Aspect Oriented Programming (AOP)

Uma visão geral da programação orientada a aspectos. Usando AspectJ

Objetivos

- O objetivo dessa apresentação é proporcionar uma visão geral sobre a programação orientada a aspecto.
- O que é e como funciona a AOP
- O que é e como funciona AspectJ

O que é AOP?

- **A AOP é uma alternativa para resolver problemas que nem as técnicas de programação Orientada à Objetos e nem as técnicas de programação Estruturada resolvem facilmente.**

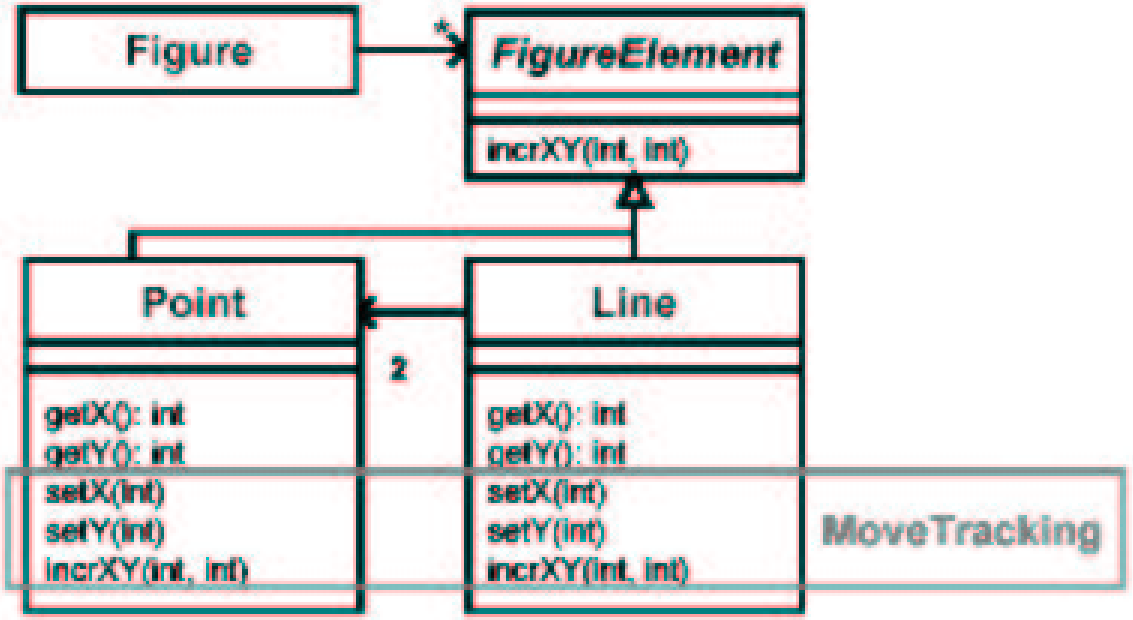
Como funciona a AOP?

- **A AOP se baseia no conceito de aspectos, para entender a AOP devemos entender o que é Aspectos.**

O que são Aspectos?

- **Aspectos são características que atravessam as funcionalidades básicas do sistema, ou seja, ele não é uma característica apenas de um componente.**
- **Dissemos que aspectos fatoram as características em comum entre os vários componentes existentes no sistema.**

Exemplo de um Aspecto



- **O campo move tracking representa um Aspecto.**

Crosscutting

- A AOP permite ao programador separar os interesses comuns do sistema (que fogem a típica divisão de classes).
- Esses comportamentos não se encaixam naturalmente dentro de um modulo de um programa.

Crosscutting

- Esse tipo de comportamento é conhecido como um comportamento que crosscutting (atravessa) o sistema
- Na OOP a unidade natural de modularização é a classe, e um comportamento do tipo crosscutting está espalhado em varias classes

Crosscutting

- Trabalhar com código que apontam para responsabilidades que atravessam o sistema gera problemas que resultam na falta de modularidade.
- Nesse ponto a AOP ajuda a manter a consistência do projeto, apresentando um novo nível de modularidade. Os aspectos.

AOP e OOP

- A AOP complementa a OOP por facilitar um outro tipo de modularidade que expande a implementação espalhada de uma responsabilidade dentro de uma simples unidade.

Como funciona a AOP?

- **Uma implementação básica de AOP, consiste em:**
  - Uma linguagem de componentes
  - Uma ou mais linguagens de Aspecto
  - Um Weaver
  - Um programa de componentes
  - Um ou mais programa de aspectos
  - Um compilador

Por que usar a AOP?

- A AOP foi desenvolvida para que os problemas de entrelaçamento e repetição de código gerado por ter que implementar varias vezes o mesmo trecho de código fosse reduzida para uma única unidade, chamada aspecto.

Por que usar a AOP?

- Por que diminui a complexidade dos componentes, visto que uma parte do código fica na definição dos aspectos.
- Por estar centralizado em uma única unidade, alterações são muito mais simples, não é preciso reescrever inúmeras classes.

Por que usar a AOP?

- Com a diminuição do tamanho do código dos componentes a complexidade é diminuída.
- Por ser uma forma melhor de gerenciar a complexidade dos componentes.
- Por ter menos código e ser menos complexo, está sujeito a menos erros.

Então por que todo mundo não usa a AOP?

- **Se é tão bom quanto parece por que não começamos a usar a AOP agora mesmo?**
- **Mesmo tendo evoluído muito a AOP anda deixa a desejar em alguns pontos:**
  - Como definimos o que é ou não um aspecto no nosso projeto???
  - Existem metodologias para definir isso???

AspectJ

- AspectJ é uma linguagem de aspecto, ela é uma extensão da linguagem Java
- Conta com um montador (Weaver) para unir o programa de componente com o programa de aspecto

AspectJ em exemplos

- Vejamos agora um exemplo da programação orientada a aspecto utilizando aspectJ.
- O exemplo foi retirado do framework “Cactus” que simplifica testes de componentes server-side Java.

AspectJ em exemplos

- Para facilitar no “debugging” foi pedido que seus desenvolvedores incluíssem duas chamadas de funções pré-determinadas.
- Na versão 1.2 do Cactus, as funções foram inseridas sem utilizar AspectJ, o que fez com que um método típico ficasse parecido o trecho de código a seguir:

AspectJ em exemplos

- **Chamadas de Log manualmente inseridas em cada método.**

```
public void doGet(JspImplicitObjects theObjects) throws
ServletException
{
  logger.entry("doGet(...)");

  JspTestController controller = new JspTestController();
  controller.handleRequest(theObjects);

  logger.exit("doGet");
}
```

AspectJ em exemplos

- **Para implementar isso uma varredura manual do código teve que ser feita, nessa varredura foram encontradas 80 situações em 15 classes diferentes onde o trecho de código deveria ser implementado manualmente.**

AspectJ em exemplos

- **Para fugir desse trabalho que dificulta a manutenção e a compreensão do código, esse estilo de programação foi substituído por um único aspecto que automaticamente age sobre todos os métodos desejados.**
- **A seguir vemos o código de aspecto:**

AspectJ em exemplos

```
public aspect AutoLog{
  pointcut publicMethods() : execution(public *
     org.apache.cactus..*(..));
  pointcut logObjectCalls() :
    execution(* Logger.*(..));
  pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
   before() : loggableCalls(){
    Logger.entry(thisJoinPoint.getSignature().toString());
  }
   after() : loggableCalls(){
    Logger.exit(thisJoinPoint.getSignature().toString());
  }
}
```

Joinpoints e Pointcuts

- **Joinpoints:** Representam pontos bem definidos em uma execução de um programa. Ex:chamadas de métodos
- **Pointcuts:** É uma construção de linguagem que junta um conjunto de Join Points baseando-se em um critério pré-definido.

Advice

- Advice é o trecho de código que é executado antes (before();), depois (after();) ou simultaneamente (around();) a um Joinpoint.
- É algo como: "Rode este código antes de todos os métodos que eu quero escrever um log."

Advice

- **Código que descreve o advice:**

```
before() : loggableCalls(){
  Logger.entry(thisJoinPoint.getSignature().toString());
}
```

- **O dispositivo utiliza a classe Logger, o metodo entry está implementado da seguinte forma:**

```
public static void entry(String message){
  System.out.println("entering method " + message);
}
```

AspectJ em exemplos

- **O resultado do aspecto “AutoLog” será:**

```
entering method: void test.Logging.main(String[])
entering method: void test.Logging.foo()
exiting method: void test.Logging.foo()
exiting method: void test.Logging.main(String[])
```

AspectJ

- A versão atual do aspectJ não permite ainda a geração de código utilizando com entradas o programa de aspecto e o programa de componente em bytecodes.

Conclusão

- **A programação orientada a aspecto cada dia que passa deixa de ser uma grande promessa para fazer parte do dia-a-dia dos desenvolvedores, ainda que existam detalhes que precisam ser melhor estudado – como a falta de uma metodologia.**

Créditos

Universidade Federal de Santa Catarina

Sistemas de Informação

Programação Orientada a Objetos II

Autores:

Fernanda Vieira

Heitor Fontana

Marcelo Braga