Engenharia de Software

Arquitectura de Software
Padrões de Arquitectura de Software

Luís Morgado

Instituto Superior de Engenharia de Lisboa
Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

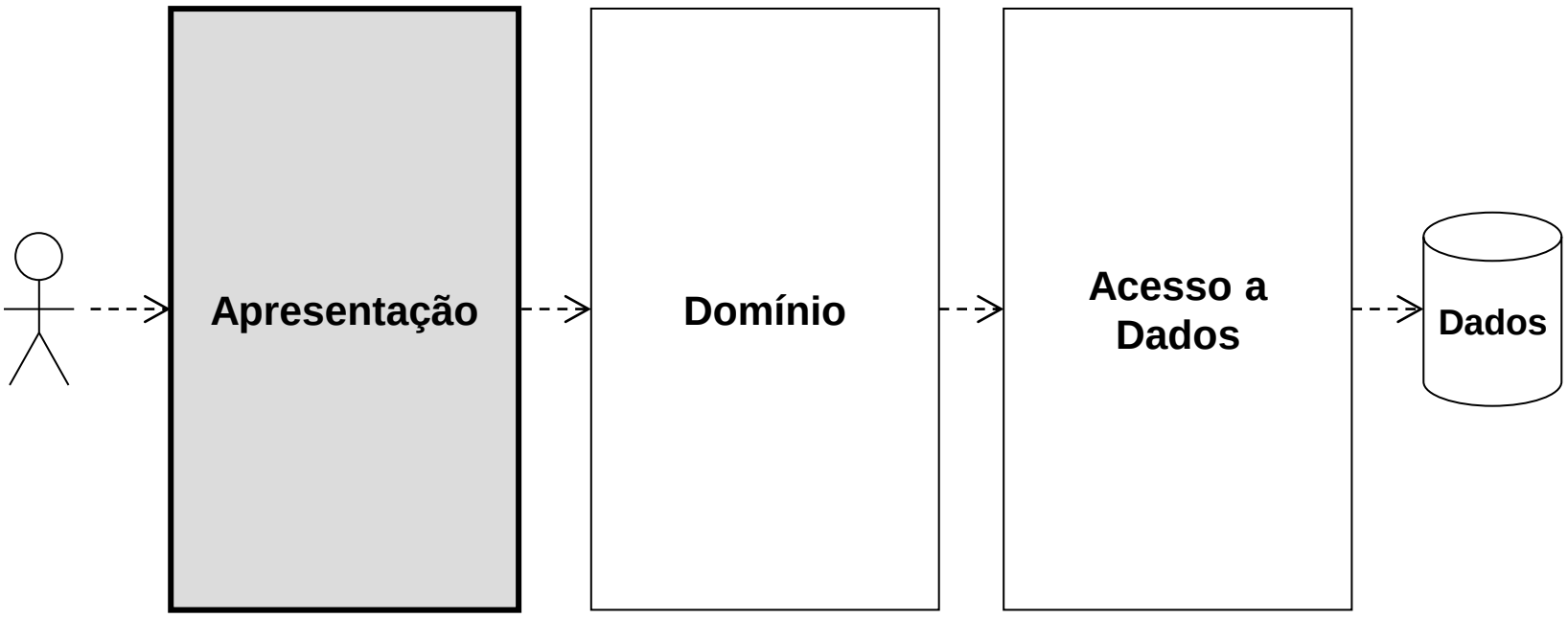
Arquitectura de 3 Camadas

Camada de Apresentação

A camada de apresentação, também designada camada de interface com o utilizador, é responsável por definir o aspeto geral da aplicação e a sua apresentação aos utilizadores, bem como por gerir a interacção com os utilizadores

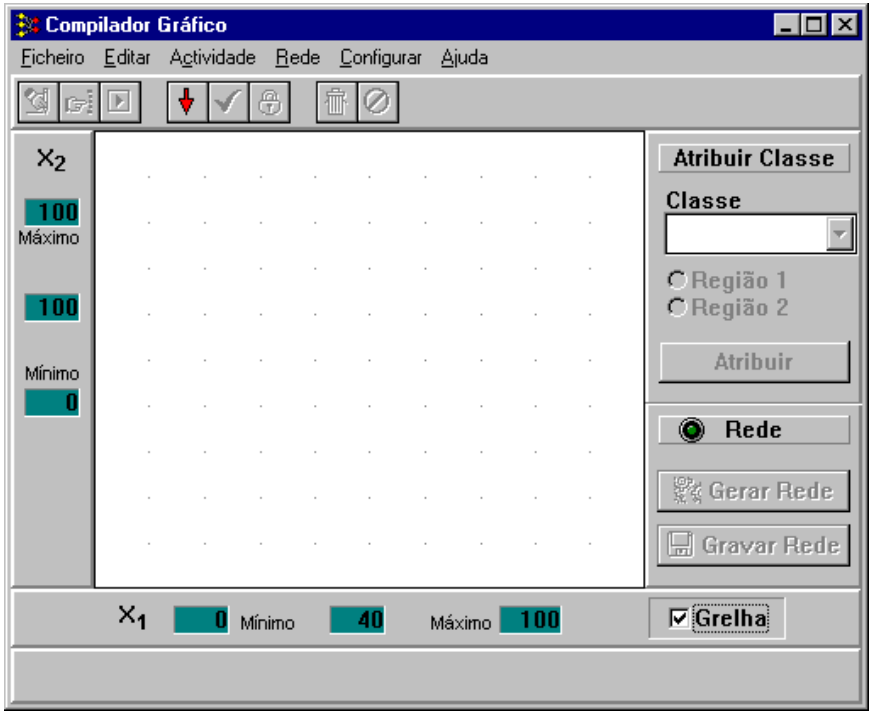
É a primeira camada aplicacional, podendo ser acedida através de diferentes tipos de dispositivos cliente, realizando a ligação com a camada de domínio

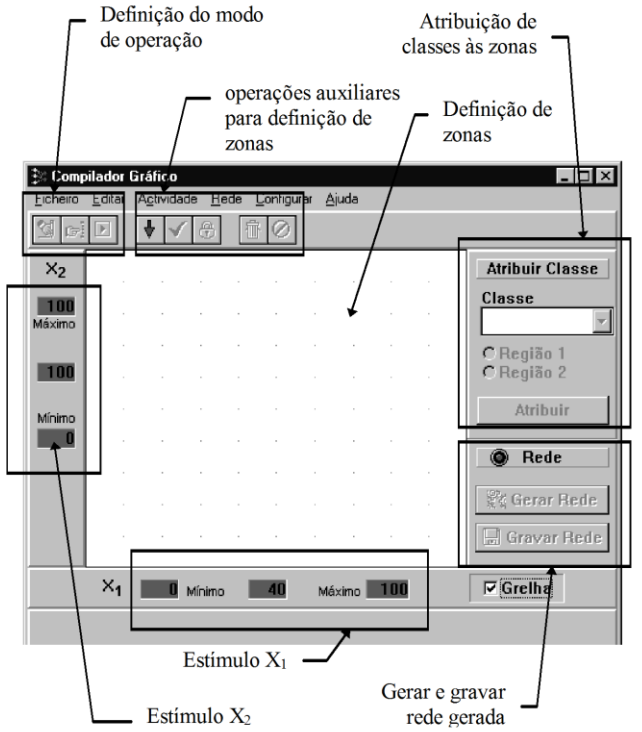
Não deve conter qualquer lógica que não seja relativa à interacção com o utilizador



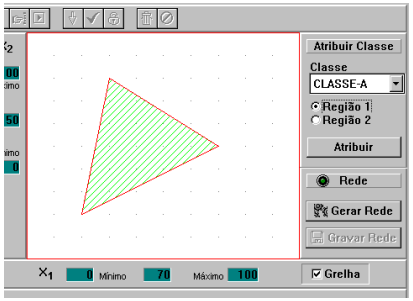
Camada de Apresentação

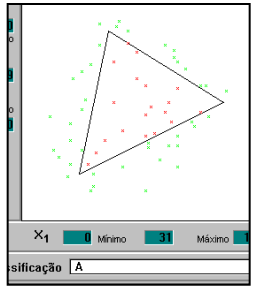
Exemplo de interface de interacção com o utilizador
(Compilador gráfico de redes neuronais)





Como separar as responsabilidades relativas à lógica de apresentação da lógica de domínio?





Camada de Apresentação

Principais padrões de mecanismos utilizados

- Padrão *Model-View-Controller* (MVC)
  - Define a organização de uma aplicação em três tipos de partes: modelo (*Model*), vista (*View*) e controlador (*Controller*), o modelo é responsável pela lógica do domínio e pela persistência de dados, a vista é responsável pela apresentação de dados, o controlador é responsável por receber informação do utilizador e coordenar a actualização do modelo e da vista
- Padrão *Model-View-Presentar* (MVP)
  - Variação do padrão MVC que tem como objectivo separar a camada de apresentação das camadas de domínio e de acesso a dados, em que a vista é responsável pela interacção com o utilizador e um apresentador substitui o controlador, actuando como mediador entre vista e modelo
- Padrão Model-View-ViewModel (MVVM)
  - Variação do padrão MVP que tem como objectivo possibilitar uma ligação automática bidirecional (*Data Binding*) entre a vista e um adaptador vista-modelo (*ViewModel*), sem a necessidade de escrever código específico para o efeito, permitindo que as alterações feitas na vista sejam refletidas de forma automática no adaptador vista-modelo (*ViewModel*) e vice-versa

Padrão *Model-View-Controller* (MVC)

Problema

As aplicações informáticas implicam tipicamente quer a interacção com o utilizador, quer o processamento de dados internos em função dessa interacção, como organizar a relação entre as classes responsáveis pela interacção com o utilizador e as classes responsáveis pelo processamento interno?

Solução

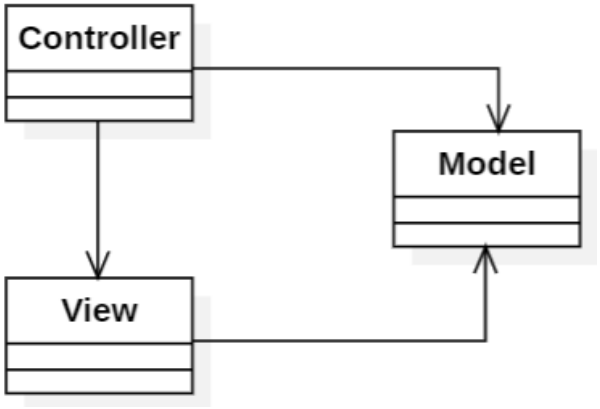
Separar o modelo de domínio, o controlo de interacção e a apresentação em três classes distintas [Burbeck,1992]

Consequências

- Separação de responsabilidades, a qual facilita a manutenção e reutilização
- Aumento da coesão, pois cada parte tem uma responsabilidade específica bem definida
- Redução do acoplamento entre camadas, permitindo que cada parte seja modificada independentemente.

Padrão *Model-View-Controller* (MVC)

Estrutura

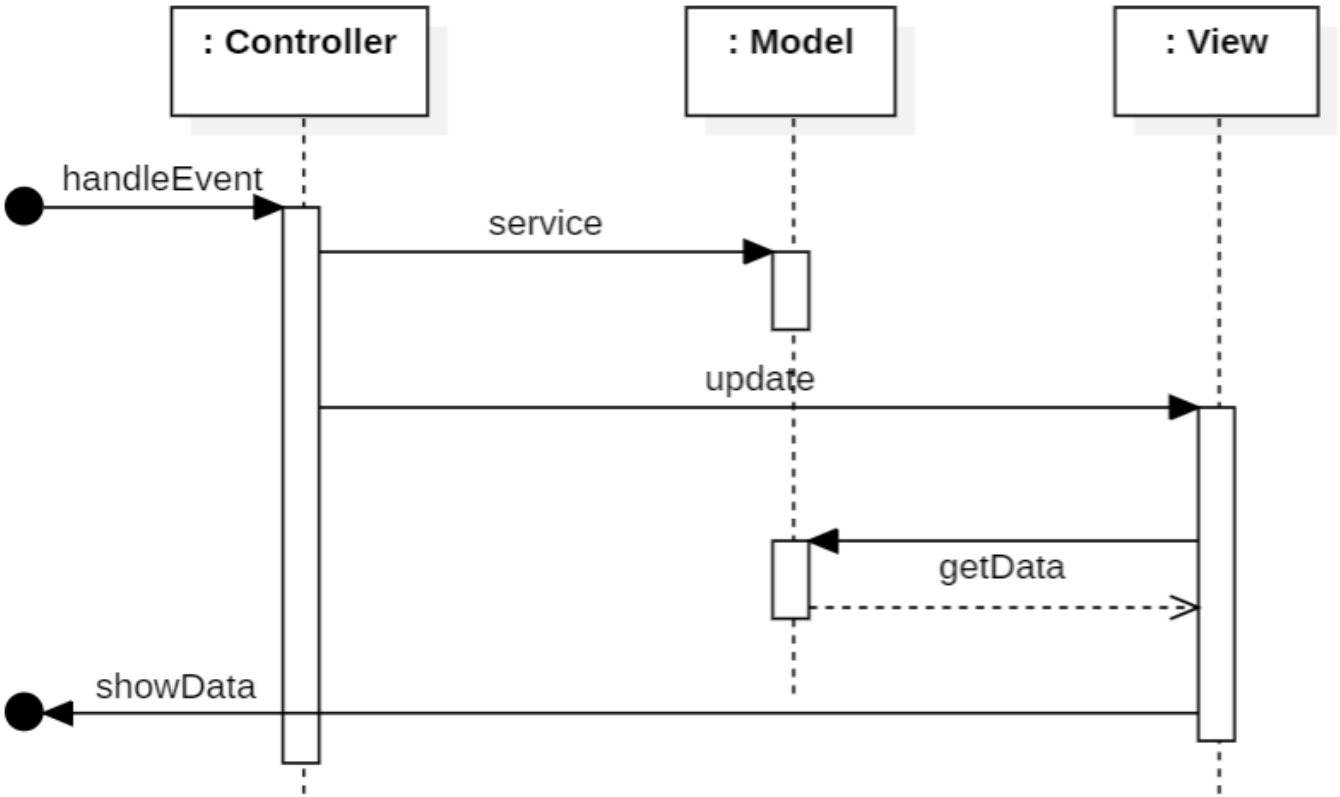


Participantes

- Modelo (*Model*)
  - Responsável pela gestão do comportamento e dos dados do domínio da aplicação, responde a solicitações de informações sobre seu estado (geralmente da vista) e responde a instruções para alterar o estado (geralmente do controlador)
- Vista (*View*)
  - Responsável pela apresentação de informação ao utilizador
- Controlador (*Controller*)
  - Responsável pela interpretação das entradas provenientes do utilizador, informando o modelo e/ou a vista para que realizem as alterações correspondentes

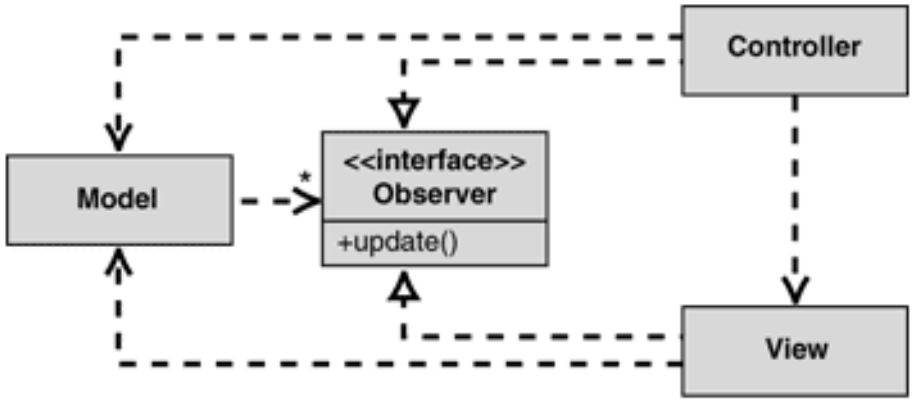
Padrão *Model-View-Controller* (MVC)

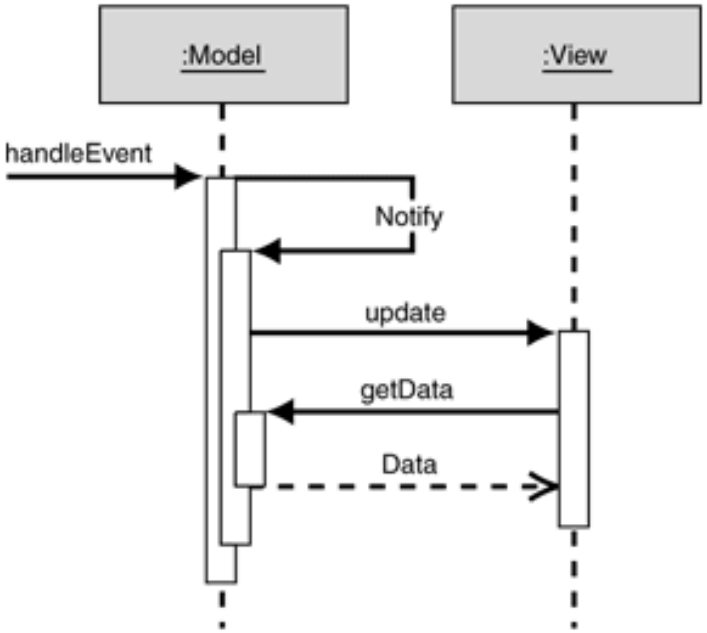
Comportamento



MVC com Modelo Activo

O *modelo* muda de estado de forma independente do controlador





Integração com o padrão ***Observer***

Evita que o *Modelo* dependa das *Vistas*

.NET Architecture Center (MSDN)
Enterprise Solution Patterns Using Microsoft .NET

Padrão *Model-View-Presenter* (MVP)

Problema

Como organizar uma aplicação com interacção com o utilizador, reduzindo o acoplamento entre a lógica de apresentação, a lógica de domínio e a lógica de acesso e persistência de dados?

Solução

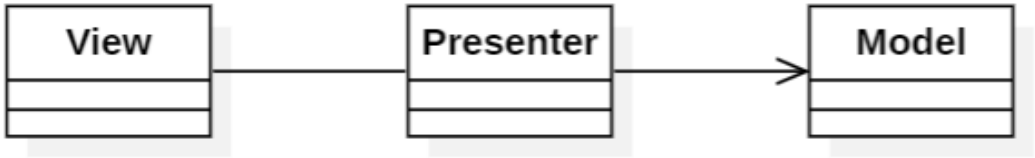
Organizar a aplicação em três componentes principais, modelo, vista e apresentador, em que o apresentador é responsável pela mediação entre a vista e o modelo

Consequências

- Separação de responsabilidades, a qual facilita a manutenção e reutilização
- Aumento da coesão, pois cada parte tem uma responsabilidade específica bem definida
- Redução do acoplamento entre camadas, permitindo que cada parte seja modificada independentemente, em particular, redução do acoplamento entre a lógica de apresentação e a lógica de acesso e persistência de dados

Padrão *Model-View-Presenter* (MVP)

Estrutura

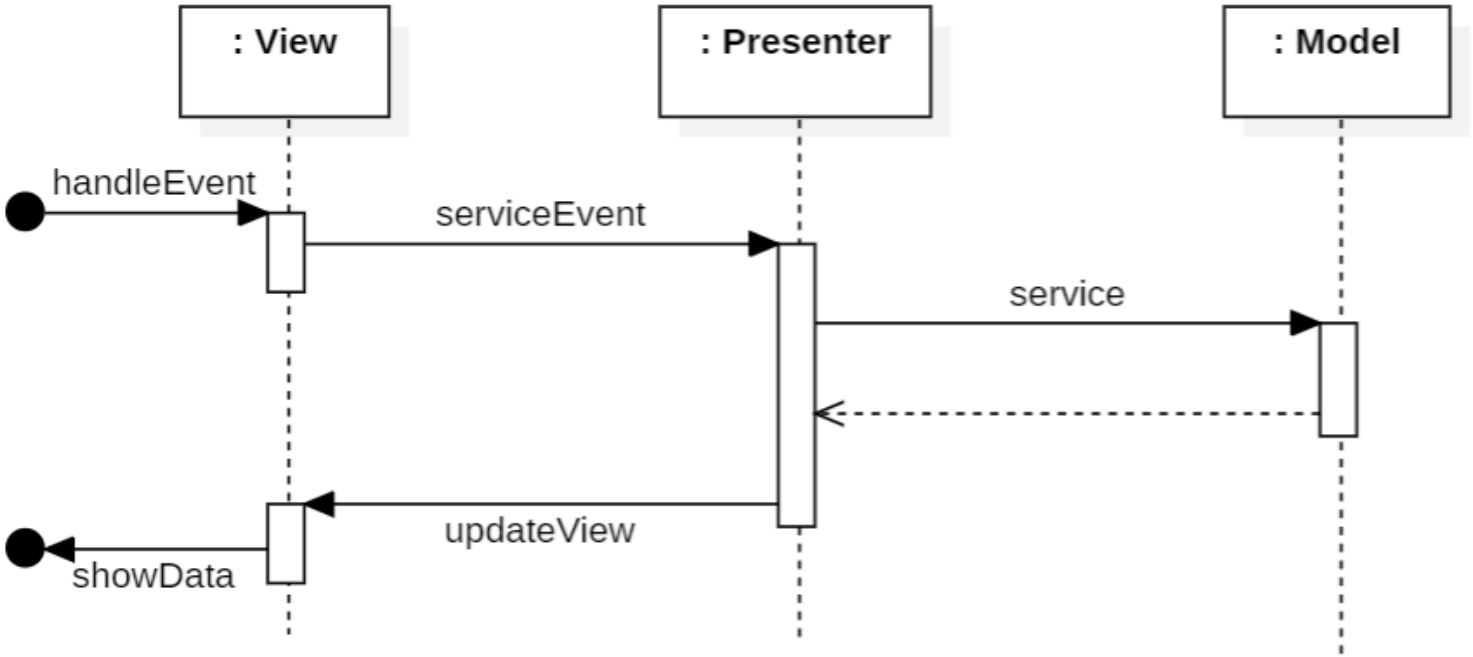


Participantes

- Modelo (*Model*)
  - Responsável pela gestão do comportamento e dos dados do domínio da aplicação, responde a solicitações de informações sobre seu estado e responde a instruções para alterar o estado
- Vista (*View*)
  - Responsável pela interacção com o utilizador
- Apresentador (*Presenter*)
  - Responsável pela mediação entre a vista e o modelo

Padrão *Model-View-Presenter* (MVP)

Comportamento



Padrões MVC e MVP

Os padrões de apresentação MVC (*Model-View-Controller*) e MVP (*Model-View-Presenter*) são utilizados para separar as responsabilidades relativas à lógica de apresentação da lógica de domínio, apresentando vantagens e desvantagens relativas

Padrão MVC

- Vantagens:
  - É um padrão bem conhecido, documentado e amplamente utilizado
- Desvantagens:
  - O controlador é responsável por coordenar a interacção entre a vista e o modelo, o que pode originar baixa coesão por sobrecarga de responsabilidades
  - A vista tem acesso directo ao modelo, o que pode originar problemas de segurança
  - Pode ser difícil de aplicar e manter à medida que a complexidade aplicacional cresce ou no contexto de interfaces de utilização complexas

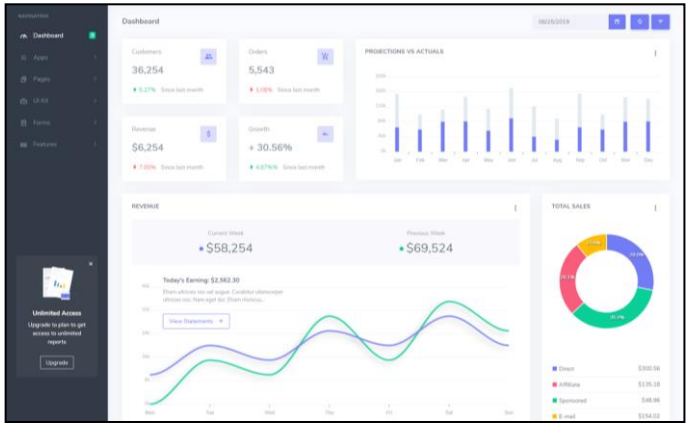
Padrão MVP

- Vantagens:
  - Separa claramente as responsabilidades entre vista, modelo e apresentador
  - Facilita a actividade de teste
  - Contribui para uma maior facilidade de reutilização de código
- Desvantagens:
  - Pode originar uma maior variedade de interacções entre as partes, resultando numa maior complexidade das respectivas interfaces

Padrão *Model-View-ViewModel* (MVVM)

Problema

Como reduzir o acoplamento entre a vista de apresentação do que o utilizador visualiza e a representação de informação do modelo de domínio no contexto de interfaces de utilização complexas?



Solução

Criar uma representação que abstrai o modelo de domínio num formato adequado para apresentação pela vista, expondo propriedades e operações específicas para o efeito

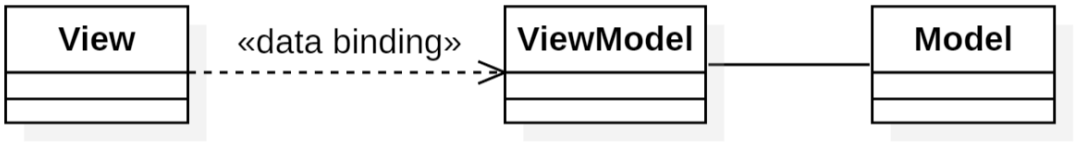
Consequências

- Aumento da coesão da vista e do modelo
- Redução do acoplamento entre vista e modelo através de mecanismos de ligação automática de dados (*data binding*)

Padrão *Model-View-ViewModel* (MVVM)

Variação do padrão MVP que tem como objectivo possibilitar uma ligação automática bidirecional (*Data Binding*) entre a vista e um adaptador vista-modelo (*ViewModel*), sem a necessidade de escrever código específico para o efeito, permitindo que as alterações feitas na vista sejam refletidas de forma automática no adaptador vista-modelo (*ViewModel*) e vice-versa

- **Características principais**
  - Adaptador de apresentação vista-modelo (*ViewModel*)
  - Actualização automática das vistas
    - ***Data binding***



Padrão *Model-View-ViewModel* (MVVM)

Vista

- Tal como nos padrões *modelo-vista-controlador* (MVC) e *modelo-vista-apresentador* (MVP), a vista é responsável pela apresentação do que o utilizador visualiza, a qual consiste numa representação de informação do modelo de domínio
- Recebe a informação resultante da interação com o utilizador, delegando o respectivo processamento no *modelo da vista* (*ViewModel*), através de mecanismos de ligação de dados (*data binding*), como propriedades ou gestores de eventos (*callbacks*), definidos para ligar a *vista* ao *modelo da vista*

Modelo

- Representa o modelo de domínio, mantendo a informação do domínio da aplicação

Modelo da Vista (*ViewModel*)

- Representa uma abstração do modelo de domínio num formato adequado para apresentação pela vista, expondo propriedades e operações específicas para o efeito
- Difere do apresentador do padrão MVP pelo facto de não ter uma associação estrutural com a vista, a ligação com a vista é realizada através das propriedades e operações que disponibiliza por ligação de dados bidirecional (*data binding*)

Requer mecanismos de ligação de dados (*data binding*), os quais podem ser disponibilizados pela plataforma de apresentação utilizada, nomeadamente de forma declarativa, ou implementados de forma específica

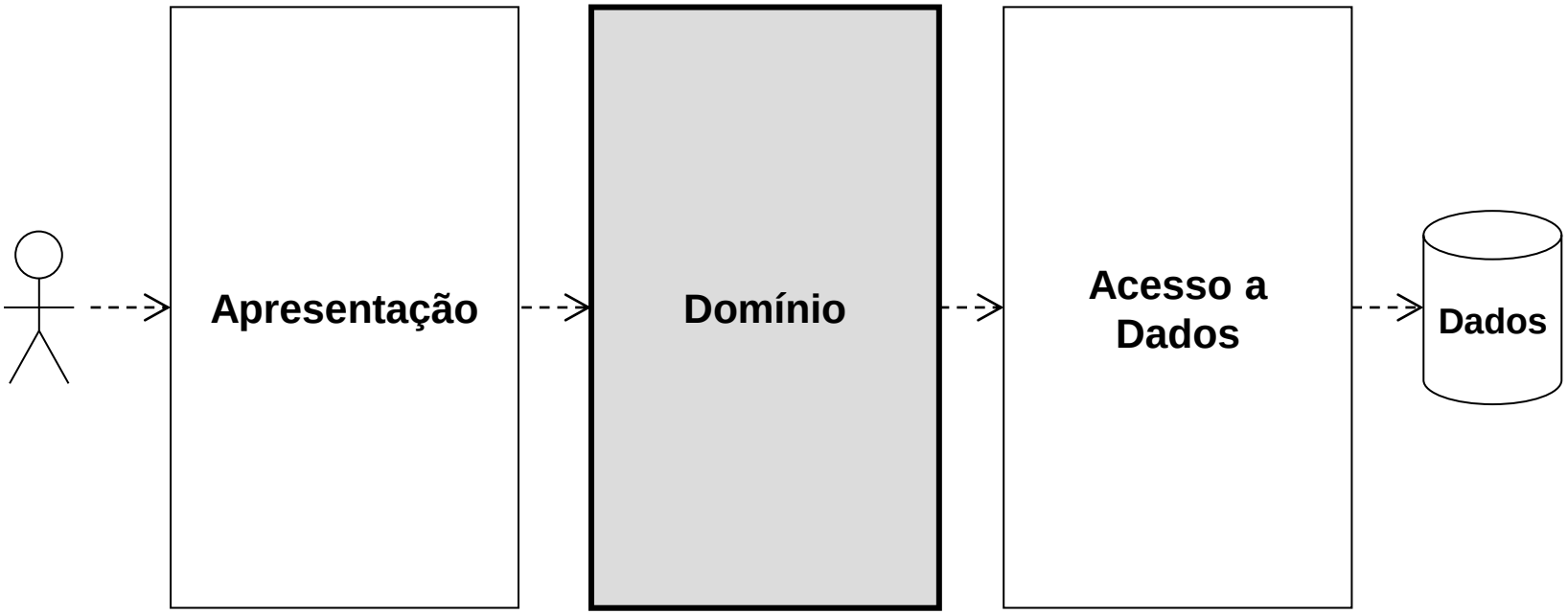
Arquitectura de 3 Camadas

Camada de Domínio (*Business Logic*)

A camada de domínio é responsável pela lógica do domínio da aplicação, contendo as classe que implementam as regras de negócio da aplicação, bem como as entidades que representam os objetos do domínio

É responsável por garantir a integridade dos dados e a consistência das operações realizadas na aplicação

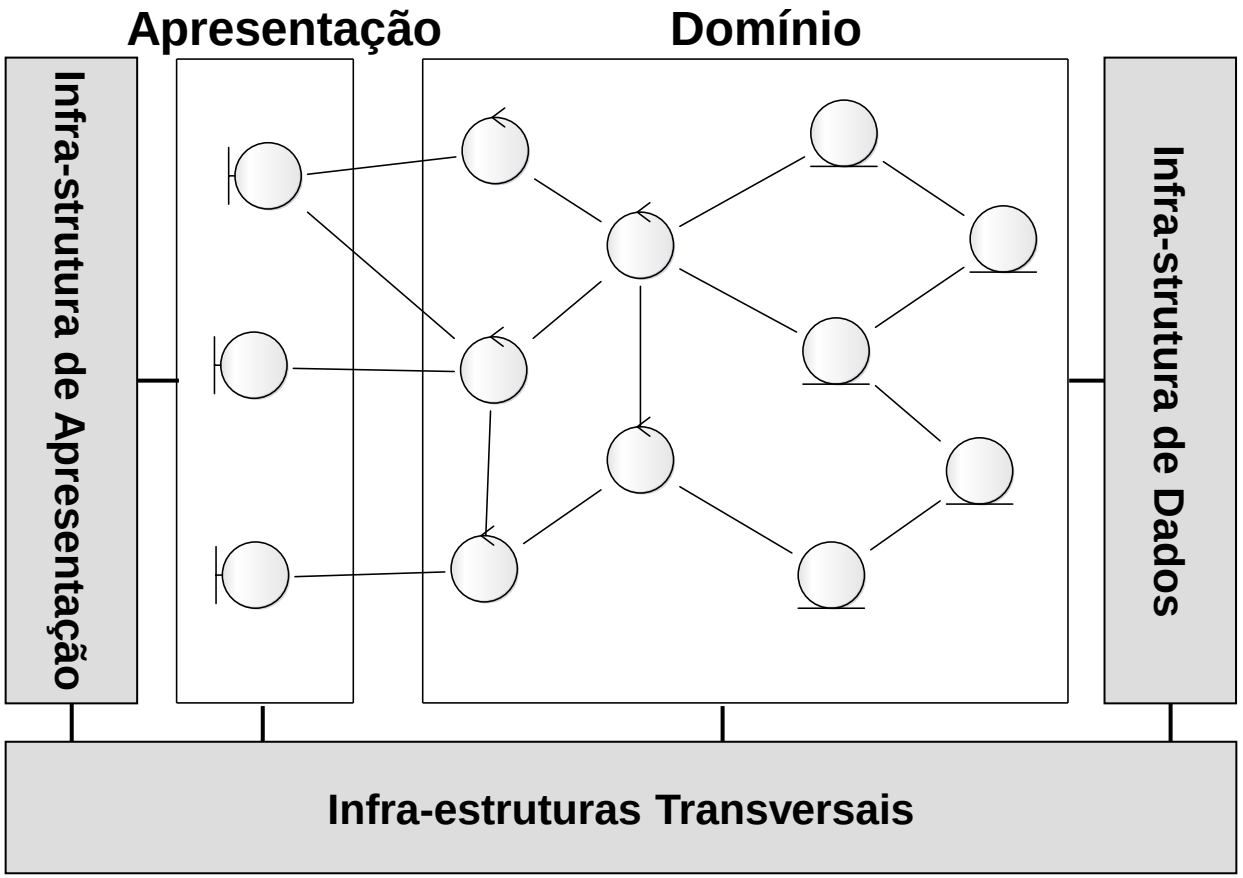
É independente da camada de apresentação e da camada de acesso a dados, o que facilita a sua reutilização



Arquitectura Orientada ao Domínio

A arquitetura orientada ao domínio é um padrão de arquitectura de software que enfatiza a importância da camada de domínio, sendo baseado no princípio de que a lógica de domínio deve ser separada das outras camadas da aplicação

É independente de tecnologias e plataformas de suporte execução (através da utilização de infraestruturas de abstracção desse suporte)



Arquitectura Orientada ao Domínio

Define um conjunto de padrões de arquitectura vocacionados para a modelação da camada de domínio, nomeadamente:

- **Entidade**
  - Representa conceitos do domínio com identidade própria
- **Objecto valor**
  - Representa dados do domínio sem identidade própria
- **Agregado**
  - Representa conceitos do domínio estruturados
- **Fábrica**
  - Responsável pela criação de partes do domínio complexas, nomeadamente agregados
- **Repositório**
  - Responsável pelo acesso e persistência de dados
- **Serviço**
  - Representa comportamento relevante do domínio que não é específico de um elemento de domínio

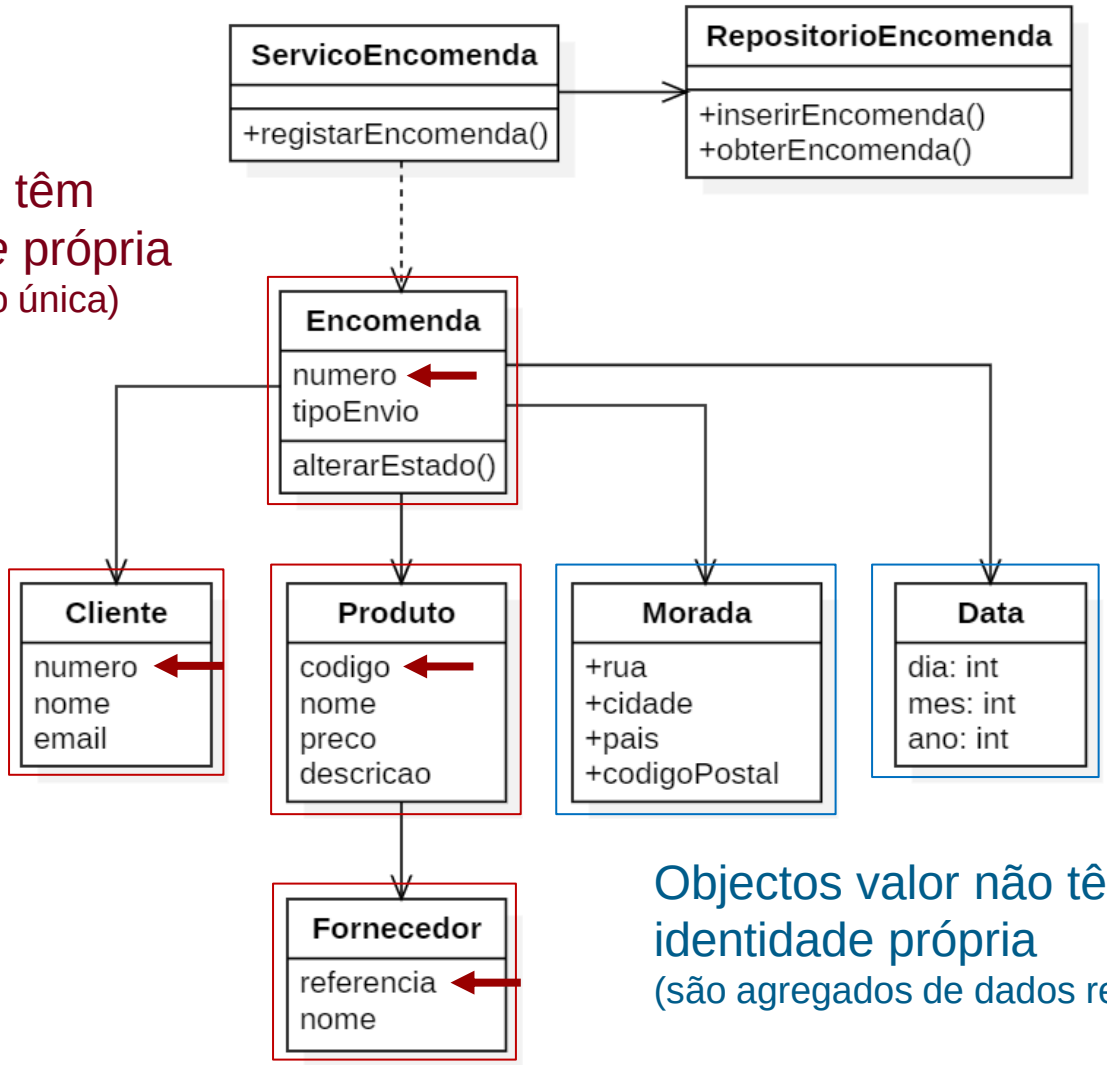
Padrões de Domínio

- **Entidade**
  - **Identidade** própria, que se mantém ao longo da operação do sistema
  - Restrição de **unicidade**
  - Manutenção de **estado** (*stateful*)
- **Objecto valor**
  - Mantém atributos de um elemento de domínio
  - Ausência de **identidade**
  - Imutáveis
  - Partilhados

Padrões de Domínio

Exemplo: Sistema de gestão de encomendas

Entidades têm *identidade* própria
(identificação única)



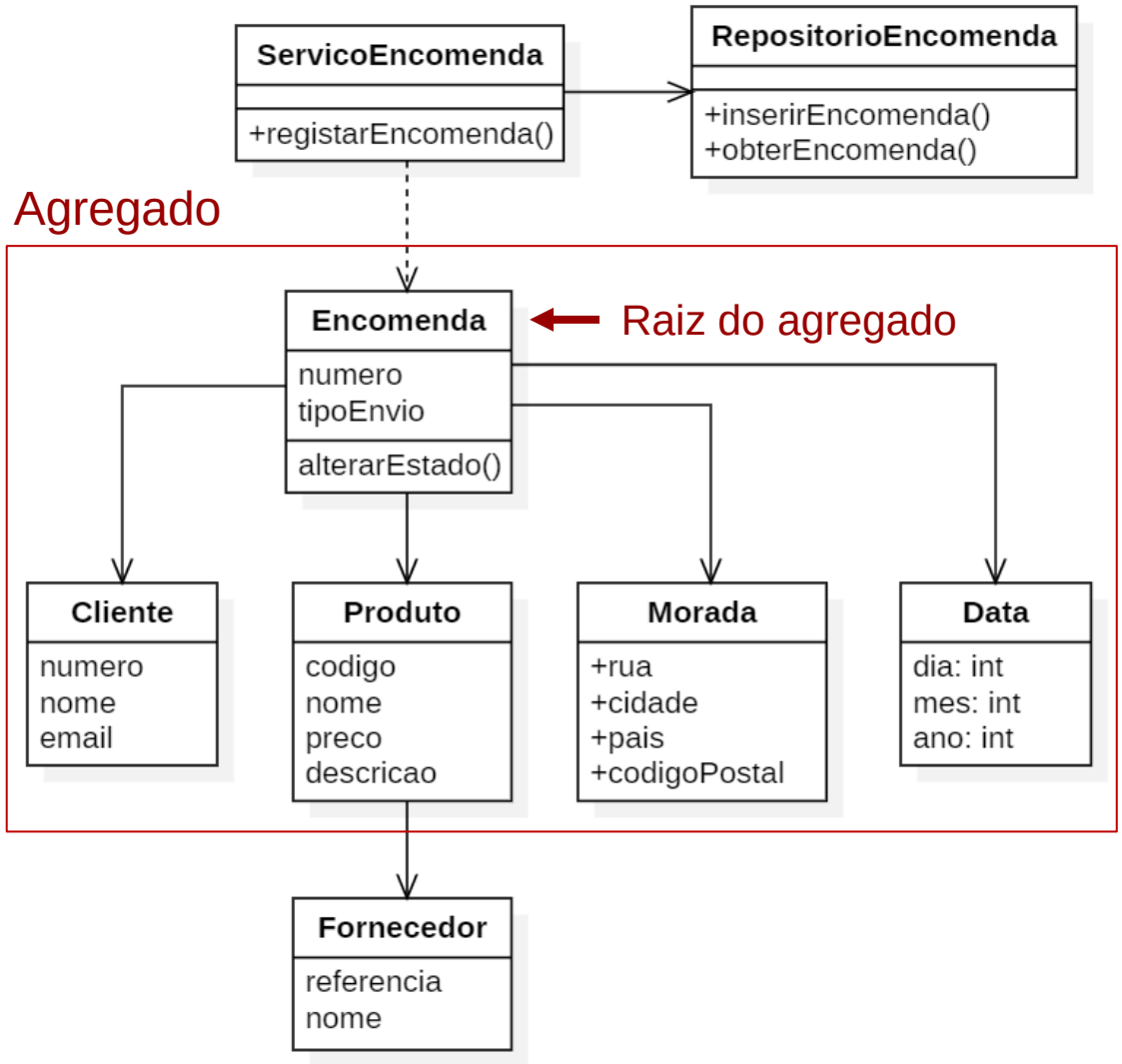
Objectos valor não têm identidade própria
(são agregados de dados relacionados)

Padrões de Domínio

- **Agregado**
  - Agregação e contexto de entidades relacionadas
  - Mantém a consistência de objectos relacionados
  - Tem um **entidade raiz**
    - Identidade global
    - Objectos agregados têm identidade local
    - Do exterior apenas a raíz é referenciada
  - Isola os elementos interiores do exterior
    - Encapsulamento

Padrões de Domínio

Exemplo: Sistema de gestão de encomendas



Padrões de Domínio

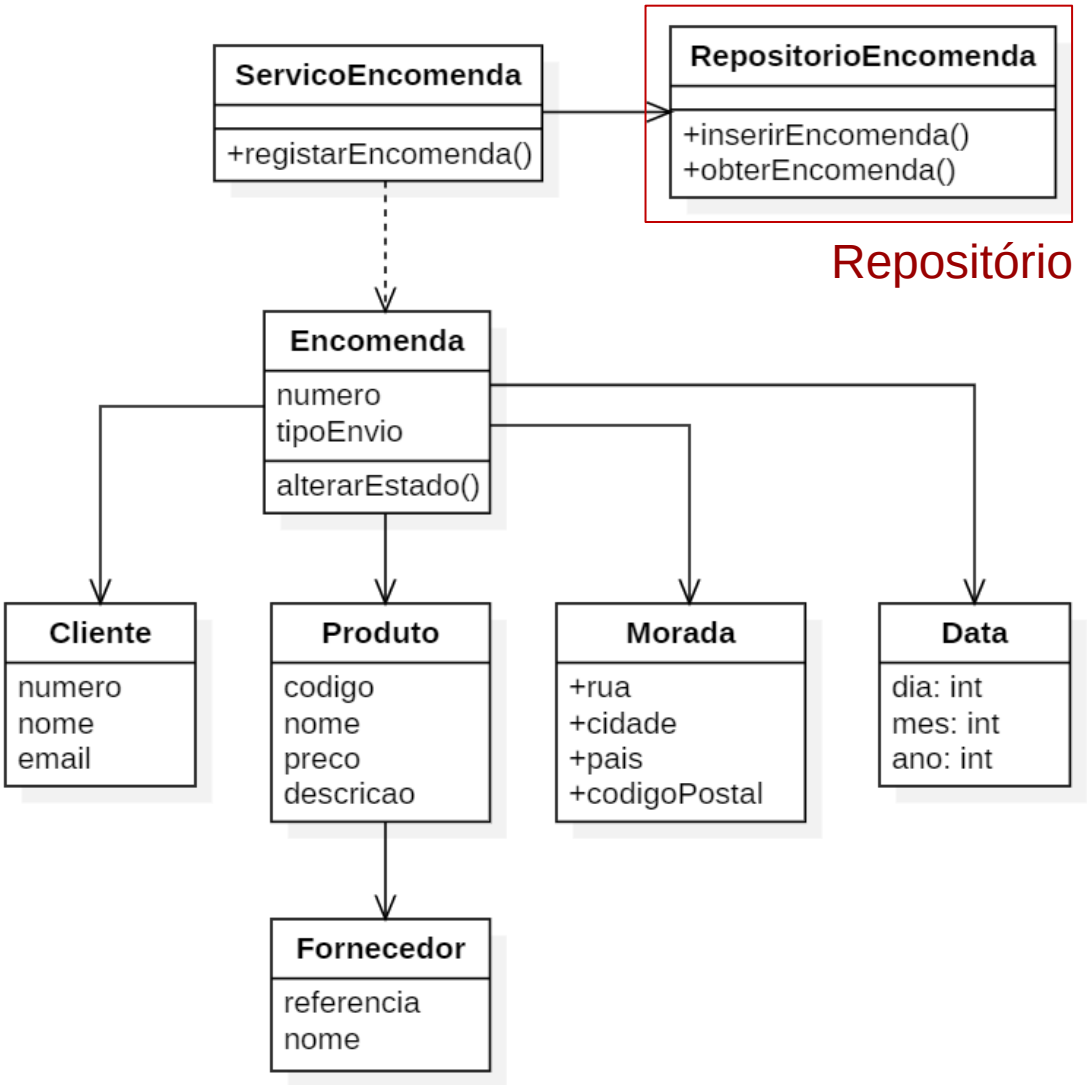
- **Fábrica**
  - **Gestão de criação** de objectos do domínio
    - Para objectos do domínio complexos
  - Encapsulamento de conhecimento acerca do modo de criação dos objectos
    - Garantia de coesão e encapsulamente
    - Em especial no caso de **agregados**
  - Processo de criação opera de forma atómica

Padrões de Domínio

- **Repositório**
  - **Gestão de armazenamento** de objectos do domínio
  - Encapsulamento do processo de armazenamento e obtenção de objectos do domínio
  - Pode utilizar diferentes estratégias de armazenamento
  - Funcionalidade
    - Inserir
    - Obter
    - Actualizar
    - Remover

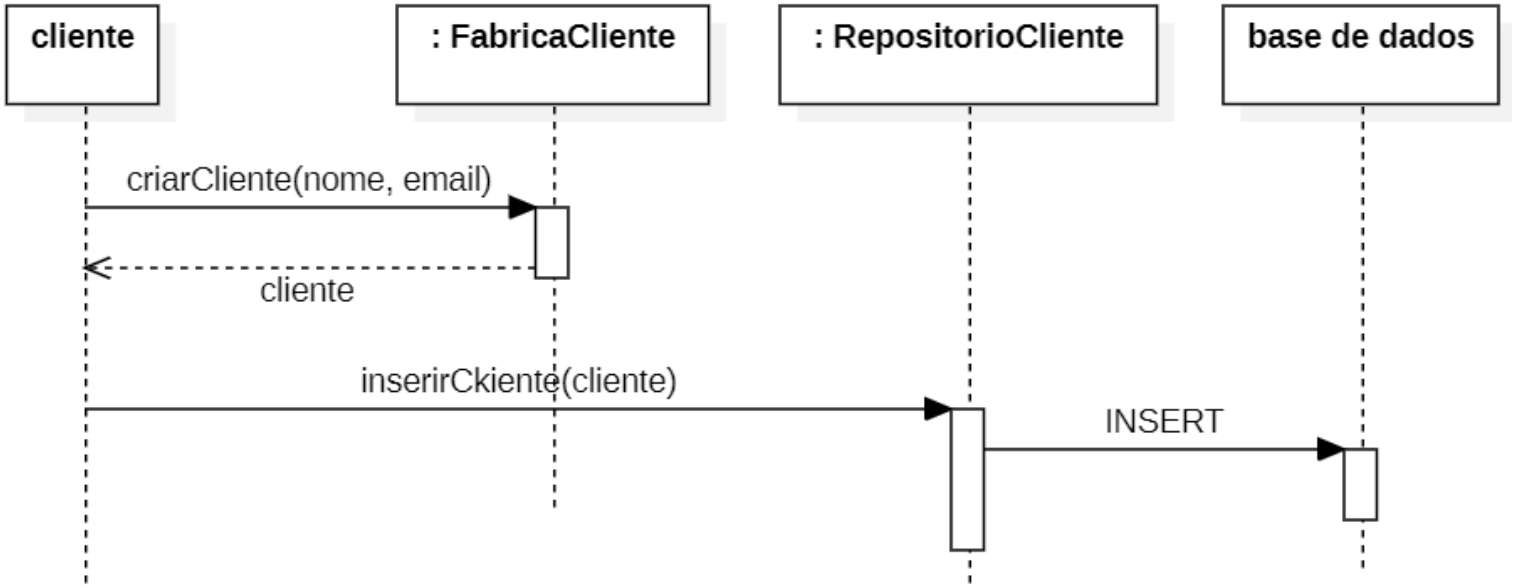
Padrões de Domínio

Exemplo: Sistema de gestão de encomendas



Padrões de Domínio

Exemplo de colaboração no acesso a um repositório



Identidade de Entidades

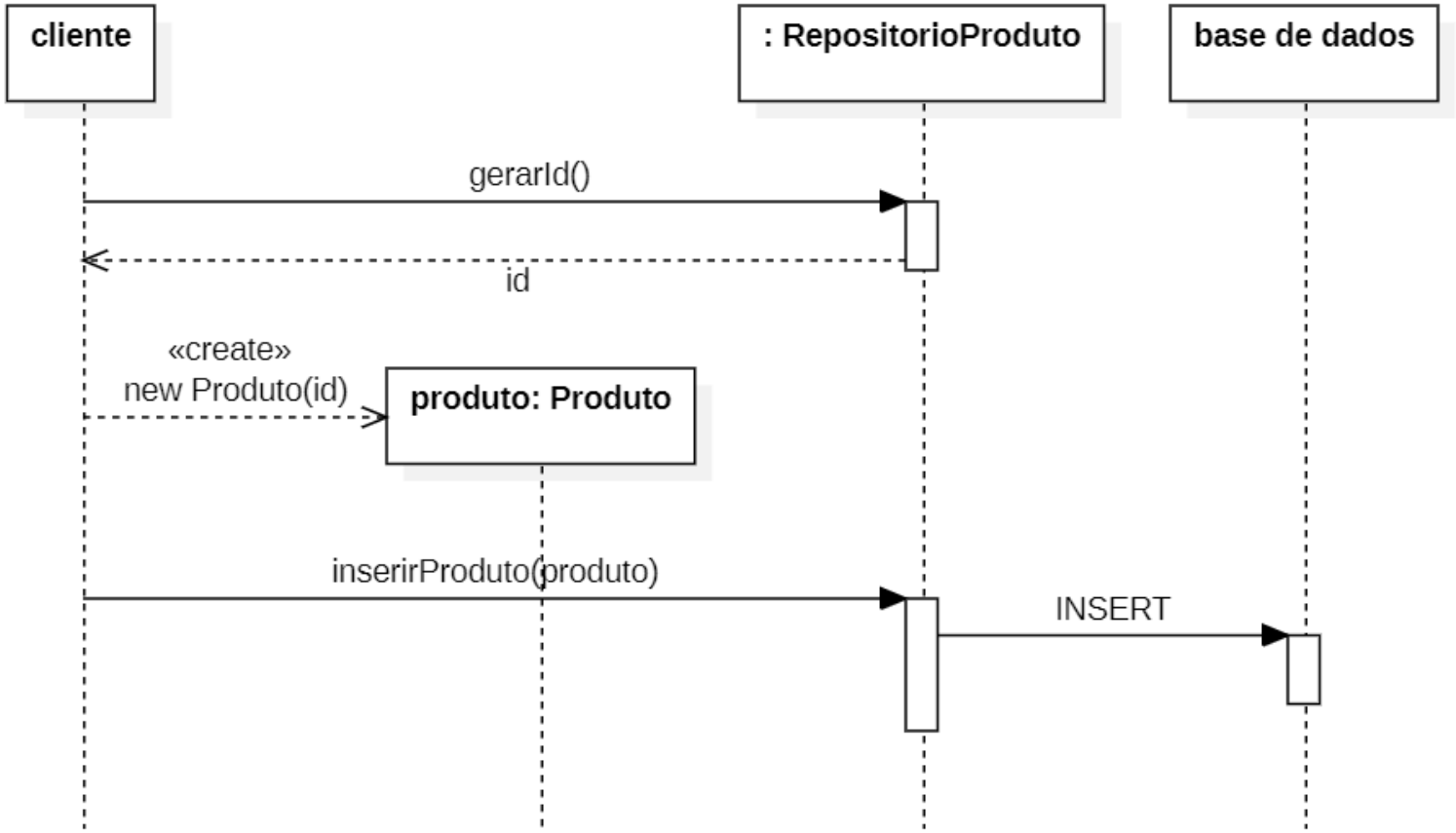
A identidade das entidades deve ser adequadamente gerida, nomeadamente, na relação com os repositórios

Duas vertentes principais de geração da identidade das entidades:

- Obtenção de identidade **prévia** à criação da entidade
  - Identidade gerada pelo repositório e atribuída quando a entidade é criada
  - O mecanismo de geração de identidade é encapsulado e mantido pelo repositório
- Obtenção de identidade **posterior** à criação da entidade
  - Identidade gerada pelo repositório a primeira vez que a entidade é persistida
  - Identidade gerada com base na plataforma de persistência de dados

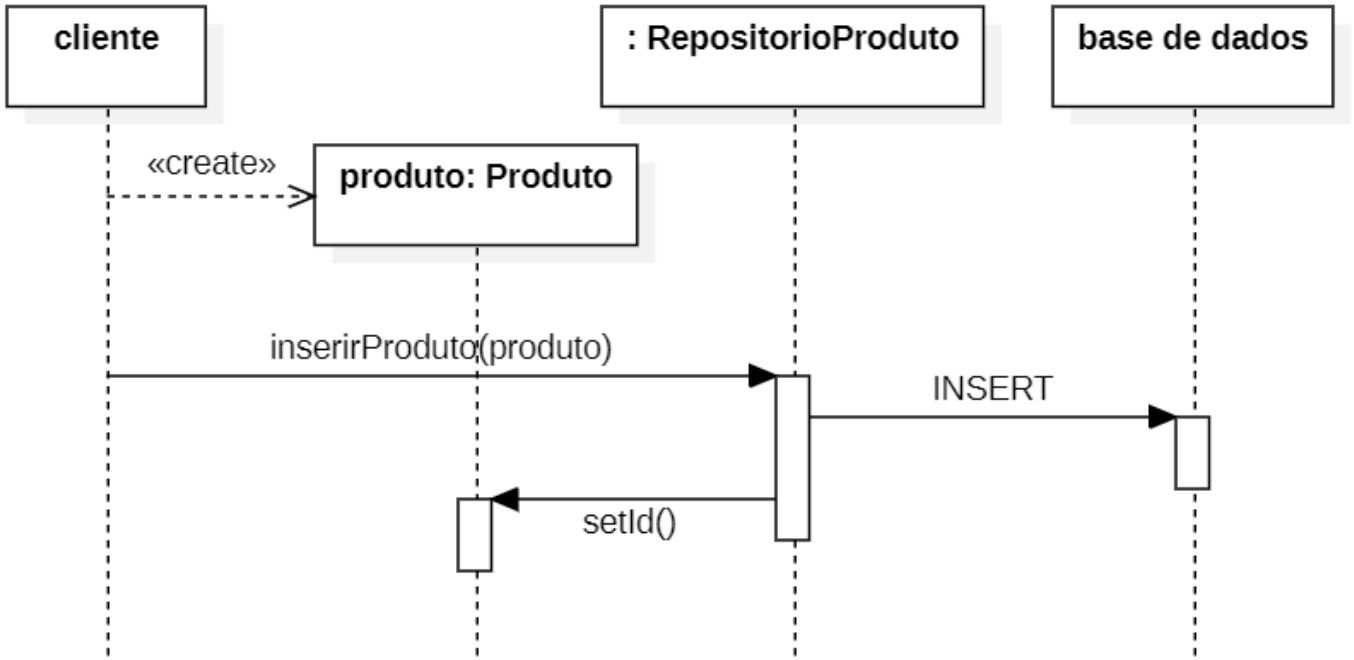
Identidade de Entidades

- Obtenção de identidade prévia à criação da entidade
  - Identidade gerada pelo repositório e atribuída quando a entidade é criada
  - O mecanismo de geração de identidade é encapsulado e mantido pelo repositório



Identidade de Entidades

- Obtenção de identidade posterior à criação da entidade
  - Identidade gerada pelo repositório a primeira vez que a entidade é persistida
  - Identidade gerada com base na plataforma de persistência de dados

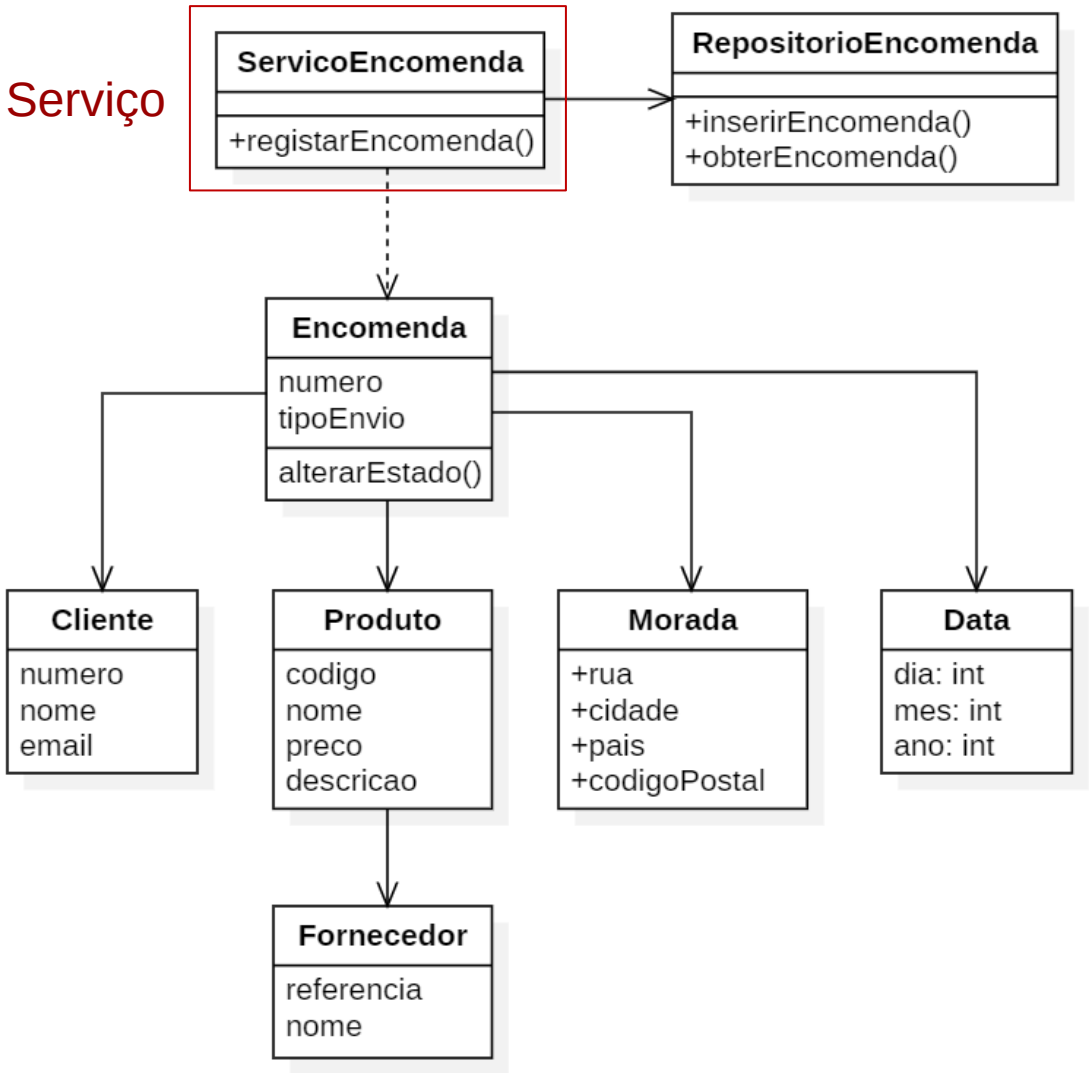


Padrões de Domínio

- **Serviço**
  - Representa comportamento relevante do domínio que não é específico de um elemento de domínio
  - **Coordena** a actividade de diferentes objectos
  - Não mantém estado (***stateless***)
  - Não deve substituir a funcionalidade específica de objectos do domínio, pelo que as operações realizadas
    - referem-se a conceitos de domínio que **não pertencem de forma natural** a **entidades** ou **objectos valor**
    - referem-se a outros objectos do domínio
    - **não mantêm estado** (*stateless*)
  - Diferente de serviços de isolamento do domínio

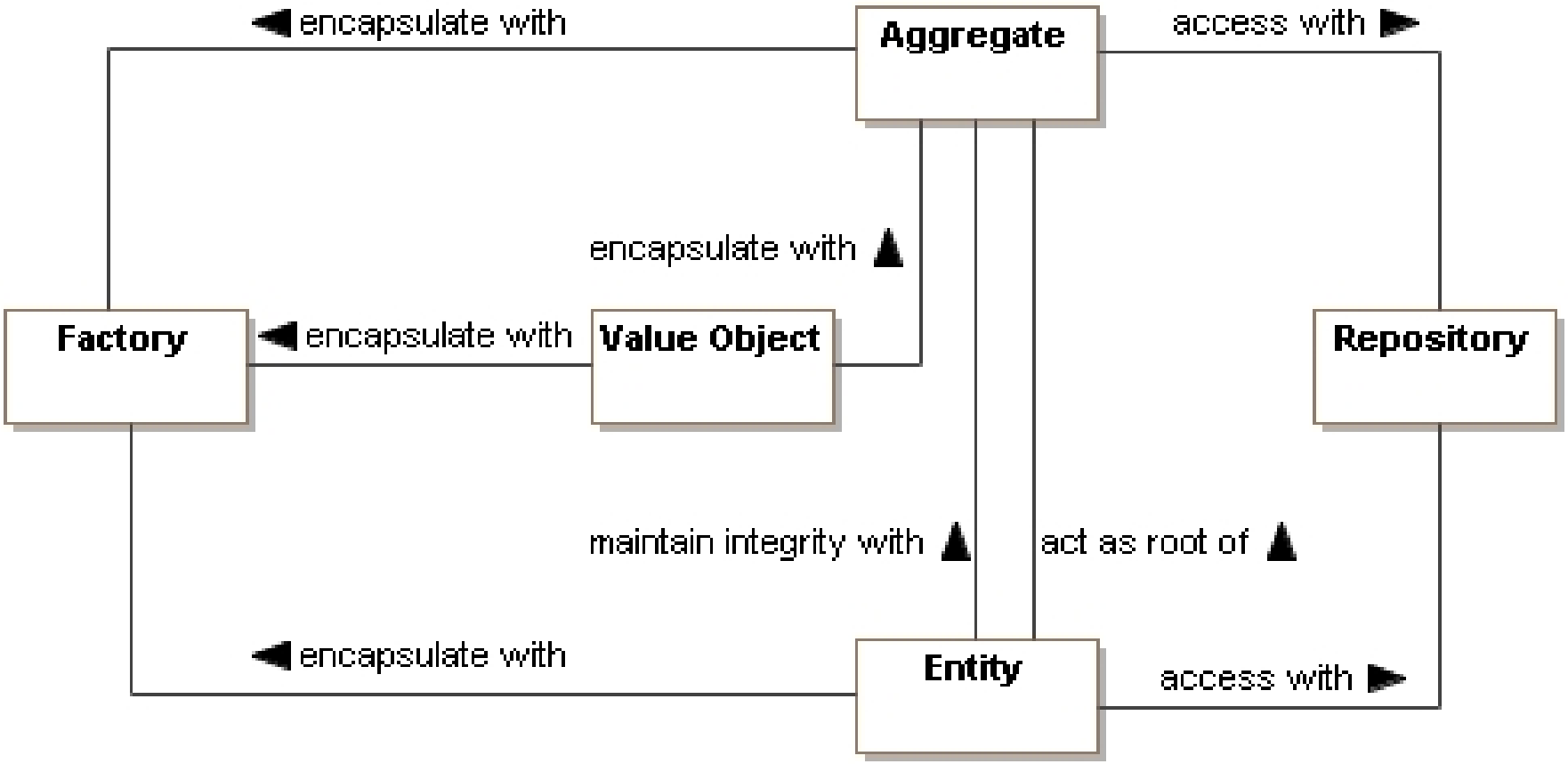
Padrões de Domínio

Exemplo: Sistema de gestão de encomendas



Padrões de Domínio

Relações conceptuais entre padrões de domínio



Bibliografia

[Pressman, 2003]

R. Pressman, *Software Engineering: a Practitioner's Approach*, McGraw-Hill, 2003.

[Gamma et al., 1995]

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Shaw & Garlan, 1996]

M. Shaw, D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Vernon, 2013]

V. Vernon, *Implementing Domain Driven Design*, Addison-Wesley, 2013.

[Parnas, 1972]

D. Parnas, *On the Criteria to Be Used in Decomposing Systems into Modules*, Communications of the ACM 15-12, 1968.

[Kruchten, 1995]

F. Kruchten, *Architectural Blueprints - The "4+1" View Model of Software Architecture*, IEEE Software, 12-6, 1995.

[Burbeck, 1992]

S. Burbeck; *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html,1992

[Booch, 2004]

G. Booch, *Software Architecture*, IBM, 2004.