Engenharia de Software

Métodos de Desenvolvimento Ágil

Luís Morgado

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

Métodos de Desenvolvimento Ágil

No contexto do desenvolvimento ágil de software foram integrados e adaptados diferentes métodos de desenvolvimento, partindo de métodos já existentes, no sentido de aumentar a eficiência em termos de recursos e de tempo necessários, e a flexibilidade no sentido de maior facilidade de adaptação à incerteza e à necessidade de mudança, nomeadamente:

- **Histórias de utilizador** (*User Stories*)
- **Protótipos exploratórios** (*Spikes*)
- **Desenvolvimento guiado por testes** (*Test-Driven Development*)
- **Programação em pares** (*Pair Programming*)
- **Refactorização** (*Refactoring*)

Histórias de Utilizador (*User Stories*)

- As *histórias de utilizador* (*user stories*) são descrições informais, em linguagem natural, da **forma como os utilizadores podem concretizar os seus objectivos** de utilização do sistema
- Têm por **objectivo tornar mais expedita a identificação e especificação das funcionalidades requeridas pelos utilizadores**, sendo utilizadas para descrever as necessidades dos utilizadores de forma concisa e focada no valor entregue
- Desempenham um papel importante no processo de desenvolvimento, no sentindo de manter o **foco nas necessidades reais dos utilizadores** e facilitando a comunicação entre a equipa de desenvolvimento e as partes interessadas no sistema (*stakeholders*), nomeadamente cliente e utilizadores
- As histórias de utilizador, são **inicialmente utilizadas para descrever os requisitos de um sistema de maneira simples, concisa e não detalhada**, servindo para manter um registo das necessidades dos utilizadores

Histórias de Utilizador (*User Stories*)

Apesar do seu carácter informal, **uma história de utilizador deve ter um formato de organização definido**, podendo ser descrita com diferentes níveis de detalhe

Exemplo:

História de utilizador inicial com menor nível de detalhe

Título:

Adicionar uma tarefa ao sistema de gestão de projectos

Participante:

Gestor de projecto

Descrição:

O gestor de projecto deseja adicionar uma tarefa ao sistema de gestão de projectos para que a tarefa possa ser gerida e acompanhada.

Histórias de Utilizador (*User Stories*)

- **Nas reuniões de planeamento dos ciclos de desenvolvimento** as histórias de utilizador **são discutidas e detalhadas**, sendo incluídos **critérios de aceitação** que definem quando a história está concluída
- Para efeitos de organização das histórias de utilizador, podem ser utilizados critérios de avaliação da sua relevância, por exemplo, se uma história é **independente, útil, negociável, estimável, pequena ou testável**
- Ao longo dos ciclos de desenvolvimento as histórias de utilizador são **detalhadas em tarefas concretas e requisitos específicos**, no sentido de facilitar a **compreensão das necessidades dos utilizadores** e o estabelecimento de **critérios claros de aceitação** para o desenvolvimento da respectiva funcionalidade

Histórias de Utilizador (*User Stories*)

Exemplo:

História de utilizador com maior nível de detalhe

Título:

Adicionar uma tarefa ao sistema de gestão de projetos

Participante:

Gestor de projeto

Descrição:

O gestor de projeto deseja adicionar uma tarefa ao sistema de gestão de projectos para que a tarefa possa ser gerida e acompanhada.

História:

1. O gestor de projeto seleciona o projecto em curso.
2. O gestor selecciona a opção "Adicionar tarefa" e preenche o formulário com as informações da tarefa, como título, descrição, prazo e responsável.
3. O gestor pressiona o botão “Guardar" para guardar a tarefa no sistema.
4. O sistema confirma a inclusão da tarefa e apresenta essa tarefa na lista de tarefas do projecto.

Critérios de aceitação:

1. A tarefa deve ser adicionada ao projecto.
2. A tarefa deve ser apresentada na lista de tarefas do projecto.

Histórias de Utilizador (*User Stories*)

Padrões comuns de histórias de utilizador

Os formatos das histórias são organizados no sentindo de manter o foco nas necessidades dos utilizadores

Padrão dos 3 Rs

As a <role - user>
I want <requirement - output>
so <reason - outcome>

O terceiro R (*reason*) é opcional

Padrão dos 5 Ws

As <who> <when> <where>
I want <what>
because <why>

Histórias de Utilizador (*User Stories*)

Exemplo: Padrão dos 3 Rs

Formato das histórias

- **Titulo**
  - Designação da história de utilização
- **Utilizador ou papel** (“*como* …”)
  - Utilizador ou papel envolvidos na história de utilização
- **O que se pretende** (“*quero* …”)
  - Como o objectivo do utilizador vai ser concretizado
- **O que se obtém** (“*tal que* …”)
  - Resultado a obter com a realização da história de utilizador

Exemplo de história de utilizador:

Título
Transferir dinheiro entre contas

Como Titular de conta

Quero rever os saldos das minhas contas e realizar a transferência de uma determinada quantia entre contas

Tal que consiga realizar a transferência e possa observar os novos saldos nas contas envolvidas na transferência

Independentemente da forma de organização, **quanto maior o rigor e precisão da descrição** de uma história de utilizador, **mais clara e eficiente será a sua realização**

Protótipos Exploratórios (*Spikes*)

Protótipos de arquitectura e de implementação **elaborados de forma expedita, num tempo curto, com o objectivo de reduzir a incerteza** acerca das soluções a desenvolver

Têm como objectivos:

- Testar a implementação de uma pequena parte da solução ou da arquitetura global da aplicação
- Possibilitar o estudo de aspectos técnicos de uma parte específica da solução
  - Validar os pressupostos técnicos
  - Escolher entre possíveis arquitecturas e estratégias de implementação

Desenvolvimento Guiado por Testes

(*Test-Driven Development* - TDD)

Método de desenvolvimento que implica **escrever primeiro casos de teste** e depois implementar o código necessário para passar nos testes

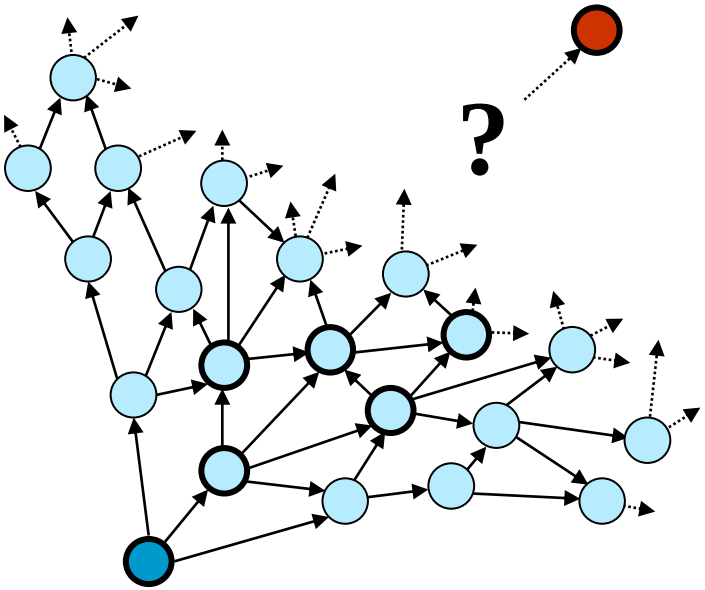
Tem como objectivo:

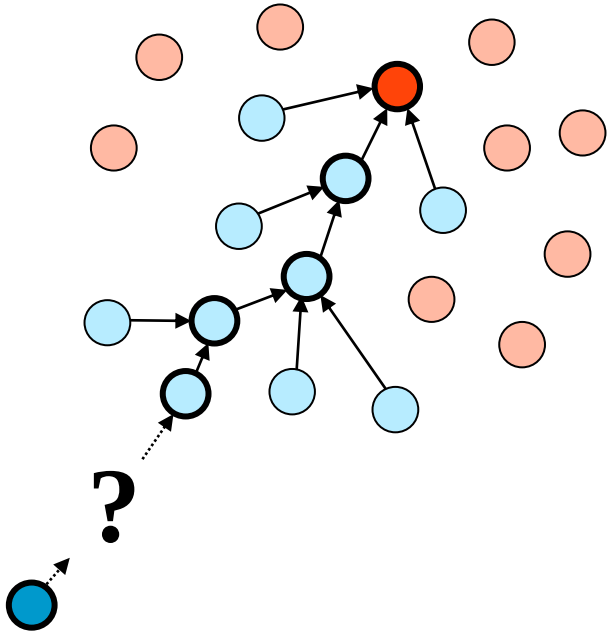
- Possibilitar um desenvolvimento orientado para a **obtenção informação (*feedback*) de forma expedita e frequente** acerca da forma como as partes do sistema satisfazem os respectivos requisitos durante o desenvolvimento do sistema

Desenvolvimento Guiado por Testes

(*Test-Driven Development* - TDD)

- Testes são escritos antes de ser escrito o código
- Testes servem para verificar se alterações não introduziram erros





Desenvolvimento clássico:

A forma de concretizar os requisitos pode ser difusa de início e, por isso, difícil de identificar a melhor forma de realizar essa concretização (arquitectura, implementação)

Desenvolvimento guiado por testes:

Testes concretizam os requisitos como ponto de partida, pelo que a forma de os atingir (satisfazer com sucesso os testes) é mais fácil de identificar

Desenvolvimento Guiado por Testes

(*Test-Driven Development* - TDD)

Problemas:

- Os testes criados num ambiente de desenvolvimento guiado por testes são, frequentemente, **criados pelo mesmo programador que desenvolveu o código que é testado**:
  - Os testes podem partilhar erros com o código a testar
  - Por exemplo, se o programador interpretar mal os requisitos para o módulo que está a desenvolver, o código e os testes estarão ambos errados
    - **Os testes terão sucesso, dando uma falsa sensação de correção**
- **Um número elevado de testes iniciais pode dar uma falsa sensação de segurança**, resultando em menos actividades adicionais de verificação e teste, por exemplo, testes de integração
  - Os testes tornam-se uma sobrecarga na manutenção de um projeto
- **A necessidade de uma alteração de requisitos** ou de arquitectura pode levar a que múltiplos testes definidos inicialmente falhem
  - A eliminação ou alteração precipitada desses testes pode levar a lacunas na cobertura dos testes, não facilmente detectáveis

Programação em Pares (*Pair programming*)

Dois programadores participam em simultâneo num desenvolvimento conjunto

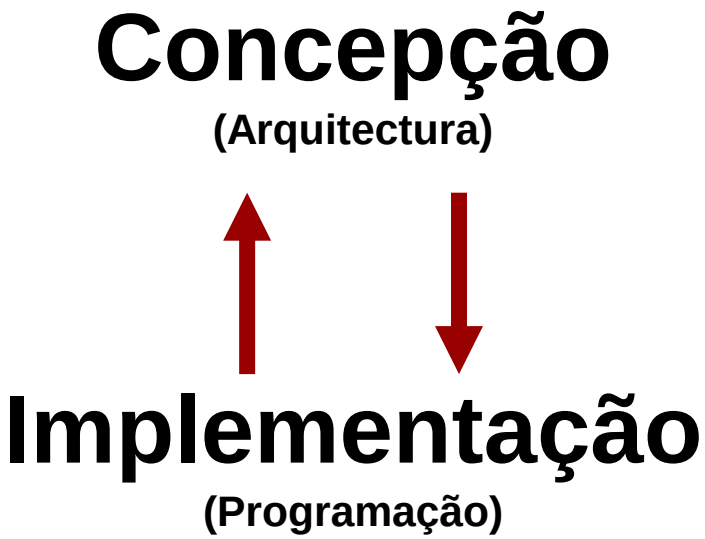
- A pessoa que está a escrever o código é designada por *condutor*, a pessoa que está a observar é designada por *observador* ou *navegador*
- Os dois programadores trocam de papéis com relativa frequência (entre meia-hora a uma hora)
- Enquanto o *observador* revê o trabalho do *condutor*, considera também uma perspectiva mais abrangente (*estratégica*) do trabalho que está a ser realizado
  - Considera e apresenta ideias para possíveis melhorias
  - Considera possíveis problemas futuros a resolver
  - O objetivo é libertar o *condutor* de modo a que este possa concentrar a sua atenção nos aspectos de curto prazo (*tácticos*) da realização da tarefa atual
  - O *observador* serve como suporte de segurança e orientação
- A programação em pares melhora significativamente a qualidade do software produzido e reduz o tempo total de desenvolvimento
- No entanto, pode aumentar o número de horas-pessoa necessárias para produzir código, em comparação com dois programadores que trabalham individualmente

Refactorização (*Refactoring*)

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

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

[Alistair Cockburn]



No desenvolvimento de software é necessário um processo iterativo de melhoria progressiva e contínua, de modo a garantir a convergência para uma solução de qualidade

Refactorização

A baixa qualidade da arquitectura e da implementação tem um impacto determinante no esforço de desenvolvimento, nomeadamente, na capacidade para adaptação à mudança

- Arquitectura e implementação que são **difíceis de compreender** são **difíceis de alterar**
- Arquitectura e implementação com **baixa coesão e alto acoplamento** são **difíceis de alterar**
- Arquitectura e implementação com **redundância** são **difíceis de alterar**
- Arquitectura e implementação com **estrutura ou dinâmica interna complexas** são **difíceis de alterar**

Refactorização

Ao longo do desenvolvimento o **nível de entropia do sistema em produção cresce** sob a forma de opções de arquitectura e de implementação menos adequadas, imperfeitas ou incorrectas

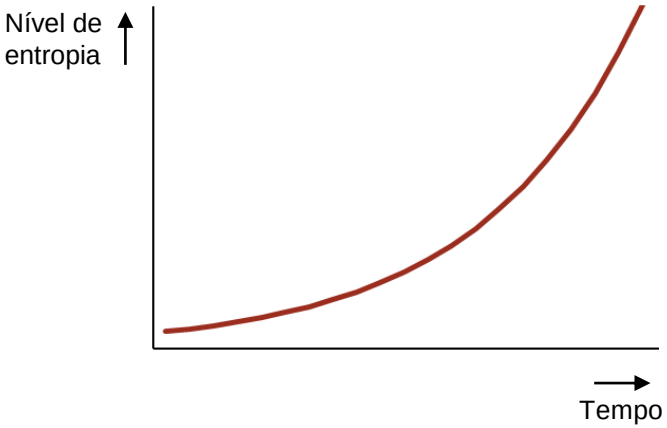
O efeito dessas opções acumula-se de forma combinatória produzindo um crescimento exponencial da entropia, se esse crescimento não for limitado o desenvolvimento pode tornar-se insustentável, ou mesmo colapsar, devido à complexidade desorganizada que se encontra latente na arquitectura e implementação

O efeito observável dessa complexidade é o **surgimento crescente de erros e defeitos**, pontualmente e parcialmente detectados nas actividades de teste

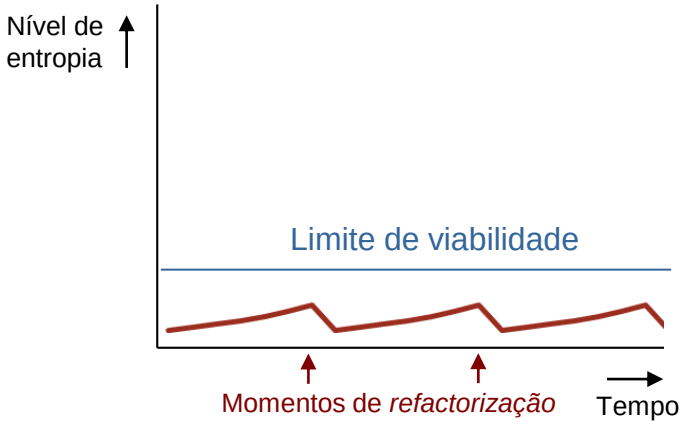
A *refactorização* tem como efeito limitar o crescimento combinatório da entropia através da sistemática revisão e melhoria de partes do sistema em desenvolvimento, sem alterar o seu comportamento observável

Desta forma, o **nível de entropia (complexidade desorganizada) é mantido dentro de limites de viabilidade** e de desenvolvimento sustentável do sistema

Desenvolvimento não sustentável
Crescimento exponencial da entropia



Desenvolvimento sustentável
Nível de entropia mantido dentro de limites de viabilidade



Refactorização

- **A *refactorização* não tem por objectivo corrigir erros nem acrescentar novas funcionalidades**
- Tem por objectivo **melhorar a arquitectura, compreensibilidade e adaptabilidade** do software em desenvolvimento, **sem alterar o seu comportamento observável**, para facilitar o desenvolvimento e manutenção no futuro
- Duas actividades principais de desenvolvimento:
  - **Adicionar funcionalidade**
  - **Refactorizar**

Refactorização

- **Tipos de refactorização**
  - Refactorização de subsistemas
  - Refactorização de mecanismos
  - Refactorização de classes
  - Refactorização de métodos
  - Refactorização de dados
  - Refactorização de expressões
  - Refactorização de hierarquia de herança
  - Refactorização geral

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 ?**
  - Antes de alteração de funcionalidade
  - Antes ou durante a correcção de erros
  - Após a realização de revisões

- Perspectiva imediata
  - Desenvolver e refactorizar mais tarde

- Perspectiva futura
  - Refactorizar para desenvolver de forma sustentada

← **Miopia do futuro**

↓

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

Refactorização

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

Refactorização

- **Problemas ao refactorizar**
  - 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 ao refactorizar**
  - 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*
      - O contrato funcional definido pela interface mantém-se, mas o contrato de processamento de excepções é alterado
      - Problemas de compilação

Refactorização

- **Problemas ao refactorizar**
  - Opções de arquitectura incorrectas
    - Alterar a arquitectura global
    - Manter a arquitectura global
      - Intervir localmente tendo em conta os problemas identificados
      - Documentar os problemas identificados 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)

Padrões de Arquitectura: Problemas

- **Críticas**
  - Desvio de foco
    - Abordagem do problema a níveis de abstracção inadequados
    - Muitos dos padrões propostos têm uma aplicação marginal
  - Ausência de suporte formal
    - O estudo de padrões de arquitectura tem sido largamente *ad hoc*
    - Numa análise feita no OOPSLA'99 aos padrões GoF (com a colaboração dos autores) foram identificados múltiplos problemas nesses padrões
  - Propiciam soluções ineficientes
    - Podem levar à produção desnecessária de código
    - É tipicamente mais eficiente a utilização de uma solução bem concebida do que a utilização de um padrão razoavelmente adequado
  - Não diferem significativamente de outras abstracções
    - Em muitos casos não diferem significativamente de outras formas de abstracção
    - A utilização de nova terminologia para descrever algo já conhecido é fonte de confusão

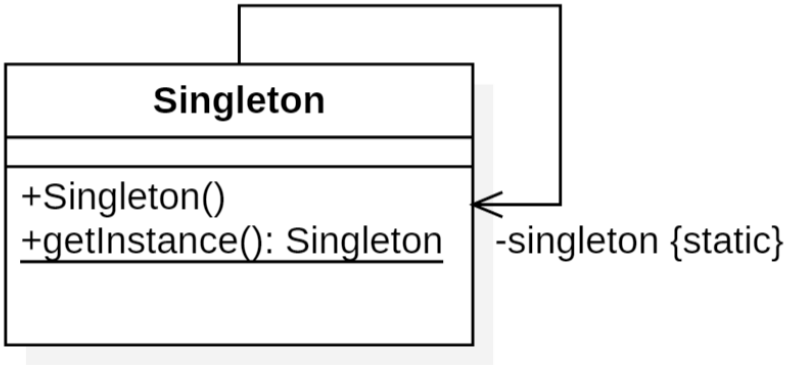
Exemplo: Padrão *Singleton*

Problema

Como garantir que é criada apenas uma instância de uma classe, tornando-a disponível a todo o código

Solução

Utilizar uma classe *Singleton* para garantir que é criada apenas uma instância da classe e assim fornecer um ponto de acesso global a essa instância



Padrão *Singleton*

- **Problemas**
  - Viola o **princípio da responsabilidade única** (*single responsibility principle*) na medida em que a classe Singleton controla a sua criação e o seu ciclo de vida
    - **Baixa coesão**
  - Introduz **estado global** numa aplicação
    - **Elevado acoplamento**
  - **Anti-Padrão**
    - Padrão potenciador de problemas

Anti-Padrões de Software

- 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
- Disponibilizam **soluções** para abordar as situações identificadas
- Directamente relacionados com a ***refactorização***

Exemplos de Anti-Padrões de Software

Anti-padrões de método de desenvolvimento

- **Programação por copiar e colar** (***Copy and paste programming***): Copiar e modificar código existente em vez de criar soluções adequadas gerais
- **Programação por permutação** (***Programming by permutation***): Tentar chegar a uma solução modificando sucessivamente o código por tentativa-erro para ver se funciona
- **Optimização prematura** (***Premature optimization***): Desenvolver desde o início para obter uma aparente eficiência, sacrificando a boa arquitectura, a facilidade de manutenção e, por vezes, até a eficiência efectiva
- **Martelo dourado** (***Golden hammer***): Assumir que uma solução conhecida é universalmente aplicável

Exemplos de Anti-Padrões de Software

Anti-padrões de método de desenvolvimento

- **Reinventar a roda** (***Reinventing the wheel***): Não adoptar uma solução existente e adequada, optando por adotar uma solução personalizada
- **Reinventar a roda quadrada** (***Reinventing the square wheel***): Não adotar uma solução existente e, em vez disso, adotar uma solução personalizada com um desempenho muito pior do que a solução existente
- **Bala de prata** (***Silver bullet***): Assumir que uma solução técnica favorita pode resolver um problema de âmbito mais alargado

Exemplos de Anti-Padrões de Software

Anti-padrões de organização de software

- **Código duplicado** (***Duplicate code***): Código que se repete, nomeadamente associado ao anti-padrão *copiar e colar*
- **Código esparguete** (***Spaghetti code***): Código intrincado, muito longo e difícil de entender
- **Confusão de entradas** (***Input kludge***): Não especificar e implementar adequadamente o tratamento de entradas, em particular de entradas possivelmente inválidas
- **Risco de ocorrência desordenada** (***Race hazard***): Não ter em conta as consequências de diferentes ordens de ocorrência de eventos
- **Sistema em chaminé** (***Stovepipe system***): Um conjunto de componentes mal relacionados que se vão acumulando e que originam esforço adicional na sua manutenção

Exemplos de Anti-Padrões de Software

Anti-padrões de organização de software

- **Dependência circular** (***Circular dependency***): Existência de dependências recíprocas directas ou indirectas desnecessárias entre objectos ou módulos de um sistema
- **Acoplamento sequencial** (***Sequential coupling***): Classes que requerem que os seus métodos sejam evocados numa determinada ordem
- **Blob**: Classe com muitos atributos e métodos em resultado de um nível baixo de coesão
- **Fragmentação excessiva** (***Excessive fragmentation***): Arquitectura com fragmentação excessiva, originando o aumento da complexidade do sistema, por exemplo em hierarquias de herança, tornando a respectiva organização difícil de compreender

Bibliografia

[Watson, 2008]
Andrew Watson, *Visual Modeling: past, present and future*, OMG, 2008.

[Meyer, 1997]
B. Meyer, *UML: The Positive Spin*, American Programmer - Special UML issue, 1997.

[Ambler & Lines, 2011]
S. Ambler, M. Lines, *UML: Disciplined Agile Delivery*, IBM, 2011.

[Selic, 2003]
B. Selic, *Brass bubbles: An overview of UML 2.0*, Object Technology Slovakia, 2003.

[Graessle, 2005]
P. Graessle, H. Baumann, P. Baumann, *UML 2.0 in Action*, Packt Publishing, 2005.

[Eriksson et al., 2004]
H. Eriksson, M. Penker, B. Lyons, D. Fado, *UML 2 Toolkit*, Wiley, 2004.

[USDT, 2005]
U.S. Department of Transportation, *Clarus: Concept of Operations*, Publication No. FHWA-JPO-05-072, 2005.

[Douglass, 2006]
B. Douglass, *Real-Time UML*, Telelogic, 2006.

[OMG, 2020]
Unified Modeling Language (Specification), OMG, 2020.