UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"

Pós-Graduação em Ciência da Computação

Álvaro Ferraz d'Arce

Avaliação da ferramenta de visualização de software $SoftVis_{OAH}$ como apoio à depuração de programas: um experimento controlado

UNESP
2012

Álvaro Ferraz d'Arce

Avaliação da ferramenta de visualização de software $SoftVis_{OAH}$ como apoio à depuração de programas: um experimento controlado

Orientador: Prof. Dr. Rogério Eduardo Garcia

Dissertação de Mestrado elaborada junto ao Programa de Pós-Graduação em Ciência da Computação - Área de Concentração em Sistemas de Computação, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

UNESP
2012

Álvaro Ferraz d'Arce

Avaliação da ferramenta de visualização de software $SoftVis_{OAH}$ como apoio à depuração de programas: um experimento controlado

Dissertação apresentada para obtenção do título de Mestre em Ciência da Computação, área de Sistemas de Computação junto ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Campus de São José do Rio Preto.

BANCA EXAMINADORA

Prof. Dr. Rogério Eduardo Garcia
Professor Assistente Doutor
UNESP–São José do Rio Preto
Orientador

Prof^{a}. Dr^{a}. Sandra Fabbri
Professora Doutora
Universidade Federal de São Carlos

Prof. Dr. Danilo Medeiros Eler
Professor Assistente Doutor
UNESP–Presidente Prudente

São José do Rio Preto, 08 de Agosto de 2012

Dedico este trabalho
à Márcia e à Vitória

Agradecimentos

Agradeço, em primeiro lugar, a Jesus Cristo, que me proporcionou este mestrado e me sustentou em todo o tempo. Obrigado, Senhor.

Agradeço à minha esposa, Márcia, que me apoiou em minha decisão de iniciar este trabalho e me deu forças durante a realização deste por meio de suas orações. Obrigado, querida.

Agradeço à minha filhinha, Vitória, que foi compreensiva durante cada momento em que não pude brincar com ela devido às atividades do mestrado. Obrigado, milagre de Deus.

Agradeço a meus pais, que me deram apoio moral e financeiro durante esta fase de minha vida. Obrigado, pai e mãe.

Agradeço a meu orientador, Rogério, pelo auxílio e direcionamento dedicados a mim. Obrigado, amigo.

Agradeço a meus companheiros da sala de permanência da pós-graduação da Unesp de Presidente Prudente – à turma de Ciência da Computação, à turma de Matemática e à turma de Estatística – em especial à Lilian, à Fernanda, à Vanessa, ao João e ao Jorge por terem participado dos experimentos, e ao Clóvis por ter me ajudado com as análises estatísticas. Obrigado, amigos.

Agradeço aos alunos do curso de graduação de Bacharelado em Ciência da Computação da Unesp de Presidente Prudente que foram voluntários na participação do experimento. Obrigado a todos.

Agradeço a todos que, direta ou indiretamente, me apoiaram e me auxiliaram no decorrer deste trabalho. Obrigado a todos vocês.

Agradeço também à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pelo apoio financeiro para a realização deste trabalho. Obrigado.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Resumo

Com o aumento da complexidade estrutural de sistemas de software, tarefas como a Compreensão de Programa tornam-se mais difíceis de serem realizadas. Quando se trata de Programas Orientados a Aspectos, a compreensão pode se tornar mais complexa em decorrência de suas novas características – algumas unidades de código interferem no comportamento de outras. Meios alternativos para a auxiliar as atividades de Compreensão de Programa podem ser utilizados, como a Visualização de Software – representações visuais podem ser usadas para prover apoio à compreensão, principalmente por permitir que o usuário interaja com as representações gráficas. Apesar de poder apoiar as tarefas de compreensão, o uso de uma ferramenta de Visualização de Software pode, também, aumentar a dificuldade do processo, pois, além de depender da interpretação das representações visuais, exige o domínio da ferramenta, a sua adequação à tarefa, o conhecimento de técnicas de visualização e a seleção e análise de quais técnicas se aplicam à tarefa de compreensão em questão. O grupo de pesquisa da FCT-UNESP vem desenvolvendo uma ferramenta de Visualização de Software – $SoftVis_{OAH}$ –, e para obter evidências de vantagens do uso de tal ferramenta, é crucial a sua avaliação. Assim, este trabalho tem por objetivo a avaliação da ferramenta de Visualização de Software $SoftVis_{OAH}$ para se obter evidências da eficácia e da eficiência de suas representações visuais aplicadas em Compreensão de Programas Orientados a Aspectos no contexto de depuração de programas.

Abstract

With the increasing structural complexity of software systems, tasks such as Program Comprehension become difficult to perform. In the case of Aspect-Oriented Programs, the comprehension can become more complex due to its new features – some units of code interfere in the behavior of others. Alternative means for applying Program Comprehension techniques can be used, as Software Visualization – visual representations can be used to provide comprehension aid, specially for allowing the user to interact with graphical representations. Although it can support the program comprehension task, the use a Software Visualization tool can, also, increase the difficulty of the process, because, besides depending on the interpretation of visual representations, it requires mastery of the tool, their suitability to the task, knowledge of visualization techniques and the selection and analysis of which techniques apply to the comprehension task in question. The research group of FCT-UNESP has developed a Software Visualization tool – $SoftVis_{OAH}$ –, and to obtain evidences of the advantages of using such a tool, it is critical its evaluation. Thus, this study aims to assess the $SoftVis_{OAH}$ tool, about the effectiveness and efficiency of its visual representations applied in Aspect-Oriented Programs Comprehension in the context of programs debugging.

CAPÍTULO

1

Introdução

1.1 Contexto e Justificativa

Segundo Karahasanovic et al. (2007), em geral, tarefas de Compreensão de Programa permanecem incompletas, requerendo então um entendimento mais aprofundado das estratégias empregadas em tais tarefas, além disso, entender mecanismos subjacentes pode melhorar a compreensão. Adquirir conhecimento sobre estrutura de grandes programas pode retardar as tarefas de Compreensão de Programa, o que motiva o uso de abordagens alternativas para apoiar tal tarefa.

O projeto de um Software Orientado a Aspectos define, entre outras características, a separação de interesses, fornecendo suporte à modularização de interesses transversais por meio de mecanismos que tornam possível a adição de comportamento a elementos selecionados da semântica de linguagem de programação, isolando assim a implementação, que de outra maneira, seria espalhada e entrelaçada por todo o código-base (Kiczales et al., 2001). Essa definição de comportamento e de separação de interesses pode seguir diversas metodologias, abordando ou não as recentes técnicas de Engenharia de Software Orientada a Aspecto, como em Ramos et al. (2008), Araújo et al. (2005) e Filman et al. (2005).

Entretanto, a Programação Orientada a Aspectos (*POA*) pode aumentar a complexidade estrutural dos sistemas de software devido às suas características (código posteriormente entrelaçado e espalhado pelo processo de *weaving*), dificultando a compreensão de sua arquitetura e estrutura, trazendo à tona desafios à Compreensão de Programas Orientados a Aspectos.

Filman et al. (2005) relatam exemplos de sua experiência na área de *POA* que comprovam essa situação, por exemplo: o projetista fornece uma especificação ao programador com definição de separação de interesses entrecortantes e o programador muda o projeto da aplicação para introduzir a separação de interesses, levando a consequências negativas de incoerência entre a implementação e o projeto original, possibilidade de incompatibilidade entre módulos do sistema devido à implementação de *aspectos*, dificuldade de manutenção e evolução do sistema

e diminuição de produtividade.

Pela fatoração do projeto, determinadas funcionalidades do programa têm parte de seu código mapeada em *aspectos*, resultando em uma fragmentação das *classes* envolvidas, tornando-os em composições de *classes* e *aspectos*. Dependendo do nível da fragmentação, a complexidade estrutural do sistema de software pode ser influenciada. Uma validação é necessária para verificar se a fatoração do projeto em *classes* e *aspectos* foi implementada corretamente, de maneira que a implementação seja coerente com o projeto.

A Visualização de Software pode ser utilizada como uma abordagem alternativa, auxiliando Engenheiros de Software a lidar com a complexidade estrutural, apoiando tarefas de Compreensão de Programa por meio de representações visuais, permitindo interagir com tais representações gráficas e provendo auxílio ao processo de entendimento. Uma ferramenta de Visualização de Software fornece suporte ao usuário para que este obtenha informações por meio de análises dos artefatos visuais disponíveis, provendo, desta maneira, auxílio ao processo de entendimento de programas.

Uma ferramenta de Visualização de Software pode auxiliar o processo de validação de *classes* e *aspectos*, tendo suas representações visuais adequadas às características da *POA*, ou seja, o código entrelaçado e espalhado gerado pelo *weaver*. Tais representações visuais tornam possíveis a visualização de estruturas habituais da Programação Orientada a Objetos (*pacotes*, *métodos* e *classes*) juntamente com as estruturas da Programação Orientada a Aspectos (*aspectos*, *adendos* e como *aspectos* entrecortam *métodos*).

1.2 Motivação e Formulação do Problema

Um grupo de pesquisa da FCT-UNESP vem desenvolvendo uma ferramenta de visualização de software – $SoftVis_{OAH}$ –, que precisa ser avaliada. A ferramenta aborda um mapeamento visual baseado em um modelo de coordenação, proposto para visualizar Programas Orientados a Aspectos, conforme ilustrado na Figura 1.1. A *Apresentação Estrutural* visa a exibir como o código fonte está organizado em *classes* e *aspectos*; A *Apresentação Inter-Unidades* objetiva exibir dependências entre unidades e entrecortes de *aspectos*; A *Lista de Métodos/Adentos* exibe os métodos e adendos das respectivas *classes* e *aspectos* selecionados; E a *Apresentação Intra-Método* objetiva exibir o comportamento intra-método após o processo de *weaving*. Pode-se observar na Figura 1.1 que as visualizações são coordenadas para permitir a exploração de diferentes níveis de visão. Cada apresentação possui um conjunto adequado de técnicas: por exemplo, há várias técnicas para visões hierárquicas, e foi escolhida a *Treemap* para representar a estrutura hierárquica do programa incluindo *aspectos* e *adendos* (Apresentação Estrutural). A visão *Hiperbólica* foi selecionada para representar dependências de *classes* acopladas com entrecortes de *aspectos* (Apresentação Inter-Unidades), e o *Grafo de Fluxo de Controle* (*GFC*) para visualizar o código fonte adendado representando o comportamento de uma parte do código, exibindo os *adendos* sobre o código após o processo de *weaving* (Apresentação Intra-Método). Pela coordenação dessas três representações visuais, um Programa Orientado a Aspecto pode ser visualizado em estrutura hierárquica, dependência de *classes*, entrecortes de

aspectos, código entrelaçado e código espalhado.

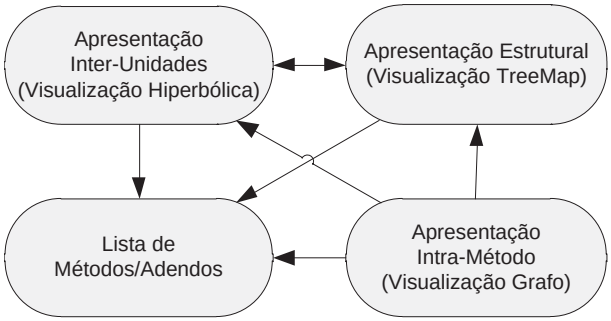


Figura 1.1: Esquema de Apresentações

Uma ferramenta de Visualização de Software – $SoftVis_{OAH}$ – tem sido desenvolvida, também pelo grupo de pesquisa desta Universidade, para implementar a abordagem de mapeamento visual baseada no modelo de coordenação exposto, com o propósito de visualizar programas Orientados a Objetos e Orientados a Aspectos (d'Arce et al., 2011, 2012). Adicionalmente, a ferramenta provê a visualização de resultados de testes estruturais (testes de *caixa branca*) e visualização de *bytecode*.

O uso da $SoftVis_{OAH}$, assim como de qualquer novo método e nova técnica, precisa ser avaliado. É preciso ter evidências que suas funcionalidades são adequadas ao artefato em questão – Programa Orientado a Aspectos – para auxiliar o entendimento do código entrelaçado e espalhado. O processo de visualização proposto por Card et al. (1999) trata de coleta de dados, sua organização e mapeamento em estruturas visuais para gerar representações visuais. Seguindo tal processo, é necessário que a ferramenta $SoftVis_{OAH}$ não possua apenas um mapeamento visual específico para Programas Orientados a Aspectos, mas também mecanismos para coletar e organizar os dados que representem o código entrelaçado e espalhado obtido após o processo de *weaving* da *POA*. Entretanto, ainda não havia evidência da eficácia e da eficiência no uso da $SoftVis_{OAH}$, o que motivou o presente trabalho, cujo objetivo é exposto a seguir.

1.3 Objetivo e Metodologia

O presente trabalho tem como objetivo geral avaliar a utilização da ferramenta $SoftVis_{OAH}$ no apoio ao entendimento de Programas Orientados a Aspectos, em comparação com o entendimento de código convencional (*ad hoc* – sem o auxílio da ferramenta). Como objetivos específicos, têm-se: verificar se a ferramenta auxilia na localização de defeitos (tarefa de informar onde os defeitos estão localizados no código fonte) e se auxilia a compreensão (abstração estrutural e entrecorte de *aspectos*) de um Programa Orientado a Aspectos.

Para que a ferramenta $SoftVis_{OAH}$ seja avaliada e para obter indícios de suas vantagens (como apoio a tarefas de Localização de Defeitos e Compreensão de Programa), um estudo experimental controlado foi conduzido de acordo com o Processo de Experimentação proposto por Wohlin et al. (2000), o que define a metodologia empregada neste trabalho. Adicionalmente, um requisito para o experimento realizado é permitir ao participante explorar os recursos disponibilizados pela ferramenta.

Com a condução do experimento, os participantes puderam fornecer, como *feedback*, não apenas medições (dados coletados) que permitiram derivar métricas quantitativas relativas ao uso da ferramenta, mas também uma avaliação qualitativa da mesma. Após a condução do experimento e avaliação dos resultados, foi possível constatar estatisticamente que a $SoftVis_{OAH}$ proveu auxílio às tarefas de Localização de Defeitos e Compreensão de Programa. Adicionalmente, algumas contribuições e lições aprendidas puderam ser obtidas para a condução de futuros experimentos.

1.4 Organização do Trabalho

Para expor o trabalho desenvolvido, esta dissertação encontra-se organizada como segue:

- No Capítulo 2 é apresentada uma revisão bibliográfica a respeito dos assuntos envolvidos na realização da pesquisa deste trabalho. São eles: Programação Orientada a Aspectos, Visualização de Informação, Visualização de Software e a ferramenta $SoftVis_{OAH}$.
- No Capítulo 3 é apresentada uma revisão bibliográfica a respeito da metodologia de pesquisa adotada, ou seja, Engenharia de Software Experimental e respectivo Processo de Experimentação.
- No Capítulo 4 é apresentada a pesquisa realizada com o objetivo de avaliar a ferramenta $SoftVis_{OAH}$, bem como os resultados obtidos.
- No Capítulo 5 são apresentadas as considerações finais deste trabalho, assim como propostas de trabalhos futuros.
- No Apêndice A são apresentados os perfis dos participantes do experimento, obtidos por meio de um questionário de caracterização.
- No Apêndice B são apresentados alguns documentos utilizados no experimento com a finalidade de coletar dados e obter *feedback* dos participantes.
- No Apêndice C é apresentado o material de treinamento para a realização das atividades da condução do experimento.

CAPÍTULO

2

Revisão Bibliográfica: Temas Relacionados

Neste capítulo é apresentada uma revisão bibliográfica sobre temas relacionados à motivação e à justificativa para a realização deste trabalho. Uma breve descrição de conceitos sobre Programação Orientada a Aspectos é apresentada para expor as características existentes em um programa que podem dificultar seu entendimento e que uma ferramenta de Visualização de Software deve tratar. Também é apresentada uma breve descrição sobre Visualização de Software, juntamente com o processo de Visualização de Informação para dar uma visão geral das tarefas envolvidas em visualização. A ferramenta $SoftVis_{OAH}$, objeto de estudo deste trabalho, foi desenvolvida e é apresentada – suas características, funcionalidades e projeções visuais. A explanação dos temas a respeito de Programação Orientada a Aspectos, Visualização de Informação e Visualização de Software é fundamental para entender o funcionamento da ferramenta.

2.1 Programação Orientada a Aspectos

Dijkstra (1976) introduziu o princípio da separação de interesses (*separation of concerns*), o qual divide o domínio do sistema de software em módulos menores, com o objetivo de entender cada módulo separadamente.

Kiczales et al. (2001) argumentaram a Programação Orientada a Objetos (*POO*) como uma tecnologia que pode auxiliar a Engenharia de Software pelo fato de o modelo de objeto subjacente fornecer um melhor ajuste aos problemas do domínio real. Mas eles apontaram problemas de programação, os quais as técnicas de *POO* não são totalmente adequadas para captar claramente as importantes decisões de projeto a respeito de diferentes funcionalidades implementadas em um sistema de software – alguns requisitos (normalmente não-funcionais) não podem ser claramente mapeados para unidades isoladas de implementação. Mecanismos para persistir objetos em uma base de dados relacional são exemplos dessas funcionalidades, comumente cha-

madas de interesses transversais, porque eles tendem a entrecortar (estarem mesclados) várias unidades de implementação (Elrad et al., 2001).

A Programação Orientada a Aspectos (*POA*) provê suporte à modularização de interesses transversais por meio de mecanismos que tornam possível a adição de comportamento a elementos selecionados da semântica da linguagem de programação, isolando assim o que de outra maneira estaria espalhado e entrelaçado por todo o código fonte (Kiczales et al., 2001).

Esses mecanismos consistem em definições de unidades de implementação (*aspectos*) que entrecortam as unidades do sistema (código base), provendo o comportamento esperado. Uma Linguagem Orientada a Aspectos genérica deve definir elementos específicos para que *aspectos* e código base sejam combinados. Elrad et al. (2001) apontaram os seguintes elementos: um modelo para descrever pontos do código base em que um comportamento adicional possa ser definido, chamado *join points*; um mecanismo para identificar esses *join points*; unidades que encapsulam as especificações de *join points* e melhorias de comportamento; e um processo para combinar código base e *aspectos*, chamado de processo de *weaving*. Os *aspectos* e como eles entrecortam um código base são elementos importantes para a Compreensão de Programa e devem ser mapeados em estruturas visuais.

Há várias extensões de linguagens para suporte de *POA*. Uma delas é o *AspectJ*, uma extensão da linguagem *Java*. No *AspectJ* são definidas novas construções para as características da *POA*. São elas:

- ***Aspectos***: São o ponto central da *POA*, consistindo em unidades que combinam especificações de *join points*, *adendos*, *declarações inter-tipo* e *compile-time detections*, além de poder conter *métodos* e variáveis.

- ***Adendos***: Consistem em estruturas que agrupam porções de código combinados em pontos específicos no programa. *Adendos after*, *before* e *around* são construções semelhantes a *métodos* que podem ser executados antes, depois e no lugar de *join points*, respectivamente.

- ***Join points***: Pontos no sistema no qual um comportamento adicional pode ser definido. Um determinado ponto da execução de um programa (ponto de mesclagem) no qual ocorre uma interação com um *aspecto*, podendo ser a chamada ou execução de um *método* ou *construtor*, a leitura ou atribuição do valor de uma variável de instância, a execução de um manipulador de exceção, a execução de um *adendo*, a inicialização estática de um campo, a inicialização de um objeto ou outros *join points*.

- ***Pointcuts***: Determinam se um determinado *join point* coincide com uma especificação. Reúne um ou mais *Join points* em um conjunto, especificando as regras de mesclagem.

- **Declarações inter-tipo**: Declaram membros – atributos e *métodos* – para serem possuídos por outros tipos. Modificam estruturas de *classes*, *interfaces* e *aspectos*, possibilitando adicionar variáveis ou *métodos*.

- ***Compile-time detections***: Permitem a definição de *warnings* (avisos) e *errors* (erros) em tempo de compilação.

O Programa Orientado a Aspectos invoca automaticamente, em tempo de execução, os *adendos* quando um *pointcut* coincide com o *join point*, definindo o comportamento entrecortante.

No Quadro 2.1 são mostradas partes de um código fonte de um Programa Orientado a Aspectos, contendo uma *classe* de conta bancária e um *aspecto* modificando-a por meio de dois *pointcuts* e dois *adendos* para registrar a transação após as chamadas aos *métodos saque()* e *deposito()*. O *weaver* do *AspectJ* transforma estaticamente o programa de modo que, em tempo de execução, ele se comporte de acordo com a semântica da linguagem, como exibido na Figura 2.1. Resumindo, essa aplicação permite registrar as operações de débito e crédito em uma conta bancária a partir de um terminal, e suas *classes* estão ilustradas na Figura 2.2.

Quadro 2.1: Porção de código fonte orientado a aspecto

```
1  public class Conta {
2      private float saldo;
3      public Conta(float saldo) {
4          this.saldo = saldo;
5      } //Conta.Conta
6      public float getSaldo() {
7          return saldo;
8      } //Conta.getSaldo
9      public void saque(float valor) {
10         saldo -= valor;
11     } //Conta.saque
12     public void deposito(float valor) {
13         saldo += valor;
14     } //Conta.deposito
15 } //class Conta
16
17 public aspect Registro {
18     pointcut saque(Conta con, float valor):
19         target(con) && args(valor) && call(* Conta.saque(float));
20     pointcut deposito(Conta con, float valor):
21         target(con) && args(valor) && call(* Conta.deposito(float));
22     after(Conta Conta, float valor) returning(): saque(Conta, valor) {
23         System.out.println("Valor do saque: " + valor);
24     } //Registro.after-returning
25     after(Conta Conta, float valor) returning(): deposito(Conta, valor) {
26         System.out.println("Valor do depósito: " + valor);
27     } //Registro.after-returning
28 } //aspect Registro
```

O *AspectJ* possui um pré-compilador denominado *Weaver* (tecelão), o qual combina as funcionalidades principais (geralmente regras de negócio) e secundárias (geralmente rotinas de apoio) dentro de um processo denominado *Weaving* (tecelagem). Esse processo produz um código *Java* puro, orientado a objetos, contendo as funcionalidades principais e secundárias entrelaçadas, conforme ilustrado na Figura 2.3.

As rotinas de apoio, anteriormente separadas das regras de negócio pela Orientação a Aspectos, encontram-se entrelaçadas ou misturadas às regras de negócio e distribuídas pelo código após o processo de *weaving*, conforme ilustrado na Figura 2.4. Após o *weaving*, o código

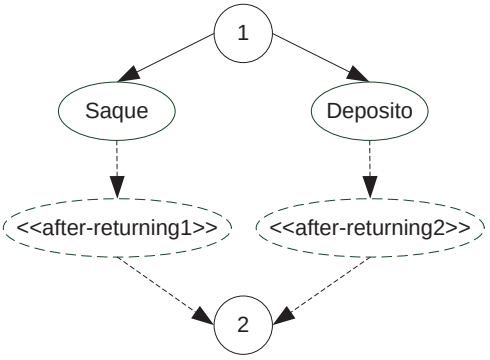


Figura 2.1: Modelo de execução após o *weaving*

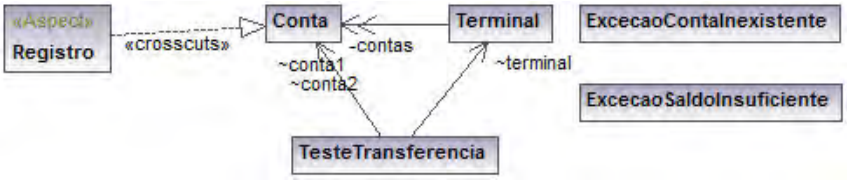


Figura 2.2: Diagrama de classe da aplicação de conta bancária, utilizada como estudo de caso

entrelaçado é então automaticamente compilado no *Java*, em uma tarefa transparente ao desenvolvedor. O código-fonte original (antes do *weaving*) pode conter defeitos que podem provocar falhas, fazendo-se necessário um modelo específico de classificação de falhas para Programas Orientados a Aspectos.

2.1.1 Classificação de Falhas de Programas Orientados a Aspectos

Alexander et al. (2004) propuseram um modelo de falhas para a Programação Orientada a Aspectos, em específico, a linguagem *AspectJ*, especificando seis tipos de falhas:

1. **Força incorreta nos padrões de *PointCuts***: Esse tipo de falha pode fazer com que os *aspectos* falhem em vez da funcionalidade principal falhar. Os *PointCuts* contêm especificações que selecionam *Join points* de um tipo específico, de acordo com uma assinatura que inclua um padrão. A força do padrão na assinatura de um *Pointcut* determina quais *Join points* têm os *adendos* associados com o *Pointcut* entrelaçados dentro de um *Join point*. Se o padrão é muito forte, alguns *Join points* necessários não são selecionados. Se o padrão é muito fraco, *Join points* adicionais, que deveriam ser ignorados, são selecionados. Ambos os casos são suscetíveis de causar um comportamento incorreto do código

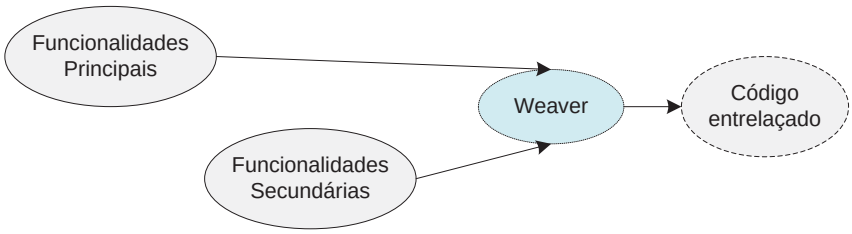


Figura 2.3: Processo de pré-compilação do *AspectJ*.

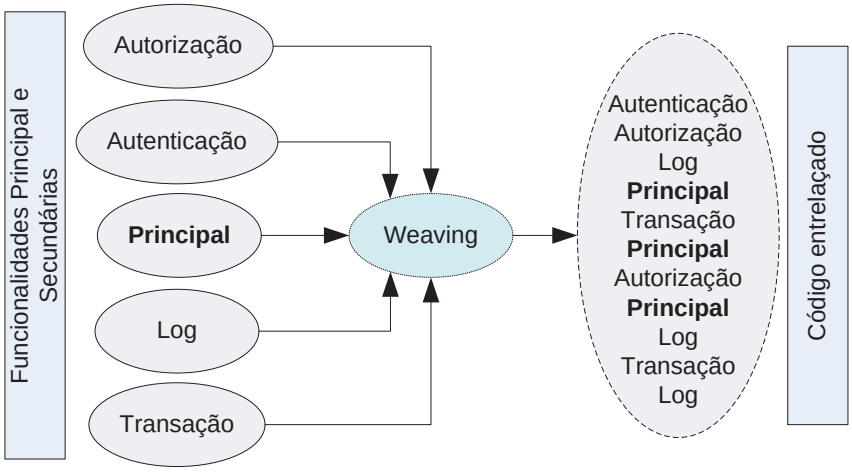


Figura 2.4: Exemplo de mesclagem e distribuição das funcionalidades

entrelaçado. As declarações no *adendo* entrelaçado e as declarações executadas após o *Join point* entrelaçado determinam se um erro de força de padrão introduz uma falha.

2. **Precedência de *Aspectos* Incorreta**: Esse tipo de falha ocorre quando múltiplos *aspectos* interagem, sendo afetados pela ordem de entrelaçamento. A ordem em que *adendos* de múltiplos *aspectos* são entrelaçados em um interesse afeta o comportamento do sistema, especialmente quando há interações mútuas entre *aspectos* por meio de variáveis de estado no núcleo do interesse. No *AspectJ*, a ordem do entrelaçamento é determinada pela especificação de precedência de *aspecto*. Por exemplo, o *aspecto* de mais alta precedência executa seu *adendo before* em um *Join point* antes de executar o *adendo before* de um *aspecto* de mais baixa precedência. A precedência não pertence a nenhum interesse, uma vez que os efeitos de um *adendo* entrelaçado são independentes entre si.

3. **Falha em estabelecer pós-condições esperadas**: Esse tipo de falha pode fazer com que *métodos* da funcionalidade principal falhem. Para obter o correto comportamento de funcionalidade esperado em contrato com o cliente (pós-condições), *adendos* entrelaçados devem permitir que *métodos* no núcleo da funcionalidade satisfaçam suas pós-condições. Definir *adendos* que não alterem comportamentos determinados em contratos em todo o contexto do código entrelaçado e com todas as combinações prováveis de *aspectos* é um desafio para os desenvolvedores de Programas Orientados a Aspectos, e uma provável fonte de defeitos.

4. **Falha em preservar estados invariantes**: Esse tipo de falha também pode fazer com que *métodos* da funcionalidade principal falhem. O comportamento de uma funcionalidade é definido em termos de uma representação física de seu estado e de *métodos* que agem nesse estado. Para estabelecer suas pós-condições, *métodos* devem certificar que estados invariantes sejam satisfeitos. Certificar que o processo de *weaving* não cause violações de estados invariantes é outro desafio para desenvolvedores de Programas Orientados a Aspectos e outra fonte de erros.

5. **Foco incorreto de fluxo de controle**: Esse tipo de falha pode fazer com que *adendos* sejam ativados no momento errado. Um designador de *Pointcut* seleciona qual *método* de um *Join point* é capturado. Tal seleção é determinada no momento da realização do processo de entrelaçamento. Entretanto, muitas vezes há casos em que a informação necessária para tomar corretamente essa decisão está disponível somente em tempo de execução. Algumas vezes *Join Points* só devem ser selecionados em um contexto particular de execução. Tal contexto pode estar dentro de uma estrutura de controle de um determinado objeto, ou dentro de um fluxo de controle que ocorre abaixo de um ponto na execução. Considerando como exemplo a chamada de um *método* recursivo, um determinado *Join point* pode ser selecionado, dependendo da designação de seu *Pointcut*, apenas quando o *método* é chamado fora de seu corpo, ou seja, sem recursividade, restringindo sua seleção. Uma falha ao restringir a execução a um contexto apropriado pode resultar em uma falha suscetível a difícil diagnóstico.

6. **Alterações incorretas em dependências de controle**: Esse tipo de falha afeta o comportamento da funcionalidade principal, assim como os tipos de falhas dos itens 3 e 4. Um determinado bloco de comandos, dependente de um determinado controle (por exemplo, um desvio), pode ter sua dependência de controle alterada por um *adendo*, alterando significativamente a semântica comportamental do *método* em questão.

2.2 Visualização de Informação e de Software

Considerando que a capacidade humana de lidar com informações de maneira visual é maior do que com dados textuais, uma representação visual elaborada a partir de um determinado conjunto de dados pode contribuir para a compreensão e análise do conteúdo desses dados (Ware, 2004). Partindo desse princípio, representações visuais podem ser elaboradas dentro dos contextos de Visualização de Informação e Visualização de Software.

2.2.1 Visualização de Informação

A Visualização de Informação provê apoio à tarefa de análise de dados por meio de representações visuais, permitindo que o usuário interaja com os gráficos produzidos como resultado de cada representação visual (Card et al., 1999).

Card et al. (1999) propuseram um modelo de referência para Visualização, descrevendo um processo de transformação e mapeamento de dados para a forma visual baseado em interação humana. As tarefas envolvidas em tal processo são suportadas por ferramentas específicas que permitem ajustes nos mapeamentos e nas transformações, por meio de controles de usuário, para abordar alguma tarefa particular. Esse processo é ilustrado na Figura 2.5.

Os dados a serem visualizados, em seu estado bruto *(Raw Data)* devem ser tabelados e organizados em um formato específico. Os dados tabelados são, então, mapeados para estruturas visuais representativas (demonstrar visualmente o significado do dado), como formas geométricas, cores e texturas. As estruturas visuais são apresentadas ao usuário em visões com base em

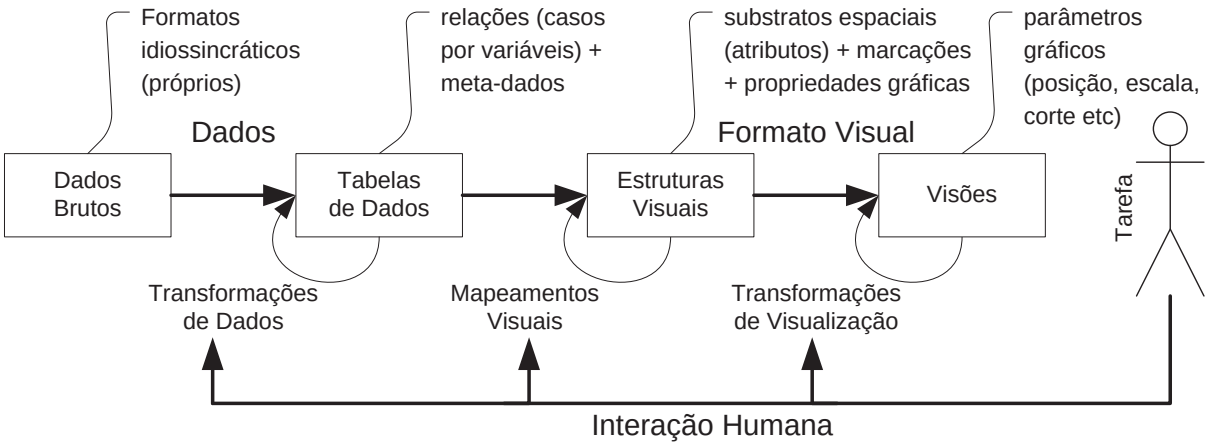


Figura 2.5: Processo de Visualização generalizado, adaptado de Card et al. (1999)

seus parâmetros gráficos, como posição (*pan*), escala (*zoom*) e corte (*crop*). O usuário pode interagir em cada etapa de transformação e mapeamento de dados, personalizando a visualização de acordo com suas necessidades.

2.2.2 Visualização de Software

A Visualização de Software visa utilizar a capacidade humana de lidar com informações visuais em tarefas de entendimento de programas. Há várias definições para Visualização de Software. Price et al. (1998) a definem como "o uso de artesanatos[1] de tipografia, *design* gráfico, animação e cinematografia com moderna interação humano-computador e tecnologia de computação gráfica para facilitar tanto o entendimento humano quanto o uso efetivo de software de computador". Essa definição abrange diversos tópicos, como visualização de programa, animação de algoritmo, programação visual, navegadores de código fonte e visualização de dados (Maletic e Collard, 2002). De acordo com Knight (2000), Visualização de Software pode ser considerada como um subconjunto especializado de Visualização de Informação, uma vez que Visualização de Informação é o processo de criação de uma apresentação gráfica de dados abstratos, usualmente não-numéricos, e Visualização de Software envolve tarefas de mapeamentos para representar dados abstratos extraídos de Artefatos de Software em estruturas visuais.

Esperam-se alguns recursos essenciais em uma ferramenta de Visualização de Software. Por exemplo, uma ferramenta deve prover mecanismos para o usuário navegar por todos os itens apresentados em uma representação gráfica. Também esperam-se mecanismos para interação do usuário com a representação, como rotação, escala (*zoom*), corte (*crop*) e ligação (*link*). Os três primeiros devem permitir interação com a representação visual, habilitando a exploração de diferentes ângulos e perspectivas, enquanto que a ligação deve permitir navegação direta de uma entidade a outra, visando facilitar a navegação entre entidades inter-relacionadas. Outra característica é a possibilidade de alternância, ou seja, a visualização permitir ao usuário mudar entre a apresentação visual e a porção de código representada, ou exibir ambas de maneira síncrona na interface.

[1] Price et al. (1998) utilizam este termo para enfatizar a utilização de formas e representações gráficas.

Essas características permitem a criação de múltiplas visões (representações visuais diferentes) para auxiliar o processo de entendimento. Um exemplo de uma ferramenta de Visualização de Software é exibido na Figura 2.6 – análise de modificações realizadas em um código fonte. Uma ferramenta de Visualização de Software pode apoiar a tarefa de Compreensão de Programa, porém, seu uso pode aumentar a dificuldade do processo de compreensão, pois exige o domínio da ferramenta e a sua adequação à tarefa, assim como depende da eficácia das representações visuais para auxiliar o processo de entendimento. Portanto, é fundamental a avaliação da ferramenta em um contexto para ter indício da eficácia de seu uso.

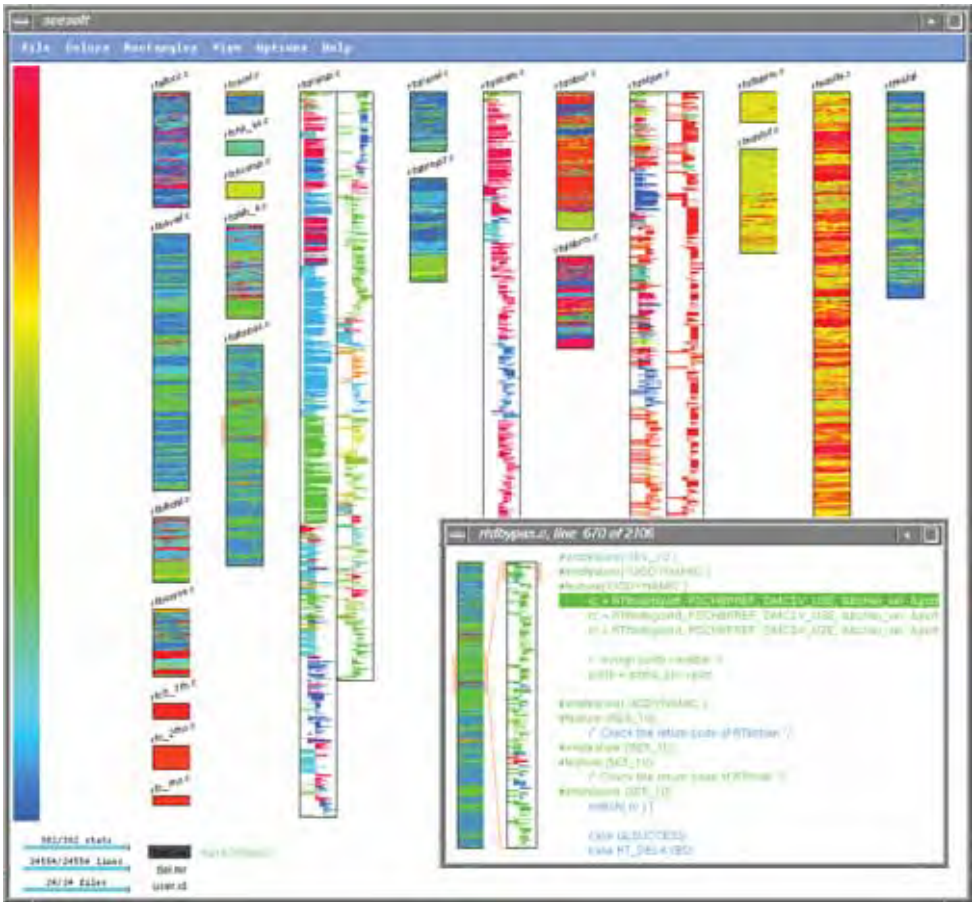


Figura 2.6: Idade do código: As linhas mais novas são exibidas em vermelho, as mais antigas em azul, com um gradiente para as linhas intermediárias (Ball e Eick, 1996).

O modelo de visualização proposto por Card et al. (1999) (Figura 2.5) pode ser instanciado no contexto da Engenharia de Software, utilizando dados extraídos de Artefatos de Software e mapeando-os para estruturas visuais que representem características do artefato a ser visualizado. Dessa maneira, é possível tornar graficamente visível a estrutura e o funcionamento do programa, provendo auxílio à Compreensão de Programa bem como à análise de resultados de testes. É importante notar que nenhuma ferramenta de Visualização de Software pode, por si só, abordar todas as tarefas de Engenharia de Software: essas tarefas vão de codificação e de depuração para projeto e re-engenharia.

O uso adequado de técnicas visuais pode facilitar a Compreensão de Programa e, consequentemente, facilitar atividades no processo de software – por exemplo, manutenção de código e

definição de casos de teste. Entretanto, o uso apropriado de técnicas de visualização depende de ferramentas que provêm cenários adequados à analise. Cada cenário pode ser criado com foco em um objetivo específico. Por definir múltiplos cenários, pode-se criar um modelo de visualização coordenada para auxiliar o processo de entendimento.

2.2.2.1 Ferramentas e Técnicas de Visualização Relacionadas

Diferentes técnicas de visualização podem ser utilizadas para criar múltiplos cenários visuais. Técnicas propostas inicialmente em Visualização de Informação podem ser utilizadas em Visualização de Software, provendo suporte ao processo de entendimento.

Uma visualização *TreeMap* (Johnson e Shneiderman, 1991, Bederson et al., 2002, Pfeiffer e Gurd, 2006) permite representar estruturas hierárquicas por meio de subdivisão de espaço. A hierarquia da estrutura é mapeada em regiões retangulares restritas e aninhadas. Diversos dados podem ser associados à dimensão e à coloração dos retângulos, por exemplo: a área pode representar a quantidade de linhas de código (*LOC*) e a coloração pode representar a complexidade estrutural de uma determinada *classe*. Na Figura 2.7 é ilustrado um exemplo de uma representação *TreeMap*.

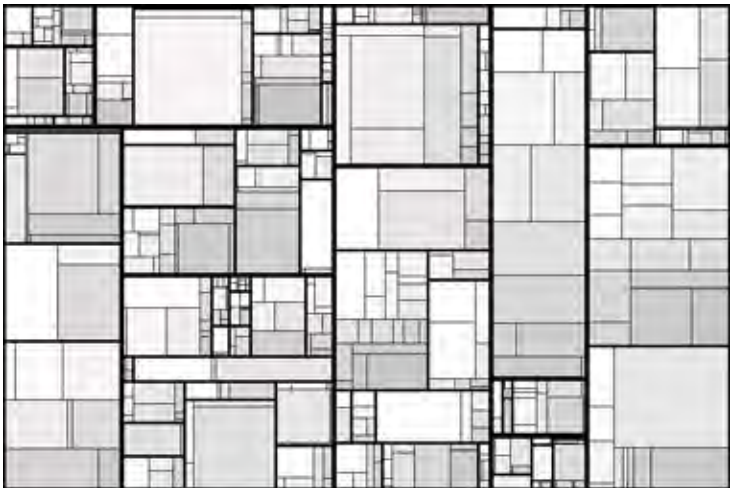


Figura 2.7: Exemplo de uma visualização *Treemap* (Bederson et al., 2002)

As *Visões Polimétricas* (Lanza, 2004, Carneiro et al., 2008) permitem representar estruturas hierárquicas por meio de nós e ligações. Os nós representam as entidades e as ligações representam a hierarquia das entidades, formando representações em forma de árvores, conforme ilustrado na Figura 2.8. Assim como no *TreeMap*, diversos dados podem ser associados à dimensão e à coloração dos nós, com a diferença de que dois dados podem ser associados à dimensão, um à largura e outro à altura.

As *Árvores Hiperbólicas* (Munzner, 1998), conforme ilustrado na Figura 2.9, permitem representar grandes estruturas hierárquicas por meio de uma projeção hiperbólica, possibilitando um maior detalhamento da vizinhança de um determinado nó ou ponto da representação visual.

A técnica *Barras e Listras* (Baldwin et al., 2009) permite a visualização de unidades e respectivos atributos. Cada unidade é representada por uma barra, e os atributos em destaque são representados por listras dentro de cada barra envolvida.

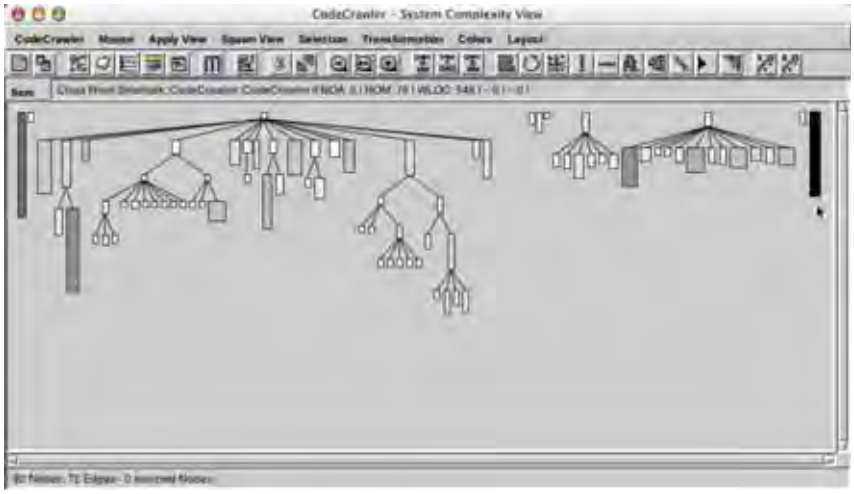


Figura 2.8: Exemplo de uma visualização Visão Polimétrica, exibindo estruturas de classes e métodos (Lanza, 2004).

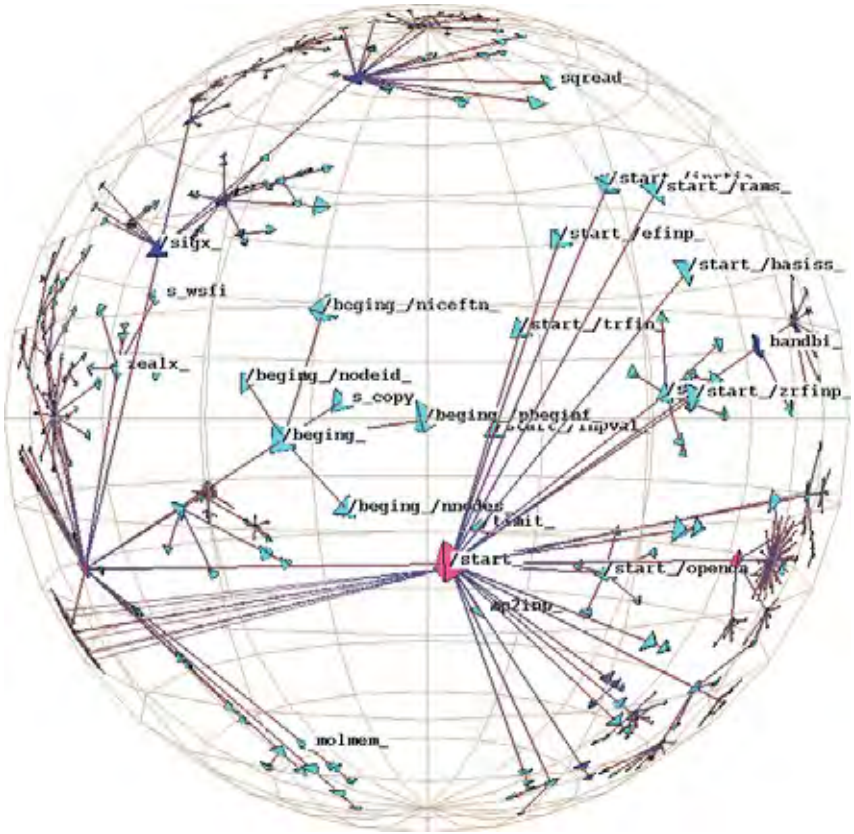


Figura 2.9: Exemplo de uma visualização *Árvore Hiperbólica* representando a função de estrutura de gráfico de chamadas para um *benchmark* de computação científica em *FORTRAN*, em que a coloração do nó indica se uma determinada variável global foi intocada (ciano), referenciada (azul), ou modificada (rosa) (Munzner, 1998).

A *UML 3D* (Gall e Lanza, 2006) consiste em uma técnica de visualização proveniente dos diagramas de classe da *UML*, representando *pacotes*, *classes* e *métodos* em estruturas de três dimensões. Como extensão dessa técnica, existem as *Cidades UML* (Wettel e Lanza, 2007), as quais mapeiam as estruturas de *pacotes*, *classes* e *métodos* em uma representação gráfica baseada em *UML 3D*, permitindo visualizar a estrutura do programa como uma cidade.

Os *Grafos de Dependência* (Wurthinger et al., 2008) permitem visualizar níveis de inter-dependência entre unidades. As unidades são representadas por nós e seus respectivos níveis de dependência com outras unidades são representadas por arestas, sendo que, quanto maior o nível de dependência entre duas unidades, mais próximas uma da outra elas são posicionadas no grafo.

Observa-se que tais ferramentas não exploram múltiplos cenários de visualização, por meio de um modelo de visualização coordenada para auxiliar o processo de entendimento das unidades de programas.

2.3 SoftVisOAH

Diferentes técnicas de visualização têm sido propostas para oferecer suporte à Visualização de Software, mas elas não conseguem representar, por si só, o código resultante de um Programa Orientado a Aspectos após o processo de *weaving* (pois não foram propostas para tal caso), e sim os *aspectos* antes do *weaving*. Exemplos de tais técnicas são *TreeMap* (Johnson e Shneiderman, 1991, Bederson et al., 2002, Pfeiffer e Gurd, 2006), *Visões Polimétricas* (Carneiro et al., 2008, Lanza, 2004), *Árvores Hiperbólicas* (Munzner, 1998), *Barras e Listras* (Baldwin et al., 2009), *UML 3D* (Gall e Lanza, 2006) e *Grafos de Dependência* (Wurthinger et al., 2008). Mediante tal situação, foi desenvolvido um mapeamento visual abordando Programas Orientados a Aspectos, resultando em um conjunto de visualizações coordenadas, implementado na $SoftVis_{OAH}$ (d'Arce et al., 2011, 2012). A arquitetura da $SoftVis_{OAH}$, apresentada na Figura 2.10, é organizada em três camadas: *Conjunto de Dados*, *Controle* e *Visualização*.

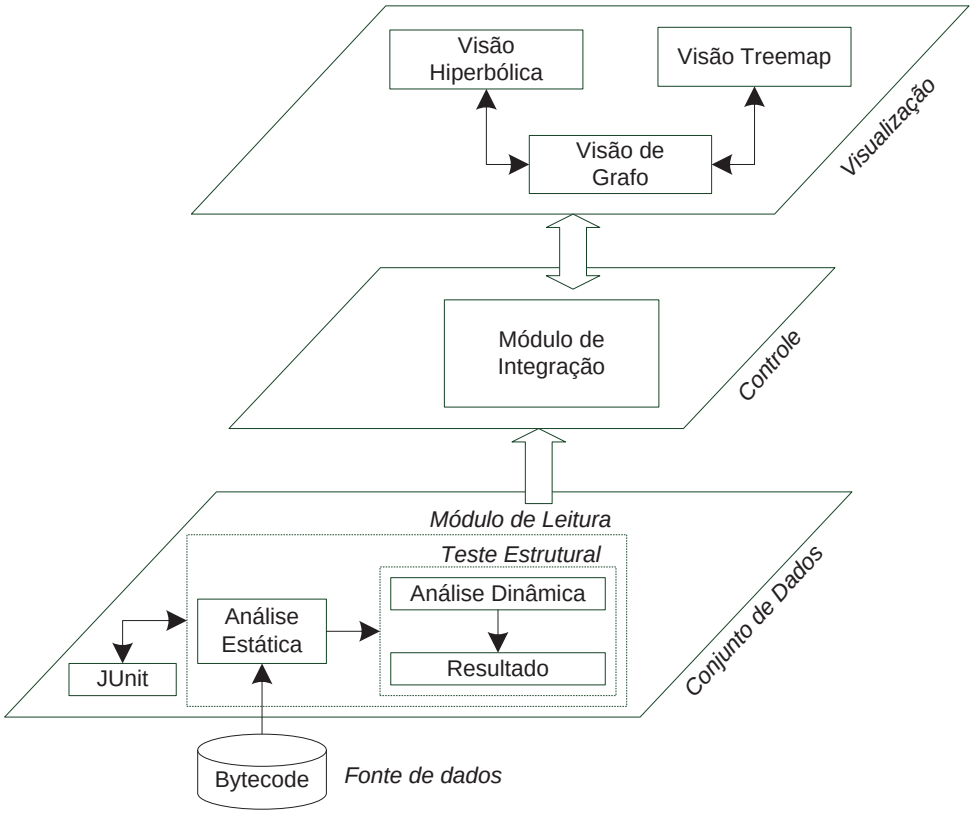


Figura 2.10: Arquitetura da ferramenta desenvolvida, $SoftVis_{OAH}$.

A *Camada de Conjunto de Dados* destina-se às análises estática e dinâmica e à criação dos conjuntos de dados a serem visualizados. O *Módulo de Leitura* lê o *bytecode* de um programa e analisa *classes*, *aspectos* e casos de teste (usando o *framework JUnit*) por meio de análise estática.

Durante a leitura, o fluxo de dados de cada unidade de código é analisado e inserido em estruturas, representando o Grafo de Fluxo de Controle (*GFC*), enquanto que novas instruções são inseridas em cada unidade de código (técnica de instrumentação, contendo chamadas de métodos e *adendos*, provendo *feedback* para permitir a monitoração dos casos de teste pela *Análise Dinâmica*), e é construída uma lista com todas as classes e *aspectos* do projeto e suas informações providas pela instrumentação. A partir dessa lista, é construída uma árvore hierárquica, a qual é utilizada posteriormente como dados para a geração da visualização *Treemap*. Ainda, informações como referências a superclasses, tipos de variáveis, parâmetros e retornos de métodos e outras são agrupadas em uma tabela de nós e uma tabela de arestas para montar um grafo de dependência (pela interpretação dessas tabelas e a organização das ligações entre os nós), o qual é utilizado para gerar a representação visual *Hiperbólica*.

Durante a realização do *Teste Estrutural*, a análise dinâmica monitora os casos de teste e registra cada porção de código visitado de cada unidade de código, formando o trajeto da execução. A partir do trajeto, os critérios de cobertura são verificados para determinar o *Resultado* dos testes – cada critério tem seu próprio padrão para verificar se o teste tem sido total ou parcialmente satisfeito. Durante a execução de um teste, a instrumentação envia à ferramenta dados sobre cada porção de código visitada. Tais dados são armazenados para serem utilizados por cada representação visual. Por meio dessas duas etapas de análise, a ferramenta reúne dados para gerar as representações visuais (Martins, 2007, Trevisan, 2010, Dutra, 2010).

Na *Camada de Controle*, o *Módulo de Integração* organiza os dados obtidos do *Módulo de Leitura* para gerar as representações visuais e provê mecanismos para a coordenação das mesmas. Esses mecanismos capturam a interação com uma representação visual para refleti-los nas demais visualizações.

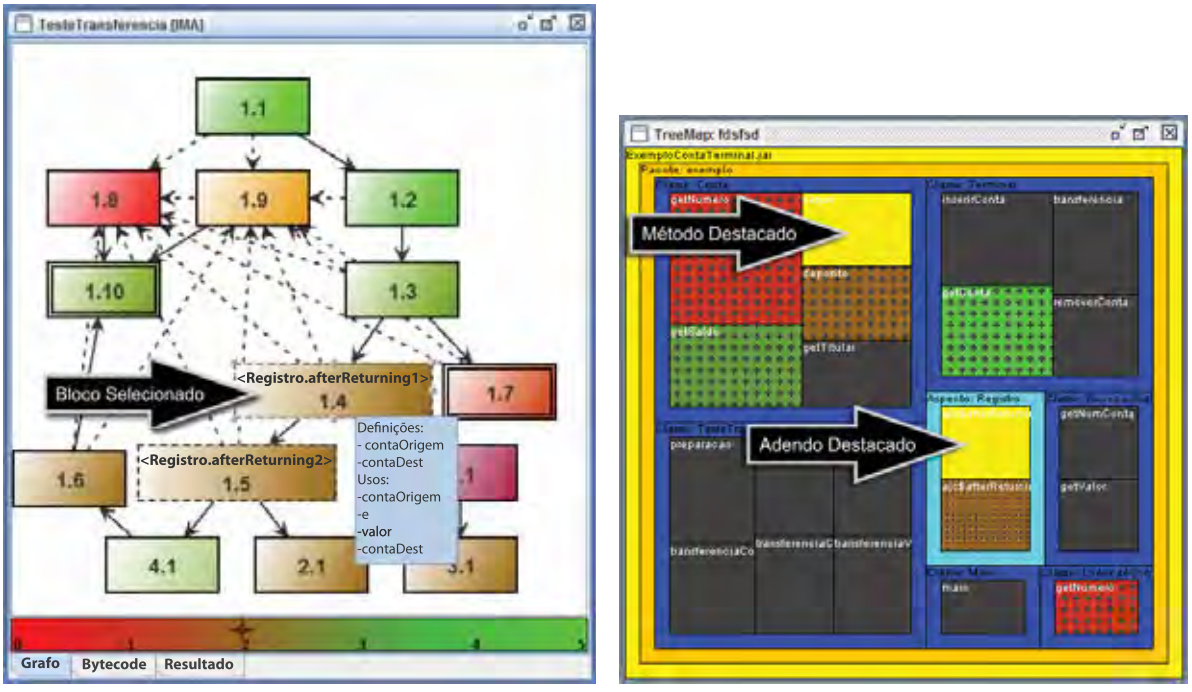
Um evento, disparado por um usuário em uma visualização específica, é capturado. A partir desse evento, o *Módulo de Integração* obtém dados sobre o elemento selecionado do programa (classe, método, *aspecto* ou *adendo*). As estruturas de dados armazenadas na camada de *Conjunto de Dados* (como a estrutura de dados de grafo, a lista de informações da instrumentação, a árvore hierárquica e as tabelas de nós e arestas) são utilizadas para realizar a coordenação entre as visualizações. Então, um evento é enviado a cada uma das outras visualizações na camada de *Visualização*, informando o novo estado da representação visual (como itens destacados), realizando, assim, a coordenação.

A *Camada de Visualização* provê as representações visuais *Grafo de Fluxo de Controle*, *Treemap* e *Visualização Hiperbólica*. Em cada representação visual, os *aspectos* são destacados e, quando um caso de teste (criado pelo usuário por meio do *JUnit*) é aplicado, a coloração de alguns componentes é feita por meio de um gradiente. O gradiente, definido do vermelho para o verde, é utilizado quando um determinado caso de teste está associado à visualização em

questão. Quanto menos o trecho de código representado na visualização é executado nos testes associados, em tom mais vermelho ele é colorido. Quanto mais o trecho de código é executado, mais verde ele é colorido. Tal mapeamento de cores permite visualizar o quanto o caso de teste cobriu o trecho de código. Essa coloração baseada no resultado de casos de teste foi definida por causa da visualização coordenada, para que os elementos possuam a mesma representação de cor em cada cenário visual. Mas para destacar a seleção de *aspectos*, uma cor diferente é utilizada.

Na representação visual *Grafo de Fluxo de Controle*, blocos de código básico são representados por retângulos contornados por linha contínua, blocos de código entrecortados por *aspectos* são representados por retângulos contornados por linha tracejada e retornos de métodos são representados por retângulos contornados por linha dupla, sendo que todos os blocos de código pertencem a códigos dentro de métodos e *aspectos*. Dentro de cada retângulo há um número indicando a sequência de execução do bloco de código. Sequências de código normais são representadas por linha contínua, e exceções são representadas por linha tracejada. Se um caso de teste é visualizado, cada retângulo é colorido de acordo com o gradiente.

A representação visual *Treemap* é utilizada para apresentar estruturas hierárquicas, exibindo -as em retângulos restritos e aninhados. Cada retângulo representa um pacote, classe, método, *aspecto* ou *adendo*. Seu tamanho representa o número de chamadas ao método ou *adendo* representado. Essa visualização está associada a um caso de teste, sendo que a coloração de cada retângulo é feita utilizando o gradiente.



(a) Visualização *Grafo* (b) Visualização *Treemap*

Figura 2.11: Instantâneos de representações visuais coordenadas obtidas com a ferramenta $SoftVis_{OAH}$

Nas Figuras 2.11(a) e 2.11(b) são exibidos o *GFC* e a visão *Treemap* do caso de teste *testeTransferencia*, o qual executa o método *transferir* (vide Quadro 2.2). Cada nó no *GFC* re-

presenta um bloco de códigos executado, e suas conexões representam os possíveis caminhos de execução, sendo que as linhas tracejadas representam chamadas automáticas geradas pela Linguagem Java (por exemplo, exceções). Todos os blocos de código, *adendos* e métodos são coloridos de acordo com seus resultados no caso de teste. A visualização *grafo* está exibindo dois *adendos* (representados por retângulos com linhas tracejadas) com seus respectivos *pointcuts* (representados por sua sequência no grafo) entrecortando um caso de teste com a classe *Conta*. A visão global da visualização *grafo* representa o código entrelaçado e espalhado.

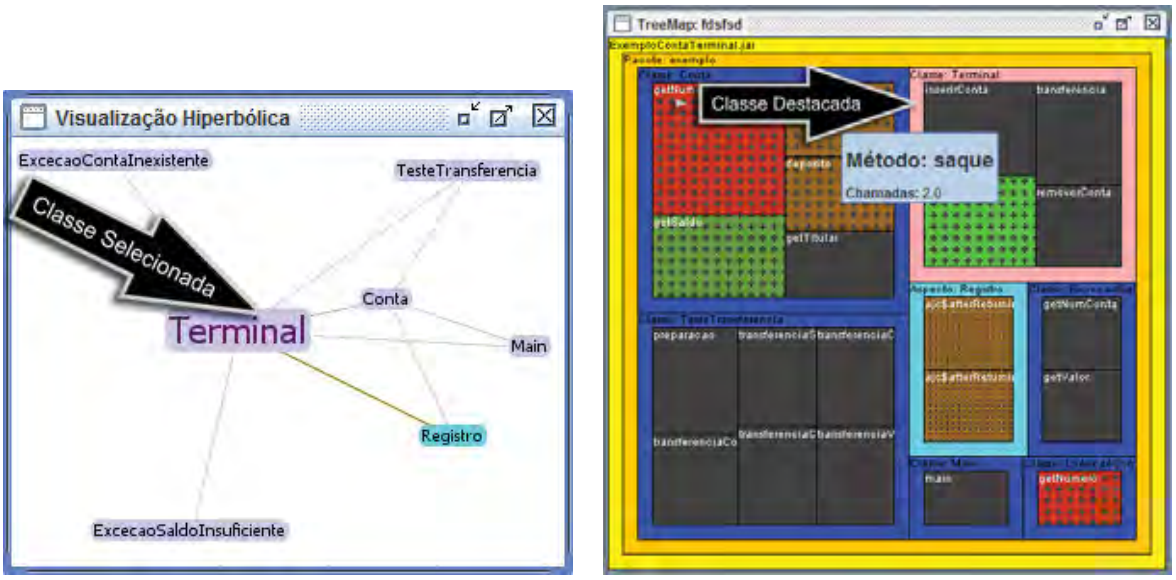
Quando um bloco de código é selecionado na visualização em *Grafo*, seu respectivo método é selecionado na visualização *Treemap*, e quando um bloco de código selecionado contém um *adendo*, o respectivo *adendo* também é selecionado na visualização *Treemap*, como apresentado na Figura 2.11: o retângulo selecionado na Figura 2.11(a) representa um bloco de código contendo a chamada para o método *saque* e seu respectivo *adendo* entrecortante, e tanto o método quanto o *adendo* são destacados na Figura 2.11(b). Desta maneira, é possível visualizar quais métodos são entrecortados por quais *adendos* de um ou mais *aspectos*.

Na representação visual *Hiperbólica*, os nós representam classes e *aspectos*, distinguidos por cores diferentes, e as arestas representam a dependência entre cada classe ou *aspecto* (chamadas de métodos e *join points*). Cada aresta é colorida de acordo com o resultado de seu teste estrutural (gradiente). Por interação, pode-se selecionar classes, *aspectos* e arestas entre eles.

Quando uma classe ou *aspecto* é selecionado na visualização *Hiperbólica*, o mesmo é movido para o centro da projeção (conforme ilustrado na Figura 2.12(a)) a respectiva classe ou *aspecto* também é selecionado na visualização *Treemap* (conforme ilustrado na Figura 2.12(b)), e seus respectivos métodos ou *adendos* são exibidos em um painel – lista de métodos ilustrada na Figura 2.13. Similarmente, quando uma aresta é selecionada na visualização *hiperbólica*, os métodos e *adendos* participantes no caso de teste também são selecionados na visualização *treemap* e a lista de métodos/*adendos* é apresentada. Na Figura 2.14(a) é possível observar a seleção de uma aresta entre a classe *classe Terminal* e o *aspecto Registro*, e os adendos participantes da ligação destacados na Figura 2.14(b). Desta maneira, é possível visualizar quais classes são entrecortadas por quais *aspectos* e quais são os respectivos *adendos* entrecortantes.

Quadro 2.2: Método chamando dois métodos entrecortados

```
1  public void transferir(int numeroOrigem, int numeroDestino, float valor) {
2    try {
3      account contaOrigem = getConta(numeroOrigem);
4      account contaDestino = getConta(numeroDestino);
5      if (contaOrigem.getBalanco() >= valor) {
6         contaOrigem.saque(valor); // MÉTODO ENTRECORTADO
7         contaDestino.deposito(valor); // MÉTODO ENTRECORTADO
8      }  else
9        throw new ExcecaoSaldoInsuficiente(contaOrigem.getNumber(),valor);
10     } catch (ExcecaoContaInexistente e) {
11       System.out.println(e.getMessage());
12     } catch (ExcecaoSaldoInsuficiente e) {
13       System.out.println(e.getMessage());
14   }
15 }
```



(a) Visualização *Hiperbólica* (b) Visualização *Treemap*

Figura 2.12: Instantâneos de representações visuais coordenadas obtidas com a ferramenta $SoftVis_{OAH}$

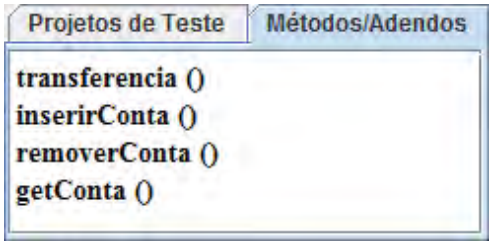
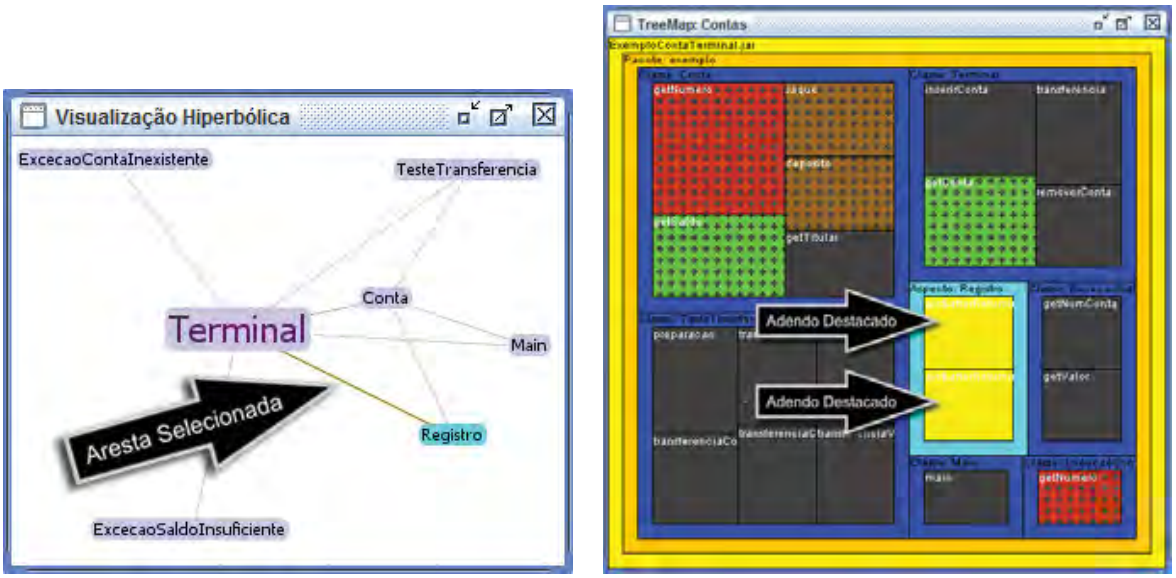


Figura 2.13: Métodos selecionados



(a) Visualização *Hiperbólica* (b) Visualização *Treemap*

Figura 2.14: Instantâneos de representações visuais coordenadas obtidas com a ferramenta $SoftVis_{OAH}$

2.4 Considerações Finais

Uma técnica de visualização deve evidenciar elementos em suas representações visuais que forneçam algo significativo para ser analisado de acordo com o artefato a ser visualizado, com as tarefas específicas e com os usuários. Artefatos diferentes possuem estruturas diferentes. Consequentemente, técnicas de visualização diferentes devem ser aplicadas, por exemplo, um código fonte utilizado como artefato pode conter estruturas hierárquicas, sequenciais ou ambas, e há algumas representações visuais que se enquadram melhor em alguns tipos de estrutura. Artefatos de software possuem conceitos e estruturas diferentes, mas relacionados, e alguns deles são pontos-chave em uma linguagem de programação – por exemplo, o conceito de um *aspecto* modificando o comportamento de uma *classe* em um Programa Orientado a Aspectos. A representação visual deve externalizar essa situação.

Para visualizar programas Orientados a Aspectos, especificamente o *AspectJ*, é interessante que uma ferramenta de Visualização de Software evidencie (em mais de um cenário visual) características do programa, como *aspectos*, *adendos*, *pointcuts* e *join points*, com o objetivo de permitir ao usuário visualizar como um *aspecto* entrecorta uma ou mais *classes* ou o quanto uma *classe* é modificada por um ou mais *aspectos*. Na ferramenta $SoftVis_{OAH}$ é implementado um mapeamento visual coordenado com o objetivo de apresentar características de programas Orientados a Aspectos por meio de três representações visuais e uma lista de conteúdos que visam a externar a organização estrutural, as relações entre classes e aspectos, e o código *adendado*[2]. O esquema de coordenação proposto torna possível destacar elementos selecionados em diferentes níveis de detalhe, permitindo ao usuário reunir informações sobre aspectos e seu espalhamento e entrelaçamento pelo código fonte – como aspectos entrecortam as estruturas do programa, modificando o comportamento destas. A ferramenta também possibilita a visualização de casos de teste criados em *JUnit*, permitindo ao usuário visualizar como um caso de teste percorre o código fonte, provendo auxílio ao entendimento do programa.

Entretanto, o uso da ferramenta $SoftVis_{OAH}$ – e o mapeamento visual proposto – pode não ter o efeito esperado, ou seja, auxiliar no entendimento de um programa Orientado a Aspectos, o que motiva e justifica a sua avaliação.

[2] código entrecortado por *adendos* de *aspectos*

CAPÍTULO

3

Engenharia de Software Experimental

Para avaliar a ferramenta $SoftVis_{OAH}$, é preciso uma abordagem metodológica que permita obter evidências sobre sua utilização. A condução de um estudo experimental controlado foi escolhida como abordagem para tal avaliação. Assim, neste capítulo é apresentada uma visão geral de Engenharia de Software Experimental e do processo de experimentação, com o objetivo de expor uma base teórica que fundamenta a metodologia adotada.

3.1 Engenharia de Software Experimental

O termo experimentação na área da pesquisa científica denota uma atividade de realização de experiências, representando uma atividade laboratorial (Travassos et al., 2002). Experiências são conduzidas em um ambiente planejado, em que variáveis de entrada são selecionadas, o processo para o tratamento e o controle dessas variáveis são definidos, a experiência é executada e os resultados são estudados.

A Engenharia de Software Experimental trata a Engenharia de Software como uma ciência de laboratório e utiliza métodos empíricos de investigação experimental, ou seja, condução de experimentos, para validar teorias e hipóteses, com o objetivo de caracterizar, avaliar, prever, controlar e melhorar o Processo de Software. Segundo Zelkowitz e Wallace (1998), "a experimentação ajuda a determinar a eficácia de métodos e teorias propostas".

Estudos experimentais vêm sendo realizados, contribuindo para a formação de uma base de conhecimento sobre Engenharia de Software que pode ajudar nas tomadas de decisão ao longo do desenvolvimento, e vêm sendo conduzidos sobre os diversos métodos e técnicas aplicáveis às diversas etapas do processo de desenvolvimento de software (Prechelt e Unger, 1998, Lanubile e Visaggio, 1995, Perry et al., 1998). Um conjunto de atividades tem sido introduzido com o objetivo de melhorar a qualidade do processo e dos produtos de software, com diversos estudos realizados em atividades como Verificação, Validação e Teste (*VV&T*) (Basili et al., 1986, 1996c,b,a, 1998, Fusaro et al., 1997, Wood et al., 1997, Zelkowitz e Wallace, 1997, Hohn, 2003,

Maldonado et al., 2006) e outras áreas (Lucia et al., 2010).

Um experimento é definido para verificar uma teoria inicial de causa e efeito. Por meio de observações, a causa recebe um determinado tratamento e o efeito conduz a um determinado resultado. Essa observação estabelece um relacionamento de tratamento e resultado, sendo que o *tratamento* consiste em variáveis manipuladas e controladas (variáveis dependentes), e o *resultado* consiste em uma variável estudada e analisada (variável dependente).

Para se conduzir um experimento é necessário seguir um protocolo que possibilite não só determinar a validade de seus resultados, mas também sua replicação (Carver, 2010). Nesta seção são apresentados os conceitos, as definições e as etapas da Experimentação em Engenharia de Software, explicando estratégias empíricas de pesquisa e o processo de experimentação.

3.1.1 Engenharia de Software como Ciência

Pelo fato de aplicar um conhecimento científico, a Engenharia de Software lida com a ciência, ou seja, um conjunto organizado de conhecimentos relacionados a um objeto em questão, no qual tais conhecimentos são obtidos por meio de observações, experiência dos fatos e por meio de um método próprio, conhecido como método científico. Sendo a ciência o processo pelo qual o homem se relaciona com a natureza (com o intento de dominá-la em benefício próprio), ela se caracteriza pela observação e busca de evidências a respeito do comportamento do objeto observado. O conhecimento científico é, então, formado a partir das conclusões do conjunto de tais observações e evidências (Wohlin et al., 2000).

Antes de aplicar o método científico para a obtenção de conhecimentos na Engenharia de Software, faz-se necessário esclarecer alguns conceitos. No método científico, fenômenos sociais ou da natureza são estudados por meio de formulação de hipóteses. Um fenômeno consiste em uma descrição de um efeito social ou natural, que pode ser observado. Uma hipótese consiste em uma formulação ainda não avaliada por meio de estudos experimentais. Os fenômenos são observados e, se as observações estiverem alinhadas com a hipótese, eles servem de evidência para esta. Uma lei da sociedade ou da natureza consiste em uma generalização de uma ou mais hipóteses, sendo baseada em observações (Wohlin et al., 2000).

Para tratar a Engenharia de Software como realmente uma engenharia, é necessário se tratar as pesquisas na área como uma ciência, ou seja, é necessário adotar métodos científicos para decidir sobre mudanças na maneira em que os softwares são desenvolvidos, fazendo uso de ciclos de modelagem, experimentação e aprendizado (Wohlin et al., 2000).

3.1.2 Estratégias Empíricas

No contexto da Engenharia de Software, existem três métodos empíricos de investigação para a condução de um estudo: *Survey* (levantamento), Estudo de Caso e Experimento (Wohlin et al., 2000).

- ***Survey***: Investigação conduzida em retrospecto, permitindo capturar um instantâneo da situação mediante técnicas e ferramentas. Utiliza entrevistas ou questionários como principal meio de coleta de dados, sejam eles qualitativos ou quantitativos, coleta esta rea-

lizada utilizando-se uma amostra representativa da população em estudo. Os resultados são analisados para extrair conclusões.

- **Estudos de Caso**: Estudos conduzidos com a finalidade de investigar uma entidade ou fenômeno em um período de tempo. Monitoram atributos presentes em projetos, atividades ou tarefas. Consiste em um estudo observacional, no qual os fatores-chave que podem afetar os resultados são identificados e a atividade é documentada. Baseado na coleta dos dados (qualitativos ou quantitativos), durante a condução do estudo, são conduzidas análises estatísticas para avaliar um determinado atributo ou o relacionamento entre diferentes atributos.

- **Experimentos**: É uma investigação formal, rigorosa, controlada e puramente quantitativa, podendo ser feita sob condições de laboratório (*in-vitro* – proporcionando um nível relativamente alto de controle sobre as variáveis) ou sob condições normais (*in-vivo*) (Travassos et al., 2002). Participantes são atribuídos a diferentes tratamentos de maneira aleatória. O efeito da manipulação é medido e as análises estatísticas são conduzidas.

Um experimento precisa ser tratado como um processo de verificação ou formulação de uma teoria. Para que o processo forneça resultados válidos, ele deve ser organizado e controlado. Existem várias metodologias de organização dos experimentos com o propósito de atingir tais objetivos. Uma delas é a do Paradigma da Melhoria Contínua (*Quality Improvement Paradigm – QIP*), consistindo em um ciclo de melhoria contínua do processo de desenvolvimento de software.

O ciclo tem início com a caracterização do processo de negócio, compreendendo a definição dos objetivos básicos e o próprio ambiente. Em seguida, são estabelecidos os objetivos quantitativos, a fim de demonstrar as expectativas razoáveis da experimentação. Com base na caracterização e nos objetivos definidos, é escolhido o processo apropriado de melhoria, levando em consideração a consistência entre os objetivos. O processo de desenvolvimento do software oferece a informação recolhida (*feedback* do projeto), além do próprio software. Tal informação serve de base para a análise, sendo ela, a avaliação das práticas atuais, a determinação dos problemas e a proposição da melhoria futura. Ao final, toda informação relevante é empacotada para futura utilização.

3.1.2.1 Abordagem Goal/Question/Metric

Um instrumento relacionado ao Paradigma da Melhoria Contínua refere-se à abordagem *Goal/Question/Metric, GQM* (Basili et al., 1994), a qual fornece a abordagem de melhoria com o modelo da medição baseado em níveis – Nível Conceitual (*Goal*), Nível Operacional (*Question*) e Nível Quantitativo (*Metric*) – conforme ilustrado na Figura 3.1.

Os objetivos principais definidos pelo *Goal/Question/Metric* são compreender, controlar e melhorar, e são focados nos fatores custo, risco, tempo e qualidade. Juntando os fatores e os objetivos, obtém-se maior compreensão do produto e do processo de software, ambos tornam-se controlados, e as atividades de melhoria do produto e do processo de software são definidas (Wohlin et al., 2000).

A abordagem GQM possui quatro fases: *planejamento*, *definição*, *coleta de dados* e *interpretação*. Na fase de *planejamento*, o projeto da medição é solicitado, definido, caracterizado e planejado, resultando no plano do projeto. Na fase de *definição*, o programa de medição é conceitualmente preparado – os objetivos, as questões, as métricas e as hipóteses são estabelecidos. Na fase de *coleta de dados*, a coleta de dados experimentais é efetivamente realizada, resultando em um conjunto de dados para interpretação. Na fase de interpretação, os dados são processados de acordo com as métricas, as questões e os objetivos definidos.

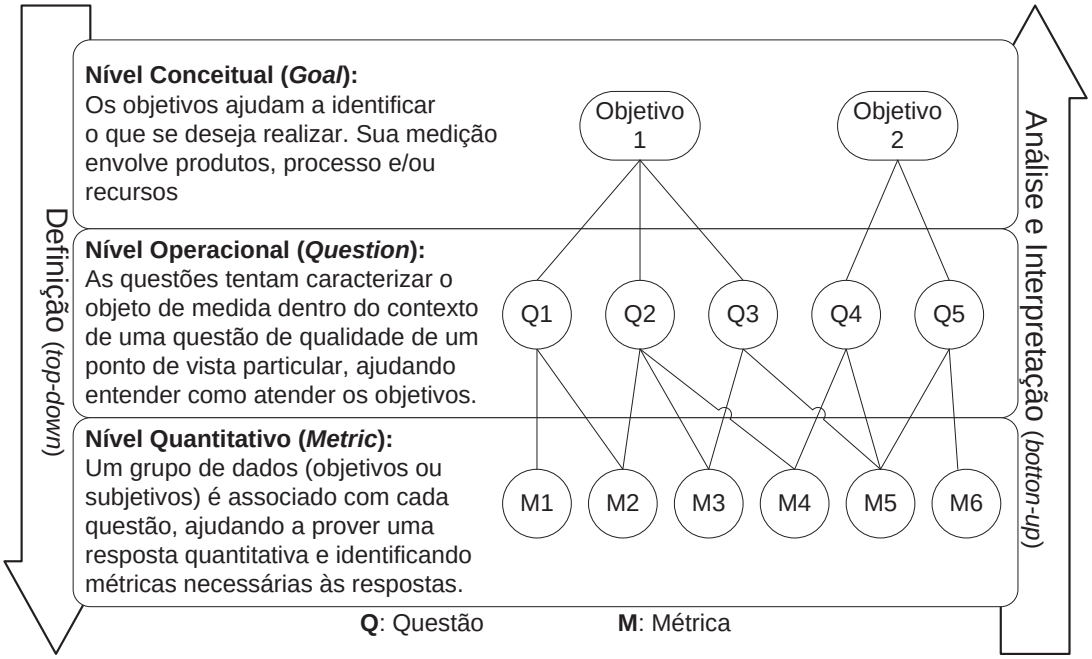


Figura 3.1: Abordagem *Goal/Question/Metric*, adaptado de Wohlin et al. (2000)

A definição do processo de experimentação é dividida nos níveis *conceitual*, *operacional* e *quantitativo*, utilizando a abordagem *top-down*, a qual consiste no estabelecimento dos objetivos, na formulação de questões baseada nos dados experimentais e na elaboração das métricas. Já a *análise* e *interpretação* usam a abordagem *bottom-up*, a qual consiste na medição para receber dados experimentais, formulação das respostas para as questões baseadas nos dados experimentais e agrupamento dessas respostas para demonstrar o grau de sucesso dos objetivos estabelecidos.

3.1.3 Experimentos na Engenharia de Software

Segundo Basili et al. (1999), a Engenharia de Software Experimental é um segmento da Engenharia de Software que utiliza experimentos para verificação e validação de teorias, hipóteses e relacionamentos, e, consequentemente, para a correção e aprimoramento das técnicas utilizadas pela equipe de desenvolvedores de software. A execução de experimentos tem como objetivos a caracterização, a avaliação, a previsão, o controle e a melhoria de produtos, processos, recursos, teorias, modelos entre outros. Segundo Travassos et al. (2002), "somente experimentos verificam teorias, somente experimentos podem explorar os fatores críticos e dar luz ao fenômeno novo para que as teorias possam ser formuladas e corrigidas".

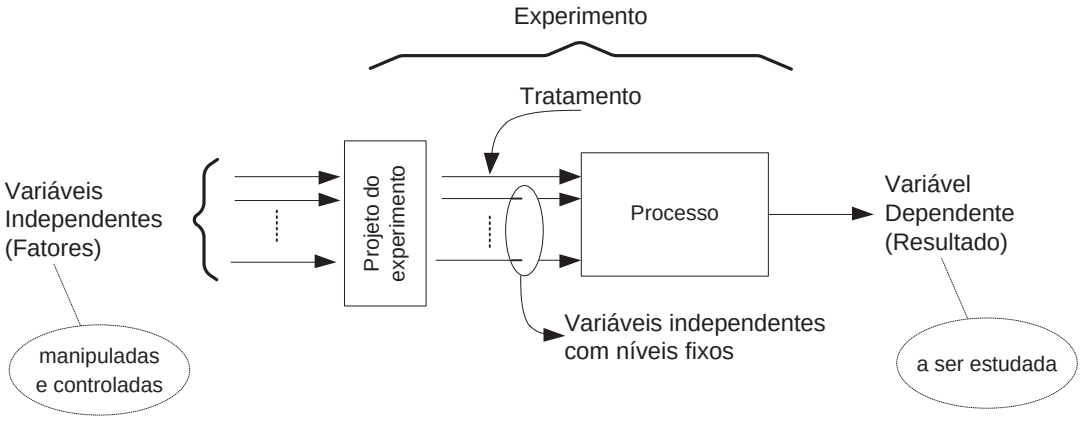


Figura 3.2: Ilustração de um experimento, adaptado de Wohlin et al. (2000)

Os elementos fundamentais de um experimento consistem em: variáveis (dependentes e independentes), objetos, participantes, contexto, hipóteses e projeto (Wohlin et al., 2000). O objetivo de tais elementos consiste em viabilizar a realização de estudos que comprovem a melhora de algum processo de desenvolvimento. Cada um deles é descrito a seguir.

- **Variáveis Independentes**: também chamadas de fatores, correspondem aos valores de entrada do experimento. Apresentam a causa dos dados do experimento. Tais variáveis são manipuladas e controladas pelo processo de experimentação (Lopes e Travassos, 2009), conforme ilustrado na Figura 3.2.

- **Variáveis Dependentes**: também chamadas de resultado, correspondem aos valores de saída do experimento. Apresentam o resultado dos dados do experimento. Geralmente, só há uma variável dependente em um experimento. Referem-se às variáveis a serem estudadas (Lopes e Travassos, 2009).

- **Objetos**: compõem a instrumentação do experimento ou ferramentas utilizadas pelos participantes da experimentação.

- **Participantes**: são as pessoas (sujeitos) que atuam no experimento, informando parâmetros ao mesmo, como valores das variáveis.

- **Contexto**: composto das condições em que o experimento é executado (ambiente de produção ou paralelo, se *in-vivo* ou *in-vitro* respectivamente).

- **Hipóteses**: suposição que se deseja avaliar.

- **Projeto**: segundo Travassos et al. (2002), determina como um experimento é conduzido. A decisão sobre alocação dos objetos e dos participantes é feita nesse momento, além da definição da maneira como os tratamentos são aplicados aos objetos.

3.1.3.1 Organização do Experimento

Os experimentos possuem suas metas definidas, para assegurar que a intenção do estudo experimental possa ser realizada durante sua execução. Também passam por um planejamento,

englobando seleção de contexto, formulação de hipótese, seleção de variáveis, seleção de participantes, projeto de estudo experimental, instrumentação e validação. Os projetos de estudos experimentais possuem alguns princípios gerais de organização (Wohlin et al., 2000, Travassos et al., 2002):

- **Aleatoriedade** (*Randomization*): os métodos estatísticos utilizados para analisar os dados requerem que as observações sejam de variáveis independentes aleatórias. É utilizada para evitar que um determinado valor interfira em outro, e também para a seleção dos participantes do experimento.

- **Agrupamento** (*Blocking*): utilizado para eliminar efeitos indesejados em um estudo, para casos em que existe um fator sobre o qual não há interesse, porém que provavelmente tem um efeito sobre a resposta. É utilizado quando existe um valor não esperado no experimento que está influenciando o resultado, gerando informações incorretas.

- **Balanceamento**: atribui para cada tratamento um número igual de participantes, contribuindo para a melhora da análise estatística.

Durante a execução dos experimentos, os tratamentos são aplicados aos participantes por meio de preparação (os participantes são escolhidos e os formulários – e demais objetos – são preparados), de execução (os participantes realizam suas tarefas de acordo com os diferentes tratamentos e os dados são coletados) e de validação dos dados (os dados coletados são validados).

Com a execução do experimento, obtém-se a medição, e com ela, as medidas de um experimento são utilizadas para que seja possível analisar o seu resultado. As hipóteses levantadas são verificadas e um dimensionamento do problema é obtido. O resultado das medidas chama-se medição, e pode pertencer às escalas[1] nominal, ordinal, intervalo ou razão (Travassos et al., 2002).

A escala *nominal* apresenta o atributo de uma entidade como o nome ou símbolo, e a classificação das entidades pode ser feita a partir dos atributos nominalmente mapeados. A escala *ordinal* ordena as entidades segundo um critério definido. Nesse caso, as afirmações como 'maior do que...' ou 'mais complexo do que...' podem ser feitas. A escala *intervalo* ordena os valores da mesma maneira que a escala *ordinal*, mas acrescenta a noção da distância relativa entre as entidades, possuindo uma unidade de medida constante e atribuindo arbitrariamente a origem (ponto zero). Na escala *razão* existe o valor do zero significativo, ou seja, possui um ponto zero único, e a razão entre medidas é significativa e invariante transformações, permitindo a realização de todas as operações aritméticas. A possibilidade de produzir as afirmações significativas, chamada também de potência da escala, cresce da escala *nominal* à escala da *razão* (Travassos et al., 2002).

[1]Travassos et al. (2002) classificam dados utilizando o termo *escala*, podendo ser nominal, ordinal, intervalo ou razão. Card et al. (1999) classificam dados utilizando o termo *tipo*, podendo ser nominal, ordinal ou quantitativo.

3.1.3.2 Validação do Experimento

Existem quatro tipos de validade que verificam, após a execução do experimento, se os dados gerados pelo mesmo são válidos. São elas: *validade de conclusão*, *validade interna*, *validade de construção* e *validade externa*.

A *validade de conclusão* diz respeito ao relacionamento entre o tratamento e o resultado, certificando que há uma relação estatística significante entre o tratamento e o resultado, ou seja, verificando se os dados gerados pelas variáveis independentes, por meio do tratamento, são relevantes para o experimento. Conclusões corretas devem ser obtidas a respeito dos relacionamentos entre o tratamento e o resultado de um experimento.

A *validade interna* é direcionada principalmente aos participantes do experimento, uma vez que verifica se algum fator não influencia as respostas fornecidas. As maneiras como as pessoas são selecionadas e divididas em classes diferentes, bem como elas são tratadas e compensadas durante o experimento na ocorrência de eventos especiais são fatores que impactam na validade interna, fazendo com que o experimento apresente um comportamento proveniente de um fator de distúrbio, e não do tratamento.

A *validade de construção* condiz ao relacionamento entre teoria e observação, verificando se a parte teórica do experimento foi formulada corretamente. Se o relacionamento entre causa e efeito é ocasional, deve-se certificar que o tratamento reflete bem a construção da causa e que o resultado reflete bem a construção do efeito.

A *validade externa* diz respeito à generalização, representando a importância do resultado do experimento para o mundo externo, ou seja, para o mundo real. Concerne à habilidade de generalizar resultados fora dos padrões do experimento, mas com a chance de ocorrer os seguintes riscos: ter participantes errados como sujeitos, conduzir o experimento no ambiente errado e realizá-lo em um momento que afete o resultado (Travassos et al., 2002, Kitchenham et al., 2002).

3.2 Processo de Experimentação

O processo de experimentação é sequencial e interativo, e define algumas atividades a serem seguidas em uma determinada ordem para que o experimento possa ser realizado, conforme ilustrado na Figura 3.3. Iniciando pela própria ideia do experimento, é necessária uma verificação desta própria ideia (ou teoria ou hipótese), e um experimento é a melhor maneira de investigá-la. Realizada e aprovada a verificação, as atividades seguintes são realizadas de acordo com o processo de experimentação, sendo este dividido em cinco principais atividades (Wohlin et al., 2000), expostas a seguir.

3.2.1 Definição de um Experimento

Na fase de definição de um experimento, sua base deve ser devidamente determinada (o porquê de conduzir o experimento), para que não haja retrabalho.

Segundo Wohlin et al. (2000), durante essa fase é importante definir um modelo do objetivo do experimento para assegurar que os aspectos importantes do experimento sejam definidos

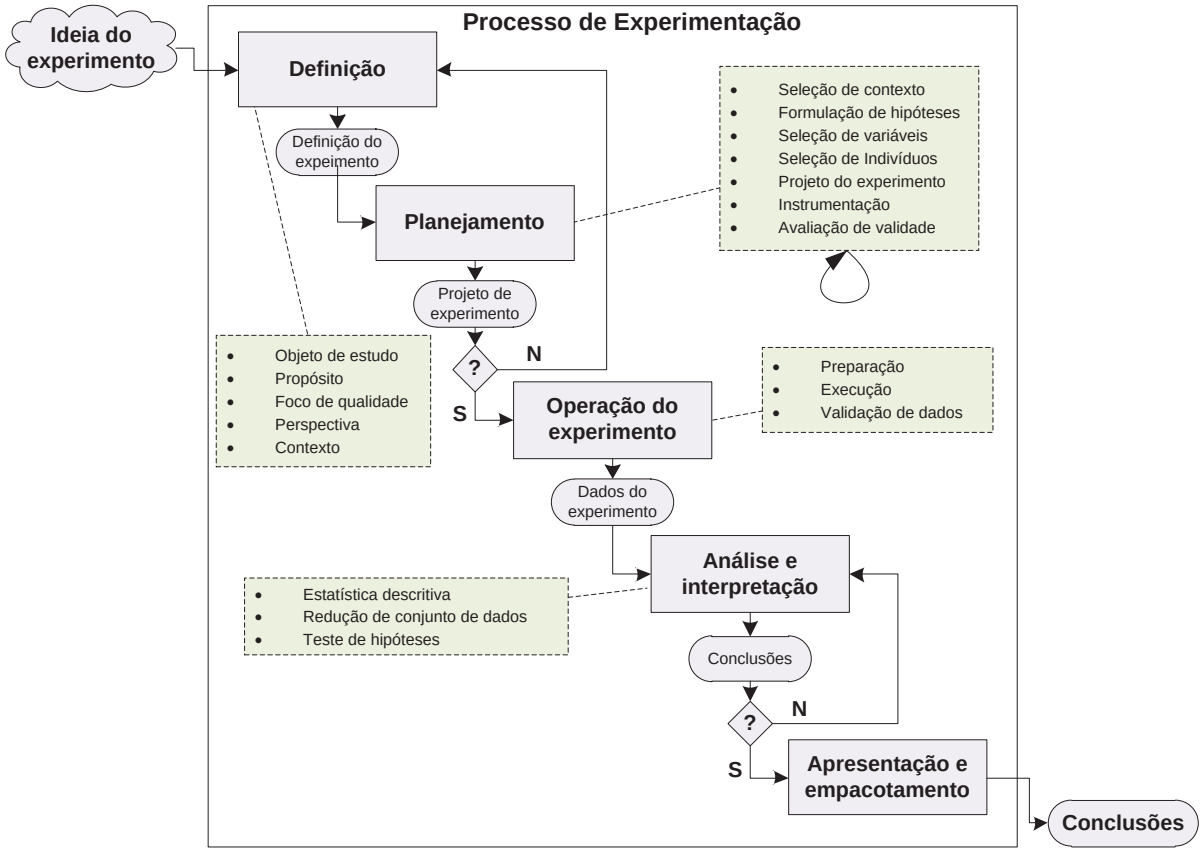


Figura 3.3: Visão do processo de experimentação, adaptado de Wohlin et al. (2000).

antes de seu planejamento de execução. À medida que o objetivo do experimento é definido, a base para formulação das hipóteses também é estabelecida. Para isso, é seguido um modelo de objetivo, com o propósito de estabelecer um alvo ao experimento: analisar o *objeto de estudo* para o *propósito* com respeito a seu *foco de qualidade* do ponto de vista da *perspectiva* dentro de um *contexto*.

O *Objeto de estudo* consiste na entidade a ser estudada dentro do experimento, podendo ser produtos, processos, modelos, métricas ou teorias, a exemplo de produto final, desenvolvimento ou inspeção de processo, crescimento de confiabilidade de modelo, implantação de nova técnica ou método de trabalho, avaliação de uma atividade laborativa, dentre outros, delimitando assim o escopo do experimento.

O *Propósito* define qual é a intenção do experimento, a exemplo de avaliação de impacto de duas técnicas diferentes, caracterizar a curva de crescimento de uma organização, obter auxílio na tomada de decisão sobre aplicação de nova ferramenta de desenvolvimento, dentre outros, formando a base para a elaboração de hipóteses do experimento na fase de planejamento.

O *Foco de qualidade* é o primeiro efeito estudado dentro do experimento, podendo ser eficácia, custo, confiabilidade, produtividade, eficiência, agilidade, cooperativismo, suporte, facilidade, acessibilidade, dentre outros. Também influencia a elaboração das hipóteses.

A *Perspectiva* consiste no ponto de vista dos quais os resultados do experimento são interpretados. Pode conter participantes tais quais desenvolvedores, analistas de sistemas, gerentes de projetos, clientes, pesquisadores, consultores, dentre outros.

O *Contexto* consiste no ambiente no qual o experimento é realizado, definindo quais indivíduos estão envolvidos, quais ferramentas (Artefatos de Software) são utilizadas e relatando a validade dos resultados do experimento. Pode-se criar uma caracterização dos indivíduos e das ferramentas, a exemplo de experiência, idade, altura e tamanho do grupo (para indivíduos), como também tamanho, complexidade, domínio da aplicação, prioridade (para ferramentas).

3.2.2 Planejamento de um Experimento

Na fase de *planejamento* é definido como o experimento deve ser conduzido. Um planejamento deve ser criado e seguido para que o experimento possa ser controlado, caso contrário, não há diferença entre um experimento (estudo controlado) e um estudo de caso (estudo observacional).

O planejamento é dividido em seis etapas iterativas: *Seleção de contexto*, *Formulação de hipóteses*, *Seleção de variáveis*, *Seleção de indivíduos*, *Projeto do experimento*, *Instrumentação* e *Avaliação de validade*.

3.2.2.1 Seleção de contexto

Com base na definição do experimento, pode-se decidir por realizar o experimento em um ambiente real (*in-vivo*) ou simulado (*in-vitro*), utilizando apoio de profissionais ou de estudantes, realizando estudos válidos a um contexto específico ou a um domínio genérico da Engenharia de Software.

A condução de um experimento em um ambiente real, com o apoio de profissionais, provê melhores resultados, mas envolve riscos e custos. Há também a opção de se conduzir o experimento paralelamente a um projeto em andamento, reduzindo os riscos, mas mantendo os custos extras. E há a opção de se conduzir o experimento com o apoio de estudantes, reduzindo os custos.

3.2.2.2 Formulação de hipóteses

O teste de hipóteses é o foco da análise estatística de um experimento, e consiste em tentar rejeitar uma hipótese formulada utilizando os dados coletados durante a condução do experimento. Se há a possibilidade de a hipótese ser rejeitada, então conclusões podem ser obtidas (Wohlin et al., 2000, Travassos et al., 2002).

A própria definição do experimento é formalizada por meio de hipóteses. Pelo menos duas hipóteses devem ser formuladas: uma nula e uma alternativa. A *hipótese nula* define que não há nenhuma tendência ou padrão subjacentes na definição do experimento. Essa é a hipótese que o experimentador tenta rejeitar (Wohlin et al., 2000).

3.2.2.3 Seleção de variáveis

As variáveis dependentes e independentes devem ser selecionadas antes de se dar início ao projeto do experimento. Com relação às variáveis independentes, não é uma tarefa fácil escolhê-las, pois geralmente requer do experimentador (ou facilitador) um conhecimento relacionado ao assunto do objetivo do experimento (Wohlin et al., 2000). Pelo fato de as variáveis (as quais estão sendo escolhidas) terem de ser independentes, elas devem ser possíveis de serem controladas, além de exercerem algum efeito sobre as variáveis a serem estudadas, ou seja, as variáveis

dependentes.

Com relação às variáveis dependentes, o efeito dos tratamentos é mensurado em cada variável dependente. Geralmente, só há uma variável dependente, a qual deve ser derivada diretamente das hipóteses.

3.2.2.4 Seleção de indivíduos

Segundo Robson (1993), a seleção das pessoas é importante na condução de um experimento. A seleção das pessoas que participam da experimentação está diretamente ligada à generalização dos resultados do experimento, podendo tal seleção ser probabilística, quando é conhecida a probabilidade de selecionar cada indivíduo, e não probabilística, quando essa probabilidade é desconhecida.

Um fator a ser considerado é o tamanho da amostra ou seleção das pessoas que deseja generalizar nos resultados do experimento. Uma amostragem maior é necessária caso haja uma grande variabilidade estatística dentro da população disponível, assim como a análise dos dados também pode influenciar na escolha do tamanho da amostragem. Em geral, quanto maior a amostragem, menor a ocorrência de erros nos resultados do experimento (Wohlin et al., 2000).

3.2.2.5 Projeto do experimento

Após a definição das questões ou problemas a serem investigados pelo experimento durante a definição do objetivo (fase de definição) e formulação de hipóteses (fase de planejamento), tendo todas as ideias a respeito do experimento bem definidas, deve-se projetar o experimento, tendo por base as questões assumidas (utilização de pessoas, ferramentas, escalas de medida, dentre outras). Segundo Wohlin et al. (2000), um estudo bem projetado pode ser aproveitado em projetos futuros, formando uma base de conhecimento.

Alguns aspectos devem ser considerados ao se projetar um experimento, dentre eles, aleatoriedade, agrupamento e balanceamento (vide Seção 3.1.3.1). Para a maioria desses experimentos, uma hipótese de exemplo é formulada e modelos de análise estatística são sugeridos para cada projeto.

3.2.2.6 Instrumentação

O objetivo geral da instrumentação é prover meios de realizar e monitorar a experimentação sem afetar seu controle. Antes da execução do experimento, os instrumentos escolhidos na fase de planejamento são preparados para o experimento – por exemplo, documentos, *checklists*, fluxogramas, dentre outros. Os participantes devem ser treinados a utilizar corretamente tais instrumentos (Wohlin et al., 2000).

As medições em experimentos são conduzidas por meio de coleta de dados. Em caso de experimentos com uso intenso de pessoas, os dados são geralmente coletados por meio de questionários e entrevistas. Nesses casos, há um planejamento a ser feito: preparar formulários e questionários, validá-los e disponibilizá-los aos participantes.

Os resultados do experimento não devem ser afetados pela instrumentação, ou seja, eles devem ser os mesmos, independentemente de como a instrumentação é utilizada no experimento. Caso contrário, os resultados são inválidos (Wohlin et al., 2000).

3.2.2.7 Avaliação de validade

Um aspecto fundamental que envolve um experimento é verificar o quão válidos são os resultados. Essa verificação é feita pelas análises das ameaças e validades que um experimento está exposto em decorrência de seu projeto. São elas: validade de conclusão, validades internas, validades de construção e validades externas.

3.2.3 Operação de um Experimento

Durante a fase operacional, os tratamentos são aplicados aos sujeitos, ou seja, o experimento é posto em prática. Como a maioria dos experimentos na Engenharia de Software envolve pessoas, elas devem ser devidamente preparadas para que se empenhem no experimento, a fim de que o mesmo apresente resultados válidos (Wohlin et al., 2000).

Conforme ilustrado na Figura 3.4, a fase operacional é dividida em três etapas: *Preparação*, *Execução* e *Validação de dados*. Todas estas etapas são influenciadas pelo comportamento dos sujeitos envolvidos. Por isso, é necessário convencê-los e motivá-los a participar do experimento de maneira efetiva (Wohlin et al., 2000).

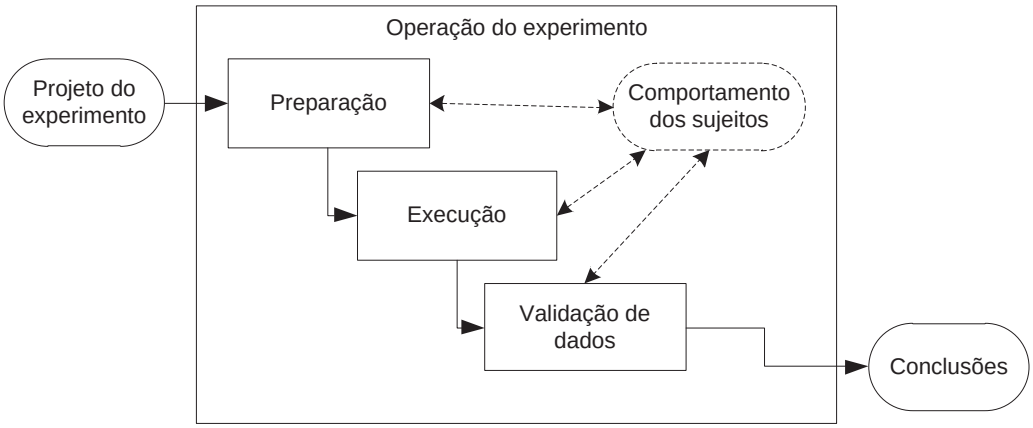


Figura 3.4: Etapas da operação de um experimento, adaptado de Wohlin et al. (2000)

Na etapa de *preparação*, o passo inicial consiste em escolher e informar os participantes a respeito do experimento e preparar o material necessário. Segundo Wohlin et al. (2000), é importante que as pessoas estejam motivadas a participar do experimento e que o facilitador ou responsável pelo experimento facilite a inserção do participante no ambiente de pesquisa no qual ele será submetido, alocando, quando possível, tarefas as quais ele já está habituado (por exemplo, escrever linhas de código em uma linguagem que o participante já conheça). Todos os instrumentos e ferramentas devem estar preparados para o experimento, e os participantes aptos para utilizar tais instrumentos.

Na etapa de *execução*, o experimento pode ser executado de maneiras diferentes, em várias reuniões ou apenas em uma, em reuniões curtas ou reuniões longas, em agrupamento de pessoas ou com pessoas isoladas ou grupos isolados, em atividades presenciais ou não presenciais. A maneira como a execução é realizada depende do objetivo do experimento.

A coleta dos dados pode ser feita tanto pelos participantes quanto pelo experimentador, manualmente ou com o uso de ferramentas informatizadas. Segundo Wohlin et al. (2000), se a

coleta é feita manualmente pelos participantes, não há como testar diretamente inconsistência nos dados (somente quando da intervenção do experimentador ou de alguma questão do participante).

Na etapa de *validação dos dados*, segundo Wohlin et al. (2000), é tarefa do experimentador ou facilitador verificar se os participantes participaram do experimento de maneira efetiva. Essa tarefa é feita pela verificação e validação dos dados informados por eles, para que os resultados do experimento sejam válidos.

3.2.4 Análise e Interpretação de um Experimento

Na fase de *Análise e Interpretação*, os dados coletados durante a fase de execução são interpretados para que o experimentador possa chegar a uma conclusão. Para que seja possível obter essa conclusão, os dados são interpretados de maneira quantitativa por meio de técnicas de estatística descritiva, redução de conjunto de dados e testes de hipóteses.

3.2.4.1 Estatística descritiva

O objetivo da estatística descritiva é obter um senso de como os dados estão distribuídos, além de descrever e apresentar graficamente alguns aspectos dos dados (por exemplo, em que posição de alguma escala os dados são positivos e quão concentrados ou espalhados eles estão).

3.2.4.2 Redução de Conjunto de Dados

Após a aplicação dos métodos estatísticos, é necessária uma verificação para confirmar se algum dado, estatisticamente representado, realmente representa uma informação correta ou esperada.

Uma boa maneira de se identificar erros no conjunto de dados consiste em projetar gráficos de dispersão e *boxplot*, tornando as distorções (*outliers*) visíveis. Identificadas as distorções (não apenas com base nas coordenadas dos gráficos), deve-se decidir o que se fazer com elas. Se uma distorção é causada por um raro evento, o qual nunca ocorre novamente dentro do contexto do experimento, ela pode ser removida. Se a distorção é causada por uma variável que não foi considerada anteriormente, pode-se optar por removê-la ou não, considerando os cálculos e modelos a serem refeitos (Wohlin et al., 2000).

3.2.4.3 Testes de hipóteses

O conceito básico do teste de hipóteses é verificar a possibilidade de rejeitar uma determinada hipótese nula, ou seja, uma hipótese que descreve algumas propriedades as quais o experimentador tenta rejeitá-las. Como as hipóteses englobam distribuições estatísticas, o experimentador se habilita de vários testes, técnicas, métodos e cálculos estatísticos na tentativa de rejeitar a hipótese nula. Dentre elas: I-Test, Mann-Whitney, F-test, T-Test pareado, Wilcoxon, Teste de sinal ANOVA, Kruskal-Wallis, Chi-2 e Teste-Z (Wohlin et al., 2000, Devore, 2006).

Conforme o experimentador consegue ou não rejeitar a hipótese nula, outras hipóteses vão sendo elaboradas e o experimentador pode desenvolver suas conclusões de acordo com os resultados. Se as hipóteses são rejeitadas, são obtidas conclusões baseadas na influência das variáveis independentes sobre as dependentes, informando que o experimento é válido. Se as hipótese nulas não podem ser rejeitadas, não é possível obter conclusões, apenas a de que não

há uma diferença significante em termos estatísticos entre os tratamentos das variáveis, mas as lições aprendidas são importantes em termos práticos.

3.2.5 Apresentação e Empacotamento de um Experimento

De acordo com Mendonça et al. (2008), replicações experimentais são muito importantes para o progresso da Engenharia de Software Experimental, sendo um dos mecanismos-chave para confirmar descobertas experimentais anteriores, além de serem utilizadas para transferir conhecimento experimental, treinar participantes e expandir uma base de evidências experimentais. Mendonça et al. (2008) ressaltam a necessidade de se utilizar padronizações de empacotamento para que o experimento possa ser devidamente replicado por outros grupos de pesquisadores.

Os resultados obtidos da condução de um experimento devem ser empacotados (organizados) para posteriores divulgações (apresentações) e replicações. Garcia et al. (2008) apresentam uma proposta de ontologia que pode ser utilizada para guiar a organização do Pacote de Laboratório. De acordo com Carver (2010), é necessário seguir um protocolo que possibilite determinar não apenas a validade de seus resultados, mas também sua replicação.

Dentre os itens a serem empacotados, alguns merecem destaque. Os resultados e conclusões devem ser documentados, tanto para fins de divulgação quanto de replicação. O projeto do experimento, quando bem documentado, facilita sua replicação, incluindo objetivos, hipóteses, variáveis, tratamentos, verificações, dados coletados, análises, material de treinamento, artefatos, problemas encontrados, lições aprendidas, sugestões de evolução do material utilizado, dentre outros.

3.2.6 Considerações Finais

Experimentação na área da pesquisa científica denota uma atividade laboratorial (Travassos et al., 2002), e a Engenharia de Software Experimental trata a Engenharia de Software como uma ciência de laboratório, consistindo de um estudo concernente à Engenharia de Software, utilizando o método científico de condução de experimentos – verificando uma teoria inicial de causa e efeito – para caracterizar, avaliar, prever, controlar e melhorar o Processo de Software.

De acordo com Silva (2010), a Engenharia de Software Experimental consiste em um estudo sobre a própria Engenharia de Software, provando na prática o que a teoria da Engenharia de Software prega.

A condução de um experimento segue um determinado processo de experimentação. Esse processo engloba a realização de diversas atividades (cuja quantidade e complexidade variam de acordo com as características do estudo) no decorrer das fases de experimentação.

Um experimento não prova teoria alguma, mas a formula e a verifica (Wohlin et al., 2000), além de ajudar a determinar a eficácia de métodos e teorias propostas (Zelkowitz e Wallace, 1998). Para isso, o processo de experimentação provê um método controlado e disciplinado para a condução de um estudo por meio de seus principais elementos: variáveis, objetos, sujeitos (ou participantes), contexto do experimento, hipóteses e o projeto do experimento.

Neste trabalho, o objetivo é a avaliação da ferramenta a $SoftVis_{OAH}$. Considerando que a abordagem de entendimento de programa envolve o fator humano, é necessária a condução

de um experimento, especialmente para avaliar técnicas de entendimento de programa (Lucca e Penta, 2006). Há a necessidade de coletar dados de utilização da ferramenta de uma maneira que estes possam ser devidamente analisados, comparados e utilizados em cálculos estatísticos. Desta maneira, foi necessária a condução de um estudo experimental controlado – conduzido seguindo o processo de experimentação proposto por Wohlin et al. (2000) – para avaliar a eficácia e a eficiência da ferramenta de uma maneira mais realista, além de obter indícios de suas vantagens, permitindo aos participantes explorar os recursos da ferramenta.

CAPÍTULO

4

Experimento para Avaliação da Ferramenta de Visualização de Software

Para avaliar a eficácia e a eficiência do uso da ferramenta $SoftVis_{OAH}$, um estudo experimental controlado foi conduzido utilizando o Processo de Experimentação proposto por Wohlin et al. (2000). Um estudo piloto foi conduzido inicialmente, pois não havia disponibilidade de um grupo de prováveis participantes (um mínimo de 12 pessoas) para repetir o experimento em caso de alguma ameaça grave à sua validade. Assim, o objetivo desse estudo piloto consiste essencialmente em avaliar o projeto experimental, assim como o material utilizado como instrumentação, servindo como uma preparação para o experimento final (Maldonado et al., 2006).

Tanto no experimento piloto quanto no final, os participantes utilizaram a ferramenta e proveram medições de *feedback* que permitem derivar métricas quantitativas e avaliações qualitativas. A concepção do experimento foi a mesma para ambos, e é apresentada na Seção 4.1. O projeto experimental do estudo piloto, assim como as lições aprendidas com a sua condução são apresentadas na Seção 4.3. A partir dessas lições, foram feitas adequações no projeto experimental e no material utilizado para o experimento, apresentadas na Seção 4.4 juntamente com os resultados obtidos.

4.1 Definição do Experimento

Conforme apresentado no Capítulo 3, nesta etapa são definidos os objetivos e as hipóteses do experimento; é realizada a seleção das variáveis e a seleção dos indivíduos; o projeto do experimento é definido e a instrumentação a ser utilizada é preparada; os métodos de análise de dados são selecionados; e a validade da experimentação é analisada. Nesta seção, todos esses itens são apresentados.

4.1.1 Objetivos do Experimento

O objetivo do experimento, definido seguindo o paradigma de melhoria contínua GQM (*Goal/-Question/Metric*), é descrito a seguir:

- Objeto de Estudo:
  - Ferramenta de Visualização de Software $SoftVis_{OAH}$
- Propósito:
  - Avaliar e comparar a utilização da ferramenta $SoftVis_{OAH}$ como apoio a tarefas de Revisão de Programa em relação à Revisão de Programa *ad hoc*, conforme as seguintes atividades:
    - Revisão de Programa *ad hoc* – entendimento de programa e localização de defeitos (indicação do local do defeito no código fonte) sem o apoio da ferramenta $SoftVis_{OAH}$, apenas com o apoio do ambiente de desenvolvimento *Eclipse SDK*;
    - Revisão de Programa com visualização – entendimento de programa e localização de defeitos com o apoio da ferramenta $SoftVis_{OAH}$, além do apoio do ambiente de desenvolvimento *Eclipse SDK*.
- Enfoque de Qualidade:
  - Tempo;
  - Quantidade de defeitos localizados;
  - Entendimento do programa.
- Perspectiva:
  - Do experimentador.
- Contexto:
  - Laboratório de informática da Universidade Estadual Paulista Júlio de Mesquita Filho, Unesp, no campus da Faculdade de Ciências e Tecnologia.

Assim sendo, o objetivo segundo o paradigma *GQM*, consiste em analisar a ferramenta de Visualização de Software $SoftVis_{OAH}$ com o propósito de avaliar a sua utilização (comparando a Revisão de Programa *ad hoc* e a Revisão de Programa apoiada pela ferramenta) com respeito ao tempo e à quantidade de defeitos localizados, da perspectiva do experimentador, no contexto de um laboratório de pesquisa.

É importante frisar que, nesta pesquisa, *localização de defeitos* refere-se à atividade de encontrar trechos de código fonte em que um dado defeito se encontra. Os participantes não

precisam realizar tarefas de detecção de defeitos[1], pois existem outras ferramentas com o objetivo de auxiliar tal tarefa (Lemos et al., 2009, Ferrari et al., 2010). Para efetuar a localização de defeitos, uma lista de defeitos inseridos no código fonte é fornecida aos participantes, que devem indicar onde o defeito está localizado.

4.1.2 Questões e Métricas

As seguintes questões – identificadas pela letra *Q* – devem ser verificadas e as respectivas métricas – identificadas pela letra *M* – devem ser aplicadas para possibilitar responder às questões:

- **Q1**: A utilização da ferramenta no apoio à Compreensão de Programa auxilia o entendimento do programa?
  - **M1.1**: Quantidade de *adendos* (de todos os *aspectos*) contidos no código fonte.
  - **M1.2**: Quantidade de *adendos* descritos corretamente.
- **Q2**: A utilização da ferramenta no apoio à Localização de Defeitos torna a localização mais eficaz (maior quantidade)?
  - **M2.1**: Quantidade de defeitos previamente inseridos no código fonte.
  - **M2.2**: Quantidade de defeitos localizados por cada revisor de programa.
- **Q3**: A utilização da ferramenta no apoio à Localização de Defeitos torna a localização de defeitos mais eficiente (mais rápida)?
  - **M3.1**: Tempo médio (em minutos) despendido em cada localização de defeitos.
  - Métricas *M2.1* e *M2.2*.

4.2 Planejamento

O estudo experimental foi planejado conforme as seções a seguir, de acordo com o Processo de Experimentação proposto por Wohlin et al. (2000).

4.2.1 Formulação de Hipóteses

A utilização da ferramenta como apoio à Revisão de Programa deve ser analisada tanto em relação ao auxílio provido ao entendimento do programa quanto à eficácia e à eficiência providas na localização dos defeitos, sendo que a eficácia na localização de defeitos é um indício de que o participante entendeu o programa. As hipóteses a serem contrastadas referem-se, portanto, ao auxílio à Compreensão de Programa, à eficácia na localização de defeitos e à eficiência na localização de defeitos. As hipóteses relativas ao auxílio à Compreensão de Programa são:

[1] Revelar a existência de um defeito é objetivo da atividade de teste

- **Hipótese Nula (H_0^1)**: A utilização da ferramenta $SofVis_{OAH}$ não provê auxílio à Compreensão de Programa.
  (Auxílio-CP_{VS}) = $\emptyset$
  De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.1, se a quantidade de *adendos* descritos corretamente com auxílio da ferramenta for menor que ou igual à quantidade de *adendos* descritos corretamente sem ela (*ad hoc*), a *Hipótese Nula (H_0^1)* não pode ser rejeitada, ou seja, a utilização da $SoftVis_{OAH}$ não provê auxílio à Compreensão de Programa. Esta hipótese nula baseia-se na quantidade de descrições corretas de *adendos* contidos em *aspectos* – considera-se que, se o participante tiver descrito corretamente os *adendos/aspectos*, ele entendeu o código fonte (referente aos *aspectos* e respectivas *classes* entrecortadas) para prover tal descrição. Para a verificação desta hipótese, utiliza-se a questão *Q1* e as respectivas métricas *M1.1* e *M1.2*. Na Figura 4.1, observa-se que na relação de causa e efeito há uma dependência, indicada com uma seta tracejada, de conhecimentos prévios necessários à realização da compreensão do programa, são eles: o conhecimento dos participantes em *Eclipse*, em *JUnit*, em *POA*. Além desses, o conhecimento sobre técnicas de *Visualização de Software* e, especificamente, o uso da ferramenta de $SofVis_{OAH}$, se faz necessário para Compreensão de Programa com o apoio da ferramenta. Vale ressaltar que, a indicação de tais pré-requisitos nessa relação de causa e efeito tem por objetivo indicar a necessidade do projeto experimental em prover os conceitos necessários para os participantes realizarem adequadamente as atividades, isolando assim uma ameaça de validade. A mesma indicação de pré-requisitos é observada nas demais figuras que representam as relações de causa e efeito das demais hipóteses.

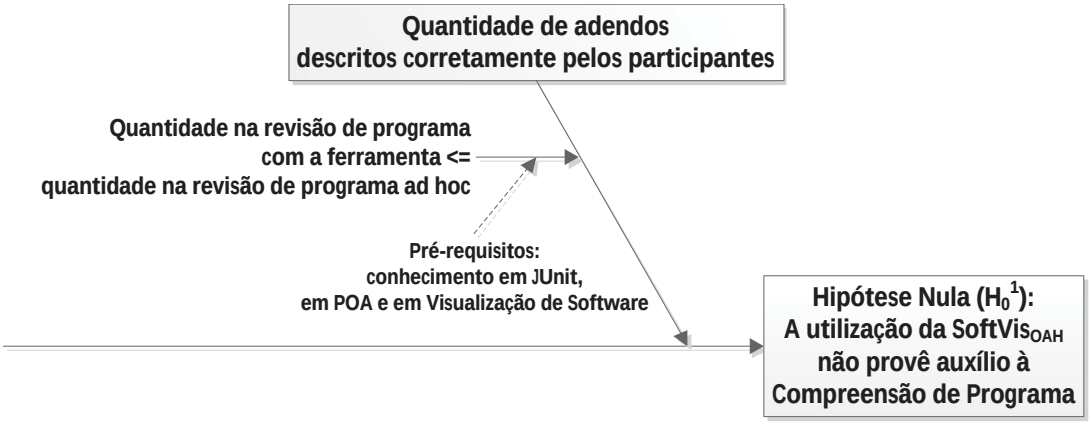


Figura 4.1: Diagrama de Causa e Efeito para a Hipótese Nula (H_0^1)

- **Hipótese Alternativa (H_1^1)**: A utilização da ferramenta $SofVis_{OAH}$ provê auxílio à Compreensão de Programa.
  (Auxílio-CP_{VS}) $\neq \emptyset$
  De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.2, se a quantidade de *adendos* descritos corretamente com auxílio da ferramenta for maior que a quantidade de *adendos* descritos corretamente sem ela (*ad hoc*), a *Hipótese Nula (H_0^1)* pode ser re-

jeitada, ou seja, a utilização da $SoftVis_{OAH}$ provê auxílio à Compreensão de Programa. Para a verificação desta hipótese alternativa, utiliza-se a mesma questão e respectivas métricas utilizadas na *Hipótese Nula (H_0^1)*.

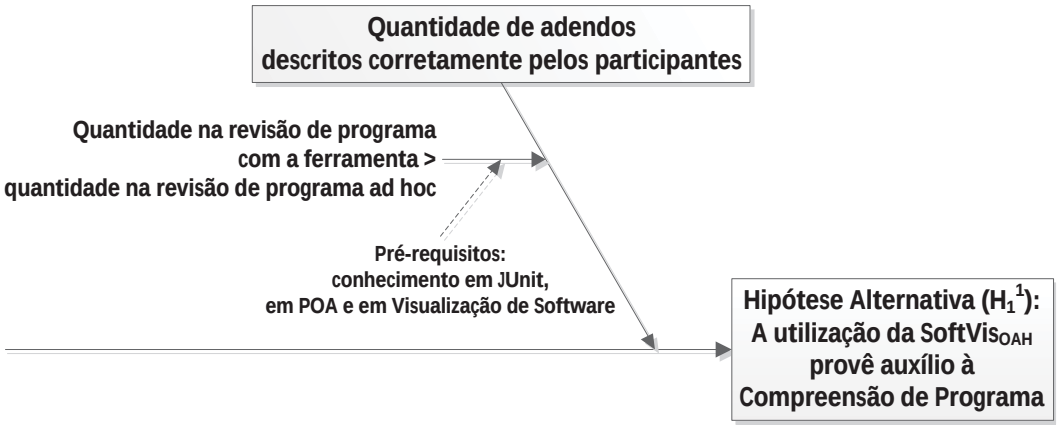


Figura 4.2: Diagrama de Causa e Efeito para a Hipótese Alternativa (H_1^1)

As hipóteses relativas à eficácia na localização de defeitos são:

- **Hipótese Nula (H_0^2)**: A utilização da ferramenta $SofVis_{OAH}$ não proporciona maior eficácia que a localização de defeitos *ad hoc*.
  (Eficácia-LD$_{VS}$) $\leq$ (Eficácia-LD$_{AH}$)
  De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.3, se a quantidade de defeitos localizados com o auxílio da ferramenta for menor que ou igual à quantidade de defeitos localizados sem ela (*ad hoc*), a *Hipótese Nula (H_0^2)* não pode ser rejeitada, ou seja, a utilização da $SoftVis_{OAH}$ não proporciona maior eficácia para a localização de defeitos. Esta hipótese baseia-se na quantidade de defeitos localizados por cada participante. Para a verificação desta hipótese, utiliza-se a questão *Q2* e as respectivas métricas *M2.1* e *M2.2*.

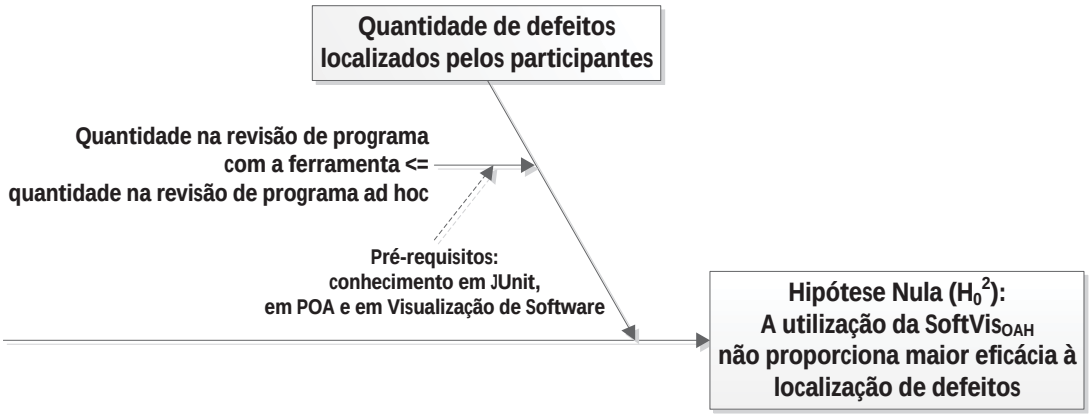


Figura 4.3: Diagrama de Causa e Efeito para a Hipótese Nula (H_0^2)

- **Hipótese Alternativa (H_1^2)**: A utilização da ferramenta $SofVis_{OAH}$ proporciona maior eficácia que a localização de defeitos *ad hoc*.

(Eficácia-LD$_{VS}$) > (Eficácia-LD$_{AH}$)
De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.4, se a quantidade de defeitos localizados com o auxílio da ferramenta for maior que a quantidade de defeitos localizados sem ela (*ad hoc*), a *Hipótese Nula (H_0^2)* pode ser rejeitada, ou seja, a utilização da $SoftVis_{OAH}$ proporciona maior eficácia à localização de defeitos. Para a verificação desta hipótese alternativa, utiliza-se a mesma questão e respectivas métricas utilizadas na *Hipótese Nula (H_0^2)*.

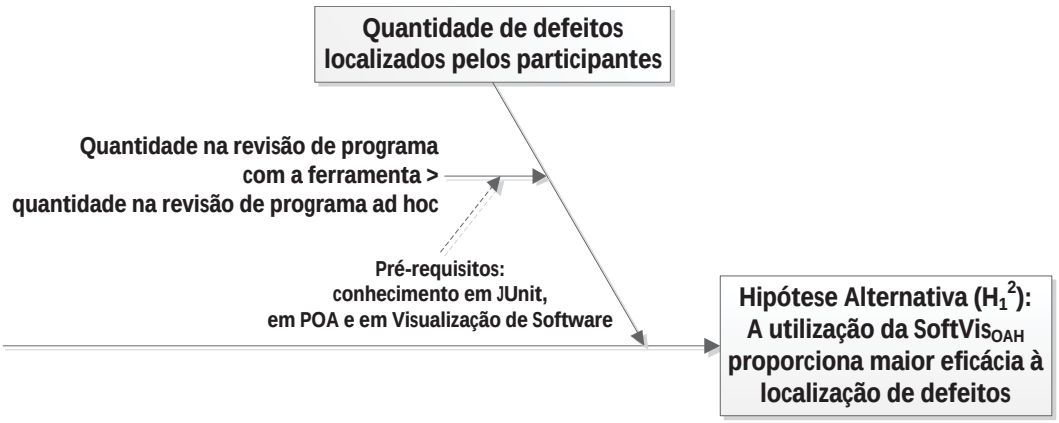


Figura 4.4: Diagrama de Causa e Efeito para a Hipótese Alternativa (H_1^2)

As hipóteses relativas à eficiência na localização de defeitos são:

- **Hipótese Nula (H_0^3)**: A utilização da ferramenta $SofVis_{OAH}$ não proporciona maior eficiência que a localização de defeitos *ad hoc*.
  (Eficiência-LD$_{VS}$) $\leq$ (Eficiência-LD$_{AH}$)
  De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.5, se o tempo despendido pelos participantes para localizar os defeitos com o auxílio da ferramenta for menor que ou igual ao tempo despendido para localizar os defeitos localizados sem ela (*ad hoc*), a *Hipótese Nula (H_0^3)* não pode ser rejeitada, ou seja, a utilização da $SoftVis_{OAH}$ não proporciona maior eficiência para a localização de defeitos. Esta hipótese baseia-se no tempo médio dispendido em cada localização de defeito, e para a verificação desta hipótese, utiliza-se a questão *Q3* e as respectivas métricas *M3.1*, *M2.1* e *M2.2*.

- **Hipótese Alternativa (H_1^3)**: A utilização da ferramenta $SofVis_{OAH}$ proporciona maior eficiência que a localização de defeitos *ad hoc*.
  (Eficiência-LD$_{VS}$) > (Eficiência-LD$_{AH}$)
  De acordo com seu diagrama de causa e efeito, apresentado na Figura 4.6, se o tempo despendido pelos participantes para localizar os defeitos com o auxílio da ferramenta for maior que o tempo despendido para localizar os defeitos localizados sem ela (*ad hoc*), a *Hipótese Nula (H_0^3)* pode ser rejeitada, ou seja, a utilização da $SoftVis_{OAH}$ proporciona maior eficiência à localização de defeitos. Para a verificação desta hipótese alternativa, utiliza-se a mesma questão e respectivas métricas utilizadas na *Hipótese Nula (H_0^3)*. O

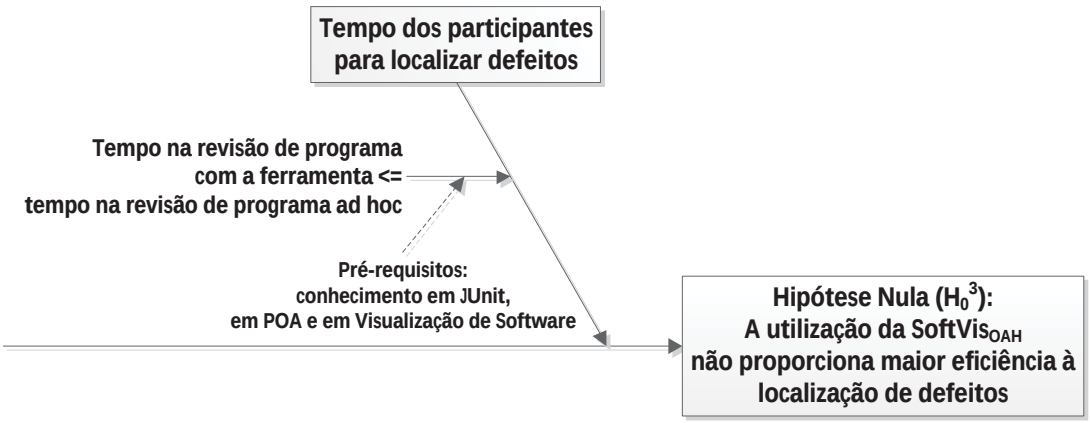


Figura 4.5: Diagrama de Causa e Efeito para a Hipótese Nula (H_0^3)

conhecimento dos participantes em *Eclipse*, em *JUnit*, em *POA* e em Visualização de Software é um pré-requisito para a realização das atividades de localização de defeitos.

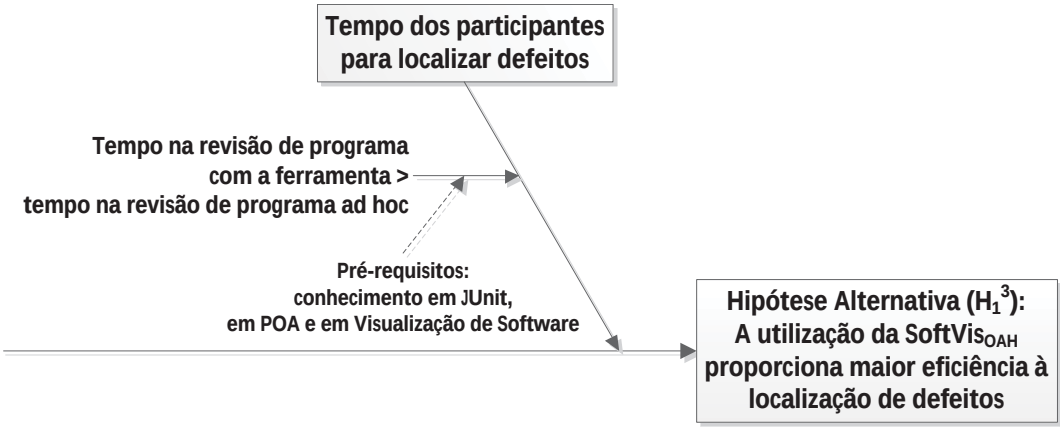


Figura 4.6: Diagrama de Causa e Efeito para a Hipótese Alternativa (H_1^3)

4.2.2 Seleção de Variáveis

As variáveis consideradas no experimento foram definidas de acordo com os requisitos para a utilização da ferramenta $SoftVis_{OAH}$ e com as métricas definidas para avaliar o seu uso. Foram classificadas da seguinte maneira:

- Variáveis independentes:
  - Quantidade de defeitos previamente inseridos no código-fonte. Essa quantidade permite verificar o índice de eficácia e de eficiência da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ e da Revisão de Programa *ad hoc*.
  - Quantidade de *adendos* de *aspectos* contidos no código fonte. Essa quantidade permite verificar o nível de auxílio provido à Compreensão de Programa pela Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ e pela Revisão de Programa *ad hoc*.
- Variáveis dependentes:

- Quantidade de defeitos localizados por meio da Revisão de Programa *ad hoc* e da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$. Cada participante deve localizar o máximo de defeitos para a avaliação de ambas as revisões de programa.
- Tempo (em minutos) para a localizar os defeitos por meio da Revisão de Programa *ad hoc* e da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$. Cada participante deve localizar cada defeito em um menor tempo para a avaliação de ambas as revisões de programa.
- Nível de auxílio provido à Compreensão de Programa da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ e da Revisão de Programa *ad hoc*. Esse nível é utilizado na análise de a partir de qual revisão de programa os participantes obtiveram um maior auxílio ao entendimento do programa por meio da descrição de cada *adendo* contido em cada *aspecto* do artefato de software.
  $\frac{AdendosDescritosCorretamente}{AdendosTotais}$
- Índice de eficácia da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ e da Revisão de Programa *ad hoc*. Esse índice é utilizado na análise da eficácia (quantidade de defeitos localizados) de ambas as revisões de programa.
  $\frac{DefeitosLocalizados}{DefeitosInseridos}$
- Índice de eficiência da Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ e da Revisão de Programa *ad hoc*. Esse índice é utilizado na análise da eficiência (tempo despendido na localização de cada defeito) de ambas as revisões de programa.
  $\frac{DefeitosLocalizados}{DefeitosInseridos}$ / TempoMédioEmMinutos

4.2.3 Seleção de Indivíduos

Os indivíduos foram escolhidos mediante seleção determinística, envolvendo alunos de graduação em Ciência da Computação (estágio avançado) e de mestrado em Ciência da Computação. Tais alunos têm o conhecimento necessário em Programação Orientada a Objetos (elementos e definições básicas, como *objeto*, *herança*, *classe*, *método* e *associação*) e conhecimento teórico em atividades de revisão e testes de programas. A maioria deles tem conhecimento necessário em *Java* (implementação dos elementos e dos conceitos da Orientação a Objetos) para participar nas atividades do experimento. Como a maioria não tem conhecimento em Programação Orientada a Aspectos, em *JUnit* e em técnicas de *Visualização de Software*, que também são necessários, foram realizados treinamentos para prover condições mínimas para a realização das atividades – tais treinamentos são apresentados na Seção 4.3.1.

4.2.4 Instrumentação

Os instrumentos utilizados na condução do experimento foram definidos de acordo com os requisitos tecnológicos para a execução de ambas as revisões de programa e com os requisitos de conhecimento prévio para a realização das mesmas. A instrumentação é dividida nos seguintes

grupos: *Ferramentas Informatizadas* (identificadas pelas letras *FI* e descritas na Tabela 4.1), *Materiais de Realização de Testes* (identificados pelas letras *RT* e descritos na Tabela 4.2), *Materiais de Treinamento* (identificados pelas letras *MT* e descritos na Tabela 4.3), *Questionários* (identificados pelas letra *Q* e descritos na Tabela 4.4) e *Formulários* (identificados pelas letra *F* e descritos na Tabela 4.5).

Tabela 4.1: Instrumentação – Ferramentas Informatizadas

FI01	Ambiente Java – necessário para as revisões de programa e execução da ferramenta $SoftVis_{OAH}$.
FI02	Eclipse SDK – necessário para as localizações de defeitos nos artefatos de software.
FI03	Plugin *AJDT* (*AspectJ Development Tool*) para o *Eclipse* – Programação Orientada a Aspectos em *AspectJ* para o Eclipse.
FI04	$SoftVis_{OAH}$ – Ferramenta de Visualização de Software.

Tabela 4.2: Instrumentação – Materiais de Realização de Testes

RT01	Artefato de software *GanttProject* com defeitos pré-inseridos – escolhido pelo fato de ser um projeto de código aberto construído no Eclipse (o qual permite a instalação do *AspectJ*).
RT02	Artefato de software *Memoranda* com defeitos pré-inseridos – escolhido pelo fato de ser um projeto de código aberto construído no Eclipse (o qual permite a instalação do *AspectJ*).
RT03	Relação de defeitos inseridos no artefato *RT01* – cada participante deve informar a localização de cada defeito contido nessa relação.
RT04	Relação de defeitos inseridos no artefato *RT02* – cada participante deve informar a localização de cada defeito contido nessa relação.
RT05	Artefatos de software para treinamento na ferramenta – é necessário um treinamento prático na ferramenta $SoftVis_{OAH}$, onde cada participante visualizará alguns artefatos de software.

Tabela 4.3: Instrumentação – Materiais de Treinamento

MT01	Slides de apresentação do experimento.
MT02	Slides de treinamento em Visualização de Software e Compreensão de Programa.
MT03	Informativo de preenchimento de questionários e formulários – especifica como preencher devidamente os questionários e formulários utilizados.
MT04	Slides de treinamento em Programação Orientada a Aspectos.
MT05	Slides de treinamento em *JUnit*.
MT06	Slides de treinamento da ferramenta $SoftVis_{OAH}$.

4.2.4.1 Artefatos de Software

Dois Artefatos de Software foram utilizados (o *GanttProject* e o *Memoranda*), ambos são projetos de código aberto desenvolvidos em *Java*. O *GanttProject* consiste em um sistema de

Tabela 4.4: Instrumentação – Questionários

Q01	Levantamento de perfil – para caracterização dos participantes.
Q02	*Feedback* do treinamento em Programação Orientada a Aspectos.
Q03	*Feedback* do treinamento em *JUnit*.
Q04	*Feedback* do treinamento em Visualização de Software e Compreensão de Programa.
Q05	*Feedback* do treinamento na ferramenta $SoftVis_{OAH}$.
Q06	*Feedback* da revisão de programa *ad hoc*.
Q07	*Feedback* da revisão de programa com o apoio da ferramenta $SoftVis_{OAH}$.

Tabela 4.5: Instrumentação – Formulários

F01	Localização de defeitos e Compreensão de Programa – os participantes devem informar cada defeito localizado, o horário de cada localização e a descrição do funcionamento de cada *adendo* contido em cada *aspecto* do artefato de software.
F02	Termo de consentimento de participação no experimento – cada participante deve preencher este termo de consentimento, elaborado com o auxílio de um advogado.

gerenciamento de projetos e está disponível em <http://www.ganttproject.biz>. O *Memoranda* consiste em um sistema de controle de agenda e de apontamentos pessoais e encontra-se disponível a partir de <http://memoranda.sourceforge.net>. Eles foram escolhidos devido ao fato de os participantes selecionados terem conhecimento prévio a respeito de suas funcionalidades, assim os participantes não têm dificuldades em entender a descrição do defeito para a localização. A interface principal das duas aplicações é apresentada nas figuras 4.7 e 4.8.

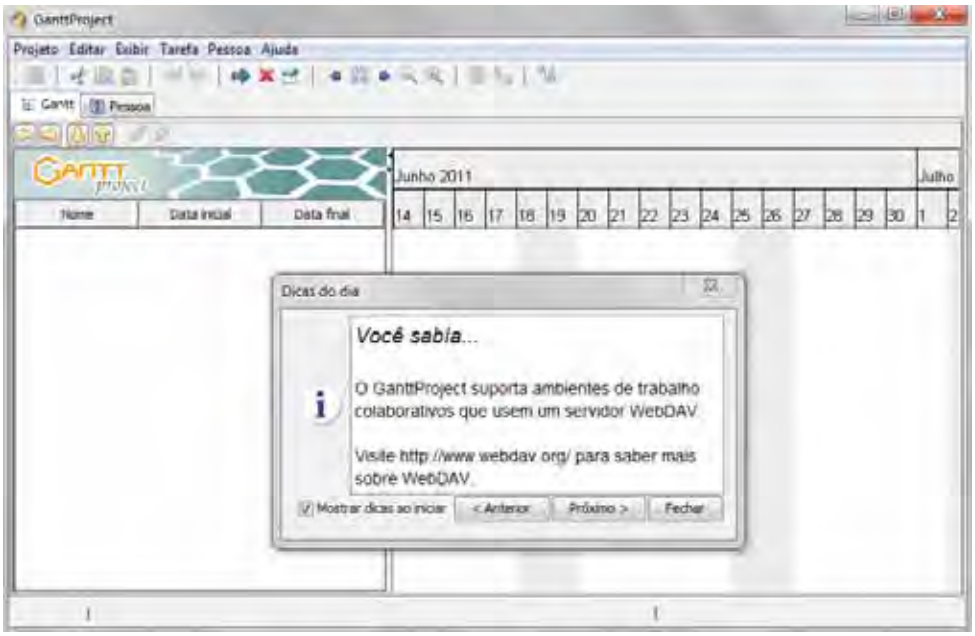


Figura 4.7: Interface principal do artefato de software *GanttProject*.

Originalmente escritos em *Java*, os sistemas foram modificados (fatorando-se algumas fun-

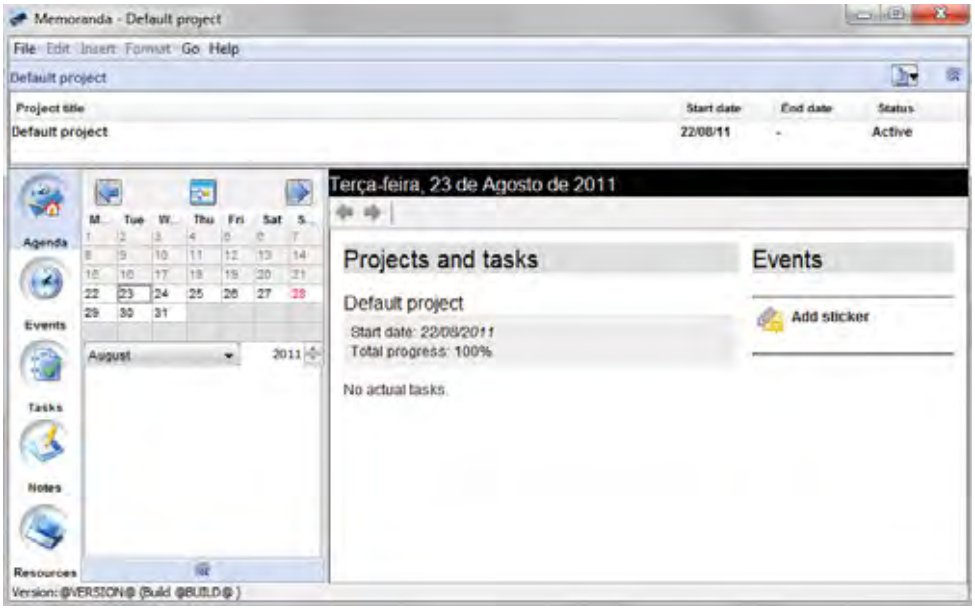


Figura 4.8: Interface principal do artefato de software *Memoranda*.

cionalidades para *AspectJ*) para serem utilizados como artefatos para a avaliação da ferramenta $SoftVis_{OAH}$. Após a fatoração das funcionalidades e desconsiderando os subprojetos não utilizados no experimento, a estrutura do *GanttProject* e do *Memoranda* resultou conforme exposto na Tabela 4.6. Foram inseridos 4 defeitos em cada artefato de acordo com as classificações de Shull et al. (2000) e Alexander et al. (2004)). No artefato *GanttProject*, foram inseridos: 1 defeito para *Fato Incorreto*, 1 defeito para *Falha em estabelecer pós-condições esperadas*, 1 defeito para *Falha em preservar estados invariantes* e 1 defeito para *Alterações incorretas em dependências de controle*. No artefato *Memoranda*, foram inseridos: 1 defeito para *Informação estranha*, 1 defeito para *Força incorreta nos padrões de Pointcuts*, 1 defeito para *Falha em preservar estados invariantes* e 1 defeito para *Fato Incorreto*.

Tabela 4.6: Dimensão dos artefatos de software

Artefato	**Pacotes**	**Classes**	**Aspectos**
GanttProject	45	482	5
Memoranda	24	128	7

4.2.5 Preparação da Análise

O fator deste experimento consiste na análise de um código fonte com dois tratamentos, com e sem o uso da ferramenta $SoftVis_{OAH}$. De acordo com as hipóteses formuladas, a ferramenta $SoftVis_{OAH}$ pode ou não auxiliar as atividades de Compreensão de Programa, pode ou não aumentar a eficácia/eficiência nas atividades de localização de defeitos.

Para verificar a primeira hipótese nula formulada (*Hipótese Nula* (H_0^1)), referente ao auxílio provido ao entendimento de programa (código fonte) com e sem o uso da $SoftVis_{OAH}$, foram utilizadas a relação de *adendos* do programa e a relação de *adendos* descritos corretamente – cada participante teve que descrever corretamente cada *adendo* de cada *aspecto* con-

tidos no código fonte, ou seja, descrever a interferência do *adendo* no comportamento de seus métodos-alvo. A partir da descrição fornecida por cada participante, contendo o nome do pacote, o nome do arquivo, o nome do *aspecto*, o *adendo* junto com seu *pointcut* e a descrição do funcionamento do *adendo*), é realizada uma análise preliminar para validar a descrição, classificando-a como correta ou incorreta. A partir dessa classificação, a porcentagem de descrições corretas fornecida por cada participante é considerada para a verificação (rejeição ou aceitação) da hipótese. Por se tratar de um fator com dois tratamentos envolvendo valores percentuais de acerto, o método estatístico adotado foi o *Teste-T* (Wohlin et al., 2000, Devore, 2006).

Para verificar a segunda hipótese nula formulada (*Hipótese Nula (H_0^2)*), referente à eficácia proporcionada para a localização de defeitos com e sem o uso da $SoftVis_{OAH}$, foram utilizadas a relação de defeitos previamente inseridos no código fonte do artefato de software e a relação de defeitos localizados – cada participante teve que informar, em um formulário próprio, o número/código do erro inserido (fornecido na lista de erros inseridos), a localização do mesmo no código fonte e a hora atual. A partir da descrição da localização do defeito fornecida por cada participante (contendo o nome do pacote, o nome do arquivo, o nome do *aspecto*/*classe*, o *adendo*/*método* e os números inicial e final das linhas de código, é realizada uma análise preliminar para validar a localização do defeito, classificando-a como correta ou incorreta. A partir dessa classificação, a quantidade de localizações corretas fornecida por cada participante é considerada para a verificação (rejeição ou aceitação) da hipótese. Por se tratar de um fator com dois tratamentos envolvendo apenas dois valores fixos possíveis para cada defeito (localizado e não localizado), o método estatístico adotado foi o *Teste-Z* (Devore, 2006).

Para verificar a terceira hipótese nula formulada (*Hipótese Nula (H_0^3)*), referente à eficiência proporcionada para a localização de defeitos com e sem o uso da $SoftVis_{OAH}$, foram utilizadas a relação de defeitos previamente inseridos no código fonte do artefato de software e a relação de defeitos localizados (contendo o horário da localização). A partir das localizações corretas, o tempo médio dispendido em cada localização é considerado para a verificação (rejeição ou aceitação) da hipótese. Por se tratar de um fator com dois tratamentos envolvendo valores de médias de tempo, o método estatístico adotado foi o *Teste-T* (Wohlin et al., 2000, Devore, 2006).

Para auxiliar o processo de análise, foi elaborada uma planilha eletrônica com os objetivos de analisar os dados coletados dos participantes (tanto da caracterização quanto dos *feedbacks*, principalmente os dados resultantes das revisões de programa), efetuar os cálculos estatísticos, auxiliar na redução de conjuntos de dados e validar os questionários e formulários aplicados durante o experimento (de acordo com o processo de análise e interpretação da abordagem *Goal/Question/Metric*).

4.2.6 Avaliação de Validade

Com respeito à *Validade de Conclusão*, todos os indivíduos devem realizar a Revisão de Programa *ad hoc* e a Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$. A verificação de hipóteses deve ser realizada por meio de verificação das questões levantadas.

Com respeito à *Validade Interna*, todos os indivíduos foram submetidos aos mesmos treina-

mentos e atividades, não havendo variação que a comprometa.

Em relação à *Validade de Construção*, conforme discussão apresentada nos diagramas de causa e efeito das hipóteses, os conceitos necessário sobre Programação Orientada a Aspectos (*AspectJ*), *JUnit* e técnicas de Visualização de Software foram providos para não comprometer os resultados das atividades realizadas. Todos os indivíduos receberam treinamento na ferramenta $SoftVis_{OAH}$, além de que a mesma deve ser disponibilizada apenas no momento em que os grupos estiverem realizando a Revisão de Programa com o apoio da ferramenta $SoftVis_{OAH}$. O código fonte utilizado na Revisão de Programa com apoio da ferramenta $SoftVis_{OAH}$ é diferente do código fonte utilizado na Revisão de Programa *ad hoc*, impedindo que uma Revisão de Programa interfira na outra. Todos os indivíduos utilizaram o mesmo ambiente de desenvolvimento (*Eclipse*).

Com respeito à *Validade Externa*, os participantes da experimentação são estudantes da área de Ciência da Computação com foco em desenvolvimento de sistemas para terem as condições mínimas para participarem do experimento. Com isso, espera-se ser possível generalizar os resultados obtidos para estudantes de computação, futuros profissionais que atuarão no mercado.

4.3 Estudo Piloto: Planejamento e Operação

Para validar o experimento e avaliar a instrumentação e os tratamentos, um experimento piloto foi inicialmente conduzido e analisado, não com o objetivo de obter resultados estatísticos, mas para melhorar o projeto experimental e o material utilizado para o experimento final.

4.3.1 Planejamento

O experimento foi conduzido de acordo com a Tabela 4.7, na qual são informados o tempo de duração de cada atividade, uma denominação significativa para a atividade em si, seguida da instrumentação. Por exemplo, a primeira atividade, Introdução, tem duração de cinco minutos e utiliza a instrumentação *MT01* (*Slides* de apresentação do experimento – vide Tabela 4.3).

4.3.2 Operação do Experimento

Para a realização do experimento piloto, inicialmente os indivíduos selecionados (4 alunos voluntários) responderam a um questionário de levantamento de perfil, conforme o Anexo B.2. Durante ambas as revisões de programa, *ad hoc* e com o apoio da ferramenta $SoftVis_{OAH}$, os participantes realizaram o entendimento do programa e a localização de defeitos no código fonte. Na localização de defeitos, os participantes utilizaram o *JUnit* para gerar casos de teste. Durante as tarefas de Revisão de Programa com o apoio da ferramenta $SoftVis_{OAH}$, os casos de teste gerados no *JUnit* foram executados dentro da ferramenta.

4.3.3 Análise e Interpretação: Resultados Parciais

Para a realização da análise e interpretação dos dados coletados da operação do experimento, foram verificados formulários de localização de defeitos e de descrição de *adendos* de *aspectos*. Cada participante tentou informar as localizações dos defeitos inseridos (disponibilizados em uma lista) e tentou descrever cada *adendo* de cada *aspecto* do artefato de software durante as

Tabela 4.7: Projeto do experimento piloto

Dia 1 – Treinamentos		
0:05 h	Introdução – *MT01*	
0:05 h	Termo de consentimento – *MT03* e *F02*	
0:10 h	Caracterização dos participantes – *Q01*	
0:30 h	Treinamento em Programação Orientada a Aspectos – *MT04*	
0:10 h	*Coffee-Break*	
0:20 h	Continuação do treinamento sobre Programação Orientada a Aspectos	
0:40 h	Treinamento em JUnit – *MT05*	
0:15 h	*Feedback* de treinamento – *MT03*, *Q02* e *Q03*	
Dia 2 – Revisão de Programa – Parte 1		
	Grupo A	**Grupo B**
0:05 h	Sorteio dos grupos	
0:10 h	Preparação para a Revisão de Programa	
0:45 h	Revisão *ad hoc* de Programa – *RT01, RT03, F01, FI01, FI02 e FI03*	Revisão *ad hoc* de Programa – *RT02, RT04, F01, FI01, FI02 e FI03*
0:10 h	*Feedback* da Revisão de Programa – *MT03 e Q06*	
0:10 h	*Coffee-Break*	
0:30 h	Treinamento em Visualização de Software e Compreensão de Programa – *MT02*	
0:15 h	*Feedback* de treinamentos e de revisão de revisão de programa – *Q04*	
Dia 3 – Revisão de Programa – Parte 2		
	Grupo A	**Grupo B**
0:40 h	Treinamento da Ferramenta $SoftVis_{OAH}$ – *MT06 e RT05*	
0:10 h	Preparação para a Revisão de Programa	
0:10 h	*Coffee-Break*	
0:45 h	Revisão de Programa suportada pela ferramenta $SoftVisOAH$ – *RT02, RT04, F01, FI01, FI02, FI03 e FI04*	Revisão de Programa suportada pela ferramenta $SoftVisOAH$ – *RT01, RT03, F01, FI01, FI02, FI03 e FI04*
0:15 h	*Feedback* de treinamento e da Revisão de Programa – *MT03, Q05 e Q07*	

revisões de programa, com e sem o apoio da $SoftVis_{OAH}$.

Para exemplificar as atividades realizadas pelos participantes na localização dos defeitos e nas descrições dos *adendos* dos *aspectos*, é ilustrada na Figura 4.9 o código de um *aspecto* com um defeito previamente inserido em seu *adendo* na linha 17. Para informar a localização desse defeito, o participante teve que compreender o programa (com e sem o auxílio da $SoftVis_{OAH}$, dependendo do tipo de Revisão de Programa) e informar sua localização (bem como a hora em que a localização foi realizada) em um formulário, conforme ilustrado na Figura 4.10, em que está destacada a localização do defeito referente à Figura 4.9. Para descrever os *adendos* dos *aspectos*, o participante teve que compreender o programa e informar as descrições em um formulário, conforme ilustrado na Figura 4.11 (a descrição do *adendo* do *aspecto* ilustrado na Figura 4.9 está destacada).

Durante a análise dos dados coletados, cada formulário foi verificado pelo experimentador e a contagem dos defeitos corretamente localizados, do tempo de localização e dos *adendos* corretamente descritos foi realizada e os dados foram classificados conforme informado nas tabelas 4.11 e 4.12.

Devido à baixa amostragem (por se tratar de um experimento piloto), a qual não permite

```
 8  public aspect Annotations {
 9
10      pointcut Notes(EditorPanel obj): execution(* EditorPanel.jbInit()) && target(obj);
11
12      void around(EditorPanel obj): Notes(obj) {
13          if (!Configuration.get("DISABLE_L10N").equals("yes"))
14              net.sf.memoranda.ui.htmleditor.util.Local.setMessages(Local.getMessages());
15          obj.editor = new HTMLEditor();
16          proceed(obj);
17          obj.editor.editor.setEditable(false);
18          obj.initCSS();
19      }
20
21  }
```

Figura 4.9: Código de um *aspecto* com um defeito inserido.

Defeitos Localizados						
Defeito Nº	Hora	Pacote	Arquivo	Classe/Aspe	Método/Adendo	Linhas
4	20:50	memoranda.ui	EventsPanel.java	EventsPanel	saveEvents()	349
	00:10					349
5	21:02	memoranda.htmeditor.ui	Annotations.aj	Annotations	around():Notes()	17
	00:12					17
1	21:15	memoranda	Start.java	Start	main	56

Figura 4.10: Localização de defeitos informada em planilha eletrônica, especificando o número, o pacote, o arquivo, a *classe/aspecto*, o *método/adendo*, a linha inicial e a linha final do defeito.

a realização de análises estatísticas, foram analisados os dados obtidos para validar o projeto do experimento. Quatro participantes foram aleatoriamente divididos em dois grupos. Com relação à formação acadêmica, cada grupo conteve 50% de estudantes de mestrado e 50% de estudantes de graduação em estágio final. Os participantes não tinham experiência em Programação Orientada a Aspectos, em *JUnit* e em atividades de revisão e manutenção de código, além de baixa experiência com o ambiente *Eclipse* e com Visualização de Software (um deles tinha usado o *TreeMap* anteriormente). Os perfis dos participantes por grupo estão sumarizados na Tabela 4.8. Por exemplo, apenas um participante respondeu *sim* à questão *Conhece a técnica de visualização TreeMap*, o que representa 50% dos participantes no *Grupo B*.

A quantidade de localizações de defeitos e o tempo médio despendido estão relatados na Tabela 4.9. Os participantes do *Grupo A* aplicaram a revisão de código *ad hoc*, e três localizações de defeitos foram reportadas despendendo em média 10 minutos cada uma. Pelos dados reportados, os participantes localizaram menos defeitos utilizando a SoftVis$_{OAH}$, mas as localizações de defeitos foram mais rápidas (o tempo médio para localização de defeitos foi menor com o uso da ferramenta), o que permite concluir que todos chegaram mais rapidamente à localização,

Descrição de Aspectos				
Pacote	Arquivo	Aspecto	Adendo	Descrição do Adendo
memoranda.ui	Finalization.aj	Finalization	after(): AppPointcut()	Tenta desabilitar a splashscreen após sua exec
memoranda.ui.htmleditor.util	Annotations.aj	Annotations	void around(EditorPanel obj): Notes(obj)	Desabilita editor.
memoranda.date	Date.aj	Date	before(JNCalendarPanel obj): dayBack(d	Altera o dia em 0. Deveria ser -1

Figura 4.11: Descrição de *adendos* de *aspectos* em planilha eletrônica, especificando o pacote, o arquivo, o *aspecto*, o *adendo* e a descrição.

Tabela 4.8: Perfil de Participantes em Percentagem

Descrição	**Grupo *A***	**Grupo *B***
Desenvolveram sistemas em *Java* para uso próprio	50,0%	50,0%
Desenvolveram sistemas em *Java* para propósitos acadêmicos	100,0%	100,0%
Desenvolveram sistemas em *Java* profissionalmente	0,0%	0,0%
Realizaram atividades de manutenção e revisão de código	0,0%	0,0%
Realizaram atividades de teste de programa	0,0%	50,0%
Conhecem alguma técnica ou ferramenta de Visualização de Informação ou de Software	0,0%	50,0%
Conhecem o teste de Caixa Branca	50,0%	100,0%
Utilizam Programação Orientada a Aspectos	0,0%	50,0%
Utilizam o *JUnit*	0,0%	0,0%
Utilizam o *Eclipse*	100,0%	0,0%
Já utilizaram uma ferramenta de Visualização de Software	0,0%	0,0%
Leem de documentação para entendimento de programa	0,0%	50,0%
Analisam diagramas *UML* para entendimento de programa	0,0%	100,0%
Conhecem a técnica de visualização *TreeMap*	0,0%	50,0%
Conhecem o Grafo de Dependência	50,0%	50,0%
Conhecem o Grafo de Fluxo de Controle	50,0%	50,0%

principalmente quando o tamanho (em número de *classes*) do artefato analisado aumenta.

Tabela 4.9: Localizações de Defeitos

Grupo	**Revisão**	**Defeitos Localizados**	**Percentual**	**Tempo Médio**
A	*Ad Hoc*	3	37,5%	10 min
A	$SoftVis_{OAH}$	1	12,5%	5 min
B	*Ad Hoc*	2	25,0%	8 min
B	$SoftVis_{OAH}$	1	12,5%	6 min

Foram realizados dois *Testes-T bicaudais com variância desigual* – para *localizações de defeitos* e *tempo na localização de defeitos* – com o objetivo de comparar os resultados de ambas as revisões de programas, mas sem ter como objetivo a verificação das hipóteses. Para *localizações de defeitos*, foi obtido um *P-Value* de 62,48% para o *Grupo A* e de 71,77% para o *Grupo B*, representando que os participantes não obtiveram o benefício adequado da ferramenta. Para *tempo na localização de defeitos*, não foi possível obter o *P-Value* devido à pequena quantidade de defeitos localizados utilizando a $SoftVis_{OAH}$ (baixa amostragem).

Foi observado que os participantes levaram mais tempo que o esperado para começarem a utilizar a $SoftVis_{OAH}$, e considerando o tempo limitado para realizar as tarefas de revisões de código – de acordo com o plano do experimento – eles não tiveram o tempo necessário para

completar adequadamente as tarefas.

Além da localização de defeitos, os participantes realizaram atividades de entendimento de programa descrevendo o funcionamento dos *aspectos* e sua influência no comportamento das *classes* envolvidas. Pela análise das descrições, foi possível verificar que os participantes descreveram algumas funcionalidades de alguns *aspectos*, mas não descreveram sua influência no comportamento das *classes*. Os participantes reportaram que a $SoftVis_{OAH}$ proporciona maior facilidade em identificar estruturas de altos níveis. Como mencionado, eles não exploraram adequadamente o *Grafo de Fluxo de Controle*.

Além disso, pela análise dos questionários de *feedback*, foi possível verificar que os resultados relativos à localização de defeitos e ao entendimento de programa, sãos devidos ao pouco tempo despendido em treinamento em *POA*, em *JUnit* e na ferramenta $SoftVis_{OAH}$, bem como à pouca experiência dos participantes com *POA* e nenhuma experiência com *JUnit*, requisitos estes para o uso adequado da ferramenta.

4.3.4 Lições Aprendidas

O experimento envolve múltiplos conceitos e tecnologias, e o domínio desses conceitos e tecnologias é fundamental para explorar adequadamente os recursos da ferramenta. Consequentemente, o treinamento (tempo e materiais utilizados) são importantes para minimizar a ameaça à avaliação, e neste experimento piloto foi possível identificar os seguintes pontos a serem melhorados:

- O tempo de treinamento deve ser aumentado e o material de treinamento deve ser revisto, incluindo alguns exercícios para facilitar o entendimento dos conceitos e tecnologias envolvidos – foi possível observar que os participantes tiveram dificuldades iniciais para entender o significado de mapeamentos visuais (como os conceitos de *POA* são mapeados em atributos visuais);

- Alternativamente, mais participantes com experiência em *POA* e *JUnit* devem ser selecionados, uma vez que estes são requisitos das atividades envolvidas na avaliação da ferramenta;

- O informativo de preenchimento de questionários e formulários (instrumento *MT03*) deve conter mais exemplos de preenchimento, pois algumas dificuldades foram observadas;

- No formulário de localização de defeitos e compreensão de programa (instrumento *F01*), um campo para ordem de localização e para o horário da última localização devem ser incluídos para reduzir o tempo de análise dos formulários;

- A respeito dos materiais de realização de treinamento, para a operação do experimento, todos devem ser previamente encriptados e disponibilizados (i.e. arquivos *Zip* protegidos por senha) para eliminar o tempo despendido na distribuição manual e para não permitir o acesso dos participantes antes do tempo estabelecido;

- O uso de uma ferramenta de apoio à experimentação pode minimizar a ocorrência de ameaças internas.

Em decorrência das lições aprendidas do experimento piloto, as adequações do projeto experimental final foram efetuadas e o experimento controlado foi conduzido utilizando uma ferramenta desenvolvida para a condução de experimentos. Os detalhes do projeto do experimento final estão apresentados nas seções a seguir.

4.4 Estudo Final: Planejamento, Operação e Resultados

De acordo com uma das lições aprendidas do experimento piloto, foi utilizada uma ferramenta de apoio à experimentação – *FAE* (Almodova, 2010), desenvolvida por um grupo de pesquisa desta Universidade – a qual consiste em uma ferramenta *Web* que requer o *Google Chrome* como navegador, o *Apache Tomcat* como servidor de aplicações e o *MySQL* como servidor de banco de dados. Para atender tais requisitos, foi configurado um servidor *Linux CentOS 6* contendo um servidor de aplicação *Tomcat*, um servidor web *Apache*, um Sistema de Gerenciamento de Banco de Dados *MySQL* e um *frontend PhpMyAdmin* para o gerenciamento do banco de dados da ferramenta *FAE*, e o base de dados utilizada pela ferramenta foi configurada para atender às necessidades do experimento em questão.

4.4.1 Planejamento

As atividades propostas no projeto experimental são apresentadas na Tabela 4.10 juntamente com o tempo de duração e a instrumentação utilizada. Ressalta-se que, diante das lições aprendidas com a condução do experimento piloto, as seguintes modificações foram efetuadas. Na primeira atividade do *Dia 1*, houve uma junção das duas atividades iniciais do experimento piloto e foi incluída a atividade *Distribuição de senhas de acesso*. Essa distribuição de senhas introduz a aleatoriedade para a distribuição dos participantes nos grupos, permitindo a remoção da atividade *Sorteio dos grupos* que existia no projeto piloto – primeira atividade do segundo dia. A instrumentação para a atividade *Caracterização dos participantes* mudou de *Q01* para *FI05* devido ao uso da ferramenta *FAE*. Foi incluída a atividade *Treinamento na ferramenta FAE* com respectiva instrumentação. Foram aumentados os tempos de duração das tarefas de treinamento em Programação Orientada a Aspectos e *JUnit*. Para todas as coletas de dados em questionários, foi removido o instrumento *MT03*, pois as instruções de preenchimento foram incorporadas às planilhas eletrônicas de coleta de dados. Para todas as revisões de programas, sua respectiva preparação e os *feedbacks* a respeito dos treinamentos realizados, foi introduzido o instrumento *FI05* (*Google Chrome*) para a utilização da ferramenta *FAE*. Os materiais de treinamento *MT04*, *MT05* e *MT06* foram modificados, contendo mais detalhes e exemplos práticos.

Tabela 4.10: Projeto do experimento final

Dia 1 – Treinamentos		
0:10 h	Introdução, Termo de Consentimento e Distribuição de senhas de acesso – *MT02 e FI02*	
0:10 h	Treinamento na ferramenta *FAE – MT07 e FI05*	
0:05 h	Caracterização dos participantes – *FI05*	
1:20 h	Treinamento em Programação Orientada a Aspectos – *MT04*	
0:15 h	*Coffee-Break*	
1:20 h	Treinamento em *JUnit – MT05*	
0:15 h	*Feedback* de treinamentos – *Q02, Q03 e FI05*	
Dia 2 – Revisão de Programa – Parte 1		
	Grupo A	**Grupo B**
0:05 h	Preparação para a Revisão de Programa	
1:30 h	Revisão *ad hoc* de Programa – *RT01, RT03, F01, FI01, FI02, FI03, FI05*	Revisão *ad hoc* de Programa – *RT02, TR04, F01, FI01, FI02, FI03, FI05*
0:10 h	*Feedback* da Revisão de Programa – *Q06*	
0:15 h	*Coffee-Break*	
1:00 h	Treinamento em Visualização de Software e Compreensão de Programa – *MT02*	
0:15 h	*Feedback* de treinamento – *Q04, FI05*	
Dia 3 – Revisão de Programa – Parte 2		
	Grupo A	**Grupo B**
1:20 h	Treinamento na Ferramenta $SoftVis_{OAH}$ – *MT06 e FI04*	
0:05 h	Preparação para a Revisão de Programa – *FI05*	
0:15 h	*Coffee-Break*	
1:30 h	Revisão de Programa suportada pela ferramenta $SoftVis_{OAH}$ – *RT02, RT04, F01, FI01, FI02, FI03, FI04, FI05*	Revisão de Programa suportada pela ferramenta $SoftVis_{OAH}$ – *RT01, RT03, F01, FI01, FI02, FI03, FI04, FI05*
0:15 h	*Feedback* de treinamento e da Revisão de Programa – *Q05, Q07 e FI05*	

4.4.2 Instrumentação

A instrumentação inicialmente proposta sofreu algumas alterações motivadas pelas lições aprendidas com a condução do experimento piloto. Os formulários e questionários, distribuídos anteriormente em papel, passaram a ser disponibilizados em planilhas eletrônicas e distribuídos por meio da ferramenta *FAE*. Em decorrência disso, o instrumento *MT03* (Informativo de Preenchimento de Questionários e Formulários) foi removido e as instruções de preenchimento passaram a ser embutidas em cada questionário e formulário. Além disso, como a condução do experimento passou a ser realizada com o auxílio de uma ferramenta *FAE*, foi adicionado no grupo *Ferramentas Informatizadas* o instrumento *FI05* – Navegador *Google Chrome*.

4.4.2.1 Artefatos de Software

Os artefatos de software utilizados permaneceram os mesmos: o *GanttProject* e o *Memoranda*. Entretanto, para aumentar o tamanho da amostra a ser utilizada nas análises estatísticas, para o experimento final foram inseridos novos defeitos (ao total de 7 defeitos em cada um), de acordo com as classificações de Shull et al. (2000) e Alexander et al. (2004):

- *GanttProject* – 7 defeitos inseridos no total

  - Fato Incorreto: 2 defeitos.
  - Força incorreta nos padrões de *Pointcuts*: 1 defeito.
  - Falha em estabelecer pós-condições esperadas: 2 defeitos.
  - Falha em preservar estados invariantes: 1 defeito.
  - Alterações incorretas em dependências de controle: 1 defeito.

- *Memoranda* – 7 defeitos inseridos no total
  - Fato Incorreto: 2 defeitos.
  - Informação estranha: 1 defeito.
  - Força incorreta nos padrões de *Pointcuts*: 1 defeito.
  - Falha em estabelecer pós-condições esperadas: 1 defeito.
  - Falha em preservar estados invariantes: 2 defeitos.

Ressalta-se que o domínio de aplicação dos artefatos de software não exerceu influência na análise dos resultados do experimento, pois a lista de defeitos inseridos (em suas respectivas classificações) foi disponibilizada aos participantes e cada participante teve que apenas tentar localizar no código-fonte cada defeito indicado na lista, sem a necessidade de conhecer as classificações dos respectivos defeitos.

4.4.3 Operação do Experimento

Para a realização do experimento, dezessete participantes foram aleatoriamente divididos em dois grupos – os participantes de código ímpar foram alocados para o *Grupo A* e os participantes de código par foram alocados para o *Grupo B*. Cada participante teve seu perfil caracterizado por meio de um questionário de levantamento de perfil – diferentemente dos formulários e questionários distribuídos em planilhas eletrônicas, o questionário de levantamento de perfil foi disponibilizado na própria ferramenta *FAE*.

Os participantes realizaram os mesmos tipos de atividade, revisões de programa *ad hoc* e com o apoio da ferramenta $SoftVis_{OAH}$, exercendo entendimento do programa e a localização de defeitos no código fonte.

4.4.4 Análise e Interpretação: Resultados

Pelos dados reportados na Tabela 4.11, os participantes descreveram corretamente mais *adendos* de *aspectos* utilizando a $SoftVis_{OAH}$. Em relação ao quesito *auxílio à Compreensão de Programa*, referente à *Hipótese Nula* (H_0^1), foi aplicado o *Teste-T Bicaudal com Variação Desigual* Wohlin et al. (2000), Devore (2006), tendo sido obtido um *P-Value* de 1,17%. Considerando o nível padrão de significância de 5%, foi possível rejeitar a *Hipótese Nula* (H_0^1) e, portanto, constata-se que **a utilização da ferramenta $SoftVis_{OAH}$ provê auxílio à Compreensão de Programa**.

Tabela 4.11: Descrições de Adendos

Grupo	Revisão	Percentual
Geral	*Ad Hoc*	13,33%
Geral	$SoftVis_{OAH}$	43,53%
A	*Ad Hoc*	2,08%
A	$SoftVis_{OAH}$	55,00%
B	*Ad Hoc*	23,33%
B	$SoftVis_{OAH}$	33,33%

Pelos dados reportados na Tabela 4.12, os participantes localizaram mais defeitos utilizando a $SoftVis_{OAH}$ e as localizações foram mais rápidas com o uso da ferramenta. Em relação ao quesito *quantidade de defeitos localizados*, referente à *Hipótese Nula (H_0^2)*, foi aplicado o *Teste-Z para Diferença de Proporções* (Devore, 2006) e foi obtido um *P-Value* de 0,76%. Considerando o nível padrão de significância de 5%, foi possível rejeitar a *Hipótese Nula (H_0^2)*, constatando que **a utilização da ferramenta $SoftVis_{OAH}$ proporciona maior eficácia que a localização de defeitos *ad hoc***.

Tabela 4.12: Localizações de Defeitos

Grupo	Revisão	Defeitos Localizados	Percentual	Tempo Médio/Defeito
Geral	*Ad Hoc*	26	21,85%	0:23:17h
Geral	$SoftVis_{OAH}$	43	36,13%	0:12:25h
A	*Ad Hoc*	12	21,43%	0:21:18h
A	$SoftVis_{OAH}$	25	44,64%	0:09:23h
B	*Ad Hoc*	14	22,22%	0:24:37h
B	$SoftVis_{OAH}$	18	28,57%	0:18:30h

Em relação ao quesito *tempo de localização de defeitos*, referente à *Hipótese Nula (H_0^3)*, foi aplicado o *Teste-T Bicaudal com Variação Desigual* Wohlin et al. (2000), Devore (2006) e foi obtido um *P-Value* de 1,04%, sendo possível rejeitar a *Hipótese Nula (H_0^3)*, constatando que **a utilização da ferramenta $SoftVis_{OAH}$ proporciona maior eficiência que a localização de defeitos *ad hoc***.

Durante a avaliação da *Hipótese Nula (H_0^2)*, foram observados diversos apontamentos de defeitos em trechos errados do código (a localização reportada difere da localização do defeito). Por ter sido observado um número expressivo de localizações erradas, motivou a formulação de uma hipótese não prevista no projeto experimental. Essa hipótese relaciona o uso da ferramenta como um fator que minimiza a localização errada, ou seja: *Hipótese Nula (H_0^4)* – o uso da ferramenta $SoftVis_{OAH}$ não ajuda a diminuir erros na localização de defeitos; A *Hipótese Alternativa (H_1^4)* é que o uso da ferramenta ajuda a diminuir erros na localização de defeitos. De acordo com os dados reportados na Tabela 4.13, os participantes localizaram incorretamente menos

defeitos utilizando a $SoftVis_{OAH}$ e as localizações incorretas também foram mais rápidas com o uso da ferramenta. Em relação a esse novo quesito – *quantidade de defeitos localizados incorretamente* – de maneira análoga ao quesito *quantidade de defeitos localizados*, foi aplicado o *Teste-Z para Diferença de Proporções* Devore (2006) e foi obtido um *P-Value* de 0,87%, constatando que **a utilização da $SoftVis_{OAH}$ ajuda a diminuir a localização incorreta**.

Tabela 4.13: Localizações incorretas de defeitos

Grupo	Revisão	Defeitos Localizados	Perc.	Tempo Médio/Defeito
Geral	*Ad Hoc*	24	20,17%	0:17:45h
Geral	$SoftVis_{OAH}$	11	9,24%	0:12:49h
A	*Ad Hoc*	11	19,64%	0:19:00h
A	$SoftVis_{OAH}$	7	12,50%	0:12:51h
B	*Ad Hoc*	13	20,63%	0:16:42h
B	$SoftVis_{OAH}$	4	6,35%	0:12:45h

Tabela 4.14: Resultados das Análises Estatísticas

Quesito	Teste utilizado	P-Value
Auxílio à Compreensão de Programa	*Teste-T Bicaudal com Variação Desigual*	1,17%
Quantidade de Defeitos Localizados	*Teste-Z para Diferença de Proporções*	0,76%
Tempo de Localização de Defeitos	*Teste-T Bicaudal com Variação Desigual*	1,04%
Quantidade de defeitos localizados incorretamente	*Teste-Z para Diferença de Proporções*	0,87%

Na Tabela 4.14 estão reportados os resultados mencionados das análises estatísticas. Durante a condução do experimento final, foram feitas as seguintes observações:

- Alguns participantes dispenderam muito tempo executando os artefatos de software, influenciando na quantidade e tempo de localização de defeitos em ambas as localizações de defeitos;
- Alguns participantes tentavam por muito tempo buscar defeitos apenas interagindo com as visualizações da ferramenta, sem analisar o código-fonte;
- Poucos participantes não sabiam como programar em *Java*, e esse fato poderia ter influenciado no resultado final das localizações de defeitos;
- No início da utilização da $SoftVis_{OAH}$, alguns participantes reportaram desorientação nas visualizações, principalmente devido ao tamanho dos artefatos de software (número de classes e métodos);

- Alguns participantes dispenderam a maior parte do tempo na localização de defeitos, outros na descrição dos *adendos* dos *aspectos*, influenciando no resultado individual de localizações de defeitos e compreensão de programas.

Com relação à formação acadêmica, o experimento conteve 23,53% de estudantes de mestrado e 76,47% de estudantes de graduação em estágio final, cujos perfis estão sumarizados em médias e porcentagens nas tabelas A.1 e A.2, respectivamente, no Apêndice A.

4.5 Considerações Finais

Pelas análises dos resultados do experimento, foi possível rejeitar as três hipóteses nulas definidas, constatando estatisticamente que a utilização da ferramenta de Visualização de Software $SoftVis_{OAH}$ provê auxílio à Compreensão de Programa, proporciona maior eficiência que a localização de defeitos *ad hoc* e, também, proporciona maior eficácia. Um novo fato foi constatado: a utilização da ferramenta ajuda a diminuir a localização incorreta de defeitos.

O experimento conduzido consiste em um experimento *Tipo I* (Wohlin et al., 2000), pois possui um fator (avaliação de atividades de Revisão de Programa) com dois tratamentos (com e sem o auxílio da ferramenta $SoftVis_{OAH}$). Durante a fase de análise de dados do experimento não foi constatada a ocorrência de erros. Quanto ao erro *Tipo I* (que refere-se à rejeição equivocada da hipótese nula), apesar da baixa amostragem, os resultados obtidos ($P - value$) foram consideralvelmente mais baixos que o nível de significância adotado ($\alpha <= 5\%$). E com a rejeição das hipóteses nulas, os testes não estão sucetíveis ao erro *Tipo II*.

Ressalta-se que o projeto experimental foi planejado e preparado para minimizar as influências que poderiam interferir na avaliação das hipóteses. De acordo com os diagramas de causa e efeito apresentados para cada hipótese, o nível de conhecimento em Programação Orientada a Aspectos, em JUnit e em técnicas de Visualização de Software poderia influenciar o resultado da avaliação das hipóteses. Assim, os materiais de treinamentos utilizados foram preparados com os conceitos necessários para a realização das atividades. Da mesma maneira, o tempo para as atividades foi ajustado para não comprometer os resultados.

A quantidade de participantes do experimento final (17 ao todo) foi o bastante para a avaliação das hipóteses, mas ainda não foi o bastante para se avaliar possíveis relações e influências. Por exemplo, foram coletados dados referentes ao perfil dos participantes, mas a homogeneidade de perfis não possibilitou avaliar a influência de algum fato nos resultados obtidos – a experiência dos participantes com *JUnit*, com *Programação Orientada a Aspectos* e/ou com *Visualização de Software* interfere nos resultados? Com a realização de novos experimentos (replicações) com número maior de participantes espera-se ter a possibilidade de analisar outras questões que possam influenciar os resultados.

CAPÍTULO

5

Conclusões

A Visualização de Software consiste em uma importante abordagem para Compreensão de Programa, principalmente por que os artefatos de software são abstratos. Tendo como foco o código fonte, a obtenção de conhecimento é difícil, especialmente pela quantidade de linhas de código em sistemas de software e sua organização. Em se tratando de programas orientados a aspectos, um novo desafio se apresenta, uma vez que as novas características da *POA* devem ser externializadas em projeções visuais significativas.

Uma ferramenta de Visualização de Programa Orientado a Aspecto deve expor a organização do código entrelaçado e espalhado obtido pelo *weaver*. Foi proposto um mapeamento visual para apresentar características de Programas Orientados a Aspecto, e a ferramenta $SoftVis_{OAH}$ foi desenvolvida abrangendo o mapeamento e a análise do *bytecode* gerado a partir do processo de *weaver*. Essas visualizações visam a externalizar organização estrutural, as relações entre classes e *aspectos*, e o código "adendado". Pelo mapeamento proposto é possível destacar elementos selecionados em diferentes níveis de detalhe, permitindo ao usuário reunir informações a respeito de *aspectos* e seu entrelaçamento e espalhamento pelo código fonte. Adicionalmente, os casos de teste ao longo das representações visuais permitem ao usuário perceber como os casos de teste percorrem o código fonte. A proposta implementada na ferramenta $SoftVis_{OAH}$ parecia promissora, de acordo com os estudos iniciais. Entretanto, não havia evidência significativa obtida com o rigor necessário para permitir conclusões efetivas sobre o uso da ferramenta.

Para avaliar a eficácia e a eficiência da ferramenta de uma maneira mais realista, foi elaborado um estudo experimental controlado, apresentado nesta dissertação. O estudo foi conduzido utilizando o processo de experimentação proposto por Wohlin et al. (2000), aplicado a artefatos de Programa Orientado a Aspectos com defeitos previamente inseridos em um código fonte. Os dados provenientes da condução do experimento foram coletados, e, com base na análise dos resultados, foi possível concluir estatisticamente que a ferramenta de Visualização de Software $SoftVis_{OAH}$ auxilia as atividades de Compreensão de Programa e proporciona eficácia e eficiência às atividades de localização de defeitos.

5.1 Contribuições, Limitações e Dificuldades

As contribuições do trabalho desenvolvido podem ser apresentadas em duas categorias: rejeição das hipóteses nulas apresentadas no projeto do experimento; e lições aprendidas. Com relação às hipóteses, vale ressaltar que o projeto experimental foi planejado e preparado para minimizar as influências que poderiam interferir na avaliação das hipóteses, mas isso não caracteriza uma situação de *busca de resultado* (*fishing results*), pois tem o objetivo de minimizar ameaças que comprometam a validade interna. Por exemplo, de acordo com os diagramas de causa e efeito apresentados para cada hipótese, o nível de conhecimento em Programação Orientada a Aspectos, em *JUnit* e em técnicas de Visualização de Software poderia influenciar o resultado da avaliação das hipóteses. Assim, os materiais de treinamentos utilizados foram preparados com os conceitos necessários para a realização das atividades. Da mesma maneira, o tempo foi ajustado para que os participantes pudessem realizar as atividades sem comprometer os resultados.

Algumas lições aprendidas são importantes para outros pesquisadores interessados em conduzir experimentos controlados para avaliar ferramentas em geral e, especialmente, as ferramentas de visualização de software. Dentre elas, ressalta-se:

- O tempo e o material utilizados em treinamento na ferramenta é fundamental e deve ser avaliado em um experimento piloto. Adicionalmente, recomenda-se que sejam propostos exercícios durante o treinamento, planejados para explorar todos os recursos disponíveis (principalmente recursos de interação).

- A efetividade das metáforas visuais em expor características do software, em geral são propostas para serem intuitivas. Entretanto, a interpretação das metáforas é subjetiva e, consequentemente, um estudo (experimento controlado) para avaliar a ferramenta deve tratar de minimizar a subjetividade na interpretação. E isso deve ser tratado no material de treinamento.

- O tempo para a realização das atividades utilizando a ferramenta deve ser balanceado: tempo em demasia pode fazer com que o participante perca o foco e a motivação; por outro lado, pouco tempo pode não permitir que o participante explore os recursos oferecidos pela ferramenta. Recomenda-se que, para o balanceamento do tempo a ser despendido no uso da ferramenta, sejam considerados fatores como a experiência dos participantes relacionada aos recursos e tecnologias empregados, o tamanho do artefato (codigo fonte) utilizado, dentre outros.

Uma dificuldade encontrada foi a escolha e a preparação dos códigos-fonte utilizados no experimento: deveria ser escolhido um artefato em tamanho necessário para não permitir que um participante o compreendesse com facilidade (e dessa maneira não evidenciaria o auxílio obtido com o uso da $SoftVis_{OAH}$), mas também para não impossibilitar sua análise em um tempo compatível. Por não ter sido encontrada uma aplicação adequada (considerando o tamanho) desenvolvida com o uso de *POA* e *AspectJ*, foi preciso refatorar algumas funcionalidades

dos códigos-fonte escolhidos, originalmente desenvolvidos com o paradigma de Programação Orientada a Objetos. Posteriormente, os defeitos foram inseridos. Além disso, a escolha dos códigos-fonte utilizados sofreu uma restrição tecnológica devido à descontinuidade do *AspectJ* para o ambiente *NetBeans*.

Uma limitação observada refere-se à baixa amostragem e ao contexto de execução do experimento. A generalização do resultado pode ser comprometida, pois com a participação de estudantes não foi possível avaliar o uso da ferramenta por pessoas com experiência (profissionais de mercado). Entretanto, assim como a experiência pode influenciar positivamente nas atividades executadas sem o uso da $SoftVis_{OAH}$ – melhorando quantidades de defeitos localizados e o tempo, por exemplo –, também pode se beneficiar da utilização da ferramenta. Tal fato foi observado no experimento – alunos com mais experiência em programação localizaram corretamente mais defeitos, com e sem o uso da $SoftVis_{OAH}$ – embora não tenha siginificado estatístico.

Ressalta-se que um artigo para periódico está sendo redigido e deve ser submetido em breve.

5.2 Trabalhos Futuros

Como trabalhos futuros pode-se apontar três categorias de possibilidades. A primeira delas é a replicação do experimento apresentado nesta dissertação em outros contextos – na verdade em outras instituições de ensino superior, onde pesquisadores manifestaram interesse e se dispuseram a colaborar. Com tais replicações, seria possível comparar os resultados das replicações com os resultados ora apresentados por meio da condução de uma meta-análise. Além disso, com a condução das replicações, a amostra de dados é acumulada, permitindo a realização de outras análises ainda não efetuadas, por exemplo avaliar a influência do perfil dos participantes no resultado do experimento.

A segunda categoria refere-se à replicação do experimento com alguma variação. Tomando-se o projeto experimental apresentado como base, pode-se variar alguns instrumentos, como os artefatos utilizados, para um novo experimento.

A terceira categoria de possibilidade refere-se à avaliação de novas versões da ferramenta $SoftVis_{OAH}$. Com a implementação de novas funcionalidades – por exemplo, disponibilização de filtros de pacotes/classes/*aspectos*, destaque de classes/*aspectos* contendo o método *main()*, e novas técnicas de visualização – o experimento apresentado será usado como base para a comparação das representações visuais. Para tanto, serão necessárias adaptações no projeto experimental e no material utilizado para a realização de um novo experimento.

Referências Bibliográficas

ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. *Towards the systematic testing of aspect-oriented programs*. Relatório Técnico CD-4-105, Departament of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

ALMODOVA, J. M. A. Extensão de uma ferramenta de apoio à experimentação aplicada à engenharia de software, monografia de Conclusão de Curso, Universidade Estadual Paulista, 2010.

ARAÚJO, J.; BANIASSAD, E.; CLEMENTS, P. *Early aspects: The current landscape*. Relatório Técnico, Carnegie Mellon University, 2005.

BALDWIN, J.; MYERS, D.; STOREY, M.; COADY, Y. Assembly code visualization and analysis: An old dog can learn new tricks! In: *PLATEAU '09 Workshop at Onward!*, 2009.

BALL, T.; EICK, S. G. Software visualizatio in the large. *IEEE Computer*, v. 29, n. 4, p. 33–43, 1996.

BASILI, V. R.; CALDIERA, G.; LANUBILE, F.; ; SHULL, F. Investigating focused techniques for understanding frameworks. *Proc. of the 1st lnternational Workshop on Empirical Studies of Software Maintenance (WESS 96)*, p. 49–53, 1996a.

BASILI, V. R.; CALDIERA, G.; LANUBILE, F.; ; SHULL, F. Studies on reading techniques. *Proc. of the Twenty-First Annual Software Engineering Workshop*, v. SEL-96-002, p. 59–65, 1996b.

BASILI, V. R.; CALDIERA, G.; ROMBACH, H. D. *Goal/question/metric paradigm*. Encyclopedia of Software Engineering, 1994.

BASILI, V. R.; GREEN, S.; LAITENBERGER, O.; LANUBILE, F.; SHULL, F.; SØRUMGÅRD, S.; ZELKOWITZ, M. V. The empirical investigation of perspective-based reading. *Empirical Software Engineering: An International Journal*, v. 1, n. 2, p. 133–164, 1996c.

BASILI, V. R.; GREEN, S.; LAITENBERGER, O.; LANUBILE, F.; SHULL, F.; SØRUMGÅRD, S.; ZELKOWITZ, M. V. Lab package for the empirical investigation of perspective-based reading. Avaliable in: http://www.cs.umd.edu/projects/SoftEng/ESEG/manual/pbr_package/ manual.html, 1998.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, v. 12, n. 7, p. 733–743, 1986.

BASILI, V. R.; SHULL, F.; LANUBILE, F. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, v. 25, n. 4, 1999.

BEDERSON, B. B.; SHNEIDERMAN, B.; WATTENBERG, M. Ordered and quantum treemaps: Making effective use of 2d space to display hierarchies. *ACM Transactions on Graphics (TOG)*, v. 4, n. 21, p. 833–854, 2002.

CARD, S. K.; MACKINLAY, J. D.; SHNEIDERMAN, B. *Readings in information visualization – using vision to think*. MK Publishers Inc, 1999.

CARNEIRO, G.; MAGNAVITA, R.; MENDONÇA, M. Combining software visualization paradigms to support software comprehension activities. In: *4th ACM symposium on Software visualization (SoftVis'08)*, NY, USA, 2008, p. 201–202.

CARVER, J. C. Towards reporting guidelines for experimental replications: A proposal. In: *International Workshop on Replication in Empirical Software Engineering Research (RESER)*, 2010.

D'ARCE, A. F.; GARCIA, R. E.; CORREIA, R. C. M. Coordinated visualization of aspect oriented programs. *I Workshop Brasileiro de Visualização de Software (WBVS 2011). Anais do I Workshop Brasileiro de Visualização de Software*, p. 25–32, publicado, 2011.

D'ARCE, A. F.; GARCIA, R. E.; CORREIA, R. C. M.; ELER, D. M. Coordination model to support visualization of aspect-oriented programs. In: *Proceedings of the Twenty-Fourth International Conference on Software Engineering & Knowledge Engineering (SEKE 2012)*, San Francisco Bay, USA, 2012, p. 168–173.

DEVORE, J. L. *Probabilidade e estatística para engenharia e ciências*. São Paulo: Pioneira Thomson Learning, 2006.

DIJKSTRA, E. *A discipline of programming*. Englewood Cliffs, NJ, USA: Prentice Hall, 1976.

DUTRA, L. S. Visualização hierárquica de programas orientados a aspectos. Faculdade de Ciências e Tecnologia – Universidade Estadual Paulista "Júlio de Mesquita Filho", technical Report, 2010.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming – introduction. *Commun. ACM*, v. 10, n. 44, p. 29–32, 2001.

FERRARI, F. C.; BURROWS, R.; LEMOS, O. A. L.; GARCIA, A.; MALDONADO, J. C. Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In: *Simpósio Brasileiro de Engenharia de Software (SBES 2010)*, Salvador, aceito para publicação, 2010.

FILMAN, R. E.; ELRAD, T.; CLARKE, S.; AKSIT, M. *Aspect-oriented software development*. Addison Wesley, 2005.

FUSARO, P.; LANUBILE, F.; VISAGGIO, G. A replicated experiment to assess requirements inspection techniques. *Empirical Software Engineering*, v. 2, n. 1, p. 39–57, 1997.

GALL, H.; LANZA, M. Software analysis visualization. In: *28th International Conference on Software Engineering (ICSE Shangai 2006)*, 2006.

GARCIA, R. E.; HOHN, E. N.; BARBOSA, E. F.; MALDONADO, J. C. An ontology for controlled experiments on software engineering. In: *SEKE (Software Engineering & Knowledge Engineering)*, Knowledge Systems Institute Graduate School, 2008, p. 685–690.

HOHN, E. N. *Técnicas de leitura de especificação de requisitos de software: estudos empíricos e gerência de conhecimento em ambientes acadêmico e industrial*. Dissertação de Mestrado, Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo, (in Portuguese), 2003.

JOHNSON, B.; SHNEIDERMAN, B. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In: *Proceedings of the 2nd conference on Visualization '91*, IEEE Computer Society Press, 1991, p. 284–291.

KARAHASANOVIC, A.; LEVINE, A. K.; THOMAS, R. Comprehension strategies and diffculties in maintaining object-oriented systems: An explorative study. *Journal of System and Software*, v. 7, n. 80, p. 1541–1559, 2007.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of aspectj. *ECOOP 2001 – Object Oriented Programming*, v. 2072, n. 2001, p. 327–354, 2001.

KITCHENHAM, B. A.; PFLEEGER, S. L.; HOAGLIN, D. C.; EMAM, K. E.; ROSENBERG, J. Preliminary guidelines for empirical research in software engineering. *IEEE Transaction on software engineering*, v. 28, n. 2, 2002.

KNIGHT, C. System and software visualization. *World Scientifc Publishing Company*, v. 0, n. 0, 2000.

LANUBILE, F.; VISAGGIO, G. Evaluating empirical models for the detection of high-risk components: Some lessons learned. 20th Annual Software Engineering Workshop, 1995.

LANZA, M. Codecrawler - polymetric views in action. In: *19th International Conference on Automated Software Engineerings*, 2004, p. 394–395.

LEMOS, O. A. L.; FRANCHIN, I. G.; MASIEIRO, P. C. Integration testing of object-oriented and aspect-oriented programs: a structural pairwise approach for java. *Science of Computer Programming*, v. 74, n. 10, p. 861–878, 2009.

LOPES, V. P.; TRAVASSOS, G. H. Experimentação em engenharia de software: Glossário de termos. In: *Experimental Software Engineering Latin American Workshop (ESELAW'09)*, 2009.

LUCCA, G. A. D.; PENTA, M. D. Experimental settings in program comprehension: Challenges and open issues. In: *14th International Conference on Program Comprehension (ICPC'06)*, 2006, p. 229–234.

LUCIA, A. D.; GRAVINO, C.; OLIVETO, R.; TORTORA, G. An experimental comparison of ER and UML class diagrams for data modelling. *Empirical Software Engineering*, v. 15, n. 5, p. 455–492, 2010.

MALDONADO, J. C.; CARVER, J.; SHULL, F.; FABBRI, S.; DÓRIA, E.; MARTIMIANO, L.; MENDONÇA, M.; BASILI, V. R. Perspective-Based Reading: a Replicated Experiment Focused on Individual Reviewer Effectiveness. *Empirical Software Engineering: An International Journal - (EMSE)*, v. 11, n. 1, 2006.

MALETIC, J. I.; COLLARD, A. M. M. A task oriented view of software visualization. In: *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT.02)*, 2002.

MARTINS, R. M. Teste estrutural de programas orientados a aspectos. Faculdade de Ciências e Tecnologia – Universidade Estadual Paulista "Júlio de Mesquita Filho", technical Report, 2007.

MENDONÇA, M. G.; MALDONADO, J. C.; DE OLIVEIRA, M. C. F.; CARVER, J.; FABRI, S. C. P. F.; SHULL, F.; TRAVASSOS, G. H.; HÖHN, E. N.; BASILI, V. R. A framework for software engineering experimental replications. *13th IEEE International Conference on Engineering of Complex Computer Systems*, 2008.

MUNZNER, T. Exploring large graphs in 3d hyperbolic space. *IEEE Computer Graphics and Applications*, v. 18, n. 4, p. 18–23, 1998.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large scale software development: An observational case study. In: *International Conference on Software Engineering*, 1998, p. 251–260.

PFEIFFER, J.-H.; GURD, J. R. Visualisation-based tool support for the development of aspect-oriented programs. In: *Proceedings of the 5th international conference on Aspect-oriented software development*, New York, NY, USA: ACM, 2006, p. 146–157. Disponível em http://doi.acm.org/10.1145/1119655.1119676 (Acessado em Julho/2011)

PRECHELT, L.; UNGER, B. A series of controlled experiments on design patterns: Methodology and results. In: *Proc. Softwaretechnik'98, GI Conference, Paderborn*, 1998, p. 53–60.

PRICE, B.; BAECKER, R.; SMALL, I. An introduction to software visualization. software visualization – programming as a multimedia experience. *MIT Press*, p. 3–27, 1998.

RAMOS, R. A.; DE CASTRO, J. B.; ARAÚJO, J.; MOREIRA, A.; ALENCAR, F. M. R.; PENTEADO, R. D. Early aspects refactoring. In: *CIbSE'08*, 2008, p. 238–252.

ROBSON, C. *A resource for social scientists and practitioners-researchers*. Blackwell, 1993.

SHULL, F.; RUS, I.; BASILI, V. How perspective-based reading can improve requirements inspections. *IEEE Software*, p. 73–79, 2000.

SILVA, C. T. *Um experimento na engenharia de requisitos para caracterização do processo e sua adequação às práticas específicas do cmmi*. Dissertação de Mestrado, C.E.S.A.R. – Centro de Estudos e Sistemas Avançados do Recife, (in Portuguese), 2010.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. *Introdução à engenharia de software experimental*. Relatório Técnico, Programa de Engenharia de Sistemas e Computação (COPPE-UFRJ), Rio de Janeiro-RJ, rT-ES-590/02, 2002.

TREVISAN, M. Visualização de software orientado a aspectos usando visualização hiperbólica. Faculdade de Ciências e Tecnologia – Universidade Estadual Paulista "Júlio de Mesquita Filho", technical Report, 2010.

WARE, C. *Information visualization perception for design*. San Francisco, CA, USA: Morgan Kaufmann Publisher, 2004.

WETTEL, R.; LANZA, M. Visualizing software systems as cities. *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, p. 92–99, 2007.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. *Experimentation in software engineering: An introduction*. Boston (USA): Kluwer Academic Publishers, 2000.

WOOD, M.; ROPER, M.; BROOKS, A.; MILLER, J. Comparing and combining software defect detection techniques: A replicated empirical study. *6th European Software*

Engineering Conference/5th ACM SIESOFT Symposium on the Foundations of Software Engineering, LNCS, v. 1301, p. 262–277, 1997.

WURTHINGER, T.; WIMMER, C.; MOSSENBOCK, H. Visualization of program dependence graphs. In: *International Conference on Compiler Construction, LNCS 4959*, 2008, p. 193–196.

ZELKOWITZ, M. V.; WALLACE, D. Experimental validation in software engineering. *Information and Software Technology*, v. 39, n. 11, p. 735–743, evaluation and Assessment in Software Engineering, 1997.

ZELKOWITZ, M. V.; WALLACE, D. R. Experimental models for validating technology. *Computer*, v. 31, n. 5, p. 23–31, iEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

APÊNDICE

A

Perfil dos Participantes

A.1 Perfil de Participantes em Média

Tabela A.1: Perfil de Participantes em Média

Descrição	**Geral**	**Grupo *A***	**Grupo *B***
Anos cursando formação acadêmica	4,06	3,38	4,67
Anos de conclusão de formação acadêmica	0,00	0,00	0,00
Anos de experiência com desenvolvimento de sistemas Java	1,24	1,00	1,44
Anos de experiência em atividades de manutenção e revisão de código	1,82	3,63	0,22
Anos de experiência em em atividades de teste de programa	1,47	2,75	0,33
Anos de experiência em atividades de compreensão de programa	2,06	3,88	0,44
Anos de experiência com Programação Orientada a Aspectos	0,06	0,00	0,11
Anos de experiência com o *JUnit*	0,06	0,13	0,00
Anos de experiência com o *Eclipse*	0,65	0,63	0,67
Anos de experiência com uma ferramenta de Visualização de Software	0,00	0,00	0,00

A.2 Perfil de Participantes em Percentagens

Tabela A.2: Perfil de Participantes em Percentagem

Descrição	**Geral**	**Grupo *A***	**Grupo *B***
Desenvolveram sistemas em *Java* para uso próprio	23,53%	25,00%	22,22%
Desenvolveram sistemas em *Java* para propósitos acadêmicos	76,47%	62,5,0%	88,89%
Desenvolveram sistemas em *Java* profissionalmente	5,88%	0,00%	11,11%
Realizaram atividades de manutenção e revisão de código	35,29%	50,00%	22,22%
Realizaram atividades de teste de programa	29,41%	37,50%	22,22%
Conhecem alguma técnica ou ferramenta de Visualização de Informação ou de Software	11,76%	25,00%	0,00%
Utilizam Programação Orientada a Aspectos	5,88%	0,00%	11,11%
Conhecem o teste de Caixa Branca	70,59%	62,50%	77,78%
Utilizam o *JUnit*	5,88%	12,5%	0,00%
Utilizam o *Eclipse*	52,94%	62,50%	44,44%
Para entender um programa, utilizam leitura do código fonte	100,00%	100,00%	100,00%
Para entender um programa, utilizam leitura da documentação	70,59%	62,50%	77,78%
Para entender um programa, utilizam análise de diagramas *UML*	64,71%	62,50%	66,67%
Para entender um programa, utilizam outros meios	58,82%	75,00%	44,44%
Já utilizaram uma ferramenta de Visualização de Software	5,88%	12,50%	0,00%
Conhecem a técnica de visualização *TreeMap*	11,76%	12,50%	11,11%
Conhecem o Grafo de Dependência	29,41%	50,00%	11,11%
Conhecem o Grafo de Fluxo de Controle	52,94%	50,00%	55,56%

APÊNDICE

B

Documentos

B.1 Termo de Consentimento

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO

Pelo presente instrumento, declaro que _____ _____, portador da cédula de identidade RG nº _____ e do CPF nº _____, concorda em participar da pesquisa científica de nível acadêmico de mestrado intitulada **Avaliação de Ferramenta de Visualização de Software por meio de Experimento Controlado**, realizando atividades de **Compreensão de Programa** e estando ciente dos seguintes termos:

A participação na pesquisa terá duração de aproximadamente 3 (três), dias não necessariamente consecutivos, com as datas a serem determinadas, neste ano de 2011, tendo como local a **Faculdade de Ciências e Tecnologia** da **Universidade Estadual Paulista Júlio de Mesquita Filho**, **UNESP**, situada à Rua Roberto Simonsen, 305, CEP 19060-90, nesta cidade de Presidente Prudente/SP. As informações do participante serão tratadas de maneira anônima e confidencial, ou seja, em nenhum momento será divulgado o nome do participante em qualquer fase do estudo. Os dados coletados poderão ser utilizados em pesquisas futuras, permanecendo o anonimato do participante. A participação é voluntária e sem remuneração, podendo o participante desistir a qualquer momento de participar da pesquisa. A participação consistirá em responder questionários e realizar atividades de compreensão de código de programa de computador, não havendo riscos de qualquer natureza. O benefício relacionado à participação consiste em aumentar o conhecimento científico para a área de Ciência da Computação.

Declaro estar ciente do inteiro teor deste termo de consentimento e estar de acordo em participar da pesquisa proposta.

Presidente Prudente, ____ de _____ de 2012.

Assinatura do participante

Assinatura do orientando Assinatura do orientador

ID do participante: _____

B.2 Questionário de Perfil de Participante

ID.

1. Formação acadêmica:
   a: Graduação
   b: Especialização
   c: Mestrado
   d: Doutorado
   e: Outro (especificar):

2. Em relação a sua formação acadêmica, você:
   a: Está cursando
   b: Já concluiu

3. Tempo de formação acadêmica (em anos):

4. Em relação a sua experiência com desenvolvimento de sistemas em Java (marque "S" ou "N"):
   ( ) Nunca desenvolvi sistemas em Java
   ( ) Tenho desenvolvido sistemas em Java para uso próprio
   ( ) Tenho desenvolvido sistemas em Java na área acadêmica
   ( ) Tenho desenvolvido sistemas em Java profissionalmente

5. Tempo de experiência com desenvolvimento de sistemas Java (em anos):

6. Você já realizou atividades de manutenção e revisão de código?
   a: Sim
   b: Não

7. Caso a resposta seja afirmativa, indique o tempo de experiência em atividades de manutenção e revisão de código (em anos):

8. Você já realizou atividades de teste de programa?
   a: Sim
   b: Não

9. Caso a resposta seja afirmativa, indique o tempo de experiência em atividades de teste de programa (em anos):

10. Você já realizou atividades de Compreensão de Programa?
    a: Sim
    b: Não

11. Caso a resposta seja afirmativa, indique o tempo de experiência em atividades de Compreensão de Programa (em anos):

12. Você conhece alguma técnica ou ferramenta de Visualização de Informação ou de Visualização de Software?
    a: Sim
    b: Não

13. Você utiliza Programação Orientada a Aspectos?
    a: Sim
    b: Não

14. Caso a resposta seja afirmativa, indique o tempo de experiência com Programação Orientada a Aspecto (em anos):

15. Você conhece o Teste de Caixa Branca?
    a: Sim
    b: Não

16. Você utiliza o JUnit?
    a: Sim
    b: Não

17. Caso a resposta seja afirmativa, indique o tempo de experiência com o JUnit (em anos):

18. Você utiliza o Eclipse?
    a: Sim
    b: Não

19. Caso a resposta seja afirmativa, indique o tempo de experiência com o Eclipse (em anos):

20. Para compreender um programa, o que você utiliza (marque "S" ou "N")?
    ( ) Leitura do código fonte
    ( ) Leitura da documentação
    ( ) Análise de diagramas UML
    ( ) Outros:

21. Você já utilizou alguma ferramenta de Visualização de Software?
    a: Não
    b: Sim, poucas vezes
    c: Sim, consideráveis vezes
    d: Sim, muitas vezes

22. Caso a resposta seja afirmativa, indique o tempo de experiência com a utilização da ferramenta de Visualização de Software (em anos):

23. Caso a resposta do item 21 seja afirmativa, quais foram os propósitos de você ter utilizado a ferramenta de Visualização de Software (marque "S" ou "N")?
    ( ) Curiosidade
    ( ) Compreensão de Programa
    ( ) Teste de programa
    ( ) Outros:

24. Caso a resposta do item 21 seja afirmativa, a utilização da ferramenta de Visualização de Software o ajudou?
    a: Não
    b: Sim, pouco

c: Sim, consideravelmente

d: Sim, muito

25. Você conhece a técnica de visualização Treemap?

    a: Sim

    b: Não

26. Você conhece o Grafo de Dependência?

    a: Sim

    b: Não

27. Você conhece o Grafo de Fluxo de Controle?

    a: Sim

    b: Não

B.3 Questionário de Feedback de Treinamento

ID.

1. Você compreendeu o objetivo desta pesquisa?
   a: Sim
   b: Não
   c: Parcialmente

2. Você compreendeu o que é Programação Orientada a Aspecto?
   a: Sim
   b: Não
   c: Parcialmente

3. Você compreendeu a utilidade de uma ferramenta de Visualização de Software?
   a: Sim
   b: Não
   c: Parcialmente

4. Como você julga o treinamento recebido sobre a ferramenta de Visualização de Software apresentada?
   a: Ruim
   b: Regular
   c: Bom
   d: Ótimo
   e: Excelente

5. Quais os seus comentários para melhorar os treinamentos teóricos e práticos que você recebeu?

B.4 Questionário de Feedback de Revisão de Programa

ID.

1. Com qual tipo de Revisão de Programa você mais de adequou?
   a: Revisão de Programa *ad hoc*
   b: Revisão de Programa com suporte da ferramenta de Visualização de Software
   c: Indiferente

2. Avalie como foi o nível de ajuda provida pela ferramenta durante a revisão do programa:
   a: Nenhuma ajuda
   b: Pouca ajuda
   c: Ajuda considerável
   d: Muita ajuda

3. Avalie como foi o nível de ajuda provida pela ferramenta durante a revisão do programa no quesito localização de defeitos:
   a: Nenhuma ajuda
   b: Pouca ajuda
   c: Ajuda considerável
   d: Muita ajuda

4. Na sua opinião, quais são os pontos positivos na utilização da ferramenta em Revisão de Programa?

5. Na sua opinião, quais são os pontos negativos na utilização da ferramenta em Revisão de Programa?

6. Quais os seus comentários para melhorar a ferramenta de Visualização de Software?

B.5 Formulário de Localização de Defeitos e Descrição de Adendos

ID.

Localização dos Defeitos					
Hora	Nome do Pacote	Nome do Arquivo	Nome da Classe/Aspecto	Nome do Método/Adendo	Linhas

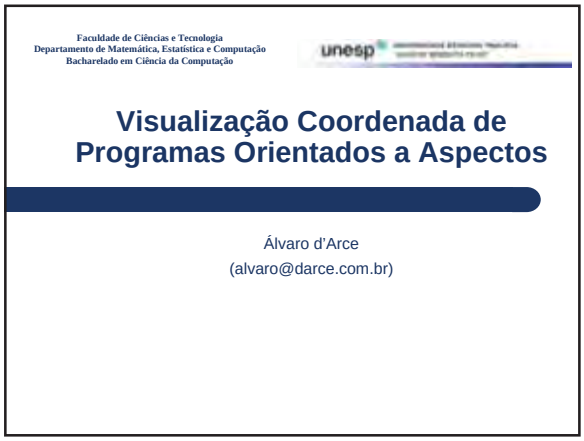
Descrição dos Adendos				
Nome do Pacote	Nome do Arquivo	Nome da Classe/Aspecto	Nome do Método/Adendo	Descrição

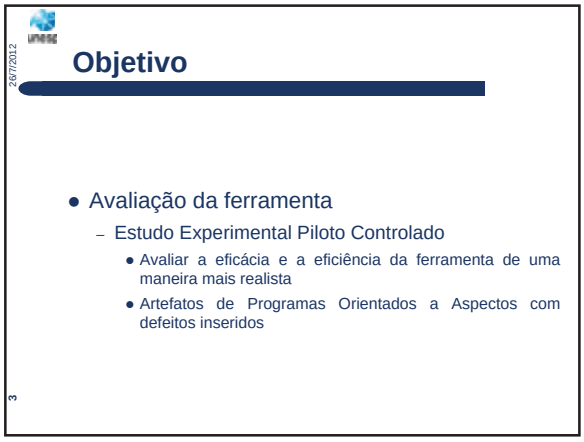
APÊNDICE

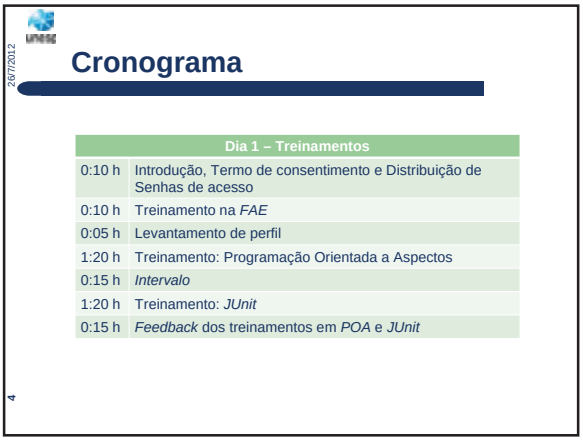
C

Slides de Treinamento

C.1 Apresentação do Experimento







26/7/2012

Cronograma

Dia 2 – Revisão de Programa – Parte 1		
	Grupo A	**Grupo B**
0:05 h	Preparação para a Revisão de Programa	
1:30 h	Revisão de Programa *ad hoc*	Revisão de Programa *ad hoc*
0:10 h	*Feedback* da Revisão de Programa *ad hoc*	
0:15 h	*Intervalo*	
1:00 h	Treinamento: Visualização de Software e Compreensão de Programa	
0:15 h	*Feedback* do treinamento de VS e CP	

5

26/7/2012

Cronograma

Dia 3 – Revisão de Programa – Parte 2		
	Grupo A	**Grupo B**
1:20 h	Treinamento da Ferramenta de *VS*	
0:05 h	Preparação para a Revisão de Programa	
0:15 h	*Intervalo*	
1:30 h	Revisão de Programa apoiada pela Ferramenta de *VS* utilizando	Revisão de Programa apoiada pela Ferramenta de *VS*
0:10 h	*Feedback* do Treinamento da Ferramenta de VS e da Revisão de Programa com a ferramenta	

6

26/7/2012

Ferramentas

- FAE
- Ambiente Java
- Eclipse SDK
  - Plugin AJDT
- $SoftVis_{OAH}$

7

26/7/2012

Artefatos de Software

- Dois projetos open-source em Java:
  - *GanttProject*
    - Gerenciamento de projetos
  - *Memoranda*
    - Gerenciamento de agenda e apontamentos pessoais

8

C.2 Treinamento na ferramenta FAE

Faculdade de Ciências e Tecnologia
Departamento de Matemática, Estatística e Computação
Bacharelado em Ciência da Computação

unesp

FAE
Ferramenta de Apoio à Experimentação

Álvaro d'Arce
(alvaro@darce.com.br)

26/7/2012

Tópicos

FAE – Ferramenta de Apoio à Experimentação

2

26/7/2012

FAE
Tela Principal

3

26/7/2012

FAE
Login

4

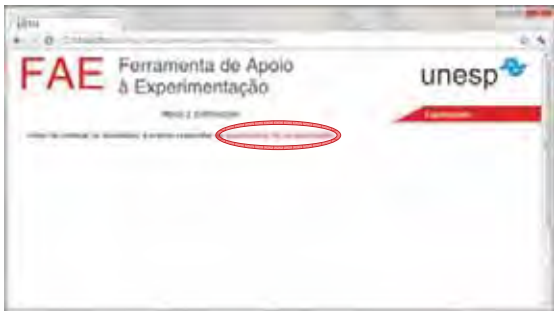
26/7/2012

FAE
Selecionando Experimento

5

26/7/2012

FAE
Primeiro acesso ao experimento



6

26/7/2012

FAE
Questionário de Caracterização [1/3]

7

26/7/2012

FAE
Questionário de Caracterização [2/3]

8

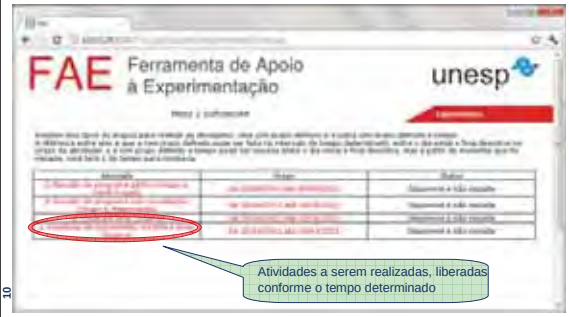
26/7/2012

FAE
Questionário de Caracterização [3/3]

9

26/7/2012

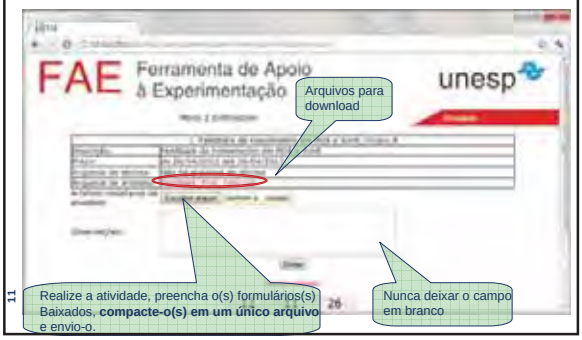
FAE
Atividades [1/3]



10

26/7/2012

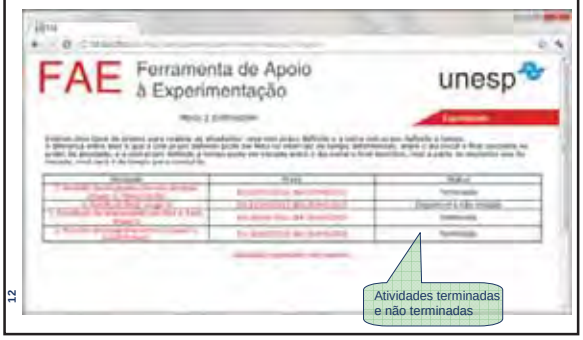
FAE
Atividades [2/3]



11

26/7/2012

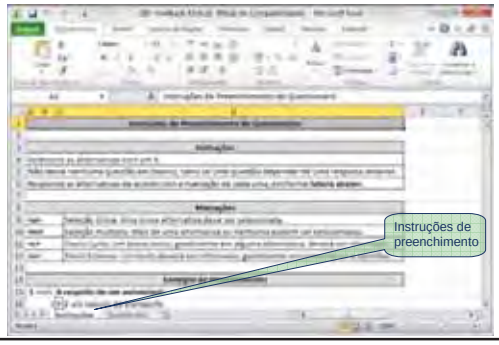
FAE
Atividades [3/3]



12

26/7/2012

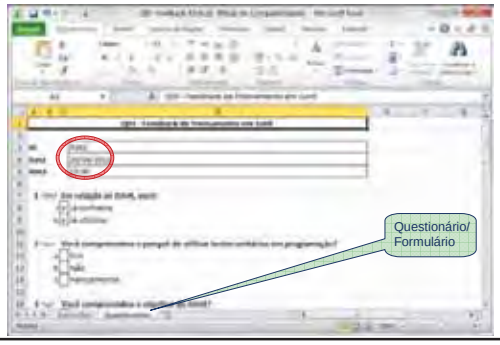
FAE
Questionários/Formulários [1/2]



13

26/7/2012

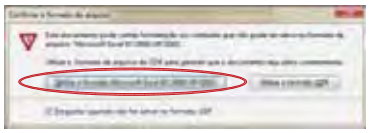
FAE
Questionários/Formulários [2/2]



14

26/7/2012

FAE
Salvar Arquivos para Compactação e Envio

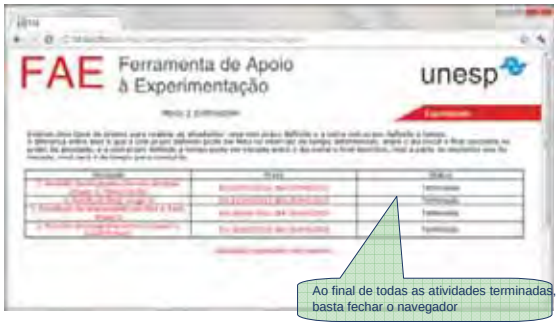


- No Libre Office: Selecione *formato Excel* ao salvar arquivos
- Compacte os arquivos a serem enviados em um único arquivo zip

15

26/7/2012

FAE
Finalizando



16

26/7/2012

Dúvidas?

17

C.3 Treinamento em Programação Orientada a Aspectos

Faculdade de Ciências e Tecnologia
Departamento de Matemática, Estatística e Computação
Bacharelado em Ciência da Computação

unesp

AspectJ

Álvaro d'Arce
Rogério Garcia

26/7/2012

Roteiro

Introdução
Programação Orientada a Objeto
Programação Orientada a Aspecto
AspectJ

2

26/7/2012

Introdução

- Os termos Desenvolvimento Estruturado e Orientação a Objeto dizem respeito à modularidade do sistema
  - São maneiras distintas de se dividir um sistema em partes
- A divisão em partes é importante para se reduzir a complexidade
  - É muito difícil para um ser humano compreender um sistema de grande porte se este for monolítico, sem fronteiras claras que definem suas funções

3

26/7/2012

Roteiro

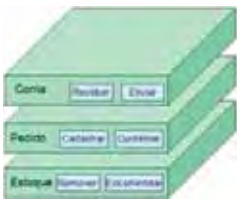
Introdução
Programação Orientada a Objeto
Programação Orientada a Aspecto
AspectJ

4

26/7/2012

Programação Orientada a Objeto

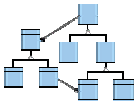
- Define que a separação de interesses deve acontecer em duas dimensões, em termos de dados (objetos) e em termos das funções (métodos) que utilizam cada tipo de dados



5

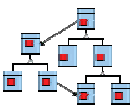
26/7/2012

Programação Orientada a Objeto
Interesses Transversais



```
class Tracing {
  public static void entry(String s) {
    System.out.println("entry: " + s);
  }
  public static void exit(String s) {
    System.out.println("exit: " + s);
  }
}
```

```
class Person {
  private String name = "";
  public void setName(String name) {
    Tracing.entry("setName("+name+")");
    this.name = name;
    Tracing.exit("setName()");
  }
  ...
}
```



6

Programação Orientada a Objeto
Interesses Transversais

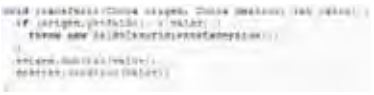
```
public void executarOperacao(){
    // efetuar logging
    ...
    // verificar autorização e autenticação
    ...
    // iniciar transação
    ...
    // efetuar regra de negócios
    ...
    //finalizar transação
    ...
    //liberar recursos adquiridos
    ...
}
```

Funcionalidades secundárias

misturadas com as funcionalidades principais

Programação Orientada a Objeto
Interesses Transversais

- Código simples:

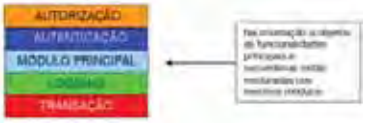


Programação Orientada a Objeto
Interesses Transversais

- Código com log, controle de transação e tratamento de exceção:

Programação Orientada a Objeto
Interesses Transversais

- Os interesses transversais causam dois problemas:
  - **Espalhamento** (*spread code*): interesse espalhado em vários módulos
  - **Emaranhamento** (*tangled code*): único módulo implementa interesses distintos



Roteiro

Introdução

Programação Orientada a Objeto

Programação Orientada a Aspecto

AspectJ

Programação Orientada a Aspecto

- Tecnologia que provê o encapsulamento dos interesses transversais em módulos (aspectos) fisicamente separados do restante do código
- Permite que os objetos tratem apenas dos interesses de negócio (funcionalidades principais) a que são destinados sem se importar com a maneira com que os interesses transversais (funcionalidades secundárias) serão tratados pelos aspectos
- Não tem o intuito de ser um novo paradigma autônomo, mas sim uma nova tecnologia que deve aliar-se à *POO*

26/7/2012

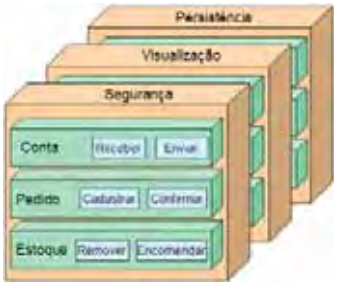
Programação Orientada a Aspecto
Interesses Transversais

- Requisitos não-funcionais (como registro para auditoria e de *tracing*) são interesses transversais ou entrecortantes, pois sua implementação "corta" a estrutura de módulos do sistema
- Além de decompor o sistema em objetos (dados) e métodos (funções), objetos e funções são decompostos de acordo com o interesse sendo servido
- Cada interesse é agrupado em um módulo distinto (aspecto)

13

26/7/2012

Programação Orientada a Aspecto
Interesses Transversais

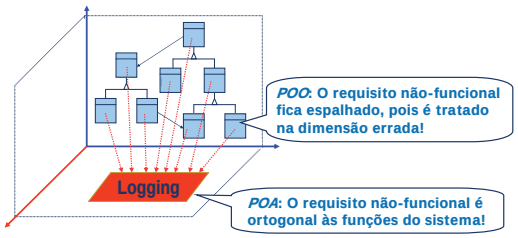


Organização almejada

14

26/7/2012

Programação Orientada a Aspecto
Interesses Transversais



15

26/7/2012

Programação Orientada a Aspecto
Interesses Transversais

- Exemplos:
  - Segurança
  - Persistência de dados
  - Auditoria e depuração
  - Tratamento de erros
  - Concorrência e sincronização
  - Regras e restrições do negócio
  - Entre outros

16

26/7/2012

Programação Orientada a Aspecto
Conceitos

- **Join Points** (Pontos de Junção)
  - Pontos identificáveis na execução de um programa
  - Ex: chamada de método, acesso a atributos (*get* / *set*), ocorrência de exceção etc
- **Pointcuts** (Pontos de Corte)
  - Regras criadas para especificar o local onde ocorrerá uma alteração no sistema
  - Localizam *join points* que atendam às regras impostas
  - Capturam dados do contexto de execução de cada *join point*

17

26/7/2012

Programação Orientada a Aspecto
Conceitos

- **Advices** (Adendos)
  - Código executado nos *join points* (regra torna-se válida)
  - Condição temporal, possibilitando alteração no fluxo, antes e/ou depois do *join point*
- **Aspects** (Aspectos)
  - Unidade de código similar a uma classe
  - Encapsula diversos *pointcuts* e *advices*

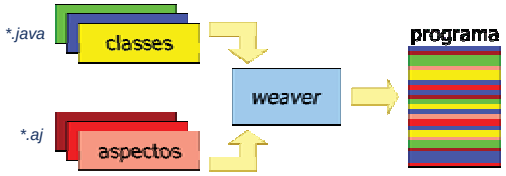
18

Programação Orientada a Aspecto
Conceitos

19

Programação Orientada a Aspecto
POO + POA

- **Classes**: encapsulam requisitos funcionais
  - Arquivos ***.java***
- **Aspectos**: encapsulam requisitos transversais
  - Arquivos ***.aj***
- **Weaver**: entrelaça classes e aspectos em um programa



20

Programação Orientada a Aspecto
Weaving

21

Roteiro

Introdução
Programação Orientada a Objeto
Programação Orientada a Aspecto
AspectJ

22

AspectJ

- É uma extensão do Java
- Seus programas consistem em construções da linguagem Java combinadas com as construções definidas pela linguagem *AspectJ*

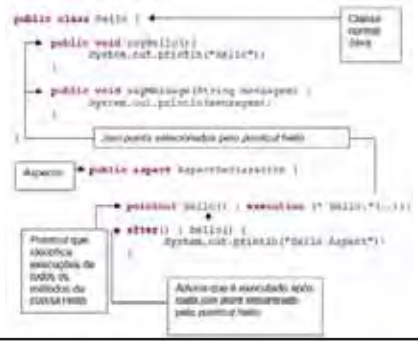
23

AspectJ
Estruturas

- Aspect
- Join point
- Pointcut
- Advice
- Introduction/Inter-type declaration

24

AspectJ
Aspect



25

AspectJ
Aspect

- Ponto central da Programação Orientada a Aspecto
- Estrutura que agrupa os *joinpoints*, *pointcuts*, *advices* e *introductions*,
  - Pode conter métodos e variáveis.

26

AspectJ
Join Point

- Ponto definido (identificável) no fluxo de execução de um programa, onde aspectos podem interceptar classes, interrompendo o seu curso natural para inserção de novas rotinas
- Ex:
  - Chamada/execução de um método/construtor
  - Acesso a um atributo
  - Tratamento de exceção
  - Inicialização de classe

27

AspectJ
Join Point

28

AspectJ
Pointcut

- Predicado que define um conjunto de *join points*
- É possível reunir *join points* de diferentes tipos e nomear o conjunto
- Permite a exposição do contexto do programa no *join point*
  - possibilitando ao aspecto acessar variáveis locais do programa
- Pode ou não ser nomeado

29

AspectJ
Pointcut

- Sintaxe:

```
public pointcut <nome> ([argumentos]): <design> (<assinatura>)
```

restrição de acesso

declaração de *pointcut*

nome do *pointcut*

parâmetros

designador (tipo) dos *join points*

predicado

30

AspectJ
Pointcut

- <designador>
  - call
  - execution
  - initialization
  - handler
  - get
  - set
  - this
  - target
  - args
  - within
  - withincode
  - cflow
  - cflowbelow

Pointcuts por tipo de join points
Pointcuts de objetos em execução
Pointcuts baseados na estrutura léxica
Pointcuts de fluxo de controle

26/7/2012 31

AspectJ
Pointcut

- <assinatura>

Caracteres especiais

Caractere	Significado
*	Qualquer sequência de caracteres não contendo pontos
..	Qualquer sequência de caracteres, inclusive contendo pontos
+	Qualquer subclasse de uma classe

26/7/2012 32

AspectJ
Pointcut

- <desig> + (<assinatura utilizando caracteres especiais>) = designadores genéricos
  - Qualquer tipo de retorno:
    ```
    call(* NomeClasse.NomeMetodo(..));
    ```
  - Qualquer classe:
    ```
    execution(* *.NomeMetodo(..));
    ```
  - Qualquer método:
    ```
    call(* NomeClasse.*(..));
    ```
  - Nome parcial:
    ```
    execution(* NomeClasse.meu*(..));
    ```
  - Classe e subclasse:
    ```
    call(NomeClasse+.new(..));
    ```

26/7/2012 33

AspectJ
Pointcut - Exemplos

Todas as chamadas a creditar de qualquer conta

```
pointcut logCredito():
    call (* Conta*.creditar(double));

pointcut logDebito():
    call (* Conta*.debitar(double));
```

26/7/2012 34

AspectJ
Pointcut - Exemplos

- Pointcuts por tipo de join points

```
call(void Conta.creditar(double))
execution(void Conta.creditar(double))
get(double Conta.saldo)
set(String Conta.numero)
```

26/7/2012 35

AspectJ
Pointcut - Exemplos

- Pointcuts por tipo de join points

```
call(void Cliente.*(*))
```

Todas as chamadas a qualquer método de Cliente que tenha apenas um parâmetro qualquer e retorne void

```
call(* Cliente.*(*))
```

Todas as chamadas a qualquer método de Cliente que tenha apenas um parâmetro qualquer e retorne qualquer tipo

26/7/2012 36

AspectJ
Pointcut - Exemplos

- Pointcuts por tipo de join points

```
execution(void Conta.set*(..))
```

Todas as execuções a métodos set de Conta que tenham qualquer número de parâmetros, de qualquer tipo

```
execution(void Conta+.set*(..))
```

Execuções a qualquer método set de Conta e suas subclasses

26/7/2012 37

AspectJ
Pointcut - Exemplos

- Pointcuts por tipo de join points

```
get(String Cliente.*)
```

Todos os acessos de leitura a atributos de Cliente do tipo String

```
set(* *.*)
```

Todos os acessos de escrita a qualquer atributo de qualquer classe - do sistema todo

26/7/2012 38

AspectJ
Pointcut - Exemplos

- Pointcuts por tipo de join points

```
call(Cliente.new(..))
```

Todas as chamadas a qualquer construtor de Cliente

```
handler(IOException)
```

Todos os lançamentos da exceção IOException

26/7/2012 39

AspectJ
Pointcut - Exemplos

- Pointcuts de objetos em execução

restringe tipo de onde vem a chamada

```
poincut setContadaFachada():
      call(* Conta.creditar(*)) &&
      this(Fachada);
```

diferentes!

```
poincut setContaPoupanca():
      execution(* Conta.creditar(*)) &&
      this(Poupanca);
```

26/7/2012 40

AspectJ
Pointcut - Exemplos

- Pointcuts de objetos em execução

restringe tipo de quem recebe a chamada

```
poincut setContaPoupanca():
      call(* Conta.creditar(*)) &&
      target(Poupanca);
```

26/7/2012 41

AspectJ
Pointcut - Exemplos

- Pointcuts baseados na estrutura léxica

```
poincut setSaldoEmConta():
      call(* Conta.setSaldo(..)) &&
      within(Conta);
```

só queremos as chamadas a setSaldo que forem feitas de dentro da classe Conta

26/7/2012 42

AspectJ
Pointcut – Exemplos (1/2)

- Pointcuts de fluxo de controle

```
pointcut setSaldoNoSaque():
      call(* Conta.setSaldo(..)) &&
      cflow(execution(* Fachada.saque(..)));
```

só queremos as chamadas a setSaldo que forem feitas no fluxo de execução do método saque da Fachada

43

AspectJ
Pointcut – Exemplos (2/2)

- Pointcuts de fluxo de controle

```
public class Fachada { ...
  public void saque(String num, double valor){
     ...c.debitar(valor);...
  }
public class Conta { ...
  public void debitar(double valor){
    if (this.getSaldo() >= valor)
      this.setSaldo(this.getSaldo()-valor);
  }
  ...
  public void setSaldo(double saldo) {
     this.saldo = saldo;
  }
}
```

Seria interceptado apenas quando saque for chamado

44

AspectJ
Pointcut – Expressões lógicas

- Ligação de designadores

Operador	Significado	Exemplo
!	Negação	! Receita
||	Ou	Receita || Ingrediente
&&	E	Receita && Ingrediente

45

AspectJ
Pointcut – Expressões lógicas - Exemplos

- Ligação de designadores

```
poincut setCliente():call(* Cliente.set*(*));
pointcut setConta():call(* Conta.set*(*));

poincut sets(): setCliente() || setConta();
```

OU

```
poincut sets():
     call(* Cliente.set*(*)) ||
     call(* Conta.set*(*));
```

46

AspectJ
Advice

- Trechos de códigos associados a *pointcuts* que injetam um novo comportamento em cada *joinpoint* representado pelo *pointcut*

- *Sintaxe:*

```
especificação(): pointcut() {
   //... código do advice
}
```

47

AspectJ
Advice

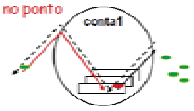
Especificação	Significado
`before()`	Adendo executado antes do *join point* ser alcançado
`after() returning(<tipo><var>)`	Adendo executa após a execução normal do *join point* (sem exceção)
`after() throwing(<tipo><var>)`	Adendo executa após a execução com exceção do *join point*
`after()`	Adendo executado após a execução do *join point* em qualquer situação
`around()`	Adendo executado quando o *join point* é alcançado, substituindo sua execução ou executando com o contexto alterado (utilizando o comando `proceed()`)

48

AspectJ
Advice - Exemplos

- before()

before executa antes de chegar no ponto



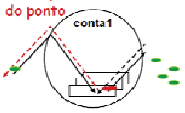

```
pointcut setSaldo():
  call(* Conta.setSaldo(..));

before(): setSaldo(){
  System.out.println("vai mudar o saldo");
}
```

49

AspectJ
Advice - Exemplos

- after()

after executa na hora que estiver voltando do ponto




```
pointcut setSaldo():
  call(* Conta.setSaldo(..));

after(): setSaldo(){
  System.out.println("mudou o saldo");
}
```

50

AspectJ
Advice - Exemplos

- after() returning() / thowing()

```
pointcut debitos():
  call(* Conta.debitar(..));

after() returning: debitos(){
  System.out.println("debito deu certo!");
}

after() throwing: debitos(){
  System.out.println("debito deu errado!");
}
```

51

AspectJ
Advice - Exemplos

- around()

```
pointcut logDebito(Conta c,double v):
  call (* Conta.debitar(double)) &&
  target(c) && args(v);
void around(Conta c,double v):logDebito(c,v){
    if(v > c.getSaldo())
        System.out.println("Sem saldo!");
    else
        proceed(c,v);
}
```

52

AspectJ
Advice – Passagem de Contexto - Exemplos

- Passagem e utilização de contexto

```
pointcut logCredito(Conta c):
  call (* Conta.creditar(double)) &&
  target(c);
```

conta c que recebeu a chamada

```
after(Conta c): logCredito(c){
  System.out.println("ocorreu credito");
  System.out.println("num: "+c.getNumero());
}
```

imprime numero da conta creditada

53

AspectJ
Advice – Passagem de Contexto - Exemplos

- Passagem e utilização de contexto

```
pointcut logCredito(Conta c,double v):
  call (* Conta.creditar(double)) &&
  target(c) && args(v);
```

define o parametro - valor do credito definido

```
after(Conta c,double v): logCredito(c,v){
  System.out.println("ocorreu credito");
  System.out.println("num: "+c.getNumero());
  System.out.println("valor: " + v);
}
```

imprime valor

54

AspectJ
Advices x Métodos

- Advices não tem nome
- Advices não podem ser chamados diretamente (são chamados pelo sistema)
- Advices não têm especificadores de acesso
- Advices têm acesso a outras variáveis especiais do this

55

AspectJ
Introduction - Exemplos

- Também chamado de *Inter-type Declaration*
- Introdução de novos atributos e métodos a estrutura de uma classe
  - Sintaxe:

```
private <tipo> <classe>.<atributo>;

public <tipo> <classe>.<nomeMetodo>(<param>) {
    //corpo do método
}
```

56

AspectJ
Introduction - Exemplos

```
public aspect RegraSaldoMinimo {
   private float Conta.saldoMinimo; //Introduz atributo

   public float Conta.obtemSaldoDisponivel() { //Introduz método
      return getSaldo() - saldoMinimo;
   }

   after(Conta conta) : execution(ContaPoupanca.new(..)) &&
                        this(conta) {
      conta.saldoMinimo = 25;
   }
}
```

Executa quando uma instância de ContaPoupanca é criada, atribuindo o saldo mínimo de 25

57

AspectJ
Introduction

- Além de introduzir membros, pode-se acrescentar relacionamento de herança a classes
  - Fazer com que classe(s) implemente(m) interface(s):
    `declare parents: <classe1> implements <classe2>;`
  - Fazer com que classe(s) seja(m) filha(s) de uma classe:
    `declare parents: <classe1> extends <classe2>;`
- A nova hierarquia de classes só é útil no contexto dos aspectos

58

AspectJ
Reflexão

- Existem três objetos que podem ser acessados de dentro de um *advice* para determinar informações sobre onde o aspecto está sendo executado:
  - thisJoinPoint
  - thisJoinPointStaticPart
  - thisJoinPointEnclosingStaticPart

59

AspectJ
Criando um projeto (HelloWorld)

- Criando uma classe java
  - Botão direito sobre o projeto -> *New* -> *Class*
  - Nome da classe: *principal*
- Código na classe:

```
public class principal {
    public static void main (String[] args)
    {
        System.out.println("Hello na main!");
    }
}
```

60

26/7/2012

AspectJ
Criando um projeto (HelloWorld)

- Criando um aspecto
  - Botão direito sobre projeto -> *New -> Other...*
- *-> AspectJ -> Aspect*
- Nome do aspecto: *aspecto*

61

26/7/2012

AspectJ
Criando um projeto (HelloWorld)

- Código do aspecto:

```
public aspect aspecto {
    pointcut primeiroHello() : execution (* principal.*(..));
    pointcut terceiroHello() : execution (* principal.*(..));

    before(): primeiroHello()
    {
        System.out.println("Hello antes da main!");
    }

    after(): terceiroHello()
    {
        System.out.println("Hello depois da main!");
    }
}
```

62

26/7/2012

AspectJ
Criando um projeto (HelloWorld)

- Visualizando referências cruzadas
  - *principal.java:*

63

26/7/2012

AspectJ
Criando um projeto (HelloWorld)

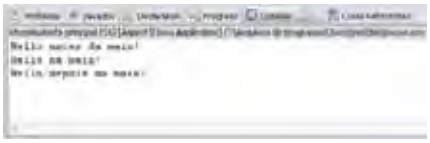
- Visualizando referências cruzadas
  - *aspecto.aj:*

64

26/7/2012

AspectJ
Criando um projeto (HelloWorld)

- Saída:

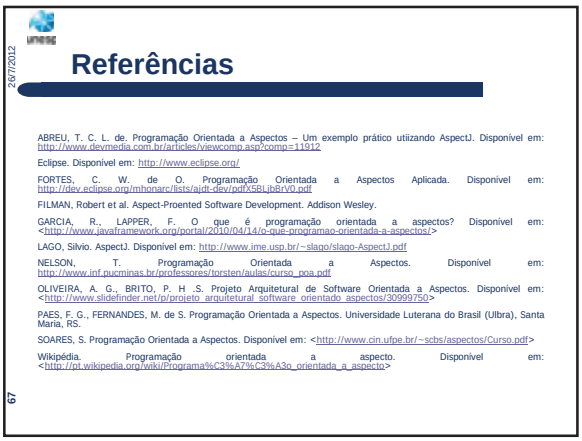


65

26/7/2012

Dúvidas?

66



26/7/2012

Referências

ABREU, T. C. L. de. Programação Orientada a Aspectos – Um exemplo prático utiizando AspectJ. Disponível em: http://www.devmedia.com.br/articles/viewcomp.asp?comp=11912

Eclipse. Disponível em: http://www.eclipse.org/

FORTES, C. W. de O. Programação Orientada a Aspectos Aplicada. Disponível em: http://dev.eclipse.org/mhonarc/lists/ajdt-dev/pdfX5BLjbBrV0.pdf

FILMAN, Robert et al. Aspect-Proented Software Development. Addison Wesley.

GARCIA, R., LAPPER, F. O que é programação orientada a aspectos? Disponível em: <http://www.javaframework.org/portal/2010/04/14/o-que-programao-orientada-a-aspectos/>

LAGO, Silvio. AspectJ. Disponível em: http://www.ime.usp.br/~slago/slago-AspectJ.pdf

NELSON, T. Programação Orientada a Aspectos. Disponível em: http://www.inf.pucminas.br/professores/torsten/aulas/curso_poa.pdf

OLIVEIRA, A. G., BRITO, P. H .S. Projeto Arquitetural de Software Orientada a Aspectos. Disponível em: <http://www.slidefinder.net/p/projeto_arquitetural_software_orientado_aspectos/30999750>

PAES, F. G., FERNANDES, M. de S. Programação Orientada a Aspectos. Universidade Luterana do Brasil (Ulbra), Santa Maria, RS.

SOARES, S. Programação Orientada a Aspectos. Disponível em: <http://www.cin.ufpe.br/~scbs/aspectos/Curso.pdf>

Wikipédia. Programação orientada a aspecto. Disponível em: <http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_orientada_a_aspecto>

67

C.4 Treinamento em JUnit

Faculdade de Ciências e Tecnologia
Departamento de Matemática, Estatística e Computação
Bacharelado em Ciência da Computação

Testes de Unidade com JUnit

Álvaro d'Arce
(alvaro@darce.com.br)

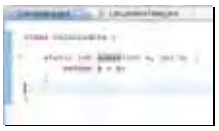
Testes de Programas
JUnit

- Framework de criação de testes automatizados para desenvolvimento Java
- Possui API que habilita o desenvolvedor a facilmente criar casos de teste em Java
- Provê abrangente facilidade de asserção
  - Verificar resultados esperados x resultados reais
- Utiliza um princípio fundamental da programação XP:
  - Criação e execução de testes deve ser fácil

JUnit
Questões

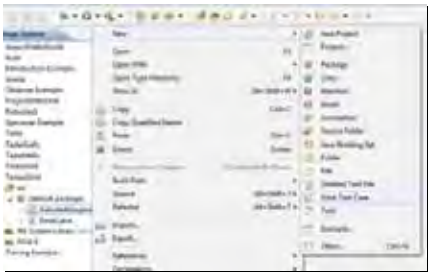
- Casos de teste são definidos em classes separadas
  - Sem acesso a partes encapsuladas
- Testes são realizados a partir da interface de um objeto ao mundo externo
- Hábito:
  1. Codifique um pouco...
  2. Teste um pouco...
  3. Codifique um pouco...
  4. Teste um pouco...
  - Resumo: Objeto pronto, teste-o

JUnit no Eclipse
Exemplo: Calculadora



Classe a ser testada

JUnit no Eclipse
Criando um Caso de Teste



- Botão direito na classe a ser testada (Calculadora)
- *New*
- *JUnit Test Case*

JUnit no Eclipse
Criando um Caso de Teste

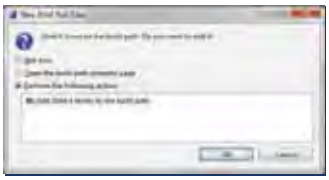
- Nome do caso de teste
- Classe a ser testada
- *Next*

JUnit no Eclipse
Criando um Caso de Teste

- Selecionar métodos a serem testados
- *Finish*

7

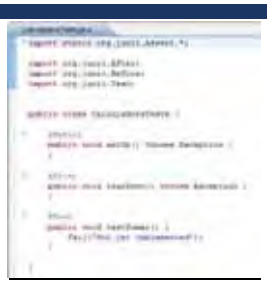
JUnit no Eclipse
Criando um Caso de Teste



- *Perform the following action...*
- *Ok*

8

JUnit no Eclipse
Criando um Caso de Teste



- Esqueleto do caso de teste
- *testSomar()*:
  - Testes para o método somar()

9

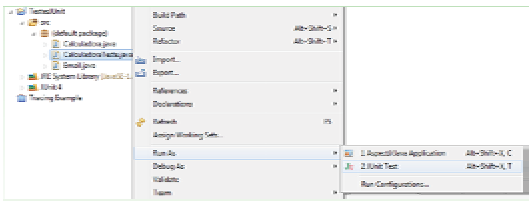
Exemplo: Calculadora
Implementando o Caso de Teste criado

Classe a ser testada

Caso de teste

10

Exemplo: Calculadora
Executando o Caso de Teste - Plugin



- Botão direito no caso de teste
- *Run As*
- *JUnit Test*

11

Exemplo: Calculadora
Resultado dos testes - Plugin

Resultado sem falhas

Exemplo de resultado com falhas

- Falha: quando uma “afirmação” (assertion) falha
- Erro: quando ocorre uma exceção
  - Ex.: *NullPointerException*, *ArrayIndexOutOfBoundsException* ...

12

26/7/2012

JUnit
Assertions

Sintaxe do Método	Descrição	Teste passa se
assertEquals([String mensErro,] *esperado*, *atual*)	Compara dois valores	*esperado*.equals (*teste*)
assertFalse([String mensErro,] boolean *condicao*)	Avalia uma expressão booleana	*condicao* == false
assertTrue([String mensErro,] boolean *condicao*)		*condicao* == true
assertNotNull([String mensErro,] Object *objeto*)	Compara um objeto com nulo	*objeto* != null
assertNull([String mensErro,] Object *objeto*)		*objeto* == null
assertNotSame([String mensErro,] Object *esperado*, Object *atual*)	Compara dois objetos	*esperado* != *atual*
assertSame([String mensErro,] Object *esperado*, Object *atual*)		*esperado* == *atual*
fail([String mensErro])	Causa uma falha no teste atual (comumente usado em manipulação de exceções)	

13

26/7/2012

JUnit
Anotações do JUnit 4

- **@Test**
  - Identifica que o método é um método de teste.
- **@Before**
  - Executa o método antes de cada teste. Este método pode ser usado para preparar o ambiente de teste (Ex: ler dados do usuário). Substituto do setUp().
- **@After**
  - Executa o método após cada teste. Substituto do tearDown().
- **@BeforeClass**
  - Executa o método antes do início de todos os testes
    - Ex: conectar base de dados

14

26/7/2012

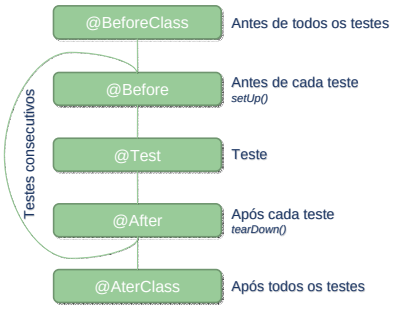
JUnit
Anotações do JUnit 4

- **@AfterClass**
  - Executa o método após todos os testes finalizarem (Ex: desconectar base de dados).
- **@Ignore**[**("Comentário")**]
  - O método é ignorado.
- **@Test (***expected=IllegalArgumentException.class***)**
  - Testa se o método levanta a exceção especificada.
- **@Test (***timeout=100***)**
  - Falha se o teste demorar mais que 100 milissegundos

15

26/7/2012

JUnit
Sequência de Desenvolvimento

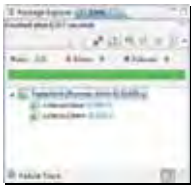


16

26/7/2012

Exemplo
Sequência dos métodos

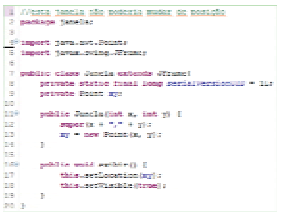
```
public class TesteJUnit {
    private Collection colecao;
    @BeforeClass
    public static void inicializacaoGeral() {
        System.out.println("@BeforeClass: inicializacaoGeral()");
    }
    @AfterClass
    public static void finalizacaoGeral() {
        System.out.println("@AfterClass: finalizacaoGeral()");
    }
    @Before
    public void antes() {
        colecao = new ArrayList();
        System.out.println("@Before: antes()");
    }
    @After
    public void apos() {
        colecao.clear();
        System.out.println("@After: apos()");
    }
    @Test
    public void colecaoVazia() {
        assertTrue(colecao.isEmpty());
        System.out.println("@Test: colecaoVazia()");
    }
    @Test
    public void colecaoItem() {
        colecao.add("Primeiro");
        assertEquals(1, colecao.size());
        System.out.println("@Test colecaoItem");
    }
}
```



17

26/7/2012

Exemplo: Janela
Classe Janela



18

Exemplo: Janela
Caso de Teste: JanelaTest


```
package janela;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class JanelaTest {
    private Janela jan;

    @Before
    public void setUp() throws Exception {
        jan = new Janela(30, 20);
    }

    @After
    public void tearDown() throws Exception {
        jan = null;
    }

    @Test
    public void testExibir() {
        jan.exibir();
        assertEquals(jan.getLocation().x, 30);
        assertEquals(jan.getLocation().y, 20);
        assertTrue(jan.getTitle(), "30,20");
    }
}
```

- A classe *Janela* está correta?

Exemplo: Janela
Caso de Teste: JanelaTest


```
package janela;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class JanelaTest {
    private Janela jan;

    @Before
    public void setUp() throws Exception {
        jan = new Janela(30, 20);
    }

    @After
    public void tearDown() throws Exception {
        jan = null;
    }

    @Test
    public void testExibir() {
        jan.exibir();
        jan.setLocation(50, 50);
        assertEquals(jan.getLocation().x, 50);
        assertEquals(jan.getLocation().y, 20);
        assertEquals(jan.getTitle(), "50,20");
    }
}
```

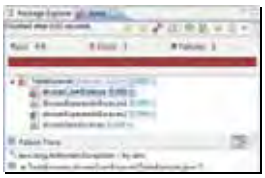
Mudar posição

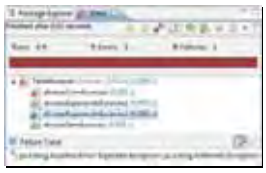
- A classe *Janela* está correta?

Exemplo: Esperando exceções
Caso de Teste


```
public class TesteExcecao {

    @Test
    public void divisaoComExcecao() {
      int i = 4 / 0; //teste resulta em erro por causa da exceção
      System.out.println("divisaoComExcecao(): " + i);
    }

    @Test(expected = ArithmeticException.class)
    public void divisaoEsperandoExcecao1() {
      int i = 4 / 0; //teste passa, pois espera a exceção
      System.out.println("divisaoEsperandoExcecao": " + i);
    }

    @Test(expected = ArithmeticException.class)
    public void divisaoEsperandoExcecao2() {
      int i = 4 / 4; //teste falha, pois não ocorre a exceção esperada
      System.out.println("divisaoEsperandoExcecao2(): " + i);
    }

    @Test
    public void divisaoSemExcecao() {
      int i = 4 / 2;
      System.out.println("divisaoSemExcecao(): " + i);
    }
}
```


Exemplo: Esperando exceções
Resultado





- ***divisaoComExcecao()***: erro
  - Exceção ocorrida (divisão por zero)
- ***divisaoEsperandoExcecao1()***: passou
  - Ocorreu exceção esperada
- ***divisaoEsperandoExcecao2()***: falhou
  - Não ocorreu exceção esperada
- ***divisaoSemExcecao()***: passou

Exemplo: Memoria
Classe MemoriaS 1/3


```
public abstract class MemoriaS {
    public static final int BYTE=1; //define unidade como BYTE
    public static final int MB=2;   //define unidade como MB
    public static final int KB=3;   //define unidade como KB
    public static final int GB=4;   //define unidade como GB
    protected double total;
    protected double utilizadoKB;
    protected int unidade;

    //Construtor. Recebe como parâmetro o total a ser gravado e a unidade de dados
    public MemoriaS(int newTotal, int newUnidade){
        this.total=newTotal;
        this.unidade=newUnidade;
        this.utilizadoKB=0;
    }

    //Construtor. Recebe como parâmetro o total a ser gravado e define a unidade como "KB"
    public MemoriaS(int newTotal){
        this(newTotal, KB);
    }

    //Busca valor gravado
    public double getUtilizadoKB(){
        return this.utilizadoKB;
    }

    //Retorna perda na gravação
    public abstract double getPerda();
```


Exemplo: Memoria
Classe MemoriaS 2/3


```
    //Retorna espaço disponível real em KB
    public abstract double getEspacoDisponivelRealKB();

    //Retorna espaço disponível em KB
    public double getEspacoDisponivelKB(){
        return this.getConverteKB(this.total);
    }

    //Utilizado na gravação
    public boolean GravaMD(int newTamanho){
        if (this.getConverteKB(this.total) - this.utilizadoKB >= newTamanho) {
            this.utilizadoKB = this.utilizadoKB + newTamanho;
            return true;
        }
        return false;
    }

    //Converte unidade para KB
    public double getConverteKB(double valor){
        if (this.unidade == BYTE) {
            return valor / 1024;
        } else if (this.unidade == MB) {
            return valor * 1024;
        } else if (this.unidade == GB) {
            return valor * 1024 * 1024;
        }
        else return valor;
    }
```

Exemplo: Memoria
Classe MemoriaS 3/3

26/7/2012

```
61      //Busca unidade
62      public String getUnidade(){
63          if (this.unidade == BYTE) {
64              return "BYTE";
65          } else if (this.unidade == KB){
66              return "KB";
67          } else if (this.unidade == MB) {
68              return "MB";
69          } else return "GB";
70      }
71
72      //Busca percentual disponivel
73      public double getPercentualDisponivel(){
74          return (this.getConverteKB(this.total) - this.utilizadoKB)
75                  / this.getConverteKB(this.total) * 100;
76      }
77
78      public String toString(){
79          return "Percentual Disponivel: " + getPercentualDisponivel() +
80                  "%\nEspaco Livre: " + getEspacoDisponivelKB()+"KB\nEspaco Disponivel Real: " +
81                  getEspacoDisponivelReal(0) + "KB\nTemp: " + getVersa() + "KB";
82      }
83
84      public static void main(String args[]){
85          MemoriaS hd = new HD("66327", 10, MemoriaS.MB);
86          MemoriaS cd = new CD(650, MemoriaS.MB);
87          hd.GravaKB(100*24);
88          cd.GravaKB(660000);
89          System.out.println(hd);
90          System.out.println(cd);
91      }
92  }
```

25

Exemplo: Memoria
Classe CD 1/2

26/7/2012

```
3   public class CD extends MemoriaS {
4       public static final int ABERTO =1; //define o estado como aberto
5       public static final int FECHADO=2; //define o estado como fechado
6       protected int estado;
7
8       //Construtor. Recebe o total a ser gravado e a unidade
9       public CD(int newTotal, int newUnidade){
10          super(newTotal, newUnidade);
11          this.estado = ABERTO;
12      }
13
14      //Reescreve método abstrato. Retorna espaço real disponível em KB
15      public double getEspacoDisponivelRealKB() {
16          return this.getEspacoDisponivelKB()-super.getUtilizadoKB();
17      }
18
19      //Reescreve método abstrato. Retorna perda na gravação
20      public double getPerda() {
21          return this.getEspacoDisponivelKB()*0.02;
22      }
23
24      //Reescreve método abstrato. Utilizado para gravar, recebendo o tamanho
25      public boolean GravaKB(int newTamanho) {
26          if (this.estado == ABERTO) {
27              if (super.GravaKB(newTamanho)) {
28                  this.estado = 2;
29                  return true;
30              } else return false;
31          }
32          else return false;
33      }
```

26

Exemplo: Memoria
Classe CD 2/2

26/7/2012

```
35      //Busca estado (aberto ou fechado)
36      public String getEstado(){
37          if (this.estado == ABERTO)
38              return "ABERTO";
39          else
40              return "FECHADO";
41      }
42
43      public String toString(){
44          return "CD Estado: " + this.getEstado() + "\n" + super.toString();
45      }
46  }
```

27

Exemplo: Memoria
Classe HD

26/7/2012

```
3   public class HD extends MemoriaS {
4       protected String numeroSerie;
5
6       //Construtor. Recebe o numero da série, o tamanho do HD e a unidade
7       public HD(String newNumeroSerie, int newTotal, int newUnidade){
8           super(newTotal, newUnidade);
9           this.numeroSerie=newNumeroSerie;
10      }
11
12      //Reescreve método abstrato. Retorna o espaço real disponível em KB
13      public double getEspacoDisponivelRealKB() {
14          return this.getEspacoDisponivelKB()-super.getUtilizadoKB();
15      }
16
17      //Reescreve método abstrato. Retorna a perda na gravação
18      public double getPerda() {
19          return this.getEspacoDisponivelKB() / 1024 / 100;
20      }
21
22      //Busca o número de série do HD
23      public String getNumeroSerie(){
24          return this.numeroSerie;
25      }
26
27      public String toString(){
28          return "HD Numero de Serie: " + this.getNumeroSerie() + "\n"+super.toString();
29      }
30
31  }
```

28

Exemplo: Memoria
Caso de Teste MemoriaTeste

26/7/2012

```
3   import static org.junit.Assert.*;
4   import org.junit.After;
5   import org.junit.Before;
6   import org.junit.Test;
7
8   public class MemoriaTeste {
9       CD cd;
10      HD hd;
11
12      @Before
13      public void setUp() throws Exception {
14          cd = new CD(650000, 2);
15          hd = new HD("01", 1000, 3);
16      }
17
18      @Test
19      public void testGravaKB() {
20          assertEquals(cd.GravaKB(100000), true);
21          assertEquals(cd.GravaKB(200001), false);
22
23          assertEquals(hd.GravaKB(2000), true);
24          assertEquals(hd.GravaKB(2002), false);
25      }
26
27      @After
28      public void tearDown() throws Exception {
29          cd = null;
30          hd = null;
31      }
32  }
```

29

Exemplos
Classe Utils

26/7/2012

```
3   import java.math.BigInteger;
4
5   public class Utils {
6
7       public static String concatenaPalavras(String... palavras) {
8           StringBuilder buf = new StringBuilder();
9           for (String palavra : palavras)
10              buf.append(palavra);
11          return buf.toString();
12      }
13
14      public static String calculaFatorial(int numero) throws IllegalArgumentException {
15          if (numero < 1)
16              throw new IllegalArgumentException("parâmetro zero ou negativo (" + numero + ")");
17          BigInteger fatorial = new BigInteger("1");
18          for (int i = 2; i <= numero; i++)
19              fatorial = fatorial.multiply(new BigInteger(String.valueOf(i)));
20          return fatorial.toString();
21      }
22  }
```

30

Exemplos
Caso de Teste TesteUtils

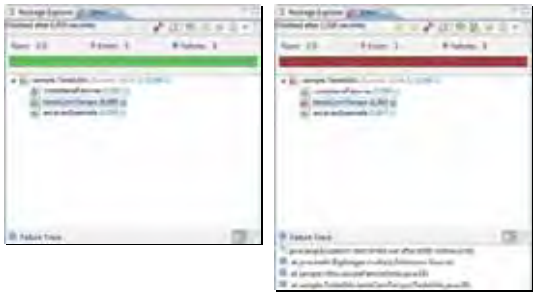
26/7/2012

```
import junit.framework.JUnit4TestAdapter;
import org.junit.Test;
import static org.junit.Assert.*;

public class TesteUtils {

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(TesteUtils.class);
    } //Necessário para criação de suítes via wizard - bug do JUnit 4

    @Test
    public void concatenaPalavras() {
        System.out.println("*** concatenaPalavras() ***");
        assertEquals("Hello, world!", Utils.concatenaPalavras("Hello", ", ", "world", "!"));
    }

    @Test(timeout = 1000)
    //Se o teste não terminar em 1 segundo, ele é interrompido e falha
    public void testeComTempo() {
        System.out.println("*** testeComTempo() ***");
        final int fatorialDe = 1 + (int) (20000 * Math.random()); // qualquer número 1 e 20000
        System.out.println("Computando fatorial de " + fatorialDe);
        System.out.println(fatorialDe + "! = " + Utils.calculaFatorial(fatorialDe));
    }

    @Test(expected = IllegalArgumentException.class)
    public void excecaoEsperada() {
        System.out.println("*** excecaoEsperada() ***");
        final int fatorialDe = -5; //número inválido para calculaFatorial()
        System.out.println(fatorialDe + "! = " + Utils.calculaFatorial(fatorialDe));
    }
}
```

31

Exemplos
Resultados de TesteUtils

26/7/2012



32

Exemplos
Classe Vetores

26/7/2012

```
public final class Vetores {

    //verifica se 2 vetores são iguais
    public static boolean vetoresIguais(int[] a, int[] b) {
        if ((a == null) || (b == null))
            throw new IllegalArgumentException("argumento nulo");
        if (a.length != b.length)
            return false;
        for (int i = 0; i < a.length; i++)
            if (a[i] != b[i])
                return false;
        return true;
    }

    //produto interno de 2 vetores
    public static int produtoInternoVetorial(int[] a, int[] b) {
        if ((a == null) || (b == null))
            throw new IllegalArgumentException("argumento nulo");
        if (a.length != b.length)
            throw new IllegalArgumentException(
                    "vetores de tamanhos diferentes ("
                            + a.length + ", " + b.length + ")");
        int sum = 0;
        for (int i = 0; i < a.length; i++)
            sum += a[i] * b[i];
        return sum;
    }
}
```

33

Exemplos
Caso de Teste TesteVetores

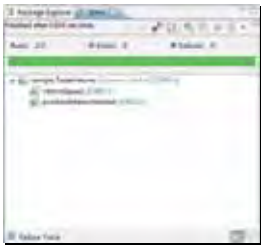
26/7/2012

```
import junit.framework.JUnit4TestAdapter;
import org.junit.Test;
import static org.junit.Assert.*;

public class TesteVetores {

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(TesteVetores.class);
    } //Necessário para criação de suítes via wizard - bug do JUnit 4

    @Test
    public void vetoresIguais() {
        System.out.println("*** vetoresIguais() ***");
        assertTrue(Vetores.vetoresIguais(new int[]{}, new int[]{}));
        assertTrue(Vetores.vetoresIguais(new int[]{0}, new int[]{0}));
        assertTrue(Vetores.vetoresIguais(new int[]{5, 6, 7}, new int[]{5, 6, 7}));

        assertFalse(Vetores.vetoresIguais(new int[]{1}, new int[]{0}));
        assertFalse(Vetores.vetoresIguais(new int[]{0}, new int[]{0, 0}));
        assertFalse(Vetores.vetoresIguais(new int[]{0}, new int[]{}));

        assertFalse(Vetores.vetoresIguais(new int[]{0, 0, 0}, new int[]{0, 0, 1}));
        assertFalse(Vetores.vetoresIguais(new int[]{0, 0, 0}, new int[]{0, 1, 0}));
        assertFalse(Vetores.vetoresIguais(new int[]{0, 0, 0}, new int[]{1, 0, 0}));
    }

    @Test
    public void produtoInternoVetorial() {
        System.out.println("*** produtoInternoVetorial() ***");
        assertEquals(0, Vetores.produtoInternoVetorial(new int[]{0}, new int[]{0}));
        assertEquals(0, Vetores.produtoInternoVetorial(new int[]{0}, new int[]{5}));
        assertEquals(-20, Vetores.produtoInternoVetorial(new int[]{-1, 4}, new int[]{15, -5}));
    }
}
```

34

Exemplos
Resultado de TesteVetores

26/7/2012



35

Referências

26/7/2012

Eric M. Burke & Brian M. Coyner. Java Extreme Programming Cookbook, O'REILLY.

Site Oficial do JUnit. Disponível em <http://www.junit.org>

Manual JUnit. Disponível em <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>

JUnit Cookbook. Disponível em <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

JUnit, um framework de testes. Disponível em <http://www.webartigos.com/artigos/junit-um-framework-de-testes-passo-a-passo-pelo-ide-netbeans/86117>

36

C.5 Treinamento em Visualização de Software e Compreensão de Programa

Faculdade de Ciências e Tecnologia
Departamento de Matemática, Estatística e Computação
Bacharelado em Ciência da Computação

Visualização de Software e Compreensão de Programa

Álvaro d'Arce
(alvaro@darce.com.br)

Roteiro

Visualização de Informação
Compreensão de Programa
Visualização de Software
Considerações Finais

2

Visualização de Informação

- A capacidade humana de lidar com informações visuais é maior do que com dados textuais

[Iepsen; Luzzardi;Loh, 2007]

3

Visualização de Informação
Definições

- "Processo de criação de uma apresentação gráfica de dados abstratos, usualmente não-numéricos"

[Knight, 2000]

4

Visualização de Informação
Definições

- "Representação de semântica ou significado de informação ... "
- "... Ferramenta de exploração visual que permite ao usuário interagir com o conteúdo visualizado e compreender seu significado"

[Chen, 2005]

5

Visualização de Informação
Objetivo

- Provê apoio à tarefa de análise de dados por meio de representação visual, permitindo que o usuário interaja com os gráficos produzidos como resultado da representação visual

[Card et al., 1999]

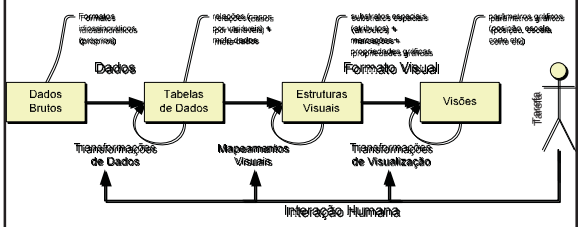
6

Visualização de Informação
Processo de Visualização

- Para que um conjunto ou amostra de dados possa ser visualizado, é necessário que tais dados sejam submetidos, primeiramente, a um processo o qual inclui etapas de transformações e mapeamentos de dados

7

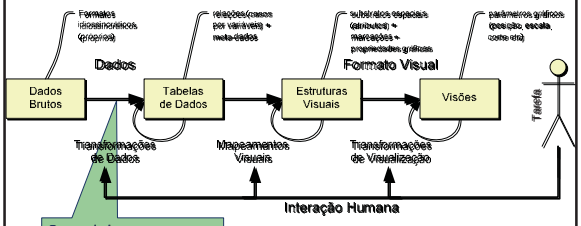
Visualização de Informação
Processo de Visualização



Processo de Visualização generalizado, adaptado de Card et al (1999)

8

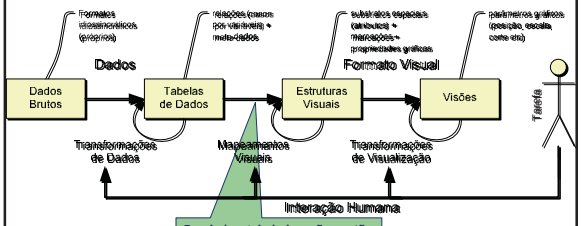
Visualização de Informação
Processo de Visualização



Os dados a serem visualizados, em seu formato bruto devem ser tabelados e organizados em um formato específico

9

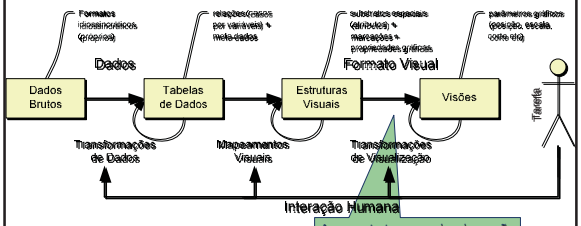
Visualização de Informação
Processo de Visualização



Os dados tabelados são, então, mapeados para estruturas visuais representativas (demonstrar visualmente o significado do dado), como formas geométricas, cores e texturas

10

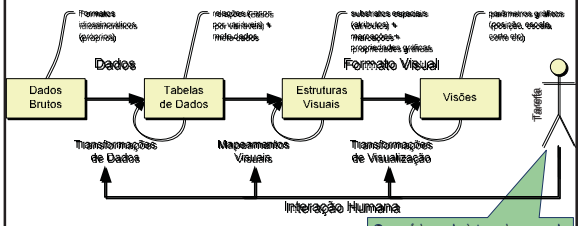
Visualização de Informação
Processo de Visualização



As estruturas visuais são apresentadas ao usuário em visões com base em seus parâmetros gráficos, como posição, escala, corte e outros

11

Visualização de Informação
Processo de Visualização



O usuário pode interagir em cada etapa de transformação e mapeamento de dados, personalizando a visualização de acordo com suas necessidades

12

26/7/2012

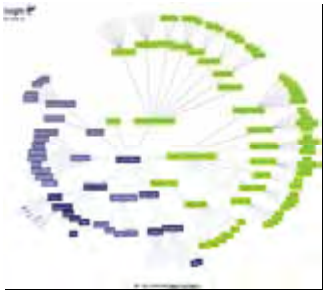
Visualização de Informação
TreeMap

Emissão total de CO_2 de combustíveis fósseis em 2008 (milhões de toneladas)

13

26/7/2012

Visualização de Informação
Árvore Hiperbólica

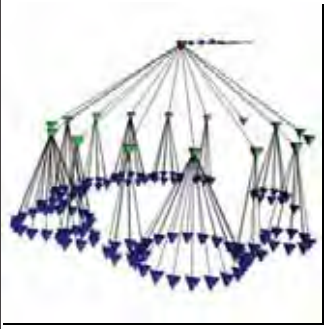


- Permite representar grandes estruturas hierárquicas por meio de uma projeção hiperbólica
  - possibilitando um maior detalhamento da vizinhança de um determinado nó ou ponto da representação visual

14

26/7/2012

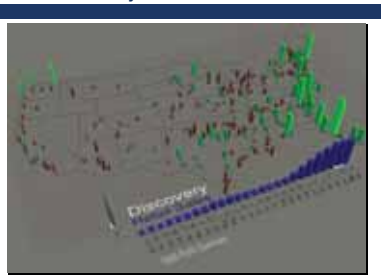
Visualização de Informação
Cones Euclidianos



- Estruturas Hierárquicas
- Mais informações que 2D

26/7/2012

Visualização de Informação
Vendas no Varejo

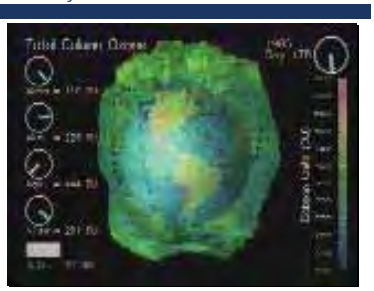


- Exibição
  - Latitude e Longitude (eixos X e Y)
  - Lucro (eixo Z)
  - Região (cor)

16

26/7/2012

Visualização de Informação
Concentração de Ozônio



- Exibição
  - Camada de Ozônio (eixos X, Y e Z)
  - Concentração de Ozônio (cor)

17

26/7/2012

Visualização de Informação
FilmFinder

- Exibição
  - Ano (eixo X)
  - Qualidade (eixo Y)
  - Classificação (cor)
- Sliders
  - Título, Ator, Atriz, Diretor, Duração e Censura

18

Roteiro

Visualização de Informação
Compreensão de Programa
Visualização de Software
Considerações Finais

19

Compreensão de Programa

- Envolve o entendimento individual de um programador ou analista a respeito do quê um programa faz e como o faz, a fim de realizar mudanças funcionais e extensões sem introduzir defeitos

[Corritore e Wiedenbeck, 2001]

20

Compreensão de Programa

- Papel estratégico na Engenharia de Software.
  - Consome tempo, esforço e representa um custo considerável nas atividades de desenvolvimento e manutenção

[Porto, 2009]

- Devido ao aumento de complexidade estrutural dos sistemas de software, tarefas de Compreensão de Programa tornam-se mais complexas de serem realizadas

21

Compreensão de Programa

- Tais situações motivam o uso de meios alternativos para apoiar a aplicação de técnicas de Compreensão de Programa
  - Visualização de Software

22

Roteiro

Visualização de Informação
Compreensão de Programa
Visualização de Software
Considerações Finais

23

Visualização de Software

- Abordagem alternativa
  - para auxiliar engenheiros de software a lidar com complexidade estrutural
  - no auxílio às tarefas de Compreensão de Programa
    - permitindo ao usuário interagir com representações gráficas
    - por meio de auxílio no processo cognitivo, analisar dados obtidos

24

26/7/2012

Visualização de Software
Definições

- "Uso de artesanatos de tipografias, *design* gráfico, animação e cinematografia com moderna interação humano-computador e tecnologia de computação gráfica para facilitar o entendimento humano e o uso efetivo de *software* de computador"

[Price et al, 1998]

25

26/7/2012

Visualização de Software
Definições

- Subconjunto especializado de Visualização de Informação... Envolve tarefas de mapeamentos para representar dados abstratos extraídos de Artefatos de Software em estruturas visuais"

[Knight, 2000]

26

26/7/2012

Visualização de Software
Definições

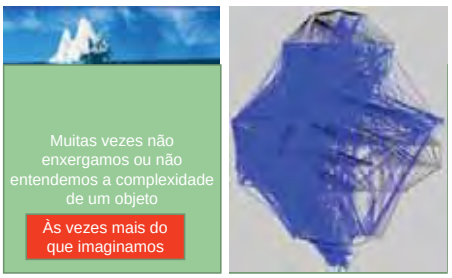
- Abrange tópicos, como visualização de programa, animação de algoritmo, programação visual, navegadores de código-fonte e visualização de dados

[Maletic e Collard, 2002]

27

26/7/2012

Visualização de Software
Auxílio em Compreensão de Programa



28

26/7/2012

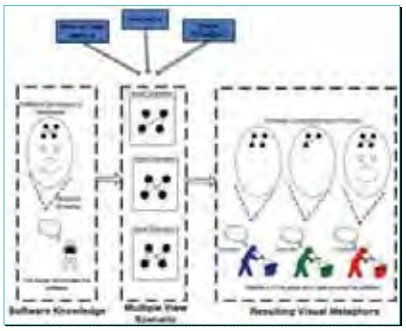
Visualização de Software
Auxílio em Compreensão de Programa

- Utiliza a capacidade humana de lidar com informações visuais em tarefas de entendimento de programas
  - Representações gráficas de um artefato de software
- Apoio a tarefas de Compreensão de Programa
  - Auxiliar o processo cognitivo
    - Evidenciar elementos que forneçam algo significativo para ser analisado

29

26/7/2012

Visualização de Software
Exemplo de Processo Cognitivo



30

Visualização de Software
Técnicas de Visualização

- Diferentes técnicas de visualização podem ser utilizadas para criar múltiplos cenários visuais
- Técnicas propostas inicialmente em Visualização de Informação podem ser utilizadas em Visualização de Software, provendo suporte ao processo de compreensão.

26/7/2012 31

Visualização de Software
TreeMap

- Permite representar estruturas hierárquicas por meio de subdivisão de espaço.
- A hierarquia da estrutura é mapeada em regiões retangulares restritas e aninhadas.
- Diversos dados podem ser associados à dimensão e coloração dos retângulos
  - Ex: a área pode representar a quantidade de linhas (LOC) e a coloração pode representar a complexidade estrutural de uma determinada classe

26/7/2012 32

Visualização de Software
TreeMap (SourceMiner)

- Estrutura pacote-classe-método
  - Área: Complexidade ou LOC
  - Cor: Complexidade, Funcionalidade ou LOC

26/7/2012 33

Visualização de Software
Visão Polimétrica

- Permite representar estruturas hierárquicas por meio de nós e ligações.
- Os nós representam as entidades e as ligações representam a hierarquia das entidades, formando representações em forma de árvores
- Diversos dados podem ser associados à dimensão e coloração dos nós, com a diferença de que dois dados podem ser associados à dimensão, um à largura e outro à altura

26/7/2012 34

Visualização de Software
Visão Polimétrica (SourceMiner)

- Estruturas hierárquicas
  - Círculo: Classe externa
  - Cor marrom: Classe abstrata
  - Cor azul: Classe concreta
  - Cor azul clara: Classe interna e concreta
  - Altura: LOC
  - Largura: Número de métodos

26/7/2012 35

Visualização de Software
Barras e Listras

- Permitem a visualização de unidades e respectivos atributos.
- Cada unidade é representada por uma barra, e os atributos em destaque são representados por listras dentro de cada barra envolvida

26/7/2012 36

26/7/2012

Visualização de Software
Barras e Listras (AJDT Visualizer)

- Permite verificar como os aspectos afetam as classes em um projeto

37

26/7/2012

Visualização de Software
UML 3D

- Proveniente dos diagramas de classe da UML
- Representa pacotes, classes e métodos em estruturas de três dimensões

38

26/7/2012

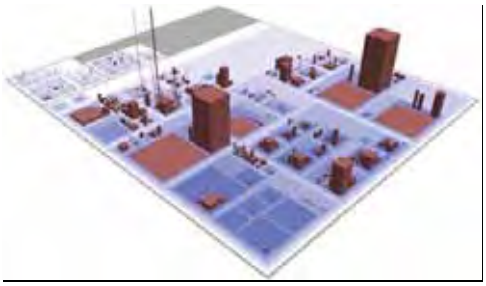
Visualização de Software
Grafo e Matriz de Dependência

- Interação entre grafo e matriz de dependência

39

26/7/2012

Visualização de Software
UML City



40

26/7/2012

Visualização de Software
UML City

41

26/7/2012

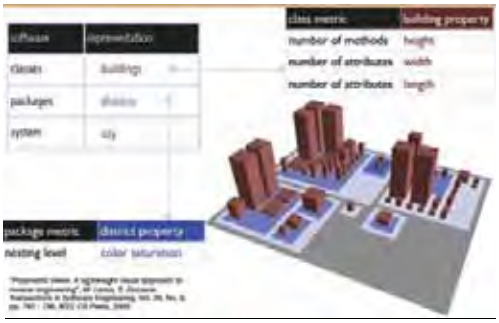
Visualização de Software
UML City

42

Visualização de Software
UML City

43

Visualização de Software
UML City

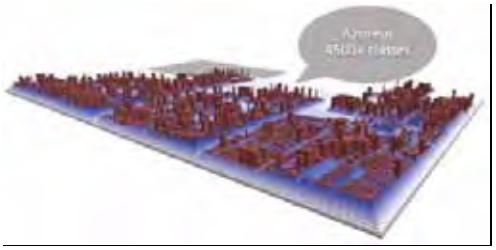


44

Visualização de Software
UML City

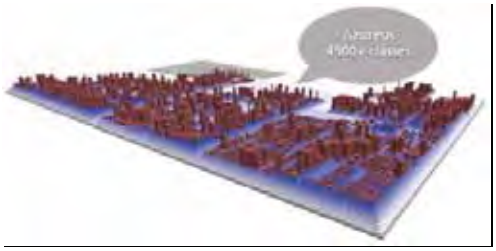
45

Visualização de Software
UML City



46

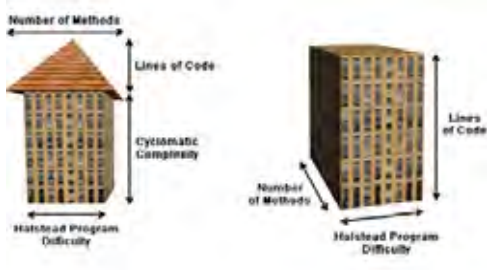
Visualização de Software
UML City



- Pacotes:
  - Saturação de cor denotando nível de aninhamento
  - Reforçado pela altitude

47

Visualização de Software
Outras cidades



48

Visualização de Software
Outras cidades

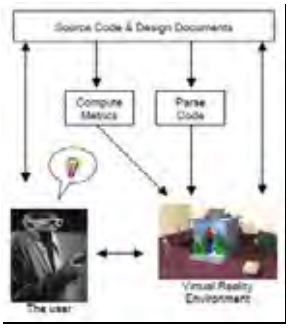
O visual das formas geométricas forma a cidade

49

Visualização de Software
Outras cidades

50

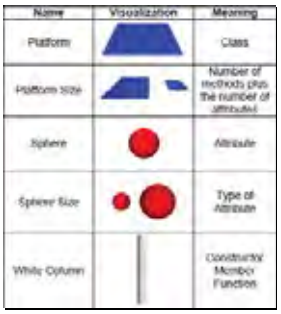
Visualização de Software
Analisando Estruturas com Realidade Virtual



- Mapeamento de dados OO para VRML
  - Melhor visualização
  - Melhor navegação

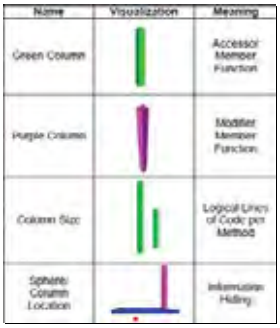
51

Visualização de Software
Analisando Estruturas com Realidade Virtual



52

Visualização de Software
Analisando Estruturas com Realidade Virtual



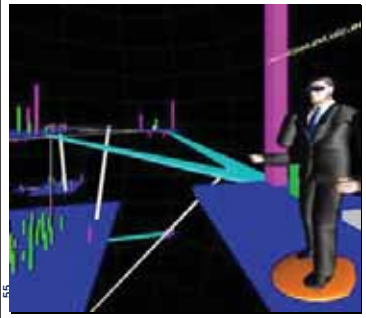
53

Visualização de Software
Analisando Estruturas com Realidade Virtual

54

26/7/2012

Visualização de Software
Analisando Estruturas com Realidade Virtual

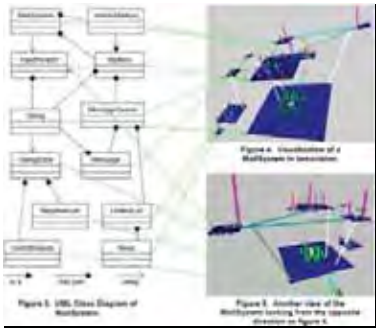


- Usuário remoto imerso em realidade virtual investigando a visualização de um sistema de software

55

26/7/2012

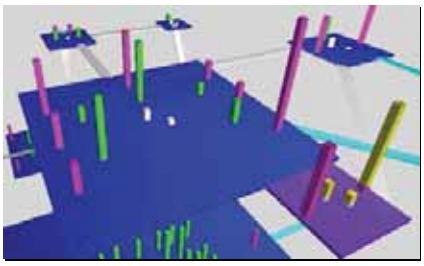
Visualização de Software
Analisando Estruturas com Realidade Virtual



56

26/7/2012

Visualização de Software
Analisando Estruturas com Realidade Virtual

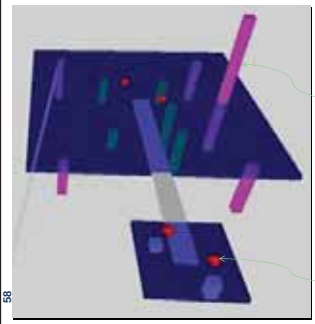


Classes MailBox e AdminMailBox

57

26/7/2012

Visualização de Software
Analisando Estruturas com Realidade Virtual



- Visão invertida das classes LinkedList e Node
  - Elementos de dados privados são vistos juntamente com uma função membro privada em LinkedList
  - Plataforma de classe semi-transparente

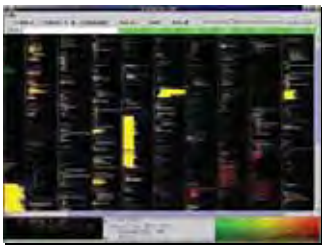
Atributos escondidos (abaixo da plataforma)

58

26/7/2012

Visualização de Software
Testes em Tempo de Execução

- Auxílio ao debug
  - Baseado em casos de testes
- Modelo de espectro de cores (declarações):
  - + vermelha, + falhas
    - Suspeitas, devem ser investigadas
  - + verdes, - falhas
    - Revelam confiança
  - amarelo: ambiguidade
    - Falharam em alguns casos e tiveram sucesso em outros



59

26/7/2012

Visualização de Software
Considerações de uma ferramenta

- Esperam-se alguns recursos essenciais:
  - Evidenciar elementos em suas representações visuais que forneçam algo significativo para ser analisado de acordo com o artefato, tarefas específicas e usuários
  - Mecanismos para o usuário navegar por todos os itens apresentados

60

26/7/2012

Visualização de Software
Considerações de uma ferramenta

- Mecanismos de interação com o usuário
  - Rotação, Escala (zoom) e Corte (crop)
    - Exploração de diferentes ângulos e perspectivas
  - Ligação (link)
    - Navegação de uma entidade a outra
      - Visa facilitar a navegação entre entidades inter-relacionadas
  - Alternância
    - Mudança entre a representação visual e a porção de código apresentada
    - Ou exibir ambas de maneira síncrona na interface

61

26/7/2012

Roteiro

Visualização de Informação
Compreensão de Programa
Visualização de Software
Considerações Finais

62

26/7/2012

Considerações Finais

- Compreensão de Programa
  - Focalizando no código fonte, a obtenção conhecimentos pode se tornar mais difícil
    - especialmente pela quantidade de linhas em sistemas de software e sua organização
- Visualização de Software
  - É importante porque a maioria dos artefatos de software são abstratos, pois um artefato de software não possui representação física
  - Abordagem alternativa no auxílio à Compreensão de Programa

63

26/7/2012

Referências

CARD, S. et al. Readings in Information Visualization. MK Publishers, Inc. 1999.
CARD, S. et al. The Structure of the Information Visualization Design Space
ROSARIO, E. et al. Mapping Nominal Values to Numbers for Effective Visualization
CHEN, C. Visualization Viewpoints. Agosto/2005
PORTO, D. CRIST: Um Apoio Computacional para Atividades de Inspeção e Compreensão de Código. UFSC. Maio/2009.
VICENZI, A. et al. Técnica de Teste Caixa Branca. Slides. Universidade Federal de Goiás. Agosto/2008
MARTINS, R. Uma Ferramenta de Teste Estrutural de Programas Orientados a Aspectos Usando AspectJ. FCT-Unesp. Presidente Prudente. Dezembro/2007
SourceMiner – UserGuide
CHAPMAN, M. AOP@Work: New AJDT releases ease AOP development. Disponível em <http://www.ibm.com/developerworks/java/library/j-aopwork9/>. Acessado em Novembro/2010.
AJDT: AspectJ Development Tools. Demonstration: The Visualizer. Disponível em <http://www.eclipse.org/ajdt/demos/#VISUALISER-DEMO>. Acessado em Novembro/2010.

64

C.6 Treinamento na SoftVisOAH

Faculdade de Ciências e Tecnologia
Departamento de Matemática, Estatística e Computação
Bacharelado em Ciência da Computação

unesp

SoftVis$_{OAH}$ Ferramenta de Visualização de Software

Álvaro d'Arce
(alvaro@darce.com.br)

26/7/2012

Tópicos

- **Ferramenta de Visualização de Software**
- Gerando o *bytecode* pelo *Eclipse*
- Considerações Finais

2

26/7/2012

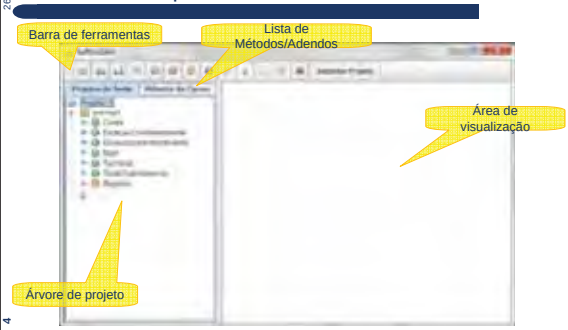
SoftVis$_{OAH}$ Introdução

- Ferramenta de Visualização
  - *SoftViz$_{OAH}$*
  - Mapeamento visual abordando Programas Orientados a Objetos e a Aspectos
    - *Java* e *AspectJ*
  - Resultados de Testes Estruturais
    - Casos de Teste em *JUnit*
  - Visualização de *bytecode*

3

26/7/2012

SoftVis$_{OAH}$ Tela Principal

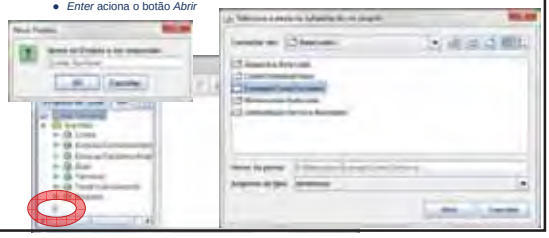


4

26/7/2012

SoftVis$_{OAH}$ Importando um Projeto

- Selecione *Importar Projeto* na barra de ferramentas
- Dê um nome ao projeto a ser importado
- Selecione a pasta contendo o bytecode do programa
  - Percorra as pastas com o mouse, sem teclar *Enter*
    - *Enter* aciona o botão *Abrir*



5

26/7/2012

SoftVis$_{OAH}$ Visualização Geral

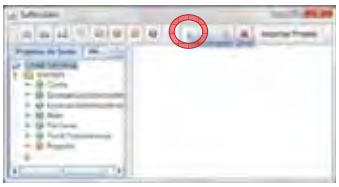
- Visualiza a estrutura geral de um programa
- Visualização Hiperbólica:
  - Visualiza classes, aspectos e suas dependências
- TreeMap:
  - Visualiza estrutura hierárquica
  - Pacotes
    - Classes
      - Métodos
    - Aspectos
      - Adendos

6

26/7/2012

SoftVis$_{OAH}$
Visualização Geral

- Selecione a raiz do projeto. Em seguida, selecione o botão *Visualização Geral* da barra de ferramentas

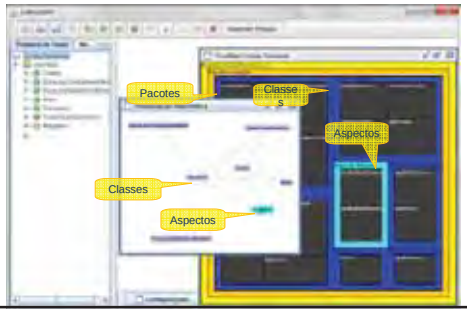


7

26/7/2012

SoftVis$_{OAH}$
Visualização Geral

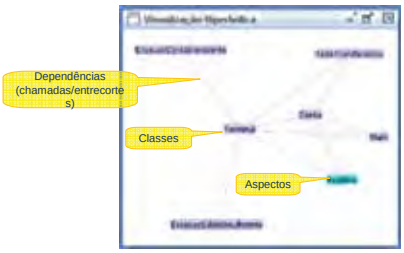
- Minimize a janela *Configurações* e arraste/redimensione as demais



8

26/7/2012

SoftVis$_{OAH}$
Visualização Geral: Hiperbólica



9

26/7/2012

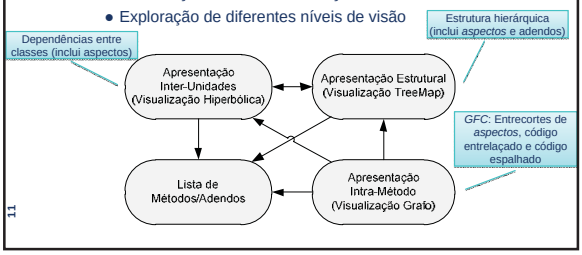
SoftVis$_{OAH}$
Visualização Geral: TreeMap

10

26/7/2012

SoftVis$_{OAH}$
Abordagem de Visualização Múltipla

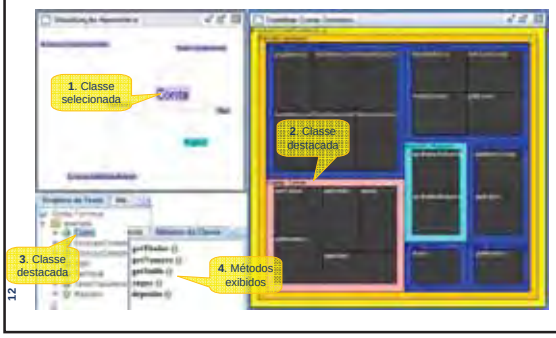
- Abordagem de mapeamento visual
  - Proposta para visualizar Programas *OO* e *OA*
  - Coordenação de 3 visualizações e 1 lista
    - Exploração de diferentes níveis de visão



11

26/7/2012

SoftVis$_{OAH}$
Coordenação: Hiperbólica - TreeMap



12

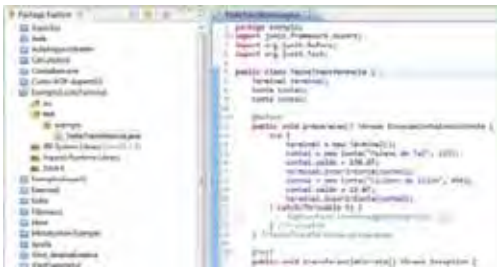
26/7/2012

SoftVis$_{OAH}$
Coordenação: TreeMap - Hiperbólica

13

26/7/2012

SoftVis$_{OAH}$
Visualização de Casos de Teste



- Os casos de teste devem ser criados em *JUnit*. No exemplo, a *IDE* utilizada foi o *Eclipse*.
  - **Caso de teste a ser visualizado, *exemplo.TesteTransferencia*.**

14

26/7/2012

SoftVis$_{OAH}$
Visualização de Casos de Teste

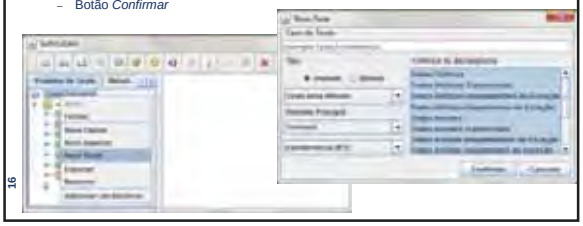
- Testes Unitários (Caixa Branca)
  - Testa o funcionamento interno do software por meio de casos de teste criados pelo usuário em *Junit*
- Para que as visualizações possam prover a devida ajuda ao entendimento do programa:
  - Casos de teste devem ser criados de maneira a **exercer chamadas** entre classes/aspectos

15

26/7/2012

SoftVis$_{OAH}$
Visualização de Casos de Teste

- Com o botão direito do mouse sobre a raiz do projeto, selecione *Novo Teste*
  - Informe o pacote e a classe do caso de teste
  - Informe a classe e o método a ser percorrido no caso de teste
  - Informe os critérios de abrangência
  - Botão *Confirmar*

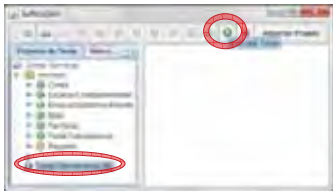


16

26/7/2012

SoftVis$_{OAH}$
Visualização de Casos de Teste

- O caso de teste adicionado aparecerá no final da árvore de projeto
- Selecione-o e em seguida selecione o botão *Executar Teste*



17

26/7/2012

SoftVis$_{OAH}$
Visualização de Casos de Teste

- Minimize a janela *Configurações* e arraste/redimensione as demais

26/7/2012

SoftVis$_{OAH}$
Janela de Resultado de Casos de Teste

- Aba *Resultado*:
  - Exibe os resultados dos critérios de abrangência e o resultado do caso de teste criado em *JUnit*

26/7/2012

SoftVis$_{OAH}$
Gradiente de Resultados de Casos de Teste

- Os elementos das visualizações de um caso de teste são coloridos de acordo com um gradiente do vermelho para o verde
  - Quanto **mais** o código representado obteve **sucesso** no caso de teste, seu elemento representativo é colorido mais para o **verde**
  - Quanto **menos** o código representado obteve **sucesso** no caso de teste, seu elemento representativo é colorido mais para o **vermelho**

26/7/2012

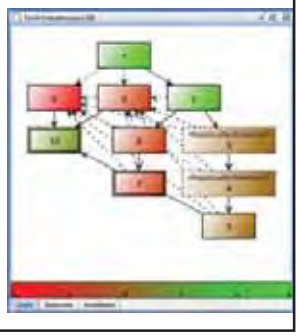
SoftVis$_{OAH}$
Elementos coloridos pelo gradiente

- **TreeMap:**
  - **Retângulos de métodos e adendos**
- **Hiperbólica:**
  - **Arestas de métodos/adendos envolvidos no caso de teste**
- **Grafo**
  - **Retângulos de sequência de código (métodos/adendos)**
- **Bytecode**
- **Lista de Métodos/Adendos**

26/7/2012

SoftVis$_{OAH}$
Janela de Resultado de Casos de Teste

- Aba *Grafo*:
  - Exibe o Grafo de Fluxo de Controle do caso de teste criado em *JUnit* (código entrelaçado, após o *weaver* do *AspectJ*)
  - Blocos contornados por linha simples representam trechos de código de métodos
  - Blocos contornados por linha tracejada representam códigos de adendos entrecortando métodos
  - Blocos contornados por linha dupla representam saídas de métodos
  - Linhas contínuas representam fluxos de controle
  - Linhas tracejadas representam fluxos de controle de exceções



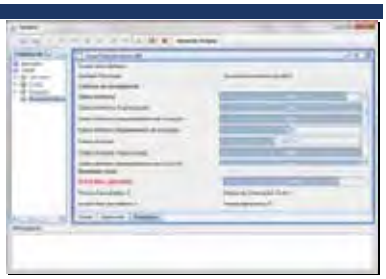
26/7/2012

SoftVis$_{OAH}$
Janela de Resultado de Casos de Teste

- Aba *Bytecode*:
  - Exibe o bytecode do caso de teste

26/7/2012

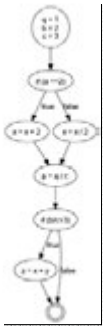
SoftVis$_{OAH}$
Teste de Caixa Branca



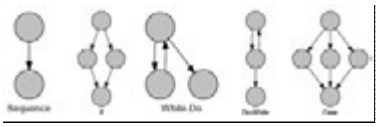
- Testa o funcionamento interno do software por meio de casos de teste do *JUnit*

24

SoftVis$_{OAH}$
Teste de Caixa Branca
Grafo de Fluxo de Controle



- Técnica utilizada no Teste da Caixa Branca
- Notação gráfica utilizada para abstrair o fluxo de controle lógico de um programa



26/7/2012
25

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Para que um teste estrutural seja bem-sucedido ele precisa alcançar alguns requisitos de cobertura na estrutura das unidades testadas, representados pelos critérios de abrangência.
- Os critérios de abrangência garantem que certos pontos importantes do código foram visitados durante a execução do teste.

26/7/2012
26

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todos-Vértices
  - "Cada vértice (bloco de comandos) deve ser executado pelo menos uma vez. Essa é a forma mais fraca de teste estrutural." [Martins]
- Todos-Vértices-Transversais
  - "Todos os vértices transversais devem ser executados pelo menos uma vez, ou seja, todas as execuções de adendos dentro de uma unidade são exercitadas." [Martins]

26/7/2012
27

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todos-Vértices-Independentes-de-Exceção
  - "Todos os vértices que são alcançáveis sem passar por uma aresta de exceção devem ser executados pelo menos uma vez." [Martins]
- Todos-Vértices-Dependentes-de-Exceção
  - "Todos os vértices que não são alcançáveis sem passar por uma aresta de exceção devem ser executados pelo menos uma vez." [Martins]

26/7/2012
28

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todas-Arestas
  - "Cada aresta (desvio de fluxo) deve ser percorrida pelo menos uma vez. Nesse caso, cada instrução condicional deve ser testada pelo menos uma vez para verdadeira e uma vez para falsa." [Martins]
- Todas-Arestas-Transversais
  - "Todas arestas transversais devem ser executadas pelo menos uma vez; ou seja, todos os desvios para um adendo dentro da unidade são executados." [Martins]

26/7/2012
29

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todas-Arestas-Independentes-de-Exceção
  - "Todas as arestas que podem ser alcançadas sem passar por uma aresta de exceção devem ser percorridas pelo menos uma vez." [Martins]
- Todas-Arestas-Dependentes-de-Exceção
  - "Todas as arestas que não podem ser alcançadas sem passar por uma aresta de exceção devem ser percorridas pelo menos uma vez." [Martins]

26/7/2012
30

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todas-Definições
  - "Para cada definição de variável no programa, deve ser percorrido o caminho até um uso qualquer da variável." [Martins]
- Todos-Usos
  - "Para cada definição devariável no programa, deve ser percorrido o caminho até cada uso da variável." [Martins]

26/7/2012 31

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todos-Usos-Transversais
  - "Para cada definição de variável no programa, deve ser percorrido o caminho até os usos dessas variáveis que estão em vértices transversais." [Martins]
- Todos-Usos-Independentes-de-Exceção
  - "Para cada definição de variável no programa, deve ser percorrido o caminho até todos os usos da variável alcançáveis sem passar por uma aresta de exceção." [Martins]

26/7/2012 32

SoftVis$_{OAH}$
Teste de Caixa Branca
Critérios de Abrangência

- Todos-Usos-Dependentes-de-Exceção
  - "Para cada definição de variável no programa, deve ser percorrido o caminho até todos os usos da variável que não são alcançáveis sem passar por uma aresta de exceção." [Martins]

26/7/2012 33

SoftVis$_{OAH}$
Exemplo
Método a ser visualizado

```
public void transfencia(int numeroOrigem, int numeroDestino , float valor) {
  try {
    account contaOrigem = getConta(numeroOrigem) ;
    account contaDestino = getConta(numeroDestino) ;
    if (contaOrigem.getBalanco() >= valor) {
      contaOrigem.saque(valor);     // MÉTODO ENTRECORTADO
      contaDestino.deposito(valor); // MÉTODO ENTRECORTADO
    } else
      throw new ExcecaoSaldoInsuficiente(contaOrigem.getNumber(), valor);
  } catch (ExcecaoContaInexistente e) {
    System.out.println(e.getMessage()) ;
  } catch (ExcecaoSaldoInsuficiente e) {
    System.out.println(e.getMessage()) ;
  }
}
```

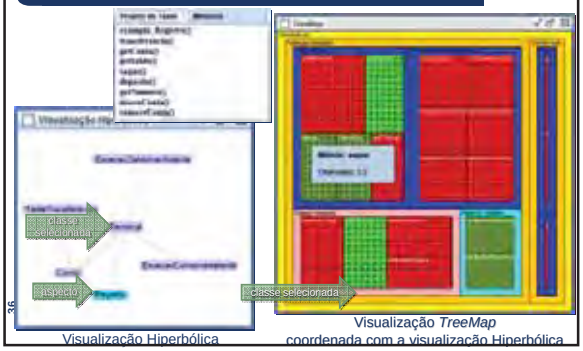
26/7/2012 34

SoftVis$_{OAH}$
Exemplo
Aspecto a ser visualizado

```
public aspect Registro {

   pointcut saque (Conta con, float valor):
      target(con) && args(valor) && call(* Conta.saque(float));

   pointcut deposito(Conta con, float valor):
      target(con) && args(valor) && call(* Conta.deposito(float));

   after(Conta Conta, float valor) returning(): saque(Conta, valor ) {
      System.out.println("Valor do saque: " + valor) ;
   }

   after(Conta Conta, float valor) returning(): deposito(Conta, valor ) {
      System.out.println("Valor do depósito: " + valor) ;
   }
}
```

26/7/2012 35

SoftVis$_{OAH}$
Exemplo
Visualização Coordenada

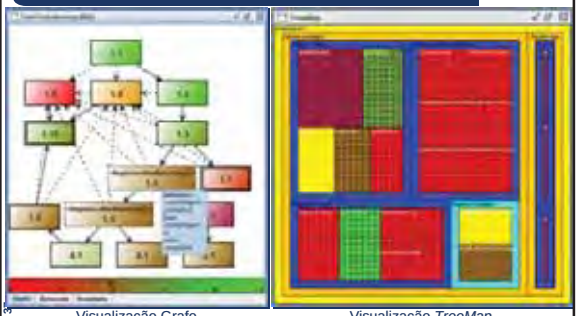


Visualização Hiperbólica

Visualização *TreeMap* coordenada com a visualização Hiperbólica

26/7/2012 36

SoftVis$_{OAH}$
Exemplo
Visualização de Caso de Teste

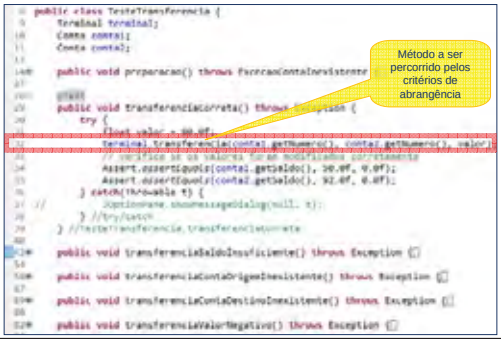


Visualização Grafo
caso de teste - método *transferencia()*

Visualização *TreeMap*
coordenada com a visualização Grafo

26/7/2012 — 37

SoftVis$_{OAH}$
Visualizando o caso de teste TesteTransferencia() sobre o método terminal.transferencia()



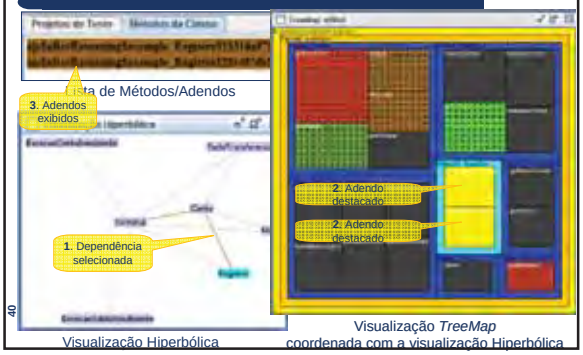
26/7/2012 — 38

SoftVis$_{OAH}$
Visualizando chamada de Métodos/Adendos entre 2 classes/aspectos em um caso de teste

- Quando visualizando um caso de teste, clicando em uma aresta na *Visualização Hiperbólica*, os métodos/adendos participantes do caso de teste são destacados na *Visualização Treemap* e exibidos na Lista de *Métodos/Adendos*
- Desta maneira, é possível visualizar os métodos/adendos envolvidos nas associações entre classes/aspectos

26/7/2012 — 39

SoftVis$_{OAH}$
Visualizando chamada de Métodos/Adendos entre 2 classes/aspectos em um caso de teste

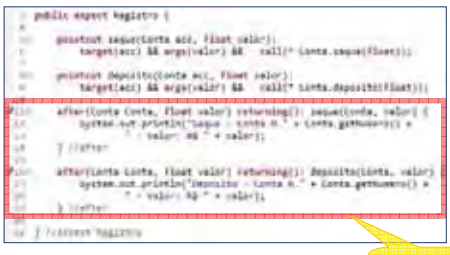


Visualização Hiperbólica

Visualização *TreeMap*
coordenada com a visualização Hiperbólica

26/7/2012 — 40

SoftVis$_{OAH}$
Adendos destacados na visualização coordenada



26/7/2012 — 41

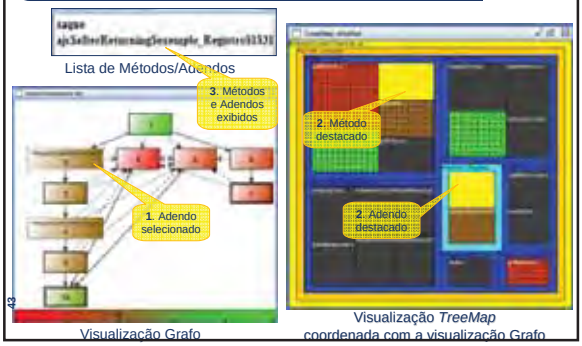
SoftVis$_{OAH}$
Visualizando métodos adendados no caso de teste

- Quando visualizando um caso de teste, clicando em um adendo *Visualização Grafo*, o respectivo adendo e os métodos entrecortados são destacados na *Visualização Treemap* e exibidos na Lista de *Métodos/Adendos*
- Desta maneira, é possível visualizar métodos entrecortados por adendos no caso de teste

26/7/2012 — 42

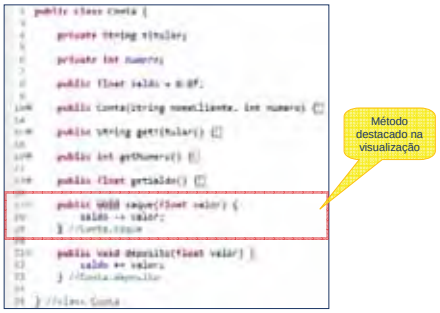
SoftVis$_{OAH}$

Visualizando chamada de Métodos/Adendos entre 2 classes/aspectos



SoftVis$_{OAH}$

Métodos entrecortados destacados na visualização coordenada



Dúvidas?

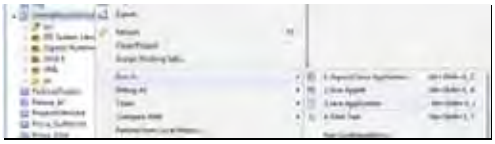
Tópicos

- Ferramenta de Visualização de Software
- **Gerando o *bytecode* pelo *Eclipse***
- Considerações Finais

Eclipse

Gerando o bytecode para a Ferramenta

Executando a aplicação

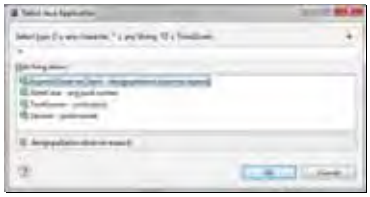


- É necessário executar ao menos uma vez a ferramenta para gerar as configurações para a exportação do *.jar*
- Botão direito no projeto, opção *Run As*, opção *Java Application*

Eclipse

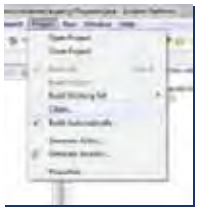
Gerando o bytecode para a Ferramenta

Executando a aplicação



- Selecione a classe inicial da aplicação (contendo o método *main*)
- Botão *Ok*
- Encerre a aplicação executada (se for o caso)
- As configurações de exportação do *.jar* estão agora gravadas

Eclipse
Gerando o bytecode para a Ferramenta
Exportando as classes

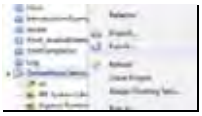


- Menu Project
  - Certifique-se de que *Build Automatically* está marcado
  - Opção *Clean...*

Eclipse
Gerando o bytecode para a Ferramenta
Exportando as classes

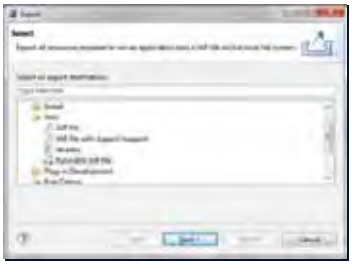
- Opção *Clean projects selected below*
- Marque o projeto atual
- *Ok*
- Os arquivos *.class* foram completamente reconstruidos

Eclipse
Gerando o bytecode para a Ferramenta
Exportando o *jar*



- Botão direito sobre o projeto
- Opção *Export...*

Eclipse
Gerando o bytecode para a Ferramenta
Exportando o *jar*

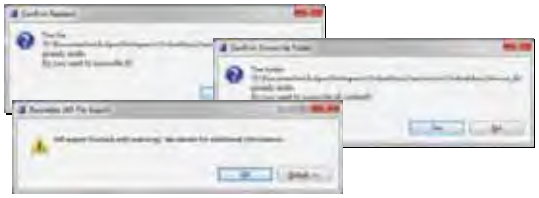


- Grupo *Java*, item *Runnable JAR file*
- Botão *Next*

Eclipse
Gerando o bytecode para a Ferramenta
Exportando o *jar*

- Em *Launch configuration*, selecione o projeto atual
  - Configurações geradas pela execução da aplicação
- Em *Export destination*, informe o nome do arquivo *.jar* a ser criado
  - Informe o nome do projeto para o arquivo *.jar*, dentro da subpasta *bin* do projeto
- Em *Library handling*, selecione *Copy requered libraries into...*
- Botão *Finish*

Eclipse
Gerando o bytecode para a Ferramenta
Exportando o *jar*



- Se for o caso, confirme as sobreposições de arquivos e pastas
- Ao final, botão *Ok*
- A subpasta *bin* conterá o bytecode pronto para ser lido pela ferramenta
- Obs: Ao executar ou limpar a aplicação, o *.jar* deve ser exportado novamente.

Tópicos

- Ferramenta de Visualização de Software
- Gerando o *bytecode* pelo *Eclipse*
- **Considerações Finais**

26/7/2012 55

Considerações Finais

- Mapeamento Visual
  - Externalizar características da *POA*
    - Aspectos e seu espalhamento pelo código fonte
  - Visualizações Coordenadas
    - Organização estrutural
    - Relações entre classes e aspectos
    - Código entrecortado
- Ferramenta de Visualização
  - Casos de teste ao longo das representações visuais
    - Permitem ao usuário perceber como os casos de teste percorrem o código fonte
      - Provendo auxílio ao processo de entendimento

26/7/2012 56

Dúvidas?

26/7/2012 57