimento de Software

Supondo que se deseja desenvolver um determinado sistema de software, duas características serão importantes para que balizemos esse desenvolvimento segundo uma metodologia de engenharia de software:

- Um processo de desenvolvimento
- Uma linguagem de modelagem

Um **processo de desenvolvimento** corresponde a um conjunto de atividades a serem desenvolvidas por um ou mais trabalhadores da indústria do software que seguindo uma determinada sequência lógica de passos, conseguem determinar as necessidades dos usuários e transformá-las em um sistema de software. Para documentar e dar suporte às atividades executadas em um processo de desenvolvimento, um conjunto de artefatos de software são criados, desenvolvidos em uma determinada linguagem de modelagem. Exemplos de artefatos de software são diagramas, gráficos, textos e tabelas com finalidades explícitas.

Uma **linguagem de modelagem** corresponde a uma linguagem padronizada na qual artefatos de software possam ser desenvolvidos. Essa linguagem é geralmente uma linguagem visual, que especifica a sintaxe e a semântica de classes de diagramas, utilizados explicitamente para modelar partes de um sistema de software ou de planos de construção de sistemas de software.

Neste curso, estaremos utilizando o processo de desenvolvimento chamado de processo **Unificado** e a linguagem de modelagem chamada de **UML**. Observe que UML é apenas uma ***linguagem***, e não um ***processo*** de desenvolvimento. O processo Unificado utiliza a linguagem UML como linguagem de modelagem, assim como outros processos poderiam também utilizá-la. Existem outros processos mais antigos, tais como OMT, Booch, OOSE (Jacobson - Use Cases), Fusion, Shlaer-Mellor, Coad Yourdon, etc.

3.1 Ferramentas de Desenvolvimento

Para tornar a construção dos artefatos de software (edição dos diagramas) um processo mais amigável, algumas ferramentas de desenvolvimento foram desenvolvidas. Algumas dessas ferramentas são comerciais. Exemplos de ferramentas comerciais incluem:

- Rational Rapsody
- Enterprise Architect
- StarUML
- Visual Paradigm for UML

Existem também ferramentas oriundas de projetos do tipo open-source que possuem licensa de uso livre. Um exemplo de ferramenta desse tipo é o:

- ArgoUML
- UMLet
- UML Designer

Há ainda algumas ferramentas proprietárias com versões livres, ou community edition, tais como:

- Astah
- Papyrus
- Modelio

As ferramentas de edição UML estão em constante mudança. Algumas existem desde o início do UML 1.0, e outros foram aparecendo e desaparecendo ao longo do tempo. Uma compilação das ferramentas abertas e fechadas, com as datas das últimas atualizações pode ser encontrada em:

https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools

Assim como existem produtos comerciais para a edição de diagramas (e outras coisas mais), existem também produtos que dão apoio à implantação de metodologias de desenvolvimento. Um desses produtos é o RUP, acrônimo de *Rational Unified Process*. O RUP é um conjunto de páginas em HTML que servem de suporte ao usuário, fazendo com que todas as etapas do processo Unificado sejam documentadas e cumpridas. A Rational participou ativamente do desenvolvimento do processo Unificado. De uma certa maneira, o RUP é uma implementação, na forma de um produto, do próprio processo Unificado. Estaremos, nesse curso, seguindo o processo Unificado, mas não estaremos adotando o RUP, e sim uma versão simplificada deste com propósitos didáticos. Após a Rational Software ser adquirida pela IBM, o RUP (como produto) foi integrado a um outro produto, o *RMC - Rational Method Composer*, onde outros processos (principalmente oriundos de metodologias ágeis) podiam contribuir para a composição de um processo sob medida para uma empresa ou para um projeto em específico.

PARTE 1

A Linguagem UML de Modelagem

4. Introdução à Linguagem UML

A Linguagem UML (Unified Modeling Language) é uma linguagem de modelagem que foi criada visando-se a criação de **modelos abstratos de processos**. Em princípio, não existem restrições quanto aos tipos de processos sendo modelados. Tanto podem ser processos do mundo real como processos de desenvolvimento de software ou ainda detalhes internos do próprio software. Assim, tanto podemos utilizar o UML para descrever o mundo real - por exemplo - a organização interna de uma empresa, como os detalhes internos que descrevem um sistema de software. A descrição de um processo envolve a determinação de duas classes básicas de termos:

- Os elementos estruturais que compõem o processo
- O comportamento que esses elementos desenvolvem quando interagindo

Assim, o UML tanto pode ser utilizado para a análise dos elementos ontológicos participantes de um processo como do comportamento destes elementos no processo. (o adjetivo *ontológico* diz respeito a algo que é porque é, simplesmente, sem entrar em detalhes de porque é ou como é).

Particularmente no tocante à engenharia de software, a linguagem UML pode ser utilizada para modelar todas as etapas do processo de desenvolvimento de software, bem como produzir todos os artefatos de software necessários à documentação dessas etapas.

Na verdade, o UML é a convergência de diversas outras linguagens de modelagem utilizadas em diferentes processos de desenvolvimento de software propostos até então.

Apesar de ser uma linguagem formal (ou seja, é definida na forma de uma gramática - no caso a partir de uma meta-linguagem de descricão), o UML é uma linguagem visual, baseada em diferentes tipos de diagramas. Neste capítulo, teremos a oportunidade de conhecer de maneira sumarizada alguns dos diagramas utilizados na linguagem UML. Em capítulos posteriores, iremos entrar em detalhes dos diversos diagramas necessários para a construção de um software.

Uma das características interessantes do UML é a existência de **mecanismos de extensão**, que permitem que o UML, como linguagem, possa ser estendido, resultando a criação de novos tipos de diagramas. Os mecanismos de extensão do UML são os chamados **Profiles**. Diferentes Profiles podem ser construídos utilizando-se **estereótipos**, os ***tagged values*** e as **restrições**.

A linguagem UML, por meio de seus diagramas, permite a definição e design de threads e processos, que permitem o desenvolvimento de sistemas distribuídos ou de programação concorrente. Da mesma maneira, permite a utilização dos chamados **patterns** (patterns são, a grosso modo, soluções de programação que são reutilizadas devido ao seu bom desempenho) e a descrição de **colaborações** (esquemas de interação entre objetos que resultam em um comportamento do sistema).

Um dos tipos de diagramas particularmente úteis para modelarmos processos são os chamados diagramas de atividades. Por meio deles, é possível especificarmos uma sequência de procedimentos que compõem um processo no mundo real. Diagramas de atividade podem portanto ser utilizados para descrever o processo de desenvolvimento de software (por exemplo, o processo

Unificado).

Uma outra característica do UML é a possibilidade de efetuarmos uma descrição hierárquica dos processos. Com isso, é possível fazermos um refinamento de nosso modelo, descrevendo o relacionamento entre diferentes níveis de abstração do mesmo.

Para implementar esse refinamento, utilizamos o expediente de definir abstratamente **componentes** de um determinado modelo por meio de suas **interfaces**. Assim, esses componentes são definidos somente em função de suas entradas e saídas, deixando a definição dos *internals* do componente para um nível de abstração posterior. Essa caracterísitca permite ao UML um **desenvolvimento incremental** do modelo.

Por fim, a semântica dos diagramas UML é determinada por uma linguagem de restrição chamada de OCL (Object Contraint Language), que determina de maneira não-ambígua a interpretação a ser dada a seus diagramas.

Todas essas características fazem da linguagem UML uma linguagem de modelagem moderna, eficiente e conveniente para o desenvolvimento de especificações e definições com relação ao sistema em desenvolvimento.

4.1 História do UML

As linguagens de modelagem começaram a aparecer na engenharia de software no final dos anos 70, com o desenvolvimento dos primeiros sistemas orientados a objeto. A partir daí, foram sucessivamente empregadas em diversos experimentos em diferentes abordagens orientadas a objeto. Diversas técnicas influenciaram estas primeiras linguagens: **os modelos entidade/relacionamento**, o **SDL** (Specification and Description Language) e diversas outras. Entretanto, apesar de sua conveniência, o número de linguagens de modelagem passou de pouco mais de 10 para mais de 50 até 1994. Junto a este fato, começaram a proliferar diferentes métodos de desenvolvimento, detonando uma verdadeira "guerra de métodos". Alguns dos métodos mais famosos são por exemplo o método Booch, o OMT, o método Fusion, o OOSE e o OMT-2. Uma característica dessa guerra de métodos é que todos os métodos tinham linguagens de modelagem que eram muito similares conceitualmente, entretanto empregando uma notação diferente. Algumas tinham características que faltavam em outras, que por sua vez tinham outras características que estas não tinham. Estava clara a necessidade de algum tipo de padronização. O desenvolvimento do UML começou no final de 1994, quando Booch e Rumbaugh passaram a trabalhar em conjunto, envidando esforços para integrar suas metodologias.

Uma primeira versão preliminar do UML (versão 0.8) surgiu em outubro de 1995, quando Jacobson se uniu ao grupo. Vale a pena ressaltar quem são Booch, Rumbaugh e Jacobson. Estes indivíduos, conhecidos na vulgata como ***los tres amigos***, foram as principais lideranças no pensamento científico voltado para a engenharia de software orientada a objetos durante a "guerra de métodos. O esforço que fizeram por tentar um diálogo acabou por permitir o fim da guerra dos métodos e a criação do UML. De fato, a partir dos esforços conjuntos de Booch, Rumbaugh e Jacobson (que fundaram, mais ou menos por essa data a Rational Software - empresa que norteou o desenvolvimento do UML), acabou resultando no lançamento público da versão 0.91 do UML, em outubro de 1996 (as versões anteriores não eram públicas). A partir desse esforço, uma RFP

(Request for Proposal) foi realizada pela OMG buscando contribuições da comunidade para o estabelecimento de uma linguagem unificada. Antes de prosseguirmos, vale a pena esclarecer o que é uma **RFP** e o que é a **OMG**.

Antes de mais nada a RFP. Esse termo, RFP está relacionado com um mecanismo de criação colaborativa de padrões que vem sendo utilizado na Internet para a criação de normas e procedimentos relacionados à Internet. Assim, quando um grupo de indivíduos verifica que algum tipo de protocolo - ou qualquer coisa - que esteja funcionando na Internet precisa ser padronizado, esse grupo lança um RFP - um *Request for Proposal* - uma requisição de propostas. Essa RFP é um documento onde o grupo sumariza uma proposta para o protocolo ou padrão que se deseja desenvolver, e requisita comentários da comunidade internet sobre a proposta, para que ela seja avaliada de maneira independente por múltiplas fontes e possa evoluir de maneira natural e democrática. Comentários e sugestões são agregados à proposta original e esta evolui até se transformar em uma norma ou padrão.

Sobre a OMG - a OMG - ou *Object Management Group*, é uma organização sem fins lucrativos, composta por representantes de diversas empresas que trabalham no desenvolvimento de software orientado a objetos, que se dedica ao esforço de desenvolver normas relacionadas ao desenvolvimento de software orientado a objetos. Dentre outras normas, a OMG é responsável pelo padrão CORBA, e diversos outros padrões desenvolvidos de maneira colaborativa na Internet.

Voltando agora à história do UML - a partir das diversas contribuições que resultaram da RFP, lançou-se o UML 1.0. Dentre as empresas que enviaram sugestões estavam nomes de peso tais como a Digital Equipment Corp., a HP, a i-Logix, o IntelliCorp, a IBM, a ICON Computing, a MCI Systemhouse, a Microsoft, a Oracle, a Rational Software, a Texas Instruments e a Unisys. Em janeiro de 1997, novas contribuições lançaram o UML 1.1. Dentre as empresas que colaboraram desta vez estavam a IBM, a ObjecTime, a Platinum Technology, a Ptech, a Taskon & Reich Technologies e a Softeam.

A partir da versão 1.1, a comunidade de desenvolvimento de software começa a fazer uma aderência maciça ao UML. Em novembro de 1997, o UML foi adotado como norma pela OMG, que estabelecu um RTF (Revision Task Force) para aperfeiçoar pequenos detalhes na linguagem. Em junho de 1999 o RTF libera a versão UML 1.3 e em setembro de 2001 é lançado o UML 1.4. Em março de 2003, publica-se a versão 1.5, que combina os detalhes da versão 1.4 com uma semântica de ações. Em julho de 2004, a versão 1.4.2 é criada, sendo simultaneamente publicada como a norma ISO/IEC19501. Em julho de 2005 publica-se a versão 2.0 da norma, que traz diversas inovações (dentre outras, substituindo o diagrama de colaboração pelo diagrama de comunicações - uma versão aperfeiçoada do mesmo). A versão 2.1 do UML nunca foi publicada como uma especificação formal, mas em agosto de 2007 disponibilizou-se a versão 2.1.1 e em novembro de 2007 a versão 2.1.2. A versão 2.2 da norma foi publicada em fevereiro de 2009 e em maio de 2010 publicou-se a versão 2.3. Em março de 2011 publicou-se a versão 2.4 e em agosto de 2011 saiu uma versão corrigida, a 2.4.1. Em junho de 2015 foi publicada a versão 2.5, que é a versão mais atual da norma até a data da elaboração deste documento [OMG 2015].

5. Elementos de Modelagem

Diagramas podem ser entendidos, de maneira simplificada, como grafos contendo **nós**, **caminhos** entre os nós e **textos**. Esses nós podem ser símbolos gráficos (figuras bidimensionais) tão complexos e estruturados quanto possível. Os caminhos, podem ser linhas cheias ou pontilhadas com possíveis decorações especiais nas pontas. Por fim, os textos podem aparecer em diferentes posições, sendo que conforme a posição em que aparecem, podem ter um significado diferente. O que torna tal grafo um diagrama é o relacionamento visual que existe entre os diferentes elementos que o compõem. Este relacionamento na verdade espelha algum tipo de relacionamento entre os elementos envolvidos. Por exemplo, o relacionamento de **conexão** entre elementos do diagrama, modela algum tipo de relação entre os elementos conectados. Relacionamentos de conexão são usualmente expressos por meio de linhas ligando figuras bidimensionais. O relacionamento de **inclusão** (por exemplo, a inclusão de um símbolo dentro da área correspondente a outro símbolo), denota a inclusão de um determinado elemento em outro. Por fim, a **proximidade visual** entre símbolos (por exemplo, um símbolo estar perto de outro símbolo ou figura bidimensional dentro de um diagrama), denota algum tipo de relação entre esses símbolos.

Os elementos de um diagrama podem ser: **ícones**, **símbolos bidimensionais**, **caminhos** e **strings**. Os **ícones** são figuras gráficas de tamanho e formato fixo que não podem ser expandidas para ter algum tipo de conteúdo. Podem aparecer como participantes de um símbolo bidimensional, como terminadores de caminhos ou simplesmente isoladamente. Os **símbolos bidimensionais** são figuras bidimensionais que podem ter tamanho variável e que podem ser expandidos de modo a conter outros elementos, tais como listas de strings ou outros símbolos bidimensionais. Podem ser divididos em compartimentos de tipo similar ou diferente. Os **caminhos** são sequências de linhas cujas extremidades estão conectadas a outros elementos. Caminhos podem ter terminadores. Por fim, os **strings** são textos apresentando diversas informações cujo significado depende de onde aparecem.

5.1 Estereótipos

Estereótipos são tipos especiais de Strings que servem para modificar a semântica de elementos de um diagrama. Assim, os estereótipos permitem que se definam novos elementos de modelagem em termos de elementos já conhecidos. Em princípio, estereótipos podem ser aplicados a qualquer elemento de modelagem. A notação de um estereótipo é na forma «estereótipo». Com o uso de estereótipos, novos tipos de diagramas podem ser criados dentro da linguagem UML. Veja alguns exemplos de estereótipos na figura 5.1 a seguir.

Nestes casos, as classes "Janela Principal", "Controle Janela", "Dados Iniciais" e "Usuário" são marcadas pelos estereótipos boundary, control, entity e actor. Essas classes, ditas classes estereotipadas, alternativamente podem ser representadas por símbolos gráficos diferenciados para representar esses elementos modificados. Observe que estereótipos podem ser aplicados a qualquer elemento de modelagem, não somente classes. Com isso, relacionamentos, componentes, etc… podem ser estereotipados também. Relacionamentos estereotipados podem adquirir uma notação diferente, tais como linhas tracejadas, etc, ou ainda ter simplesmente seu estereótipo declarado na forma de texto.

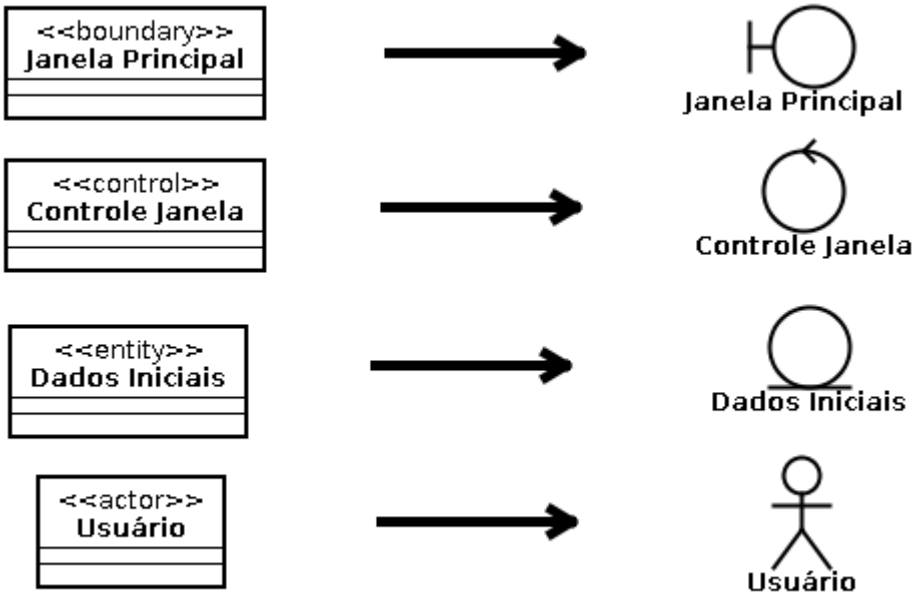


Figura 5.1: Exemplos de Estereótipos

Observe que cada elemento UML pode ter no máximo um único estereótipo. Assim, não é possível aplicarmos estereótipos a elementos que já sejam estereotipados. Alguns tipos de elementos contém um conjunto de estereótipos previamente definidos. Por exemplo, classes podem assumir estereótipos do tipo «actor», ou associações entre casos de uso podem assumir os estereótipos «extends» ou «include». O uso de estereótipos pode ser utilizado para modificar o modo de geração de código. Assim, pode-se usar um ícone diferente para modelar tipos de entidades em etapas específicas do processo de modelagem, tal como faremos por exemplo na fase de análise. Estereótipos podem ser utilizados em qualquer caso em que uma extensão é necessária para modelar alguma característica particular que os elementos padrões não evidenciam por si só. Na figura 5.2 a seguir, temos exemplos de diferentes visualizações de estereótipos.

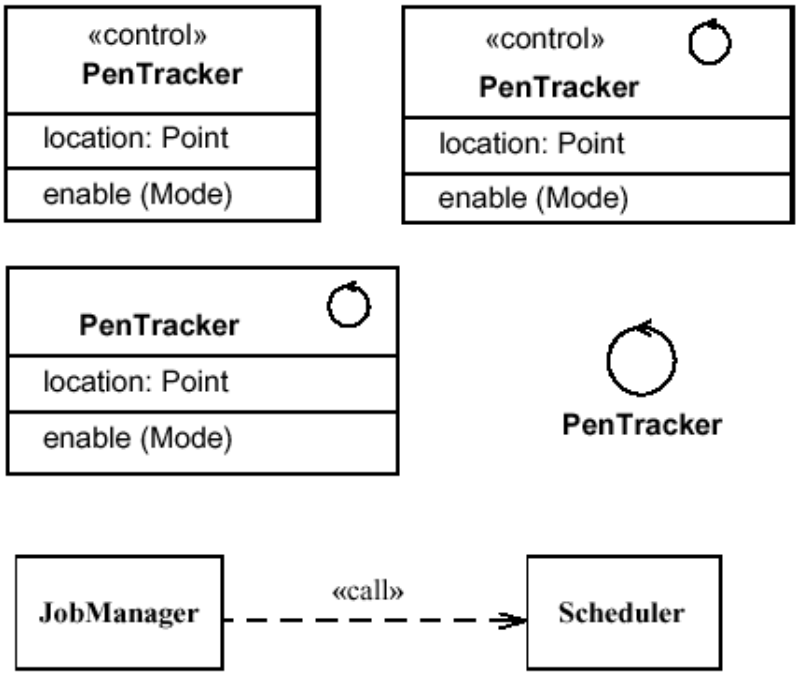


Figura 5.2: Exemplos de Diferentes Visualizações para Estereótipos

5.2 Notas

Notas são descrições de texto que complementam a informação contida em algum diagrama. As notas devem estar ancoradas a um elemento por meio de uma linha pontilhada. Um exemplo é dado na figura 5.3.

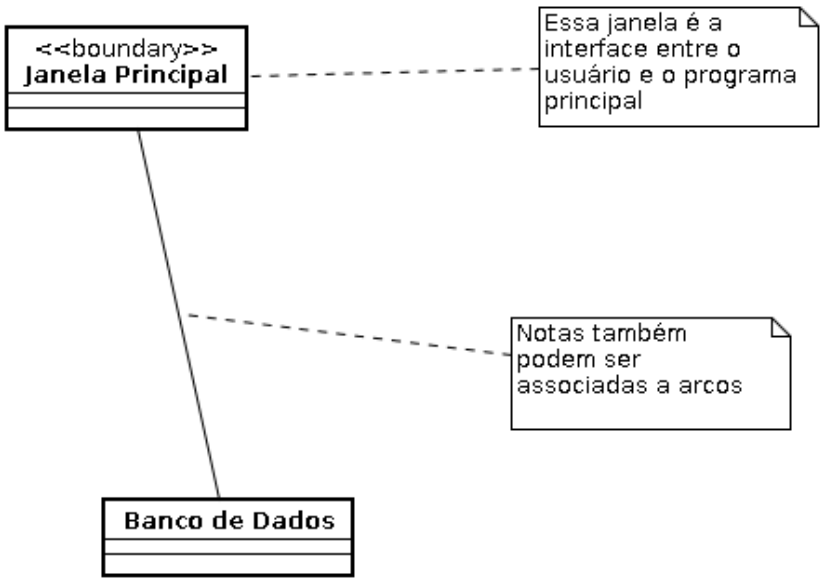


Figura 5.3: Exemplo de Nota

5.3 Pacotes

Pacotes são agrupamentos de elementos de modelagem em um único elemento, permitindo uma descrição do modelo em múltiplos níveis. Pacotes podem ser aninhados assim dentro de outros pacotes, de maneira recursiva, da mesma maneira que diretórios de arquivos em sistemas computacionais. Em princípio, qualquer elemento UML pode ser agrupado em um pacote. Pacotes podem também se relacionar com outros pacotes. Dentre os possíveis relacionamentos, os mais comuns são os relacionamentos de dependência e de generalização. Exemplos de pacotes podem ser vistos na figura 5.4 .

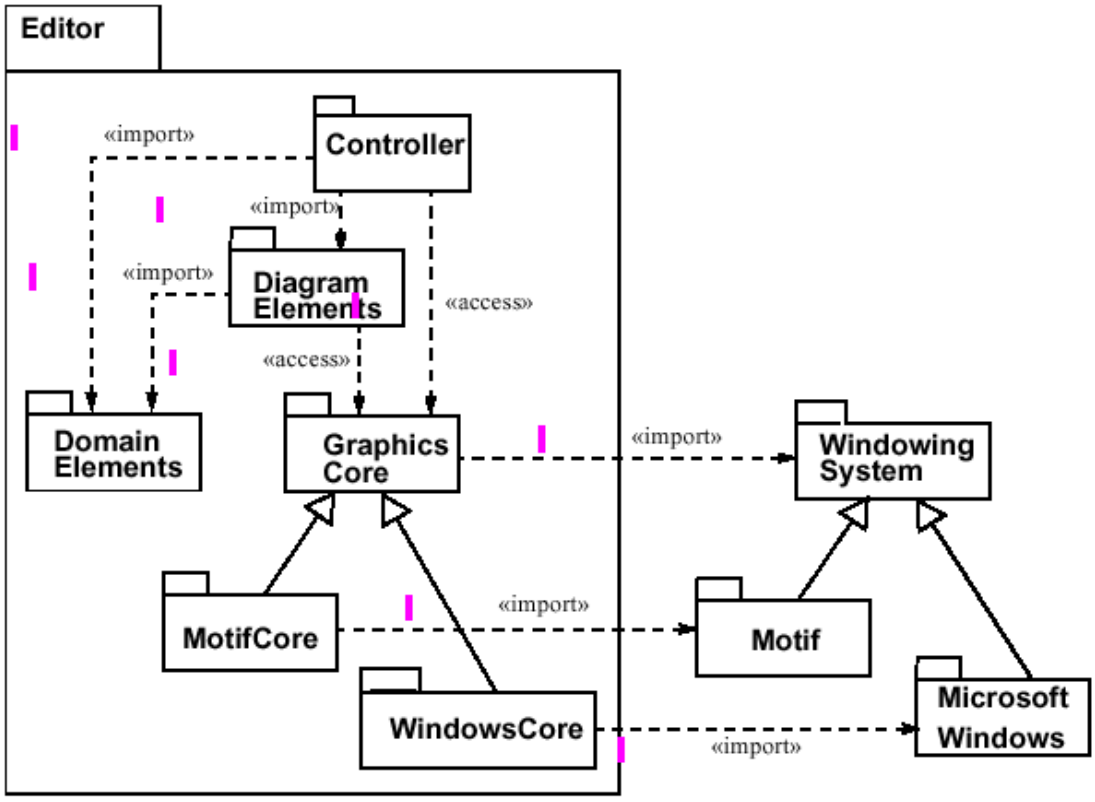


Figura 5.4: Exemplo de Pacotes

5.4 Sub-Sistemas

Sub-sistemas são um tipo de pacote específico, denotados pelo estereótipo «subsystem». Na verdade, subsistemas representam uma unidade comportamental em um sistema físico, ou seja, unidades que funcionam de maneira coesa e que podem ser vistas externamente como uma única entidade. Essa unidade pode oferecer interfaces e ter operações. Uma característica dos subsistemas é que seu conteúdo pode ser particionado em elementos de especificação e realização.

A especificação de um sub-sistema consiste na definição de operações sobre o sub-sistema, ao mesmo tempo que se definem outros elementos de especificação associado, tais como casos de uso ou máquinas de estado.

Subsistemas podem ou não instanciáveis. Sub-sistemas não instaciáveis servem meramente como unidades de especificação para o comportamento dos elementos nele contidos.

Um subsistema pode possuir compartimentos, que permitem a distribuição dos elementos que o compõem em espaços reservados, tais como na figura 5.5.

Podem também possuir interfaces, que permitem especificar um conjunto de operações para o acesso ao sub-sistema, como na figura 5.6.

A norma UML permite diversas diferentes notações para subsistemas. Exemplos dessas notações estão nas figuras 5.7,5. 8 e 5.9.

É possível fazer-se um mapeamento entre as partes de especificação e realização de um sub-sistema, usando os três compartimentos de um sub-sistema, conforme a figura 5.10.

Assim, os elementos de realização estão mapeados nos elementos de especificação e nas operações declaradas do sub-sistema. Da mesma forma, é possível fazermos um mapeamento do sub-sistema em uma interface. Entretanto, somente os elementos de realização com relevância são declarados, e esse mapeamento pode ser expresso de diferentes maneiras.

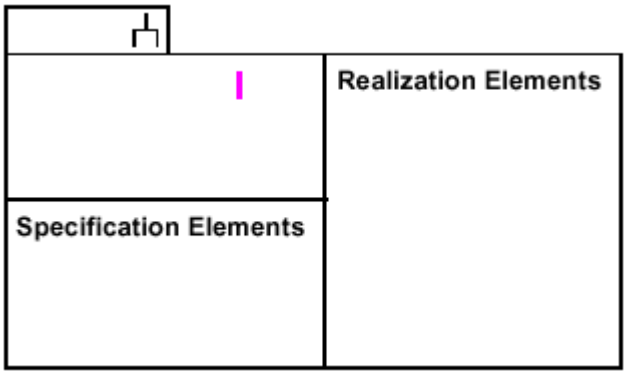


Figura 5.5: Exemplo de Subsistema com Compartimentos

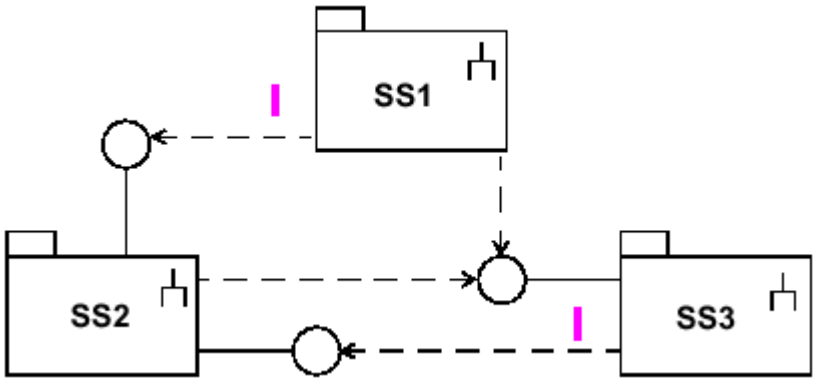


Figura 5.6: Exemplo de Subsistemas com Interfaces

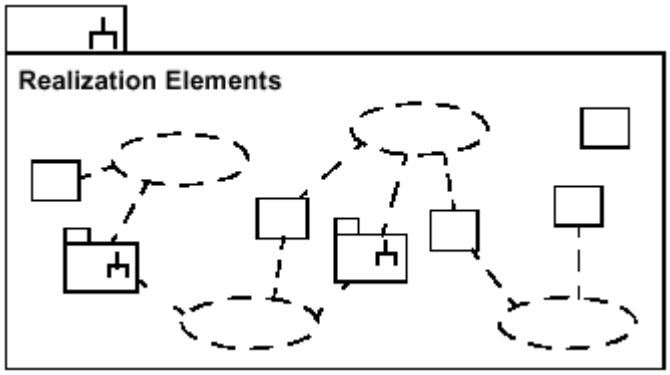


Figura 5.7: Exemplo de Diferentes notações para Subsistemas

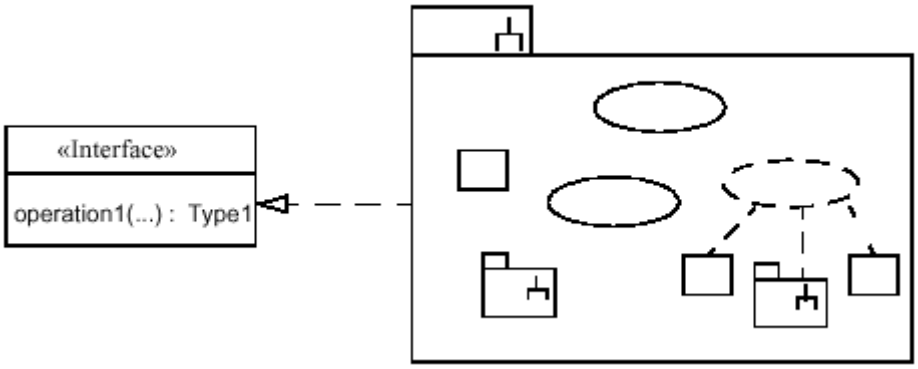


Figura 5.8: Exemplo de Notação de Subsistema

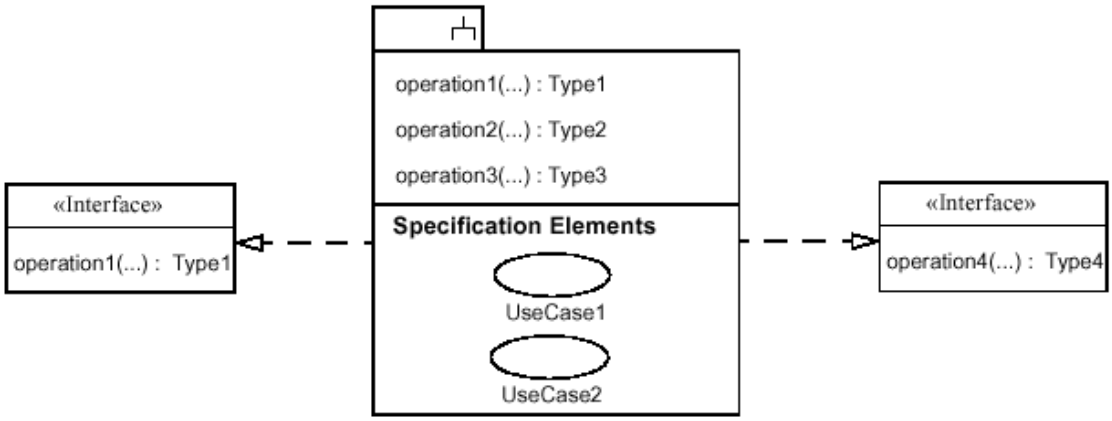


Figura 5.9: Exemplo de Subsistema com 2 Compartimentos

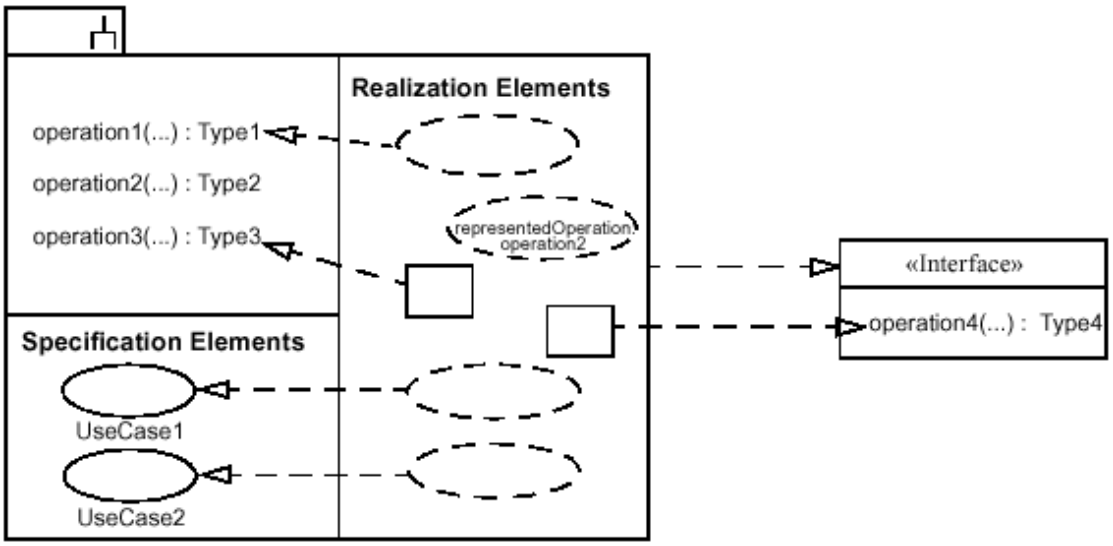


Figura 5.10: Exemplo de Mapeamento entre Elementos de Subsistemas

5.5 Restrições (*Constraints*)

Restrições são relacionamentos semânticos entre elementos de modelagem que especificam condições e proposições que necessitam ser mantidas. Algumas delas são pré-definidas, mas de maneira geral restrições também podem ser definidas pelo usuário. A figura 5.11 a seguir apresenta alguns exemplos de restrições:

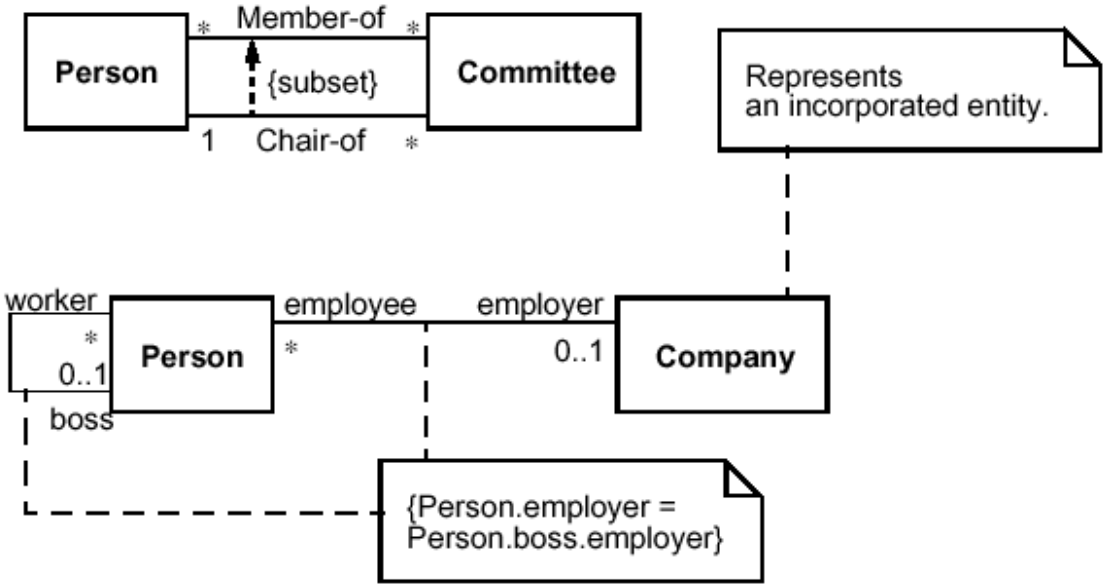


Figura 5.11: Exemplo de Restrições (Constraints)

5.6 Os Diagramas UML

Os diagramas UML podem, de uma maneira geral, se servir de todos estes expedientes de modelagem: estereótipos, tagged values, notas, pacotes, sub-sistemas e restrições. Detalhes dos diferentes diagramas serão introduzidos mais a frente, quando de sua utilização nas diversas etapas de modelagem. Uma visão geral do "jeitão" que marca cada tipo de diagrama poderá ser obtido a partir das atividades propostas a seguir. Os diagramas em questão são os seguintes:

- Diagramas Estruturais
    - Diagramas Estruturais Estáticos (Diagramas de Classes)
    - Diagramas de Componentes
    - Diagramas de Distribuição (*Deployment*)
- Diagramas Comportamentais
    - Diagramas de Casos de Uso
    - Diagramas de Interação
        - Diagramas de Sequência
        - Diagramas de Comunicação (Colaboração)
    - Diagramas de Atividades
    - Diagramas de Estado

6. Diagramas Estruturais Estáticos

Diagramas estruturais estáticos denotam a estrutura estática de um modelo em particular. Em diagramas deste tipo, representamos:

- Coisas que existem (tais como classes, tipos e objetos)
- A Estrutura Interna Dessas Coisas
- Relacionamentos entre essas coisas

Com isso, diagramas estruturais estáticos não mostram informações temporais relacionadas com os conceitos representados, somente informações estruturais. Existem basicamente dois tipos fundamentais de diagramas estruturais estáticos:

- Diagramas de Classes
- Diagramas de Objetos

Diagramas de classes mostram conjuntos de classes e tipos relacionados entre si. Podem ainda representar templates e classes instanciadas (objetos), alémde relacionamentos (associações e generalizações), bem como o conteúdo destas classes (atributos e operações)

Diagramas de objetos mostram instâncias compatíveis com algum diagrama de classes em particular e o relacionamento estrutural entre elas. A diferença básica que existe entre um diagrama de classes e um diagrama de objetos é que em diagramas de objetos preponderem as instâncias de classes, ao passo que em diagramas de classes preponderam as classes em si. Fundamentalmente, em termos formais ambos os diagramas são semelhantes. Uma diferença prática é que em diagramas de objetos podem haver diversas instâncias de uma mesma classe, pois o intuito é representar as instâncias, ao passo que em diagramas de classes o intuito é representar a organização das classes.

6.1 Diagramas de Classes

Formalmente falando, diagramas de classes são grafos de elementos do tipo *Classifier* conectados por diversos tipos de relacionamentos estáticos. Podem ainda conter pacotes e outros tipos de elementos gerais. Em princípio, um diagrama de classes representa uma visão do modelo estrutural estático, que pode ser entendidio como a união de todos os diagramas de classe e de objetos, da mesma maneira que podemos projetar uma figura tridimensional em diversos planos bi-dimensionais.

O tipo *Classifier* pode constituir-se de *Classes*, *Interfaces* e *DataTypes*. *Classes* são descritores para um conjunto de objetos com estrutura, comportamento e relacionamentos similares. Seu modelo diz respeito à **intensão** de uma classe, ou seja, as regras que a definem enquanto uma classe.

Exemplos de representações de classes podem ser vistas na figura 6.1.

Como se pode depreender dos diferentes exemplos, o símbolo gráfico utilizado para representar uma classe é uma caixa, possivelmente dividida em compartimentos. Esses compartimentos são

utilizados em diferentes situações, dependendo se a classe pertence a um modelo de análise, design ou implementação. Tais compartimentos possuem nomes default, mas podem ser nomeados também caso seja necessário.

O primeiro compartimento, de cima para baixo é chamado de compartimento do nome, contendo o nome da classe em questão e opcionalmente um estereótipo, um conjunto de propriedades (tagged-values) ou um ícone referente ao estereótipo.

Os compartimentos seguintes são chamados de compartimentos de listas, podendo acomodar listas de atributos, operações ou outros.

6.1.1 Compartimento de Atributos

O compartimento de atributos (normalmente o primeiro compartimento depois do compartimento do nome), é utilizado para mostrar os atributos de uma classe. A sintaxe padrão para a descrição dos atributos nesse compartimento é a seguinte:

```
visibility name [ multiplicity ] : type-expression = initial-value { property-string }
```

A visibilidade corresponde a um flag (+, # ou -) correspondendo a:

- + público
- # protegido
- - privado

A multiplicidade é usada para representar arrays (por exemplo [3] ou [2..*] ou ainda [0..7].

Os atributos de classe (também chamados de atributos estáticos) devem aparecer sublinhados.

6.1.2 Compartimento de Operações

O compartimento de operações mostram as operações definidas para uma classe e/ou os métodos supridos por uma classe. Sua sintaxe padrão é a seguinte:

```
visibility name ( parameter-list ) : return-type-expression { property-string }
```

onde visibilidade é semelhante ao usado para os atributos.

Cada elemento de parameter-list tem a seguinte sintaxe:

```
kind name : type-expression = default-value
```

onde **kind** deve ser **in**, **out**, ou **inout**

O UML define algumas propriedades como default. Exemplos de propriedades default incluem:

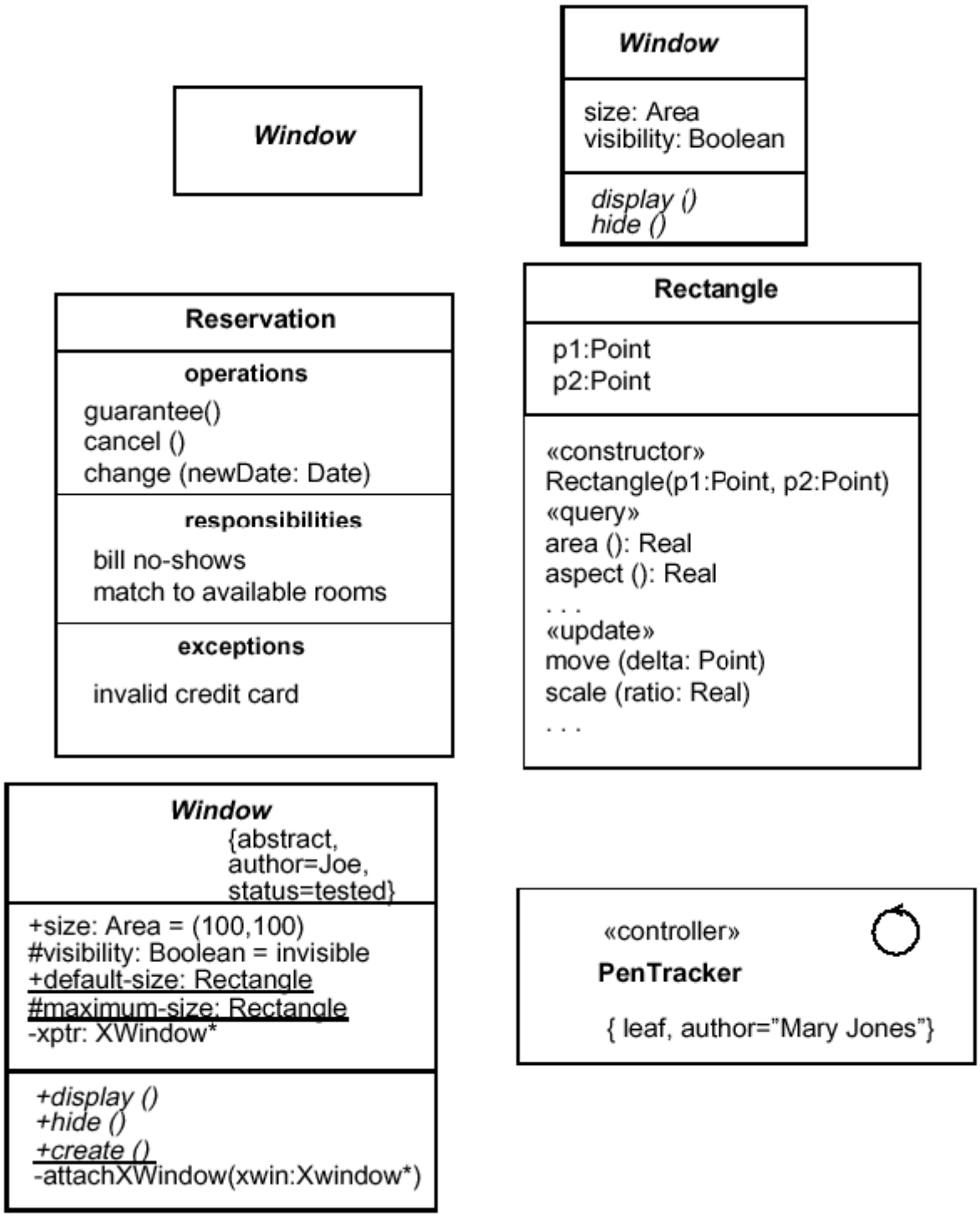


Figura 6.1: Exemplos de Diferentes Notações de Classes

- **`{query}`** a operação não modifica o estado do sistema
- **`{concurrency = name}`**, onde **`name`** deve ser um dos nomes: **`sequential`**, **`guarded`**, **`concurrent`**
- **`{abstract}`** especifica operações não dotadas de implementação

6.1.3 Classes Conceituais e Classes de Implementação

Classes podem ser de dois tipos básicos: Classes Conceituais ou Classes de Implementação.

Classes Conceituais podem não incluir métodos, mas simplesmente prover especificações comportamentais para sua operação. Podem ter atributos e associações.

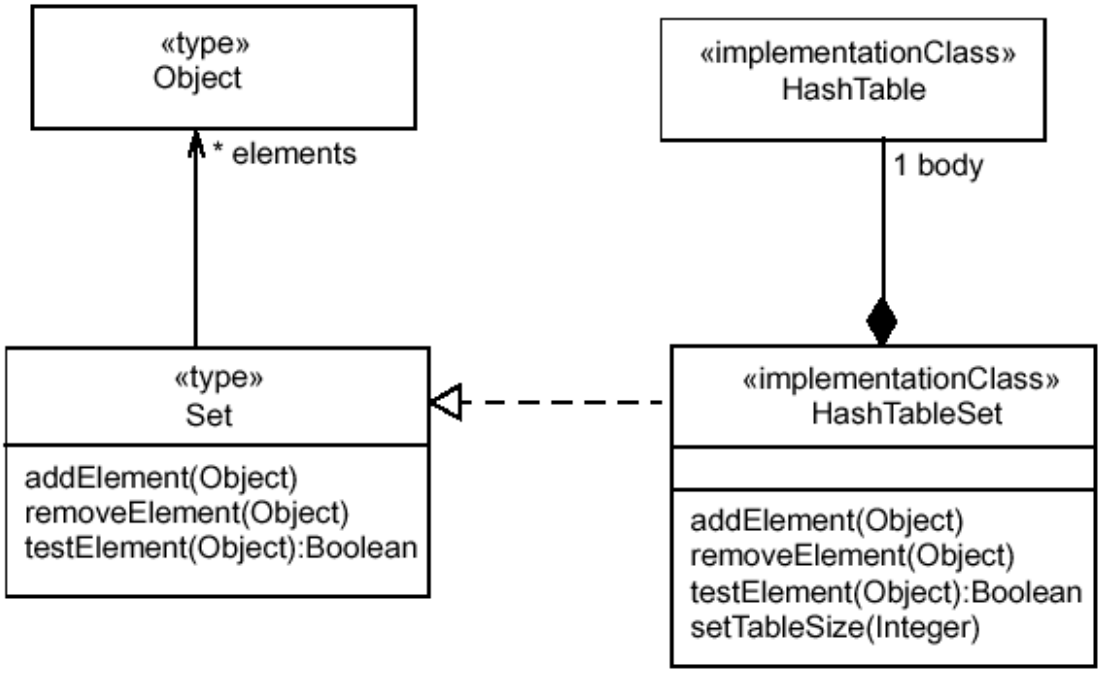


Figura 6.2: Classes Conceituais e Classes de Implementação

Classes de Implementação definem uma estrutura de dados física (para atributos e associações) e métodos de um objeto como implementados em linguagens tradicionais (C++, Java, etc). Normalmente dizemos que uma classe de implementação **realiza** uma classe conceitual, se ela provê todas as operações especificadas em uma classe conceitual. Dessa forma, é pssível que uma única classe de implementação realize diversas classes conceituais.

Veja o exemplo da figura 6.2. Nesse exemplo, as classes da esquerda são classes conceituais (devido à presença do estereótipo «type»).As classes da direita são classes de implementação (devido à presença do estereótipo «implementationClass»).

6.1.4 Interfaces

Interfaces são especificadores para um conjunto de operações externamente visíveis de uma classe, componente ou outro tipo de classifier (incluindo um sub-sistema) sem a especificação de sua estrutura interna. Desta forma, cada interface especifica somente uma parte limitada do comportamento de uma classe. Outra característica é que interfaces não possuem implementação. Da mesma forma, não possuem atributos, estados ou associações, mas somente operações.

Interfaces podem ter relacionamentos do tipo generalização. Formalmente, são equivalentes a uma classe abstrata sem atributos e sem métodos, com somente operações abstratas.

A notação para interfaces no UML é a de uma classe sem compartimento de atributos, com o estereótipo «interface» ou simplesmente um círculo (visão estereotipada da interface). Uma dependência abstrata entre uma classe e uma interface pode ser representada por uma linha do tipo generalização tracejada ou por uma linha cheia ligada a um círculo representando a interface, conforme na figura 6.3.

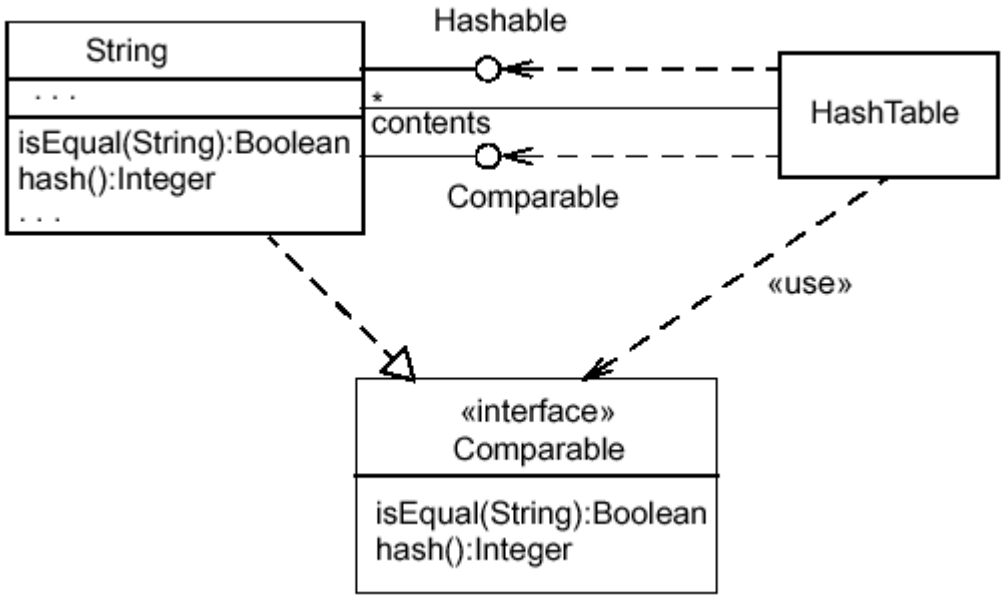


Figura 6.3: Exemplos de Interfaces

Observe na figura que a interface Comparable está representada na parte inferior na forma de uma classe estereotipada e um pouco mais acima na forma de um círculo. A classe String, a esquerda, implementa a interface Comparable. Essa implementação está representada de duas maneiras na figura: primeiro por meio da generalização tracejada abaixo, e segundo por meio da associação (linha cheia) entre a classe String e a representação estereotipada da interface Comparable (na forma de um círculo, mais acima).

6.1.5 Classes e Pacotes

Cada diagrama de classes corresponde a um pacote na notação UML. Da mesma forma, mais de um pacote podem aparecer no mesmo diagrama. Em algumas situações, pode ser necessário referenciarmos classes em outros pacotes não disponíveis no diagrama corrente. Neste caso, a classe deve ser referenciada da seguinte forma:

```
Package-name :: class-name
```

Um exemplo desse uso está apresentado na figura 6.4 a seguir:

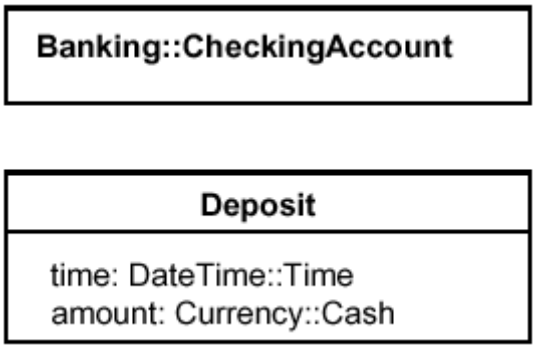


Figura 6.4: Exemplo de Classes Oriundas de Outros Pacotes

Da mesma maneira um elemento pode referenciar elementos em outros pacotes. Com isso, torna-se possível criar-se diversos níveis internos de pacotes. Quando se deseja referenciar um tipo de dependência entre pacotes, pode-se utilizar o estereótipo «access» sobre a dependência para indicar que o conteúdo de um pacote faz referência a elementos do outro pacote. A única restrição é que a visibilidade entre os pacotes seja condizente. Dependências estereotipadas com «import», ao contrário, liberam o acesso e automaticamente carregam os nomes com a visibilidade apropriada no espaço de nomes do pacote, fazendo a importação, o que que evita o uso de nomes de caminhos para referenciar as classes. Um exemplo de dependência estereotipada é mostrada na figura 6.5 a seguir:

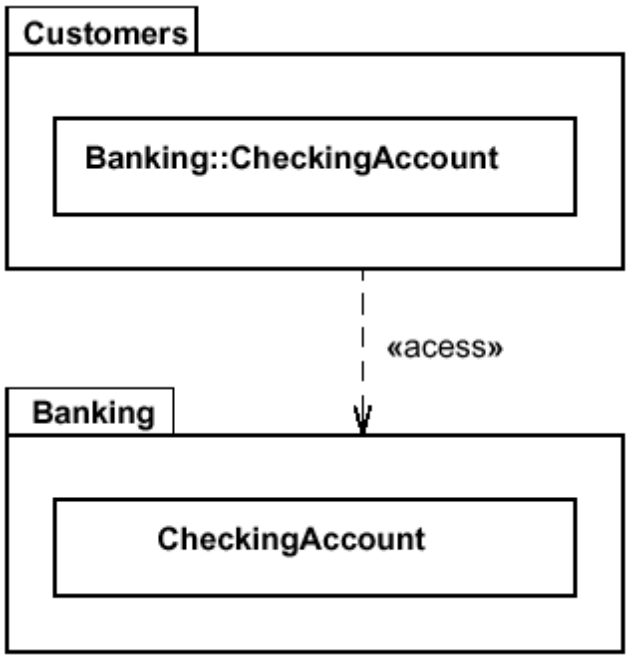


Figura 6.5: Exemplo de Dependência Estereotipada

6.2 Diagramas de Objetos

Diagramas de objetos são grafos de instâncias de classes, incluindo objetos e variáveis. Podemos dizer que sob certo aspecto um diagrama de classes não deixa de ser uma instância de um diagrama de classes mostrando detalhadamente o estado do sistema em um determinado instante de tempo. O formato do diagrama de objetos, como já dissemos, é semelhante ao do diagrama de classes, sendo diferenciado somente por seu uso. O uso de diagramas de objetos é normalmente limitado, sendo utilizado para mostrar exemplos de estruturas de dados em pontos estratégicos do sistema. Cada objeto em um diagrama de objetos representa uma instância particular de uma classe, tendo uma identidade e um conjunto de valores de atributos que lhe são próprios.

A notação para objetos no UML é derivada da notação de classe, com a ressalva que o nome do objeto/classe aparece sublinhado. Assim, quando o compartimento de nome de uma classe aparece sublinhado, não se trata de uma classe, mas sim de uma instância desta, ou seja, de um objeto. A caixa de objetos pode ter dois compartimentos: o primeiro mostra o nome do objeto, seu estereótipo e propriedades:

`objectname : classname`

onde **`classname`** pode incluir o caminho completo do pacote onde se encontra a classe que o objeto referencia, por exemplo:

`display_window: WindowingSystem::GraphicWindows::Window`

ou seja, o objeto **`display_window`** é um objeto da classe **`Window`**, que fica no pacote **`GraphicWindows`** que fica no pacote **`WindowingSystem`**.

Caso haja herança múltipla, as classes devem ser separadas por vírgulas.

O segundo compartimento de um objeto (caso ele exista), pode mostrar os valores específicos dos atributos do objeto, na forma de uma lista, onde cada linha deve ser como segue:

`attributename : type = value`

Objetos compostos são representações de objetos de alto nível, ou seja, que contém outros objetos como partes. Na figura 6.6 a seguir, vemos diversos exemplos de objetos:

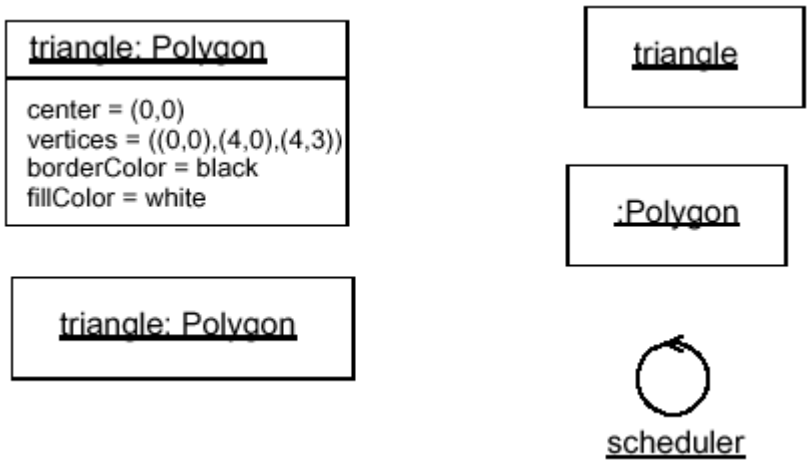


Figura 6.6: Exemplos de Objetos

A figura 6.7 a seguir mostra um exemplo de um objeto composto:

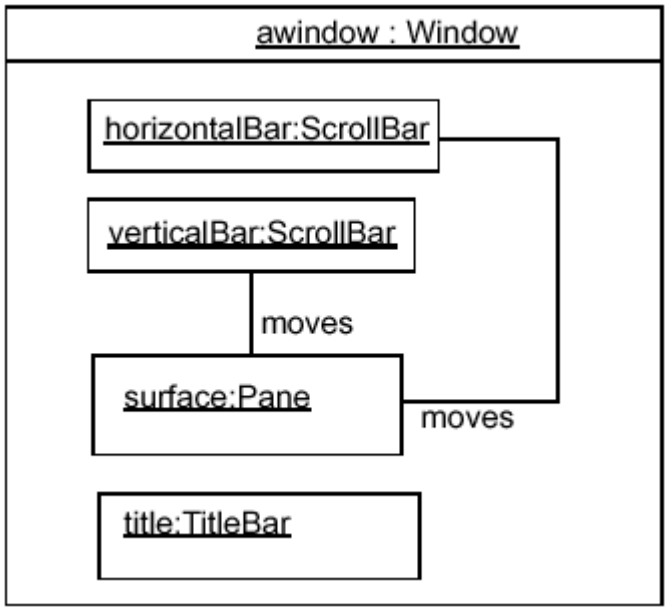


Figura 6.7: Exemplo de Objeto Composto

6.3 Relacionamentos entre Classes e Objetos

Classes e objetos podem estar conectados por algum tipo de relacionamento. A linguagem UML prevê três tipos básicos de relacionamentos:

- Associações
- Generalizações
- Dependências

As associações podem ainda ser sub-divididas em três sub-tipos básicos:

- Associações Simples
- Agregações
- Composições

Os relacionamentos podem ser binários, ternários ou de ordem superior. Relacionamentos binários são mostrados como linhas conectando dois símbolos do tipo *classifier*. Essas linhas podem ter uma variedade de decorações para diferenciar suas propriedades. Relacionamentos ternários ou de ordem superior podem ser exibidos como diamantes conectados por linhas a símbolos de *classifers*.

6.3.1 Associações Simples

Associações simples representam que existe alguma conexão entre dois elementos do tipo *classifier*, de tal forma que um deve manter alguma referência ao outro. Associações simples são representadas na forma de uma linha cheia conectando os dois *classifier*. Uma associação simples pode possuir ainda um nome e duas extremidades. O nome da associação é grafado na forma de um *String*, posicionado normalmente próximo ao centro da linha que representa a associação. As extremidades podem possuir ainda uma **navegabilidade**, uma **multiplicidade** e um **papel**. Observe a figura 6.8 a seguir:

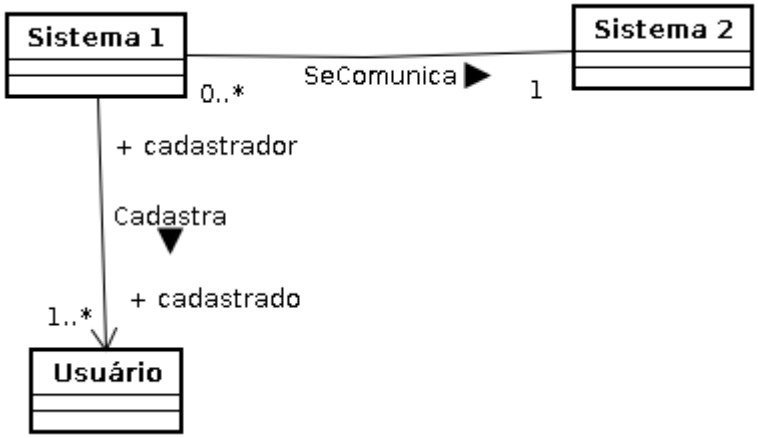


Figura 6.8: Exemplo de Associação Simples

A classe *Sistema1* está associada à classe *Sistema2* por meio de uma associação com nome *SeComunica*. Essa associação representa que um objeto do tipo *Sistema1* pode se comunicar com apenas 1 único (vide a multiplicidade na extremidade próxima a *Sistema2*) objeto do tipo *Sistema2*. Entretanto, um objeto de tipo *Sistema2* pode se comunicar com 0 ou mais objetos do tipo *Sistema1*. Nesse caso, como não existe nenhum tipo de decoração na ponta das linhas de *SeComunica*, dizemos que essa associação é bi-direcional. Na outra associação apresentada, entre *Sistema1* e *Usuário*, temos um exemplo de uma associação uni-direcional. Isso ocorre pois existe uma decoração chamada de navegabilidade na extremidade de *Cadastra* que se conecta a *Usuário*. A leitura da multiplicidade se faz da seguinte maneira: um objeto do tipo *Sistema1* pode cadastrar 1 ou mais objetos do tipo *Usuário*. Como a associação é unidirecional, não há associação na direção de *Usuário* a *Sistema1*. Observe ainda o símbolo > após o nome da associação (*Cadastra*). Esse símbolo indica a leitura que se deve fazer. Assim, é o *Sistema1* quem cadastra o *Usuário*, e não vice-versa. A figura apresenta ainda os papéis associados às extremidades da associação *Cadastra*. Nesse caso, o *Sistema1* é o cadastrador (papel público, pois aparece o +), e o *Usuário* é o cadastrado (também público).

Podem existir associações do tipo ou-exclusivo, chamadas também de associações XOR. Associações desse tipo indicam uma situação onde somente uma dentre diversas potenciais associações podem ser instanciadas em um determinado instante, para uma dada instância. Qualquer instância do classificador poderá participar de somente uma das associações indicadas. Este é um exemplo da aplicação de uma restrição a uma associação. Um exemplo de uma associação XOR é apresentado na figura 6.9 a seguir:

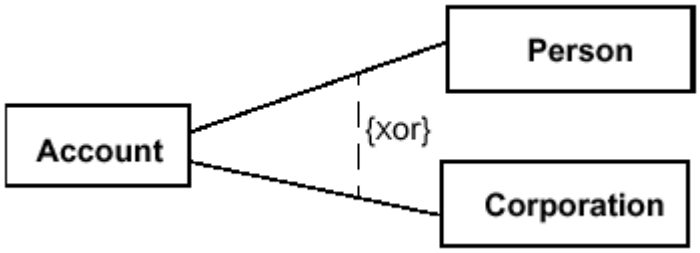


Figura 6.9: Exemplo de Associação do tipo XOR

6.3.2 Qualificadores

Qualificadores são atributos ou listas de atributos cujos valores servem para particionar o conjunto de instâncias associadas a uma instância através de uma associação. Assim, podemos entender que qualificadores são atributos de uma associação. Exemplo do uso de qualificadores são mostrados na figura 6.10 a seguir:

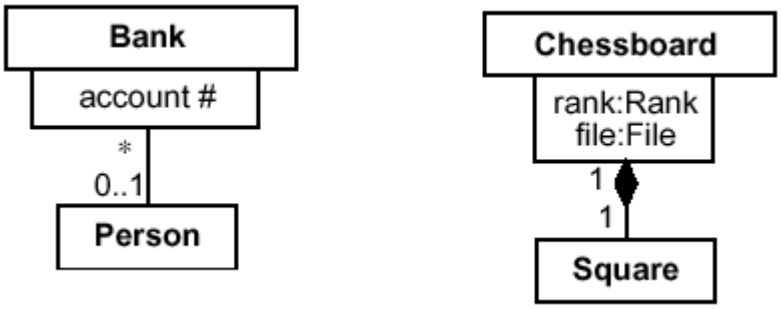


Figura 6.10: Exemplo de Qualificadores

6.3.3 Agregações e Composições

Agregações são um tipo especial de associação onde o elemento associado corresponde a uma parte do elemento principal. Composições são um tipo especial de agregação onde necessariamente a parte indicada deve existir. Um exemplo contendo uma agregação e uma composição é mostrado na figura 6.11 a seguir:

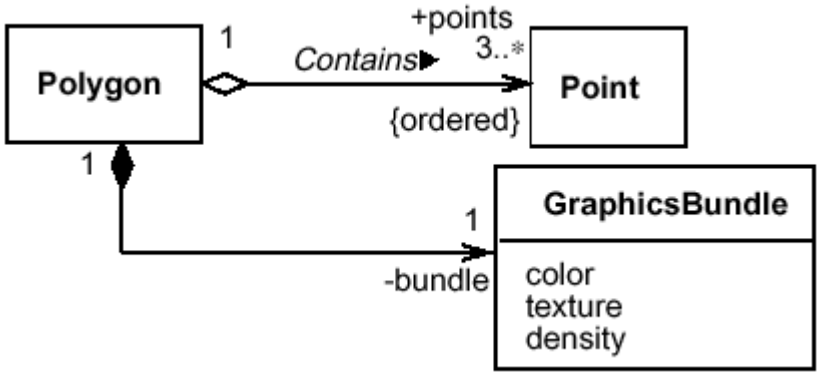


Figura 6.11: Exemplo de Agregação e Composição

Um objeto da classe *Polygon* pode conter diversos objetos da classe *Point*, entretanto terá somente um único objeto da classe *GraphicsBundle*. A diferença básica entre uma agregação e uma composição é que na agregação, o número de partes associadas à classe principal é variável e pouco importa. No caso de uma composição, esse número é definido, de tal forma que não faz sentido pensarmos o objeto da classe principal sem os objetos que o compõem. O melhor exemplo para uma composição é a idéia de uma *Mão*, que é formada pela composição de 5 objetos da classe *Dedo*.

Existem diversas maneiras de representar uma composição. A maneira da figura acima é uma delas. Outras maneiras são apresentadas na figura 6.12 abaixo.

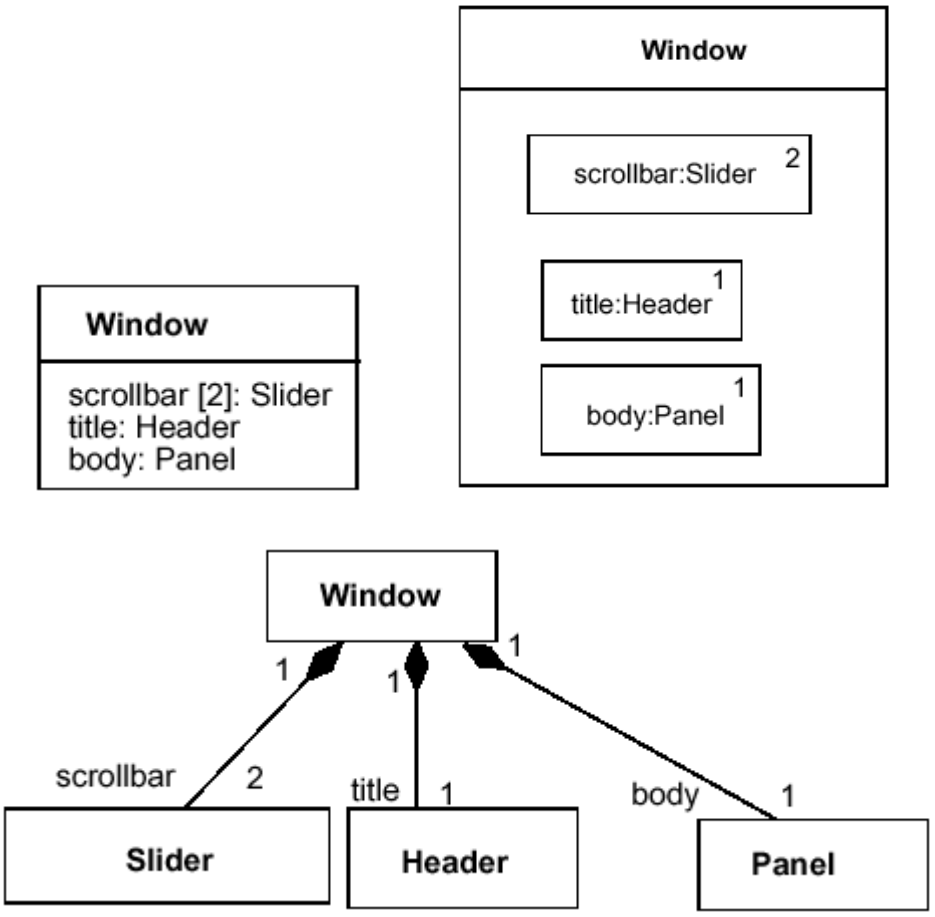


Figura 6.12: Exemplos de Composição

6.3.4 Classe de Associação

Quando uma associação necessitar uma representação diferenciada, por exemplo, tendo atributos ou operações associadas, podemos utilizar o conceito de uma classe de associação. Uma classe de associação é uma associação que ao mesmo tempo possui propriedades de uma classe (ou uma classe que tem propriedades de uma associação. Uma classe de associação corresponde a um único elemento, apesar de seu aspecto dual. Um exemplo de classe de associação é apresentado na figura 6.13 a seguir:

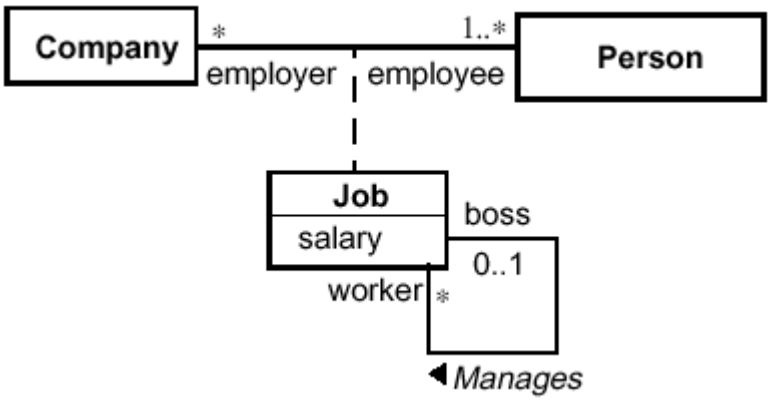


Figura 6.13: Exemplo de Classe de Associação

6.3.5 Associações N-árias

Associações n-árias são associações entre três ou mais classifiers (onde um mesmo classifier pode aparecer mais de uma vez). Neste caso, a multiplicidade em um papel representa o número potencial de instâncias de uma tupla na associação quando os outros N-1 valores são definidos. Associações n-árias não podem conter marcadores de agregação. Um exemplo de uma associação n-ária é apresentada na figura 6.14 a seguir:

Figura 6.14: Exemplo de Associação N-ária

6.3.6 Generalizações

Generalizações são relacionamentos taxonômicos entre um elemento mais geral (o pai) e um elemento mais específico (o filho) que deve ser consistente com o primeiro elemento e que adiciona informações adicionais. Generalizações podem ser utilizadas para classes, pacotes, casos de uso e outros elementos. Exemplos de generalizações são apresentados na figura 6.15 a seguir:

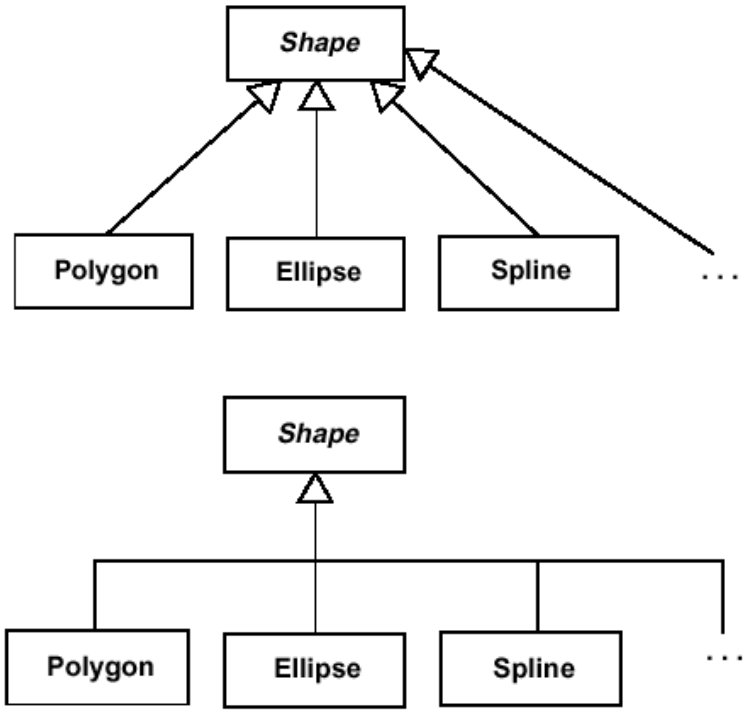


Figura 6.15: Exemplos de Generalizações (Herança)

A figura 6.16 a seguir apresenta generalizações com restrições e descrições:

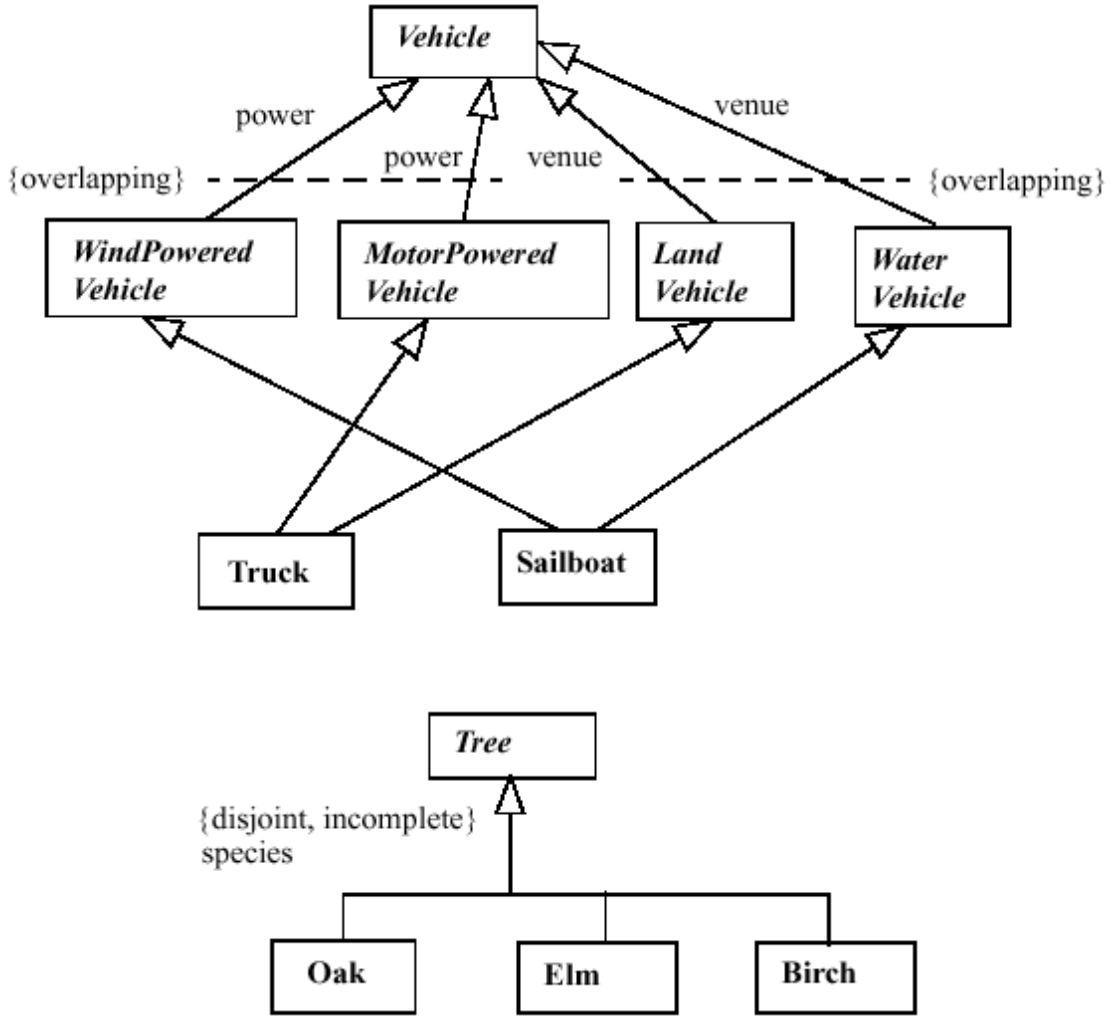


Figura 6.16: Exemplos de Generalização com Restrições e Descrições

6.3.7 Dependências

Dependências indicam um relacionamento semântico entre os dois elementos de modelagem (ou conjunto de elementos de modelagem). As dependências relacionam os elementos de modelagem por si só, não demandando um conjunto de instâncias para seu significado, e normalmente indicam situações em que uma mudança em um dos elementos pode demandar uma mudança no elemento que dele depende.

A linguagem UML estabelece ainda um conjunto de estereótipos padrões para dependências: access, bind, derive, import, refine, trace e use.

Outro recurso é a indicação de elementos derivados por meio de dependências, tais como na figura 6.17 a seguir:

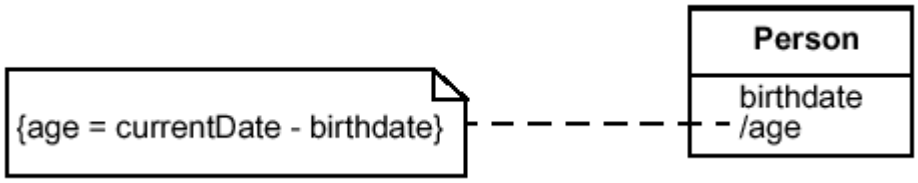


Figura 6.17: Exemplo de Dependência

Outros exemplos de dependências podem ser vistos na figura 6.18 a seguir:

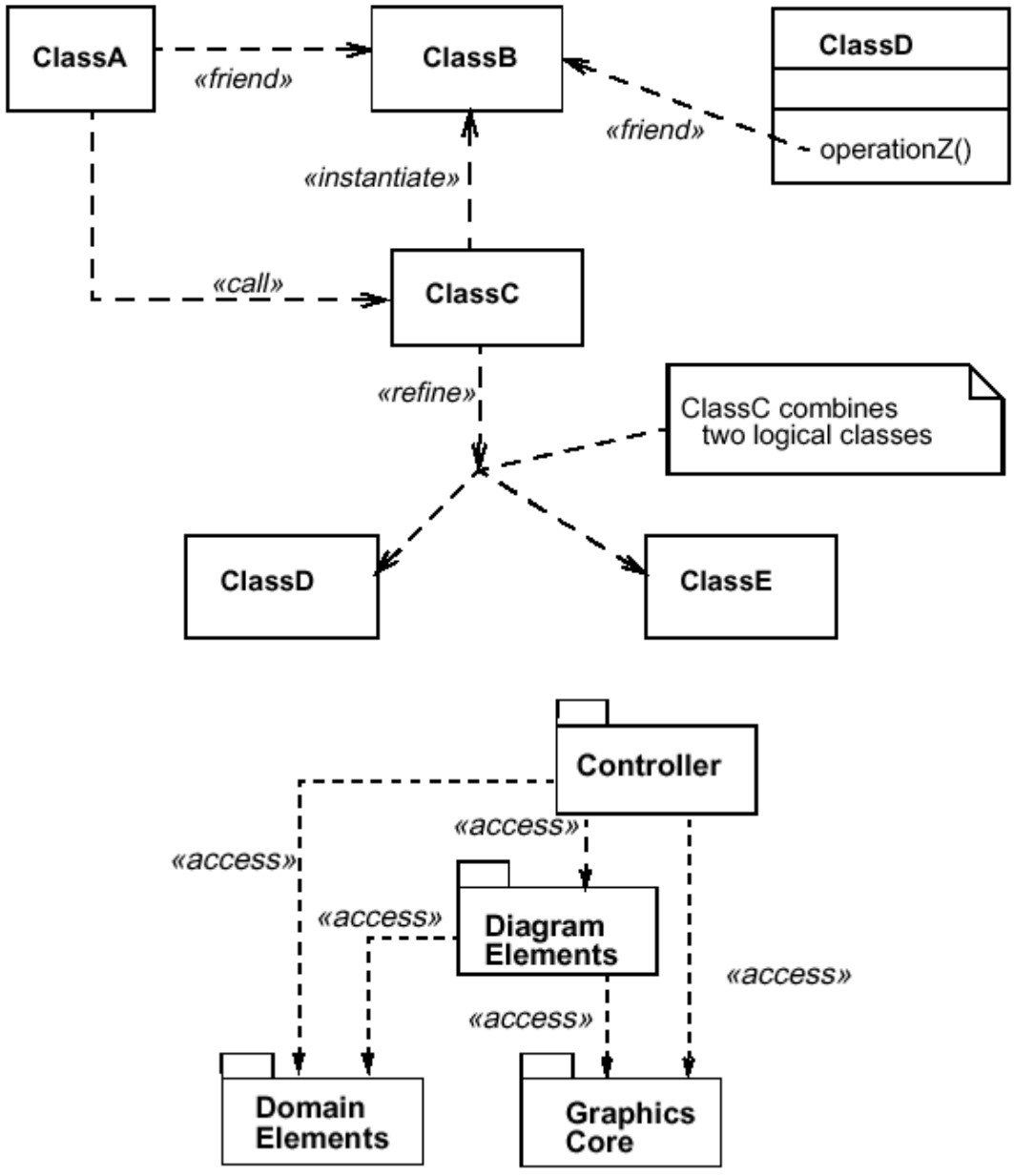


Figura 6.18: Outros Exemplos de Dependências

7. Diagramas de Componentes

7.1 Componentes

A linguagem UML especifica um conjunto de construtos que podem ser utilizados para definir sistemas de software de tamanho e complexidade arbitrários. Em particular, define-se o conceito de **componente**, como uma unidade modular com um conjunto de interfaces bem definidas, que pode ser substituído dentro de seu ambiente. O conceito de componente é oriundo da área de desenvolvimento baseado em componentes, onde um componente é modelado durante o ciclo de desenvolvimento e refinado sucessivamente durante a instalação e execução do sistema.

Um aspecto importante do desenvolvimento baseado em componentes é o reuso de componentes previamente construídos. Um componente pode ser considerado como uma unidade autônoma dentro de um sistema ou subsistema. Ele deve possuir uma ou mais **interfaces**, que podem ser potencialmente disponibilizadas por meio de **portas**, e seu interior é normalmente inacessível. O acesso a um componente deve ocorrer única e exclusivamente por meio de suas interfaces. Apesar disso, um componente pode ser dependente de outros componentes, e a linguagem UML provê mecanismos para representar essa dependência, indicando as interfaces que um componente demanda de outros componentes. Esse mecanismo de representação de dependências torna o componente uma unidade encapsulada, de forma que o componente pode ser tratado de maneira independente. Como resultado disso, componentes e subsistemas podem ser reutilizados de maneira bastante flexível, sendo substituídos por meio da conexão de diversos componentes por meio de suas interfaces e dependências.

A linguagem UML suporte a especificação tanto de **componentes lógicos** (e.g. componentes de negócios, componentes de processo) como de **componentes físicos** (e.g. componentes EJB, componentes CORBA, componentes COM+ ou .NET, componentes WSDL, etc), junto com os artefatos que os implementam e os nós em que esses componentes são instalados e executados.

Um componente possui um caráter hierárquico, que pode ser explorado no processo de construção de um sistema. Desta forma, um componente pode ser visto, do ponto de vista externo, como um elemento executável do sistema, que se conecta com outros componentes para integrar a composição deste sistema. Por outro lado, de uma perspectiva interna, um componente pode ser visto como também um conjunto integrado de componentes menores que se integram, constituindo o componente em seu nível superior. Dessa forma, um componente representa uma parte modular de um sistema que encapsula seu conteúdo e cuja manifestação pode ser substituído no ambiente em que se insere.

Um componente define seu comportamento por meio da definição de suas interfaces disponibilizadas e das interfaces de que depende. Dessa forma, um componente pode ser substituído por outro componente que possua um mesmo conjunto de interfaces disponibilizadas. A construção de um sistema envolve a montagem de componentes, uns aos outros, por meio de suas interfaces. Um arranjo de componentes montado desta maneira, constitui-se de um **artefato**. Artefatos podem ser instalados em diferentes ambientes de execução e colocados em execução nestes.

Um componente é uma unidade auto-contida que encapsula o estado e o comportamento de um grande número de objetos. Um componente especifica um contrato formal dos serviços que provê a seus clientes e dos serviços que necessita de outros componentes em termos de suas interfaces disponibilizadas e requeridas.

Um componente é uma unidade substituível que pode ser remanejada tanto durante o design como na implementação, por outro componente que lhe seja funcionalmente equivalente, baseado na compatibilidade entre suas interfaces. De modo similar, um sistema pode ser estendido adicionando-se novos componentes que tragam novas funcionalidades. As interfaces disponibilizadas e requeridas podem ser organizadas opcionalmente por meio de portas. Portas definem um conjunto de interfaces disponibilizadas e requeridas que são encapsuladas de maneira conjunta.

Certo número de estereótipos padrão são previstos para serem utilizados com componentes. Por exemplo, o estereótipo «subsystem» pode ser utilizado na modelagem de componentes de larga-escala. Os estereótipos «specification» e «realization» podem ser utilizados para representar componentes com especificações e realizações distintas, onde uma única especificação pode ter múltiplas realizações.

A notação UML para um componente segue a notação de uma classe estereotipada com o estereótipo «component». Opcionalmente, no canto superior direito, pode-se incluir um ícone específico, constituído por um retângulo maior e dois pequenos retângulos menores sobressaindo-se em seu lado esquerdo. Essa representação pode ser visualizada na figura 7.1 a seguir.

Figura 7.1: Exemplo da Notação de um Componente, com sua Interface disponibilizada

Na figura 7.2, apresenta-se um componente com suas interfaces disponibilizadas e requeridas. Neste exemplo, o componente Order disponibiliza as interfaces ItemAllocation e Tracking, e requer as interfaces Person, Invoice e OrderableItem.

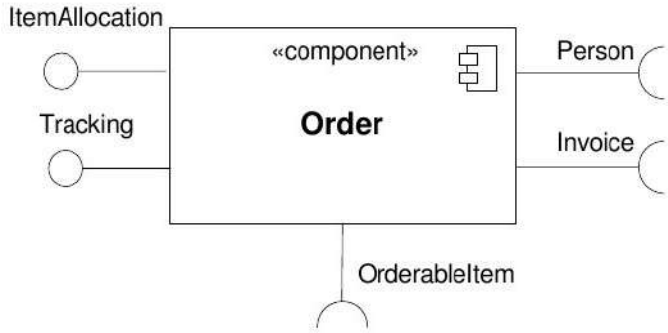


Figura 7.2: Exemplo da Notação de um Componente com Interfaces Disponibilizadas e Requeridas

A figura 7.3 a seguir apresenta outras notações para componentes, com diferentes compartimentos modelando diferentes aspectos do componente. Em ambos os casos, as interfaces dispobilizadas e requeridas estão representadas internamente em compartimentos, ao invés de externamente, por meio da conexão com interfaces.

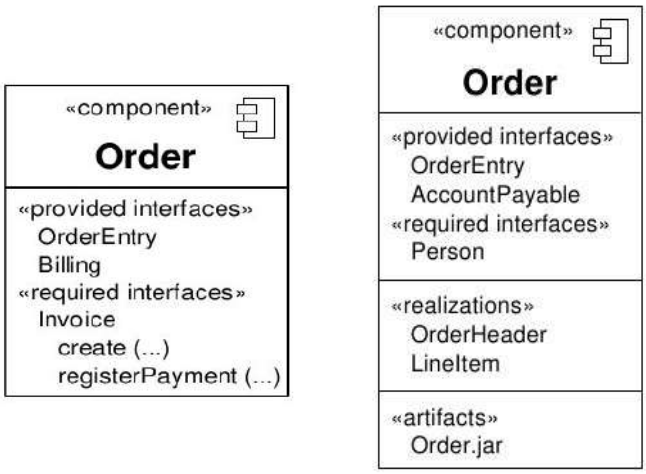


Figura 7.3: Outros Exemplos de Notação para Componentes

A figura 7.4 apresenta uma outra notação alternativa, onde as interfaces disponibilizadas e requeridas são modeladas detalhadamente.

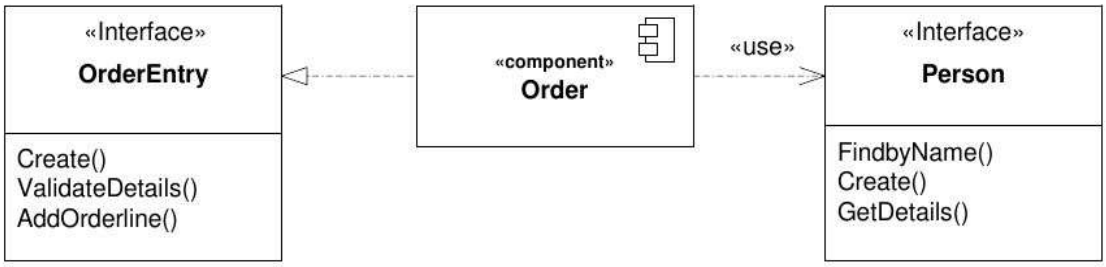


Figura 7.4: Notação para Componentes com Interfaces Detalhadas

A figura 7.5 a seguir, apresenta um exemplo em que as classes utilizadas para implementar um componente são mostradas de maneira explícita.

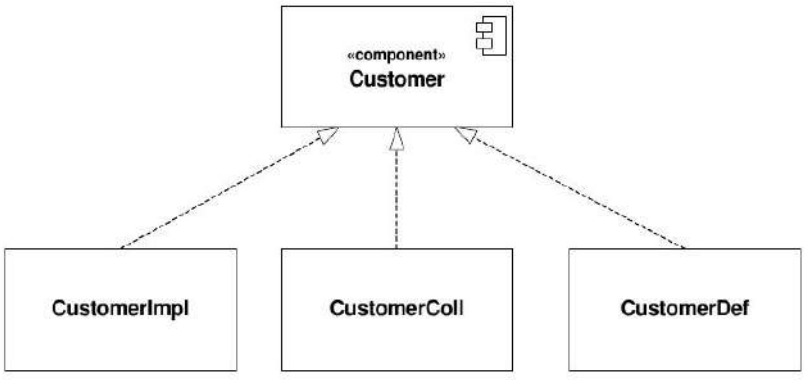


Figura 7.5: Classes que realizam um Componente

A figura 7.6 apresenta a estrutura interna de um componente, realizado pela composição de diversos outros componentes. Observe como as interfaces requeridas de alguns componentes são interligadas às interfaces disponibilizadas por outros componentes, para a montagem do componente de nível superior.

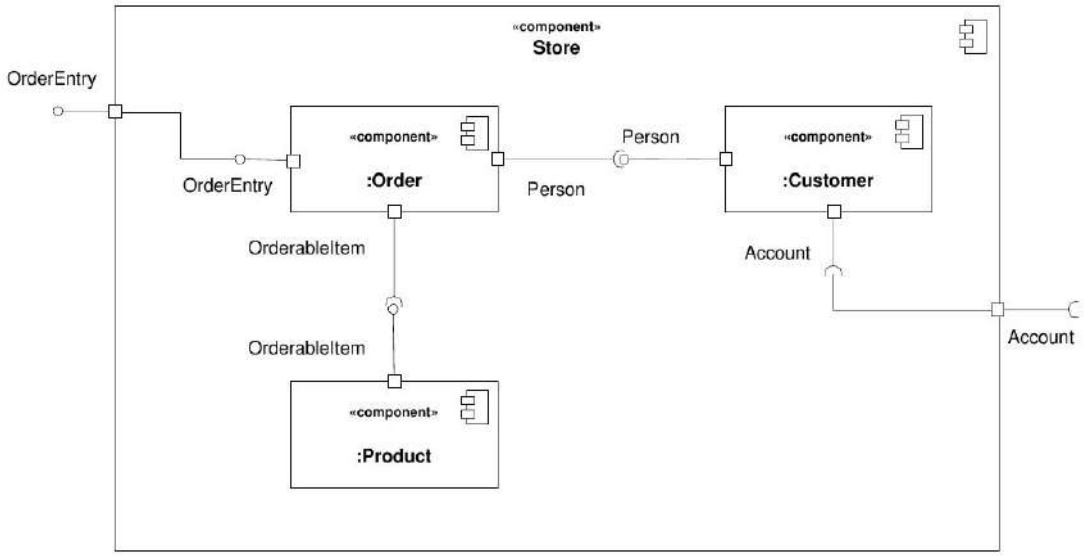


Figura 7.6: Perspectiva Interna de um Componente

Observe também na figura 7.6 acima, a definição de portas, representadas como pequenos quadrados onde as interfaces requeridas e disponibilizadas são conectadas aos componentes. Uma visão alternativa representando a dependência do componente **Order** dos componentes **Account** e **Product** pode ser visualizada na figura 7.7 a seguir:

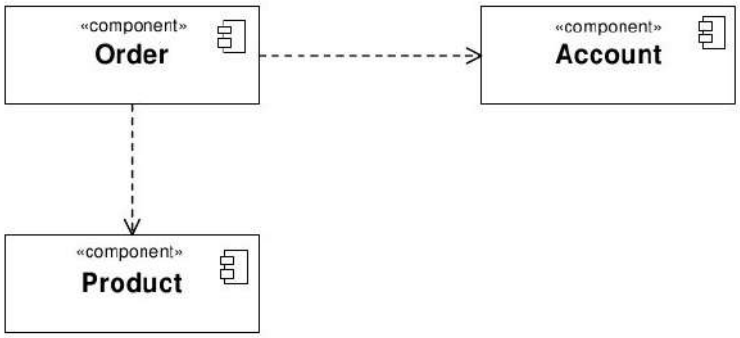


Figura 7.7: Representação alternativa para a dependência entre componentes

Uma outra representação alternativa pode ser vista ainda na figura 7.8.

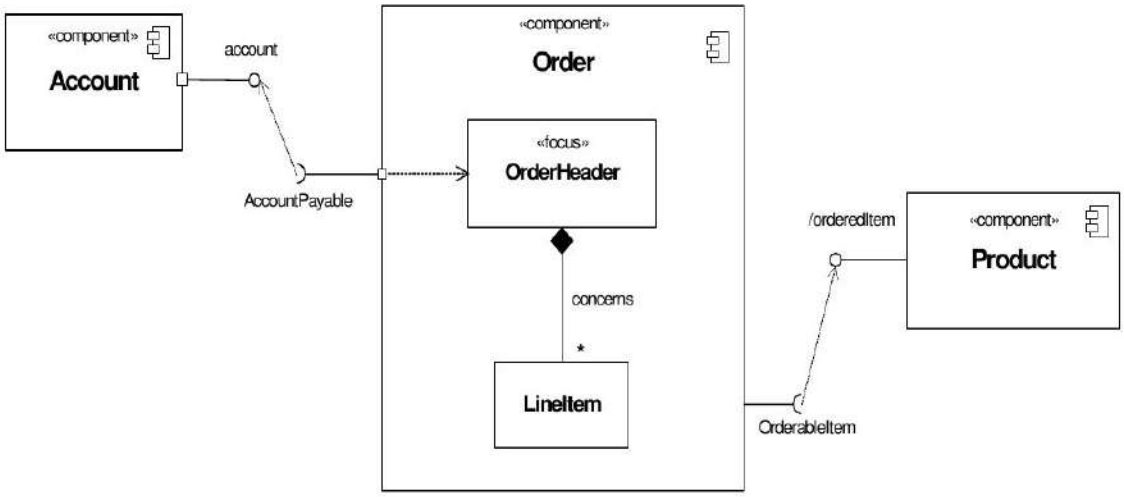


Figura 7.8: Outra visão alternativa para a montagem de componentes

7.2 Diagrama de Componentes

Diagramas de componentes, interconectando diferentes componentes em arranjos mais complexos, podem ser desenvolvidos conectando-se as interfaces disponibilizadas por um componente com as interfaces requeridas de outros componentes. Um exemplo dessa interconexão é apresentado na figura 7.9 a seguir:

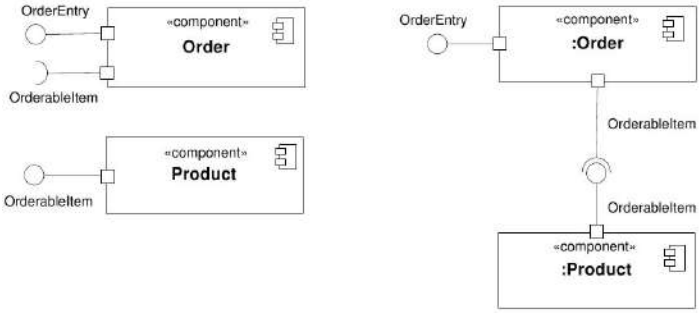


Figura 7.9: Composição de Sistemas pela Integração de Componentes

Em casos em que diversos componentes requerem a mesma interface, estes podem ser interconectados entre si, para efeito de simplificação. Um exemplo é apresentado na figura 7.10 a seguir:

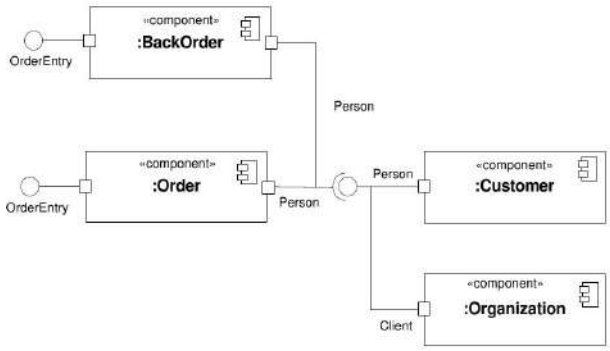


Figura 7.10: Interconexão de Interfaces Requeridas

Um exemplo de um diagrama de componentes, integrando diversos componentes em um artefato é mostrado na figura 7.11.

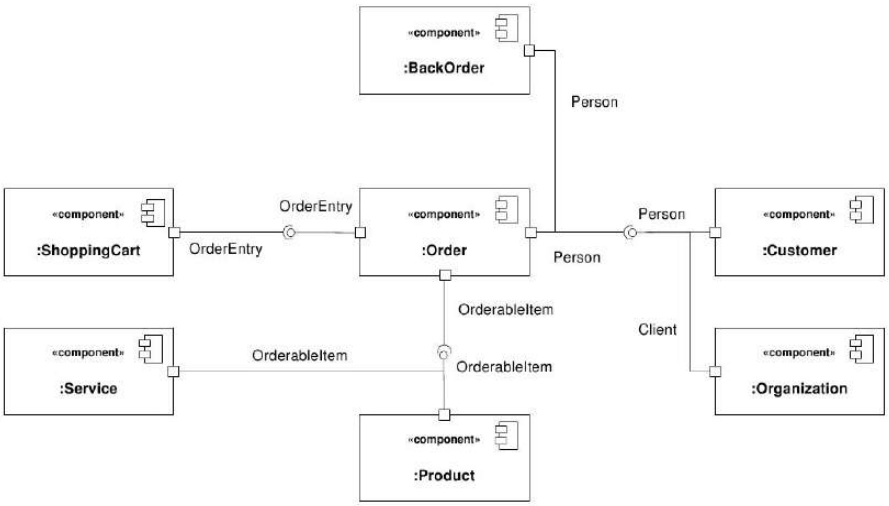


Figura 7.11: Exemplo de Diagrama de Componentes

8. Diagramas de Deployment

A linguagem UML prevê os assim chamados **diagramas de deployment** para representar uma estrutura física (normalmente de hardware), onde um conjunto de artefatos de software são instalados para compor uma configuração de um sistema.

Essa estrutura física é constituída por nós, conectados por vias de comunicação, criando uma rede de complexidade arbitrária. Nós são tipicamente definidos de maneira recursiva, podendo representar tanto dispositivos de hardware como ambientes de execução de software. Artefatos representam elementos concretos do mundo físico, resultados de um processo de desenvolvimento. Os diagramas de deployment normalmente são suficientes para representar a grande maioria das aplicações modernas de sistemas computadorizados. Em casos onde modelos de instalação mais elaborados sejam necessários, os diagramas de deployment podem ser estendidos por meio de profiles ou meta-modelos, de forma a representar os ambientes de hardware ou software desejados.

8.1 Artefatos

Um **artefato** é a especificação de um conjunto concreto de informações que é utilizado ou produzido por um processo de desenvolvimento de software, ou para a instalação ou operação de um sistema computacional. Exemplos de artefatos incluem arquivos de modelos, arquivos de código-fonte, scripts, arquivos executáveis, tabelas em bancos de dados, documento de texto, mensagem de e-mail ou qualquer outro resultado de um processo de desenvolvimento.

Um artefato representa, portanto, um elemento concreto do mundo físico. Uma instância particular do artefato (ou cópia do artefato) é instalada em uma instância de um nó. Artefatos podem manter associações com outros artefatos que podem estar aninhados dentro de si próprio. Diversos estereótipos estão previstos na norma, especificando o tipo de artefato. Por exemplo, «source» ou «executable». Estes estereótipos podem ser ainda especializados mais ainda em um profile. Por exemplo, o estereótipo «jar» poderia ser definido como uma subclasse de «executable» de forma a designar arquivos Java executáveis.

Na linguagem UML, um artefato é apresentado utilizando-se o retângulo de uma classe ordinária, com o uso da palavra-chave «artifact». Alternativamente, este retângulo pode acrescentar ainda um pequeno ícone no canto superior direito do retângulo.

Um exemplo de um artefato pode ser visualizado na figura 8.1 a seguir:

Figura 8.1: Exemplo de um Artefato

Na figura 8.2, apresenta-se um exemplo mostrando como pode-se detalhar a constituição de um artefato (por meio de componentes), utilizando-se de relações de dependência estereotipadas.

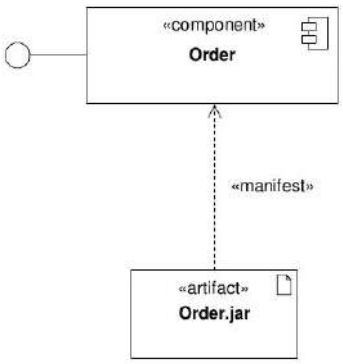


Figura 8.2: Exemplo Mostrando a Constituição de um Artefato

8.2 Nós

Um **nó** é um recurso computacional onde artefatos podem ser instalados para posterior execução. Nós podem ser interconectados por meio de conexões que definem estruturas de redes. Estas conexões representam caminhos de comunicação possíveis entre os nós. Topologias específicas de redes podem ser definidas por meio dessas conexões. Nós hierárquicos (ou seja, nós dentro de nós) podem ser definidos utilizando-se associações do tipo composição, ou definindo-se uma estrutura interna para aplicações de modelagem avançada.

Arcos tracejados com o uso do keyword «deploy» podem ser utilizados para representar a capacidade de um nó de suportar um determinado tipo de artefato. Alternativamente, isso pode ser representado utilizando-se artefatos aninhados dentro do nó. Em ambos os casos, isso significa que o artefato correspondente encontra-se instalado no nó.

A figura 8.3 a seguir, ilustra a notação de um nó.

Figura 8.3: Exemplo da Notação de um Nó

A figura 8.4 apresenta dois artefatos instalados em um nó.

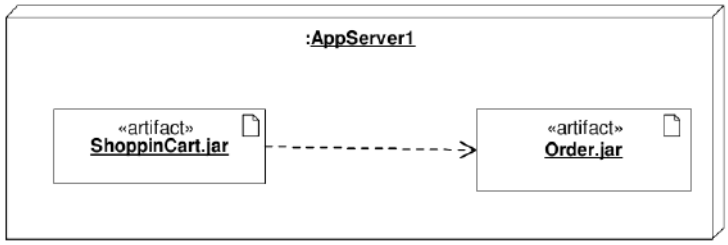


Figura 8.4: Exemplo de Artefatos instalados em um nó

A figura 8.5 ilustra uma notação alternativa para indicar a instalação destes mesmos artefatos no nó.

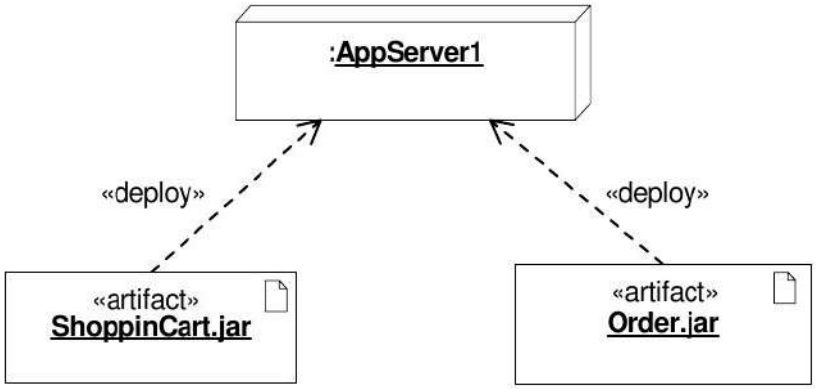


Figura 8.5: Notação alternativa para indicar a instalação de artefatos em um nó

A figura 8.6 a seguir ilustra ainda uma terceira maneira de se representar a instalação de artefatos em um nó.

Figura 8.6: Outra alternativa para representar a instalação de artefatos em um nó

A figura 8.7 mostra a instalação de artefatos aninhados em um nó.

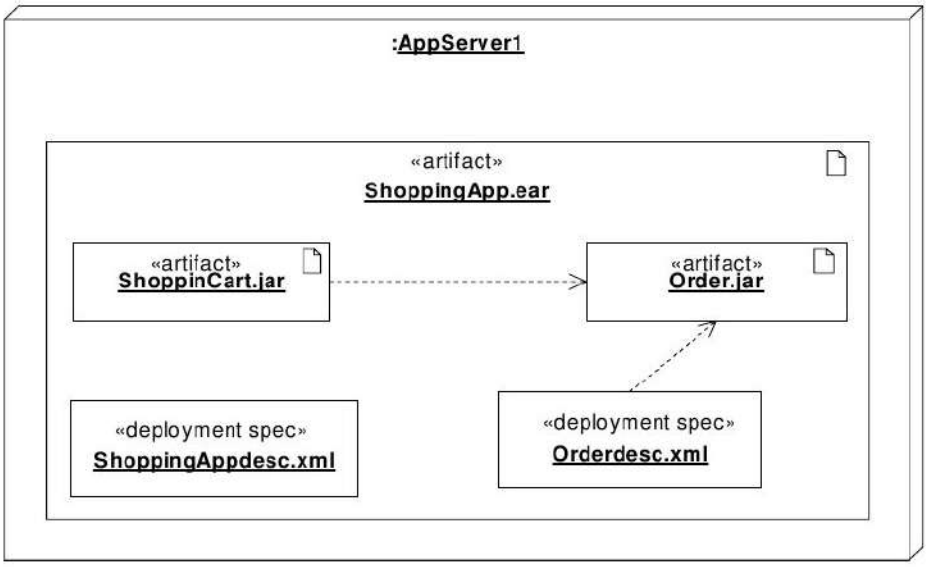


Figura 8.7: Exemplo de Artefatos aninhados instalados em um nó

A figura 8.8 mostra dois nós conectados entre si.

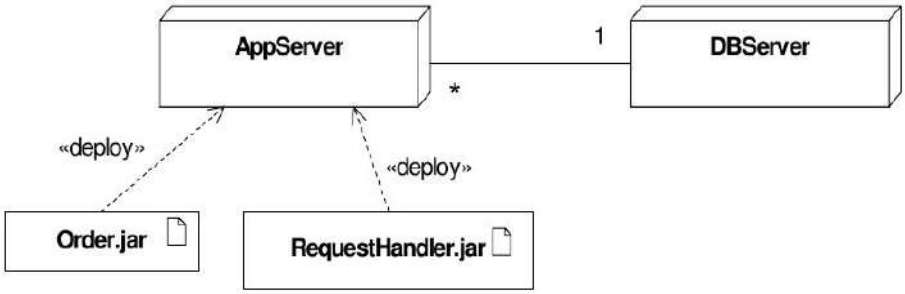


Figura 8.8: Exemplo de Conexão entre Nós

Nós podem ainda receber estereótipos, para representar diferentes tipos de plataformas de execução. Exemplos de estereótipos desse tipo incluem os estereótipos «device» para representar plataformas de hardware e «executionEnvironment» para representar plataformas de software com ambientes executáveis. Um exemplo de um nó estereotipado como «executionEnvironment» pode ser visto na figura 8.9.

Figura 8.9: Exemplo de Nó estereotipado

Na figura 8.10, vemos como nós estereotipados podem ser aninhados um sobre os outros para representar sistemas mais complexos.

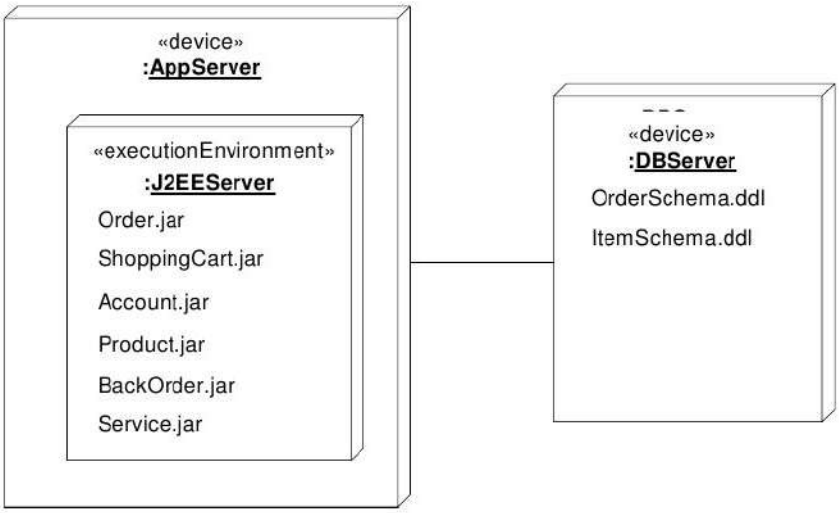


Figura 8.10: Exemplo de aninhamento de Nós estereotipados

9. Diagramas de Casos de Uso

9.1 Casos de Uso

Casos de uso são abstrações de pequenas histórias narrativas envolvendo a interação entre um ou mais usuários (chamados de atores) e o sistema. A idéia é que estes casos de uso representem, por meio dessas pequenas histórias, as funcionalidades de um sistema. Imagina-se um conjunto de atores necessários a operar o sistema e passa-se a descrever o fluxo dos acontecimentos, onde o ator executa uma ação, e o sistema responde de alguma maneira a essa ação, até que alguma funcionalidade tenha sido contemplada. Especificar um caso de uso é como observar em nossa mente um conjunto de atores imaginários operando o sistema que pretendemos desenvolver, e descrever o que esses atores fazem e como o sistema responde. Normalmente utiliza-se somente um ator interagindo na maioria dos casos de uso, mas há casos em que mais de um ator podem ser necessários. Imagine um sistema de caixa de padaria, em que o caixa (o primeiro ator) executa a maioria das ações de acesso ao sistema, mas o cliente (o segundo ator) precisa em algum instante digitar sua senha de cartão de crédito, caso faça o pagamento com cartão. Para esse caso de uso, teríamos dois atores necessários para que a funcionalidade "Pagamento com Cartão de Crédito" pudesse ser implementada. Apesar da grande maioria de casos de uso demandar somente um único ator, pode ser que diferentes atores, representando diferentes papéis de usuários do sistema tenham diferentes níveis de permissão de acesso às funcionalidades. Dessa forma, um caso de uso como "Fazer o Login no Sistema" talvez possa ser realizado por um ator chamado "Usuário Não-Autenticado", mas os demais casos de uso demandem um ator chamado "Usuário Autenticado". Outros casos de uso, ainda, podem demandar um ator chamado "Administrador".

Podemos entender um caso de uso, portanto, como uma sequência de ações (uma atividade, segundo a terminologia UML), executadas na forma de uma interação entre os atores e o sistema, onde ao final da atividade existe alguma vantagem ou ganho para o(s) usuário(s). Observe que não basta se descrever uma atividade para que esta seja um caso de uso. É necessário que haja um benefício direto ao usuário em função dessa atividade. Existem atividades que são casos de uso, e existem atividades que são somente sub-atividades de um caso de uso. O que determina que uma atividade seja um caso de uso é o benefício decorrente do resultado dessa atividade. Dessa forma, dizemos que um caso de uso é equivalente a uma funcionalidade do sistema. Essa equivalência entre "casos de uso" e "funcionalidades" é um dos princípios que orienta o "Processo Unificado" de desenvolvimento de software. Dessa forma, o Processo Unificado captura (ou elicita) os requisitos, por meio da descoberta e especificação de diferentes casos de uso. Para representar uma coleção de casos de uso, suas possíveis relações e quais os atores envolvidos com cada caso de uso, desenvolveu-se o diagrama de casos de uso UML.

9.2 Diagrama de Casos de Uso

Os diagramas de casos de uso mostram atores e casos de uso juntos com seus relacionamentos. Um exemplo de um diagrama de casos de uso pode ser visto na figura 9.1 a seguir:

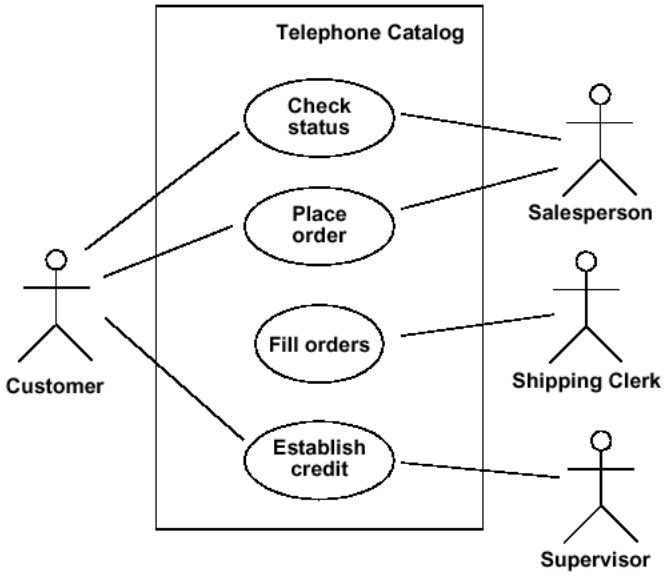


Figura 9.1: Exemplo de Diagrama de Casos de Uso

O motivo da existência dos diagramas de caso de uso são basicamente os seguintes. Em primeiro lugar, representar quais os atores envolvidos com cada caso de uso. Em segundo lugar, o diagrama de casos de uso permite que se represente possíveis relações entre os casos de uso. Mas talvez o mais importante motivo para termos os diagramas de casos de uso é o fato de que estes diagramas são os únicos diagramas, em toda a documentação desenvolvida no Processo Unificado que representam o sistema em sua totalidade. Os diagramas de caso de uso permitem uma "visão geral" do sistema, onde todas as suas funcionalidades estão concentradas. Esta visão geral permite aos desenvolvedores apreciar, durante a fase de especificação do sistema, se o sistema está suficientemente especificado, ou se faltam ainda funcionalidades necessárias. Dessa forma, torna-se um diagrama bastante importante e útil durante a fase de especificação do sistema.

9.3 Pontos de Extensão

Casos de uso podem ter pontos de extensão, ou seja, referências a uma localização dentro de fluxo de atividades onde sequências de ações de outros casos de uso podem ser inseridas. Cada ponto de extensão tem um único nome dentro de um caso de uso. Um exemplo de casos de uso com pontos de extensão é mostrado na figura 9.2 a seguir:

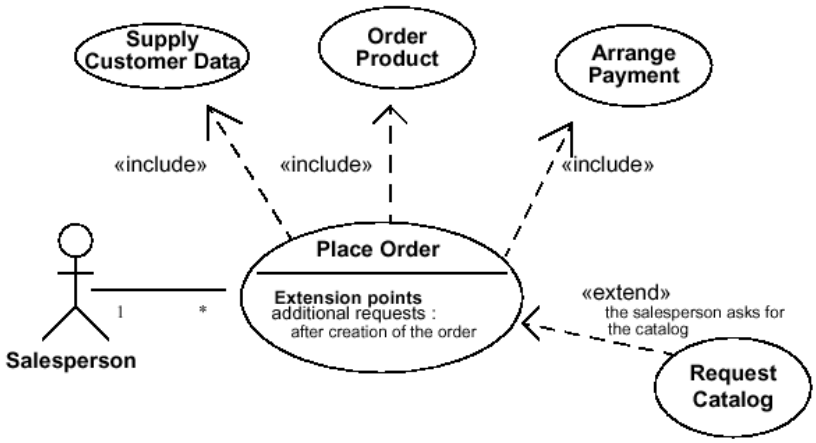


Figura 9.2: Caso de Uso com Ponto de Extensão

9.4 Relações entre Casos de Uso

Durante a fase de especificação dos requisitos, usualmente se levanta o maior número possível de casos de uso e os atores que dele participam. Estes casos de uso são considerados de maneira independente uns dos outros. O resultado é alguma coisa como o diagrama da figura 9.3 a seguir.

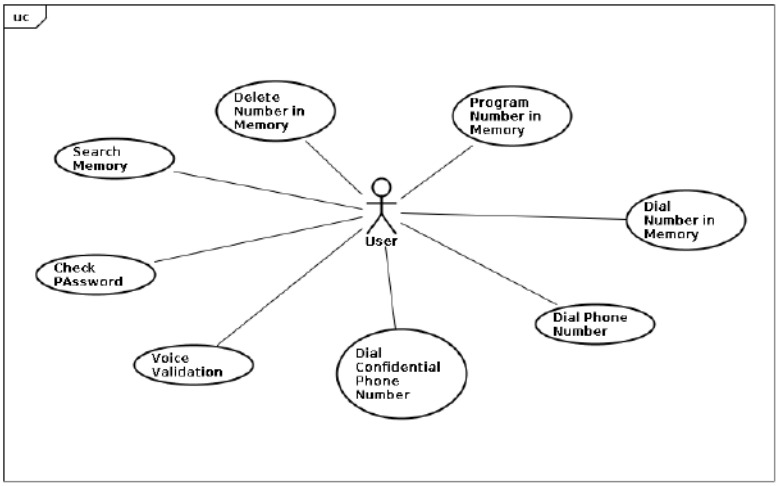


Figura 9.3: Diagrama de Casos de Uso Inicial

À medida que o conhecimento sobre o sistema vai crescendo, entretanto, e o número de casos de uso passa a crescer também, torna-se necessário estruturar o diagrama de casos de uso, e passar a considerar as relações que os casos de uso podem ter uns cons os outros, bem como a relação que os atores podem ter uns com os outros. Durante a etapa do detalhamento dos casos de uso, pode ser que algum caso de uso em particular tenha se tornado demasiadamente complexo ou detalhado, quando então se torna útil desmembrá-lo em mais de um casos de uso. Da mesma forma, pode ser que se tenha percebido que muitos casos de uso possuem uma estrutura comum, que se repete diversas vezes. Dessa forma, torna-se necessário fazer um *refactoring* dos casos de uso, de tal forma que os mesmos sejam repensados e devidamente estruturados, de modo a gerar um modelo que seja homogêneo, consistente e simples de ser interpretado.

Basicamente, 4 tipos de relacionamentos podem ser indicados em diagramas de casos de uso:

- Associações
- *Extend*
- *Include*
- Generalização

As **associações** denotam a participação de atores em um caso de uso. É o único tipo de relacionamento entre um ator e um caso de uso. Um relacionamento do tipo **extend** é uma relação entre um caso de uso A para um caso de uso B que indica que uma instância do caso de uso B pode ser aumentada (sujeita a condições específicas de extensão) pelo comportamento especificado por A. Esse comportamento é inserido conforme especificado por um ponto de extensão em B. Um relacionamento do tipo **include** é uma relação entre um caso de uso A para um caso de uso B que indica que uma instância do caso de uso A contém o comportamento especificado por B. Esse comportamento é incluído na localização indicada em A por um ponto de extensão. Um relacionamento do tipo **generalização** entre um caso de uso A e um caso de uso B indica que A é uma especialização de B. Normalmente, essa generalização implica em que B é uma descrição mais abstrata de uma situação, e A é uma descrição mais detalhada. Além dos casos de uso, é possível utilizarmos generalizações para indicarmos o relacionamento entre dois atores. Um exemplo deste tipo é apresentado na figura 9.4 a seguir:

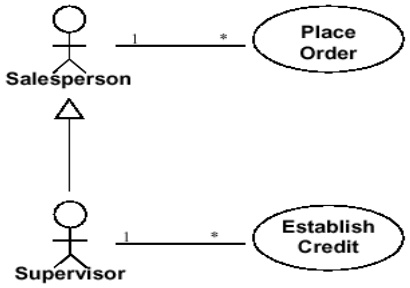


Figura 9.4: Generalização entre Atores

Observe que nesse caso, o ator Supervisor é um um tipo de Salesperson, ou seja, uma especialização de um Salesperson. Observe ainda pelo exemplo que as associações entre atores e casos de uso podem também conter uma cardinalidade.

As associações entre os atores e os casos de uso podem ainda possuir uma navegabilidade (seta entrando ou saindo do caso de uso). Essa navigabilidade indica quem inicia o caso de uso. Caso a seta seja do ator para o caso de uso, é o ator quem inicia a interação. Caso seja do caso de uso para o ator, é o sistema quem inicia a interação. Veja o exemplo na figura 9.5 a seguir:

Figura 9.5: Exemplo de Navegabilidade em Casos de Uso

Neste exemplo, que mostra o caso de uso Participação em Conferência Eletrônica, é o professor quem inicia as atividades. O aluno passa a interagir posteriormente, a partir da iniciativa do sistema.

Durante o detalhamento dos casos de uso (que, assumiremos neste texto, seja feito por meio de um diagrama de atividades), podem haver atividades ou partes de atividades que são semelhantes em diversos casos de uso. Pode-se detectar uma situação como essa quando observamos que certas partes dos diagramas de atividades se repetem em mais de um casos de uso. De forma a reduzir a redundância, os casos de uso podem ser então reestruturados para tornar o modelo como um todo mais enxuto. Assim, esses trechos de casos de uso podem se tornar casos de uso independentes, que são reutilizados no modelo por meio da relação de generalização.

Além disso, outra possível análise é tentar identificar descrições adicionais ou opcionais de funcionalidade. Sabemos que o mecanismo de extensão permite a inserção de adições ao comportamento básico de casos de uso. Como vimos, esse relacionamento inclui as condições para a extensão e o ponto de extensão, onde o caso de uso deve ser inserido. Durante a etapa de estruturação de casos de uso, o que se faz é tentar identificar situações desse tipo, alterando o diagrama de casos de uso inicial de modo a incluir relacionamentos de extensão. Normalmente, os diagramas de atividades que detalham esses casos de uso demandam ser retrabalhados também, de forma a refletir as mudanças implementadas.

Por fim, uma última análise que se faz durante a estruturação dos casos de uso é tentar identificar descrições repetidas de funcionalidade. Sabemos que o relacionamento de inclusão permite a inserção incondicional e explícita do comportamento de um caso de uso em outros casos de uso. Nesses casos, tenta-se então verificar se situações desse tipo ocorrem, e nesse caso procede-se à inserção de relacionamentos do tipo include no diagrama de casos de uso.

Uma dúvida muito frequente que ocorre é a dificuldade em diferenciar os relacionamentos de generalização e de inclusão. Temos que ter muito cuidado pois esses dois tipos de relacionamentos são muito semelhantes. A grande diferença que existe entre esses dois é que no relacionamento de generalização, o caso de uso que é generalizado é um caso de uso mais abstrato que, apesar de semelhante, será diferente em cada uma de suas especializações. Um bom exemplo é o exemplo mostrado na figura 9.6 abaixo, fruto da estruturação do modelo de casos de uso que apresentamos na figura 9.3.

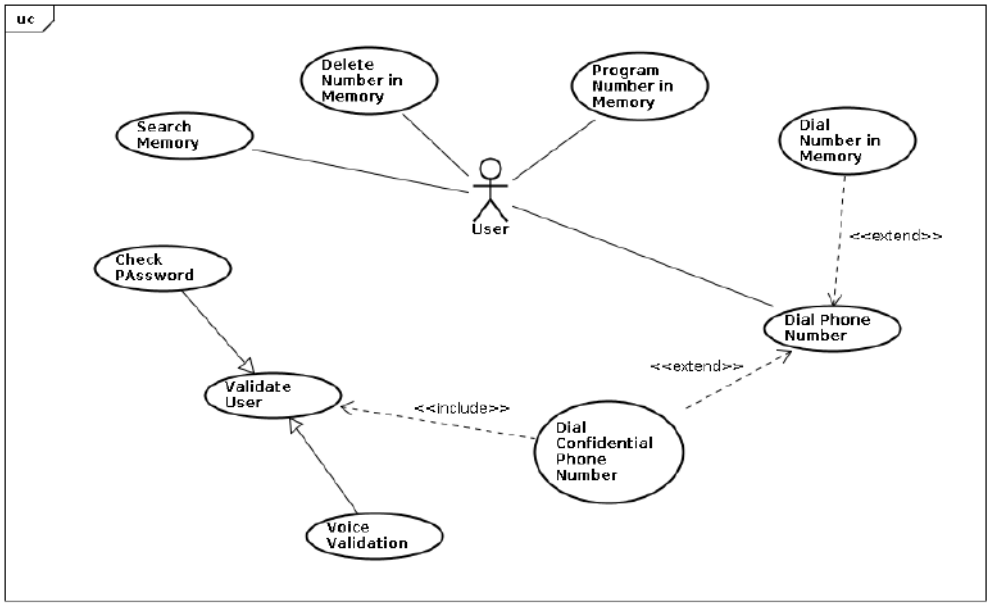


Figura 9.6: Exemplo de Diagrama de Casos de Uso Estruturado

Observe que anteriormente tínhamos dois casos de uso, o caso de uso *CheckPassword* e o *VoiceValidation*, que apresentavam um compartilhamento de funcionalidades. Ambos serviam para efetuar uma validação do usuário. Uma maneira de refinar nosso modelo foi criar um novo caso de uso abstrato chamado *ValidateUser*, que passou então a ser uma abstração, tanto de *CheckPassword* como de *VoiceValidation*. Para compreendermos a diferença entre o uso de uma generalização e de uma inclusão, observe que a descrição de uma checagem de password não tem nada a ver, em princípio com uma validação vocal. Em termos concretos, eles são casos de uso completamente diferentes. Entretanto, quando abstraímos o que ocorre em cada um desses casos de uso, vemos que ambos nada mais fazem do que validar o usuário. Por isso, usamos uma relação de generalização. O relacionamento do tipo inclusão, ao contrário, inclui totalmente o caso de uso como um sub-trecho de um caso de uso mais complexo. Assim, devemos ter cuidado durante esse refinamento. Em alguns casos, será necessário fazermos abstrações e incluirmos novos casos de uso mais abstratos. Em outros, vamos tentar verificar a mera repetição de situações e utilizaremos inclusões. Observe que no diagrama do exemplo, verificamos que a validação do usuário ocorre somente quando ocorre uma discagem de número confidencial. Por isso, colocamos o caso de uso *ValidateUser* como uma inclusão de *DialConfidentialPhoneNumber*. Por outro lado, tanto *DialConfidentialPhoneNumber* quanto *DialNumberInMemory* extendem a descrição de *DialPhoneNumber*. Por isso, colocamos ambos como extensões de *DialPhoneNumber*.

Repare que no processo de desenvolvimento, após chegarmos ao diagrama de casos de uso estruturado, pode ser necessário que modifiquemos o detalhamento dos casos de uso feito anteriormente, como por exemplo a inclusão de novos casos de uso (vide a inclusão de *ValidateUser* no exemplo).

10. Diagramas de Interação

Diagramas de interação mostram padrões de interação entre instâncias de objetos. Na verdade, de acordo com a norma UML, os diagramas de interação podem aparecer em duas formas diferentes, os diagramas de sequência e os diagramas de comunicação (também chamados de diagramas de colaboração, na versão 1 da norma UML). As informações em ambos os diagramas é equivalente, mas cada tipo de diagrama enfatiza um aspecto particular da interação. Os diagramas de sequência mostram a sequência explícita de estímulos entre objetos e são melhores para especificações de tempo real e para cenários complexos. Os diagramas de comunicação mostram o relacionamento entre as instâncias e são melhores para o entendimento de todos os efeitos sobre uma determinada instância, bem como para um design procedural. Uma regra prática para sabermos quando devemos utilizar diagramas de sequência ou diagramas de comunicação é a seguinte: se o número de objetos interagindo é grande, e o número de mensagens sendo trocado entre cada objeto é pequeno, devemos dar preferência aos diagramas de comunicação. Se, ao contrário, o número de objetos é pequeno, mas o número de mensagens sendo trocadas entre eles é grande, então o mais adequado é a utilização de um diagrama de sequência

10.1 Diagramas de Sequência

Um diagrama de sequência mostra uma interação na forma de uma sequência temporal de mensagens sendo enviadas entre instâncias. Em particular, mostra as instâncias participando de uma interação por meio de *lifelines* e o estímulo que trocam entre si arranjadados na forma de uma sequência temporal. Assim, um diagrama de sequência não explicita as associações entre objetos, embora possamos inferir que havendo uma mensagem de um objeto a outro no diagrama, é necessário que haja uma associação entre eles (pois caso contrário, o objeto não seria capaz de enviar a mensagem).

A idéia de ***lifeline*** é a idéia de representar temporalmente o ciclo de vida de um objeto executando um papel bem específico em uma interação. Assim, cada objeto participando da interação possui sua própria *lifeline*. Representa-se uma *lifeline* por meio de uma linha vertical saindo de baixo do objeto e varrendo todo o espaço do diagrama. Dessa forma, entende-se que o eixo vertical representa o passar do tempo. As setas entre diferentes *lifelines* denotam uma comunicação ocorrendo entre os objetos envolvidos em um instante do tempo. A existência e a duração do objeto em um papel pode ser mostrada, mas o relacionamento entre os objetos não é mostrado explicitamente como no caso dos diagramas de comunicação. Uma *lifeline* pode se dividir em duas ou mais *lifelines* concorrentes para expressar uma condicionalidade, ou seja, fluxos de sequência alternativos, dependentes de condições que são indicadas no diagrama. Um exemplo de diagrama de sequência pode ser visto na figura 10.1 a seguir:

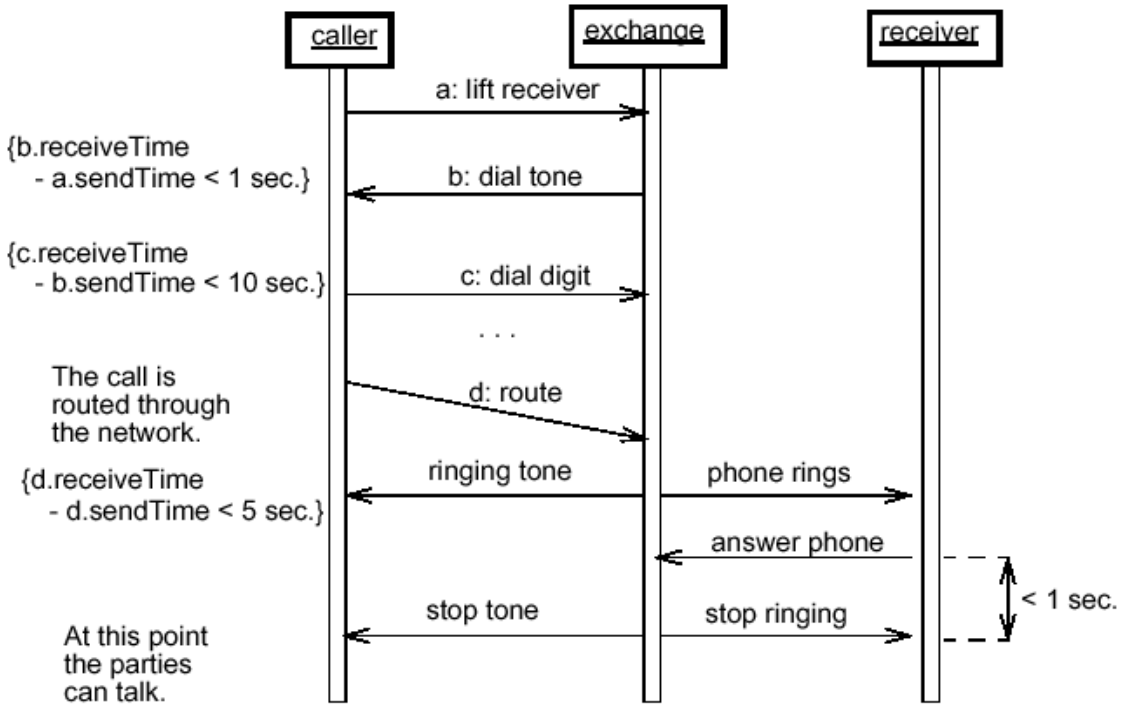


Figura 10.1: Exemplo de Diagrama de Sequência

Observe nesse exemplo que todos os objetos estão ativos (ou com o chamado foco de controle) durante todos os instantes de tempo. Esse fato é representado pelas linhas cheias percorrendo todo o *lifeline*. Esse tipo de representação pode também ser utilizado quando não se deseja representar explicitamente o momento em que um objeto está ativo, quando é criado ou destruído. Entretanto, todas essas informações podem ser representadas. Um exemplo mostrando a criação de objetos, o momento em que estão ativos e o momento em que são destruídos pode ser visto na figura 10.2.

Nesta figura, diversos recursos de representação dos diagramas de sequência são representados. Por exemplo, o **foco de controle** ou ativação, mostra o período no qual um objeto está executando uma ação. Durante o foco de controle, a lifeline apresenta uma linha cheia e dupla. Quando o objeto não está com o foco de controle, a lifeline se transforma em uma linha tracejada.

Observe também um exemplo de **mensagens condicionais** entre **ob1** e **ob3**. Dependendo do valor de x, o objeto **ob1** emite a mensagem **foo(x)** para **ob2** ou a mensagem **bar(x)** para **ob3**.

Um exemplo de **recursão** é mostrado também na figura. Uma recursão acontece quando um objeto manda uma mensagem para si próprio, como faz o objeto **ob1** com a mensagem **more()**.

No diagrama vemos também a representação da **criação de objetos**. Os objetos que existiam previamente encontram-se alinhados ao topo do diagrama. No caso os objetos **ob3** e **ob4** já existiam ao início do processo. Os objetos **ob1** e **ob2**, entretanto, foram criados durante a interação. Essa criação é representada com o deslocamento do objeto para o instante de tempo em que este foi criado. Assim, o objeto **ob1** foi criado pelo usuário por meio da mensagem **op()** e o objeto **ob2** foi criado por meio da mensagem **foo(x)**.

Do mesmo modo, representamos a **destruição de um objeto** por meio da marca X em algum ponto

da lifeline do objeto. Assim, os objetos **ob1** e **ob2** são destruídos nos pontos marcados por X no diagrama.

Devemos ainda observar que diferentes tipos de setas ocorrem no diagrama. Na verdade, o UML estabelece diferentes signficados para diferentes tipos de setas. Cada tipo de seta representará um diferente tipo de comunicação.

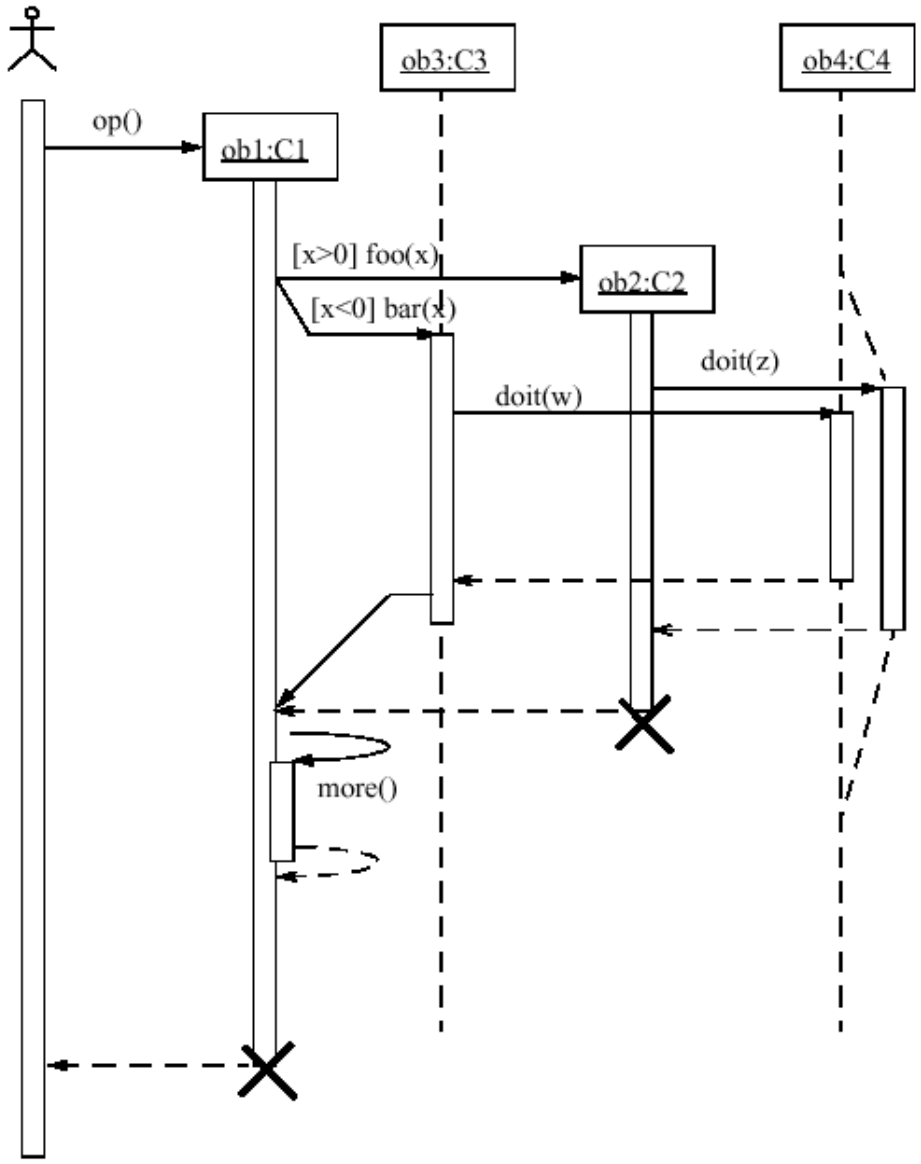


Figura 10.2: Exemplo de Diagrama de Sequência com Representação da Ativação dos Objetos

As setas do tipo → são utilizadas para representar **chamadas de procedimento ou outro tipo de fluxo de controle**. Toda uma sequência aninhada é completada antes que a sequência mais externa termine. Este tipo de seta pode ser usada em chamadas de procedimento ordinárias, mas pode também ser usada em objetos concorrentes quando um deles manda um sinal e espera uma sequência de comportamentos ser completada.

As setas do tipo  são utilizadas para representar um **fluxo de controle fraco**. Assim, cada seta desse tipo mostra a progressão do próximo passo na sequência. Normalmente as mensagens desse tipo são assíncronas.

As setas do tipo são utilizadas para representar **estímulos assíncronos**. Este tipo de seta é utilizado no lugar do anterior quando se quer mostrar explicitamente uma comunicação assíncrona entre dois objetos.

Por fim, as setas do tipo  (tracejadas) são utilizadas para representar um **retorno de uma chamada de procedimento**. Assim, são usadas em conjunto com as setas do primeiro tipo, ao final de cada procedimento. Setas desse tipo são utilizadas no diagrama da figura 2.

10.2 Diagramas de Comunicação

Um diagrama de comunicação mostra a interação entre objetos organizada de acordo com os papéis de cada objeto na interação, e sua ligação entre si. Ao contrário de um diagrama de sequência, mostra explicitamente o relacionamento entre objetos executando os diversos papéis. Da mesma maneira, não mostra o tempo como uma dimensão separada, ou seja, de forma explícita como nos diagramas de comunicação. Entretanto, a sequência de mensagens é mostrada na forma de números atribuídos a cada mensagem, que indicam a ordem em que as mensagens são enviadas. Assim, a informação temporal, apesar de não estar colocada de forma explícita, também está presente nos diagramas de comunicação.

Os diagramas de comunicação podem ser desenvolvidos em duas formas diferentes: ao **nível de especificação** (papéis de classifier, papéis de associação e mensagens) ou ao **nível de instância** (objetos, links e estímulos). No primeiro caso, os diagramas modelam papéis e a estrutura de colaboração entre esses papéis, sendo que no segundo caso, mostra-se os objetos que assumem esses papéis.

Um exemplo de diagrama de comunicação é apresentado na figura 10.3 a seguir:

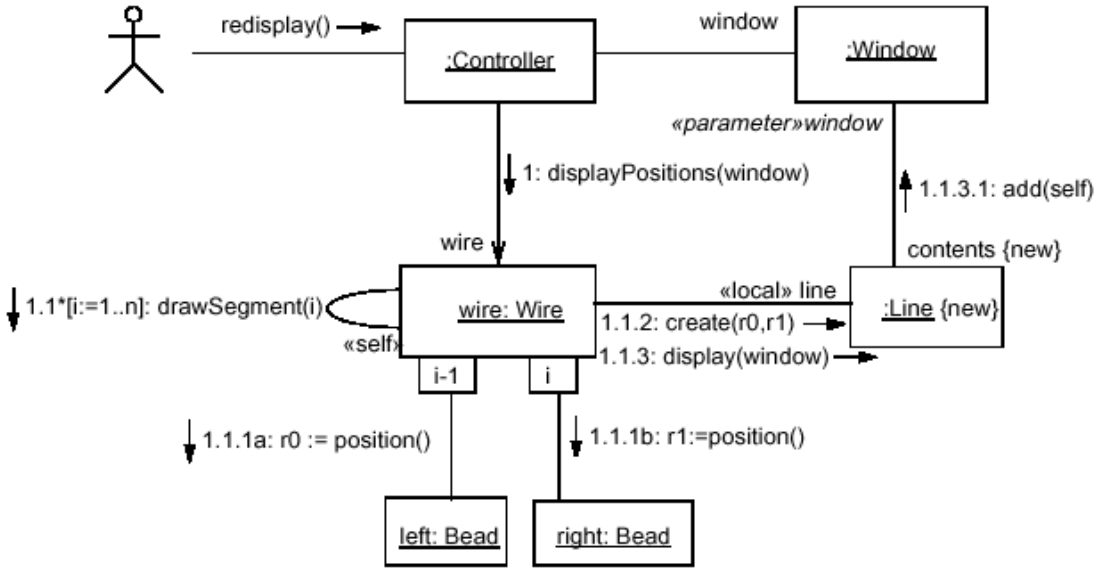


Figura 10.3: Exemplo de Diagrama de Comunicação

Observe alguns detalhes desse exemplo. Em primeiro, observe que todos os nomes nas caixas estão grifados. Isso indica que se trata de instâncias das classes e não das classes por si só. Alguns objetos possuem nomes, ao passo que outros apenas indicam a classe da instância. Observe que os relacionamentos entre as classes estão indicados, da mesma forma que apareceriam em um diagrama de classes. Entretanto, sobre cada relacionamento, existe uma pequena seta representando uma mensagem que é enviada de um objeto a outro. Observe especialmente o exemplo de recursão, onde o objeto **wire** manda um conjunto de mensagens para si próprio. Observe a sintaxe dessa mensagem. O caracter * indica que se trata de um conjunto de mensagens, e não de uma única mensagem. Em seguida, o string [i :=1..n] indica que se trata de uma recursão de n mensagens indexadas por i. Observe ainda o tráfego de objetos. A mensagem 1.1.1a obtém como retorno um objeto **r0**, que é enviado posteriormente na mensagem 1.1.2 como um parâmetro para o objeto **:Line**. Um outro recurso muito utilizado também em diagramas de comunicação é o dos multi-objetos, ilustrado na figura 10.4 a seguir:

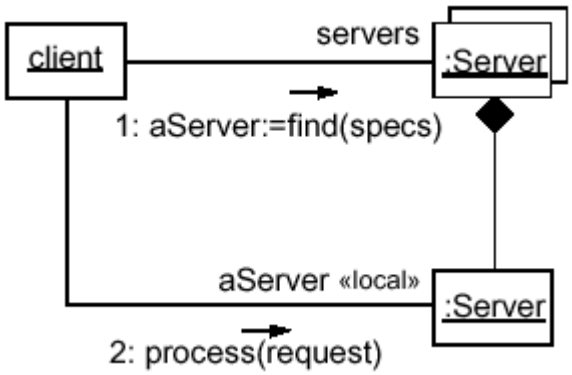


Figura 10.4: Exemplo de Multi-objetos

Um **multi-objeto** representa um conjunto de objetos posicionados em uma extremidade de uma associação que contenha multiplicidade maior que 1. Assim, os multi-objetos são utilizados para representar mensagens que são enviadas à coleção inteira de objetos ao invés de um único objeto

dentro da coleção. Multi-objetos podem ter diferentes implementações em linguagens de programação. A mais conhecida delas corresponde aos chamados arrays ou vetores. Entretanto, linguagens mais avançadas como a linguagem Java possuem diversas outras implementações de multi-objetos, tais como os objetos do package Collections, que permitem a representação de listas, conjuntos, e outros tipos de coleções mais sofisticadas.

11. Diagrama de Atividades

O diagrama de atividades é um diagrama UML utilizado para modelar o aspecto comportamental de processos. É um dos diagramas que mais sofreu mudanças em seu meta-modelo, desde seu surgimento no UML 1.0. Neste diagrama, uma **atividade** é modelada como uma sequência estruturada de **ações**, controladas potencialmente por nós de decisão e sincronismo. Em seu aspecto mais simples, um diagrama de atividades pode ser confundido com um fluxograma. Entretanto, ao contrário de fluxogramas, os diagramas de atividades UML suportam diversos outros recursos, tais como as **partições** e os nós do tipo *fork* e *merge*, além da definição de **regiões de interrupção**, que permitem uma modelagem bem mais rica do que simplesmente um fluxograma.

Um exemplo de um diagramas de atividade simples é mostrado na figura 11.1 a seguir.

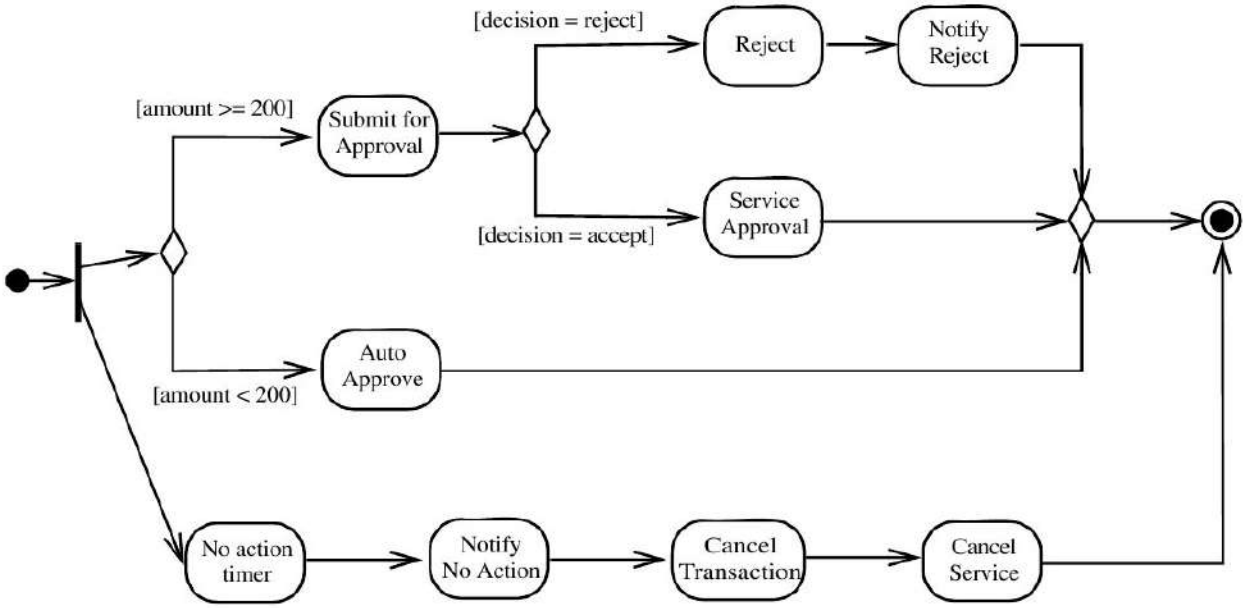


Figura 11.1: Exemplo de um Diagrama de Atividades Simples

Neste diagrama, o ponto preto à esquerda indica o início da atividade, as caixas com cantos arredondados representam ações, os pequenos losangos são nós de decisão e a barra vertical preta é um nó de sincronização do tipo *fork*. Os arcos conectando as ações representam a sequência em que as ações devem ser executadas, sendo que nos arcos que saem dos nós de decisão existem **condições de guarda**, que decidem qual será a próximo ação a ser executada. Essas condições de guarda devem ser proposições mutuamente exclusivas, de tal forma que para n arcos saindo de um nó de decisão, somente um deles pode ser verdadeiro a cada instante de tempo. Por fim, no canto direito do diagrama encontra-se um nó de finalização, que denota o final da atividade.

A figura 11.22 a seguir mostra um diagrama de atividades com partições. Diagramas com partições permitem que processos onde múltiplos agentes atuam, potencialmente em paralelo, possam ser representados.

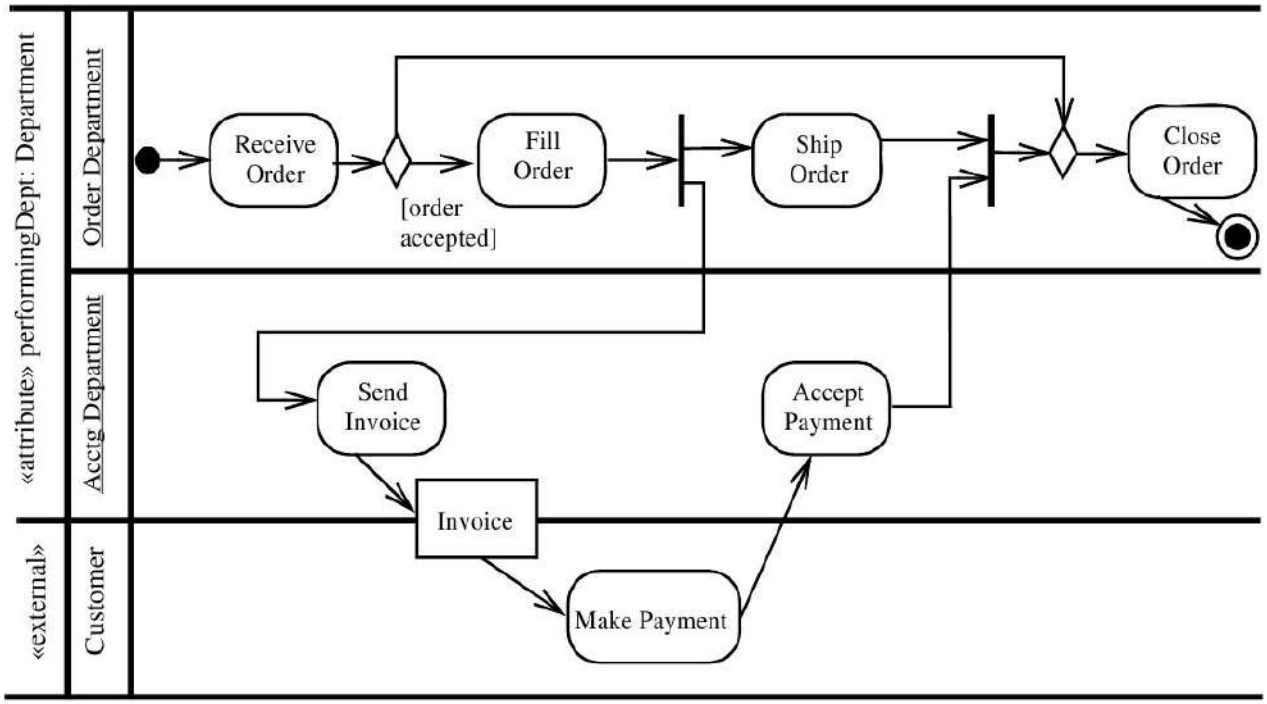


Figura 11.2: Diagrama de Atividades com Partições

11.1 Ação

Uma ação representa um passo elementar de uma atividade, ou seja, um passo que não pode ser decomposto dentro de uma atividade. Uma atividade representa um comportamento que pode ser composto por ações ou outras sub-atividades. Uma ação pode ter um conjunto de arcos de entrada e de saída, que especificam o fluxo de controle e de dados para outros nós. Uma ação não inicia sua execução até que todas as suas condições de entrada sejam satisfeitas. Somente quando uma ação é terminada que a ação subsequente fica habilitada.

Uma ação é representada conforme a figura 11.3 a seguir:

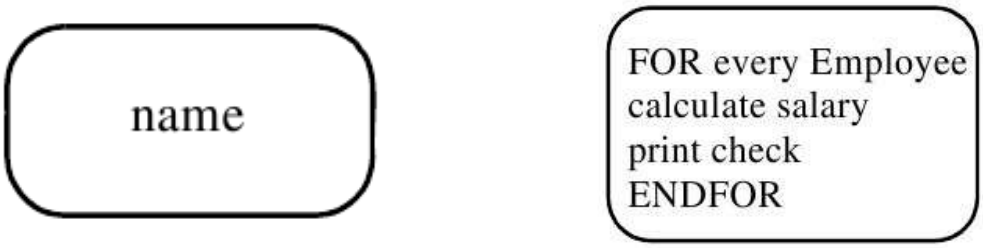


Figura 11.3: Exemplos de Ações

Alternativamente, ações podem ser definidas com pré-condições, que definem as condições necessárias para que a ação possa ser executada, e pós-condições, que definem o estado depois que a ação é executada. Exemplos de situações com essa podem ser vistos na figura 11.4 a seguir.

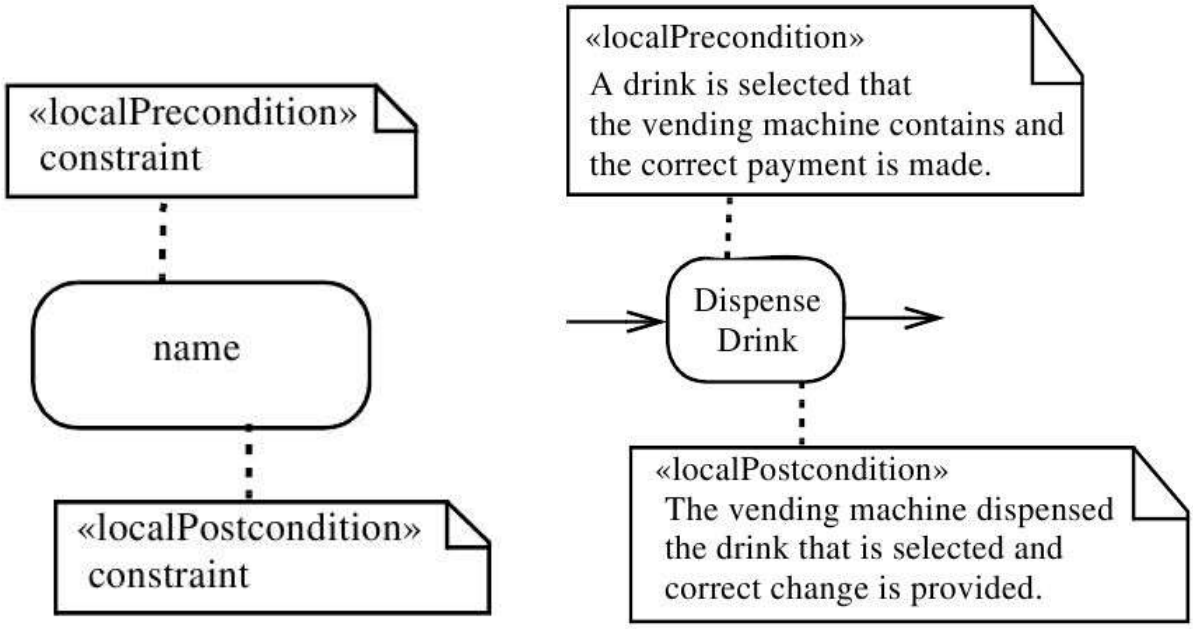


Figura 11.4: Ações definidas com Pré-condições e Pós-condições

11.2 Atividades

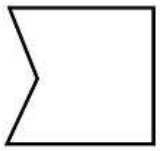
Atividades podem ser representadas por sequências de ações e também de sub-atividades. Dessa forma, para representar uma sub-atividade dentro de uma atividade (ou seja, todo um conjunto de ações ou sub-atividades), utiliza-se uma representação semelhante a de uma ação, com um pequeno ícone no canto direito inferior. A notação para uma atividade pode ser vista na figura 11.5 a seguir.

Figura 11.5: Exemplo de Atividade

Do ponto de vista formal, uma atividade conforme representada na figura 11.5 não é exatamente uma atividade, mas uma ação especial, chamada de CallBehaviorAction, que de maneira atômica invoca a execução de toda uma atividade. Entretanto, para efeitos práticos, podemos entendê-la como uma atividade de-per-si.

11.3 Eventos

Outros elementos que podem aparecer em um diagrama de atividades correspondem a eventos. Eventos são mudanças de estado instantâneas que propiciam o início de uma outra ação. Existem basicamente três representações para eventos. Para representar um evento único que, caso aconteça, propicia o início de uma ação subsequente, utiliza-se a ação especial *AcceptEventAction*, representada pela notação indicada na figura 11.6.



Accept event action

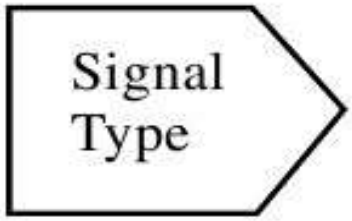
Figura 11.6: Evento que Propicia o início de uma Ação subsequente

Para representar um evento periódico, que acontece de tempos em tempos, e a cada vez que aconteça favoreça o início de uma ação subsequente, utiliza-se a notação apresentada na figura 11.7 a seguir.

Accept time event action

Figura 11.7: Evento Temporizado

Para representar a geração de um evento deliberado, ao final de uma ação, utiliza-se a notação indicada na figura 11.8 a seguir.



Send signal action

Figura 11.8: Geração de um Evento

As figuras 11.9, 11.10 e 11.11 a seguir ilustram o uso de eventos junto com ações.

Figura 11.9: Evento e Ação correspondente

Figura 11.10: Geração e Recepção de Evento

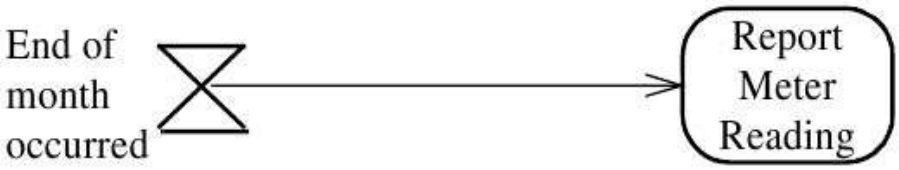


Figura 11.11: Exemplo de Evento Temporizado e Ação

11.4 Objetos

Além do fluxo de controle, que especifica uma sequência de ações que definem um processo, um diagrama de atividades também pode representar o fluxo de dados acontecendo em um processo. Esse fluxo de dados pode ser representado definindo-se explicitamente os objetos necessários para que uma ação possa ser realizada, bem como os objetos gerados após a finalização de uma ação. Um objeto é representado da mesma maneira que em um diagrama de classes, entretanto sem a necessidade de estar sublinhado. Alguns exemplos de uso de objetos podem ser vistos na figura 11.12 a seguir.

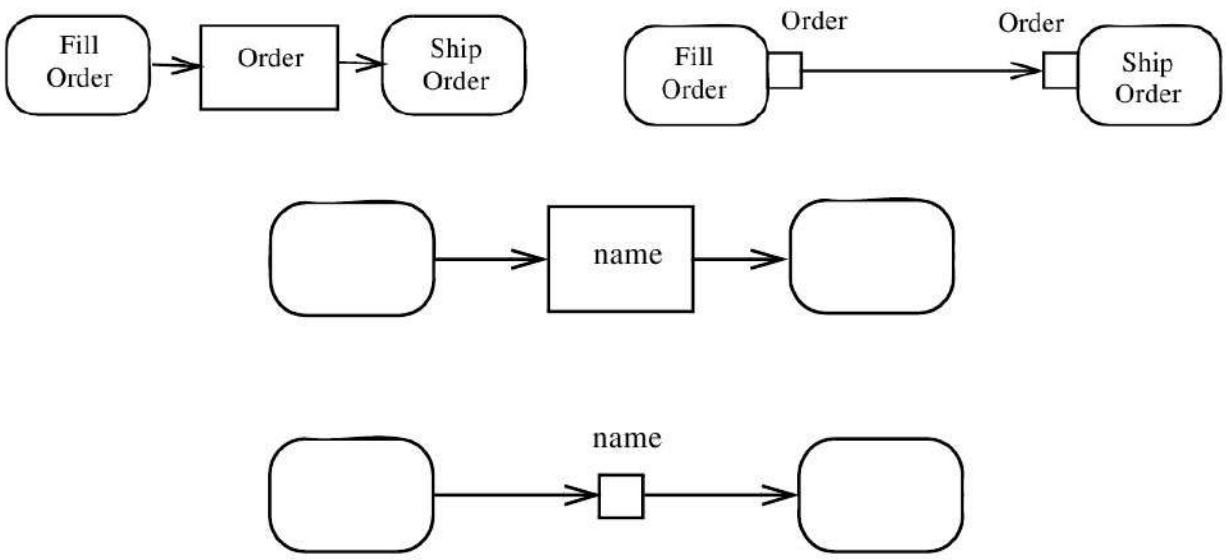


Figura 11.12: Exemplos do Uso de Objetos entre Ações

Objetos podem também ser passados como parâmetros para atividades completas. Veja o exemplo da figura 11.13.

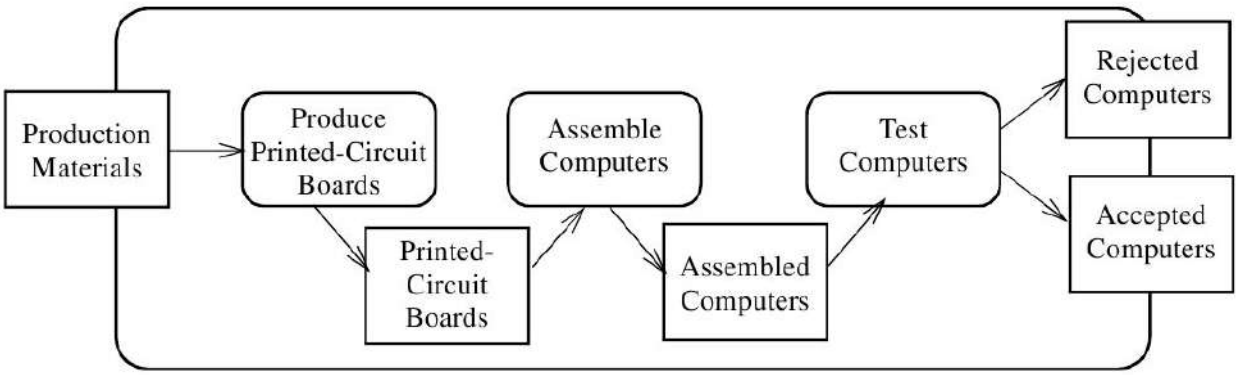


Figura 11.13: Objetos como Parâmetro para Atividades

11.5 Nós de Controle

Além de ações, atividades, eventos e objetos, os diagramas de atividade admitem um conjunto de assim chamados nós de controle, pois controlam o fluxo do processso. Esses nós podem ser visualizados na figura 11.14 a seguir.

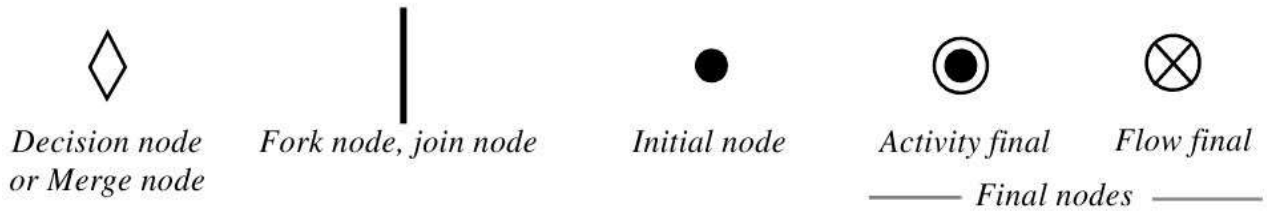


Figura 11.14: Nós de Controle

O primeiro nó de controle é o **nó de decisão**. Observe que um losango pode representar tanto um nó de decisão como um nó do tipo *merge*. Quando existe um único arco de entrada no nó e diversos arcos de saída, ele representa um nó de decisão. Nesse caso, cada arco de saída deve indicar uma condição de guarda, entre colchetes, condição que deve ser satisfeita para caracterizar o respectivo arco como a sequência de controle. Quando há diversos arcos de entrada e um único nó de saída, tem-se um nó do tipo *merge*. Esse nó é utilizado para agregar diversos fluxos de controle em um só.

O segundo tipo de nó de controle é o nó de sincronização. Esse nó é utilizado para representar fluxos de controle que ocorrem em paralelo, ou seja, ações que devem acontecer em paralelo. Nós de sincronização podem ser de dois tipos diferentes. O primeiro deles é um nó do tipo fork, que ocorre quando se quer indicar que, a partir daquele instante, as ações subsequentes devem acontecer em paralelo. Um exemplo de um nó do tipo fork é indicado na figura 11.15.

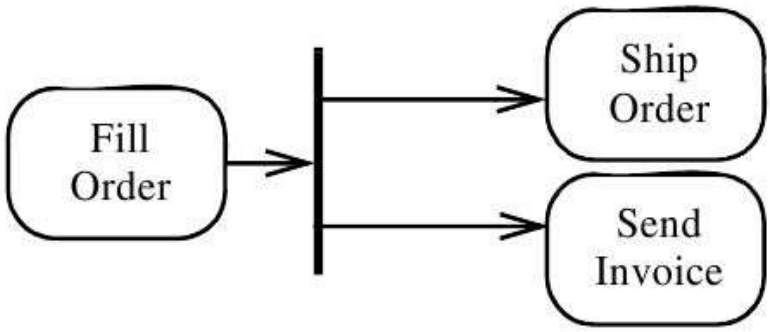


Figura 11.15: Exemplo de Nó do tipo fork

Na figura 11.16 temos o outro caso de sincronização, o nó do tipo *join*.

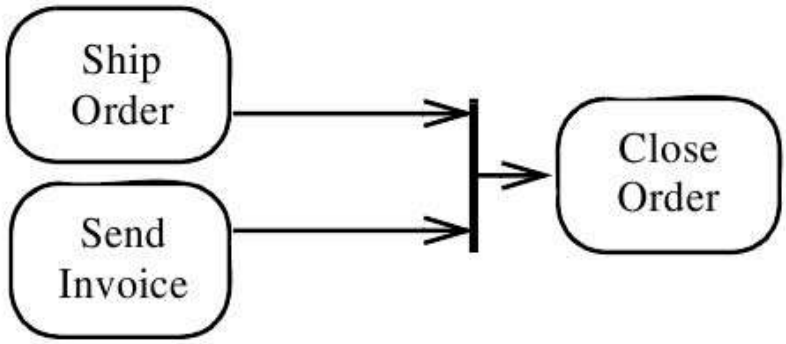


Figura 11.16: Exemplo de Nó do tipo join

Os nós do tipo join servem para indicar um ponto de sincronização entre fluxos de ação que estão acontecendo em paralelo. Somente quando todas as ações que chegam a um join estiverem concluídas é que a ação consecutiva pode ser executada.

O nó de início e de final já foram apresentados anteriormente. É importante, entretanto, ressaltar a diferença que existe entre um nó do tipo final de atividade e um nó do tipo final de fluxo. Quando o final de uma ação alcança um nó do tipo final de atividade, isso significa que a atividade como um todo, representada pelo diagrama termina. Quando o final de uma ação alcança um nó do tipo final de fluxo, somente o fluxo em questão é que termina, mas a atividade continua, em outros fluxos que estejam ainda em funcionamento.

11.6 Interrupções e Regiões de Interrupção

Em diagramas de atividade UML 2.0, introduziu-se o conceito de interrupção e regiões de interrupção. As interrupções (ou excessões) são notacionadas como arcos em forma de raio, conforme pode ser ilustrado na figura 11.17 a seguir.

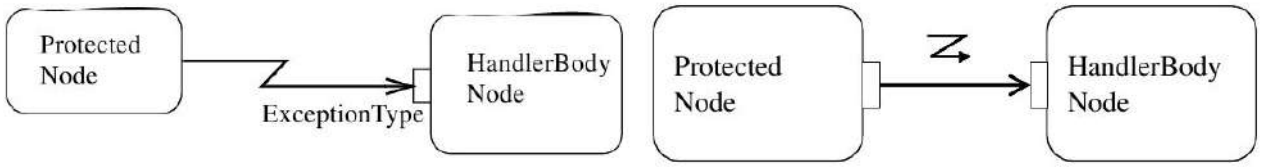


Figura 11.17: Exemplos de Notação para Interrupções

Pode ser interessante indicar o conjunto de ações onde as interrupções podem ocorrer. Quando isso for interessante, essas regiões podem ser demarcadas por meio de retângulos com linhas tracejadas. Um exemplo desta notação encontra-se na figura 11.18 a seguir.

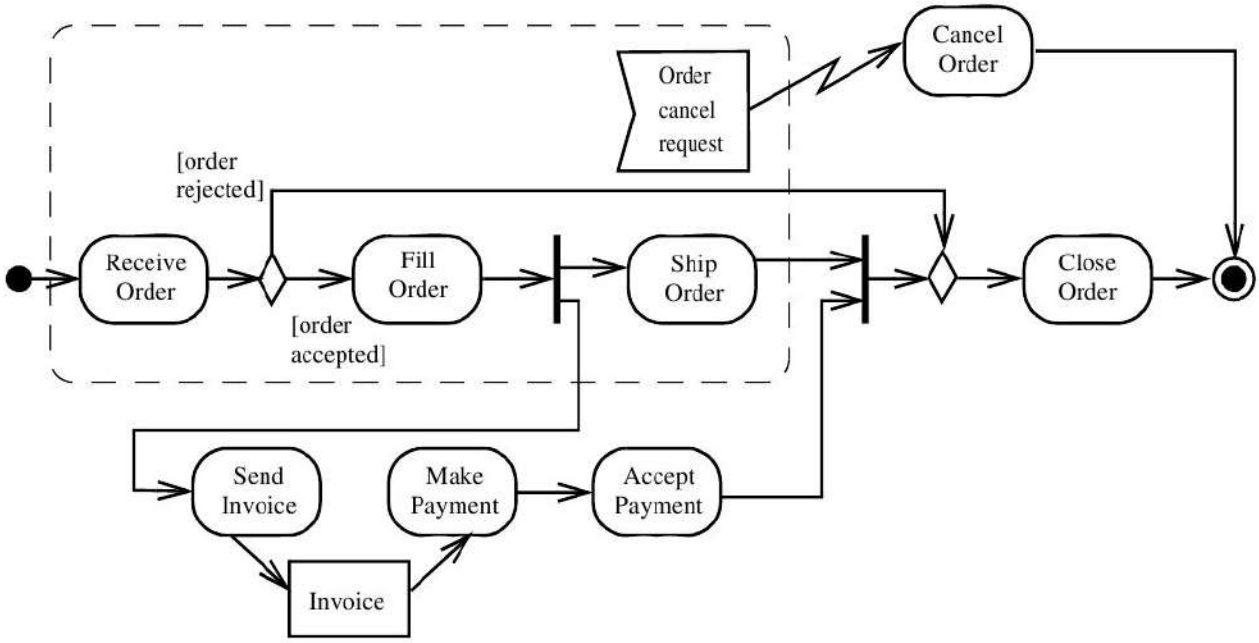


Figura 11.18: Regiões de Interrupção

Na figura 18 acima, representa-se que o evento de requisição de cancelamento de pedido só pode acontecer na região demarcada, ou seja, enquanto alguma das ações dentro da região estiverem sendo executadas. For a da região de interrupção, o evento não deve ser considerado.

11.7 Pontos de Extensão

Para evitar que linhas longas conectando pontos extremos de um diagrama tornem o diagrama muito poluído, podem ser utilizados pontos de extensão. Um exemplo de um ponto de extensão pode ser visto na figura 11.19 a seguir:

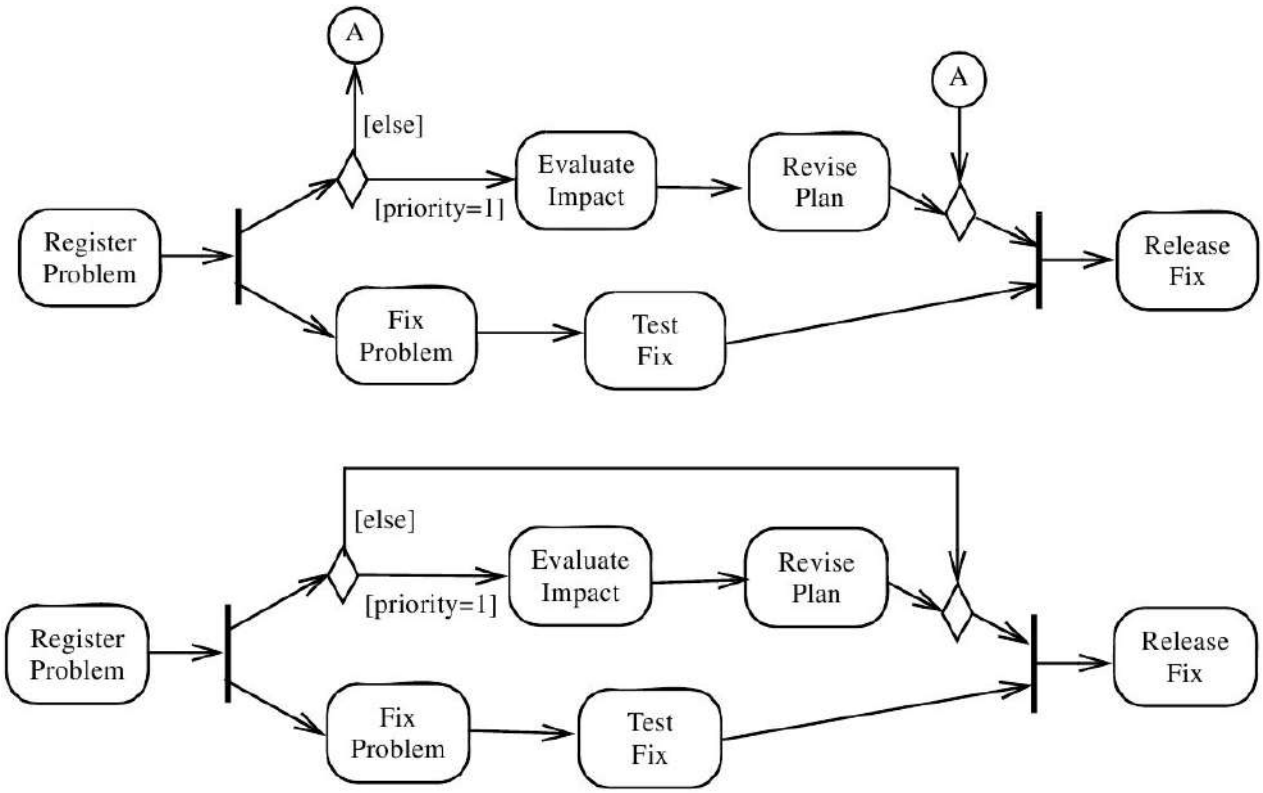


Figura 11.19: Pontos de Extensão

11.8 Partições

Sem o conceito de partições, os diagramas de atividade UML funcionam meramente como extensões de fluxogramas. A introdução do conceito de partição traz toda uma rica gama de representações para os diagramas UML. As partições podem ser representadas como indicado na figura 11.20 a seguir.

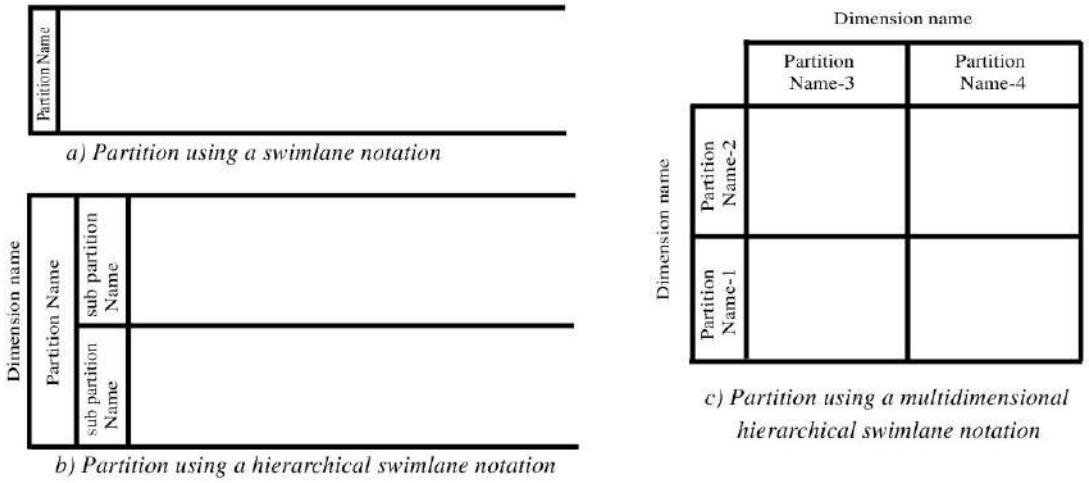


Figura 11.20: Exemplos de Partições

As partições servem para indicar que diferentes ações são executadas por diferentes agentes dentro de um processo. O caso mais interessante e que é dos mais importantes para nós é utilizar diagramas de atividade com partições para representar casos de uso. Nesse caso, uma partição é utilizada para representar o usuário e outra para representar o sistema. Um exemplo é apresentado na figura 11.21 a seguir.

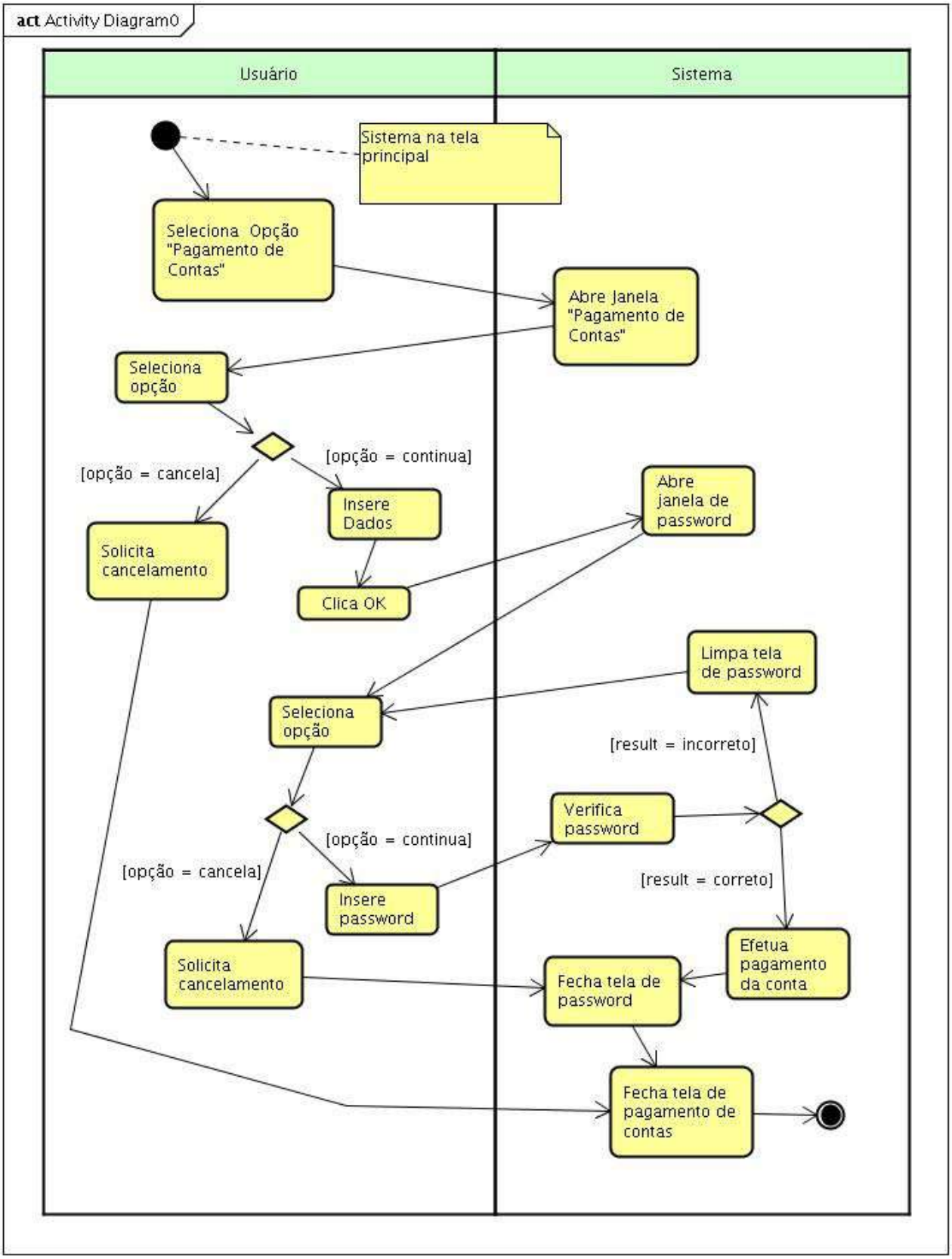


Figura 11.21: Uso de Diagrama de Atividades para Representar Casos de Uso

Diagramas de atividade como os da figura 11.21 são utilizados para detalhar os casos de uso levantados durante a especificação do sistema. Nesses diagramas, assume-se que o usuário do sistema realiza certas ações e o sistema, em resposta, reage realizando certas tarefas. Com isso, o comportamento do sistema pode ser especificado.

11.9 Outros Recursos em Diagramas de Atividades

Os diagramas de atividades possuem ainda, de acordo com a versão 2.3 da norma UML, outros recursos não apresentados aqui neste texto, que podem elevar sobremaneira a complexidade do diagrama. Dentre eles, o recurso de regiões de expansão, os pinos de entrada e saída, os nós estruturados, os conjuntos de parâmetros e outros, podem ser consultados diretamente no texto da norma.

12. Diagramas de Estado (Máquinas de Estado)

Os diagramas de estado (ou máquinas de estado, como aparecem na versão 2.3 da norma UML) são utilizadas para modelar um comportamento discreto em sistemas de transição entre estados finitos. Existem basicamente dois usos para máquinas de estado: máquinas de estado comportamentais e máquinas de estado para protocolos.

Máquinas de estado comportamentais podem ser utilizadas para especificar o comportamento de vários tipos de elementos. Por exemplo, podem ser utilizadas para modelar o comportamento de entidades individuais (objetos), por meio da modificação dos valores de seus atributos. O formalismo de máquina de estados neste caso é uma variante orientada a objetos dos *Statecharts* de Harel.

Máquinas de estado para representar protocolos expressam as transições legais que um objeto pode desenvolver. Com seu uso, pode-se definir o ciclo de vida de objetos, ou uma determina ordem na invocação de suas operações. Para este tipo de máquina de estado, interfaces e portas podem estar associados.

Os diagramas de estado UML possuem diversos detalhes que o tornam uma poderosa ferramenta para a modelagem de transição entre estados. Entretanto, neste texto, não iremos tratar de todos seus recursos, nos limitando a abordar suas características básicas. Leitores interessados nos detalhes mais sofisticados devem se remeter diretamente à norma.

12.1 Estado

Um estado modela uma situação durante a qual alguma condição (usualmente implícita) se mantém. Essa invariância tanto pode representar uma situação estática, como um objeto aguardando que um evento externo ocorra, como condições dinâmicas, como um processo apresentando um determinado comportamento. A notação para um estado pode ser visualizada na figura 12.1 abaixo.

Figura 12.1: Representação para um Estado

Alternativamente, um estado pode ter seu espaço interno subdividido em compartimentos, conforme ilustra a figura 12.2.

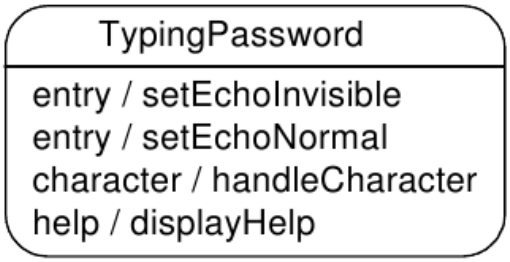


Figura 12.2: Exemplo de Estado com Compartimento Interno

O compartimento interno pode abrigar uma lista de ações ou atividades que podem ser realizadas quando se entra (*entry*), sai (*exit*) ou permanece (*do*) no estado. Outros tipos de comportamentos podem também ser definidos. A figura 12.3 mostra um exemplo de um pequeno diagrama de estados que usa esses recursos.

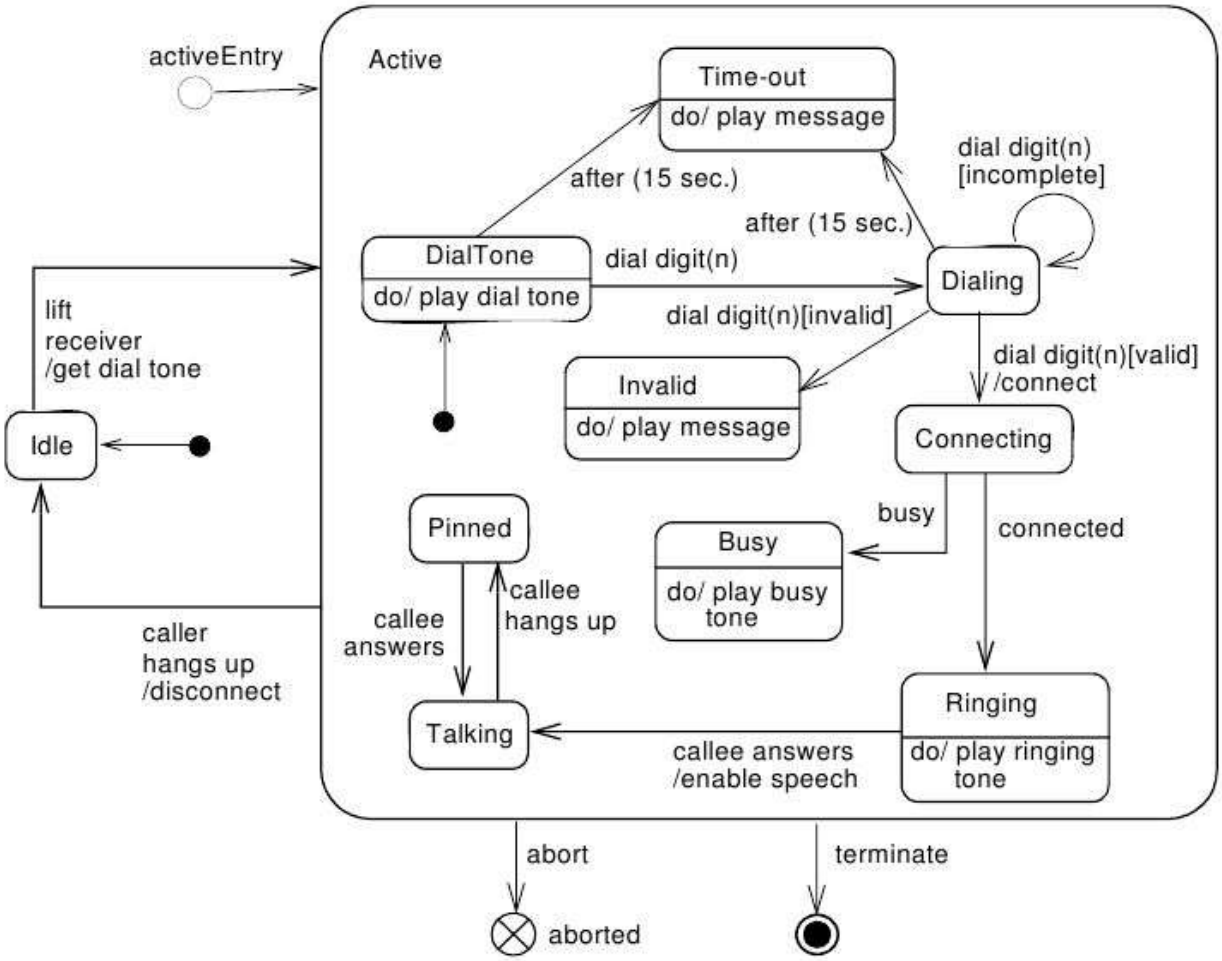


Figura 12.3: Exemplo de um Diagrama de Estado

A notação para os arcos que ligam os estados pode assumir a seguinte sintaxe, visualizada na figura 12.4.

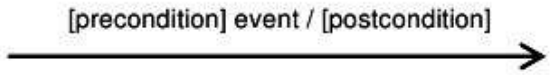


Figura 12.4: Notação para os Arcos entre Estados

Os estados não necessariamente precisam utilizar os compartimentos internos. O diagrama mostrado a seguir na figura 12.5 também é um diagrama de estados.

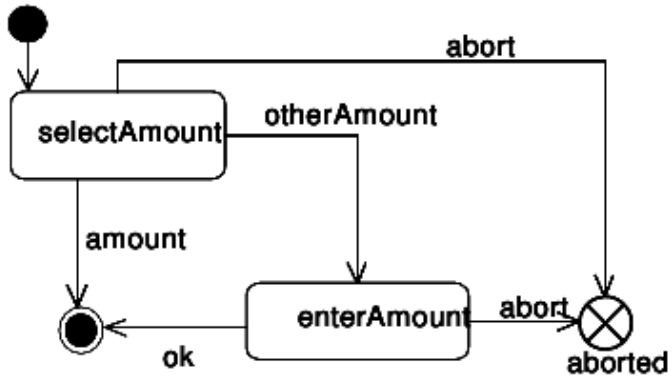


Figura 12.5: Estados sem Compartimentos Internos

Os diagramas de estado UML podem adquirir níveis bastante sofisticados de complexidade. A figura 12.6 a seguir ilustra um pouco da complexidade possível

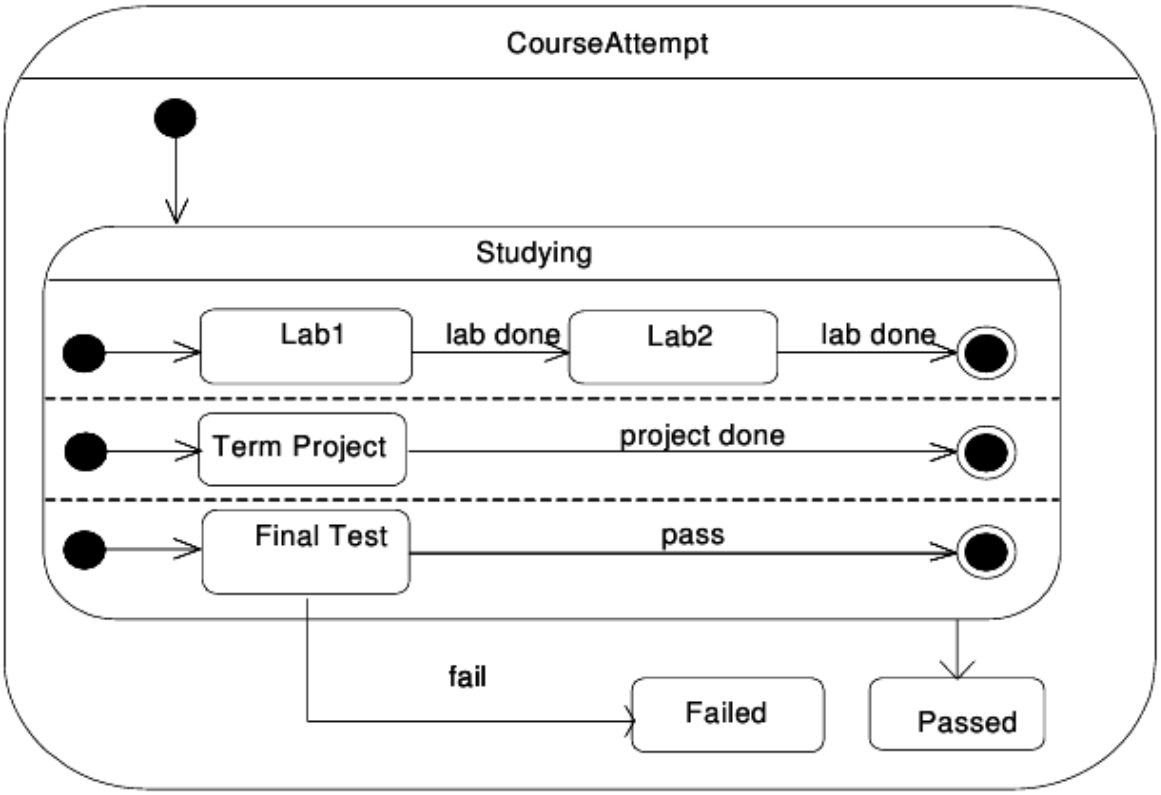


Figura 12.6: Diagrama de Estados mais Complexo

Como os estados de um diagrama de estados não demandam o uso dos compartimentos internos, observa-se que a notação é bastante semelhante à notação de uma ação em um diagrama de atividades. É necessário, entretanto, deixar claro que os dois diagramas têm uma semântica bastante diferente entre si. Nos diagramas de atividades, o comportamento está expresso fundamentalmente nos nós do diagrama. Cada nó representa um pedaço de comportamento. No diagrama de estados, ao contrário, todo o comportamento se encontra nos arcos do diagrama, sendo que os nós do diagrama de estados representa o que está nos arcos do diagrama de atividades, e os nós dos diagramas de atividades representam o que está nos arcos dos diagramas de estado. Assim, apesar de visualmente bastante similares, do ponto de vista semântico, o que é representado em cada diagrama é exatamente o oposto um do outro.

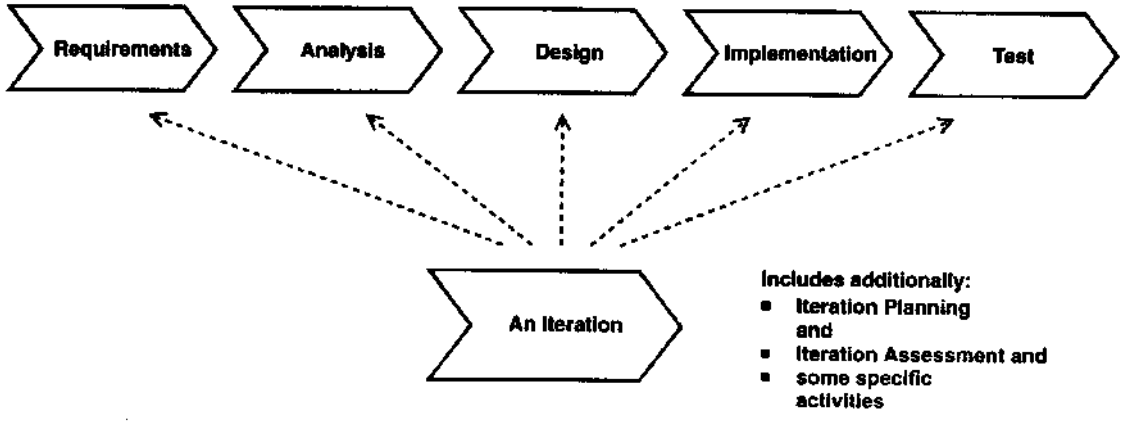
Em termos notacionais, já houve uma grande mudança nos diagramas de atividades e diagramas de estado, passando-se da norma UML 1 para o UML 2. As ações do diagrama de atividades no UML 1 tinha uma notação ligeiramente diferente da atual, que se faz muito mais próxima dos estados do diagrama de estados. Este autor supõe que no futuro, novas modificações poderão ocorrer nestes diagramas, para evitar ambiguidades.

PARTE 2

O Processo Unificado de Desenvolvimento de Software

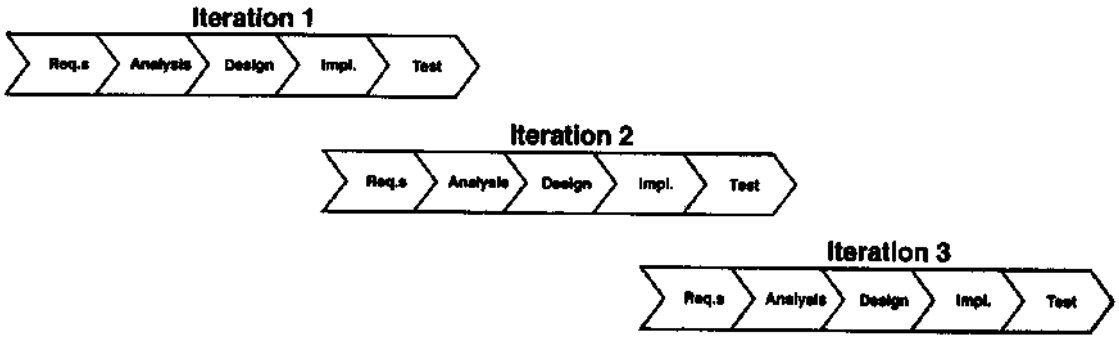
13. O Processo Unificado

O processo Unificado preconiza o chamado **desenvolvimento iterativo** de sistemas. Isso implica que o desenvolvimento do sistema se dará por meio de múltiplas iterações, passando por diversas fases.

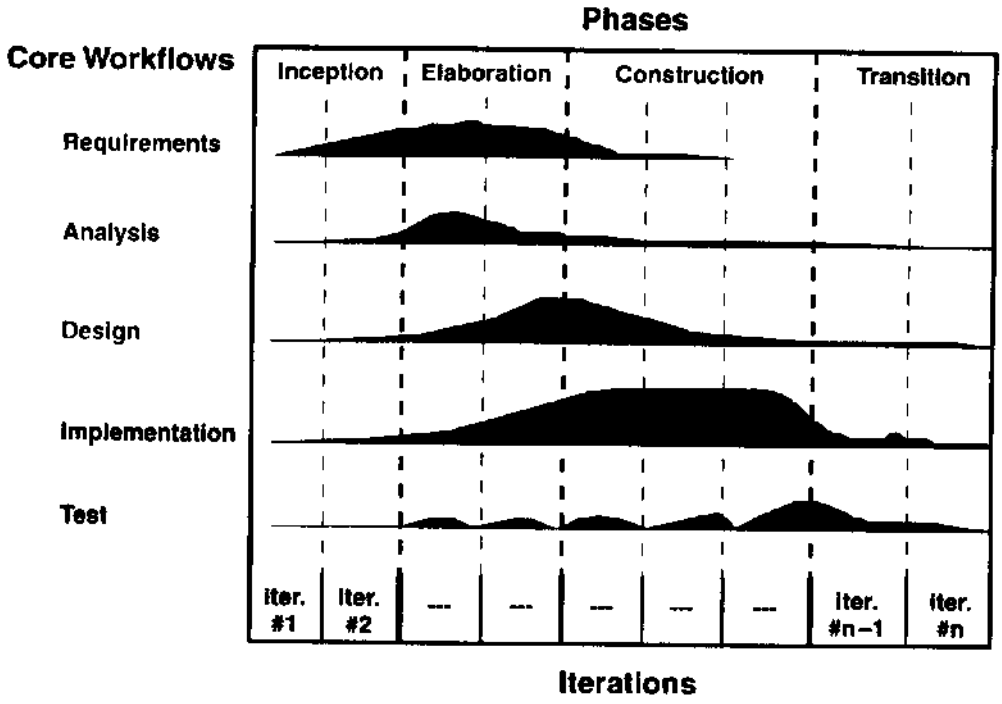


Cada iteração passa por etapas de:

- Especificação dos requisitos (ER)
- Análise (A)
- Design (D)
- Implementação (I)
- Testes (T)

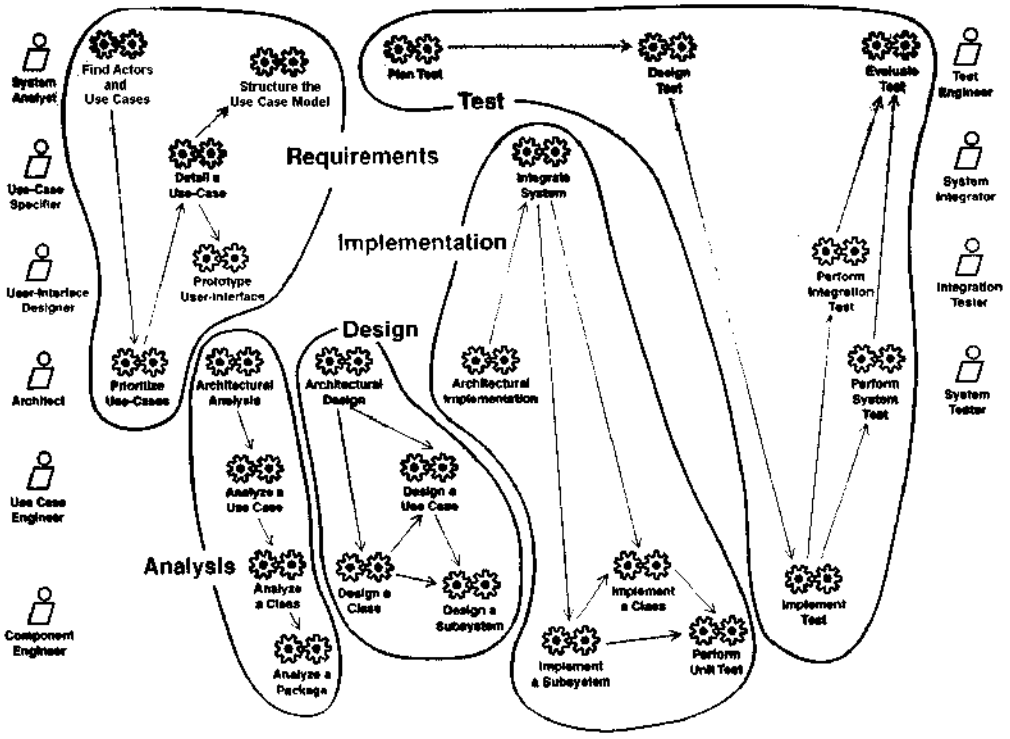


As múltiplas iterações irão então repetir as etapas de ER, A, D, I e T a cada iteração. Entretanto, a quantidade de tempo gasta em cada iteração não será a mesma em cada iteração. As primeiras iterações demandarão maior esforço nas fases de ER e menor esforço nas fases de I e T, por exemplo. Nas iterações mais avançadas (próximas do release do sistema), as fases de ER terão menor tempo dispendido e as fases de I e T terão maior tempo. Uma visão aproximada desse dispêndio de tempo pode ser dada na figura seguinte, que ilustra o ciclo de vida para o desenvolvimento de um software



Observe que nas primeiras iterações, o esforço na ER é maior e cresce até chegar a um pico e depois decai. O mesmo ocorre com as outras atividades.

Cada etapa (ER, A, D, I, T) é formada por um conjunto bem definido de atividades, ilustradas na figura a seguir:



Observe que diferentes trabalhadores estarão responsáveis por diferentes atividades, sendo que algumas atividades podem até mesmo ser executadas em paralelo. Essas atividades serão estudadas em detalhes nos módulos seguintes do curso. Vocês, alunos, terão a chance de exercitar-se em cada uma das atividades que compõem o processo Unificado, no desenvolvimento do sistema que será implementado durante o curso. Somente as atividades da etapa de Teste terão que ser suprimidas, pois não dispomos de ferramentas adequadas para sua efetivação.

Observe que normalmente a etapa de especificação dos requisitos é a etapa em que se faz a proposição do problema. A etapa da análise corresponde à investigação do problema. A etapa de design apresenta uma solução lógica para o problema e a fase de implementação e testes é responsável pela geração de código que resolve o problema

Desde sua proposição original, em (Jacobson et.al. 1999), o Processo Unificado sofreu diversas modificações e aperfeiçoamentos. Nessas modificações, algumas fases foram acrescentadas, outras foram modificadas e aperfeiçoadas para tornar o processo mais eficiente e menos ambíguo. Uma das modificações proposta neste ínterim foi a preconização de uma fase anterior ao ciclo de desenvolvimento envolvendo múltiplas iterações, que neste documento será chamada de pré-projeto. Normalmente, as atividades desenvolvidas no pré-projeto são executadas uma única vez durante o ciclo de desenvolvimento de software, ao contrário das fases de especificação de requisitos. As atividades do pré-projeto envolvem normalmente a assim chamada "Modelagem do Negócio". Como na maioria das situações, o software sendo desenvolvido visa aumentar a produtividade ou melhorar a eficiência de um negócio ou empresa, presume-se que seria de alguma utilidade para o desenvolvimento do software conhecer em detalhes os aspectos do negócio da empresa. Essa presunção origina-se da interpretação de que os requisitos de um software devem expressar as necessidades reais do contratante, e não seus desejos. Muitas vezes o contratante de um desenvolvimento de software não tem claras suas reais necessidades, sendo que o puro atendimento de seus desejos poderia levar à construção de um software que não atendesse as reais necessidades do cliente, levando a seu descontentamento com o resultado.

Em nossa customização do processo Unificado que iremos estudar para fins didáticos, utilizaremos 3 diferentes artefatos que envolvem a modelagem de negócios, e que por conseguinte serão desenvolvidas nesta fase de pré-projeto:

- A Elaboração do Documento de Visão do Problema
- O Modelo Conceitual de Domínio
- Negócios e Processos de Negócios

Após desenvolvido o pré-projeto, iniciaremos efetivamente o ciclo de iterações que compõem o processo Unificado, estudando as fases de Especificação de Requisitos, Análise, Design, Implementação e Testes.

14. O Documento de Visão do Problema

Durante a fase do pré-projeto, ou seja, antes de começarmos as iterações que correspondem ao projeto de um software, segundo o processo Unificado, é necessário fazermos um planejamento prévio de como esse desenvolvimento se dará. Esse planejamento se consolida na criação de um documento que é chamado de **Documento de Visão do Problema**. Esse nome se justifica, pois seu conteúdo dará uma visão geral de como se imagina o sistema que pretendemos construir. Não existe uma **norma** regulando como esse documento de visão deve ser desenvolvido. Entretanto, utilizaremos como guia uma sugestão dada no livro de Larman (Larman 1998), de um conjunto de capítulos que de um modo geral constituem uma boa sequência de elementos para definir a visão de um problema. Em seu livro, Larman propõe a criação de 5 capítulos, entitulados:

- Compreensão do Problema
- Proposta de Solução de Software
- Visão Geral dos Pré-Requisitos
- Planejamento Logístico
- Glossário

No capítulo referente à **Compreensão do Problema**, devemos incluir um texto, geralmente 2 ou 3 parágrafos, com uma descrição verbal do domínio do problema. Essa descrição tentará abstrair o problema que se tenta resolver, sem envolver conjuntamente a proposta de software que (seguramente) já temos para sua resolução. Podemos entender esse capítulo como sendo uma espécie de elicitação das necessidades dos investidores, ou seja, aqueles que em princípio estão nos pagando para que façamos o software. Por exemplo … vamos supor que em nossa proposta de tema, sugerimos a criação de um banco de dados para uma loja de peças de automóvel, sugerindo que nossos investidores possuem uma loja de peças de automóveis e resolveram gastar dinheiro desenvolvimento um sistema de software para resolver algum problema que lá apareceu (pois se não existe um problema, um comerciante não se iria pensar em gastar dinheiro sem mais nem menos !). Na compreensão do problema, devemos tentar abstrair o real problema que se coloca e que motivou essa proposta. Ou seja, na verdade o problema que se tem é o fato de que uma loja de peças de automóveis possui um número muito grande de dados e informações que precisam ser organizados, acessadas e atualizadas de maneira eficiente. Tais informações envolvem clientes, fornecedores, produtos, peças, etc. que necessitam ser organizadas e controladas em seu cotidiano de funcionamento. Devemos portanto descrever aqui quais seriam as particularidades desse problema em questão, descrevendo características relacionadas ao domínio em questão (no caso, uma loja de peças de automóveis), e não a solução previamente escolhida para combatê-la. Muitas vezes, o cliente já vem com o que ele pensa ser uma solução. Por exemplo … o cliente chega e já vai dizendo … "quero um banco de dados para cadastrar as auto-peças". Um bom engenheiro de computação deve nessa hora saber questionar a si mesmo se é exatamente essa a solução que o cliente precisa. Caso contrário, após o programa ficar pronto, você terá um cliente insatisfeito, pois apesar de você ter construído o que ele pediu, isso que ele pediu não resolve o problema dele. Por exemplo, se o negócio do cliente envolve também um estoque rotativo, utilizado por vários funcionários, talvez o que ele precise realmente seja um controle de estoque mais eficiente, que informe também qual funcionário fez uso de qual peça. Um simples cadastro de peças não resolveria por exemplo um problema de "sumiço" de peças do estoque, que é o real problema do

cliente e que ele implicitamente deseja solucionar. Precisamos pois ser bem críticos, para garantir que conhecemos o problema de nosso cliente, e que a solucão que estaremos propondo realmente é o que ele precisa. Nem sempre um banco de dados é a solução para os problemas de um cliente. Veja por exemplo outros exemplos de problemas:

- Deseja-se desenvolver um novo produto eletrônico (e.g. um telefone celular), com um conjunto rico de funções (e.g. discagem rápida, memória, discagem de emergência, etc.)
- Têm-se um processo físico que se deseja modelar e analisar, de tal forma a viabilizar a predição de seu comportamento e o teste do impacto de diferentes decisões nos resultados
- Têm-se uma organização, com toda sua logística e métodos, e se deseja melhorar sua eficiência, confiabilidade e custo
- Deseja-se produzir algum tipo de produto de maneira eficiente e confiável, utilizando máquinas automátizadas

Para cada um dos domínios acima, existe claramente um problema que necessita ser resolvido. No presente capítulo, nossa missão é descrever sucintamente esse problema.

No capítulo referente à **Proposta de Solução via Software**, devemos, aí sim, descrever sumariamente como estamos visualizando um sistema de software para resolver o problema. Aqui devem ser estabelecidos metas e objetivos que desejamos atingir com o software que será desenvolvido. Esse capítulo, a semelhança do anterior, será um texto enxuto que descreva de maneira geral qual é o tipo de sistema de softare que se preconiza para o problema do cliente. Por exemplo, para o problema da loja de peças, a solução óbvia é a criação de um banco de dados que gerencie não somente as peças mas também os funcionários e o estoque do estabelecimento. Para os outros exemplos apontados, teríamos (na ordem em que foram apresentados):

- Usar microprocessadores ou micro-controladores como parte do produto e um sistema embutido que proveja as funcionalidades desejadas
- Desenvolver um simulador para simular o processo e prover estatísticas para a análise de decisões
- Uso de ferramentas de produtividade (e.g. ferramentas do tipo office, e-mail, web-sites, etc), e aplicações de software de propósito específico, de tal forma a automatizar parte do processo organizacional
- Desenvolvimento de um software de controle de sistemas para controlar as máquinas envolvidas na produção dos produtos desejados

No capítulo referente à **Visão Geral dos Pré-Requisitos**, devemos definir basicamente um conjunto de características que desejamos para nosso sistema, na forma de funções e atributos do sistema. As funções do sistema devem descrever basicamente o que se supõe que o sistema deve **fazer**. Os atributos do sistema devem descrever o que se supõe que o sistema deve **ser** ou **ter**. Por exemplo, funções de um sistema editor de plantas residenciais incluiriam desenhar plantas, alterá-las, imprimí-las, exportá-las em formatos de outros editores, projetar a planta tridimensionalmente, calcular a quantidade materiais necessários para a obra, etc. Para esse mesmo exemplo, atributos do sistema seriam por exemplo, dizer que o sistema opera em múltiplos sistemas operacionais, permite a edição de paredes, janelas, portas e pisos, opera em preto e branco ou colorido, permite a inserção de móveis para a visualização da projeção 3D, etc. Observe que as funções denominam ações que o sistema deve executar, ao passo que os atributos modelam características de capacidade ou de variações que o sistema apresenta. Outro ponto é que os atributos sempre podem ser colocados na forma atributo-valor:

Sistemas operacionais	-	múltiplos
Caracteres editáveis	-	paredes, portas, janelas, pisos
Tipos de Impressão	-	preto e branco e colorida
Tipos de Visualização	-	com móveis ou sem móveis

Para determinarmos as funções e atributos do sistema, devemos fazer um brainstorm, normalmente junto com o cliente, propondo "features" que passariam a caracterizar o sistema. Tome muito cuidado para não confundir funções com atributos. Tanto funções como atributos devem ser colocados na forma de tabelas, indicando, dentre outras coisas os seguintes quesitos:

Status da Função/Atributo	-	(proposto, aprovado, incorporado, validado, etc..)
Custo Estimado de Implementação	-	(em termos de tipos de recursos e homens-hora)
Prioridade	-	(crítica, importante, desejável, dispensável)
Nível de Risco na Implementação	-	(crítico, significativo, ordinário)

O Status da Função/Atributo indicará qual o status da função ou atributo em questão. Assim, pode-se planejar com o cliente quais as funções/atributos que se desejam implementar. O custo estimado de implementação permite ao cliente fazer escolhas quanto às funções atributos que deseja custear (lembrando que tudo tem um custo). A prioridade define se a omissão da função ou atributo pode comprometer o projeto como um todo ou se a função atributo poderia ser suprimida sem problemas, caso o cliente deseje. Por fim o nível de risco na implementação caracteriza as dificuldades previstas na incorporação da função/atributo, caso o cliente opte por incluí-lo.

O capítulo **Planejamento Logístico** visa determinar a equipe de trabalho que será envolvida no projeto e detalhes do planejamento logístico para o desenvolvimento em questão. Esse capítulo deve apresentar os seguintes ítens:

- Equipes de Trabaho
- Grupos Afetados
- Fatos Presumidos
- Riscos
- Dependências

A *equipe de trabalho* deve indicar claramente as pessoas que estarão envolvidas no projeto. No caso de vocês, deve conter o nome, RA e e-mail de cada um dos elementos participantes do grupo. Em um caso geral, poderia ainda conter endereços e telefones dos elementos participantes, bem como qualquer outra informação relevante, tal como a formação acadêmica, anos de experiência, etc.

O ítem *grupos afetados* deve indicar quais os grupos afetados pelo projeto em questão. Em uma grande empresa, muitas vezes o projeto de um software envolve diversos grupos dentro da empresa. De um modo geral é necessário saber-se quais serão os grupos afetados, ou seja, que estão esperando a conclusão de nossas atividades para dar continuidade em seus próprios projetos.

O ítem *fatos presumidos* deve incluir os fatos que se presumem verdadeiros antes de se iniciar o projeto. Por exemplo, presume-se que determinado equipamente estará a disposição, ou que determinados elementos estarão a disposição para colaborar com o projeto. Muitas vezes é importante se documentar os fatos presumidos, para que o cliente fique ciente das condições em que o planejamento logístico está sendo efetuado, de tal forma que, caso haja alguma modificacão nas condições operacionais, ele esteja ciente de que isso poderá afetar o projeto. Da mesma forma, os fatos presumidos servem para estabelecer um acordo mútuo entre o cliente e a empresa de desenvolvimento, onde cada um está ciente das condições em que se dará o projeto e o que a equipe

de desenvolvimento espera do cliente em relação a colaboração, recursos, etc.

O ítem *riscos* apontam fatos, eventos ou situações que podem levar ao fracasso ou atrasos no projeto, bem como as potenciais consequências do fracasso ou do atraso. É importante o cliente estar ciente dos riscos envolvidos e ao mesmo tempo saber que os desenvolvedores também estão cientes da possibilidade de que ocorram.

O ítem *dependências* devem identificar outros parceiros, sistemas ou produtos dos quais o projeto depende para sua implementação, e que podem eventualmente levar ao aparecimento de riscos.

Assim, o planejamento logístico deve documentar as condições em que o desenvolvimento será efetuado e qual a logística que será implementada.

O capítulo **glossário**, por final, deve conter um levantamento do vocabulário relativo ao domínio, ou seja, contendo os principais termos utilizados para descrever as características do problema..

15. O Modelo Conceitual de Domínio

O modelo conceitual de domínio visa a aquisição e modelagem de informações sobre o domínio organizacional sob o qual o contexto do projeto se insere. Em outras palavras, ele visa estruturar o conhecimento que se tem sobre o problema que se deseja resolver. Em alguns tipos de software, esse modelo do domínio tem uma importância somente marginal, mas para outros, tais como bancos de dados, por exemplo, o modelo conceitual de domínio tem uma importância muito grande.

De uma maneira geral, o modelo de domínio envolve o contexto onde o software será aplicado, e sua compreensão, antes da introducão do sistema de software que se pretende desenvolver. Novamente, em outras palavras: antes de nos colocarmos a pensar no software que iremos desenvolver, gastaremos um certo tempo conhecendo o problema que queremos resolver: seu contexto, sua terminologia, seu jargão, o conhecimento e descoberta de relações existentes entre seus elementos, etc. Assim, vemos que o modelo conceitual ilustra conceitos significativos de um domínio - aquilo que devemos estar cientes a respeito do domínio, representando sempre coisas do mundo real **não** componentes de software.

O modelo conceitual é obtido por meio de um procedimento chamado de Análise de Domínio. Nesta análise, os conceitos apurados são expressos por meio de um diagrama de classes UML. Os possíveis relacionamentos entre os conceitos são expressos por meio de associações entre os conceitos, e as particularidades envolvendo os conceitos representados são expressas por meio dos atributos nos diagramas de classe.

Nesse ponto, é necessário fazermos uma introdução, comparando a metodologia de análise orientada a objetos com a metodologia de análise estruturada. A análise estruturada enfoca nossos esforços na descoberta de processos ou funções, ao passo que a análise orientada a objetos focaliza a descoberta de conceitos e seus inter-relacionamentos. É exatamente isso que fazemos aqui na análise do domínio.

Um dos principais métodos na análise do domínio consiste na busca por entidades participantes da Lista de Categorias de Conceitos. Essa lista abrange as seguintes categorias:

- objetos físicos ou tangíveis
- especificações ou descrições de coisas
- lugares
- transações
- ítens sendo transacionados
- papéis de pessoas
- coisas que contém outras coisas
- coisas que são contidas em outras coisas
- regras e políticas
- eventos
- catálogos de coisas

Assim, devemos procurar em nosso entendimento do domínio, todos os conceitos que possam ser categorizados de objetos físicos ou tangíveis, especificações ou descrição de coisas, lugares, transações, … , etc..

O modelo conceitual passa então a ser assim elaborado: inicialmente fazemos um levantamento dos

conceitos candidatos utilizando a lista acima. Em seguida, fazemos a inserção dos conceitos levantados no diagrama de classes. Em seguida, passamos a procurar por possíveis associações existentes entre os conceitos, representando as mesmas também no diagrama de classes. Por fim fazemos a inserção dos atributos necessários.

Algumas "dicas" são importantes durante essas atividades. Por exemplo, … , devemos evitar a representação de conceitos irrelevantes. Ou seja, tratemos de usar nomes de coisas que realmente existem, excluindo detalhes que possam ser irrelevantes. Uma boa política é utilizarmos o princípio do cartógrafo (desenhista de mapas). O que devemos colocar em um mapa quando o elaboramos ? Vemos que os ítens que aparecem nos mapas são os ítens relevantes para a descrição geral da região coberta pelo mapa. Por exemplo, um mapa pode apresentar a localização de parques e jardins, igrejas e outros edifícios relevantes, mas dificilmente mostrará onde ficam os postos de gasolina e os números das casas na rua. Assim, durante a elaboração do modelo conceitual, devemos nos preocupar em não poluir o modelo, e introduzir somente os elementos importantes para a compreensão do domínio, evitando conceitos que não sejam relevantes. Por exemplo: imaginemos que queremos elaborar o modelo conceitual de uma aula em sala de aula. Assim, diversos conceitos importantes surgirão, tais como *lousa*, *carteira*, *matéria*, *professor*, *aluno*, *caderno*, *lápis*, *giz*. Outros conceitos, que existem de fato em uma sala de aula, tais como *cortina*, *vidro*, *interruptor*, *piso*, *teto*, etc. não são relevantes para a descrição do domínio, não sendo portanto incluídos no modelo conceitual.

Uma outra dúvida frequente que aparece durante o levantamento do modelo conceitual é saber quando um determinado conceito deve ser representado como um conceito e quando deve ser apresentado como um atributo. Essa dúvida é muito pertinente. Imagine durante o levantamento de um modelo conceitual que nos deparamos com o conceito *cliente*. Bem, analisando esse conceito, descobrimos que todo *cliente* deve ter um *endereço*. O que é o *endereço* ? Um novo conceito ou um atributo de *cliente* ? Para resolver esse conflito, utilizaremos uma regra prática muito utilizada: se um candidato a conceito X não pode ser pensado como um número ou um texto no mundo real, então X é realmente um conceito. Caso contrário é um atributo. Na dúvida, devemos sempre torná-lo um conceito separado. Com relação ao nosso *endereço*: se podemos imaginar que o *endereço* é somente um texto (algo como "Rua dos Alecrins, 32 -13083-560 Campinas SP", então devemos tornar *endereço* um atributo. Se, ao contrário, quisermos desmembrar *endereço* em *rua*, *número*, *CEP*, *cidade*, *estado*, então *endereço* passa a ser um conceito independente, e *rua*, *número*, *CEP*, *cidade* e *estado* passam a ser atributos de *endereço*.

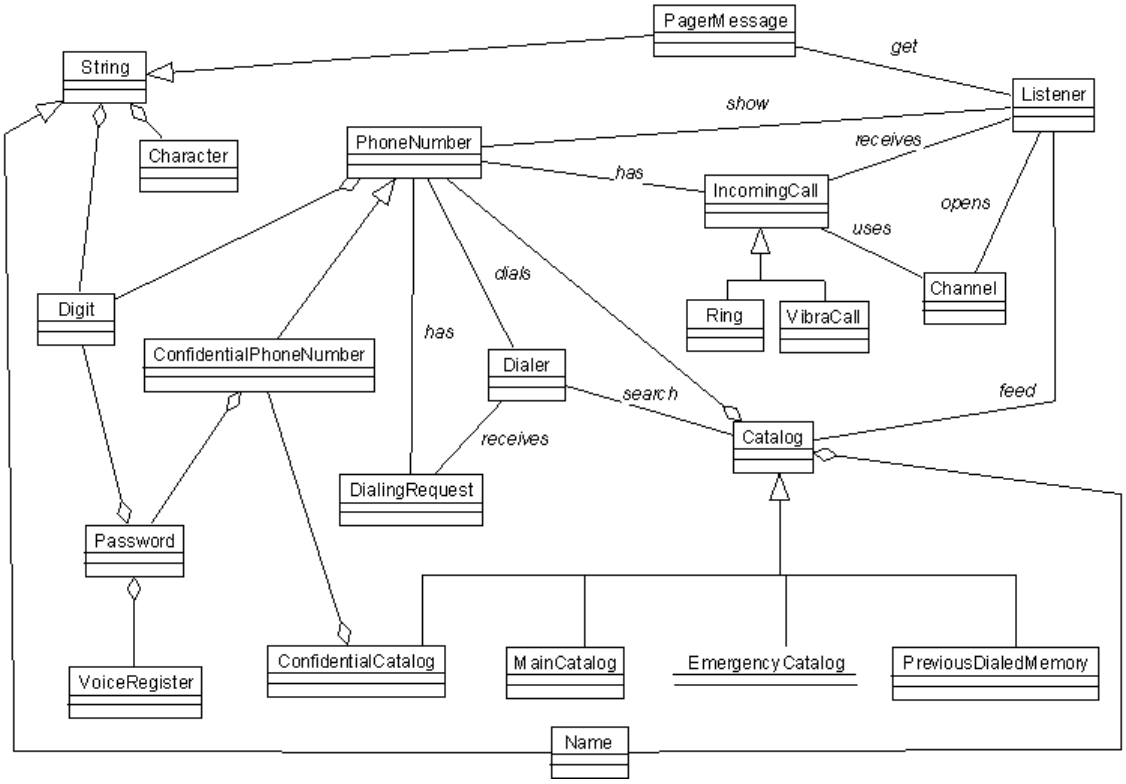
Com relação à inserção de associações: associações representam um relacionamento entre conceitos que indicam uma conexão significativa ou interessante entre conceitos. Assim, um bom critério para a elicitação de associações úteis é verificar se um possível relacionamento entre conceitos é um conhecimento que deva ser preservado durante certo tempo. Nesse caso, essa associação é útil e deve ser inserida no modelo. Uma outra maneira de descobrir associações úteis é verificar se o relacionamento corresponde a um dos relacionamentos presentes na Lista de Associações Comuns. Assim, devemos incluir uma associação no diagrama de classes entre os conceitos A e B se:

- A é uma parte física de B
- A é uma parte lógica de B
- A está fisicamente contido em B
- A está logicamente contido em B
- A é uma descrição para B
- A é um ítem transacionado por B

- A é armazenado em B
- A é membro de B
- A utiliza ou gerencia B
- A se comunica com B
- A possui ou é possuído por B
- A está próximo de B

Entretanto, devemos tomar cuidado durante o levantamento de associações. Uma armadilha comum é gastar muito tempo tentando descobrir associações. Normalmente, encontrar conceitos é muito mais importante que encontrar associações. Deve-se gastar mais tempo na descoberta de conceitos do que de associações. Muitas vezes, a inserção de muitas associações tendem a confundir, mais do que iluminar nosso modelo conceitual.

Um exemplo de um modelo conceitual de domínio referente ao domínio de telefones celulares é mostrado a seguir. Observe-o e tente interpretá-lo, para verificar se já compreende a notação UML.



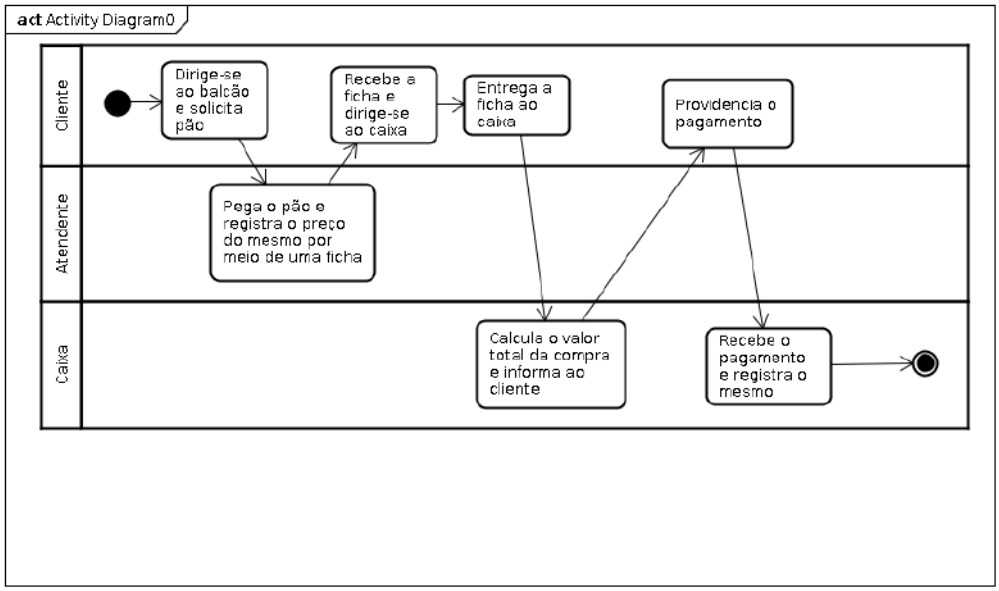
Uma última dúvida frequentemente persegue aqueles que se iniciam na arte do levantamento de modelos conceituais: saber quando parar. Até onde devemos continuar perscrutando nossas mentes atrás de mais conceitos, associações e atributos ? Normalmente não existe uma resposta fácil para esta questão. Entretanto, novamente, usaremos uma regra prática: devemos olhar para o modelo como um todo e nos perguntarmos: será que isso representa o domínio como um todo ? Se concordarmos que sim, paramos. Se não, continuamos mais um pouco. Entretanto, devemos sempre impor uma condição de limite absoluto de tempo. O que é isso ? Dizemos para nós mesmos: temos mais X minutos (ou horas, dependendo). Assim que o tempo acabar, terminamos a análise do domínio. Com qualquer modelo que tenhamos !

16. Negócios e Processos de Negócios

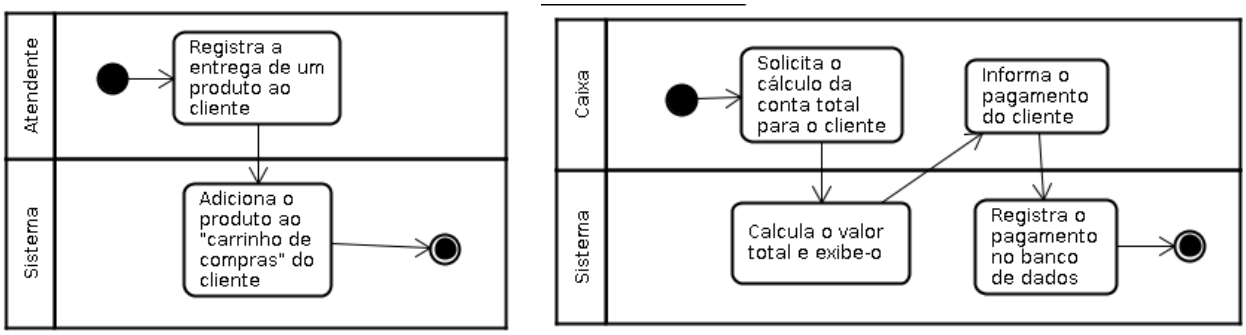
Um **negócio** é uma atividade econômica exercida por uma pessoa ou grupo de pessoas visando a obtenção de recursos econômicos. Desta forma, podemos atribuir o termo "negócio" desde às atividades desenvolvidas por profissionais autônomos como médicos, dentistas, advogados, passando por estabelecimentos de pequeno porte como comerciantes, e donos de pequenos estabelecimentos comerciais, oficinas mecânicas, restaurantes, postos de gasolina, até grandes conglomerados de pessoas organizadas em indústrias, corporações, joint-ventures e empresas multi-nacionais.

Para gerir o negócio, usualmente utiliza-se do trabalho de diferentes pessoas, possivelmente organizadas em departamentos ou seções especializadas, onde cada participante do negócio possui uma função ou atribuição, para que a dinâmica do negócio se consolide de forma a resultar na obtenção dos recursos econômicos que são o objetivo do negócio.

Um **modelo de negócios** descreve a lógica por meio da qual uma organização captura, cria, gerencia e disponibiliza valores. Além de um Modelo Conceitual de Domínio, que normalmente descreve os principais recursos (recursos humanos, materiais ou econômicos) envolvidos em um negócio, um modelo de negócio envolve a descrição de um ou mais processos de negócios. Um **processo de negócio** corresponde a uma sequência de interações entre os diferentes participantes de um negócio, (incluindo aí os clientes do negócio), cada um com sua função, de tal forma que o resultado seja a geração de valores para o negócio, bem como para os clientes do negócio. Desta forma, para a caracterização de um processo de negócio, deve-se apontar quais são os personagens envolvidos e quais as atividades desempenhadas por cada personagem. Diversas linguagens de modelagem para processos de negócios foram desenvolvidas (e.g. BPML, BPEL, BPMN). Uma alternativa, utilizando UML é o desenvolvimento de casos de uso de negócios, onde os processos de negócios são modelados de maneira análoga aos casos de uso em desenvolvimento de sistemas. A diferença entre uma caso de uso de negócios e um caso de uso do sistema pode ser bastante sutil, à medida em que sistemas computacionais são utilizados para dar apoio a processos de negócios. De uma maneira geral, quando se desenvolve casos de uso do sistema, descreve-se a interação entre diferentes usuários e o sistema, de forma a agregar algum valor ao usuário. Já nos casos de uso de negócios, a interação não é entre os usuários e o sistema, mas entre os diferentes participantes de um negócio. Vamos ilustrar essa diferença por meio de um exemplo. Imaginemos como negócio uma padaria, e um processo de negócio que seja a venda de pães a um cliente. Poderíamos detalhar esse processo de negócios por meio de um caso de uso de negócios descrito por meio de um diagrama de atividades:

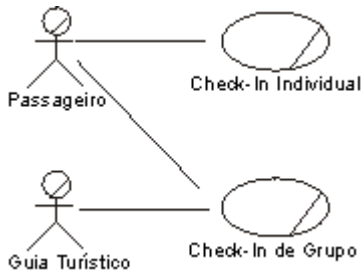


Esse mesmo caso de uso de negócios daria margem a pelo menos dois diferentes casos de uso do sistema: "Registro de Pedido" e "Pagamento de Compra".



Observe que, apesar do cliente participar do caso de uso de negócios, nos casos de uso do sistema, o cliente não aparece em nenhum momento, pois o mesmo nunca interage diretamente com o sistema. Desta forma, é importante distinguir entre **casos de uso de negócio** e **casos de uso de sistema**, e compreendermos a diferença que existe entre os dois. No caso de uso de negócio, todas as interações referentes ao negócio devem ser detalhadas. No caso de uso do sistema, somente as ações que envolvam a interação com o sistema são detalhadas, e somente os usuários que interagem com o sistema devem ser representados e ter suas ações descritas.

Casos de uso de negócios são representados em diagramas de casos de uso, da mesma maneira que os casos de uso do sistema. Alternativamente, pode ser usada uma versão estereotipada de atores e casos de uso, para indicar que se trata de casos de uso de negócios. Um exemplo deste caso é apresentado na figura a seguir:

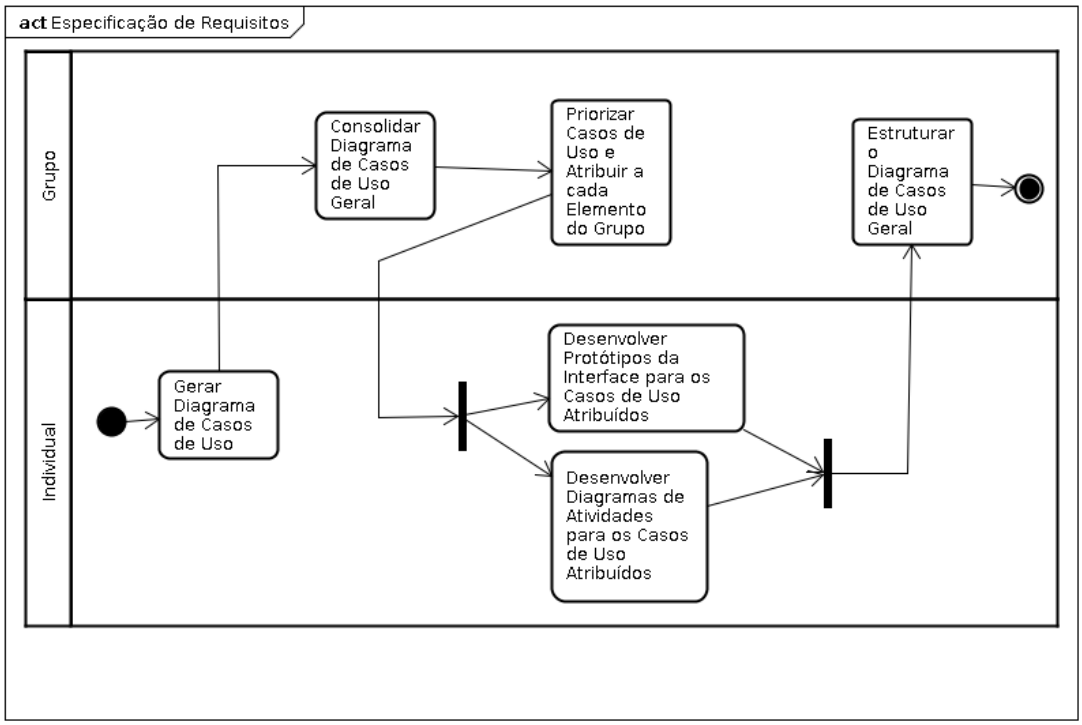


O levantamento (por meio de um diagrama de casos de uso) e a modelagem (por meio de diagramas de atividades ou texto) de casos de uso de negócios são uma etapa potencialmente importante do desenvolvimento de um sistema. Antigamente, havia um tipo de profissional explicitamente encarregado de desenvolver a modelagem de negócios, o Analista de Organização e Métodos, ou analista de O&M. Atualmente, existem vários tipos de profissionais envolvidos com a modelagem de negócios, o analista de processos de negócios, o designer de processos de negócios, o designer de negócios, etc. Existem muitas variações em como proceder a uma modelagem de negócios. Pode ser que o negócio já envolva algum tipo de sistema computacional, ou pode ser que o negócio ainda não envolva um sistema computacional. De qualquer forma, é necessário conhecer os processos de negócio para que um sistema computacional possa ser proposto para dar suporte a esse processo de negócios. Muitas vezes, caso o negócio já envolva um sistema computacional, pode ser necessário repensar os processos de negócio de forma que o uso dos recursos computacionais possam ser utilizados de modo mais produtivo para o negócio.

Em casos em que o sistema sendo desenvolvido não se destina a dar suporte a um processo de negócios, como por exemplo um sistema embutido de controle, um sistema geral de produtividade ou um jogo de computador, essa etapa de modelagem de negócios pode ser dispensada, entretanto.

17. Especificação de Requisitos

A fase de especificação de requisitos é composta por diversas atividades, algumas desempenhadas de maneira individual e outras desempenhadas em grupo, conforme mostrado a seguir:

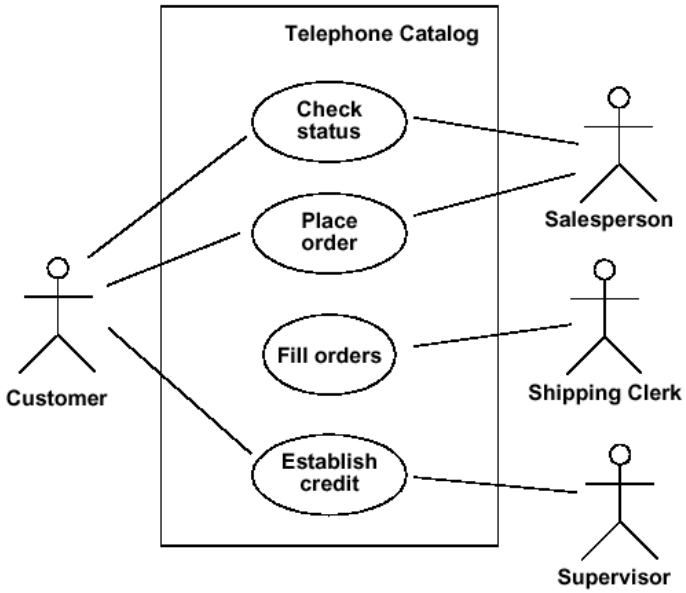


A primeira dessas atividades é a Geração do Diagrama de Casos de Uso inicial

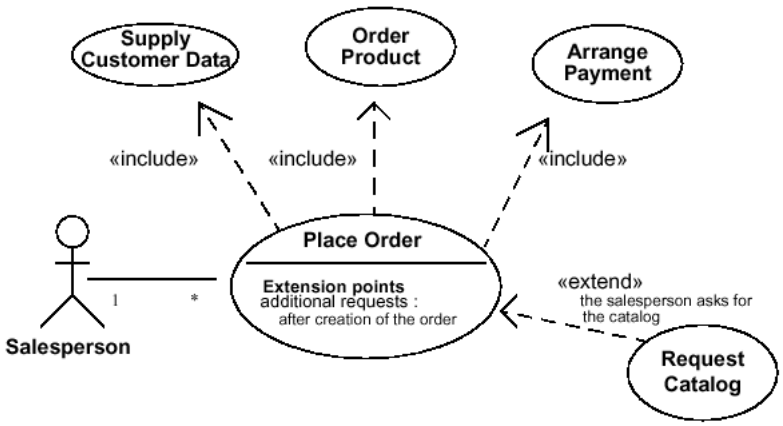
17.1 Gerar Diagrama de Casos de Uso

O objetivo dessa atividade é iniciar o processo de especificação dos requisitos, desenvolvendo cenários genéricos descrevendo a interação entre o(s) usuário(s) e o sistema. Nesta atividade, explora-se a descoberta de diferentes possíveis casos de uso. Estes casos de uso devem envolver todos os tipos de interações desejadas entre o sistema e os usuários. O resultado final desta atividade é a criação de um outline do diagrama de casos de uso.

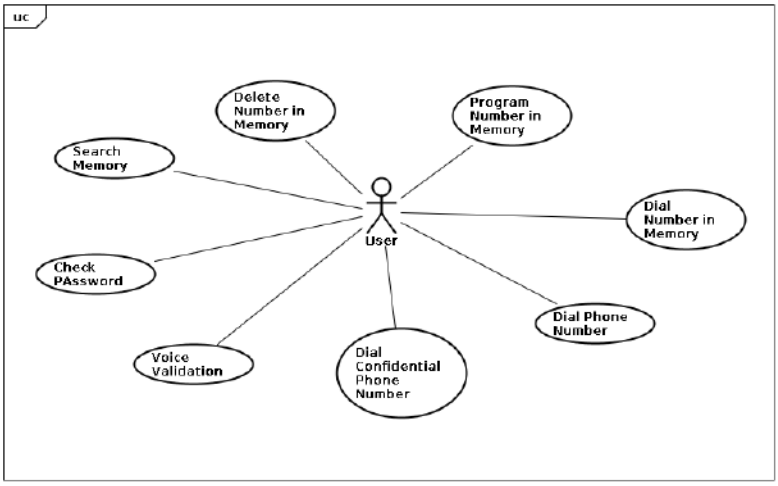
Casos de uso são abstrações de pequenas histórias narrativas envolvendo a interação entre um ou mais usuários (chamados de atores) e o sistema. A idéia é que estes casos de uso representem, por meio dessas pequenas histórias, as funcionalidades de um sistema. Os diagramas de casos de uso mostram atores e casos de uso juntos com seus relacionamentos. Um exemplo de um diagrama de casos de uso pode ser visto na figura a seguir:



Casos de uso podem ter pontos de extensão, ou seja, referências a uma localização dentro de um caso de uso onde sequências de ações de outros casos de uso podem ser inseridas. Cada ponto de extensão tem um único nome dentro de um caso de uso. Um exemplo do uso de casos de uso é mostrado na figura a seguir:



Nessa primeira atividade da especificação dos requisitos, o objetivo é levantarmos o maior número possível de casos de uso e os atores que dele participam. O diagrama criado aqui será somente um outline do modelo final. Esse diagrama será refinado em atividades posteriores. Um exemplo de um diagrama de casos de uso para um sistema de telefonia celular é apresentado na figura a seguir:



Observe que nesta atividade, o mais importante é evidenciarmos o maior número possível de casos de uso, sem maiores preocupações quanto ao possível relacionamento entre eles. Esse relacionamento será analisado posteriormente quando da estruturação do modelo de casos de uso.

Nesta atividade, cada participante da equipe deve tentar desenvolver um diagrama de casos de uso. Neste ponto, os casos de uso não precisam ainda estar estruturados. O mais importante é tentar identificar o maior número de casos de usos

17.2 Consolidar Diagrama de Casos de Uso

A etapa anterior visava uma prospecção de diferentes casos de uso, que deve ser executada individualmente por cada membro da equipe participante. Nesta etapa, realizada em grupo, as contribuições individuais de cada membro da equipe devem ser considerada, de forma a resultar um modelo único, incorporando as contribuições de todos os membros da equipe. Durante esta etapa, deve-se também uniformizar os nomes utilizados para os casos de uso, que devem, preferencialmente, reforçar o aspecto de que casos de uso transcorrem no tempo, e não são eventos individuais. Assim, deve-se preferir o nome "Verificação de Password", ao invés de "Verifica Password", "Edição do Texto", ao invés de "Edita Texto", e assim por diante. O resultado desta etapa é um diagrama de casos de uso consolidado (entretanto, ainda não estruturado).

17.3 Priorização dos Casos de Uso

O objetivo desta atividade é coletar subsídios para a priorização dos casos de uso, determinando quais deles devem ser desenvolvidos (i.e. analisados, desenhados, implementados, etc.) nas primeiras iterações e quais podem ser desenvolvidos nas iterações posteriores.

Os resultados são então capturados na visão arquitetural do modelo de casos de uso. Esta visão é considerada pelo gerente de projeto e utilizada para o planejamento do que deve ser desenvolvido na iteração corrente. Este planejamento leva em consideração também aspectos não-técnicos, tais como aspectos políticos ou estratégicos. A visão arquitetural deve destacar os casos de uso que descrevem funcionalidades críticas ou importantes (dentre outras, a inicialização do sistema e o encerramento do sistema, sem as quais um protótipo do sistema não poderia ser desenvolvido).

Assim, a escolha dos casos de uso que serão implementados na iteração corrente deve incluir

aqueles casos de uso que são mais vitais para a aplicação em questão. Casos de uso referentes a características marginais ou adicionais devem ser postergados para iterações futuras.

Em nosso caso, iremos adotar uma escala de prioridade que vai de 1 a 10, onde 1 corresponde a um caso de uso de baixa prioridade e 10 será um caso de uso de alta prioridade. Atribuiremos portanto uma prioridade a cada um dos casos de uso levantados na atividade anterior, e a seguir definiremos quais serão os casos de uso a serem implementados na corrente iteração, baseados nas prioridades atribuídas.

Como resultado dessa etapa, deve-se atribuir a cada membro da equipe de 1 a 2 casos de uso, de acordo com as prioridades levantadas.

17.4 Detalhamento dos Casos de Uso em Diagramas de Atividades

Nesta atividade, o objetivo é descrever de maneira detalhada os casos de uso selecionados na etapa anterior, referenciando de maneira minuciosa o fluxo de eventos entre atores e o sistema, incluindo-se como o caso de uso começa, termina e quais as atividades realizadas tanto pelos atores como pelo sistema.

A descrição de casos de uso pode ser realizada, em princípio, por meio de diferentes tipos de artefatos de software:

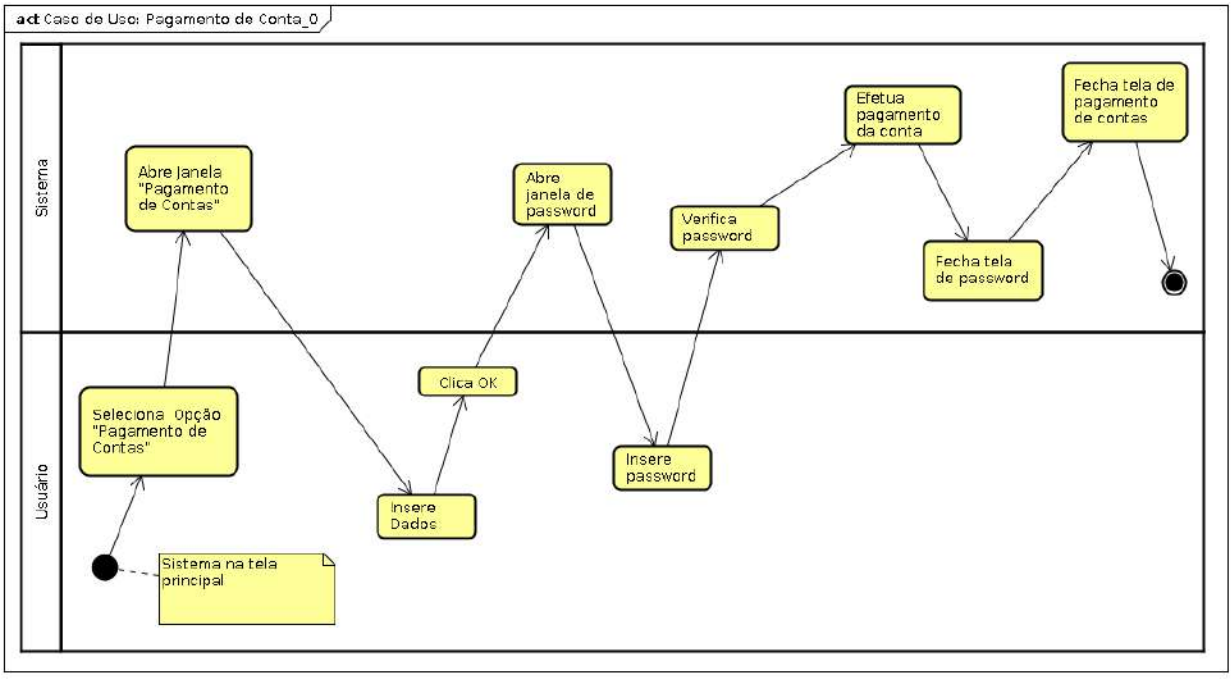
- Texto (sumarizada ou detalhada)
- Diagrama de Sequência
- Diagrama de Atividades

A escolha do artefato ideal depende do grau de complexidade do caso de uso. Quando o caso de uso é detalhado por meio de texto, diversos fluxos de atividade precisam ser descritos. O primeiro deles, chamado de fluxo principal, considera um cenário prototípico em que a funcionalidade desejada é implementada. Os demais fluxos, chamados de fluxos alternativos, consideram cenários alternativos, eventualmente onde o usuário adota outras opções (quando elas são possíveis) ou eventualmente cenários onde ocorrem desistências por parte do usuário, que solicita o cancelamento de decisões tomadas anteriormente.

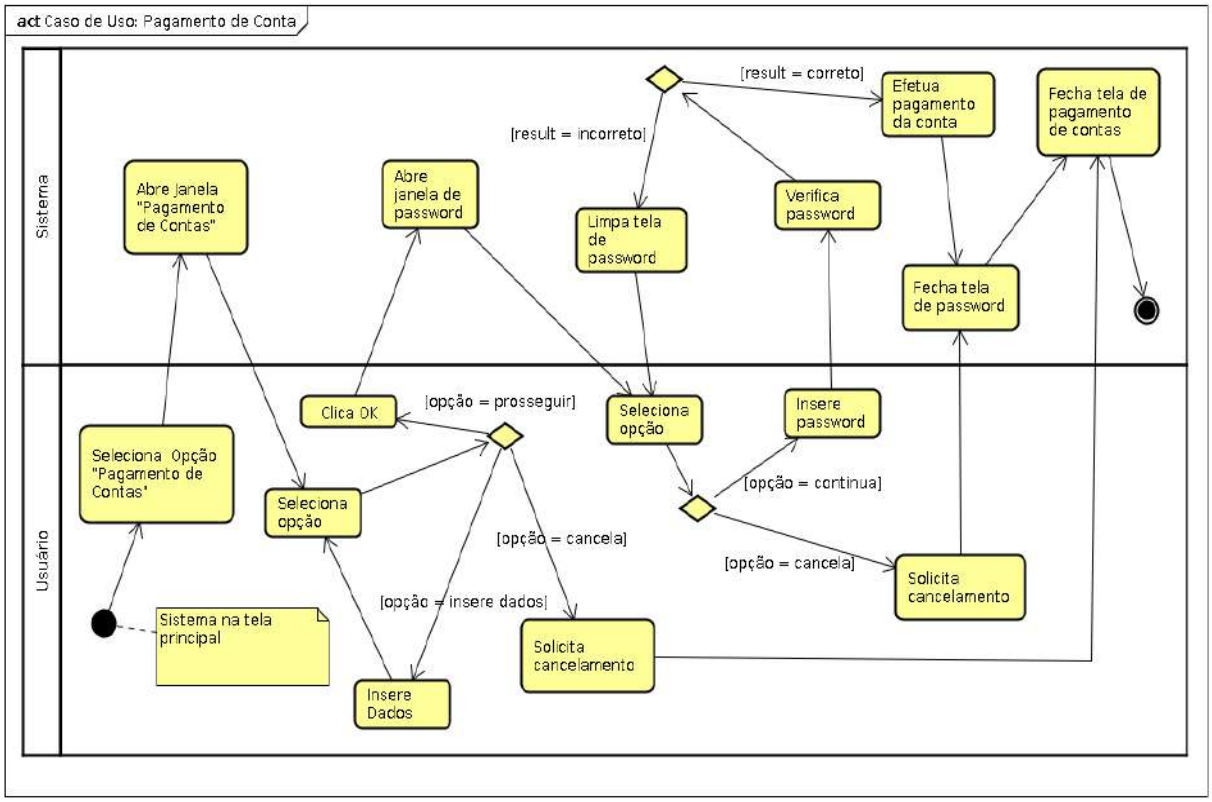
Apesar do detalhamento de casos de uso por meio de texto simples ser bastante popular na comunidade de desenvolvimento de software, uma maneira bastante efetiva de detalhar um caso de uso é por meio de um diagrama de atividades. Durante o detalhamento de um caso de uso, o número de fluxos alternativos pode ser grande e, no caso do detalhamento via texto, existe grande chance de que algum fluxo alternativo seja esquecido de ser especificado, o que acabará resultando em uma especificação deficiente. Uma das vantagens de se detalhar o caso de uso por meio de um diagrama de atividades é que todos os fluxos alternativos podem ser construídos sobre um mesmo diagrama, facilitando a verificação do mesmo quanto a fluxos potencialmente esquecidos. Utilizando um diagrama de atividades para especificar um caso de uso, utilizamos dois *swimlanes*, um para o usuário (ou um para cada usuário, quando o caso de uso possui mais de um usuário) e outro para o sistema. Os diagramas de atividades permitem uma descrição bem precisa das diversas alternativas que um caso de uso pode exibir. O único cuidado que devemos ter é o de não confundir o diagrama de atividades com um fluxograma. O diagrama de atividades deve descrever, de maneira clara e inequívoca, quais as ações realizadas pelo usuário e quais as ações realizadas pelo sistema. Um erro comum quando se constrói um diagrama de atividades é não atentar para esse ponto.

Como devemos fazer o detalhamento de um caso de uso por meio de um diagrama de atividades ? Inicialmente devemos incluir o estado inicial, normalmente acompanhado de uma pré-condição, colocada por meio de uma nota. Essa pré-condição dirá como e quando o caso de uso começa. Em seguida, devemos construir uma sequência básica que descreve o caso de uso, onde o usuário executa certas ações e o sistema responde com ações correspondentes. Na construção dessa sequência básica, chamada de fluxo principal, podemos supor que tudo dá certo e que o sistema reage da maneira correta, de acordo com a funcionalidade que se deseja obter. Por fim, deve-se definir como e quando o caso de uso termina. Essa definição normalmente se dá na forma de pós-condições, também indicadas por meio de uma nota. As pós-condições, entretanto, não são neessárias, podendo ser omitidas em alguns casos onde as mesmas são óbvias.

Um exemplo de um fluxo principal, para um caso de uso de nome "Pagamento de Conta" é descrito na figura a seguir. Neste exemplo, por ser um caso de uso curto, optamos por desenvolver as *swimlanes* horizontalmente. Na maioria das situações, entretanto o uso de *swimlanes* verticais se mostra mais adequado a descrições mais longas. Observe que na descrição do fluxo principal, detalhamos um protocolo de ações em que o usuário faz alguma coisa (ou uma sequência de coisas) e o sistema reage executando também alguma ação ou sequência de ações. É importante lembrarmos sempre de especificar como o sistema deve reagir. É muito comum nos esquecermos de dizer que o sistema precisa criar ou fechar uma janela, no caso de uma aplicação tradicional, ou exibir uma tela em um browser, no caso de uma aplicação web.



Após a definição desse fluxo básico, é necessário incluir em seguida os fluxos alternativos. Esses fluxos alternativos existem porque as vezes o usuário pode ter diferentes opções, ou nem sempre tudo dá certo durante o transcorrer de um caso de uso. O password que foi digitado pode estar incorreto, o usuário pode querer desistir de continuar com o caso de uso e desejar encerrá-lo prematuramente, ou mesmo o caso de uso pode prever múltiplas decisões em alguns pontos do mesmo. Desta maneira, é necessário adaptar o fluxo principal para incorporar os fluxos alternativos. Esses fluxos alternativos são inseridos por meio de nós de decisão, precedidos de ações do tipo "seleciona opção". A opção selecionada é então testada por meio de condições de guarda nos nós de decisão, e os diferentes tipos de caminhos alternativos podem ser inseridos



Ao detalhar um caso de uso por meio de diagramas de atividades, podemos visualizar em um mesmo diagrama dessa forma, não tão somente o fluxo principal, mas todos os possíveis fluxos alternativos. Essa visualização, ao contrário da descrição textual de casos de uso, permite que avaliemos se de fato todos os fluxos alternativos desejados foram inseridos, evitando que o sistema venha a apresentar, mais a frente, situações em que o usuário se veja em dificuldade em virtude de uma especificação mal-feita. Deve-seprestar especial atenção ao fato de que, a cada retorno do sistema, deve-se dar a opção do usuário cancelar a execução do caso de uso e finalizá-lo prematuramente.

Devemos atentar, também, que antes de um nó de decisão, precisa haver uma ação que nos leve ao cálculo de uma variável de decisão. Essa variável de decisão será então testada no nó de decisão, favorecendo o desvio dentre múltiplos caminhos. Da mesma forma, após um nó de decisão, sempre precisará haver uma ação correspondente que ratifique a decisão. Assim, se decidimos prosseguir, devemos efetuar uma ação (clicar no botão OK, por exemplo), que ratifica essa decisão. É muito comum desenvolvedores iniciantes se esquecerem desses detalhes. Verifique no diagrama de exemplo anterior, que a cada nó de decisão sempre existe uma ação anterior que calcula uma variável de decisão. Da mesma forma, sempre existe um teste dessa variável nos arcos que saem do nó de decisão, e sempre existe uma ação posterior que ratifica essa decisão.

Em nosso caso, como devemos proceder ? A cada um dos casos de uso atribuídos ao membro da equipe, o mesmo deve trabalhá-lo individualmente, prestando atenção para incluir todos os fluxos alternativos necessários ao bom desempenho da funcionalidade ensejada. Esta atividade deve ser executada, entretanto, de maneira concomitante com a atividade a seguir: a prototipação da interface com o usuário.

17.5 Prototipação da Interface com o Usuário

Nesta atividade, o principal objetivo é construir um protótipo das interfaces com o usuário. Essas interfaces não serão as interfaces definitivas, mas devem conter as informações necessárias para a efetivação dos casos de usos descritos na etapa anterior. Assim, critérios estéticos não são importantes nessa atividade. O importante é a identificação dos elementos de comunicação necessários à interação com o sistema.

Esta atividade deve ser realizada de maneira concomitante com a atividade anterior. Para que o usuário se comunique com o sistema, e este, por sua vez, se comunique com o usuário, é necessário que haja uma interface. Dessa forma, à medida que os diagramas de atividades da atividade anterior são construídos, a cada vez que o sistema reage a uma ação do usuário, essa ação deve ser descrita em termos de uma interface. Essa interface é a que é criada nesta etapa. O resultado final que se espera nesta atividade é uma lista protótipos de interface, devidamente nomeadas, que devem ser referenciadas na descrição dos diagramas de atividades.

Como já ressaltamos, essas interfaces não serão as interfaces definitivas, mas servirão de inspiração posteriormente na fase de design quando elementos não funcionais tais como cores, tamanhos, fontes, ocupação do espaço, etc serão considerados.

17.6 Estruturação do Modelo de Casos de Uso

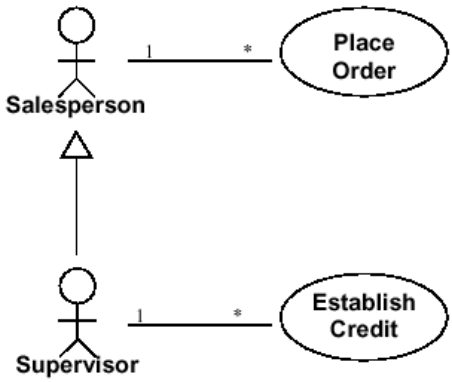
Nesta atividade, o objetivo é reestruturar os elementos do modelo de casos de uso capturados anteriormente (que podem ter sido capturados até de maneira independente entre si) e gerar um modelo que seja homogêneo, consistente e simples de ser interpretado. Durante o detalhamento dos casos de uso, pode ser que algum caso de uso em particular tenha se tornado demasiadamente complexo ou detalhado, quando então se torna útil desmembrá-lo em mais de um casos de uso. Da mesma forma, pode ser que se tenha percebido que muitos casos de uso possuem uma estrutura comum, que se repete diversas vezes nos diagramas de atividades desenvolvidos. Dessa forma, nesta etapa deve-se fazer um refactoring dos casos de uso, de tal forma que os mesmos sejam repensados e devidamente estruturados, para que a etapa de especificação de requisitos termine definitivamente. Antes de procedermos com a descrição desta atividade em maiores detalhes, vamos ver algumas características dos tipos de relacionamentos que podem surgir em versões estruturadas de diagramas de casos de uso.

Basicamente, 4 tipos de relacionamentos podem ser indicados em diagramas de casos de uso:

- Associações
- Extend
- Include
- Generalização

As **associações** denotam a participação de atores em um caso de uso. É o único tipo de relacionamento entre um ator e um caso de uso. Um relacionamento do tipo **extend** é uma relação entre um caso de uso A para um caso de uso B que indica que uma instância do caso de uso B pode ser aumentada (sujeita a condições específicas de extensão) pelo comportamento especificado por A. Esse comportamento é inserido conforme especificado por um ponto de extensão em B. Um relacionamento do tipo **include** é uma relação entre um caso de uso A para um caso de uso B que indica que uma instância do caso de uso A contém o comportamento especificado por B. Esse

comportamento é incluído na localização indicada em A por um ponto de extensão. Um relacionamento do tipo **generalização** entre um caso de uso A e um caso de uso B indica que A é uma especialização de B. Normalmente, essa generalização implica em que B é uma descrição mais abstrata de uma situação, e A é uma descrição mais detalhada. Além dos casos de uso, é possível utilizarmos generalizações para indicarmos o relacionamento entre dois atores. Um exemplo deste tipo é apresentado na figura a seguir:



Observe que nesse caso, o ator Supervisor é um um tipo de Salesperson, ou seja, uma especialização de um Salesperson. Observe ainda pelo exemplo que as associações entre atores e casos de uso podem também conter uma cardinalidade.

As associações entre os atores e os casos de uso podem ainda possuir uma navegabilidade (seta entrando ou saindo do caso de uso). Essa navigabilidade indica quem inicia o caso de uso. Caso a seta seja do ator para o caso de uso, é o ator quem inicia a interação. Caso seja do caso de uso para o ator, é o sistema quem inicia a interação. Veja o exemplo na figura a seguir:

Neste exemplo, que mostra o caso de uso Participação em Conferência Eletrônica, é o professor quem inicia as atividades. O aluno passa a interagir posteriormente, a partir da iniciativa do sistema.

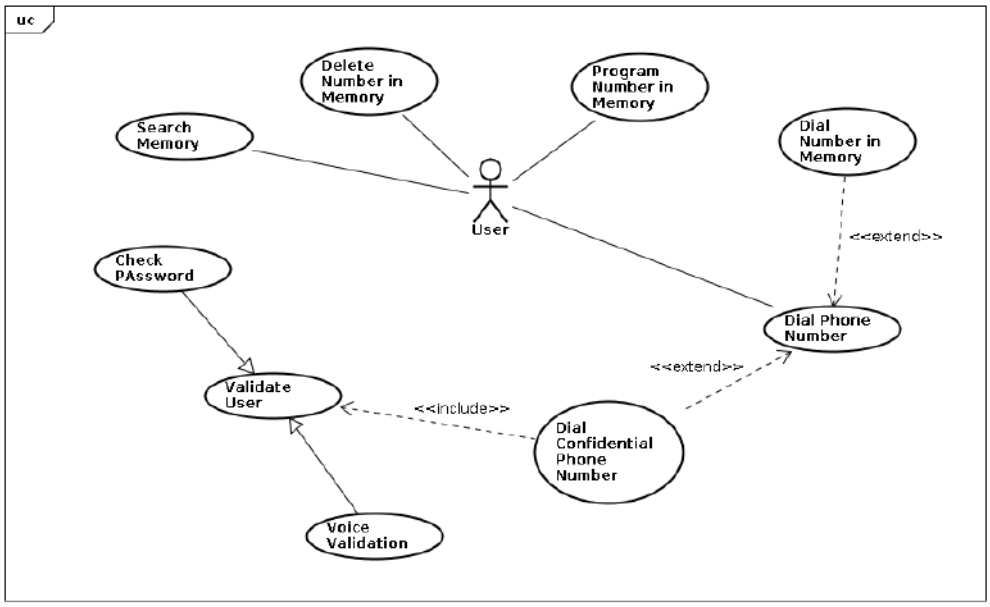
Muito bem ! Entendidos estes relacionamentos, vamos às atividades desta fase. O primeiro passo é tentar identificar descrições compartilhadas de funcionalidades entre os casos de uso levantados anteriormente. Como isso ocorre ? Durante o levantamento dos casos de uso, podem haver atividades ou partes de atividades que são semelhantes em diversos casos de uso. Pode-se detectar uma situação como essa quando observamos que certas partes dos diagramas de atividades se repetem em mais de um casos de uso. De forma a reduzir a redundância, os casos de uso podem ser então reestruturados para tornar o modelo como um todo mais enxuto. Assim, esses trechos de casos de uso podem se tornar casos de uso independentes, que são reutilizados no modelo por meio da relação de generalização. Essas modificações no diagrama de casos de uso deve ser seguida de modificações equivalentes nos diagramas de atividades, para refletir essas alterações, quando for o caso.

O segundo passo é tentar identificar descrições adicionais ou opcionais de funcionalidade. Sabemos que o mecanismo de extensão permite a inserção de adições ao comportamento básico de casos de uso. Como vimos, esse relacionamento inclui as condições para a extensão e o ponto de extensão, onde o caso de uso deve ser inserido. O que faremos então é tentar identificar situações desse tipo,

alterando o diagrama de casos de uso de modo a incluir relacionamentos de extensão. Os diagramas de atividades associados devem ser retrabalhados também, de forma a refletir as mudanças implementadas.

O terceiro passo é tentar identificar descrições repetidas de funcionalidade. Sabemos que o relacionamento de inclusão permite a inserção incondicional e explícita do comportamento de um caso de uso em outros casos de uso. Tentaremos então verificar se situações desse tipo ocorrem, e nesse caso procederemos à inserção de relacionamentos do tipo include em nosso diagrama. Alterações equivalentes devem ser implementadas nos diagramas de atividades referenciados

Nesse ponto, talvez vocês estejam se perguntando … Ahn ? Qual a diferença então entre o relacionamento de generalização e o relacionamento de inclusão. Temos que ter muito cuidado pois esses dois tipos de relacionamentos são muito semelhantes. A grande diferença que existe entre esses dois é que no relacionamento de generalização, o caso de uso que é generalizado é um caso de uso mais abstrato que, apesar de semelhante, será diferente em cada uma de suas especializações. Um bom exemplo é o exemplo mostrado na figura abaixo, fruto da estruturação do modelo de casos de uso que apresentamos anteriormente.



Observe que anteriormente tínhamos dois casos de uso, o caso de uso *CheckPassword* e o *VoiceValidation*, que apresentavam um compartilhamento de funcionalidades. Ambos serviam para efetuar uma validação do usuário. Uma maneira de refinar nosso modelo foi criar um novo caso de uso abstrato chamado *ValidateUser*, que passou então a ser uma abstração, tanto de *CheckPassword* como de *VoiceValidation*. Para compreendermos a diferença entre o uso de uma generalização e de uma inclusão, observe que a descrição de uma checagem de password não tem nada a ver, em princípio com uma validação vocal. Em termos concretos, eles são casos de uso completamente diferentes. Entretanto, quando abstraímos o que ocorre em cada um desses casos de uso, vemos que ambos nada mais fazem do que validar o usuário. Por isso, usamos uma relação de generalização. O relacionamento do tipo inclusão, ao contrário, inclui totalmente o caso de uso como um sub-trecho de um caso de uso mais complexo. Assim, devemos ter cuidado durante esse refinamento. Em alguns casos, será necessário fazermos abstrações e incluirmos novos casos de uso mais abstratos. Em outros, vamos tentar verificar a mera repetição de situações e utilizaremos inclusões. Observe que no diagrama do exemplo, verificamos que a validação do usuário ocorre somente quando ocorre

uma discagem de número confidencial. Por isso, colocamos o caso de uso *ValidateUser* como uma inclusão de *DialConfidentialPhoneNumber*. Por outro lado, tanto *DialConfidentialPhoneNumber* quanto *DialNumberInMemory* extendem a descrição de *DialPhoneNumber*. Por isso, colocamos ambos como extensões de *DialPhoneNumber*.

Repare que após chegarmos ao diagrama de casos de uso final, pode ser necessário que modifiquemos a descrição dos casos de uso feitas anteriormente, como por exemplo a inclusão de novos casos de uso (vide a inclusão de *ValidateUser* no exemplo). Essas modificações devem então ser implementadas, para que a fase de especificação de requisitos se dê por encerrada.

18. A Fase de Análise no Processo Unificado

A fase de análise, também chamada muitas vezes de *análise dos requisitos* (não confundir com *especificação dos requisitos* nem com a *análise de domínio*), corresponde a uma fase onde faremos o aprofundamento das investigações acerca das especificações dos requisitos. Assim, procederemos ao detalhamento e refinamento dessas especificações.

O grande objetivo desta fase é a obtenção de uma melhor compreensão dos requisitos, mantendo uma descrição dos requisitos que seja compreensível e auxilie no posterior design do sistema. Podemos dizer que o que se faz na fase de análise é uma espécie de tradução das especificações dos requisitos, que se encontram na *linguagem do cliente*, para uma representação que use uma *linguagem do desenvolvedor*. Assim, passamos de um **modelo de casos de uso** para um **modelo de análise**. O modelo de casos de uso corresponde a uma visão do sistema sob a óptica do cliente - ou seja - viesado para o que o cliente deseja que seja implementado. O modelo de análise corresponde a uma visão do mesmo sistema já sob um ângulo diferente, ou seja, a descoberta e detalhamento do que o usuário necessita e sua descrição, preparando o terreno para a fase de design que virá no futuro, onde serão assumidos compromissos com relação às tecnologias que serão utilizadas para sua implementação.

Veremos que um dos principais desafios a serem vencidos na fase de análise é se manter um nível abstrato de investigação, sem entrar em questões de design do sistema.

O modelo de análise permite o refinamento do modelo de casos de uso por meio da especificação dos detalhes internos do sistema a ser construído. Dessa forma, provê um melhor poder de expressão e um formalismo mais refinado, principalmente para a descrição da dinâmica do sistema. Entretanto, apesar desse maior detalhamento, o modelo de análise deve usar somente de abstrações, evitando a solução definitiva de alguns tipos de problemas, que devem ser postergadas para a fase de design.

Veremos que algumas estruturas desenvolvidas na fase de análise nem sempre serão preservadas durante o design. Essa é uma das razões para que se mantenha estas estruturas de forma abstrata durante a análise. Estruturas muito amarradas podem se tornar um empecilho durante o design tendo que ser descartadas. Estruturas mais abstratas podem ter seu escopo negociado e compromissado quanto a flexibilidade quando se passar da análise para o design.

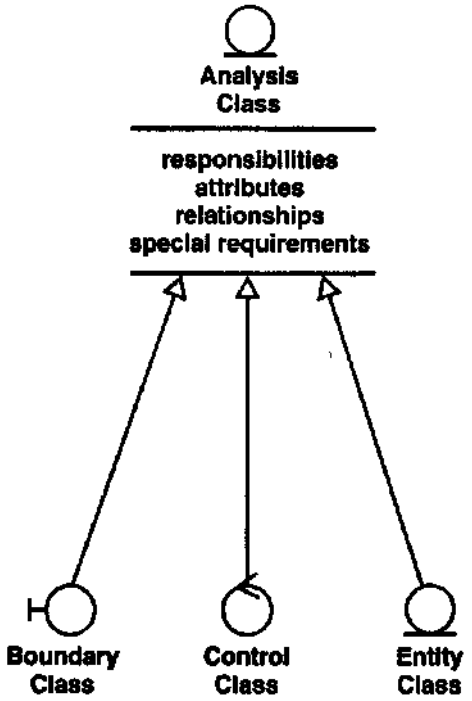
De um modo geral, podemos comparar as fases de análise e design no seguinte sentido. A fase de análise busca responder a seguinte pergunta: **o quê ?** O que o sistema deve fazer … o que o sistema deve ser … o que o sistema realiza. A fase de design, ao contrário, buscará responder a seguinte pergunta: **como ?** Como o sistema fará … como o sistema será … como o sistema realiza. Assim, durante a fase de análise, a compreensão de quais são as responsabilidades do sistema é mais importante do que como essas responsabilidades serão futuramente implementadas.

Durante a fase de análise, os principais artefatos de software desenvolvidos serão a **Arquitetura de Análise**, decomposta em diversos diagramas de classes que devem conter as **classes e pacotes de análise**. Além da arquitetura, teremos as **realizações de casos de uso** que serão modeladas por meio de **diagramas de interação** e dos **contratos** regulando o significado das mensagens nesses diagramas. Todos esses artefatos se integram, compondo a chamada **visão do modelo de análise**. Essa visão representa um ângulo por meio do qual podemos descrever o sistema, em termos de suas responsabilidades, sem nos comprometermos com sua implementação. Durante a fase de design,

uma possível implementação será escolhida e planejada.

Classes de análise representam abstrações de uma ou mais classes ou subsistemas que irão aparecer no design do sistema. Podemos imaginá-las como grânulos grossos ou de mais alto nível por meio do qual estaremos a descrever a funcionalidade do sistema que estamos desenvolvendo. Dentre outras características, veremos que as classes de análise focalizam nos requisitos funcionais do sistema, postergando os requisitos não-funcionais (tais como os detalhes da interface, etc …) para a fase de design. Assim, torna-se mais importante na fase de análise a definição de interfaces em termos de responsabilidades, que poderão futuramente sofrer diferentes implementações. Assim, deve-se preocupar com a descoberta de atributos de alto nível e relacionamentos conceituais.

Uma das maneiras de se manter esse nível abstrato durante a análise é por meio da partição de nossas classes em três estereótipos: *boundary*, *control* e *entity*, conforme a figura abaixo:



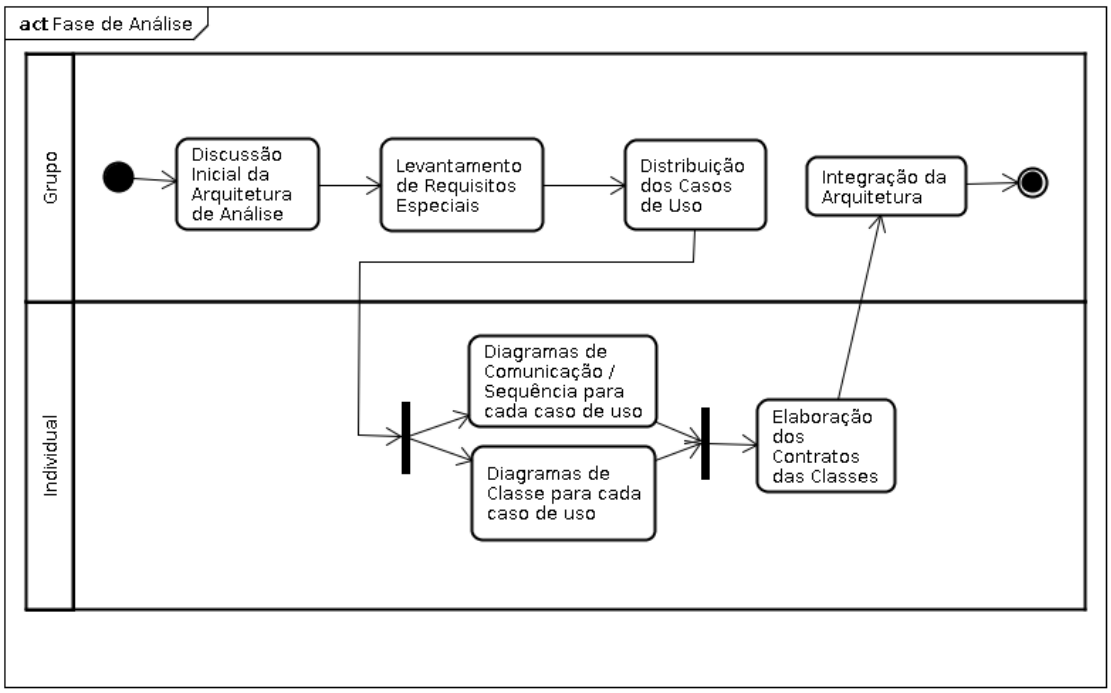
Classes do tipo *boundary* são classes utilizadas para modelar a interação entre o sistema e os atores. Essa interação envolve o recebimento e a apresentação de informações. Podemos então utilizar classes do tipo *boundary* para modelar janelas gráficas ou outros tipos de interface em que os atores enviem suas mensagens na forma de eventos, e que o sistema possa mostrar seu desempenho na forma de representações visuais ou sonoras.

Classes do tipo *control* são classes que representam a coordenação, sequenciamento, transações e controle de outros objetos. Podem ainda realizar derivações e cálculos com informações próprias ou oriundas de outros objetos. Assim, as classes do tipo *control* são classes que fazem alguma coisa, que realizam alguma atividade. Essa atividade pode envolver outros objetos de outras classes.

Por fim, classes do tipo *entity* são classes que modelam informações (dados) de longa duração, frequentemente dados que se desejam armazenar de maneira persistente (embora isso não seja necessário). Assim, classes do tipo *entity* são classes que comporão a estrutura de dados interna do sistema, que deverão ser manipulados por objetos de classes do tipo *control*, interagindo com o usuário por meio de classes do tipo *boundary*.

Por meio da criação de classes do tipo *boundary*, *control* e *entity*, somos capazes de descrever o sistema em um nível bem abstrato, como o necessário para os objetivos da fase de análise.

A fase de análise, em si, corresponde à sequência de atividades conforme pode ser visto na figura a seguir:



18.1 Definição Inicial da Arquitetura de Análise

O objetivo dessa fase é gerar um outline da arquitetura de análise, por meio da identificação de pacotes e classes que possam servir de pináculo para a posterior análise dos casos de uso. É certo que a arquitetura final de análise somente poderá ser concluída após a análise dos casos de uso. Entretanto, nessa etapa visa-se criar um ponto de partida unificado por meio do qual todos os diferentes desenvolvedores trabalhando individualmente sobre os casos de uso possam adotar um conjunto inicial de classes padronizado, diminuindo o esforço posterior na etapa de integração da arquitetura.

O resultado final que se espera dessa fase é um esboço inicial do diagrama de classes estruturado que constitui a **Arquitetura de Análise**. Um diagrama de classes estruturado é um conjunto de diagramas de classes associados mutuamente por uma relação de hierarquia, formando uma árvore. Em sistemas mais simples, esse diagrama será simplesmente um único diagrama de classes. Em sistemas mais complexos, a partir do diagrama raiz da árvore, muitas vezes chamado de diagrama de pacotes, associam-se diagramas subsequentes, um para cada pacote presente no diagrama raiz. Assim, sucessivamente, para cada pacote presente em um diagrama, associa-se um outro diagrama de classes, que poderá conter classes e/ou outros pacotes.

Inicia-se essa fase consultando-se os diagramas de atividade gerados na especificação, e para cada interface com o usuário identificada na especificação, gera-se uma classe do tipo *boundary*, que é inserida no diagrama de classes.

Após o levantamento de todas as classes do tipo *boundary*, deve-se acrescentar classes do tipo *control* controlando essas interfaces. Uma única classe do tipo *control* pode controlar mais de uma classe do tipo *boundary*. A escolha do número de classes do tipo *control* a serem inseridas é uma decisão dos desenvolvedores. Por fim, deve-se consultar o modelo conceitual de domínio, e dele extrair classes do tipo *entity* que se mostrem apropriadas aos tipos de dados necessários para a análise dos casos de uso. Não é necessário ser exaustivo nessa fase. Qualquer classe que não tenha uma utilidade óbvia não deve ser incluída.

À medida que o número de classes vai aumentando, pode ser conveniente re-organizá-las em pacotes. Pacotes permitem uma melhor organização do modelo de análise, dividindo-o em partes menores mais fáceis de serem gerenciadas. Essa divisão pode ser feita visando-se dividir o trabalho entre diferentes integrantes da equipe de desenvolvimento. Entretanto, deve-se fazer essa divisão alocando-se casos de usos com funcionalidades semelhantes em pacotes comuns, como por exemplo:

- casos de uso que suportam processos organizacionais comuns
- casos de uso que suportam um ator específico
- casos de uso relacionados por generalizações, extensões ou inclusões

Ao final dessa etapa, obtém-se uma versão inicial da arquitetura de análise, que servirá de ponto de partida para que a análise dos casos de uso possa ser iniciada.

18.2 Levantamento de Requisitos Especiais

Antes de proceder à análise de casos de uso, entretanto, é necessária ainda uma outra etapa preliminar. Nenhum projeto de software transcorre construindo-se o sistema todo da estaca zero. É bastante normal a reutilização de partes do sistema oriundas de sistemas anteriores, ou de componentes especialmente produzidos com a finalidade de serem reutilizados. Para uma série de requisitos do sistema, existem já soluções de prateleira prontas, que evitam o retrabalho em questões vitais da confiabilidade de um sistema. Requisitos especiais são aqueles que visam identificar a existência de partes do sistema que possam ser reutilizadas posteriormente no design. Dentre outros, requisitos especiais podem envolver requisitos de:

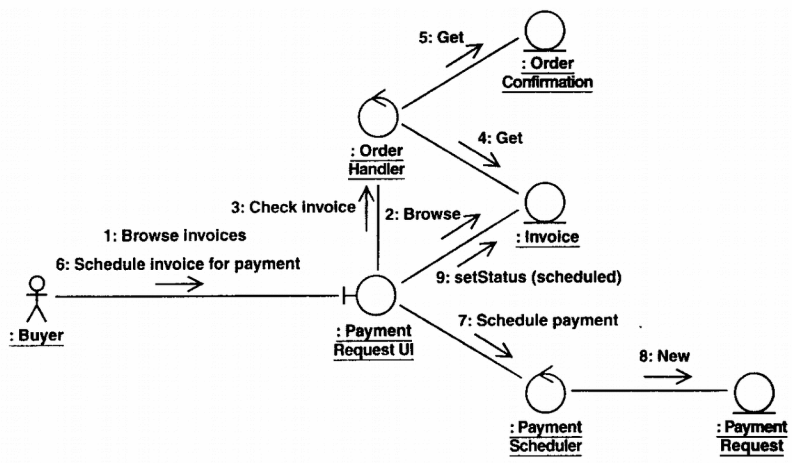
- persistência
- distribuição ou concorrência
- restrições de segurança
- tolerância a falhas
- gerenciamento de transações
- etc

Nessa fase, o grupo deve se reunir e deliberar sobre a existência de requisitos dessa natureza no projeto em desenvolvimento. Nesse momento, é suficiente a identificação de sua relevância para o sistema. Posteriormente, na fase de design, a equipe deve decidir se esses requisitos serão desenvolvidos pelo grupo ou se serão reutilizados de alguma maneira. Em alguns casos, os requisitos especiais só poderão ser encontrados durante a realização dos casos de uso. Ao final dessa fase, o resultado esperado é uma lista dos requisitos especiais que podem existir para o projeto.

18.3 Desenvolvimento dos Diagramas de Interação para cada Caso de Uso

O objetivo dessa atividade é implementar a assim chamada **Realização dos Casos de Uso**. Na fase de especificação, o sistema é visto sempre como uma unidade monolítica - o sistema. A grande mudança que se processa na fase da análise é exatamente a decomposição desse sistema em partes menores. Dessa forma, cada caso de uso detalhado por um diagrama de atividades deve agora ser realizado, desenvolvendo-se um diagrama de interação. No UML, existem basicamente dois tipos de diagramas de interação, os diagramas de comunicação (chamados também de diagramas de colaboração, no UML 1) e os diagramas de sequência. Esses dois diagramas, em princípio, são diagramas duais - um diagrama de comunicação pode ser re-escrito na forma de um diagrama de sequência e vice-versa. A escolha por um diagrama de comunicação ou um diagrama de sequência diz respeito somente ao número de objetos que devem interagir e à quantidade de mensagens que estes objetos devem trocar entre si. Caso haja um grande número de objetos, trocando entre si um número reduzido de mensagens, os diagramas de comunicação são mais adequados. Caso os mesmos objetos troquem entre si um grande número de mensagens, os diagramas de sequência mostram-se mais adequados.

Um exemplo de diagrama de comunicação é mostrado a seguir:



Observe algumas peculiaridades desse diagrama. Por exemplo, todos os nomes estão grifados, significando que se tratam de instâncias das classes e não as classes em si como no diagrama de classes. Como temos somente um objeto de cada classe não é necessário nomeá-los. Portanto, os nomes dos objetos são omitidos, e temos somente nomes do tipo **:NomeClasse.** Caso houvesse dois ou mais objetos de uma mesma classe, nomes poderiam ser acrescentados, resultando em algo como **Nome1:NomeClasse**, **Nome2:NomeClasse**.

Observe também que entre os objetos fluem mensagens que são numeradas e podem conter parâmetros. Neste ponto, somente indicamos as mensagens. Entretanto, devemos estar conscientes que em uma atividade subsequente, iremos especificar mais profundamente o significado destas mensagens por meio de contratos individuais para cada mensagem.

Essa atividade deve ser executada de maneira concomitante com a atividade a seguir, o desenvolvimento dos diagramas de classes para cada caso de uso. Muitas ferramentas CASE de edição de diagramas UML não permitem que um objeto de uma classe seja inserido em um diagrama de interação, se ele não estiver já definido em um diagrama de classes. Na prática, isso

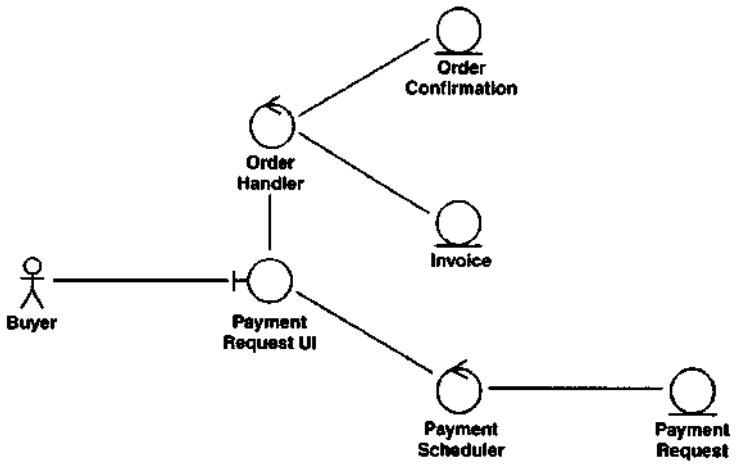
significa que o diagrama de interação precisa ser feito necessariamente em concomitância com o diagrama de classes.

18.4 Desenvolvimento de Diagramas de Classes para cada Caso de Uso

A realização de um caso de uso, além do diagrama de interação que o descreve, demanda também um diagrama de classes representando as classes que são utilizadas em um diagrama de interação. Esses diagramas de classes serão posteriormente integrados na arquitetura de análise do sistema.

Dessa forma, da mesma maneira que fizemos inicialmente no levantamento inicial da arquitetura, utilizaremos as idéias dos estereótipos *control*, *entity* e *boundary* para tentar delinear um conjunto de classes que dêem conta de realizar o caso de uso em questão. Deve-se partir das classes já existentes na arquitetura inicial de análise, e ir se inserindo novas classes à medida em que as mesmas mostrem-se necessárias para a realização do caso de uso.

Um exemplo de um diagrama de classes de análise pode ser visto na figura a seguir:



Há entretanto um desafio escondido aqui. Como os casos de uso são desenvolvidos individualmente, pode haver redundância na geração de classes entre diferentes desenvolvedores. Essa redundância precisará ser solucionada posteriormente, durante a fase de integração da arquitetura, quando os diagramas de classes desenvolvidos nessa fase devem ser consolidados em uma única arquitetura que integre todos os casos de uso do sistema.

18.5 Elaboração dos Contratos para as Classes

O objetivo desta atividade compreende a identificação e formalização das responsabilidades de uma classe de análise, baseadas em seu papel na realização de um caso de uso.

A identificação de responsabilidades pode ser perpetrada por meio da descoberta dos papéis que as diferentes classes assumem em diferentes realizações de casos de uso. Assim, as responsabilidades estão associadas às mensagens que uma classe pode receber.

A descrição das responsabilidades será formalizada por meio da elaboração dos chamados

contratos. Contratos descrevem as responsabilidades de uma determinada classe em termos de quais mudanças no estado do objeto são realizadas quando este recebe mensagens (ou invocações de métodos). Esses contratos devem descrever **O QUÊ** o objeto deve fazer, sem explicar **COMO** ele o faz. Para cada mensagem aparecendo em um diagrama de interação, deve haver um contrato correspondente.

Os contratos devem estar organizados por classe. Dessa forma, mesmo que uma mesma mensagem apareça em mais de uma classe, deve haver um contrato diferente para cada classe onde a mensagem aparece.

Um contrato está dividido em diversas seções. Cada seção provê informações sobre uma parte específica do contrato. Nem todas as seções são obrigatórias, devendo aparecer somente quando necessário. Um contrato deve ter a seguinte estrutura:

Nome: nome da operação e eventualmente seus parâmetros
Responsabilidades: uma descrição informal das responsabilidades que a operação deve implementar
Tipo:conceitual, classe de software ou interface
Referências Cruzadas: outras classes que utilizam o mesmo contrato, etc.
Notas: informações adicionais, algoritmos, etc.
Exceções: casos excepcionais
Saídas Secundárias: saídas não relacionadas à interface com o usuário, e.g. mensagens de erros, logs, etc.
Pré-Condições: condições referentes ao estado do sistema antes da execução da operação
Pós-Condições: estado do sistema após a conclusão da operação

Para fazer um contrato, devemos inicialmente identificar as operações a serem reguladas por contratos, a partir dos diagramas de colaboração. Para cada operação, devemos implementar um contrato. Começamos pela seção "Responsabilidades", descrevendo informalmente o propósito da operação. Em seguida, passa-se à seção de "Pós-Condições", descrevendo declarativamente as mudanças no estado do objeto que devem ocorrer em virtude da operação. Para descrever as pós-condições, devemos atentar para os seguintes pontos:

- Criação e destruição de instâncias de outros objetos
- Modificação de Atributos
- Formação ou destruicão de associações

As pós-condições são a parte mais importante de um contrato. Por isso, devemos ter um cuidado todo especial em sua elaboração. Particularmente, devemos ter muita atenção para não confundirmos as pós-condições com as saídas secundárias. Outro ponto importante é lembrarmos que as pós-condições não são ações a serem executadas durante a operação, mas declarações sobre o estado do objeto após a conclusão da operação contratada. Algo que não podemos também esquecer é que as pós-condições dizem respeito ao modelo de análise. Portanto, as classes dos objetos criados devem existir no diagrama de classes, do mesmo modo que as associações.

Uma pergunta que muitas vezes aparece é a seguinte: quão completas devem ser as pós-condições. Devemos ter consciência de que um conjunto de pós-condições completo e acurado é muito improvável - e nem mesmo necessário - durante a fase de análise. Os detalhes maiores poderão ser

descobertos durante a fase de design.

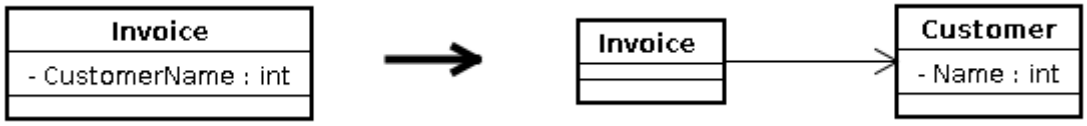
Agora voltemo-nos para as pré-condições. As pré-condições definem condições sobre o estado do sistema que são necessárias para que a operação possa ser realizada. Por exemplo: coisas que se deve testar a certo ponto da operação, ou coisas que não serão testadas, mas que são determinantes no sucesso da operação. Tais pré-condições devem ser documentadas para auxiliar a futura implementação da operação em fases futuras.

Após a determinação dos contratos, podemos nos dedicar à identificação de atributos. Os atributos especificam propriedades de uma classe de análise. Em princípio, devemos descrever, em uma classe de análise, somente os atributos que estão especificamente referenciados nos contratos. Por exemplo, caso um contrato especifique que o valor de um determinado atributo deve ser modificado, este atributo deve ser representado.

Podemos utilizar alguns guidelines para o levantamento de atributos. Por exemplo, sabemos que o nome de um atributo deve ser um substantivo. Do mesmo modo o tipo do atributo deve ser abstrato (pois estamos na análise). Assim, devemos utilizar por exemplo um tipo como "quantidade" ao invés de coisas como "inteiro". Devemos tentar reutilizar ao máximo os tipos utilizados. Devemos nos lembrar que os atributos são atrelados a cada instância do objeto e não à classe. Se houver muitos atributos, alguns deles devem ser transformados em classes. Por fim, devemos lembrar que nem sempre é necessário ter atributos.

Que atributos devemos incluir ? Somente aqueles que forem realmente necessários para satisfazer os requisitos de informação relevantes para os casos de uso sendo analisados e/ou aqueles que devem ser "lembrados" para serem utilizados em algum ponto de um caso de uso.

Algumas regras devem ser seguidas … por exemplo, … quando estivermos modelando quantidades, devemos usar um conceito separado "unidade" para mensurar atributos. Isso facilitará o entendimento do modelo e permitirá sua modificação mais facilmente. Da mesma maneira, devemos evitar fazer referência a atributos de classes externas como atributos (uma prática condenada batizada de uso de *foreign-keys*, ou chaves externas). Veja o exemplo da figura abaixo:



Na primeira especificação da classe, colocamos o nome do cliente (*CustomerName*) como um atributo de *Invoice*. Entretanto, esse tipo de representação pode nos causar problemas, pois existe uma entidade que é freguês (*Customer*) e o que queremos indicar é o nome desse freguês. Assim, a segunda representação é mais conveniente, pois permite que mudemos a estrutura da classe *Customer*, sem termos que alterar também *Invoice*.

Em seguida à identificação dos atributos, podemos nos dedicar à procura de associações e agregações. Sabemos que os objetos de análise interagem entre si por meio de conexões em diagramas de interação. Estas conexões são normalmente instâncias de associações entre as classes correspondentes. Deve-se pois estudar as conexões necessárias em um diagrama de interação de modo a determinar as associações necessárias no diagrama de classes, levando-se em conta que o número de relacionamentos entre classes deve ser minizado o quanto possível. Assim, devemos modelar somente os relacionamentos realmente necessários para se modelar a realização dos casos de uso.

Por fim, podemos identificar generalizações. Generalizações devem ser utilizadas para extrair comportamentos comuns ou compartilhados entre diferentes classes de análise. Assim, devemos olhar novamente as classes geradas anteriormente, tentando identificar comportamentos comuns que possam ser generalizados. Havendo a possibilidade, devemos então criar classes abstratas e apontar as generalizações possíveis.

Os resultados dessa atividade, portanto, serão a determinação da estrutura interna de cada classe de análise, em termos de seus atributos e seus métodos. Cada método deve estar associado a um contrato que descreve suas responsabilidades.

18.6 Integração da Arquitetura

A assim chamada Análise dos Casos de Uso (a soma das duas últimas atividades) é realizada de maneira individual por cada desenvolvedor, que gera os diagramas de interação e os diagramas de classes correpondentes para a realização dos casos de uso. Entretanto, durante essa atividade, há um grande potencial para redundância, onde classes com finalidades semelhantes e nomes distintos são geradas. Da mesma forma, pode ocorrer de classes com um mesmo nome, mas com propósitos distintos serem geradas, o que poderia gerar grande confusão, caso essas classes fossem simplesmente acrescentadas à arquitetura de maneira cumulativa.

O objetivo desta atividade é fazer uma integração criteriosa dos diagramas de classes desenvolvidos na etapa anterior à arquitetura inicial de análise desenvolvida no início da fase de análise, efetuando as correções necessárias para que se possa definir de maneira definitiva a arquitetura de análise.

Caso as classes estejam divididas em diversos pacotes, além disso deve-se garantir a independência entre os diferentes pacotes, diminuindo o acoplamento entre eles, e aumentando a coesão entre as classes dentro de um pacote.

Alguns guidelines podem ser utilizados para essa análise de pacotes. Devemos inicialmente definir e formalizar as dependências entre os pacotes, dependendo das classes que estes contiverem e seu uso de objetos de classes de outros pacotes. Essa dependência deve ser indicada explicitamente por meio de arcos do tipo dependência entre os pacotes aparecendo em algum diagrama da arquitetura. Devemos garantir que os pacotes contenham as classes corretas. Isso se faz mantendo-se os pacotes coesos, ou seja, incluindo somente objetos que estejam funcionalmente correlacionados. Devemos ainda tentar limitar as dependências entre pacotes. Assim, devemos considerar a relocação de classes que criem muitas dependências entre pacotes. Tais classes devem ser por exemplo transferidas para outros pacotes de tal forma que essas dependências diminuam.

Ao final desta atividade, os diagramas de classes levantados individualmente durante a análise dos casos de uso são descartados, e as classes correspondentes são então integradas à arquitetura de análise do sistema.

Ao final da fase de análise, os documentos gerados serão portanto os seguintes:

- Arquitetura do Sistema - composta por um diagrama de classes estruturado em diversos sub-diagramas
- Diagramas de Interação - Realização dos Casos de Uso
- Contratos - agrupados para cada classe da arquitetura
- Lista de Requisitos Especiais

19. A Fase de Design no Processo Unificado

Para compreendermos quais as finalidades da fase de design, é necessário fazermos uma comparação com os objetivos da fase de análise. A fase de análise enfatiza a compreensão dos requisitos em seu caráter detalhado. Ou seja, busca-se detalhar os conceitos e operações relacionadas ao sistema. Em outras palavras, na **fase de análise** o que se busca é saber **QUAIS** os processos, conceitos, … , etc, … relacionados ao software em desenvolvimento. Na **fase de design**, iniciamos o desenvolvimento de uma solução lógica baseada no paradigma da orientação a objetos. Ou seja, não mais nos satisfaz saber quais os processos, conceitos, etc. … relacionados com o software, mas saber **COMO** os processos, conceitos, etc… são implementados. Assim, diversos objetivos são esperados na fase de design. Dentre eles, temos:

- A aquisição de uma compreensão profunda dos fatores relacionados a **requisitos não-funcionais** e **restrições** relacionadas a diversas características do sistema, tais como: linguagens de programação, reutilização de componentes de software, sistemas operacionais, tecnologias de distribuição e concorrência, tecnologias de bancos de dados, tecnologias de interface com o usuário, tecnologias de gerenciamento de transações, etc…
- A criação dos elementos lógicos necessários para as atividades de implementação subsequentes, por meio da definição de subsistemas, interfaces e classes
- O planejamento do trabalho de implementação decomposto em pedaços que possam ser trabalhados por diferentes equipes de trabalho, possivelmente ao mesmo tempo
- Captura das principais interfaces entre subsistemas, úteis no projeto da arquitetura do software, principalmente para a sincronização entre diferentes equipes de trabalho
- Permitir a especificação de elementos lógicos a serem implementados por meio de uma notação uniforme
- Criar uma abstração objetiva da implementação do sistema, de tal forma que a implementação seja um mero refinamento do design, permitindo e.g. a geração automática de código.

Vemos portanto que a etapa de design corresponde a uma mudança de postura em relação à fase de análise. Enquanto na fase de análise a nossa preocupação era compreender bem o que o sistema deveria fazer, na fase de design já se começa um planejamento estratégico para a implementação do sistema que será desenvolvido.

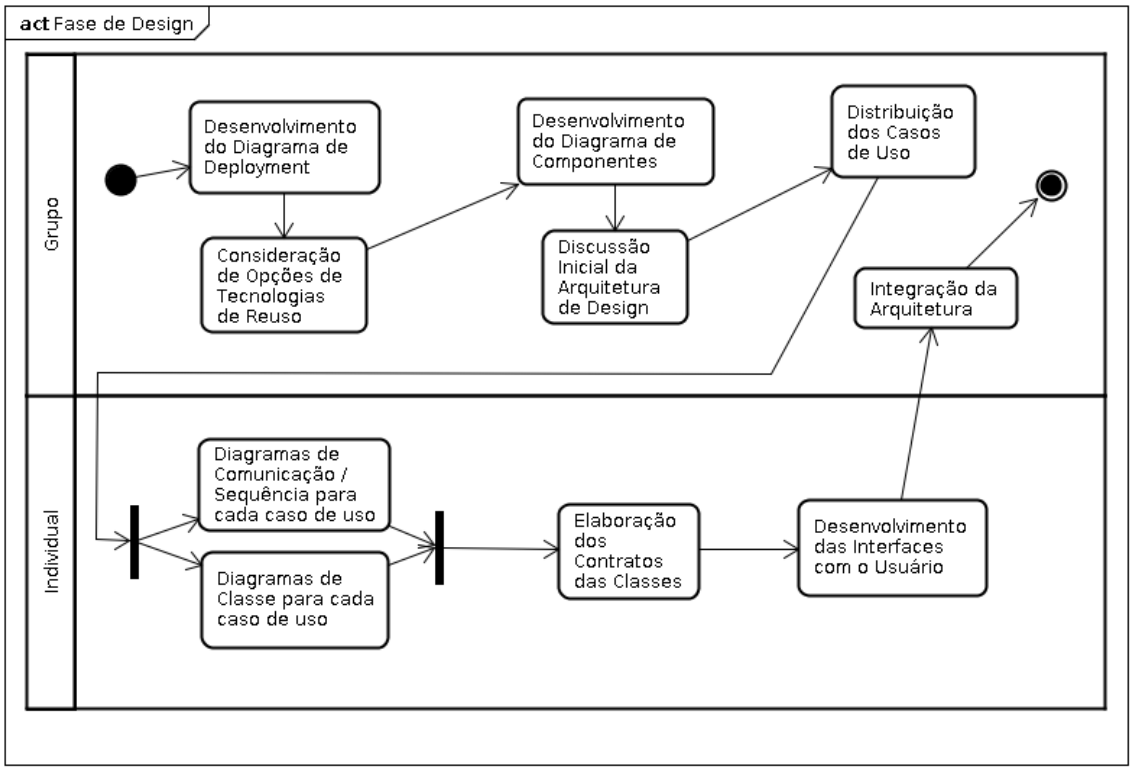
Analisando agora a etapa de design dentro do ciclo de vida de um software, ou seja, tendo em perspectiva a maturidade do desenvolvimento do mesmo em termos de iterações, veremos que o design é enfatizado nas iterações finais da fase de elaboração e o começo da fase de construção. Lembrando que as fases de especificação, análise, design, implementação e testes se repetem em cada iteração, com diferentes ênfases, veremos que somente a partir de um certo número de iterações a ênfase no design será maior. Entretanto, veremos que essa ênfase maior dura somente até o meio da fase de construção. Ao final da fase de construção, a ênfase maior passa a ser sobre a etapa de implementação. De qualquer forma, a etapa de design é uma etapa crítica, pois as principais decisões quanto à arquitetura do sistema são tomadas nessa etapa. Portanto, veremos que a documentação do design será uma documentação mais detalhada, com diversos tipos de artefatos de software:

- Diagrama de Deployment
- Diagrama de Componentes da Arquitetura
- Arquitetura de Design, constituída por um conjunto de diagramas de classes estruturado
- Diagramas de Comunicação/Sequência para cada caso de uso
- Contratos (texto) para as operações de cada classe
- Interfaces com o Usuário

Em termos de atividades, a etapa de design foi customizada a partir do meta-modelo do Processo Unificado, em dez sub-atividades básicas:

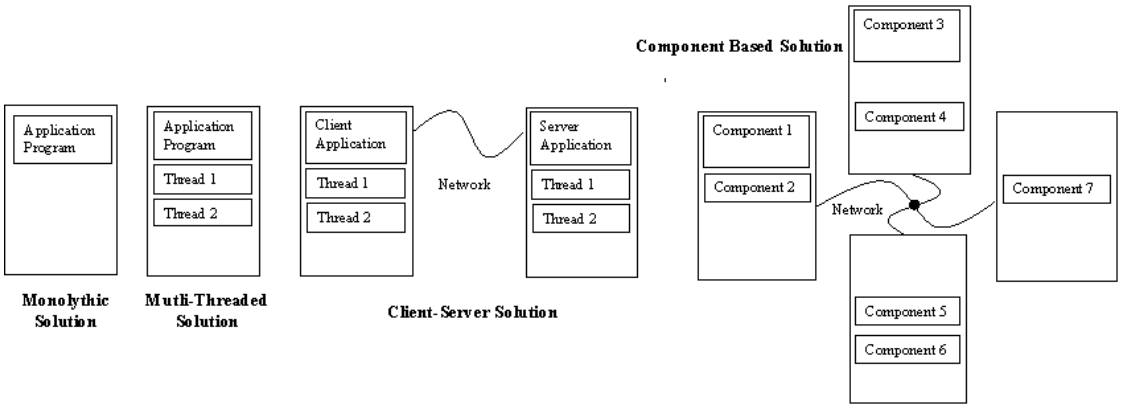
- O Desenvolvimento do Diagrama de Deployment
- A Consideração de Opções de Tecnologias de Reuso
- O Desenvolvimento do Diagrama de Componentes
- A Discussão Inicial da Arquitetura de Design
- A Distribuição dos Casos de Uso
- A Elaboração dos Diagramas de Comunicação/Sequência para cada caso de uso
- A Elaboração dos Diagramas de Classe para cada caso de uso
- A Elaboração dos Contratos das Classes
- O Desenvolvimento das Interfaces com o Usuário
- A Integração da Arquitetura

A sequência dessas atividades, algumas realizadas em grupo, outras realizadas individualmente é apresentada na figura a seguir.



19.1 Desenvolvimento do Diagrama de Deployment

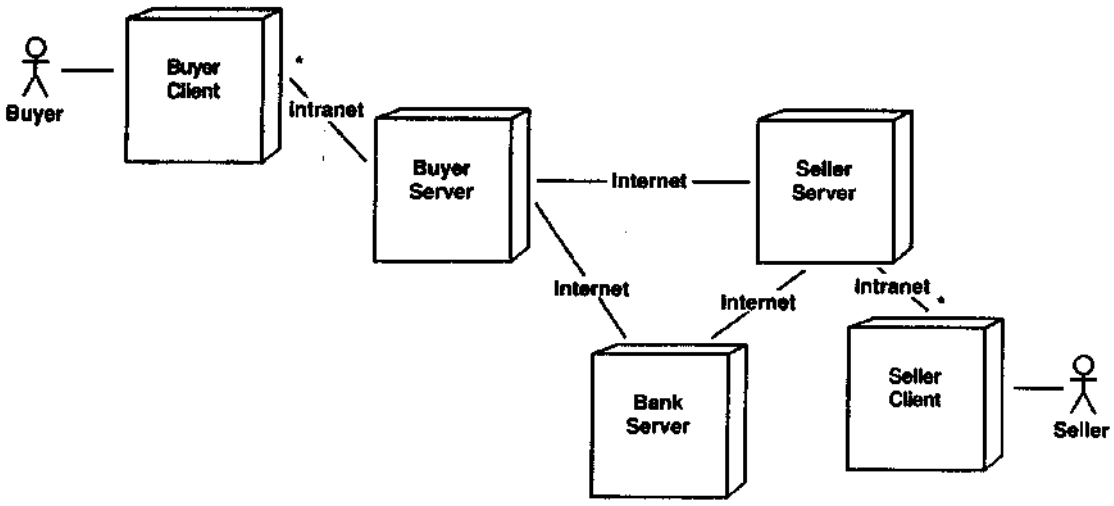
O primeiro passo de uma etapa que no meta-modelo do Processo Unificado é chamada de design arquitetural é a identificação do modelo de distribuição que se deseja implementar. Sistemas de software podem possuir diferentes modelos de distribuição, dependendo-se de como o sistema é distribuído (vide figura a seguir).



O tipo mais simples de solução é o das aplicações monolíticas. Aplicações monolíticas são aquelas que rodam de maneira completamente centralizada, dentro de uma máquina única, e em um único thread de controle. Esse tipo de solução é o mais antigo encontrado em sistemas de software e também o mais simples, pois ele desconsidera uma possível distribuição do sistema. Uma solução um pouco mais sofisticada é a que considera aplicações multi-threaded, ou seja, que se distribui por múltiplas threads de controle. Esse tipo de solução pode ser necessário, quando diversas tarefas devem ser executadas de maneira concorrente (por exemplo, uma animação tal como o ícone animado de um browser de internet, ao mesmo tempo que o sistema continua funcional). Para tornarmos essa solução mais sofisticada, teremos que promover a distribuição da aplicação em múltiplas máquinas. Assim, obtém-se a chamada solução cliente-servidor, que divide a aplicação em duas partes. Uma das partes roda em uma máquina de maior porte (o servidor), podendo se comunicar com diversos clientes, que constituem a outra parte da aplicação. Soluções do tipo cliente servidor são as mais utilizadas hoje em dia em aplicações comerciais. Diversos tipos diferentes de aplicações, tais como aplicações de web, bancos de dados, etc. utilizam esse tipo de arquitetura. Entretanto, não existe nada que nos impeça de distribuirmos nossa aplicação em mais de duas partes. Assim nasceram as primeiras aplicações baseadas em componentes. Ao contrário das aplicações do tipo cliente servidor, cada elemento participante da aplicação final pode ser um cliente, um servidor ou ambos. Assim, obtém-se uma total distribuição do sistema. Para a efetivação de soluções desse tipo, chamada de solução baseada em componentes, utilizam-se plataformas tecnológicas de suporte, tais como o CORBA, ou o DCOM.

Assim, para implementar esse primeiro passo do design arquitetural, devemos **identificar os nós e a configuração da rede** para nossa aplicação. Essa configuração pode ser fundamental para a aplicação em desenvolvimento. Aspectos dessa configuração de rede incluem quais os nós envolvidos e quais as suas capacidades, que tipos de conexões há entre os nós e quais protocolos utilizam, quais as características das conexões (em termos de capacidade, qualidade, etc) e se existe a necessidade de algum tipo de processamento redundante (para aumentar a confiabilidade do sistema, por exemplo). Conhecendo os limites e possibilidades dos nós e suas conexões, o arquiteto pode incorporar tecnologias tais como ORBs (Object Request Brokers), serviços de replicação de dados, etc.

A configuração da rede é representada em nosso modelo por meio de um ou mais diagramas de deployment. Um exemplo de diagrama de deployment é mostrado a seguir:



Havendo diferentes possíveis configurações de rede para o produto (por exemplo, incluindo configurações para testes e simulações), estas devem ser descritas em diagramas de deployment separados.

19.2 Consideração de Opções de Tecnologias de Reuso

O reuso constitui-se hoje de uma das principais técnicas para tornar o desenvolvimento de software mais rápido, ágil e eficiente. Esse reuso, entretanto, pode ser um **reuso oportunista** ou um **reuso sistemático**. Em um reuso oportunista, durante a fase de design identifica-se que certas funcionalidades do sistema em desenvolvimento são semelhantes às funcionalidades de sistemas anteriormente desenvolvidos. Com isso, ao invés de realizar novamente todo o trabalho de design destas funcionalidades, opta-se por reaproveitar o trabalho previamente efetuado. Esse reaproveitamento pode ser tanto de um código (implementação) previamente desenvolvido como de um design previamente elaborado. Apesar desse reuso oportunista muitas vezes acelerar o desenvolvimento de novos sistemas, é hoje quase um consenso entre especialistas em desenvolvimento de software que um reuso *sistemático* deve ser fomentado nas empresas de desenvolvimento de software. Em um reuso sistemático, partes potencialmente reutilizáveis de um sistema já são concebidas originalmente com essa orientação. Dessa forma, durante o design os desenvolvedores podem considerar as diversas opções disponíveis e optar pelo reaproveitamento ou não de partes do sistema que estão projetando. Nesta linha de raciocínio, pode-se ir mais além, e poderemos perceber que estas partes reutilizáveis do sistema podem vir a constituir, por si só, um produto que, tanto pode ser desenvolvido pela própria empresa, como por outras empresas, eventualmente empresas especializadas em desenvolver partes reutilizáveis de sistemas. Atualmente, já existe um grande mercado de partes reutilizáveis, sendo que algumas companhias se especializaram em desenvolver essas partes. Assim, em alguns casos, pode-se considerar a aquisição dessas partes reutilizáveis de outras empresas quando se está efetuando o design de um sistema.

Existem basicamente três maneiras de se implementar o reuso:

- O Reuso de Componentes

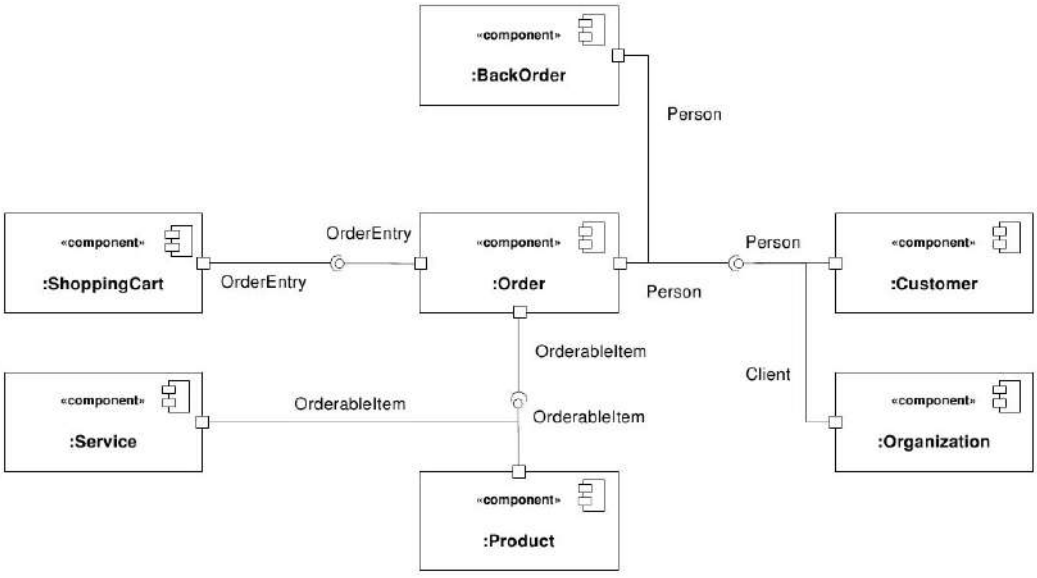
- O Reuso de Frameworks
- O Reuso de Designs

Componentes e **Frameworks** são duas maneiras diferentes de providenciar o reuso de *implementações*. Os assim chamados **Design Patterns** são uma maneira de providenciar o reuso de *designs*.

Particularmente, nesta etapa do design, cabe a nós decidir as partes do sistema que deverão ser reutilizadas, e a forma em que essa reutilização se dará. Nossas opções de reuso afetarão, consequentemente, as etapas posteriores de nosso *workflow* de design.

19.3 Desenvolvimento do Diagrama de Componentes

Esta etapa é uma etapa opcional, preparatória para a definição da arquitetura de design (que é iniciada na fase seguinte a esta), quando se opta por realizar o design do sistema orientado a componentes. Caso a opção seja realizar o design do sistema de forma orientada a frameworks, essa etapa pode ser suprimida e se passar diretamente à próxima etapa. A idéia desta fase é tentar fazer uma divisão inicial do sistema em **componentes** de alto nível, muitas vezes também chamados de **subsistemas**. Nesta etapa, que também faz parte do chamado design arquitetural no meta-modelo o objetivo é **a identificação de componentes e suas interfaces**. Subsistemas provêm um meio de organizar o design do sistema em partes gerenciáveis. A descoberta (e definição) de subsistemas pode ocorrer em diferentes etapas do desenvolvimento. Eles podem ser criados desde o início do design, ou podem ser desmembrados de outros subsistemas, a medida que um determinado pacote ou subsistema se torna muito grande e demande uma decomposição. Nesta etapa, desenvolvemos um diagrama de componentes UML, tentando modelar cada subsistema que podemos identificar de imediato como um componente, definindo-se as possíveis interfaces entre os componentes. Um exemplo de diagrama de componentes é mostrado na figura a seguir:



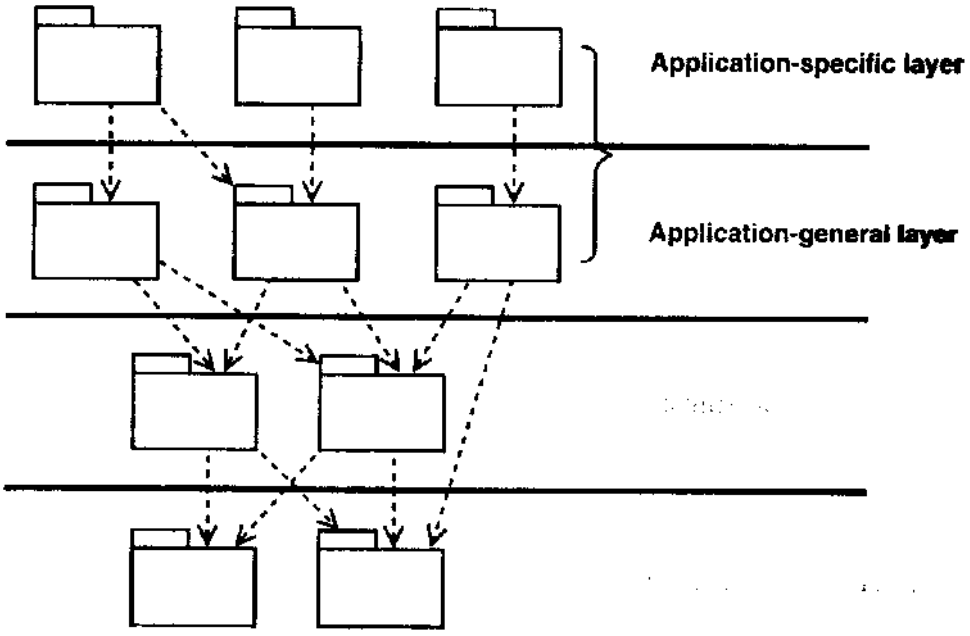
Observe na figura a representação de interfaces providas e interfaces demandadas, e o acoplamento entre elas para constituir o sistema. Esse diagrama visa constituir um primeiro ponto de partida para a definição da arquitetura de design, que deve ser iniciada na atividade a seguir.

19.4 Discussão Inicial da Arquitetura de Design

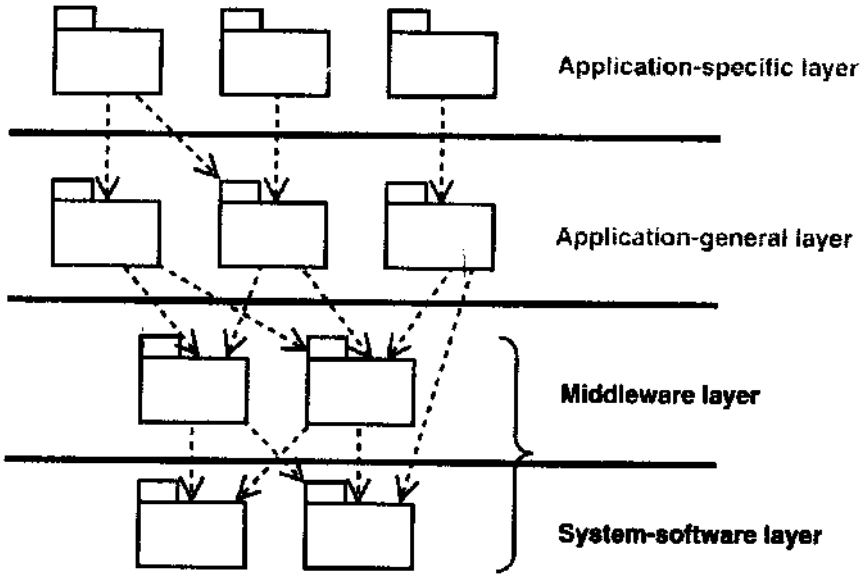
Nesta etapa, iremos desenvolver um esboço da arquitetura do sistema delineado no diagrama de componentes da atividade anterior, de forma a refiná-la e detalhá-la. Esse refinamento se dará por meio de 4 sub-etapas, onde os componentes do diagrama de componentes são transformados agora em pacotes de um diagrama de classes. As interfaces poderão continuar a ser representadas em sua forma estereotipada, pelo menos nessa fase inicial. Nas etapas de refinamento posteriores, ainda durante o design, elas devem ser substituídas pela representação não icônica, com os detalhes da interface devidamente expostos. As 4 sub-etapas são as seguintes:

- identificação de subsistemas de aplicação
- identificação de middleware e subsistemas de software de sistema
- definição das dependências entre subsistemas
- representação das interfaces de subsistemas

A primeira delas é a identificação de subsistemas de aplicação. Nesse passo se identificam os subsistemas específicos da aplicação e as camadas de aplicação geral. Os subsistemas levantados na fase de análise podem ser utilizados como base para a definição destes subsistemas. Entretanto, esses subsistemas sofrerão agora um refino aqui no design.

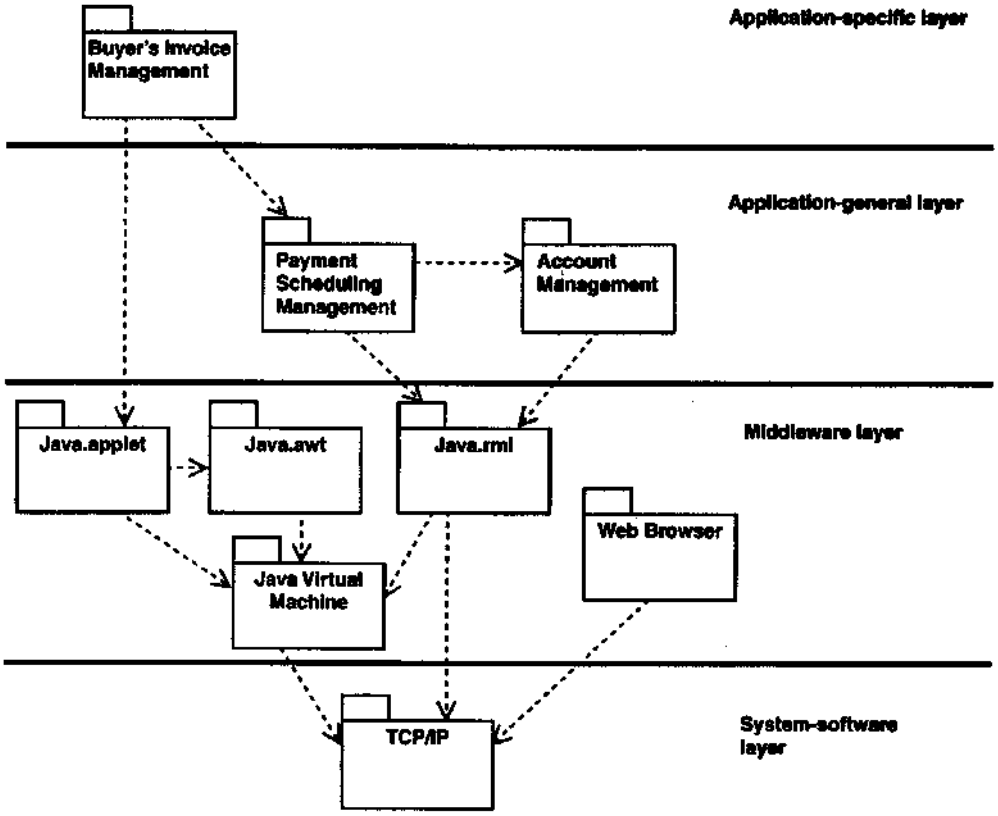


A etapa seguinte envolve a identificação do middleware e dos subsistemas de software do sistema.



O Middleware e o software de sistema são a base das funcionalidades de um sistema. Os elementos que devem ser definidos nessa sub-etapa são exatamente a determinação do sistema operacional, do banco de dados que será utilizado, do software de comunicação que se deseja utilizar, as tecnologias de objetos distribuídos, quais serão os kits de interface gráfica que iremos utilizar e as tecnologias de gerenciamento de transações que forem necessárias.

Uma vez que os subsistemas estejam devidamente identificados, procede-se então à sub-etapa de identificação das dependências entre subsistemas. Para tanto, analisam-se os subsistemas (principalmente aqueles que serão reutilizados), procurando-se identificar as possíveis dependências que hajam neles. Assim, caso haja algum tipo de relacionamento entre subsistemas, é possível haver alguma dependência entre eles. A navegabilidade da dependência deve ser equivalente a do relacionamento em questão. As dependências encontradas aqui devem ser análogas àquelas encontradas na fase de análise. Devemos nos lembrar de estar também identificando as dependências entre interfaces. Um exemplo de diagrama que identifica as dependências entre subsistemas é dado a seguir:



O passo seguinte é a **representação das interfaces dos subsistemas**. Nesse ponto é importante lembrarmos o significado de "interfaces". As interfaces dos subsistemas definem as operações que são acessíveis "de fora" do subsistema. Essas interfaces são providas por classes ou outros subsistemas (recursivamente) dentro do subsistema. Inicialmente, antes que o conteúdo de um subsistema seja conhecido, começa-se considerando as dependências entre subsistemas. Em seguida, analisa-se as classes dentro dos pacotes de análise. As interfaces para os subsistemas de middleware e de software de sistema normalmente são interfaces pré-definidas pelo produto utilizado. Um ponto importante a se colocar aqui é que não basta identificarmos quais são as interfaces, mas também identificarmos as operações disponibilizadas por cada interface.

O próximo passo é **a identificação das classes de design que são arquiteturalmente significativas**. Essas classes são normalmente derivadas das classes obtidas durante a análise. Outro tipo de classe arquiteturalmente significativa são as chamadas classes ativas, ou seja, aquelas que envolvem considerações sobre os requisitos de concorrência do sistema.

Outro tipo de classe significativa é aquela que envolve requisitos sobre desempenho, throughput, disponibilidade e tempo de resposta necessários pelos diferentes atores interagindo com o sistema (e.g. caso o ator demande certo tempo de resposta, deve-se disponibilizar um objeto ativo somente para a interação).

Outro tipo ainda envolve classes que consideram a distribuição do sistema pelos nós (e.g. caso seja necessário suportar distribuição por diversos nós, cada nó demandará pelo menos um objeto ativo para gerenciar a comunicação).

Ainda outro tipo são classes envolvendo outros requisitos, tais como requisitos de inicialização e shutdown, sobrevivência, precaução quanto a deadlocks, capacidade de reconfiguração de nós,

etc ...

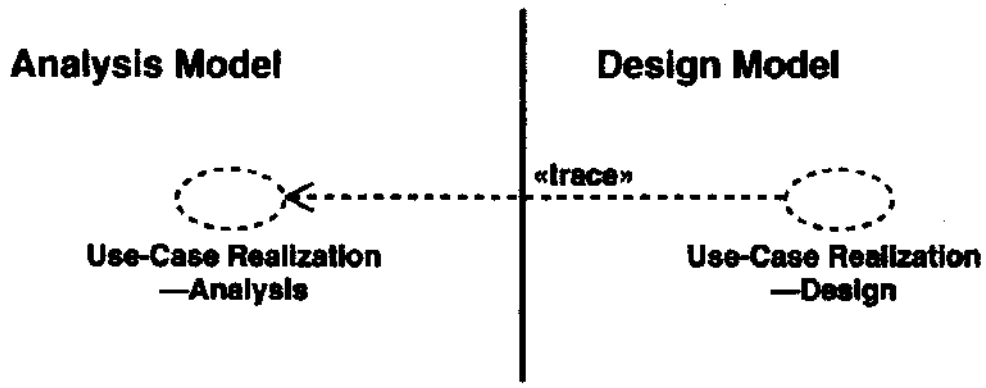
Por fim, para concluir esta etapa, desenvolve-se a **identificação de mecanismos genéricos de design**. Estudam-se os requisitos comuns, tais como os requisitos especiais identificados durante a análise, decidindo-se como implementá-los, dadas as tecnologias de implementação disponíveis. O resultado é um conjunto de mecanismos genéricos de design, instanciados em classes de design. Dentre os tipos de requisitos instanciados estão: a persistência, a distribuição de objetos (uso de objetos distribuídos), os requisitos de segurança, a deteção e recuperação de erros e o gerenciamento de transações.

Ao final desta etapa tem-se, de forma análoga ao que ocorria no design, um diagrama de classes estruturado que será o ponto de partida para a arquitetura de design do sistema.

19.5 Distribuição dos Casos de Uso

Uma vez que se disponha de uma primeira versão da arquitetura de design, desenvolvida na etapa anterior, pode-se passar às etapas subsequentes, que envolvem a chamada "realização dos casos de uso". Para tanto, deve-se distribuir os casos de uso entre os participantes da equipe, de tal forma que cada um possa desenvolver a realização dos casos de uso de maneira individual. Para cada caso de uso selecionado para desenvolvimento, iremos desenvolver um conjunto de artefatos que irão implementar a realização do caso de uso em questão. Essa realização se dará na forma das duas atividades consecutivas, a elaboração dos diagramas de sequência/comunicação e os diagramas de classe para cada caso de uso.

Assim, os casos de uso desenvolvidos na análise sofrerão agora um detalhamento que o consolidarão agora em termos de classes e subsistemas, preparando o terreno para uma implementação em uma linguagem orientada a objetos. Essa transformação pode ser representada na figura abaixo:



19.6 Elaboração dos Diagramas de Sequência/Comunicação para cada caso de uso

O principal objetivo dessa etapa é descrever as interações necessárias para a realização do caso de

uso em questão. Dessa forma, para cada caso de uso, devemos desenvolver um ou mais diagramas de interação, onde a interação entre os objetos do sistema são descritas (veja o material sobre diagramas de interação para rever os detalhes, antes de prosseguir). Nesta etapa, criamos então os diagramas de interação (sequência ou de comunicação) que descrevem o caso de uso em questão. Esses diagramas de interação podem ser um refinamento dos diagramas desenvolvidos na análise, sem as classes estereotipadas, ou mesmo toda uma redefinição destas, caso não se mostrem convenientes em virtude das opções de reuso selecionadas.

19.7 Elaboração dos Diagramas de Classes para cada Caso de Uso

De modo simultâneo à etapa anterior, desenvolvemos os diagramas de classe para os objetos utilizados nos diagramas de interação sendo desenvolvidos.

Além disso, nesta atividade promove-se o detalhamento das estruturas internas das classes, considerando-se:

- operações
- atributos
- relacionamentos dos quais participa
- métodos (como realizar as operações)
- estados válidos
- dependências para com mecanismos genéricos de design
- requisitos importantes para a implementação
- realização correta das interfaces com as quais está envolvida

19.8 Elaboração dos Contratos das Classes

Uma vez que as classes tenham seus atributos, operações e relacionamentos detalhados, deve-se agora detalhar as operações das classes desenvolvendo-se os métodos para cada operação. Esses métodos devem ser formalizados por meio de contratos, desenvolvidos de modo análogo ao que foi feito na fase de análise.

19.9 Desenvolvimento das Interfaces com o Usuário

Uma vez que a interação entre os objetos que implementam a realização dos casos de uso já tenha sido concluída, pode-se agora focar nos requisitos não funcionais que afetam diretamente a implementação. Agora sim, questões de ordem estética-funcional devem ser abordadas, tais como o lay-out definitivo das interfaces gráficas e outras questões do design final do programa.

19.10 Integração da Arquitetura

Uma vez que os casos de uso foram realizados individualmente, por cada desenvolvedor, surge agora a necessidade de que os diagramas de classe sejam integrados na arquitetura do sistema,

constituindo um único diagrama de classes estruturado, onde redundâncias entre classes sejam mitigadas, e as classes todas integradas constituindo uma única arquitetura de design do sistema.

Para tanto, os membros da equipe individual devem agora realizar uma grande reunião, onde cada desenvolvedor apresenta seu diagrama de interação realizando o caso de uso pelo qual ficou responsável, e integra as classes que utilizou na arquitetura do sistema.

Após a apresentação de cada membro, deve-se promover uma discussão sobre a estrutura geral da arquitetura, de forma a implementar melhorias e tornar a arquitetura mais consistente

Uma possível melhoria é garantir que os subsistemas sejam tão independentes quanto possível de outros subsistemas e suas interfaces, ao mesmo tempo que provêm uma interface adequada a suas operações, bem como realizem seu propósito, ou seja, ofereçam uma realização adequada das operações definidas por suas interfaces.

Inicialmente, devemos analisar os diagramas de classes e verificar se as dependências entre os diferentes subsistemas estão adequadas. Em seguida devemos analisar as interfaces de cada subsistema e verificar se podem ser aperfeiçoadas. Por fim, verificamos se todas as funcionalidades providas por cada subsistema estão devidamente representadas pelas interfaces dos subsistemas, ou seja, se para cada interface, existe uma associação com as classes de design do subsistema que provê a funcionalidade em questão.

Ao final desta etapa, a arquitetura de design fica definitivamente constituída e pode-se dar por encerrada a fase de design.

19.11 Componentes, Frameworks e Design Patterns

19.11.1 Componentes

Define-se o conceito de **componente**, como uma unidade modular com um conjunto de interfaces bem definidas, que pode ser substituído dentro de seu ambiente. O conceito de componente é oriundo da área de desenvolvimento baseado em componentes, onde um componente é modelado durante o ciclo de desenvolvimento e refinado sucessivamente durante a instalação e execução do sistema. Um aspecto importante do desenvolvimento baseado em componentes é o reuso de componentes previamente construídos. Um componente pode ser considerado como uma unidade autônoma dentro de um sistema ou subsistema. Ele deve possuir uma ou mais **interfaces**, que podem ser potencialmente disponibilizadas por meio de **portas**, e seu interior é normalmente inacessível. O acesso a um componente deve ocorrer única e exclusivamente por meio de suas interfaces. Apesar disso, um componente pode ser dependente de outros componentes, e a linguagem UML provê mecanismos para representar essa dependência, indicando as interfaces que um componente demanda de outros componentes. Esse mecanismo de representação de dependências torna o componente uma unidade encapsulada, de forma que o componente pode ser tratado de maneira independente. Como resultado disso, componentes e subsistemas podem ser reutilizados de maneira bastante flexível, sendo substituídos por meio da conexão de diversos componentes por meio de suas interfaces e dependências.

Um componente é uma unidade auto-contida que encapsula o estado e o comportamento de um grande número de objetos. Um componente especifica um contrato formal dos serviços que provê a seus clientes e dos serviços que necessita de outros componentes em termos de suas interfaces

disponibilizadas e requeridas.

Um componente é uma unidade substituível que pode ser remanejada tanto durante o design como na implementação, por outro componente que lhe seja funcionalmente equivalente, baseado na compatibilidade entre suas interfaces. De modo similar, um sistema pode ser estendido adicionando-se novos componentes que tragam novas funcionalidades. As interfaces disponibilizadas e requeridas podem ser organizadas opcionalmente por meio de portas. Portas definem um conjunto de interfaces disponibilizadas e requeridas que são encapsuladas de maneira conjunta.

19.11.2 Frameworks

Outro conceito importante, relacionado ao reuso de implementações é o conceito de framework. Frameworks de software são mini-arquiteturas reutilizáveis que provêm a estrutura genérica e comportamento para famílias de abstrações de software, junto com um contexto de metáforas que especificam sua aplicação e uso dentro de um dado domínio. Em termos de software orientado a objetos, frameworks correspondem a um conjunto de classes cooperantes que permitem uma reutilização de design para uma classe de software específica. Assim, um framework provê uma orientação arquitetural, definindo quais as responsabilidades e colaborações entre objetos são necessárias para implementar uma determinada funcionalidade padrão. Para utilizar um framework, um designer normalmente faz o sub-classeamento de classes do framework, e utiliza suas funcionalidades por meio de herança. Assim, por exemplo, em Java, temos diversos frameworks que são disponibilizados na API padrão do Java. Quando programamos janelas ou outros tipos de interfaces com o usuário, o que fazemos é utilizar o framework de janelas do Java. Quando utilizamos programação distribuída, utilizamos outro framework, fazendo uso de classes como *Socket* e *ServerSocket* pertencentes ao framework de acesso a rede do Java.

Observe que a reutilização de implementações por meio de frameworks é fundamentalmente diferente da reutilização de implementações por meio de componentes. Quando reutilizamos um framework, normalmente precisamos conhecer um certo número de diferentes classes que devem trabalhar em conjunto para que uma determinada funcionalidade seja reutilizada. Quando reutilizamos componentes, ao contrário, basta criar o(s) componente(s) e acessá-lo(s) diretamente por meio de sua(s) interface(s).

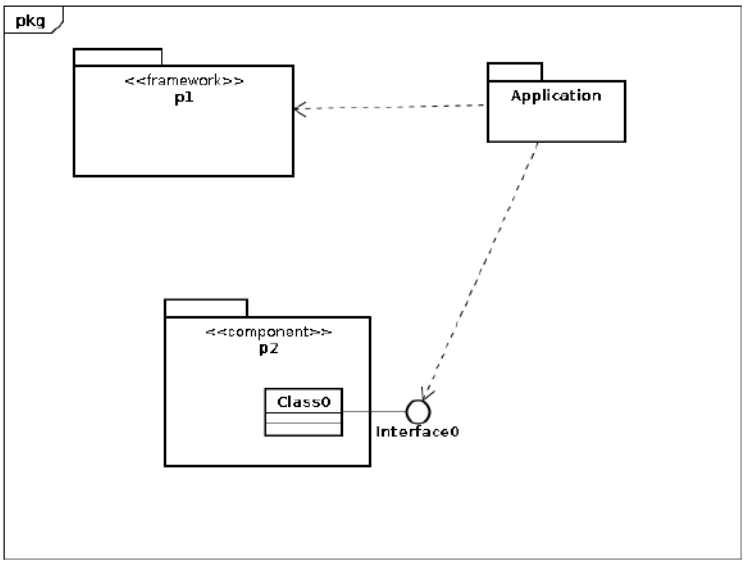


Figura 18.1: Uso de Frameworks e Componentes

A figura 18.1 acima demonstra o uso de frameworks e componentes. O pacote **p1** é um framework que é utilizado pelo pacote **Application**. O pacote **p2** é um componente, que é utilizado pelo mesmo pacote. Pode-se depreender se a arquitetura de um sistema é orientada a frameworks ou a componentes, verificando-se se a dependência que ocorre entre pacotes é direta no pacote (como no caso de **p1**) ou a alguma interface do pacote (como no caso de **p2**). Quando vamos realizar o design do sistema, temos que adotar uma postura. Ou realizamos o design do sistema utilizando-se um paradigma orientado a frameworks ou orientado a componentes. Podemos ainda adotar um paradigma híbrido, utilizando simultaneamente frameworks e componentes. Há vantagens e desvantagens em cada uma das alternativas. O uso de componentes normalmente facilita o reuso, pois basta criarmos o componente e utilizarmos o mesmo por meio de sua(s) interface(s). Entretanto, o reuso de componentes só é possível quando encontramos um componente que satisfaça nossas necessidades. Nem sempre podemos ter um componente que tenha um funcionamento exatamente da maneira que desejamos. Nestes casos, é melhor utilizar um framework, pois o mesmo é mais flexível, proporcionando uma adequação a nossas necessidades. Entretanto, o uso de frameworks normalmente é mais complicado, pois para utilizar um framework temos que conhecer detalhadamente todas as classes que o compõem, o que nem sempre é uma tarefa fácil. Por exemplo, se necessitamos fazer um acesso ao banco de dados, poderíamos utilizar um componente de acesso ao banco de dados, que abstraísse todo o processo de conexão ao banco, montagem de queries, etc. Caso este componente esteja disponível, esse seria o processo mais adequado. Entretanto, caso o componente não esteja disponível, poderia-se utilizar o framework do Java para acesso a banco de dados, o JDBC. Neste caso, teríamos que conhecer todas as classes envolvidas no uso do framework, por exemplo as classes *DriverManager*, *Connection*, *PreparedStatement*, *ResultSet*, *ResultSetMetaData*, etc., bem como seus métodos principais. Os frameworks, dessa forma, são bem mais poderosos do que os componentes (em termos de flexibilidade), mas são bem mais complexos de serem utilizados. Os componentes, por outro lado, são bem mais simples de serem utilizados, mas podem simplesmente não estarem disponíveis ou não atenderem às demandas de especificação do sistema. De maneira reversa, é bem mais simples criarmos um novo framework do que criarmos um novo componente. Para a criação de um framework, basta criarmos um conjunto de classes cooperantes que implementem uma funcionalidade. Já para criarmos um novo componente, é necessário um passo adicional que é a

criação das interfaces, e sua ligação com as classes internas ao componente, o que resulta em um pouco mais de trabalho.

Durante a fase de design, diversos fatores diferentes podem levar à escolha de uma arquitetura orientada a frameworks ou uma arquitetura orientada a componentes. Caso existam componentes disponíveis que atendam as demandas de funcionalidade, deve-se dar preferência ao uso de componentes. Caso não seja possível reutilizar diretamente um componente, deve-se considerar se vale a pena desenvolver um novo componente, reutilizar algum framework que exista disponível ou desenvolver um novo framework. Caso seja possível se abstrair a funcionalidade, de tal forma que exista grande chance futura de reuso, deve-se considerar o desenvolvimento de um novo componente. Caso esse componente possa ser reutilizado no futuro, isso pode fazer valer a pena o custo adicional em seu desenvolvimento. Caso não seja possível depreender um potencial de reuso que valha a pena, deve-se considerar um framework. Caso haja algum framework maduro que possa ser reutilizado (principalmente se esse framework já foi extensivamente testado e validado), deve-se promover a reutilização do framework. Caso o framework existente esteja ainda em um nível experimental, ou não se tenha muita confiança em seu funcionamento, deve-se optar por desenvolver um novo framework.

19.11.3 Design Patterns

O conceito de **design patterns** é hoje fundamental para que um design seja de boa qualidade. Mas … afinal … o que exatamente significa isso ?

A idéia toda sobre *design patterns* começa na experiência que os desenvolvedores de software adquiriram ao longo dos anos durante sua convivência com o desenvolvimento de software. Até mesmo programadores menos experientes já passaram pelo processo de reutilização de uma idéia boa que funcionou muito bem em um programa e que foi "reciclada" em programas posteriores. Imaginemos agora, se todas essas boas idéias fossem organizadas e sistematizadas em um grande "banco de idéias", e disponibilizadas para consumo em larga escala. Essa é a idéia por trás dos design patterns.

Ao longo dos anos, os desenvolvedores de software orientado a objetos experientes acumularam um grande repertório de princípios gerais e soluções que os guiam frequentemente em suas decisões no desenvolvimento de novos softwares. Esses princípios, uma vez identificados e isolados, podem ser formalizados/compilados, dando origem aos chamados patterns (padrões). Assim, os patterns codificam um conhecimento comum sobre princípios de como resolver problemas que aparecem repetidamente em uma dada atividade criativa. Observe que essa idéia de patterns é maior do que simplesmente sua aplicação em desenvolvimento de sistemas de software. Ela pode ser generalizada para qualquer tipo de atividade criativa.

Qual é o formato em que esses patterns são codificados ? Veremos que os patterns aparecem sempre na forma de um par problema/solução, que pode ser aplicado em novos contextos, acompanhados de conselhos sobre como devem ser aplicados. Em geral, os patterns têm um nome sugestivo, o que de certa forma enraíza o conceito do pattern em nossa memória, promovendo seu uso sempre que possível.

Em engenharia de software, a origem dos chamados design patterns se deu a partir da publicação do livro: "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (também conhecidos pela alcunha de "gangue dos quatro",

ou em inglês Gang of Four - GoF). Nesse livro, os autores isolaram um grande número de patterns significativos para o design de software e os disponibilizaram para uso público. Entretanto, o uso de patterns não se restringe à engenharia de software. Ele pode ser utilizado (e é) em qualquer tipo de atividade que envolva o design, como por exemplo em organizações, processos, no ensino, na arquitetura, etc. Mesmo com relação ao desenvolvimento de software, o uso mais famoso de patterns é na fase de design, mas podem haver outros tipos de uso. Alguns autores sugerem a utilização de patterns na fase de análise (o que seria uma espécie de patterns de análise, ao invés de design patterns). Diversos outros livros foram lançados depois do livro da gangue dos quatro. Dentre eles, o livro "Pattern-Oriented Software Architecture: A System of Patterns" (também chamado de POSA book, devido a suas iniciais), de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Somerlad e Michael Stal (que em contraposição à gangue dos quatro acabou recebendo a alcunha da "Gangue dos Cinco da Siemens" ou Gang of Five - GoV). Outra referência importante nesse sentido foi "Pattern Languages of Program Design and PLPD 2 - selected papers from 1st and 2nd conferences on Patterns Languages of Program Design", que faz uma compilação dos principais trabalhos apresentados nas primeiras conferências realizadas cujo tema foi exclusivamente os patterns.

Na verdade, a origem do termo "pattern" é derivada dos trabalhos de um arquiteto, chamado Christopher Alexander, que escreveu vários livros em tópicos relacionados ao planejamento urbano e arquitetura de construção civil. Assim, a origem primeira do termo, está relacionada ao uso de soluções comuns em arquitetura, que acabaram por criar os chamados "estilos arquitetônicos". Por exemplo, o uso de grandes colunas: um estilo grego-clássico, ou então o uso de arcos - um colonial mexicano. Em 1987, Ward Cunningham e Kent Beck decidiram utilizar algumas das idéias de Alexander no desenvolvimento de conselhos para a geração de estruturas eficientes de código em Smalltalk, que se repetiam em diversos programas.

Era o primeiro uso dos patterns relacionado a programação de sistemas. Em 1991, Jim Coplien publicou "Advanced C++ Programming Styles and Idioms" (idiomas são um tipo especial de pattern). De 1990 a 1992, os membros da GoF iniciaram a compilação de um catálogo de patterns. Em 1993 e 94, novas conferências sobre patterns surgiram e logo a seguir apareceu o livro de Design Patterns da GoF, que acabaram por criar essa área de estudos dentro da engenharia de software chamada de "design patterns".

Mas, … , afinal de contas, … o que são patterns ? Diversas respostas podem ser dadas a esta pergunta. Por exemplo: Dirk Riehle e Heinz Zullighoven definem um pattern da seguinte maneira: "Um Pattern corresponde a uma abstração de uma forma concreta que aparece recorrentemente em contextos específicos e não-arbitrários". Puxa vida … complicado, não ? Vamos ler de novo … "abstração" …. "forma concreta" …. "aparece recorrentemente" … "contextos específicos e não arbitrários". Assim vemos em primeiro lugar que um pattern não é algo concreto, mas sim uma abstração. Abstração de quê ? Ora, de formas concretas. Mas que formas concretas ? Aquelas que aparecem recorrentemente, ou seja, repetidas vezes. E onde ? Em contextos específicos e não arbitrários, ou seja, de formas que não podemos prever a princípio, e sempre de uma maneira um pouco diferente. Assim vemos que a noção de pattern está voltada para a solução de problemas de design que se repetem em diferentes contextos. Vejamos uma outra definição: "um pattern corresponde ao encapsulamento de uma informação instrutiva que captura a estrutura essencial e a efetividade de uma família de soluções de sucesso em problemas recorrentes que surgem em determinados contextos e sistemas de forças". Aqui vemos uma complementação da definição anterior, ou seja, a idéia de que os patterns capturam soluções completas e não simplesmente princípios gerais ou estratégias. Essa é uma característica muito importante de um pattern. Outra

característica importante é que os patterns são conceitos testados e aprovados e não teorias ou especulações. Assim, dizemos que os patterns são descobertos, e não inventados ! Para que um pattern seja descoberto, é necessário que ele apareça repetidas vezes dentro do código de diferentes desenvolvedores, e sempre como uma solução de sucesso. Caso a solução seja um fracasso, teremos os chamados *anti-patterns*.

Podemos dizer que o uso de frameworks corresponde a uma reutilização do design por meio da reutilização direta do código (no caso, por meio de herança). Assim, um framework nada mais é do que a implementação de um sistema de design patterns.

Entretanto, é necessário fazermos uma diferenciação bem explícita entre o que é um pattern e o que é um framework. Design patterns são entidades mais abstratas que frameworks. Frameworks estão de certa forma embutidos no código, ao passo que somente exemplos (ou instâncias) de patterns estarão embutidas no código. Uma vantagem dos frameworks é que estes podem ser formalizados diretamente em uma linguagem de programação e não somente estudados, sendo executados e reutilizados diretamente. Design patterns, ao contrário, necessitam ser implementados a cada vez que são utilizados.

Observe que um framework pode conter diversos design patterns, mas o contrário nunca é verdadeiro. Design patterns são menos especializados que frameworks. Frameworks sempre têm um domínio particular de aplicações em vista. Design patterns podem ser utilizados, em princípio, em qualquer tipo de aplicação ao contrário.

19.11.4 GRASP Patterns

Um conjunto de patterns que é particularmente importante para um design bem feito inclui os chamados GRASP Patterns (General Responsibility ASsignement Principles), que determinam a atribuição de responsabilidades a objetos durante a fase de design. Essa atribuição normalmente é feita durante a atividade de design de casos de uso e aparece durante a criação dos diagramas de interação (diagramas de comunicação e diagramas de sequência) que detalham o caso de uso. Os GRASP patterns descrevem os princípios gerais para a atribuição de responsabilidades a objetos, expressos na forma de patterns. O conhecimento desses patterns é importante durante a fase de criação dos diagramas de interação, pois permitem a criação de programas de melhor qualidade. A assimilação de GRASP patterns é especialmente útil com relação a programadores com pouca experiência, pois com seu uso, rapidamente passarão a gerar um código que imbui um conjunto de princípios básicos com os quais somente programadores experientes estão acostumados a lidar.

Os principais GRASP patterns são os identificados pelos seguintes nomes: *Expert*, *Creator*, *High Cohesion, Low Coupling* e *Controller*. Para compreendê-los melhor, é necessário antes analisarmos o que significa atribuirmos responsabilidade a objetos.

Na programação orientada a objetos, criamos o design de um caso de uso distribuindo tarefas ou responsabilidades entre múltiplos objetos participantes do sistema. Façamos uma metáfora com relação à construção de uma casa. Utilizando os princípios da programação estruturada, identificaríamos as tarefas necessárias à construção de uma casa e as executaríamos uma após a outra. Utilizando os princípios da programação estruturada, o que fazemos é contratar uma equipe de objetos e atribuir responsabilidades a cada um deles. Assim, teremos o pedreiro que terá como responsabilidade assentar tijolos, o azulejista que terá como responsabilidade assentar os azulejos, o encanador que terá como responsabilidade a parte hidráulica, o eletricista, a parte elétrica, o

marceneiro a montagem do telhado e por aí vai. Uma vez que a equipe esteja formada, criamos então uma dinâmica de trabalho em que cada objeto sabe quais são as suas responsabilidades, e passam então a interagir entre si para que a casa seja montada. Um programa orientado a objetos funciona exatamente desta maneira. Criamos um conjunto de objetos, atribuimos as responsabilidades aos objetos e depois colocamos eles para interagir. Façamos agora uma abstração dessa história para ver que tipos de responsabilidades podem existir. Basicamente existem os seguintes tipos de responsabilidade:

- Responsabilidade de Saber
    - Guardando o conhecimento consigo
    - Sabendo quem sabe
    - Sabendo como derivar um conhecimento de outros conhecimentos
- Responsabilidade de Fazer
    - Fazendo alguma coisa sozinho
    - Criando novos objetos e delegando-lhes sub-responsabilidades
    - Controlando e coordenando atividades em objetos que já existem

Observemos que existem dois tipos básicos de responsabilidades: a responsabilidade de saber e a responsabilidade de fazer. No primeiro caso, os objetos têm apenas a missão de informar, quando requisitados, alguma informação pela qual são responsáveis. No segundo caso, os objetos têm como responsabilidade exercer alguma atividade. No primeiro caso ainda, um objeto pode guardar o conhecimento em si próprio (em seus atributos), ou então ele deve fazer referências a outros objetos, que detêm o conhecimento desejado. Ou ainda, podem, a partir de outros conhecimentos, derivar o conhecimento em questão. No segundo caso, (a responsabilidade de fazer), o objeto pode fazer alguma coisa por si só (por meio de seus métodos), ou então utilizar outros objetos para ajudá-lo. Nesse caso, ele tanto pode criar por si só esses objetos, como pode controlar e coordenar um conjunto de objetos já existentes. Veremos que a maneira como atribuímos essas responsabilidades entre objetos pode fazer toda a diferença entre um design bem feito e um design mal feito. Vejamos os GRASP patterns:

PATTERN: EXPERT

PROBLEMA: Quem deve ser o responsável em prover algum tipo de informação ?

SOLUÇÃO: Devemos atribuir a responsabilidade ao objeto **expert** (ou especialista) - uma classe que tem a informação necessária para informar os dados em questão - seja por sabê-los por si próprio, ou sabendo quem sabe ou sabendo calculá-lo.

Esse pattern nos diz que mesmo que optemos por distribuir o conhecimento em diversos objetos, devemos sempre ter um objeto do tipo expert, que sabe destrinchar o conhecimento e nos oferecê-lo. Assim, sabemos sempre em quem confiar e não nos perdemos fazendo procuras ou buscas em estruturas de dados que podem ser muitas vezes confusas. Assim, ao invés de fazermos a busca pelos dados nós mesmos, devemos utilizar um expert.

Os benefícios desse pattern são vários. Em primeiro lugar, o uso de um expert mantém o encapsulamento no programa, fazendo com que o código fique inteligível. Além disso o expert

suporta um outro pattern (a ser visto a seguir), chamado low-coupling, que em geral leva a sistemas mais robustos e gerenciáveis. Da mesma maneira, como o comportamento normalmente é distribuído em múltiplas classes, encoraja-se a criação de classes do tipo "lightweight", que são mais coesas. Isso está de acordo com um outro pattern chamado de high-cohesion, que portanto também é suportado.

PATTERN: CREATOR

PROBLEMA: Quem deve ser o responsável pela criação de novas instâncias de alguma classe ?

SOLUÇÃO: Devemos atribuir a uma classe B a responsabilidade pela criação de uma instância da classe A se uma das condições abaixo for verdadeira:

- B agrega objetos do tipo A
- B contém objetos do tipo A
- B referencia objetos do tipo A
- B utiliza objetos do tipo A
- B possui os dados de inicialização que devem ser passados a A quando este é criado

Este pattern nos dá portanto as condições em que um objeto deve ser o creator de outros objetos. Seu principal benefício é que suporta o pattern low-coupling, o que implica em menores dependências de gerenciamento e maiores oportunidades de reuso.

PATTERN: LOW COUPLING

PROBLEMA: Como suportar baixa dependência e maiores potencialidades de reutilização entre classes ?

SOLUÇÃO: Devemos atribuir a responsabilidade de tal forma que o acoplamento permaneça baixo.

Para compreendermos o pattern acima, é necessário compreendermos o que significa acoplamento. Acoplamento é uma medida de quão fortemente uma classe está conectada a outras classes. Essa conexão pode ser na forma da necessidade de referenciar ou ter que utilizar a outra classe para seu próprio funcionamento. Uma classe com baixo acoplamento não é dependente de muitas outras classes. Uma classe com alto acoplamento, ao contrário, depende de diversas outras classes para seu funcionamento. Classes com esse perfil são indesejáveis, pois mudanças em alguma das classes de que depende implicam mudanças na classe em questão. Da mesma forma, classes desse tipo são mais difíceis de se compreender de forma isolada e por conseguinte mais difíceis de serem reutilizadas, pois seu uso implica na necessidade da presença adicional das classes de que depende.

Devemos observar que o low-coupling é um princípio que devemos ter em mente a cada instante, quando tomamos nossas decisões de design. É uma meta que deve estar sendo continuamente considerada. O low-coupling suporta o design de classes que são mais independentes entre si e mais re-utilizáveis. Ele não deve ser utilizado, entretanto, de maneira isolada, mas sim conjuntamente

com outros patterns. Dentre seus benefícios, temos que classes que implementam o low-coupling são mais dificilmente afetadas por mudanças em outros componentes, são mais simples de se entender e mais convenientes para o reuso.

PATTERN: HIGH COHESION

PROBLEMA: Como manter a complexidade de um sistema gerenciável ?

SOLUÇÃO: Devemos atribuir responsabilidades de tal forma que a coesão entre os objetos permaneça alta.

De novo, para compreendermos esse pattern, é necessário que compreendamos o que significa coesão. Coesão é uma medida de quão fortemente relacionadas e focalizadas estão as responsabilidades atribuidas a uma certa classe. Uma classe com responsabilidades afins, e que não se encontram sobrecarregadas de responsabilidades, possuem uma alta coesão. Uma classe com baixa coesão ou tem responsabilidades que não são afins ou então tem muitas responsabilidades. Tais classes são indesejadas pois são difíceis de compreender, difíceis de reutilizar, difíceis de manter, ao mesmo tempo que são muito frágeis, ou seja, são constantemente afetadas por mudanças no sistema.

Da mesma maneira que o low-coupling o high-cohesion é um princípio que deve ser mantido em vista durante todas as decisões de design. A delegação de responsabilidades para classes filhas pode ser uma maneira de aumentar a coesão, ao mesmo tempo que mantém o acoplamento baixo. Os benefícios da alta coesão são uma maior claridade e maior facilidade de compreensão de nosso design, ao mesmo momento em que manutenções e melhorias são simplificadas. O baixo acoplamento é sempre suportado também. Classes que suportam responsabilidades afins suportam um maior potencial de reuso, pois podem ser reutilizadas quase que diretamente em outras situações. Isso ocorre pois classes de alta coesão possuem um propósito bem específico.

PATTERN: CONTROLLER

PROBLEMA: Quem deve ser o responsável por gerenciar eventos do sistema ?

SOLUÇÃO: Devemos atribuir a responsabilidade a uma classe representando a entidade que pretensamente deveria estar reagindo ao evento (**controller**).

Esse GRASP pattern nos diz que devemos concentrar o tratamento de eventos em objetos do tipo controller, ou seja, que sejam dedicados a essa finalidade. Em princípio, teremos um número de controllers proporcional aos objetos de interface recebendo eventos do sistema operacional. Algumas linguagens, como o Java por exemplo, em seu framework de janelas favorece e recomenda o uso de controllers. Dentre outros benefícios, o uso de controllers aumenta o potencial de reuso e compreensão de componentes.

De uma maneira geral, durante o design do sistema, devemos estar considerando o uso dos GRASP patterns como diretivas na concepção de interação entre objetos. Ao fazer isso, estaremos melhorando a qualidade de nosso software, independentemente da linguagem de programação que iremos adotar para implementação.

19.11.5 GoF Patterns

A seguir, temos uma visão resumida dos GoF Patterns, conforme o livro "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Um resumo desses patterns pode ser visto na figura a seguir:

		Propósito		
		Criacional	Estrutural	Comportamental
Escopo	Classe	Factory Method	Adapter (classe)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (objeto) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

A descrição completa destes patterns está além dos propósitos deste curso. Entretanto, uma versão sumarizada dos mesmos é apresentada em ordem alfabética, a seguir.

Abstract Factory

- Provê uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas

Adapter

- Converte a interface de uma classe em outra interface que os clientes esperam. O Adapter permite que classes com interfaces incompatíveis trabalhem entre si

Bridge

- Desacopla uma abstração de sua implementação, de tal forma que ambas podem variar de maneira independente

Chain of Responsibility

- Evita o acoplamento entre o sender de uma requisição e o receiver da mesma, dando a chance a mais de um objeto de responder a uma requisição. Encadeia o objeto sendo recebido e o passa ao longo da cadeia até que um objeto o processe

Command

- Encapsula uma requisição na forma de um objeto, permitindo que se parametrize clientes com diferentes requisições, requisições de filas ou logs, suportando o undo de operações

Composite

- Compõe objetos em estruturas do tipo árvore, de forma a representar hierarquias do tipo parte-todo. Clientes podem tratar objetos individuais ou objetos compostos de maneira uniforme.

Decorator

- Atribui dinamicamente novas responsabilidades a um objeto. Constitui-se de uma alternativa flexível ao subclasseamento para estender funcionalidades

Façade

- Provê uma interface unificada a um conjunto de outras interfaces em um subsistema, de forma a tornar o subsistema mais fácil de se utilizar

Factory Method

- Define uma interface para a criação de objetos, mas delega a subclasses a decisão de que classe instanciar

Flyweight

- Usa compartilhamento para suportar um vasto número de objetos de menor granularidade de maneira mais eficiente

Interpreter

- Dada uma linguagem, define uma representação para sua gramática, em conjunto com um interpretador que utiliza a representação para interpretar sentenças da linguagem

Iterator

- Provê uma maneira para acessar elementos em um objeto agregado, de forma sequencial, sem expor sua representação interna

Mediator

- Define um objeto que encapsula a maneira como um conjunto de objetos interagem entre si. O Mediator promove um acoplamento fraco, ao evitar que os objetos se referenciem mutuamente de maneira explícita, o que permite que se varie a interação de maneira independente

Memento

- Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de tal forma que o objeto possa ser restaurado a este estado posteriormente

Observer

- Define uma dependência do tipo um-para-muitos entre objetos, de tal maneira que quando um objeto tem seu estado alterado, todos os objetos que dependem dele são notificados e atualizados automaticamente

Prototype

- Especifica que tipo de objeto criar usando uma instância prototípica, e cria novos objetos fazendo uma cópia do protótipo

Proxy

- Provê um substituto para outro objeto, que promove o acesso ao objeto desejado indiretamente

Singleton

- Garante que uma determinada classe possui somente uma única instância, e provê um ponto de acesso global a ela

State

- Permite que um objeto possa alterar seu comportamento, quando seu estado interno muda. O objeto parecerá ter modificado sua classe

Strategy

- Define uma família de algoritmos, encapsula cada um deles e torna-os intercambiáveis. O Strategy permite que o algoritmo varie independente dos clientes que o usam

Template Method

- Define o esqueleto de um algoritmo em uma operação, delegando alguns passos para subclasses. O Template Method permite que subclasses redefinam certos passos de um algoritmo, sem modificar a estrutura do algoritmo

Visitor

- Representa uma operação a ser realizada nos elementos de uma estrutura. O Visitor permite que se definam novas operações sem que seja necessário se mudar as classes dos elementos nos quais opera.

20. Implementação e Testes no Processo Unificado

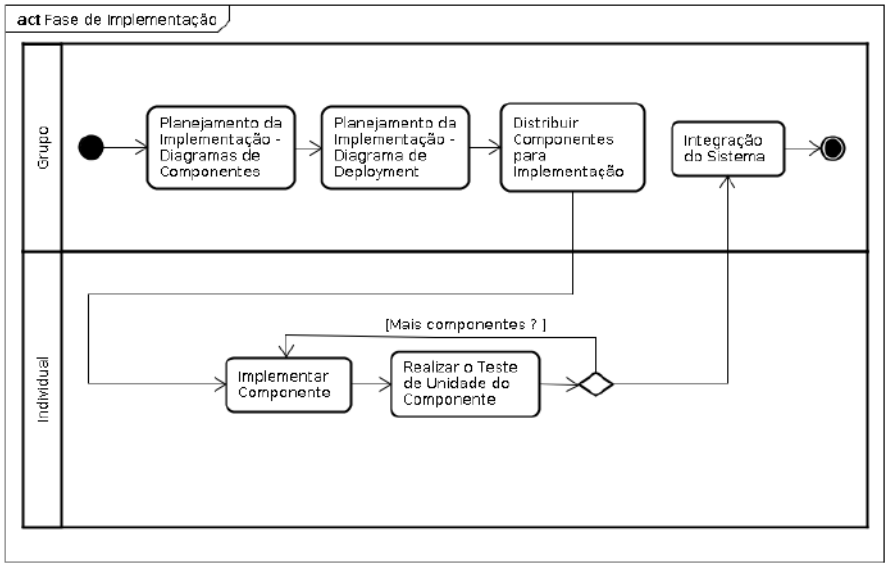
Normalmente, em sistemas de grande porte, o desenvolvimento da implementação e dos testes é distribuído dentre diversos profissionais, que se encarregam de desenvolver de maneira isolada as partes ou componentes que integram o sistema, sendo que estas partes depois devem ser integradas e testadas por meio de diversas rotinas de teste. Estes profissionais executam diferentes papéis, dentre os assim chamados Arquitetos, Integradores de Sistema, Engenheiros de Componentes, Programadores, etc. Nesta versão pedagógica do Processo Unificado, distinguiremos somente dentre as atividades que são executadas em grupo daquelas que são executadas individualmente por cada profissional.

20.1 A Fase de Implementação

A fase de implementação utiliza-se dos artefatos desenvolvidos no design para implementar o sistema em termos de seus componentes. Estes componentes podem ser o código fonte, scripts de integração, códigos binários, arquivos executáveis ou outros tipos de componentes. Os objetivos da fase de implementação podem ser descritos como a seguir:

- Planejar a integração do sistema necessária na iteração corrente
- Distribuir o sistema entre componentes executáveis localizados nos nós do modelo de distribuição
- Implementação das classes de design e dos subsistemas especificados no design (geração de código)
- Teste individual (teste de unidade) dos componentes e posterior integração em módulos executáveis.

Esses objetivos são atingidos por meio das atividades indicadas na figura a seguir:



20.1.1 Planejamento da Implementação - Diagrama de Componentes

O planejamento da implementação encontra-se dividido em nossa metodologia pedagógica em 2 sub-etapas. A primeira delas é a elaboração dos diagramas de componentes. O objetivo principal dessa fase é a criação de um plano de construção para o software, descrevendo quais as construções necessárias na presente iteração, e quais os requisitos que essa construção implementa. De maneira pragmática, isso corresponde a identificarmos quais são os componentes executáveis, bibliotecas, arquivos de configuração, arquivos de help, etc, necessários para integrar a presente iteração.

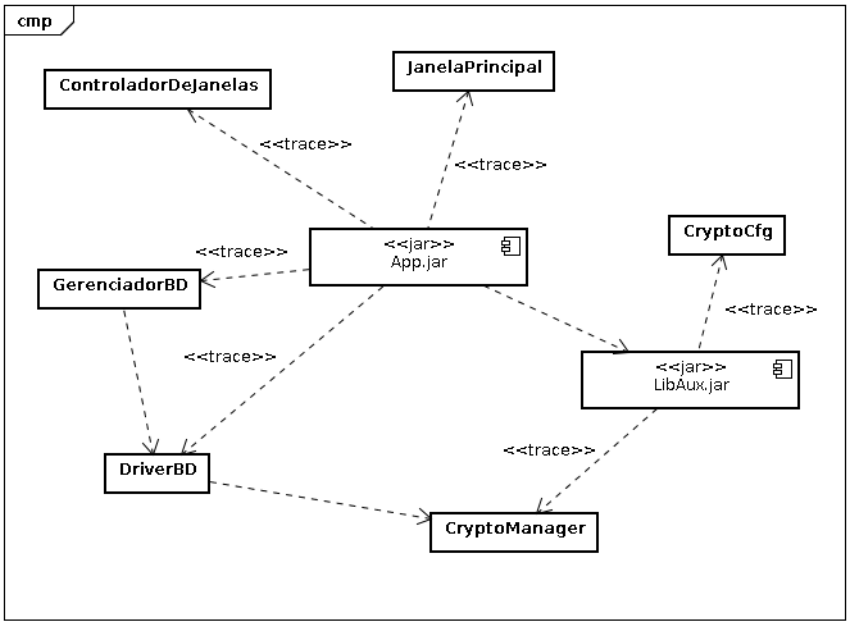
Durante o planejamento da construção, deve-se verificar eventuais implementações de iterações anteriores e fazer o reuso das partes desta que já estão consolidadas. Isso garante um caráter incremental ao desenvolvimento.

Um conceito fundamental durante a fase de implementação é o conceito de componente. Esse conceito já havia aparecido durante o design, mas aqui ele é utilizado em todo seu poder de modelagem. Componentes podem ser arquivos executáveis, arquivos de texto, bibliotecas de funções (as famosas DLL, no caso do Windows), tabelas, documentos ou outros tipos de entidades de software. A transição que se faz da fase de design para a fase de implementação é exatamente o mapeamento entre entidades de design (subsistemas, classes, etc …) em entidades do modelo de implementação (componentes). A identificação dos componentes no modelo de implementação é feita, no UML, por meio de estereótipos que identificam o tipo de componente. Assim, o UML prevê os seguintes estereótipos que podem ser aplicados a componentes:

«executable» - utilizado para identificar componentes executáveis
«file» - utilizado para identificar arquivos de propósito geral
«library» - utilizado para identificar bibliotecas de funções (bibliotecas dinâmicas, ou arquivos DLL)
«table» - utilizado para representar tabelas de propósito geral
«document» - utilizado para representar a documentação do sistema

Dois diagramas de componentes distintos são construídos nesta fase. O primeiro deles, é um diagrama de componentes que indica como os componentes de implementação estão relacionados com as classes identificadas no design. Chamamos a esse diagrama de "Diagrama de Constituição de Componentes". O segundo diagrama de componentes tem por finalidade modelar as dependências entre os componentes a serem construídos e componentes externos, tais como bibliotecas, etc. Chamamos a esse diagrama de "Diagrama de Dependências Externas".

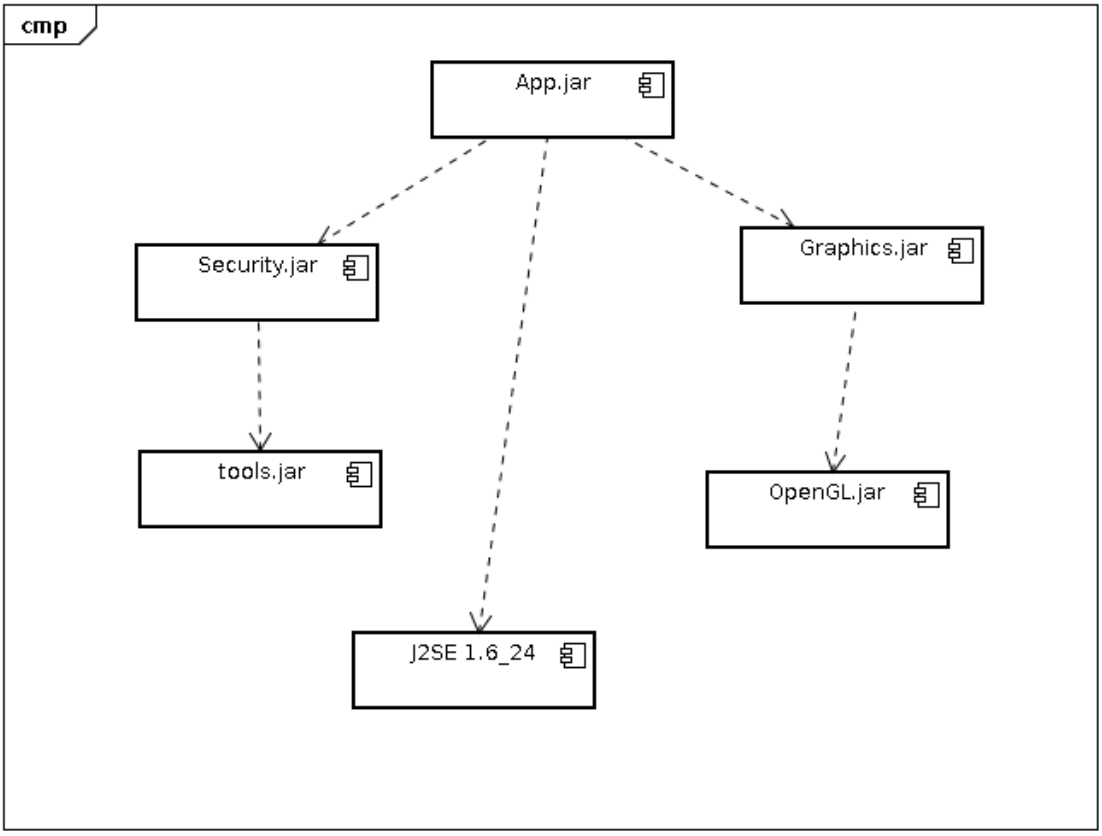
Um exemplo de um "Diagrama de Constituição de Componentes pode ser visto na figura a seguir.



Nesse diagrama, representamos os diferentes componentes executáveis ou bibliotecas dinâmicas a serem construídos durante a fase de implementação, e seu relacionamento com as classes de design que ele implementa. Para indicar o mapeamento entre as classes de design e os componentes a serem implementados, utiliza-se relacionamentos de dependência com o estereótipo «trace». Da mesma forma, indicamos no diagrama potenciais dependências entre as classes de design, que podem demandar que também seja indicada uma dependência entre diferentes componentes. No exemplo da figura, a classe GerenciadorBD depende da classe DriverBD. Entretanto, como essas duas classes estão simultaneamente no componente App.jar, isso não é grande problema. Entretanto, como a classe DriverBD depende de CryptoManager, que não está localizado em App. jar, (mas sim em LibAux.jar), isso nos obriga a incluir uma relação de dependência entre App.jar e LibAux.jar.

Na prática, construir um único "Diagrama de Constituição de Componentes" pode ser inviável, caso o sistema possua um número grande de classes. Existem diversas soluções alternativas nesse caso. Uma primeira alternativa seria criar um grande número de diagramas, onde cada um deles poderia indicar somente certo número de componentes com suas classes relacionadas. Entretanto, isso também pode se tornar inviável, dependendo-se do número de classes e componentes. Uma outra solução alternativa é agrupar as classes em pacotes, e aí representar o relacionamento «trace» diretamente com o pacote, ao invés da classe. Isso é equivalente a se dizer que o componente em questão implementa TODAS as classes do pacote. A divisão de um mesmo pacote em dois componentes significaria que na verdade este pacote está mal projetado. O ideal seria dividí-lo em dois pacotes, onde cada um deles estaria localizado em um componente diferente.

Um exemplo de um "Diagrama de Dependências Externas" pode ser visto na figura a seguir:

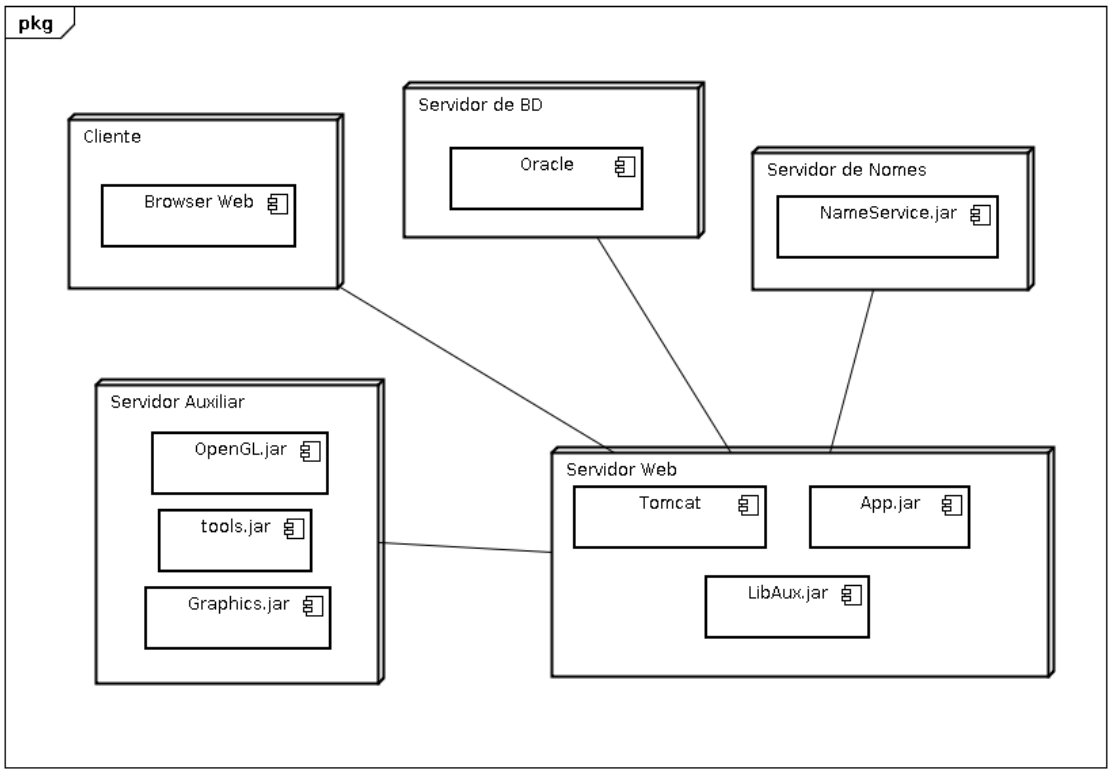


Neste diagrama, mostramos como o componente App.jar dependerá de diversas bibliotecas externas. Esse diagrama é importante durante a fase de integração do sistema, para que os scripts de compilação e linkagem possam ser construídos de forma adequada. Apesar de muitas dependências externas poderem ser definidas ainda ANTES da codificação, esse diagrama pode sofrer modificações durante a etapa posterior de implementação de componentes, quando outras dependências externas podem ser incluídas ou suprimidas, dependendo-se do caso.

20.1.2 Planejamento da Implementação - Diagrama de Deployment

Após a geração dos dois diagramas de componentes, que identificam os componentes do sistema a serem implementados, bem como suas dependências, procede-se ao mapeamento desses componentes nos diferentes nós do sistema, o que é feito por meio de um refinamento do **diagrama de deployment** desenvolvido no design, agora com a identificação dos componentes e objetos ativos (aqueles com threads de controle individual). Diagramas de deployment, conforme já havíamos visto na fase de design, mostram a configuração de elementos capazes de processar software (computadores, dispositivos periféricos, etc…) e como os componentes de software, processos e objetos se encontram instalados dentre eles. Entretanto, na fase de design, esses componentes eram apenas componentes conceituais. Aqui agora na implementação, os componentes passam a ganhar uma fisicalidade maior, ganhando um nome de arquivo e portanto refinamos o diagrama de deployment de design para registrar essa nova informação. Observemos que existem diversos tipos de componentes de software. Estes tanto podem representar módulos executáveis como código fonte ou documentação. Aqui nos diagramas de distribuição somente irão aparecer os componentes de software que representem manifestações de tempo real de unidades de código.

Componentes que não existem como entidades de tempo real (e.g. arquivos fonte ou documentação), não deverão aparecer nesses diagramas, devendo aparecer somente nos diagramas de componentes. Um exemplo do refino de um diagrama de deployment está na figura a seguir:



Observe no diagrama, que os componentes agora já ganham os nomes finais que devem assumir após a implementação. Entretanto, alguns componentes de terceiros (reutilizados), que não serão desenvolvidos aqui podem ainda manter uma representação mais informal, conforme se visualiza na figura acima.

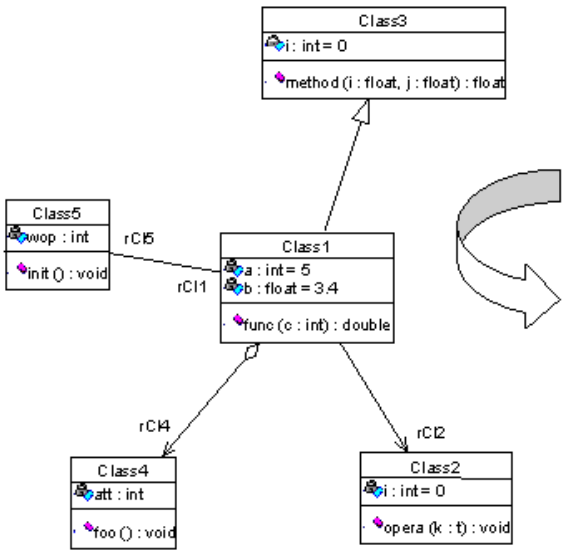
Em algumas situações, múltiplos diagramas de deployment podem ser necessários. Esse é o caso, por exemplo, quando existem múltiplas configurações de instalação possíveis, ou quando se deseja especificar configurações especiais para testes. Nesses casos, um diagrama de deployment diferente deve ser construído para cada configuração. Em alguns casos, onde testes são feitos dentro de uma máquina virtual, rodando múltiplas versões de sistemas operacionais, pode-se utilizar um nó com o estereótipo «executionEnvironment» para representar essa situação.

20.1.3 Distribuição de Componentes para Implementação

Uma vez que o planejamento para a implementação de componentes esteja concluída, esses componentes podem ser atribuídos aos diferentes desenvolvedores que irão codificá-los. Essa divisão normalmente ocorre de forma a distribuir os componentes relacionados a um mesmo caso de uso a um mesmo desenvolvedor. Após essa distribuição, os desenvolvedores podem iniciar seus trabalhos sobre os componentes individualmente.

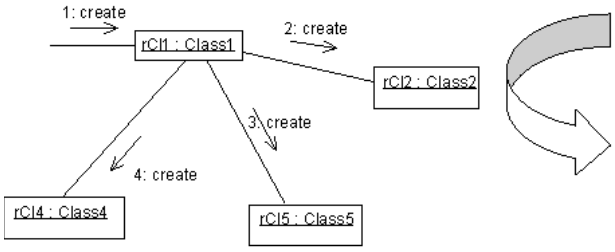
20.1.4 Implementação de Componentes

O objetivo desta sub-etapa é bem específico. Basicamente consiste na implementação das classes de design na forma de componentes. Dependendo-se da linguagem de programação utilizada, o código fonte pode estar distribuída em um ou ou mais arquivos. Dependendo-se do número de classes, poderia-se construir um diagrama de componentes para representar o mapeamento entre as classes de design e os arquivos de código-fonte. Entretanto, como o número de classes e arquivos tende a ser longo, é mais útil, na maioria das vezes, se ter uma lista em texto fazendo esse mapeamento, do que os diagramas de componentes. A geração do código fonte é feita a partir das classes de design e dos contratos que foram desenvolvidos na fase de design. Em algumas situações, os esqueletos de código podem ser gerados automaticamente por meio das ferramentas de modelagem. Um exemplo de geração de esqueleto de classe, utilizando diagramas de classe desenvolvidos na fase de design é dado na figura a seguir:




```
public class Class3 {
     int i = 0;
     float method (float i, float j)
          { ...
          }
     }

public class Class1 extends Class3 {
     int a = 5;
     float b = 3.4;
     Class2 rCl2;
     Class4 rCl4;
     Class5 rCl5;
     }

public class Class2 {
     int i = 0;
     void opera(int k)
          { ...
          }
     }

public class Class4 {
     int att;
     void foo()
          { ...
          }
     }

public class Class5 {
     int wop;
     Class1 rCl1;
     void init()
          { ...
          }
     }
```

Após a geração dos esqueletos de código das classes, procedemos à implementação das operações das classes de design em termos de métodos. Alguns tipos de operações podem também ser geradas automaticamente, sendo complementadas com edição posterior. Entretanto, a maioria das ferramentas CASE ainda não disponibiliza esse recurso. A implementação das operações das classes deve ser feita baseada nos diagramas de comunicação desenvolvidos no design, bem como a partir dos contratos também lá desenvolvidos. Um exemplo da geração de código baseada em diagramas de comunicação é dada a seguir:




```
public class Class1 extends Class3 {
     int a = 5;
     float b = 3.4;
     Class2 rCl2;
     Class4 rCl4;
     Class5 rCl5;
     void Class1()
          {rCl2 = new Class2();
           rCl5 = new Class5();
           rCl4 = new Class4();
          }
     }
```

Em seguida, procede-se à verificação de que o componente provê a mesma interface que a classe de design implementa. Por fim, eventualmente, ocorrerão os consertos de defeitos de implementações anteriores, caso a classe já tenha sido implementada em outras iterações anteriores.

20.1.5 Realizar o Teste de Unidade do Componente

O objetivo desta etapa é a realização de um teste individual do componente implementado, independente de sua posterior integração com outras classes. Basicamente, deve-se testar todas as operações de todas as classes.

Como na prática uma classe sempre dependerá de outras classes, a estratégia para viabilizar a realização dos testes de unidade é se criar "Stubs", ou seja, classes simplificadas com o mesmo nome das classes finais com as quais o componente será integrado posteriormente, que provêm uma simulação do comportamento da classe. A classe a ser testada é então integrada a esses "Stubs" para que possa ser testada. Além dos "Stubs", é necessário também prover as entradas dos parâmetros a serem enviados aos diferentes métodos das classes. Para isso, criam-se também classes especiais chamadas de "Drivers", que basicamente lêem os dados de teste de um arquivo de configuração e os enviam sistematicamente para serem testados.

Neste âmbito, diferentes tipos de testes podem ser realizados. Basicamente, temos o teste de especificação, o teste estrutural e eventualmente outros testes.

O teste de especificação visa verificar o comportamento observável externo da unidade, sem entrar no mérito de como esse comportamento é implementado. Ele focaliza portanto, nas entradas e saídas da unidade.

O teste estrutural verifica a implementação interna da unidade, fazendo com que cada trecho do código seja executado. Outros testes podem envolver testes de desempenho, uso de memória, escalabilidade e capacidade.

Observe que até este ponto, não temos ainda uma implementação funcional de nosso protótipo, mas tão somente um conjunto de classes, testadas separadamente. O release do sistema somente se consolidará, portanto, após a etapa final de testes, quando o sistema será integrado e uma construção (build, ou release) será gerado.

20.1.6 Integração do Sistema

O principal objetivo desta atividade é a geração de um build do sistema, após a integração de todas as classes. Para isso, é necessário antes a criação de scripts de compilação e linkagem, verificando que todos os componentes designados sejam incluídos. Deve-se identificar as versões corretas de cada componente que constituirá esse build, e deve-se então proceder à compilação e linkagem de todos os componentes. Após a compilação, deve-se verificar se o código todo compilou adequadamente, e em caso negativo, deve-se reverter a versão dos componentes que trouxeram problemas na compilação, de tal forma que se garanta que no final uma versão compilável está disponível para testes. Uma vez que essa versão esteja disponível, deve-se atualizar o número de versão com os componentes novos, e se fazer um update do sistema de controle de versão de forma a registrar essa nova versão. Por fim, deve-se arrolar sistematicamente a versão de cada um dos componentes utilizados para compor a versão compilável, de tal forma que os testes estejam associados a ela. Ao final desta fase, deve-se possuir uma versão compilável do software, com seu código devidamente atualizando no sistema de controle de versão. Pode-se então passar à etapa seguinte, de testes.

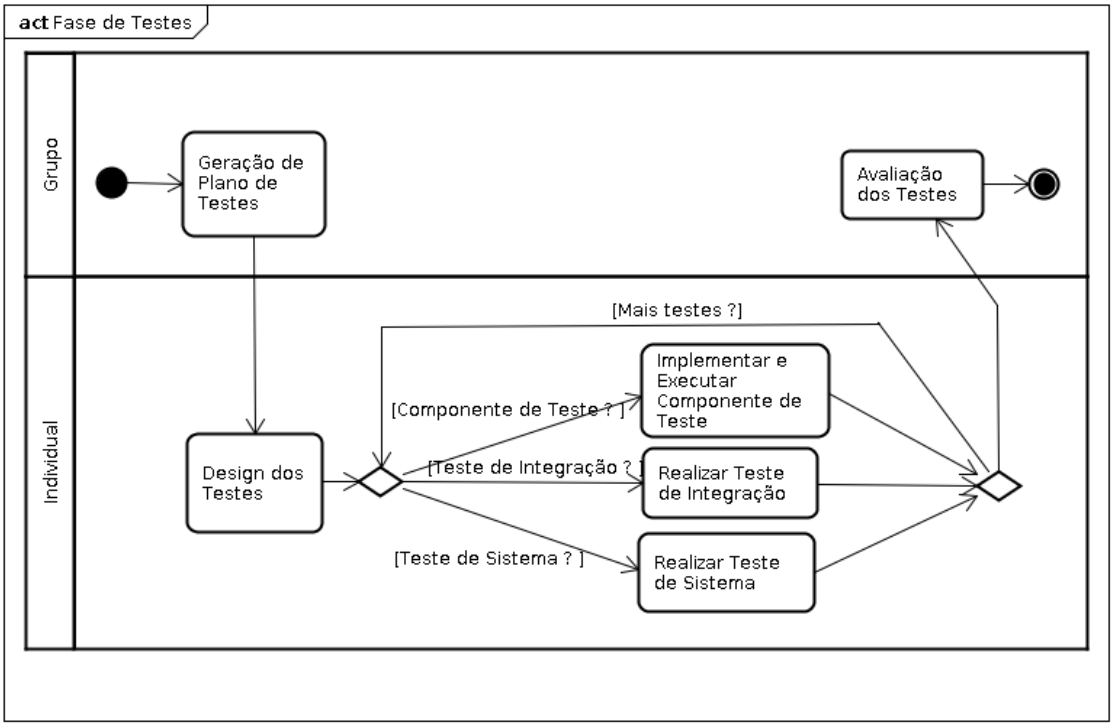
20.2 A Etapa de Testes

A etapa de testes visa a verificação dos resultados da implementação por meio do teste de cada módulo do sistema e sua integração.

Os principais objetivos desta fase são os seguintes:

- planejamento dos testes a serem executados em cada iteração
- design e implementação dos testes por meio de casos de teste que especificam o que testar e quais os procedimentos a serem utilizados para os testes
- criação de componentes de teste executáveis para a automação de testes, quando possível
- avaliação sistematica dos resultados de cada teste e encaminhamento dos módulos defectivos para retrabalho

Um sumário da fase de testes é apresentado na figura a seguir:



20.2.1 Geração do Plano de Testes

O principal objetivo desta atividade é planejar os esforços de teste dentro de uma iteração, descrevendo uma estratégia de testes, estimando os requisitos para o esforço de teste, tais como recursos humanos e do sistema, e criando uma escala de testes a ser executada.

Assim, constrói-se um plano de testes, com base nos diversos modelos utilizados nas fases de especificação, análise, design e implementação.

Assim, desenvolve-se aqui uma estratégia geral de testes, onde determina-se que tipos de testes serão executados, como executá-los, quando executá-los e como avaliar os resultados dos testes. Devemos lembrar que testes custam recursos e tempo. Portanto, deve-se efetuar uma solução de testes que tenha uma boa relação custo-benefício.

Dois tipos de testes podem ser planejados: testes de integração e testes de sistema.

Os **testes de integração** são testes que visam verificar como os componentes integrados, passam a funcionar em conjunto. Os principais testes de integração são os seguintes:

- Testes baseados em Especificação (Testes Funcionais ou Testes de Caixa-Preta)
- Testes Estruturais (Testes Caixa-Branca)

Os **testes funcionais** visam testar as diferentes funcionalidades especificadas nos casos de uso. Assim, constróem-se diferentes cenários para cada caso de uso, e reproduz-se as ações realizadas pelo usuário em cada cenário, observando-se como o sistema reage, e comparando-se com a reação esperada, conforme a especificação dos casos de uso.

Os **testes estruturais** utilizam o conhecimento sobre a estrutura do código gerada, para tentar exercitar diferentes trechos de código. Complementando os testes estruturais implementados nos

testes de unidade, eles tentam verificar a integração entre múltiplos componentes e exercitar essa integração.

Os **testes de sistema** visam testar o sistema como um todo. Os mesmos devem ser realizados normalmente depois que os testes de integração acusaram um sistema mais estabilizado. Os testes de sistema são de uma natureza diversa dos testes de integração. Basicamente, realizam-se os seguintes testes de sistema:

- Testes de Instalação
- Testes de Configuração
- Testes Negativos (Testes de Segurança)
- Testes de Stress (Testes de Desempenho)

Testes de instalação verificam se o sistema pode ser instalado na plataforma do cliente e que o sistema opera corretamente após a instalação. Assim, todas (ou um conjunto representativo de) as plataformas em que o sistema pode ser operado devem ser testadas.

Testes de configuração verificam se o sistema é capaz de operar corretamente nas diferentes configuração em que um mesmo sistema possa aparecer. Assim, para cada diferente configuração do sistema, deve haver um teste específico.

Testes negativos tentam ocasionar falhas no sistema para revelar suas fraquezas. Assim, tenta-se utilizar o sistema de maneiras para as quais ele não tenha sido projetado, por exemplo, testando-se configurações de rede defeituosas, hardware insuficiente ou demanda de uso (carga) intensiva.

Os **Testes de stress** tentam verificar como o sistema se comporta quando os recursos disponíveis são insuficientes ou estão disponíveis. Diferenciam-se dos testes negativos, pois ao contrário destes, o objetivo não é fazer o sistema falhar, mas estudar a degradação da qualidade do sistema diante da insuficiência ou baixa disponibilidade dos recursos. Da mesma forma, estes testes podem verificar se o sistema atende aos requisitos nominais de capacidade, conforme os requisitos não-funcionais levantados.

Além dos testes de integração e dos testes de sistema, podemos ainda falar dos **testes regressivos**. Os testes regressivos refazem um eventual conjunto de testes já executados em uma versão anterior do mesmo componente (ou grupo de componentes), após modificações no componente terem sido implementadas. Como essas modificações podem fazer com que testes em que o componente antes funcionasse bem agora deixe de funcionar, é necessário que esses testes sejam refeitos. Os testes regressivos podem ser testes de integração ou testes de sistema que tenham sido executados em iterações anteriores, e que necessitam ser refeitos. Nem todos os testes precisam ser refeitos. Deve-se, nessa atividade, apontar quais os testes necessitam ser refeitos.

Para compreendermos as atividades da fase de testes, é necessário ainda compreendermos alguns conceitos importantes que são os conceitos de casos de teste, procedimentos de teste e componentes de teste.

Casos de teste correspondem, de certa forma a casos de uso, só que agora não mais de uso do sistema, mas com finalidades de teste. Assim, um caso de teste especifica um cenário de teste, incluindo o que testar, com que entradas e com quais resultados esperados. Casos de teste podem estar associados a um caso de uso ou à realização do caso de uso no design.

Procedimentos de teste especificam como realizar um ou mais casos de teste.

Por fim, **componentes de teste** automatizam um ou mais procedimentos de teste.

Esses componentes de teste podem ser desenvolvidos tanto por linguagens de script ou por meio de linguagens de programação convencional.

As atividades práticas dessa fase são basicamente definir quais os testes serão realizados na presente iteração. Esses testes devem ser definidos e atribuídos aos responsáveis por desenvolver um projeto para eles. Dependendo-se do tipo de teste, deve-se indicar quais os casos de uso serão testados e quem será responsável por cada caso de uso. Deve-se normalmente utilizar formulários padronizados para cada tipo de teste, que devem ser preenchidos durante o design dos testes. Deve-se ainda, definir o esforço em tempo e pessoal a ser empregado nas atividades de teste.

O resultado final dessa fase é uma lista com os diferentes testes a serem projetados, devidamente alocados ao pessoal que deverá desenvolvê-los.

20.2.2 Design dos Testes

Os objetivos básicos desta atividade são:

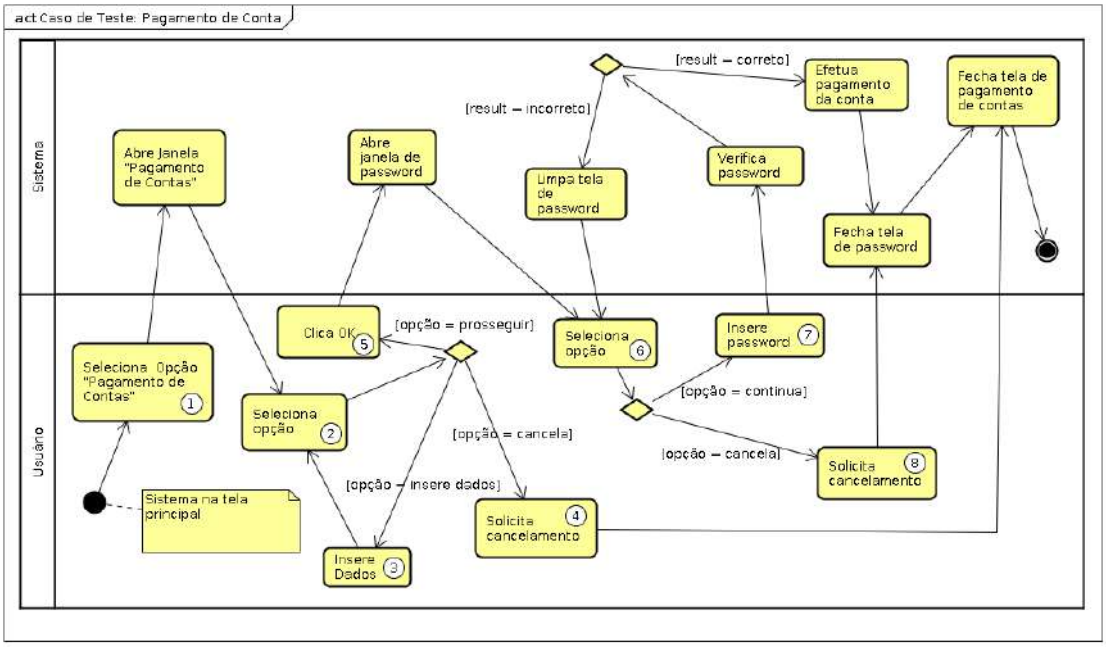
- identificar e descrever casos de teste para cada módulo
- identificar e estruturar procedimentos de teste especificando como realizar os casos de teste

Devemos iniciar pelos casos e procedimentos de teste referentes aos testes de integração, passando posteriormente aos testes de sistema, seguindo para os testes regressivos (aproveitando casos de teste de iterações anteriores), onde devemos seguir identificando e estruturando procedimentos de teste. A regra geral a ser seguida aqui é que deve-se utilizar um conjunto de testes que requeiram um mínimo esforço, dado um plano de testes. Ou seja, devemos evitar ao máximo a existência de *overlaps* entre casos de teste.

Para projetar um teste funcional, deve-se adotar o seguinte procedimento:

- Para cada caso de uso atribuído
    - Numerar cada ação do usuário no diagrama de atividades referente ao respectivo caso de uso.
    - Gerar diferentes cenários, na forma de sequências de ações exercitando diferentes caminhos no diagrama de atividades

Observe-se que podem existir múltiplos critérios para a geração de caminhos nesse caso. Algum critério deve ser definido, como por exemplo um critério do tipo "todos os nós", ou um critério do tipo "todos os caminhos". Veja o exemplo apresentado na figura a seguir:



Observe que cada ação de parte do usuário foi numerada de 1 a 8. Com isso, seguindo-se o diagrama de atividades, diversos cenários podem ser construídos:

1-2-3-2-5-6-7
1-2-4
1-2-3-2-4
1-2-5-6-7
1-2-3-2-3-2-3-2-3-2-3-2-3-2-3-2-4

Cada um destes é um cenário de teste.

Para projetar um teste negativo, deve-se verificar as classes de equivalência para os dados fornecidos nos campos das interfaces. Essas classes podem ser de início, dados válidos e dados inválidos. Deve-se então propor testes utilizando-se exemplares de cada classe de equivalência. Principalmente para dados inválidos, deve-se exercitar dados com grande potencial causador de problemas, tais como caracteres acentuados, strings com espaços em branco, caracteres de controle, etc. O teste projetado deve incluir não somente os dados de entrada, mas também o resultado esperado para cada entrada.

20.2.3 Implementação e Execução de Componente de Teste

O principal objetivo desta fase é automatizar os procedimentos de teste por meio da criação de componentes de teste, se possível. Devemos entretanto lembrar-nos de que nem todos os procedimentos de teste podem ser automatizados. A automação de testes pode ser a única maneira pela qual certos tipos de testes podem ser implementados.

Como deveríamos proceder, caso quiséssemos efetivar esta etapa ?

Inicialmente componentes de teste são criados utilizando os procedimentos de teste como entrada. Quando se utiliza uma ferramenta de automação, um conjunto de ações são previamente criadas para que o módulo em questão seja testado. A própria ferramenta se encarrega de criar os

componentes de teste. Quando estamos programando os componentes de teste explicitamente (sem o uso de uma ferramenta de automação), devemos utilizar os procedimentos de teste como uma forma de especificação para que os componentes de teste sejam desenvolvidos. Entretanto, isso só ocorre em casos muito especiais, quando o teste do sistema pode ser vital (por exemplo, em sistemas de missões espaciais, controles de submarinos ou outras atividades onde a vida humana esteja em risco).

20.2.4 Realização de Teste de Integração

O grande objetivo desta atividade é realizar os testes de integração especificados para cada módulo do sistema.

Uma sequência de testes de integração visa realizar os testes de integração relevantes por meio da execução manual dos procedimentos de teste para cada caso de teste ou por meio de algum componente de teste disponível para o procedimento de teste em questão. Em seguida, procede-se à comparação dos resultados com os resultados esperados e a discriminação dos casos que apresentaram resultados inesperados. Deve-se então relatar os defeitos encontrados ao engenheiro de componentes, para que este possa corrigir os componentes defeituosos. Por fim, deve-se relatar os defeitos ao engenheiro de testes, para que este possa estimar os resultados finais da sequência de testes.

20.2.5 Realização de Teste de Sistema

O grande objetivo desta etapa é realizar os testes de sistema especificados a cada iteração, capturando os resultados do teste. Testes do sistema visam testar o funcionamento do sistema como um todo integrado.

O teste de sistema se inicia quando os testes de integração indicam que o sistema atende as metas de qualidade quanto a integração de seus componentes para a iteração corrente (e.g. 95% dos testes de integração executam com resultado esperado). Estes testes são realizados de maneira análoga aos testes de integração (i.e. seguindo-se os procedimentos de testes desenvolvidos durante o design de testes).

20.2.6 Avaliação dos Testes

O principal objetivo desta etapa é a avaliação dos esforços de teste para a iteração corrente. Essa avaliação compreende basicamente a comparação dos resultados com as metas especificadas no planejamento de testes, bem como a criação de métricas que determinem o nível de qualidade do software, especificando maiores testes que necessitem ser realizados.

Diversas métricas poderiam ser sugeridas. Como exemplos de métricas, temos as chamadas métricas de completude e as métricas de confiabilidade.

Métricas de completude são derivadas da cobertura dos casos de teste e dos componentes de teste. Indicam basicamente qual a porcentagem de casos de teste que foram executados e qual a porcentagem de código que foi testada.

Métricas de confiabilidade são baseadas na análise dos defeitos descobertos, com relação aos casos

que tiveram um resultado como esperado.

21. Bibliografia

(Jacobson et.al. 1999) Ivar Jacobson, Grady Booch, James Rumbaugh - **The Unified Software Development Process** - Addison Wesley, 1999.

(Larman 1998) Craig Larman - **Applying UML and Patterns - An Introduction to Object Oriented Analysis and Design** - Prentice Hall Inc., New Jersey 1998.

[OMG 2015] OMG Unified Modeling Language TM (OMG UML) **Version 2.5**
http://www.omg.org/spec/UML/2.5/PDF