PDF N°13

From Models, to Stories to Architecture

Os modelos são importantes mas, não são o objetivo

Devemo-nos focar em:

1. Satisfação do cliente / stakeholder
   - prestando apoio
2. Entrega do serviço
   - Arquitetura do software: componentes e instalação
3. Qualidade
   - Lidar com trocas (atributos de qualidade)
   - Testar
4. seguir um processo ...

1. satisfação do cliente / stakeholder

O foco é o cliente!

Smart requirements

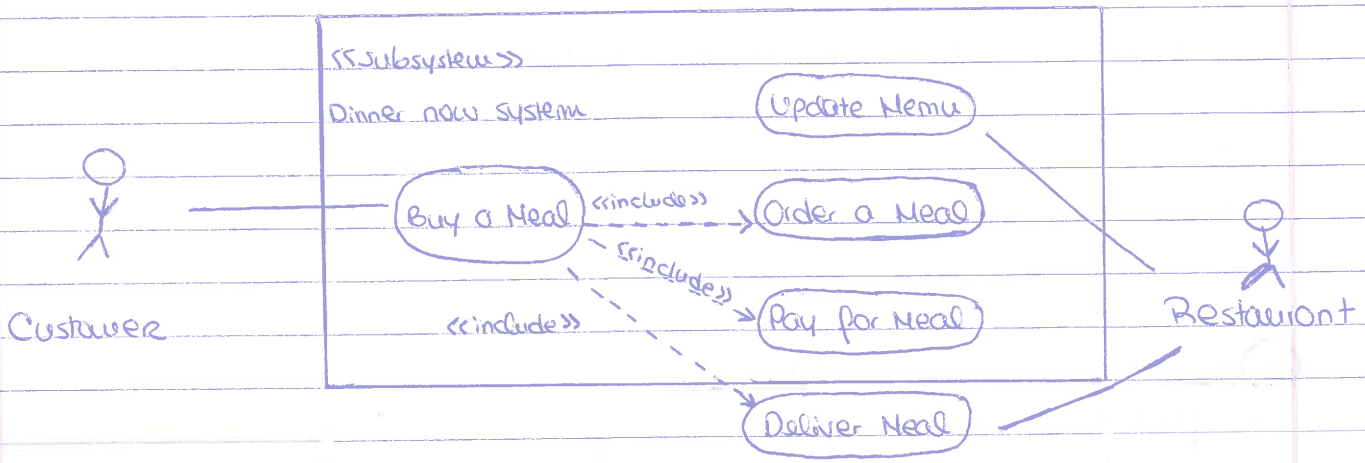
- Specif (Precise) - Específico (exato)
- Measurable - Mensurável
- Attainable (Achievable, Actionable, Appropriate) - Alingível (Realizável, Acionável, Apropriado)
- Realistic (Relevant) - Realista (Relevante)
- Time-Bound (Timely) - Com limite de tempo (Oportuno)

② Entrega do Serviço

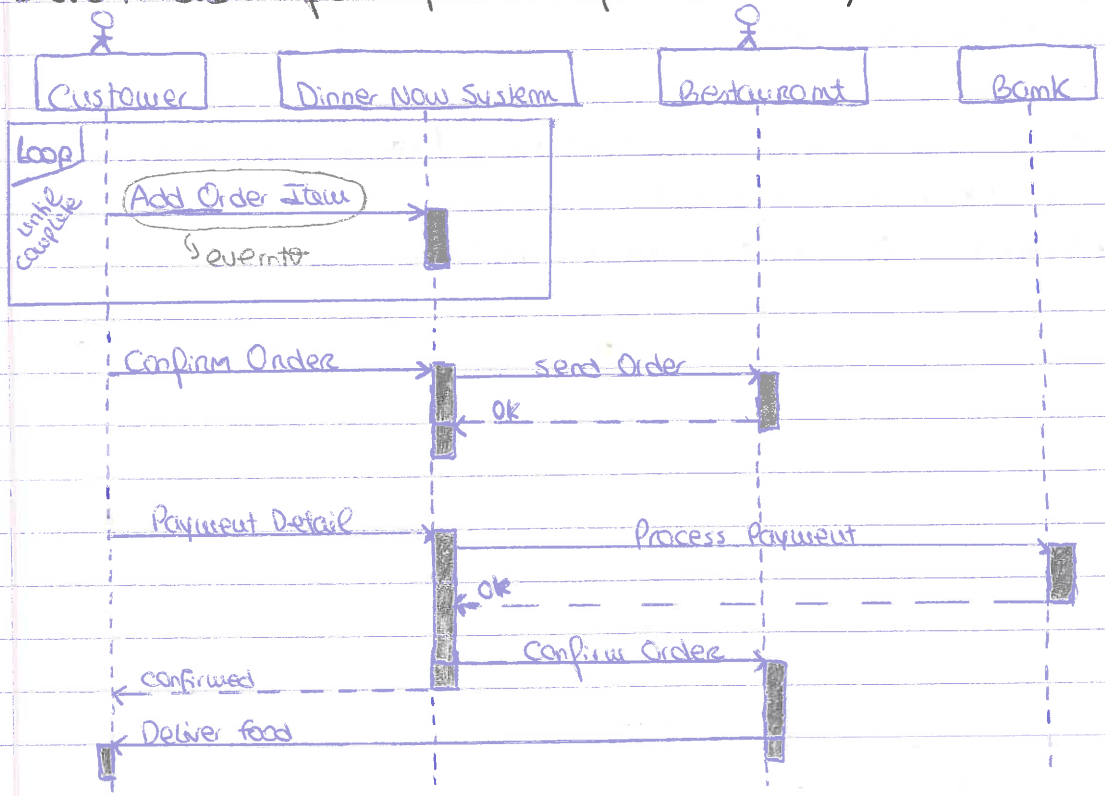
- Concluído
  - Histórias, Cenários e recursos identificados
  - Requisitos e Riscos identificados

- Desenvolver
  - O quê? } Arquitetura e
  - Onde? } Implementação
  - Qualidade?

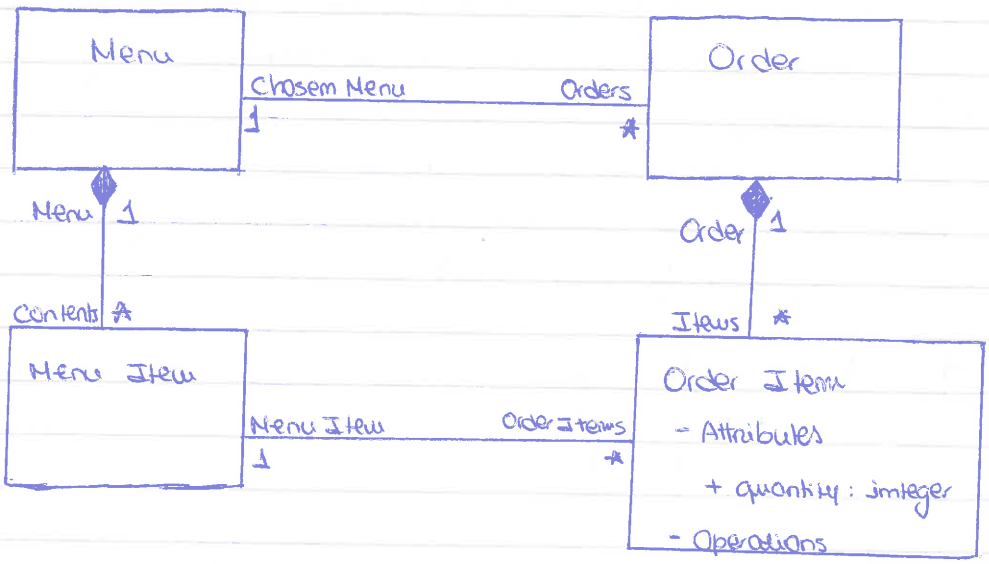
Os use case do sistema não são os mesmos para cada subsistema



Deve modelar o fluxo para identificar os dados / serviços



A informação contido no sistema com regra de negócios para todos



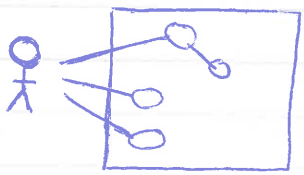
Architectual Requirements

Requisitos arquitetonicos são um subconjunto dos requisitos do sistema, determinado pela relevância arquitetonica.

Os objetivos comerciais para o sistema, a arquitetura em particular, são importantes para assegurar que a arquitetura está alinhada com a 'agenda de negócios'.

O contexto do sistema ajuda a determinar o que está dentro e fora do âmbito/alcance, o que a interface do sistema é, e quais os fatores que incidem sobre a arquitetura.

Os objetivos do sistema são traduzidos num conjunto de Use Cases que são usados para documentar os requisitos funcionais.

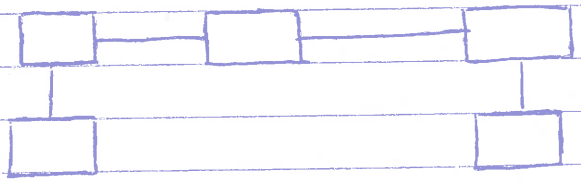


↳ System - wide Qualities
- non-functional requirements

↳ Use Cases
- functional Requirements

Conceptual Architecture

O sistema é decomposto em componentes (elementos estruturais), responsabilidades de cada componente e as interligações entre os componentes são identificados.
A intenção da arquitetura conceitual é para dirigir a atenção a uma decomposição adequada do sistema, sem se aprofundar nos detalhes de interface. Além disso, fornece um veículo útil para comunicar a arquitetura para o público em geral (não técnica), como gestão, marketing, entre outros.

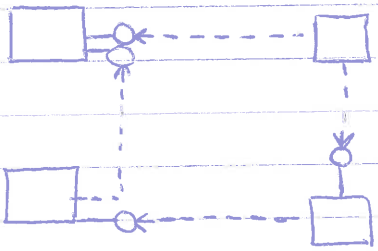


↳ Conceptual Architecture
Component Responsibilities
System-Wide Characteristics
Communicating Up & Out

Abstract

Logical Architecture

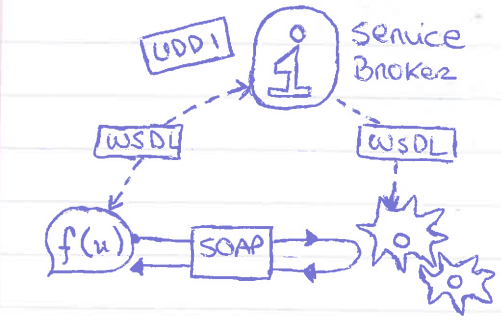
A arquitetura lógica é a especificação detalhada, definindo com precisão as interfaces de componentes e mecanismos e protocolos de conexão. Ele também fornece informações contextuais sobre os componentes e como eles devem ser usados em sistemas de montagem.
A arquitetura lógica é usada pelos designers e desenvolvedores, bem como usuários de componentes (aqueles que montam componentes em sistemas)



↳ Logical Architecture
Component Interfaces
Actionable
Unambiguous
Complete

Precise

No Prática



Mas por onde se começa?

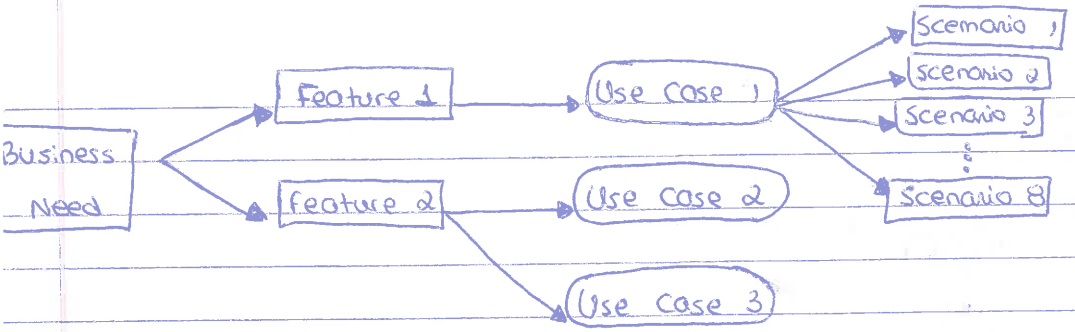
- Recursos
- Funções do sistema

As funções do sistema são blocos de construção

- Bloco de construção do sistema
- O sistema precisa de um modelo
  - como as funções do sistema se encaixam
- Apoiar a implementação do sistema
  - Requisitos
  - Fluxos
  - Informações

Funções do sistema

- "Serviços" necessários a partir do nosso sistema pelo sistema externo e / ou atores
- Responsabilidade com o mundo externo
- Identificação é baseado em Use Cases e Requisitos
  - Descrição e interação com o exterior
  - Pode ajudar a identificar se as funções do sistema funcionam
  - Normalmente essas funções podem ser "compartilhadas" entre os Use Cases
- Mudança Arquitetura → Engenharia
  - Mudança de Use Cases para funções do sistema
  - A implementação está dependente dos use cases



Onde colocá-los?

↳ Em componentes que estão em nós com a arquitetura do software do sistema

DF N°11

O que é um bom software?

O objetivo da engenharia de software é produzir um bom software

↳ funcionar em todos os PCs

↳ funcionar todos os dias

③ Qualidade

Atributos de Qualidade (ISO/IEC 9126)

Qualidade Externa e Interna

- Portabilidade

Adaptabilidade, Instabilidade, Co-Existência, Substituição, portabilidade, conformidade

- Manutenção

Capacidade de análise, Mobilidade de mudança, Estabilidade, Capacidade de Teste, manutenção, conformidade

- Usabilidade

Compreensibilidade, Capacidade de aprendizagem, Operabilidade, Atratividade, Usabilidade, conformidade

- Confiança

Maturidade, tolerância a falhas, capacidade de recuperação, confiança, conformidade

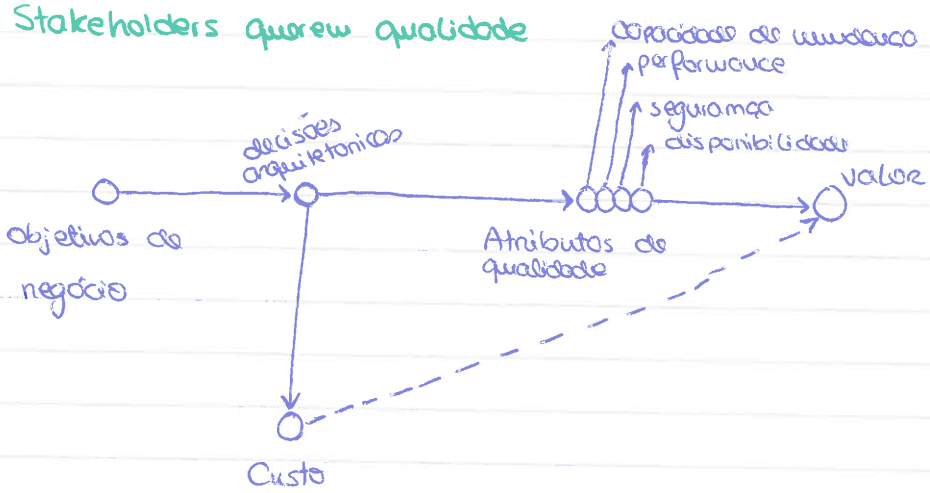
- Eficiência

"Comportamento do tempo", Recurso, utilização, eficiência, conformidade

- Funcionalidade

Adequação, precisão, interoperabilidade, segurança, Funcionalidade, conformidade

Stakeholders querem qualidade



Arquitetura e Design

→ Estilo Arquitetônico (padrão) → Castelo construído
↳ modelo

Um padrão de arquitetura expressa um esquema de organização estrutural fundamental para sistemas de software. Ele fornece um conjunto de subsistemas predefinidos, as suas responsabilidades, e inclui regras e diretrizes para a organização das relações entre eles.

→ Padrões de Projeto → Planta do Castelo

Um padrão de design fornece um esquema para aperfeiçoar os subsistemas ou componentes de um sistema de software ou a relação entre eles.
Descreve uma estrutura que resolve um problema de projeto dentro de um contexto particular.

Conhecendo Padrões e Estilos

- Ajuda a definir testes
- Definem um mecanismo
  - entre elementos: componentes, classes
  - define interação e trocas
- Vantagens
  - já conhecidos
- Bom para testar o teu sistema
  - Independentemente do âmbito do trabalho

Estilos Arquitetónicos

- Comunicação

  Arquitetura Orientada a Serviços (SOA), troca de mensagens

- Deployment (Implementação)

  Cliente/servidor, N-Tier, 3-Tier

- Domain (Domínio)

  Domain Driven Design

- Estrutura

  Baseado em componentes, Orientado a Objetos, Arquitetura em camadas

páginas 20, 21

Padrões de Projeto

Wrapper / wrappee ou Delegation

Adapter - Agrupa um objeto que provém de uma interface incompatível com o objetivo de suportar a interface desejada.

Facade - Agrupa um subsistema com um objeto que provém duma simples interface

Proxy - Agrupa um objeto com um objeto substituto que fornece funcionalidades adicionais.

Hierarquia de Heranca

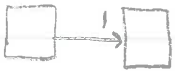
Strategy - Define uma interface com base numa classe base e implementações em classes derivados.

Factory Method - Define "createInstance", espaço reservado na classe base. Cada classe derivada chama o operador "new" e retorna uma instância de si mesmo.

Visitor - Define "accept" na hierarquia primaria. Define "visit" na hierarquia secundária, conhecido como "double dispatch".

A wrapper wraps an inheritance hierarchy

Builder – O "reader" elege um construtor. Cada construtor corresponde a uma representação ou destino diferente.



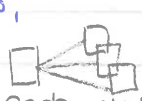
State – O finiteStateMachine elege o estado "atual" do objeto e, esse estado do objeto pode definir o "next" estado do objeto.

??? Bridge – Os modelos de wrapper "abstração" e os modelos wrappee muitos possíveis "implementações". O wrapper pode usar herança para opoior abstração especialização

Observer – O "modelo" transmite para possíveis "views", e cada "view" pode dialogar com o "modelo".

Miscellaneous (Diversos)

Abstract factory – Modelo de "plataforma" (por exemplo: sistema operativo, base de dados) com uma hierarquia de herança e modelo cada "produto" (por exemplo: widgets, serviços, estrutura de dados) com a sua hierarquia



Template Method – Define o "outline" de um algoritmo em uma classe base. Implementação comum é testada na classe base, implementação peculiar é representada por "place holders" na classe base e, em seguida, implementada em classes derivadas.

??? Flyweight – Quando dezenas de instâncias de uma classe são desejadas e desempenho "baggs" para baixo, exteriorizar o estado do objeto que é peculiar em cada instância, e exigir que o cliente passe nesse estado quando os métodos são invocados.

Singleton – Idealizar uma classe para encapsular uma única instância de si mesma, "lock out" clientes de criar os seus próprios instâncias.

Exemplo: Problemas e soluções...

→ Soluções

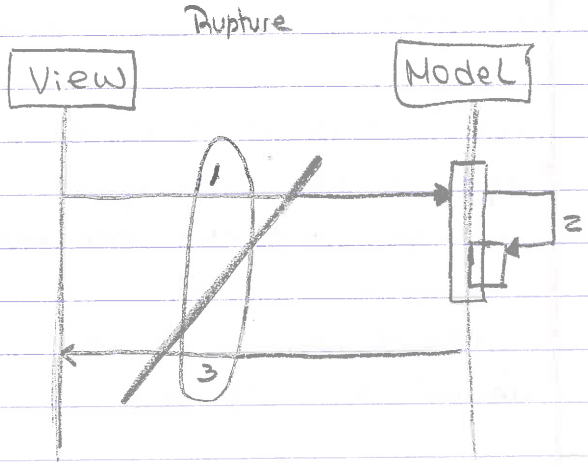
- Coupling
  Indirecção através API padrão
  Exemplo: Web Service
- Concurrency
  Gerido no tempo de execução
- Singleton
  Bom para o controlo
- Scope Control
  Local VS Remoto
  Âmbitos despazados

→ Problemas

- Coupling
  "Overhead?"
  Exemplo: Gerir a conversão de formatos de dados
- Concurrency
  Gestão
- Singleton pattern
  Um possível estrangulamento
- Scope Control
  Local VS Remoto
  "Overhead" em comunicações locais

... diferentes quando abordam acessibilidade

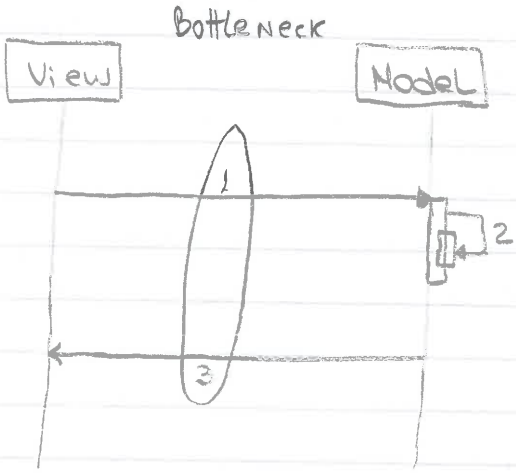
- Communications
  - J2EE:
    - tipicamente TCP/IP
    - Depende do "naming service"
  - Android:
    - TCP/IP UDP
    - SMS, GMS, 3G, 4G
- Autonomy
  - J2EE:
    - não é uma opção
  - Android:
    - Pode ser autonomo
    - Bateria



... diferentes em número de utilizadores

- Number of Users

- J2EE: Muitos
- Android: Um



- User Interface
  - Diferentes Expectativas
  - Âmbito / Alcance diferentes
  - J2EE: mundo externo
    - web (http)
    - persistência (JDBC, JPA)
  - Android: o dispositivo / o utilizador
    - "Responsive"
    - Tráfego Web "not excuse"

... Têm um impacto nítido

- Number of Users
- Reachability
  - É necessário acessibilidade
  - Crucial para J2EE
    - Normalmente TCP/IP
    - "Naming Service"
  - Relevante para Android
    - TCP/IP
    - SMS / GSM, 3G, 4G...
    - Pode ser autónomo

J2EE: Trade-offs para lidar com muitos utilizadores

- Concurrency

É necessário lidar com vários "requests" ao mesmo tempo

→ Performance: Depende da noção de tempo, caching

- Availability

Deve ser acessível e receptivo

→ Redundancy: caching

- Reliability

É preciso garantir que o trabalho seja feito corretamente

→ Transaction: "session management"

- Scability

Deve ser capaz de "crescer" rapidamente / facilmente

- Concentro-se no servidor de aplicações

→ Redundancy: "equilíbrio de carga", caching

- Security

Autenticar utilizadores e passar identidades autenticadas através das camadas

- Autorização adequada dentro de cada camada, para além das fronteiras de confiança.

- Nível lógico (application container), application server, sistema distribuído

→ Não há uma solução universal, haverá sempre trade-offs

J2EE Detalhado

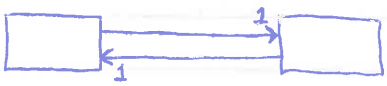
Padrões de comportamento

- Iterator - Acesso sequencial aos elementos de uma coleção
- Observer - Uma forma de notificar mudanças no número de classes
- Command - Encapsular um 'request' como um objeto
- Strategy - Encapsula um algoritmo dentro de uma classe
- State - Altera o comportamento de um objeto quando o estado muda
- Template Method - Adiar os methodos exatos de um algoritmo para uma subclasse
- Chain Of Responsability - Maneira de passar um pedido entre uma cadeia de objetos
- Mediator - Define comunicação simplificada entre as classes
- Interpreter - Forma de incluir elementos de "linguagem" num programa
- Memento - Captura e restaura o estado de um objeto interno
- Visitor - Define uma nova operação de uma classe sem alteração

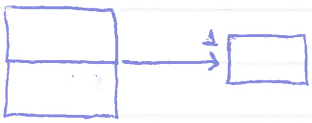
Noção de Lifecycle

Builder

State



Bridge



Observer

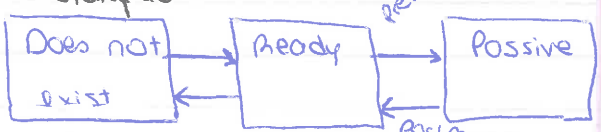
Session Bean

- Stateless

1. Dependency injection, if any
2. PostConstruct callbacks, if any

PreDestroy callbacks, if any

- stateful



1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any

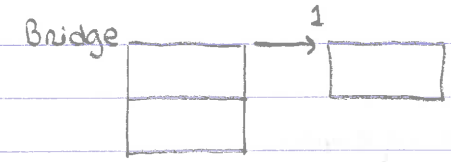
1. Remove
2. PreDestroy callback, if any

Annotations

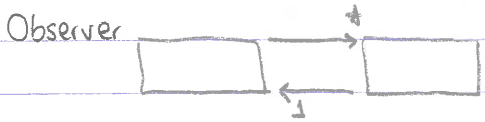
@ PostConstruct
@ PrePassivate
@ PostActivate
@ Remove
@ PreDestroy

Just to Inform: Interceptor Mechanism - Mecanismo Interceptor

```
public class CallTracer {
 @AroundInvoke
 public Object transformReturn (InvocationContext ctx) throws Exception {
    System.out.println ("---" + context.getMethod());
    return ctx.proceed();
 }
}
```




```
@Stateless
@Interceptors (CallTracer.class)
public class HelloBean {
 public String sayHello (String message) {
    return "Echo from bean: " + message;
 }
}
```



Interceptor: following the life cycle

```
import ...
import ...

public class LifecycleInterceptor {
@PostConstruct
 public void postConstruct (InvocationContext ctx) {...}
@PreDestroy
 public void PreDestroy (InvocationContext ctx) {...}
@PrePassivate
 public void PrePassivate (InvocationContext ctx) {...}
@PostActivate
 public void PostActivate (InvocationContext ctx) {...}
@Init
 public void postInit (InvocationContext ctx) {
    try {
```

```
System.out.println("LifecycleInterceptor Init");
ctx.proceed();
} catch (Exception e)
    } throw new RuntimeException(e); {
} }
```

Entities Lifecycle

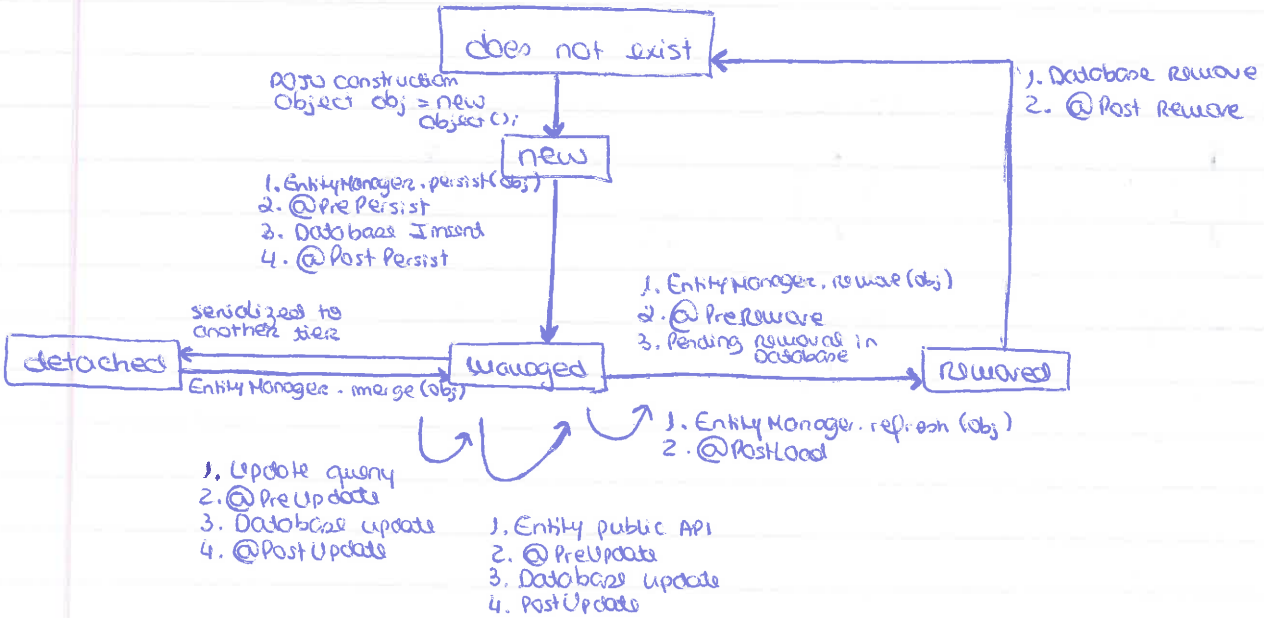
Instâncias de entidade pode ter quatro estados:

- **New** - Sem identidade de persistência e ainda não estão associados a um contexto de persistência.

- **Managed** - Tem uma identidade persistente e estão associados com um contexto de persistência.

- **Detached** - Tem uma identidade persistente e atualmente não são associados a um contexto de persistência.

- **Removed** - Tem uma identidade persistente, estão associados a um contexto de persistência, e estão programados para a remoção do armazenamento de dados.



Entities and EntityManagers

- Pode ser "tracked" através de CallBacks and Listeners
- Track de operações nas entidades
  - persisting, updating, removing, loading
- Correspondente à operação na Base de Dados
  - inserting, updating, deleting, selecting
- Track de managed entities ... not POJOs
  - @PrePersist
  - @PostPersist
  - @PreRemove
  - @PostRemove
  - @PreUpdate
  - @PostUpdate
  - @PostLoad

Managed Entities and "Units"

Abstrações para lidar com acessos

- uso de IDL neutro (web services)
- Sessão de fachada (facade) para acessar um determinado recurso - dentro e para do âmbito local

Adapter

new Interface

Facade

Simplified Interface

Proxy

Same Interface

Sample Webservice

```
import javax.ejb.Stateless;
@Stateless
public class Calculator {
    public calculator () { }
    public int add (int i, int j) {
        int k = i+j;
        System.out.println (i+"+"+j+"="+k);
        return k;
    }
}
```

```
import javax.ejb.Stateless;
import javax.jws.WebService;
@Stateless
@WebService
public class Calculator {
    public Calculator () { }
    public int add (int i, int j) {
        int k = i+j;
        System.out.println (i+"+"+j+"="+k);
        return k;
    }
}
```

Strategy

Standard polymorphism

Factory Method

Polymorphism for creation

Persistência no J2EE

- Padrões que já vimos noutros contextos

Adapter

new interface

Facade

Simplified interface

Proxy

same interface

→ Acesso aos recursos através de uma interface específica (JPA, JDBC)

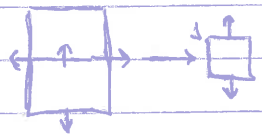
Strategy

Standard polymorphism

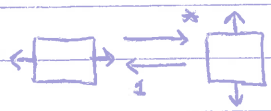
Polymorphism for creation

→ A "interaction managment" é delegada (Entity Manager) que lida com as especificidades (ex: Drivers)

Bridge



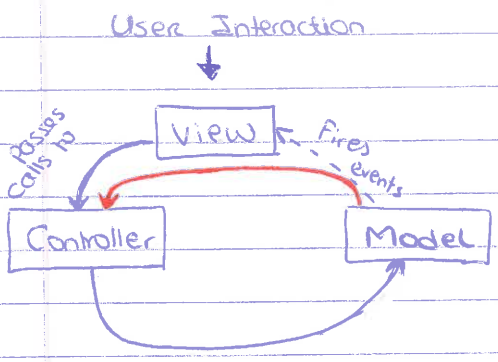
Observer

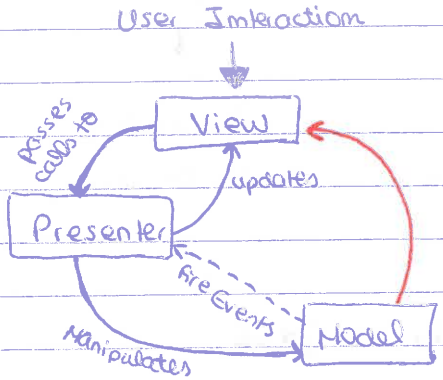


→ Acompanha as mudanças no estado das entidades (ex: Entity Lifecycle)

Mvc pattern

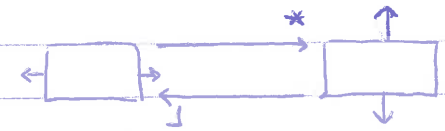
Observando mudanças



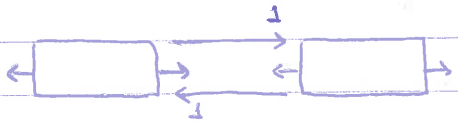


Model - View - Controller

Model - View - Presenter



Observer



State

Command - callback

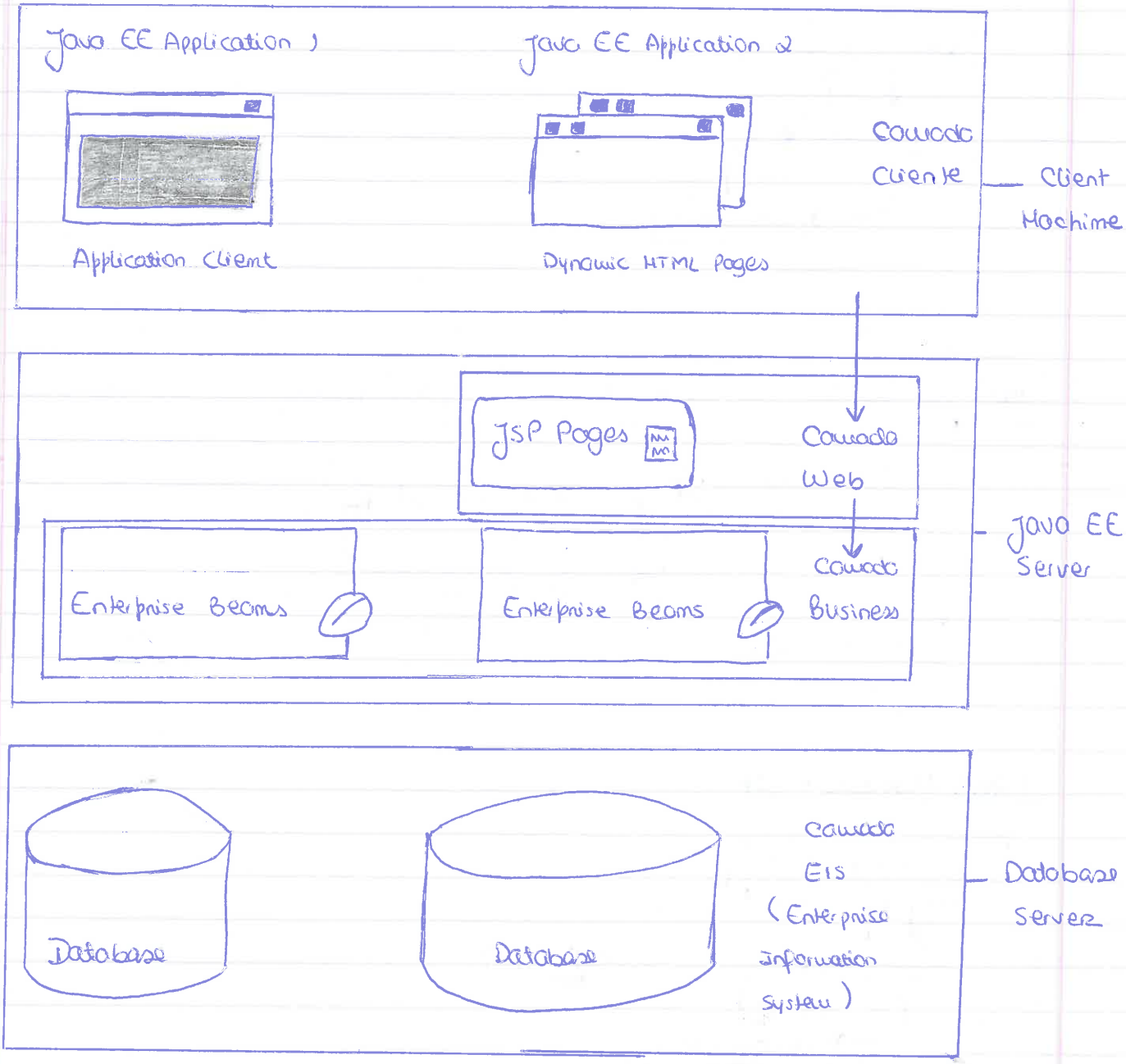
Iterator - transversal of a collection

Mediator - Many-to-Many relationships

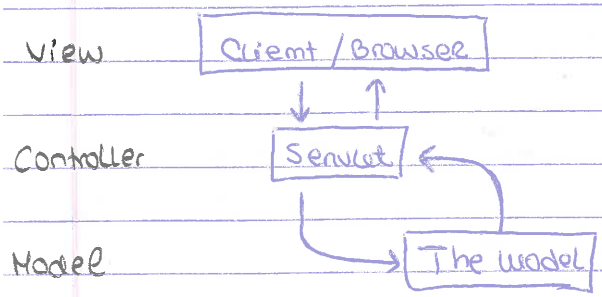
Memento - undo

Prototype - the new operator

From top to buttom : MVC Pattern

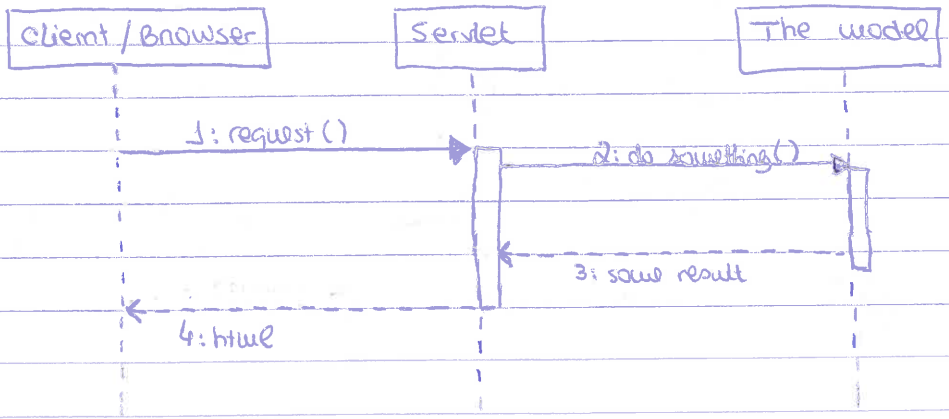


MV* : J2EE flavours

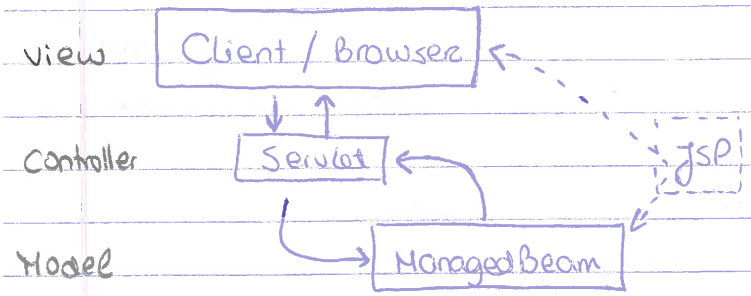


Adapter - new interface
Facade - Simplified Interface
Proxy - same Interface

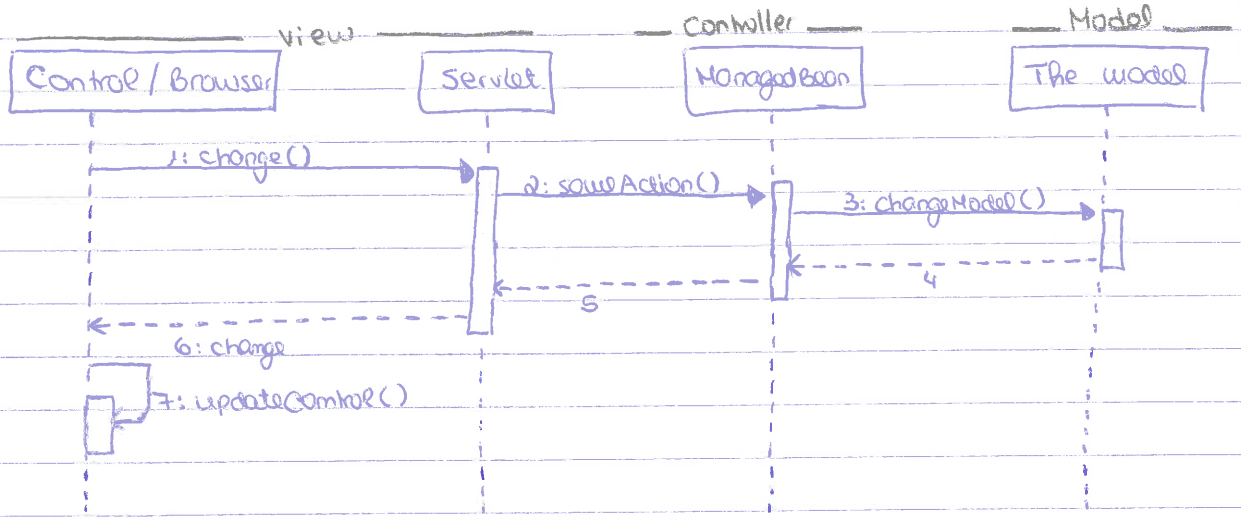
Servlet



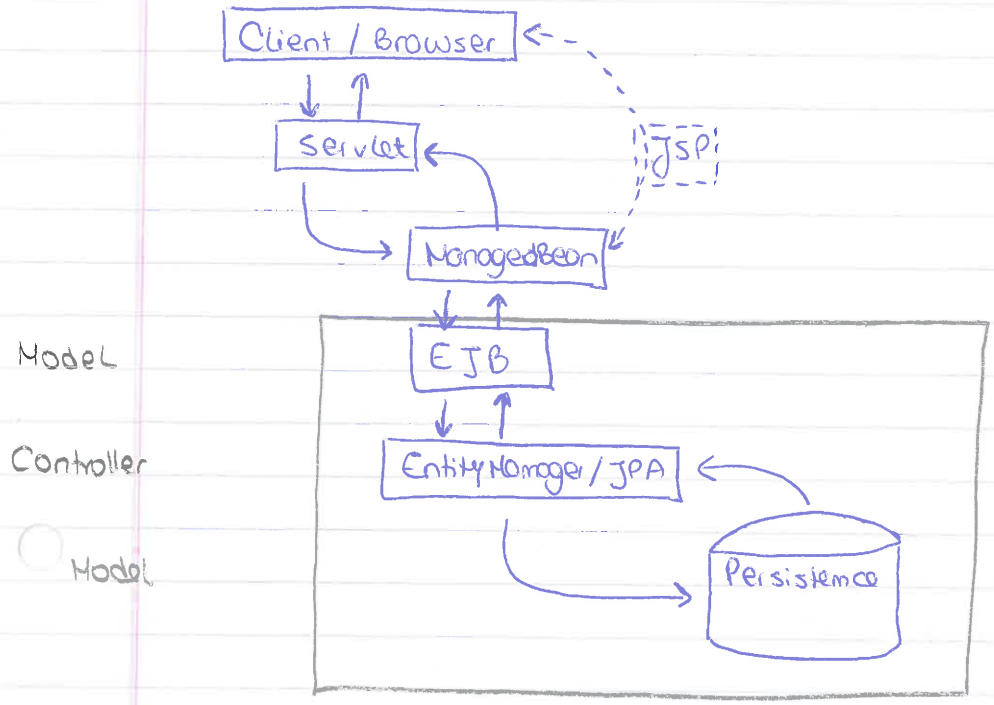
J2EE flavours : JSP

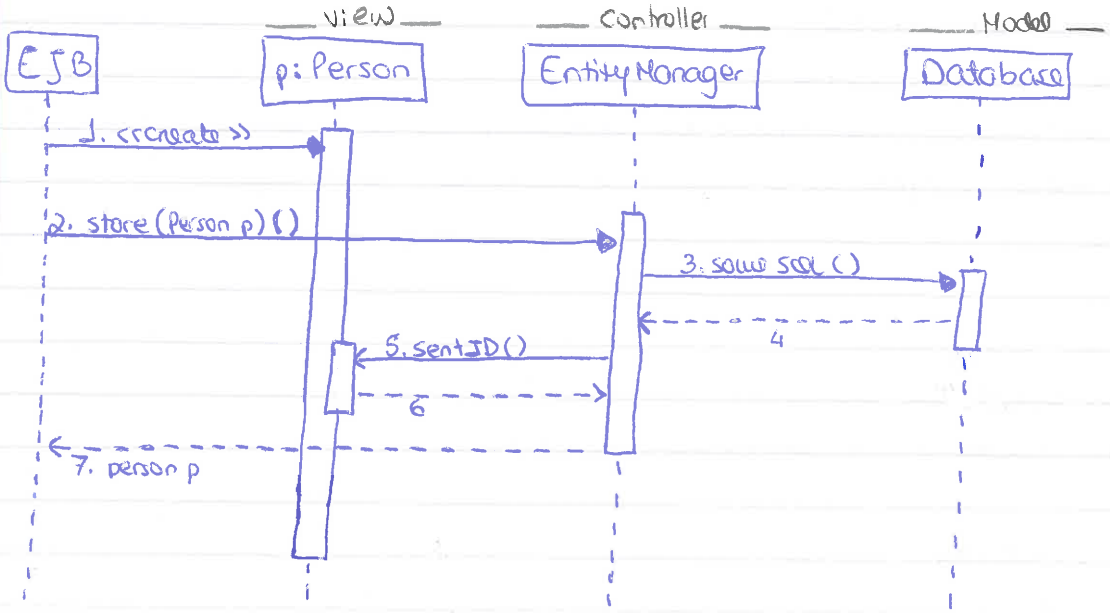


Adapter - new interface
Facade - simplified Interface
Proxy - same interface



J2EE flavours: JPA

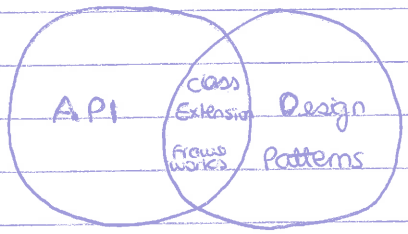




DF
AM15

Design Patterns

Outra Motivação: Re-Use



Code re-use Design re-use

Pattern objetive

- Padrões de Software ajudam-nos, porque eles
  - resolvem problemas do "mundo real"
  - Domínio na experiência de captura
  - Decisões do design (Projeto)
  - Transmitir uma versão experiente para novatos
  - Seus nomes formam coletivamente um vocabulário que ajuda os desenvolvedores a se comunicar melhor e ajudo-os a entender o sistema mais rapidamente (=> abstrações)

- Programar um bom projeto
  - Sintetizar e difundir experiência
  - Útil para novatos e para utilizadores experientes

- Fornece estruturas
  - vocabulário comum
  - Complexidade reduzida
  - Maior expressividade

- Capturar e preservar as informações do projeto
  - Articula sucintamente as decisões do projeto
  - Melhora a documentação

- facilitar a reestruturação
  - Os padrões são inter-relacionados
  - flexibilidade acional

Descrição Básico de um Template

Design Patterns: Os padrões de projeto são modelos de projetos que podem ser usados numa variedade de sistemas. Eles são particularmente apropriados em situações onde as classes possam ser reutilizadas num sistema que evolui com o tempo.
Os padrões de projeto têm 4 elementos essenciais

Name
- Aumenta o vocabulário do "designer"
- Identificador para descrever algum problema no projeto, suas soluções e consequências.
- Permite ao projeto um nível mais alto de abstração
- Mecanismo de comunicação para "designers".

Problem Description
- Descreve quando o padrão pode ser utilizado, muitas vezes em termos de "capacidade de mudança" e extensibilidade.
- Objetivo, contexto, quando se aplica
- Explica o problema e o seu contexto
- Pode incluir uma lista de condições que devem ser cumpridas antes que faça sentido aplicar o padrão.

Solution
- Expresso em termos de classes e interfaces
- Estrutura como o UML, código abstrato
- Descreve os elementos que compõem o projeto, seus relacionamentos, responsabilidades e colaborações.
- Não é a concecção ou implementação
- Um padrão é como um modelo
- fornece uma descrição abstrato de um problema do projeto e como um arranjo geral de elementos resolve.

Consequências
- Trade-offs e alternativas
- Resultados e trade-offs da aplicação do padrão
- Crítica para avaliar alternativas de projeto e custos benefícios do padrão
- O impacto do padrão sobre a flexibilidade do sistema, extensibilidade ou portabilidade pode ser considerado.

Descrebendo Padrões: Singletom

Intenção: Garantir que uma classe tem apenas uma instância e fornecer um ponto global de acesso a ela

- Nome do Padrão e Classificação
  - Importante porque se torna parte do vocabulário do projeto.

- Intenção: Breve Descrição do Modelo e sua finalidade
  - O que faz o padrão de projeto?
  - Qual a sua razão e intenção?
  - Qual é o problema que o projeto trata?

- Também conhecido como: Outros nomes que as pessoas têm para o padrão
  - Outros nomes bem conhecidos para o padrão se houver.

- Motivação

Cenário que ilustra um problema de projeto e como as estruturas das classes e objetos resolvem o problema.

- Quando um determinado objeto num sistema tem uma responsabilidade que depende de outros objetos
- Em alguns casos, é importante para essas classes terem exatamente uma instância, por exemplo, um spooler de uma impressora
- Como garantir que uma classe tem somente uma instância e que essa instância é facilmente acedido?

- Solução
  - faz o próprio classe responsável por manter o controle da sua única instância. A classe pode garantir que nenhum outra instância pode ser criado, e pode fornecer uma maneira de acessar a instância.

↳ Este é o padrão Singletom.

• Aplicabilidade

- Circunstâncias onde se aplica o padrão
- Quando o padrão deve ser aplicado?
- Como se pode reconhecer essas situações

↓

- Deve haver uma instância de uma classe, e deve ser acessível a clientes de um ponto de acesso conhecido
- A única instância deve ser extensível por subclasses, e os clientes devem ser capazes de usar uma instância estendida sem modificar o seu código (requer um construtor protegido em vez de um construtor privado)

• Estrutura e Participantes

- A representação gráfica das classes no padrão usando a notação UML para ilustrar sequências de pedidos e colaborações.
- As classes e/ou projetos que participam no padrão de projeto e suas responsabilidades

↓

- Singleton define uma operação de instância que permite aos clientes acessem o seu único instância. Também é responsável por criar a sua própria instância.

• Comsequências

- Resultados da Aplicação, benefícios e responsabilidades
- Como é que o padrão apoia os seus objetivos?
- Quais são os trade-offs e o resultado de usarem o padrão?
- Que aspeto da estrutura do sistema é que permite variar de forma independente?

↓

- Reduz "nowespacing pollution"
- Permite acesso controlado ao objeto singleton
- Facilita mudanças de ideias e permite mais que uma instância
- Permite extensão a subclasses
- Os mesmos inconvenientes de uma global, se mal usado
- A implementação pode ser menos eficiente que a global

Creational Patterns

- Abstract factory - Cria uma instância de várias famílias de classes
- Factory Method - Cria uma instância de várias classes derivadas
- Singleton - Classe da qual apenas existe uma instância
- Prototype - Um exemplo totalmente inicializado a ser copiado ou clonado.
- Builder - Separa a construção do objeto da sua representação

Abstract factory

- Intenção → Fornecer uma interface para criar famílias de objetos relacionados ou dependentes
  → A hierarquia que encapsula: muitas "plataformas" e construção de um conjunto de "produtos"
  → O novo operador é considerado prejudicial

- Problema → Se uma aplicação é para ser portável, ela precisa de encapsular dependências na plataforma. Estas "plataformas" podem incluir: sistemas de janelas, sistema operacional, base de dados, etc.
  Muitas vezes, este encapsulamento não é projetado com antecedência e muitas casos #ifdef com opção para todas as plataformas suportadas atualmente causam o proliferar como "coelhos em toda o código".

Factory Method

- Intenção → Define uma interface para criar um objeto, mas deixa que as subclasses decidam qual a classe a instanciar. Factory Method permite a instanciação da classe de adiantamento para subclasses.
  → Definição de um construtor "virtual".
  → O novo operador é considerado prejudicial.

- Problema → framework precisa de uniformizar o modelo de arquitetura para uma variedade de aplicações, mas permite que aplicações independentes definam os seus próprios objetos de domínio e garanta a sua instanciação.

- Singleton

- Intenção → Certifica-se que uma classe tem apenas uma instância, e fornece um ponto global de acesso a ela.
  → Encapsulado "just-in-time initialization" ou "initialization on first use".

- Problema → A aplicação precisa de apenas uma instância de um objeto. Além disso, a inicialização é lenta e o acesso global é necessário.

- Builder

- Intenção → Separar a construção de um objeto complexo a partir da sua representação, de modo que o mesmo processo de construção pode criar representações diferentes.
  → Analisa uma representação complexa, cria um de vários alvos.

- Problema → Uma aplicação precisa para criar os elementos de um agregado complexo. A especialização para o Conjunto (/agregado) existe no armazenamento secundário, e uma das muitas representações precisa de ser construída em armazenamento primário.

- Prototype

- Intenção → Especificar os tipos de objeto a criar usando uma instância prototípica e criar novos objetos copiando esse protótipo.
  → Co-optar uma instância de uma classe para usar como um criador de todas as ocorrências futuras.
  → O novo operador é considerado nocivo/prejudicial.

- Problema → Aplicação "hard codes" do classe objeto para criar em cada "new" expressão

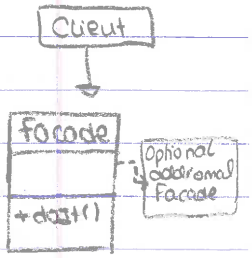
Structural Patterns

- Facade - Uma única classe que representa o sistema.
- Adapter - Agrupa Igualar interfaces de diferentes classes
- Composite - Estrutura em árvore de objetos simples e compostos
- Proxy - Um objeto que representa outro objeto.
- Bridge - Separa a interface de um objeto a partir da sua implementação
- Decorator - Adiciona responsabilidades a objetos dinamicamente
- Flyweight - Uma instância de "fine-grained" usada para o compartilhamento eficiente.

• Facade

- Intenção → Fornecer uma interface unificada para um conjunto de interfaces num subsistema. Facade define uma interface de alto nível que torna o subsistema fácil de usar.

→ Agrupa um subsistema complicado, com uma interface mais simples.



- Problema → Um segmento da comunicação do cliente precisa de uma interface simplificada para a funcionalidade global de um subsistema complexo.

• Adapter

- Intenção → Converte a interface de uma classe noutra interface esperada pelos clientes. Adapter permite que classes trabalhem juntas que não podiam de outro modo, devido às interfaces incompatíveis.

→ Agrupa uma classe existente com uma nova interface

→ Impedância de corresponder um componente antigo para um novo sistema.

- Problem → Componente "off the sheef" oferece uma funcionalidade convincente / atraente que gostaríamos de reutilizar, mas o seu "visão do mundo" não é compatível com a filosofia e arquitetura do sistema a ser desenvolvido.

- Proxy (p.44)

- Intenção → fornecer um substituto ou placeholder ou espaço reservado para outro objeto controlar o acesso a ele.

→ Agrupa e delega para proteger o componente real a partir de uma complexidade desnecessária.

- Problema → Precisar de suportar objetos pesados de recursos, e não queremos instanciar tais objetos, a menos e até que sejam realmente solicitado pelo cliente.

Behavioral Patterns

- Iterator – Acesso sequencial a objetos de uma coleção
- Observer – Forma de notificar mudanças de um número de classes
- Command – Encapsular um 'request' como um objeto
- Strategy – Encapsula um algoritmo dentro de uma classe
- State – Altera o comportamento de um objeto quando o estado muda
- Template Method – Adiar as medidas exatas de um algoritmo para uma subclasse
- Chain of Responsability – Maneira de passar um pedido entre uma cadeia de objetos
- Mediator – Define comunicação simplificada entre as classes
- Interpreter – Forma de incluir elementos de "linguagem" num programa
- Memento – Captura e restaura o estado de um objeto interno
- Visitor – Define uma nova operação de uma classe sem alteração

- Observer

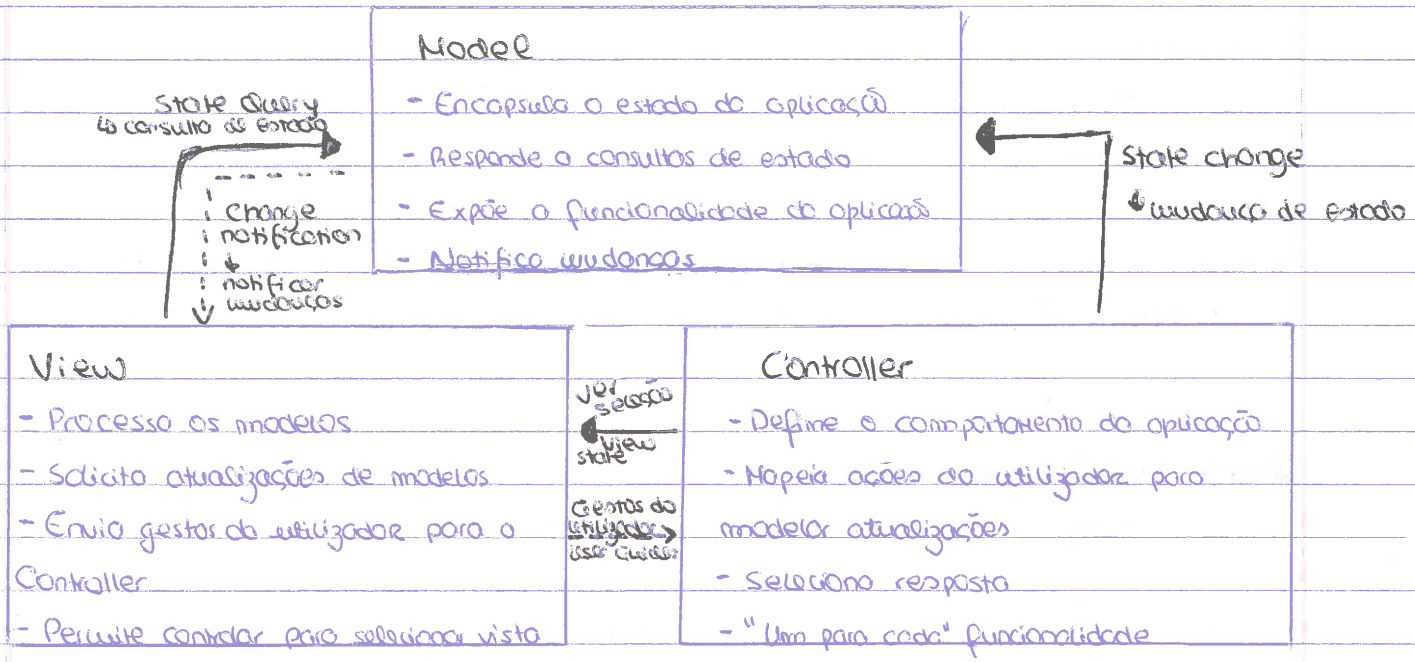
Intenção → Define uma dependência "um-para-muitos" entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

→ Encapsular o núcleo de componentes numa abstração de assuntos e o variável componentes em uma hierarquia Observer.

→ Parte "View" de "Model-View-Controller"

Problema → Um grande projeto monolítico não escala bem como nova representação gráfica, ou requisitos de monotorização são cobrados.

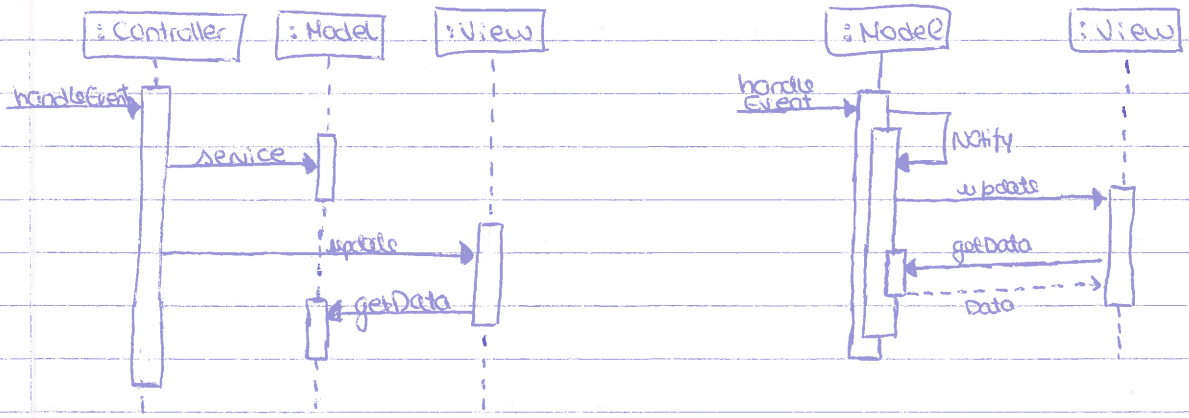
Model View Controler



→ Method Invocation

- - -> Events

MVC: GUI Pattern



• Command

- Intenção → Encapsular um 'request' (solicitação) como um objeto, permitindo assim parametrizar clientes com pedidos diferentes, o fila ou o registo de solicitações, e apoiar as operações que podem ser desfeitas.
  → Provaver "invocação de um método num objeto"
  → "Callback" objeto-orientado

- Problema → Necessidade de emitir pedidos (requests) para objetos sem saber nada sobre a operação que está sendo solicitada ou o recetor do pedido.

• Strategy

- Intenção → Definir uma família de algoritmos, encapsular cada um e torná-los intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.
  → Capturar a abstração de uma interface, enterrar os detalhes de implementação em classes derivadas.

- Problema → Uma das estratégias dominantes do design orientado a objeto é o "princípio de abrir e fechar".

• State (p.58)

- Intenção → Permitir que um objeto altere o seu comportamento quando seu estado interno muda. O objeto parece mudar o sua classe.
  → Máquina de estados orientada ao objeto
  → Wrapper + polymorphic wrappee + collaboration

- Problema → O comportamento de um objeto modifica-se em função do seu estado, e o seu comportamento deve mudar no tempo de execução, dependendo desse estado. Ou, uma aplicação é caracterizada por grandes e numerosas declarações de casos que o fluxo do vetor de controlo base no estado da aplicação.

• Memento (p. 61)

- Intenção → Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto, de modo que o objeto possa ser devolvido a este estado mais tarde.

  → Um cookie mágico que encapsula um recurso "check point"

  → Promover "desfazer" ou "reversão do estado" do objeto completo

- Problema → Necessidade de restaurar um objeto de volta ao seu estado anterior. (ex: "desfazer" ou "reverter" operações)

• Template Method (p. 63)