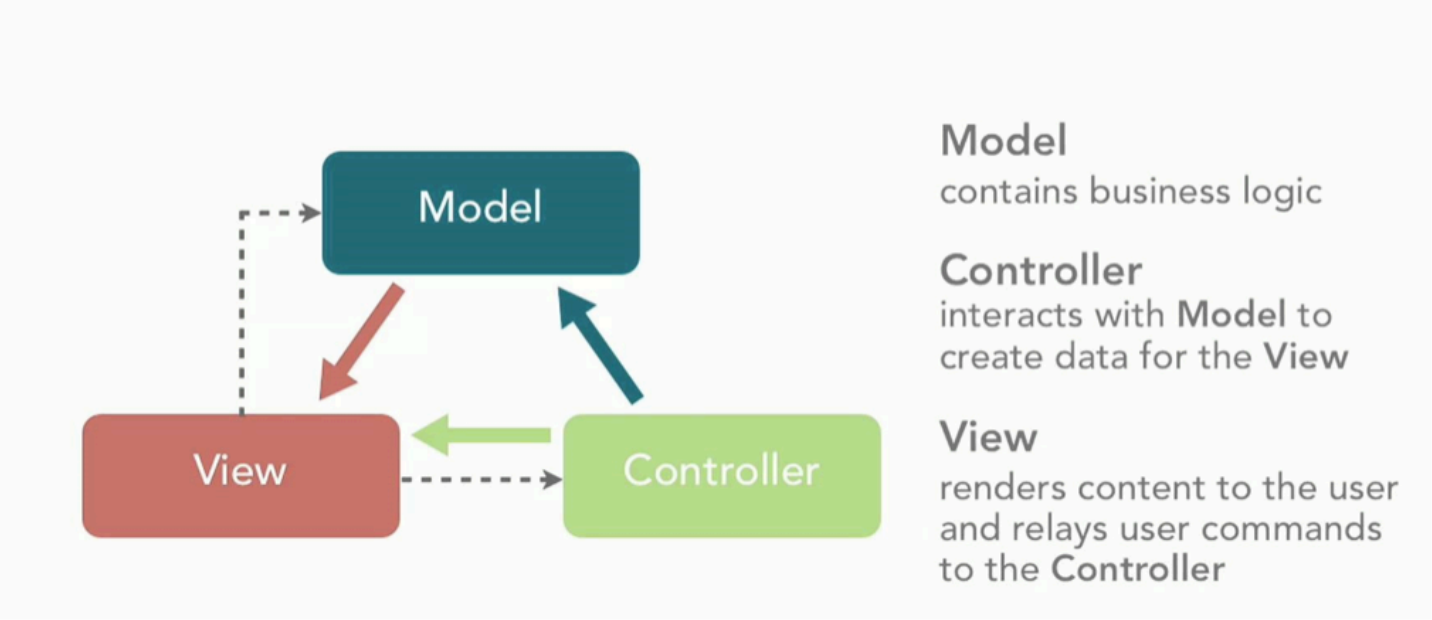
Ainda sobre MVC…



Ainda sobre MVC...

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 1-8.

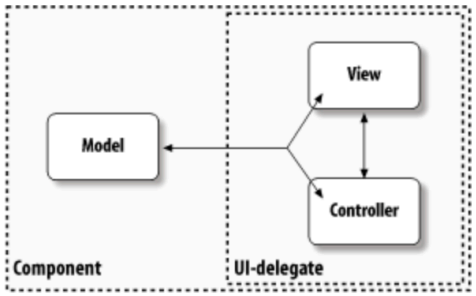


Figure 1-8. With Swing, the view and the controller are combined into a UI-delegate object

Princípios/regras

- Neste curso de Programação Orientada aos Objectos foram apresentados alguns princípios e regras relativas à construção de programas
- Elencam-se de seguida alguns dos mais importantes no sentido de fazer uma resenha dos mesmos
  - utiliza-se a abordagem apresentada em Java Program Design (ver bibliografia)

The single responsability rule

- Esta regra diz que cada classe deve ter apenas um propósito e todos os métodos estão relacionados com esse propósito
  - implica que cada classe é um contexto próprio e contido
  - quando codificamos estamos no contexto de uma entidade: um Aluno, um Ponto, uma Encomenda, etc.

The rule of encapsulation

- A implementação de uma classe deve ser opaca (logo encapsulada) para os outros componentes que a utilizam
  - apenas a classe é que sabe quais são as suas estruturas e outras variáveis de instância e como é feita a gestão das mesmas
  - os clientes de uma classe apenas conhecem a sua API

- o acesso aos valores das variáveis de instância deve ser feito através dos métodos disponibilizados

- devemos ter métodos de acesso públicos em vez de variáveis públicas

- a partilha do apontador das estruturas de dados internas constitui uma quebra de encapsulamento evidente

The most qualified class rule

- Para um determinado tema existe uma classe que é a mais qualificada para ficar com a tarefa
  - implica saber como se divide a aplicação em classes e efectuar aí as operações necessárias.
  - exemplo: o cálculo do valor de uma Encomenda deverá ser feito na classe Encomenda e não calculado externamente.

- … se quisermos mudar as regras de cálculo de uma Encomenda, e.g. introduzindo um qualquer tipo de desconto, essa lógica deverá estar contida na própria classe.

- é sempre possível escolher o contexto, a entidade, em que uma determinada operação faz mais sentido e implementá-la aí.

The rule of low coupling

- É importante reduzir o mais possível o número de dependências de uma classe
  - o objectivo é aumentar a modularidade da solução que se está a construir
  - a diminuição de dependências torna a manutenção (evolutiva e correctiva) mais simples
    - também tem vantagens na fase inicial de programação

The rule of transparency

- um cliente (uma aplicação, uma classe) deve poder utilizar uma interface (uma API) sem a necessidade de conhecer as classes que a implementam
  - situação conhecida na utilização que fizemos de List, Set, Map, Comparator, etc.
  - os métodos devem aceitar e devolver sempre que possível tipos genéricos

- é relevante para as tarefas de manutenção do código. A implementação dos métodos pode mudar e assim quem os invoca não é afectado.

- a utilização de interfaces, via o conhecimento da API, deverá ser o suficiente para que se possa utilizar uma determinada classe. Mais do que isso, é tendencialmente não respeitar a regra do encapsulamento.

The open/closed rule

- na medida do possível um programa deve estar aberto (open) a mecanismos de extensão mas fechado (closed) a mecanismos de modificação.
  - a introdução de novos conceitos é habitualmente feita através dos mecanismos de hierarquia e herança
  - o programador pode acrescentar novos conceitos adicionando informação…

- … e reescrevendo as definições herdadas, mas não deve (por vezes nem tem como o fazer, porque o código não está disponível) alterar o código das classes existentes

  - pode ter impacto noutros programas que utilizam a classe que está agora a ser alterada.

  - pode ser feito tirando partido da compatibilidade de tipos das subclasses para poder acrescentar novos conceitos

- … no projecto do Fitness introduzir novos tipos de Actividade (novas subclasses) não vai ter impacto na classe Fitness (vimos isso nas aulas PL da Fase 2…)

- … no projecto do TP introduzir novos tipos de smart devices não deverá alterar o programa.

The Liskov Substitution principle

- se Y especializa X, então quando se espera ter um objecto do tipo X também é possível utilizar um objecto do tipo Y
- este princípio está directamente relacionado com a regra apresentada anteriormente.
  - a adição de novos conceitos, via subclasses, não afecta as classes já existentes e que não conhecem este novo tipo

- … se uma Turma agrega objectos do tipo Aluno então poderá receber instâncias de Aluno como de qualquer uma das suas subclasses (e.g. AlunoTE, AlunoAtleta, etc.)

- a introdução de um tipo de Aluno, seja AlunoErasmus, não afecta o comportamento do programa, porque existe compatibilidade de tipos

  - um AlunoErasmus é um Aluno

The rule of abstraction

- As dependências de uma classe devem ser o mais genéricas possível
  - é sempre mais seguro depender de uma interface (tipo de dados) do que de uma classe
  - ao não se comprometer com o tipo de dados das classes o programa pode evoluir de forma mais controlada

The “don’t repeat yourself” rule

- um pedaço de código (e.g. método) deve existir apenas num único sítio
  - não é uma regra exclusiva da programação por objectos, mas de organização geral de um programa
  - relacionada com a regra que diz que existe sempre a classe mais apropriada para conter determinado método (ver “the most qualified class rule”)

- … muitas vezes para resolver este tipo de situação recorre-se à criação de classes abstractas

  - um sítio onde colocar código que várias classes da mesma hierarquia definem

The model-view-controller rule

- um programa deve ser concebido por forma a que o seu model, view e controller sejam (sempre que possível) classes distintas
  - providenciando independência entre as várias classes e permitindo manutenção diferenciada (muito mais evidente na view)

Programação Orientada aos Objectos

-- fecho do semestre --

Resultados de aprendizagem

- Compreender os conceitos fundamentais da PPO(Objectos, Classes, Herança e Polimorfismo);
- Compreender como os conceitos básicos da PPO são implementados em construções JAVA;
- Compreender princípios e técnicas a empregar em programação de larga escala;
- Desenvolver o modelo de classes e interfaces para um dado problema de software (modelação);
- Desenvolver e implementar aplicações Java de média escala, seguras, robustas e extensíveis;

Programa: teórica

.Paradigma da programação Orientada aos Objectos:

- Abstracção de Dados, Encapsulamento e Modularidade.
- Objectos: estrutura e comportamento.
- Mensagens.
- Classes, hierarquia e herança.
- Herança versus Composição.
- Classes abstractas.
- O princípio da substituição.
- Dynamic binding.
- Polimorfismo.
- Model-View-Controller
- Padrões arquitecturais mais comuns

Programa prática

- JAVA: Plataforma J2SE: JDK, JVM e byte-code.
- Construções básicas: tipos primitivos e operadores. Estruturas de controlo. I/O básico. Arrays.
- Nível dos objectos: Classes e instâncias. Construtores. Métodos e variáveis de instância. Modificadores de acesso. Métodos e variáveis de classe.
- Colecções genéricas. Interfaces parametrizadas. Iteradores internos e externos. Tipos List, Map e Set.
- Hierarquia de classes e herança. Overloading e overriding de métodos. Classes Abstractas. Interfaces e tipos definidos pelo utilizador. Tipo estático e dinâmico. Procura dinâmica de métodos. Polimorfismo e extensibilidade.
- Streams de input/output: de caracteres, de bytes e de objectos.
- Excepções.