Implementação e Testes no Processo Unificado

Ricardo R. Gudwin
DCA-FEEC-UNICAMP
09/11/2010

Introdução

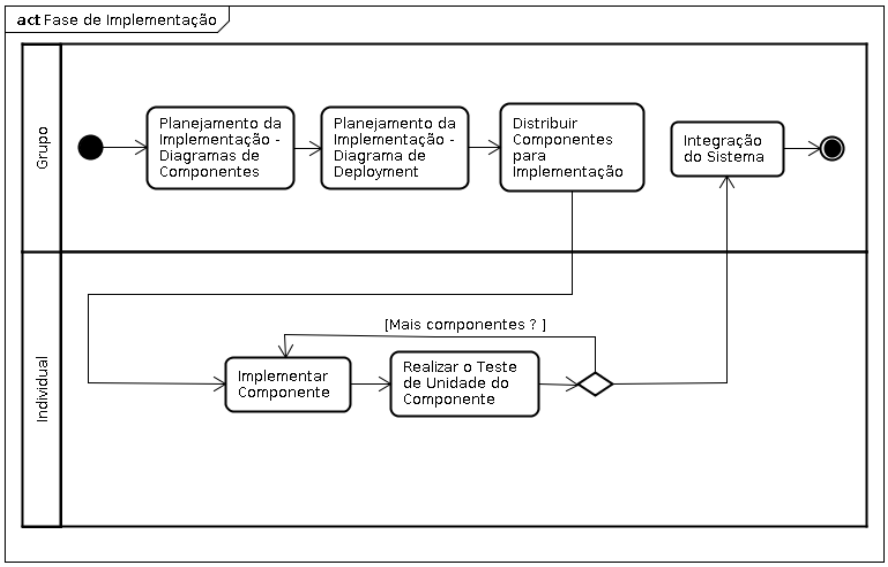
Normalmente, em sistemas de grande porte, o desenvolvimento da implementação e dos testes é distribuído dentre diversos profissionais, que se encarregam de desenvolver de maneira isolada as partes ou componentes que integram o sistema, sendo que estas partes depois devem ser integradas e testadas por 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.

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:



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.

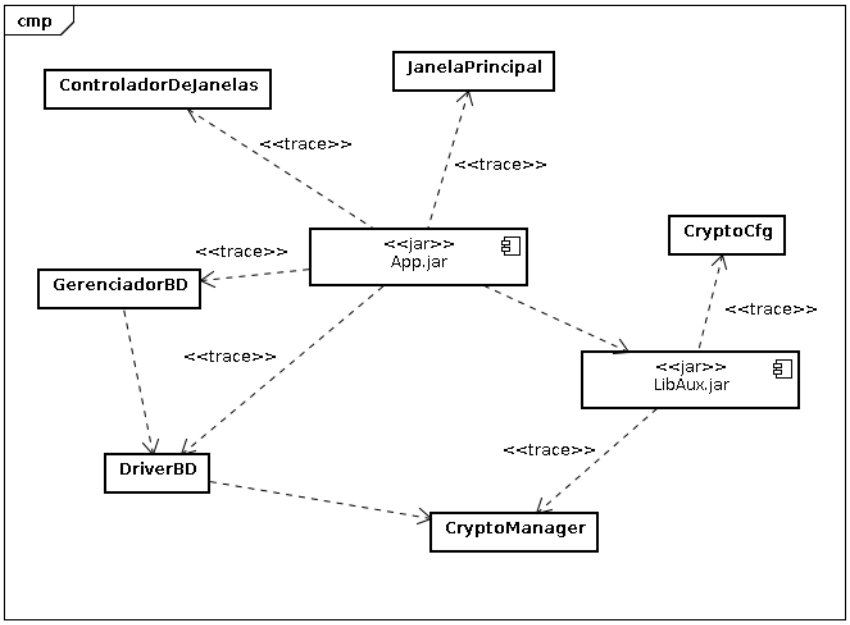
Durante o planejamento da construção, deve-se verificar eventuais implementações de iterações anteriores e fazer o reuso das partes desta que já estão consolidadas. Isso garante um caráter incremental ao desenvolvimento.

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".

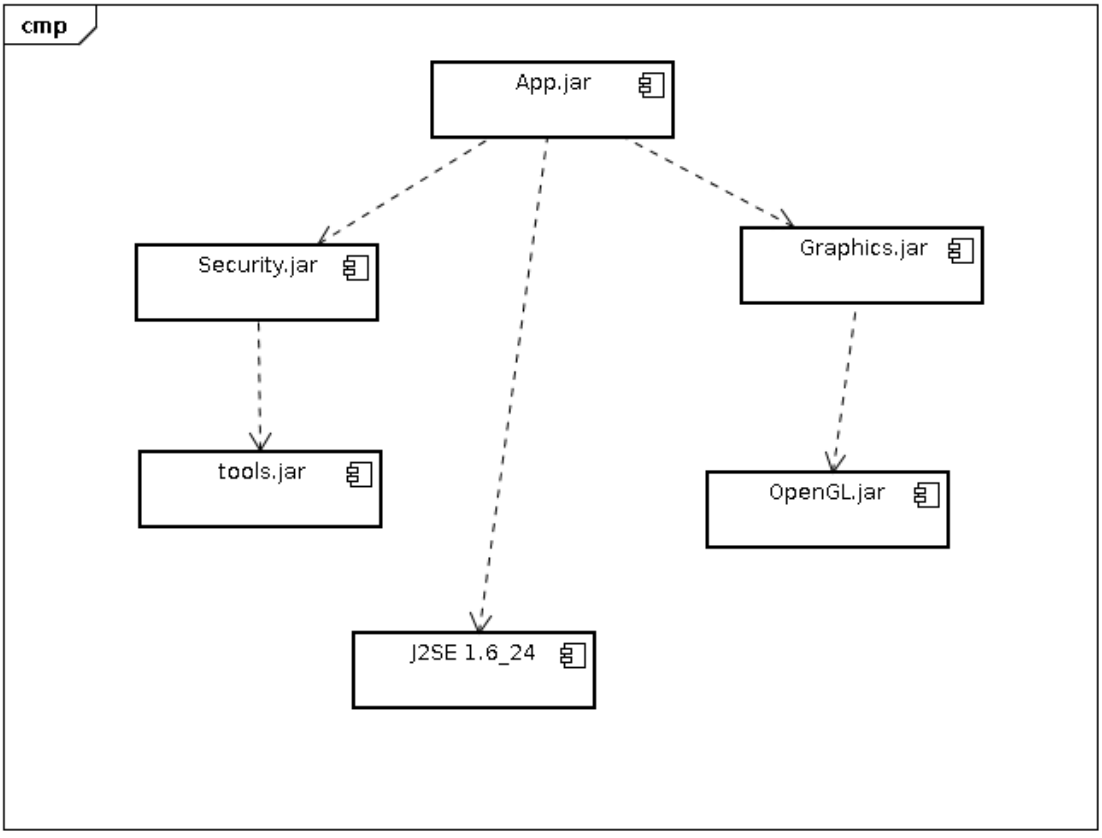
Um exemplo de um "Diagrama de Constituição de Componentes pode ser visto na figura a seguir.



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.

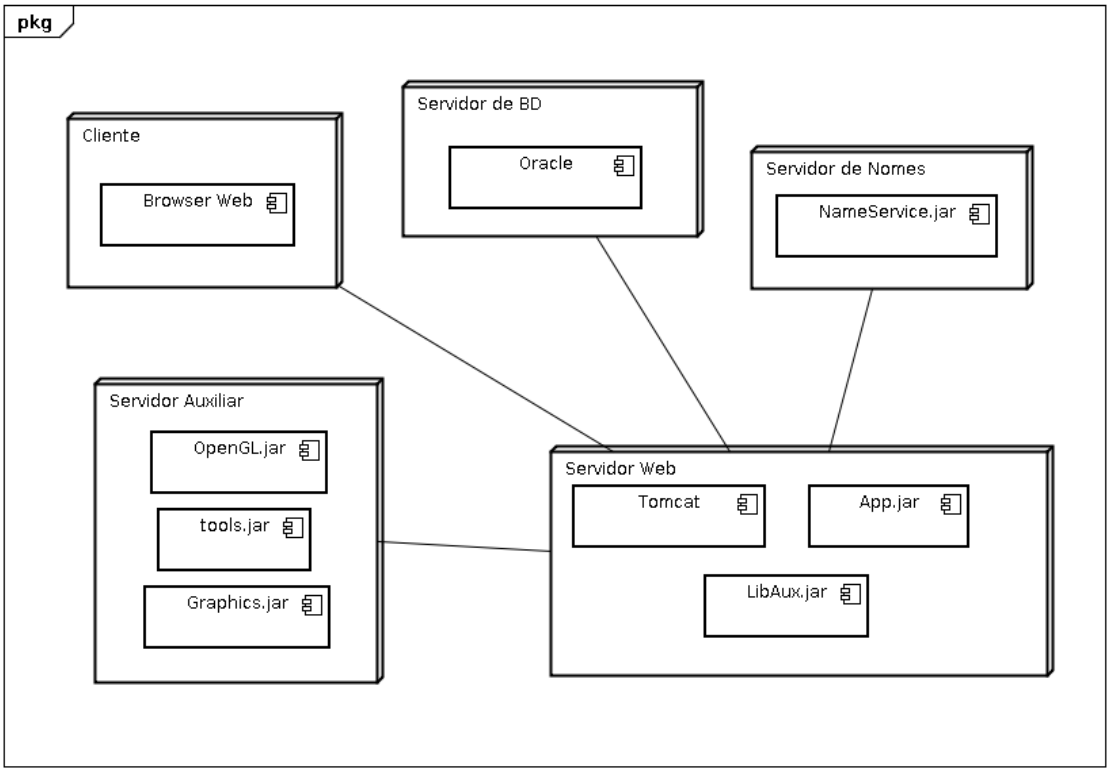
Um exemplo de um "Diagrama de Dependências Externas" pode ser visto na figura a seguir:



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.

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.

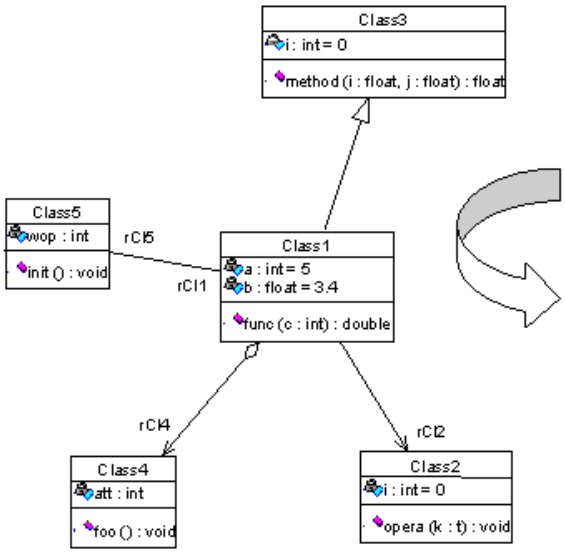
Em algumas situações, múltiplos diagramas de deployment podem ser necessários. Esse é o caso, por exemplo, quando existem múltiplas configurações de instalação possíveis, ou quando se deseja especificar configurações especiais para testes. Nesses casos, um diagrama de deployment diferente deve ser construído 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.

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.

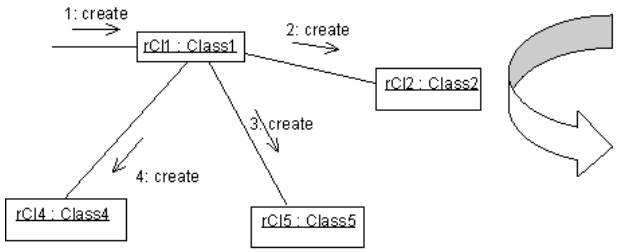
Implementação de Componentes

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.

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.

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.

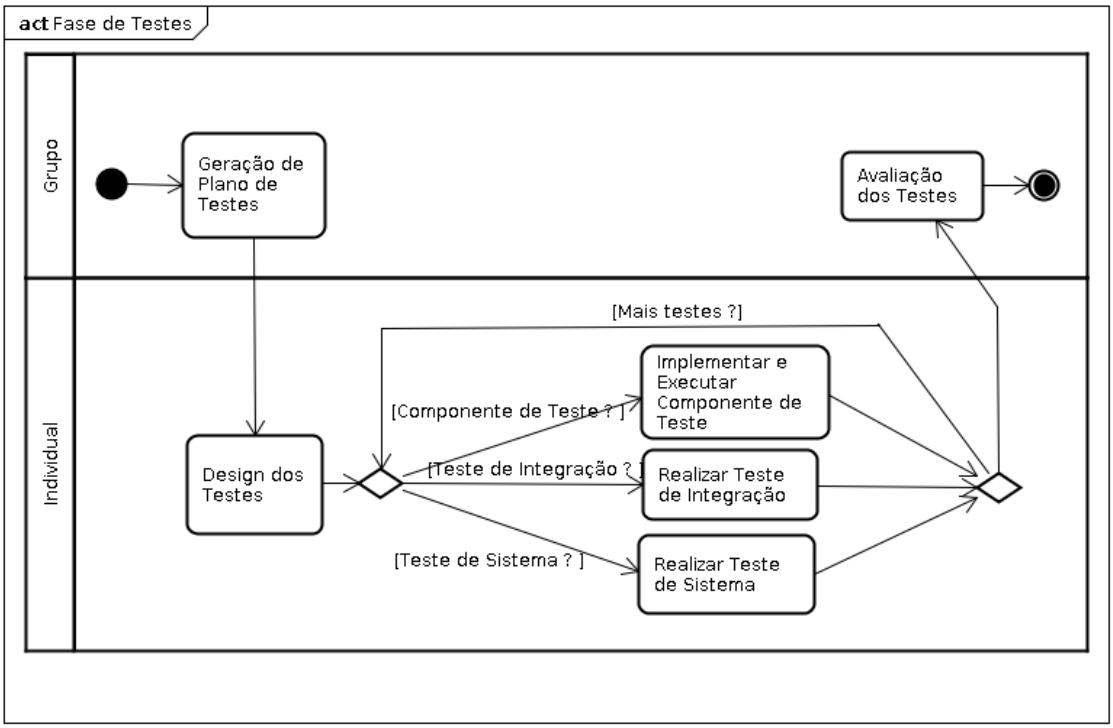
A Etapa de Testes

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:



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.

Design dos Testes

Os objetivos básicos desta atividade são:

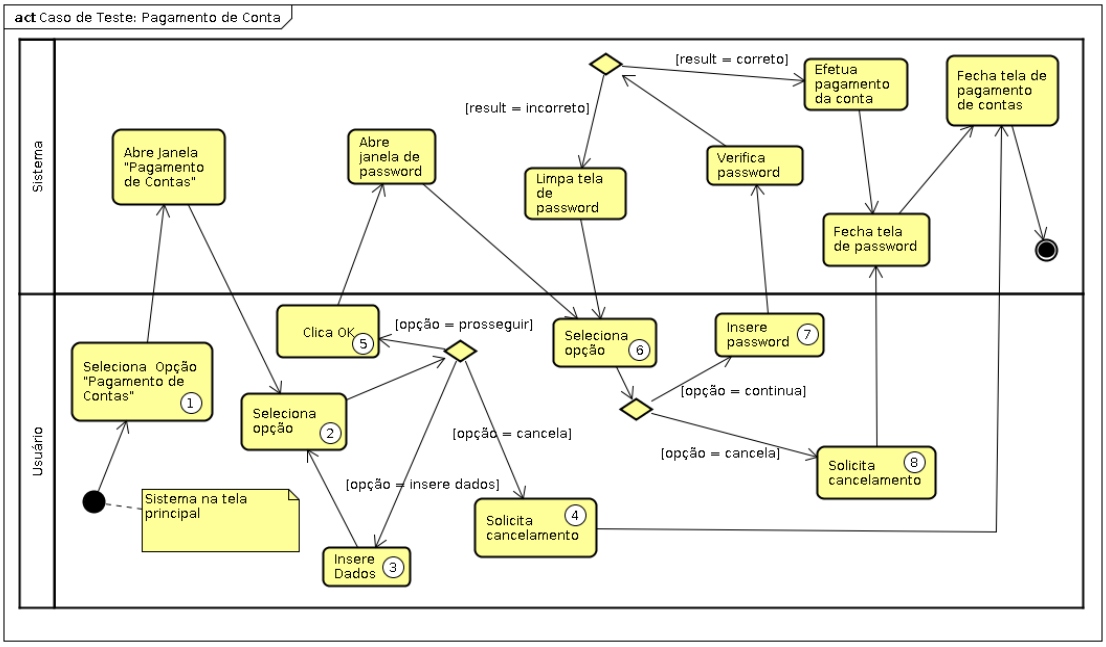
- identificar e descrever casos de teste para cada módulo
- identificar e estruturar procedimentos de teste especificando como realizar os casos de teste

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.

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).

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.

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).

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.