Engenharia de Software

Arquitectura de Software
Projecto Detalhado

Luís Morgado

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

Níveis de projecto

- Projecto de Subsistemas

  Âmbito: Subsistemas, Processos, …

  O que é especificado:

  - Organização do sistema
  - Estratégias de concorrência e comunicação entre processos
  - ...

  Sistema

- Projecto de Mecanismos

  Âmbito: Grupos de classes

  O que é especificado:

  - Instâncias de Padrões de Projecto (“Design Patterns”)
  - Utilização de classes Contentoras
  - Estratégias de gestão de erros (nível intermédio)
  - ...

  Inter-objecto

- Projecto Detalhado

  Âmbito: Classes, Interfaces, …

  O que é especificado:

  - Detalhes de implementação de atributos e operações
  - Definição de algoritmos
  - ...

  Intra-objecto

Projecto Detalhado

- No projecto detalhado são tomadas decisões acerca da estrutura interna dos elementos do modelo, em particular:
  - Estrutura de dados
  - Implementação das associações
  - Operações definidas sobre os dados
  - Visibilidade de dados e operações
  - Algoritmos utilizados na implementação das operações
  - Gestão de excepções

Estrutura de Dados

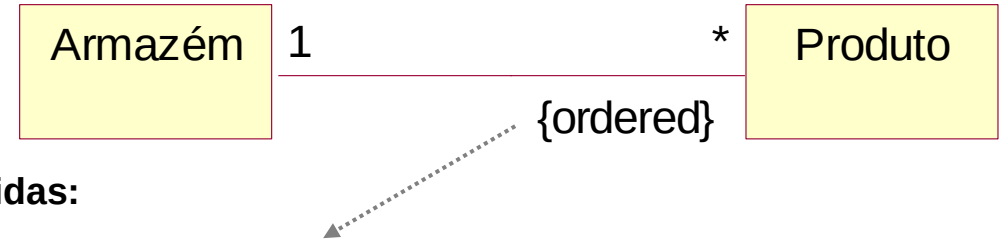
- Apesar da estrutura dos dados de um objecto ser (normalmente) simples, há que definir essa estrutura tendo em conta:
  - **Gamas de valores admissíveis para os dados**
  - **Precisão**
  - **Valores iniciais**
  - **Valores por omissão**
  - **Pré-condições**
  - **Persistência**
- As operações de manipulação dos dados devem garantir que os valores dos dados respectivos estão dentro das **gamas de valores admissíveis**, e que as **pré-condições são satisfeitas**.

Estrutura de Dados

- No caso dos atributos primitivos corresponderem a estruturas de dados há que especificar o tipo de estrutura de dados a utilizar

- Em UML isso pode ser feito através da utilização de restrições associadas ao papel ("*role*") que uma classe desempenha numa associação

- **Exemplo:**



Restrições definidas:

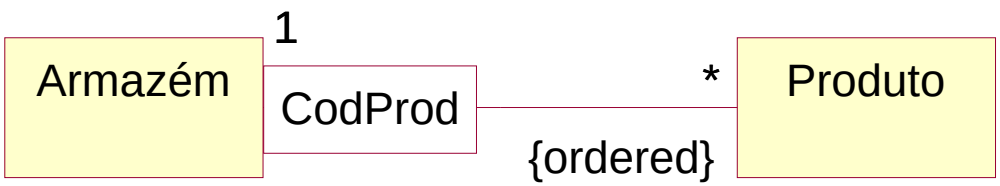
- {ordered} - os elementos são mantidos de forma ordenada
- {bag} - não é garantida a ordenação, podendo existir elementos repetidos
- {set} - não é garantida a ordenação, não sendo admitidos elementos repetidos
- {hashed} - os elementos são acedidos através de uma chave de dispersão ("*hash key*")

A selecção do tipo de *contentor* depende das características que se pretendem optimizar. Por exemplo, as árvores balanceadas são bastante eficientes na procura, contudo a inserção de novos elementos é complexa e menos eficiente.

Estrutura de Dados

- Ao definir as estruturas de dados há que ter em conta a possível qualificação das associações

- **Exemplo:**



O qualificador de uma associação designa uma chave utilizada para obter um item da colecção respectiva

Implementação:

No exemplo anterior `CodProd` deverá ser um atributo de `Produto`, podendo as operações de acesso a um produto ser definidas da seguinte forma:

```
Produto* get_the_Produto (int CodProd);
void set_the_Produto (int CodProd,Produto* Prod);
```

Implementação das associações

- A estratégia de implementação das associações deve ter em conta:
  - ***Multiplicidade***
  - ***Tipo de contenção*** (referência, valor)

- **Exemplo de implementação em C++:**

```
class ClasseExemplo {
   T t1;
   T* t2;
}
```

Apropriado apenas para ***composição*** 1 para 1

Apropriado para ***associação*** ou ***agregação*** 1 para 0 ou 1

Associações 1 para *n* devem ser implementadas com base na utilização de **contentores** adequados. Apesar de, em algumas situações, ser possível a utilização directa de *arrays* (por exemplo no caso de relações de utilização de um número definido de objectos)

Operações

- Nos modelos de análise as operações são abstraídas em termos de mensagens trocadas entre os objectos

- **Implementação**:
  - Através de ***activação directa*** (chamada dos métodos respectivos)
  - Através de ***mecanismos de comunicação entre processos***

- Nos modelos de análise (normalmente) apenas são identificadas as operações públicas. **Porquê ?**
  - No **projecto detalhado** há que **identificar outras operações utilizadas** internamente no âmbito das operações base

```
Template <class T, int size>
class Queue {
  protected:
    ...
  public:
    ..
    virtual void Put(T elem);
    virtual T Get(void);
};
```

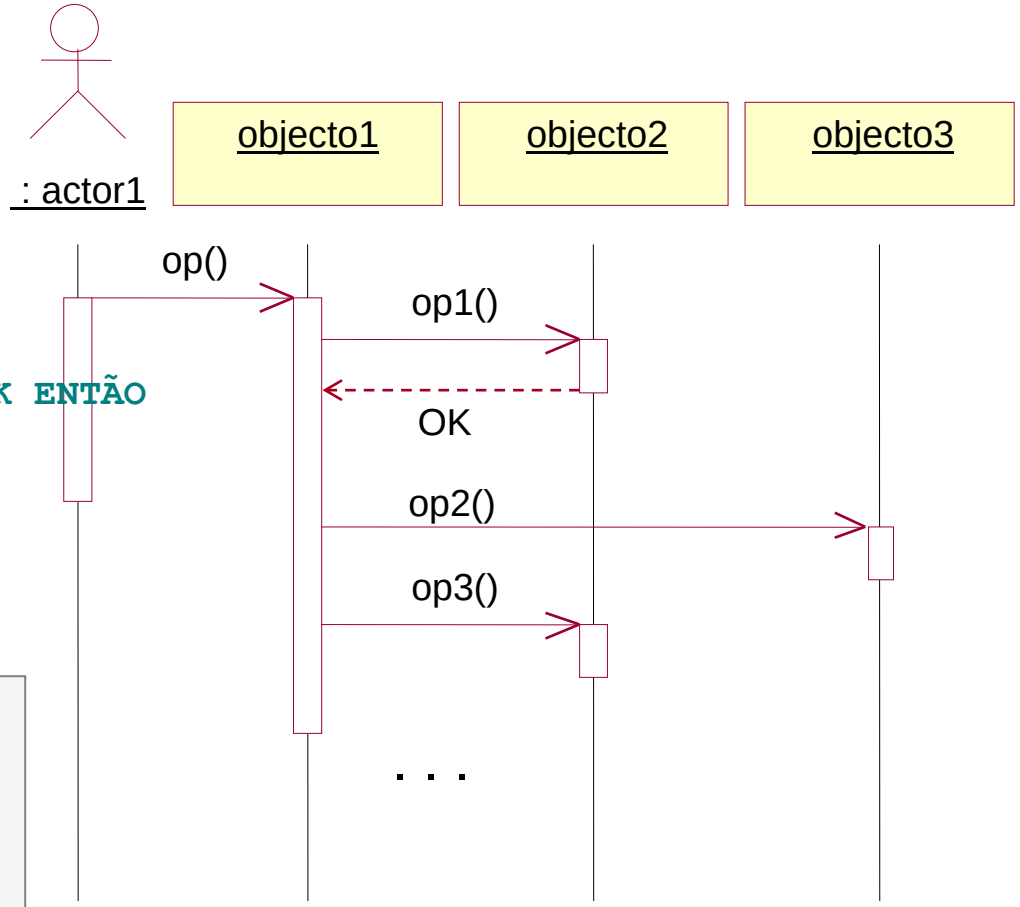
```
Template <class T, int size>
class Queue {
  protected:
    ...
    void WriteToDisk(void);
    void ReadFromDisk(void);
  public:
    ..
    virtual void Put(T elem);
    virtual T Get(void);
};
```

No caso de se pretender **reutilização**, cada classe deve disponibilizar um **conjunto completo de operações**, mesmo que num determinado contexto nem todas sejam utilizadas.

Especificação das Operações

Normalmente em *pseudo-código*

```
FUNÇÃO op ...
  SE op1() = OK ENTÃO
    op2()
    op3()
  ...
FIM op
```



Outras formas de especificação de operações:

- Diagramas de actividade
- Diagramas de transição de estado

Visibilidade

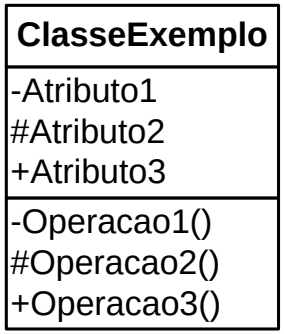
- Em UML ***visibilidade*** refere-se à acessibilidade aos elementos internos dos objectos.
- Alguns aspectos a ter em conta na definição da visibilidade:
  - Se os clientes necessitam de um elemento, então deve ser visível, caso contrário não deve ser visível (***encapsulamento***)
  - Tornar **visíveis apenas operações com uma semântica apropriada**. Por exemplo, um contentor que tenha por base uma árvore binária deverá suportar: **GetLeft**(), **GetRight**() ou **Prev**(), **Next**() ?
    - **Prev**(), **Next**() mantém a semântica essencial sem expor a implementação!
  - **Atributos nunca devem ser directamente visíveis aos clientes**
  - Quando diferentes níveis de visibilidade são necessários devem ser definidas diferentes ***interfaces***

Minimizar Acoplamento

Níveis de Visibilidade

3 Níveis base de visibilidade:

- ***Privado***: Acessível apenas no contexto da classe
- ***Protegido***: Acessível no contexto da classe e das subclasses
- ***Publico***: Acessível por outras classes



Implementação:

Mapeamento directo para os atributos de visibilidade das linguagens:

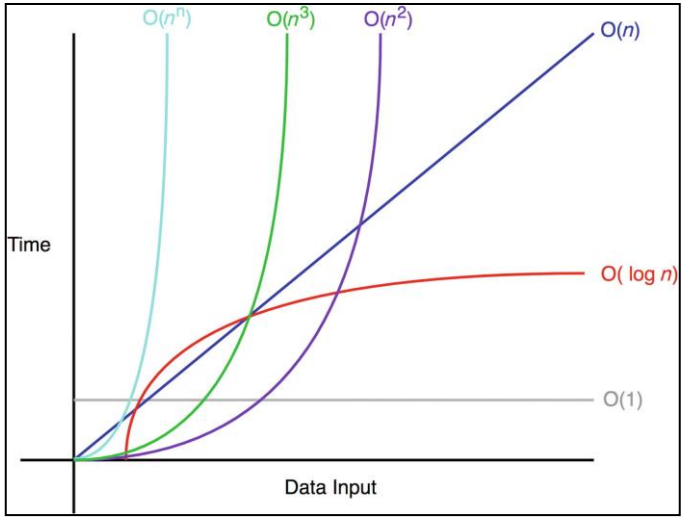
```
Private
Protected
Public
```

Algoritmos

- **Optimização dos algoritmos com base em diferentes critérios**:
  - ***Complexidade temporal*** (tempo de execução necessário)
    Notação "***ordem de***":
    - O(*c*)
    - O($\log_2 n$)
    - O(*n*)
    - O(n^2)
    - **…**

    c - constante
    n - número de elementos envolvidos no cálculo

    Algoritmos com a mesma complexidade só diferem num factor multiplicativo ou aditivo constante



  - **Outros critérios:**
    - Memória necessária
    - Simplicidade e correcção
    - Tempo e esforço de desenvolvimento
    - Facilidade de reutilização
    - Robustez e segurança

Gestão de Excepções

- **Utilização de mecanismos de gestão de excepções**:
  - Evita que possíveis situações de erro sejam ignoradas
  - Permite separar o processamento associado a situações de erro, do processamento normal
  - Permite estruturar o processamento de erros

- **Exemplo**:

```
...
if( f1(x,y) ) == NULL) {
  /* processamento de erro */
  /* passar erro ao nível superior */
  return -1;
}

if( !f2() ) {
  /* processamento de erro */
  /* passar erro ao nível superior */
  return -1;
}
```

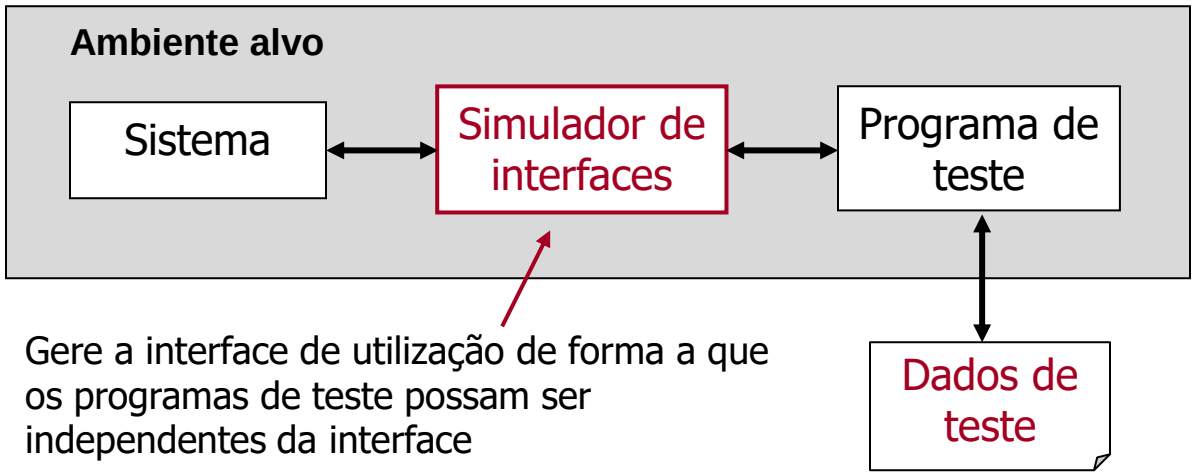
```
...
try {
  f1(x,y);
  f2();
}

/* Processamento de execpções */
catch( Falha1& f1 ) {
  /* processamento de erro */
  throw;
}
catch( Falha2& f2 ) {
  /* processamento de erro */
  throw;
}
```

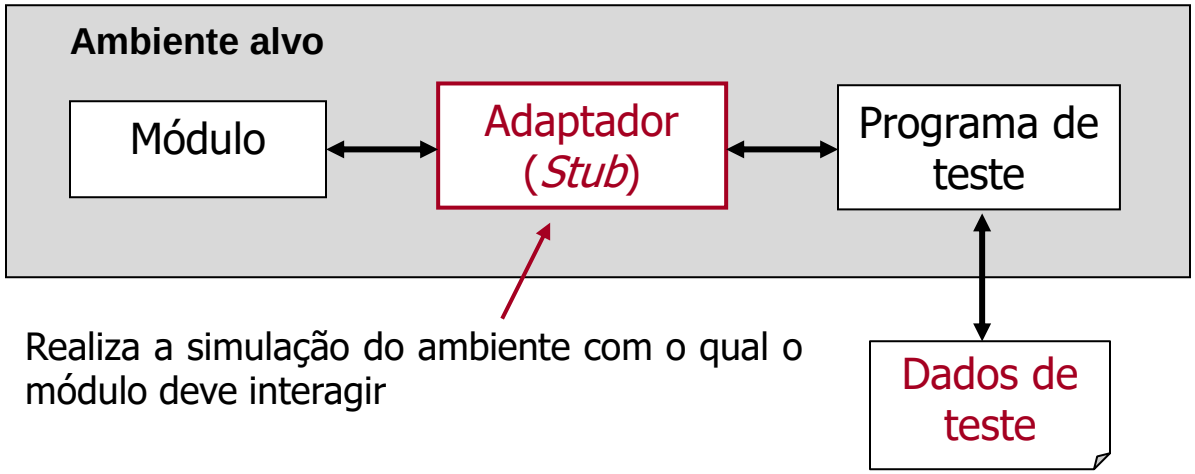
Gestão de Excepções

- Alguns aspectos a ter em conta na gestão de excepções:
  - Os mecanismos de gestão de excepções introduzem alguma sobrecarga de processamento, com a consequente degradação de desempenho
  - As operações de eliminação de objectos ("*destructors*") não devem gerar excepções, nem activar operações que gerem excepções
  - A gestão de excepções deve ser um aspecto fundamental no projecto de algoritmos
  - As excepções podem ser representadas de forma explícita como eventos nos diagramas de transição de estado, e nos diagramas de actividade
  - **Que excepções devem ser capturadas** ("*catch*") ?
    - Todas aquelas que a operação respectiva tenha contexto suficiente para tratar, ou que não façam sentido a um nível superior
  - **Que excepções devem ser geradas** ("*throw*") ?
    - Todas as outras. Contudo, deverá existir sempre um nível a que cada excepção seja processada, e cuja acção correspondente dependerá das suas características

Ambiente de Teste



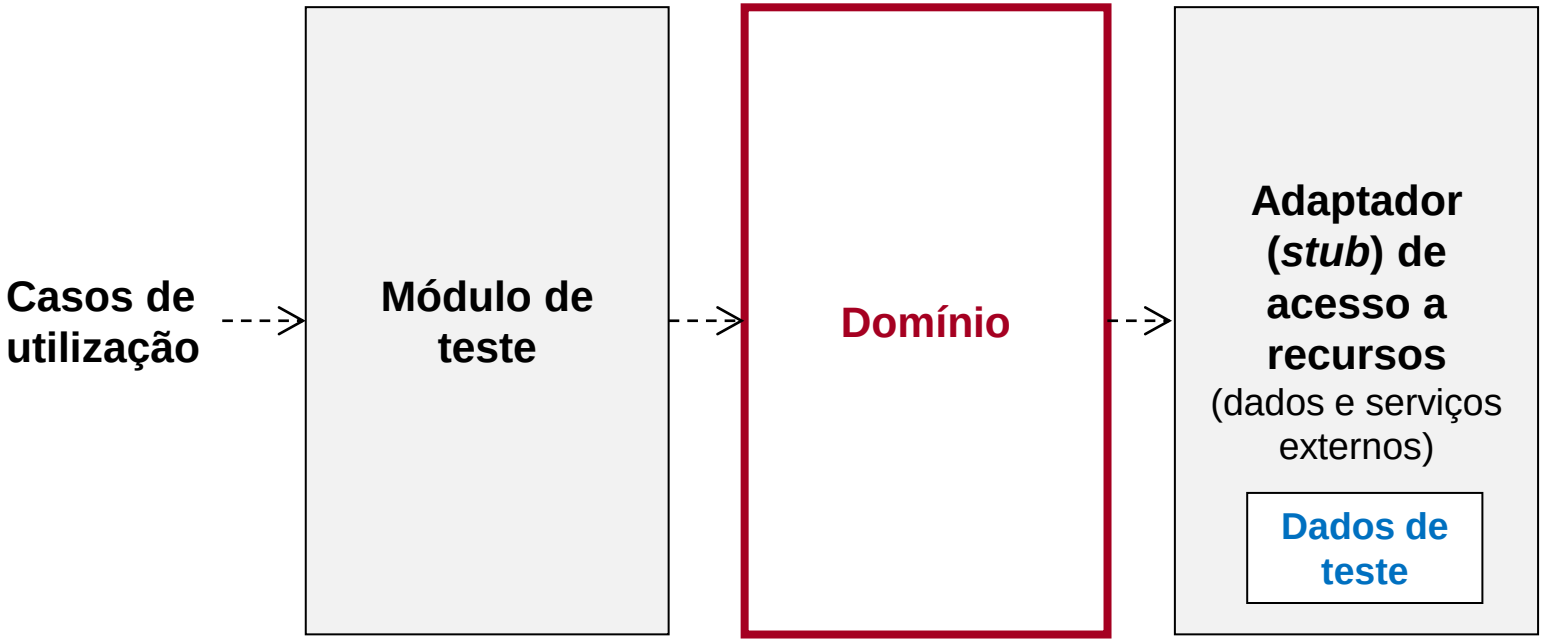
Gere a interface de utilização de forma a que os programas de teste possam ser independentes da interface



Realiza a simulação do ambiente com o qual o módulo deve interagir

Arquitectura de Teste

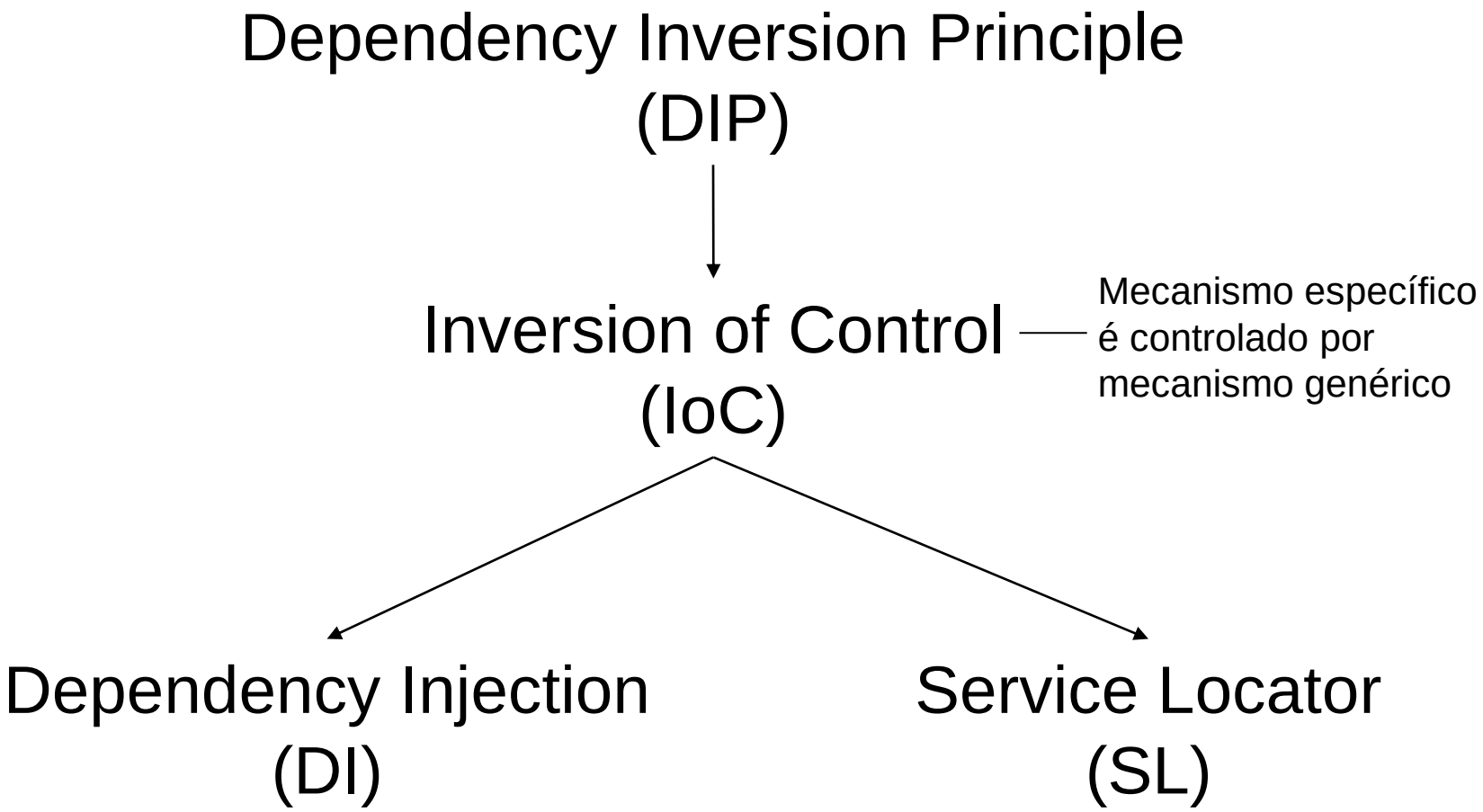
- **Exemplo**
  - Teste da camada de domínio com base em casos de utilização



Princípios de Arquitectura Orientada a Objectos

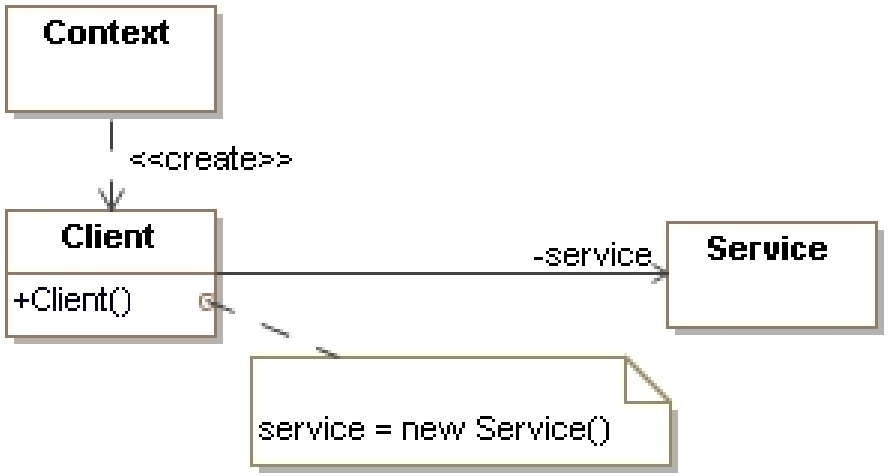
- **S**ingle Responsibility Principle
  - *A class should have one, and only one, reason to change*
  - A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
- **O**pen Closed Principle
  - *You should be able to extend a classes behavior, without modifying it*
  - Software entities should be open for extension, but closed for modification
- **L**iskov Substitution Principle
  - *Derived classes must be substitutable for their base classes* (*design by contract*)
  - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program
- **I**nterface Segregation Principle
  - *Make fine grained interfaces that are client specific*
  - Many client-specific interfaces are better than one general-purpose interface
- **D**ependency Inversion Principle
  - Depend on abstractions, not on concretions

Princípio da Inversão de Dependência

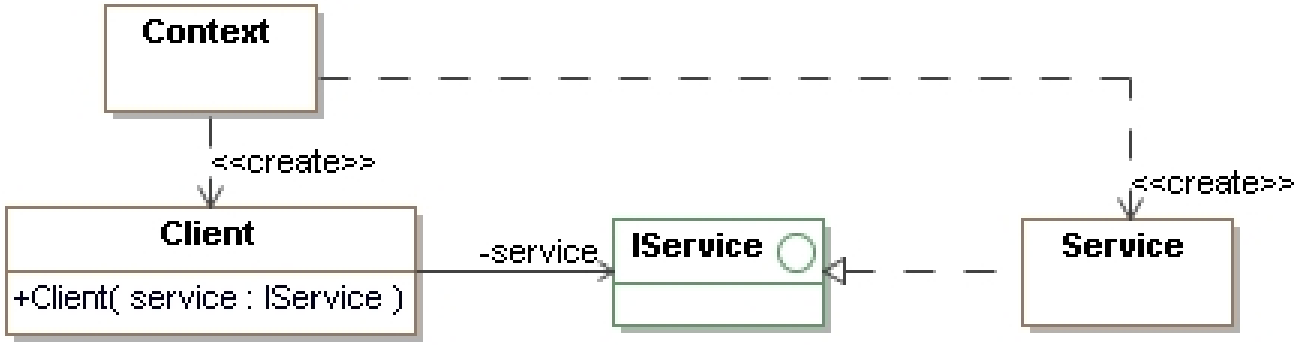


Injecção de Dependências

Sem injecção de dependência



Com injecção de dependência



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.