Engenharia de Software

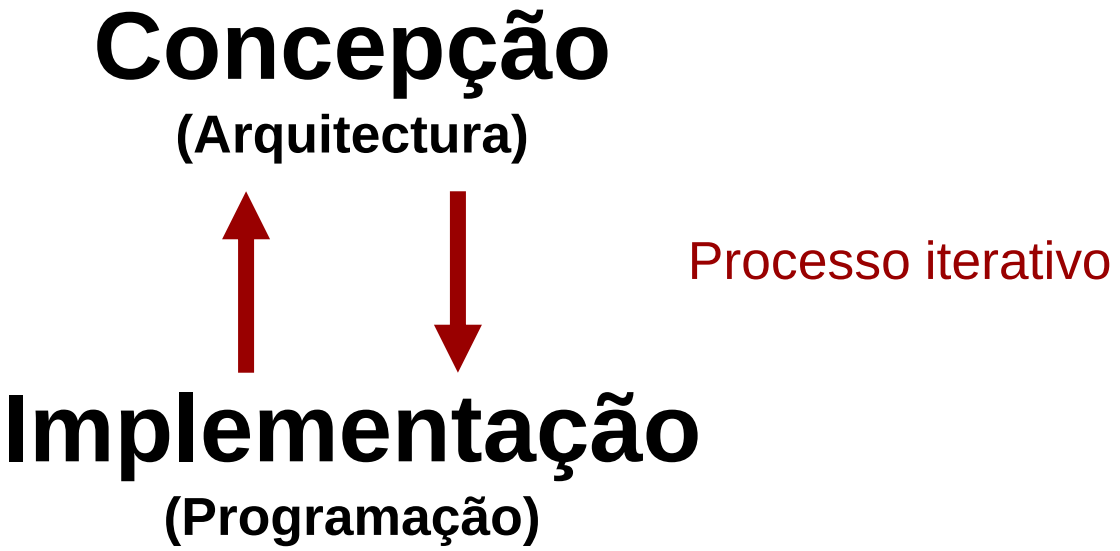
Refactorização e Anti-padrões de Software

Luís Morgado

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

Refactorização

"*With design I can think very fast, but my thinking is full of little holes*." [Alistair Cockburn]



Técnicas Associadas a Processos Ágeis

- ***Refactoring***
  - Refactoring is the **process of rewriting a computer program** or other material **to improve its structure or readability**, while explicitly **keeping its meaning or behavior**
  - Refactoring **does not fix bugs** or **add new functionality**
  - it is designed to **improve the understandability of the code** or **change its structure and design**, and remove dead code, to **make it easier for human maintenance** in the future

[Kent Beck, 2001]

Refactorização

- **Motivação**
  - Arquitectura/programas que são **difíceis de compreender** são **difíceis de alterar**
  - Arquitectura/programas **com redundância** são **difíceis de alterar**
  - Arquitectura/programas com **estrutura ou dinâmica interna complexas** são **difíceis de alterar**

Refactorização

- **Refactorização**
  - **Alteração da estrutura interna** de uma parte de software para **melhorar a sua arquitectura,** compreensibilidade e adaptabilidade, **sem alterar o seu comportamento** observável.

- Desenvolvimento de software
  - Adicionar funcionalidade
  - Refactorização

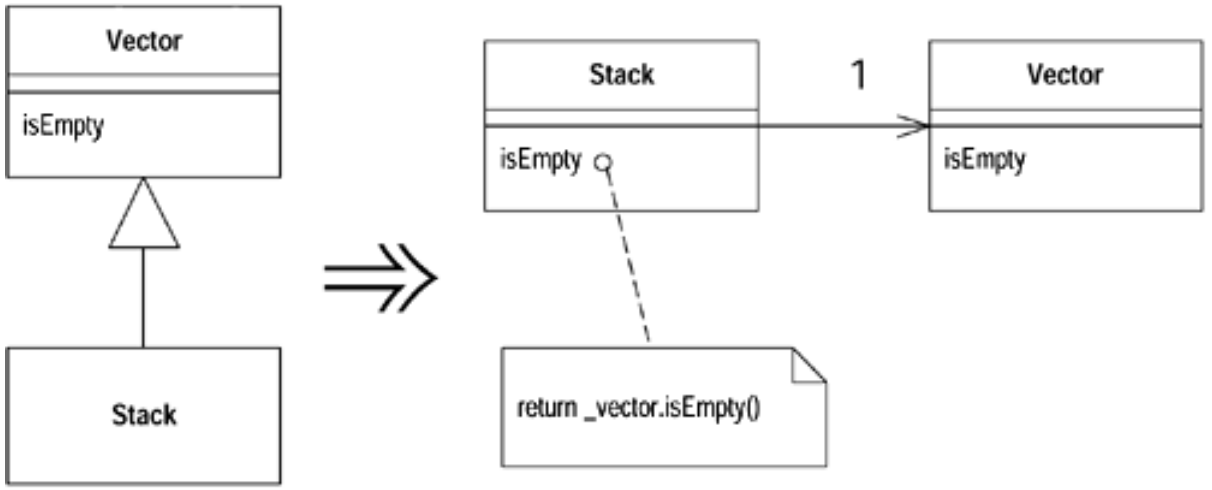
Refactorização

- Tipos de refactorização
  - Refactorização de classes
  - Refactorização de métodos
  - Refactorização de dados
  - Refactorização de expressões
  - Refactorização de hierarquia
  - Refactorização geral

Exemplo

Replace Inheritance with Delegation

- A subclass uses only part of a superclasses interface or does not want to inherit data
- Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing

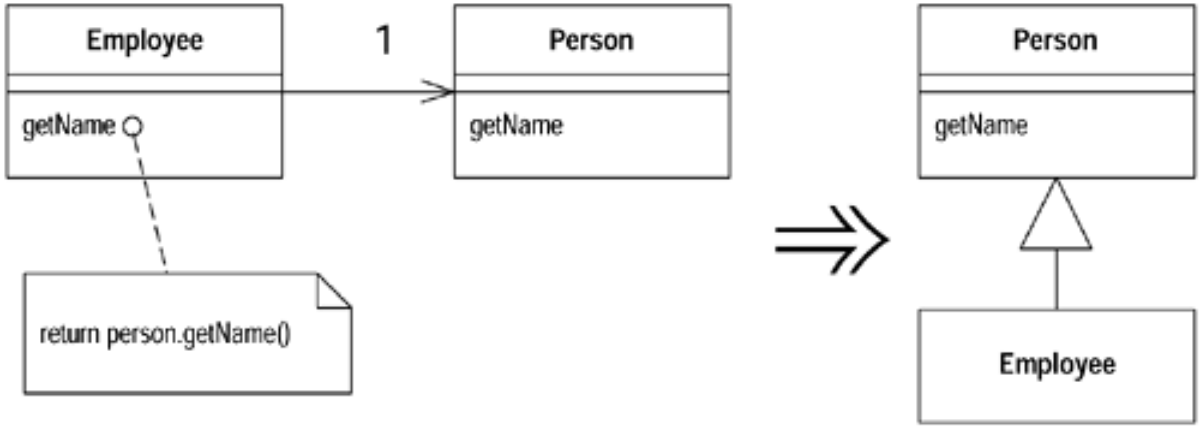


[http://sourcemaking.com/]

Exemplo

Replace Delegation with Inheritance

- You're using delegation and are often writing many simple delegations for the entire interface
- Make the delegating class a subclass of the delegate



[http://sourcemaking.com/]

Porquê refactorizar ?

- Melhoria contínua
- Revisão de software
- Detecção de erros
- Facilidade de compreensão e comunicação
- Melhoria da eficácia de desenvolvimento
- Aumento da qualidade

Refactorização

- Quando refactorizar ?
  - Alteração de funcionalidade
  - Correcção de erros
  - Realização de revisões
- Perspectiva imediata ⟵ **Miopia do futuro**
- Perspectiva futura

Acumulação combinatória de complexidade
(**efeito exponencial**)

Refactorização

- Quando não refactorizar...
  - Qualidade de arquitectura / código muito baixa
    - Refazer arquitectura
    - Reescrever código
  - Próximo do final do prazo de entrega
- Modularidade e Encapsulamento
  - Localização das intervenções

Refactorização

- Problemas
  - Bases de dados
    - Dependências do esquema da base de dados
    - Dificuldade de refactorizar código de acesso a dados
    - Camada de acesso a dados
      - Controlo da complexidade
      - Flexibilidade
      - As especificidades da organização dos dados apenas têm impacto na camada de acesso a dados

Refactorização

- Problemas
  - Alteração de interfaces
    - Necessário acesso a todo o código que utiliza a interface
    - Por exemplo, alterar o nome de um método
      - Refactorização *Renomear Método*
    - Não publicar interfaces prematuramente
    - *Java*: adicionar excepções à cláusula *throws*
      - Assinatura da interface mantém-me mas...
      - Problemas de compilação

Refactorização

- Problemas
  - Opções de arquitectura incorrectas
    - Alterar a arquitectura global
    - Manter a arquitectura global
      - Intervir localmente tendo em conta os problemas identificados
      - Documentar os problemas para alterações futuras

Indicadores de Problemas de Arquitetura

- **Rigidez**
  - Dificuldade de alteração do sistema, porque qualquer alteração afecta múltiplas partes do sistema
- **Fragilidade**
  - Quando se faz uma alteração numa parte do sistema outras partes deixam de funcionar de forma inesperada
- **Imobilidade**
  - Dificuldade de reutilização porque não é possível separar as partes entre si (existe uma teia de dependências entre as partes do sistema)

Problemas subjacentes

- **Coesão baixa**
- **Acoplamento alto**
- **Complexidade desorganizada**

Anti-Padrões de Software

- **Anti-padrões**
  - Tal como os padrões, definem um vocabulário para formas e métodos de arquitectura, mas neste caso **potenciadores de problemas**
  - Disponibilizam experiência típica de projectos reais para **reconhecer problemas antes que estes possam comprometer o sucesso** de um projecto
  - Directamente relacionados com a **refactorização** de software

Anti-Padrões de Software

Methodological anti-patterns

- **Copy and paste programming**: Copying (and modifying) existing code rather than creating generic solutions
- **Programming by permutation (or "programming by accident")**: Trying to approach a solution by successively modifying the code to see if it works
- **Premature optimization**: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- **Golden hammer**: Assuming that a favorite solution is universally applicable (See: Silver Bullet)
- **Reinventing the wheel**: Failing to adopt an existing, adequate solution
- **Reinventing the square wheel**: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
- **Silver bullet**: Assuming that a favorite technical solution can solve a larger process or problem

Anti-Padrões de Software

Software design anti-patterns

- **Accidental complexity**: Introducing unnecessary complexity into a solution
- **Pattern programming**: Using patterns and methods without understanding why
- **Coding by exception**: Adding new code to handle each special case as it is recognized
- **Error hiding**: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- **Hard code**: Embedding assumptions about the environment of a system in its implementation
- **Lava flow**: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences
- **Loop-switch sequence**: Encoding a set of sequential steps using a switch within a loop statement
- **Magic numbers**: Including unexplained numbers in algorithms
- **Magic strings**: Including literal strings in code, for comparisons, as event types etc.

Anti-Padrões de Software

Software design anti-patterns

- **Duplicate Code**: Code that is repeated - associated to the Copy and Paste anti-pattern
- **Spaghetti code**: Code that is very long and difficult to understand
- **Input kludge**: Failing to specify and implement the handling of possibly invalid input
- **Race hazard**: Failing to see the consequence of different orders of events
- **Stovepipe system**: A barely maintainable assemblage of ill-related components
- **Circular dependency**: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- **Sequential coupling**: A class that requires its methods to be called in a particular order
- **Excessive fragmentation** : A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation
- **Blob**: A class with a lot of attributes and methods

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.

[Dijkstra, 1968]

E. Dijkstra, *The Structure of the 'THE' Multiprogramming System*, Communications of the ACM 11 – 5, 1968.

[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. Chatterjee; *Messaging Patterns in Service-Oriented Architecture – Part 1*, Cap Gemini Ernst & Young - MSDN, 2004.

[Booch, 2004]

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