RAS

Capítulo 1 - Requisitos

Os requisitos definem as necessidades dos utilizadores e as restrições colocadas no sistema. É a capacidade que o sistema deve possuir para satisfazer as necessidades dos utilizadores

- Funcionais
- Não Funcionais

A classificação dos requisitos depende de quem está a olhar para eles

Requisito Funcional
Descreve a funcionalidade a ser disponibilizada aos utilizadores do sistema
Deve ser coerente(não haver contradições) e completo(considerar todos as necessidades dos clientes) - ser completo é difícil de avaliar

Requisito implícito é incluído pela equipa de desenvolvimento, com base no conhecimento que tem sobre o domínio, apesar de não ter sido requisitado
Requisito explicito é solicitado pelos clientes e esta presente na documentação

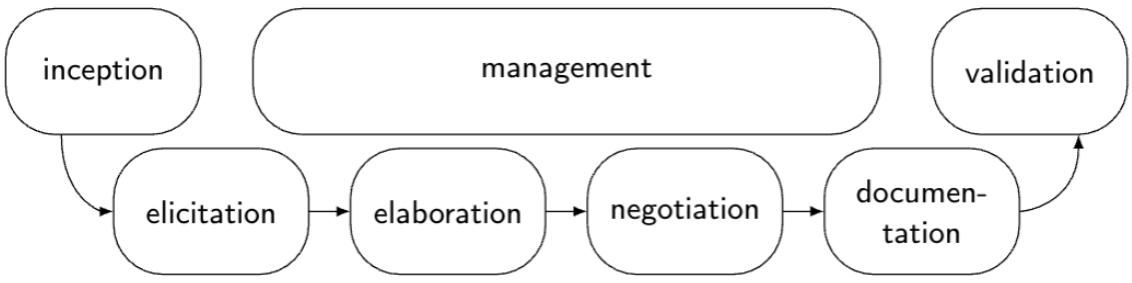
Requisito Não Funcional
Corresponde ao conjunto de restrições impostas no sistema a serem desenvolvidas (quão rápido, confiável... é o sistema)
Não altera a essência das funcionalidades do sistema (a cor de uma bola de futebol não afeta a sua funcionalidade)
Não podem ser modularizados, são normalmente propriedades emergentes do sistema, ou seja, só podem ser associados ao sistema como um todo e não individualmente a cada componente.
(A propriedade emergente de transportar uma pessoa só é oferecida pela bicicleta como um todo)
Cruciais para a decisão da arquitetura

Aspecto	Requisitos de Utilizador (Domínio do Problema)	Requisitos de Sistema (Domínio da Solução)
Foco	O que o utilizador quer ou precisa resolver.	Como o sistema deve ser construído para atender à necessidade.
Linguagem	Simples, acessível ao utilizador.	Técnica, detalhada para desenvolvedores e engenheiros.

Capítulo 2 - Requisitos de Engenharia

Processo de **descoberta, análise, documentação e verificação** das necessidades do sistema antes do seu desenvolvimento. O seu objetivo é **entender** o problema e **garantir** que as necessidades dos utilizadores são atendidas.

1. Iniciação - definir o **escopo inicial** do sistema e identificar os principais stakeholders
2. Elicitação - os requisitos são **coletados** junto aos stakeholders
3. Elaboração - os requisitos coletados durante a elicitação são analisados e **detalhados**
4. Negociação - os requisitos são **priorizados e discutidos** com os stakeholders
5. Documentação - os requisitos finais são **documentados formalmente**
6. Validação



Pode ser necessário voltar a uma etapa anterior à medida que novos requisitos forem adicionados

Capítulo 3 - Escrita de Requisitos

A técnica de rescrever requisitos deve ser impessoal, objetiva e clara

Standard format para requisitos de utilizador

a mechanism to allow a **test** for the requirement to be defined.

(users)	The hotel receptionist
(functionality)	should visualise
(object/concept)	the room number of a guest,
(test)	2 s after making the request.

User stories - põe o utilizador no foco da atenção

format

As a <type of user>**, I want to** <objective> **for** <reason>.

As a hotel receptionist, **I want to** visualise the room number of a guest **for** calling him if someone wants to contact him.

Standard format para requisitos de sistema

(system/entity)	The signal of the battery
(functionality)	must turn on,
(description)	when the charge is lower than 20 mA h.

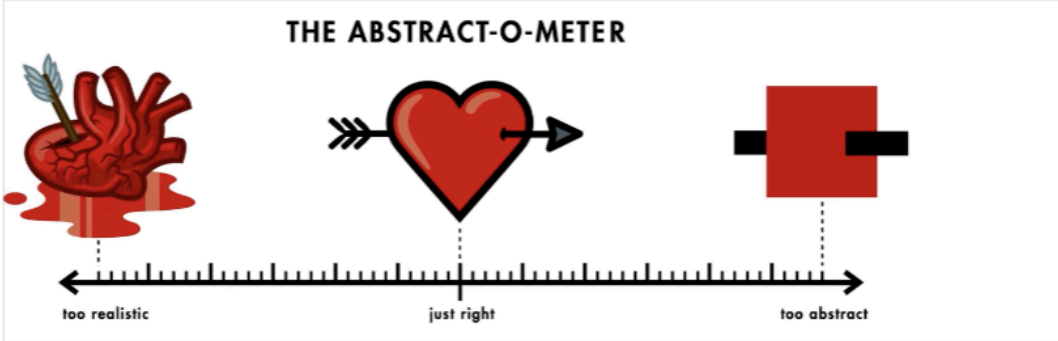
Capítulo 4 - Requisitos de Elicitação

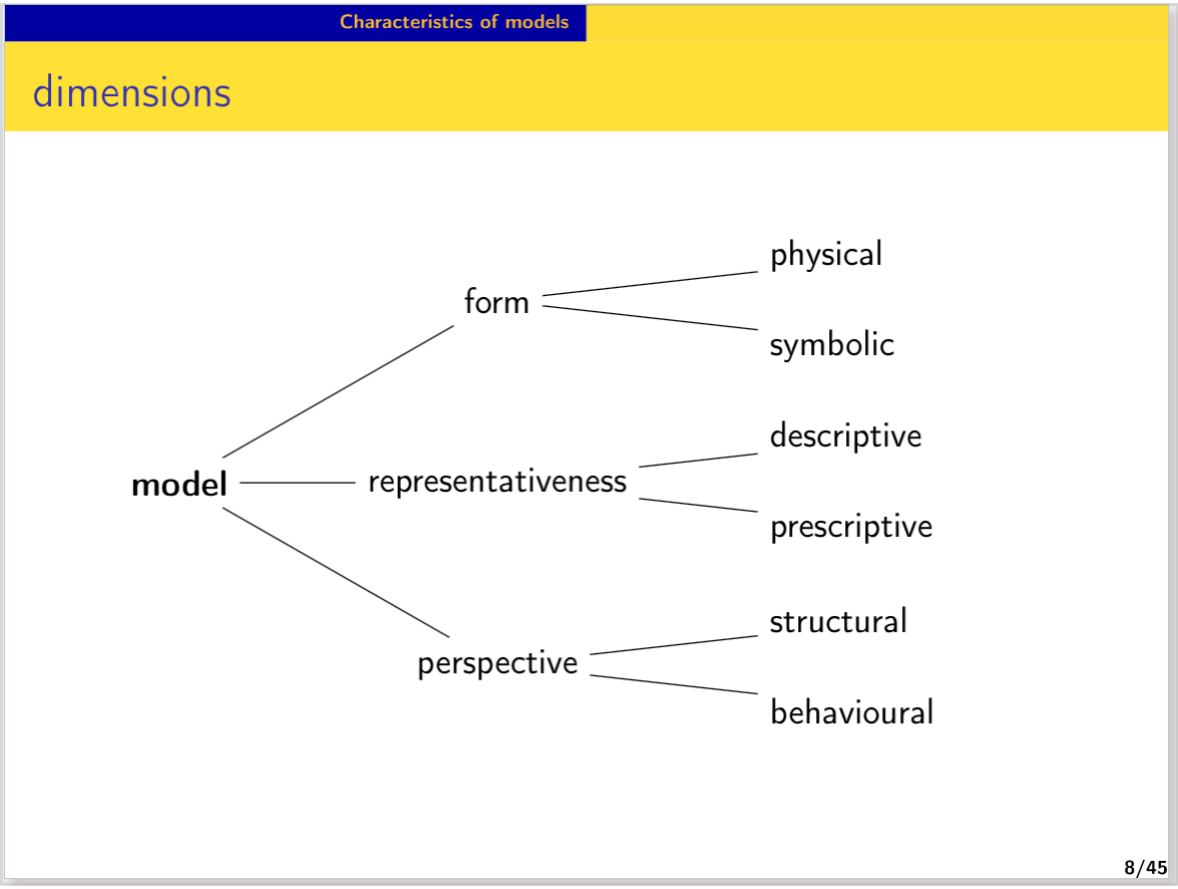
Permitem **entender** quais são os requisitos de um determinado sistema
Permitem **compreender** as necessidades e expectativas que os stackholders tem perante o sistema

Stackholder - alguma pessoa, grupo ou organização que tem algum tipo de interesse legitimo no sistema
User - alguma pessoa que opere e interage diretamente com o sistema
Cliente - entidade que encomenda e paga o desenvolvimento do sistema
Customer - alguém que paga para adquirir o sistema quando este estiver disponível
Persona - pessoa fictícia que representa um tipo importante de utilizadores de produtos em desenvolvimento

Capítulo 5 - Modelação

É o processo de identificar conceitos adequados e selecionar abstrações para construir o modelo que reflete apropriadamente o universo





`Forma`

- **fisico** - maquetes, prototipos físicos
- **simbólico** - representações UML, diagramas

`Representatividade`

- **descritivo** - descrevem um sistema (como é o funcionamento do sistema)
- **prescritivo** - sugerem como algo deve ser feito

`Perspetiva`

- **estrutural** - modelos focados na estrutura do sistema (estático)
- **comportamental** - descrevem o comportamento do sistema ao longo

do tempo(dinâmico)

Diagrama de use cases
Diagrama de classes
Diagrama de sequência
Diagrama de estados
Diagrama de atividades

Capítulo 6 - Riscos Arquiteturais

Risco = probabilidade de falha * impacto
Esforço arquitetural deve ser proposital ao risco de falha

Evolutionary design - NDUF
Não há um design inicial extenso ou detalhado (No Design Up-Front - NDUF).
O design é **evoluído gradualmente** à medida que o sistema é desenvolvido

Planned Design (Big DUF - BDUF)
O **design é totalmente planejado** antecipadamente (Big Design Up-Front - BDUF). Isso significa que todos os detalhes do sistema são definidos **antes de qualquer implementação** começar.

Minimal Planned Design (Enough DUF - EDUF ou Little DUF - LDUF)
Abordagem **intermediária** entre o design evolutivo (NDUF) e o design totalmente planejado (BDUF). Aqui, o design é **minimamente planejado** para garantir uma base sólida, mas é deixada flexibilidade para evoluir posteriormente

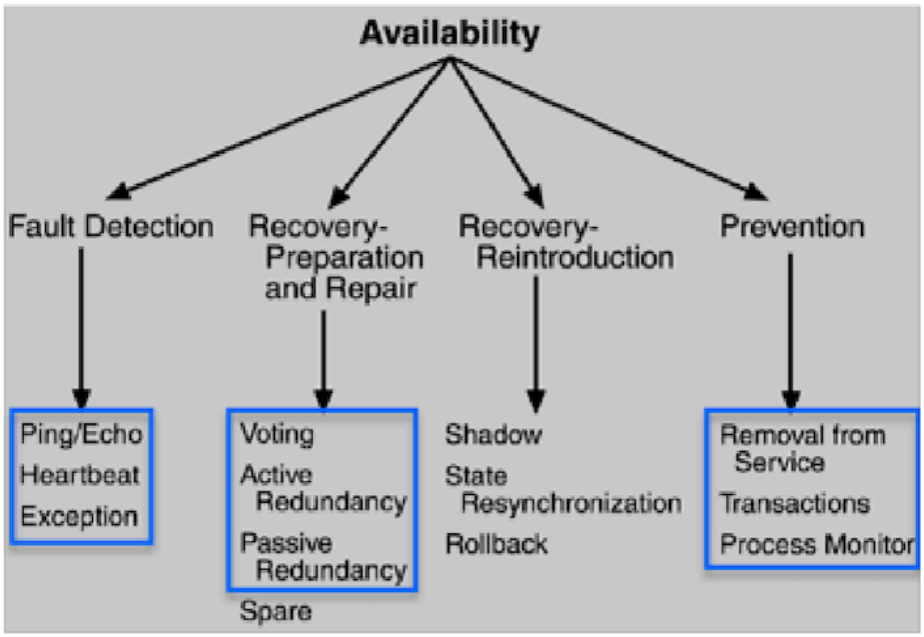
Capítulo 7 - Introdução à Arquitetura de Software(?)

Capítulo 8 - Táticas de Design

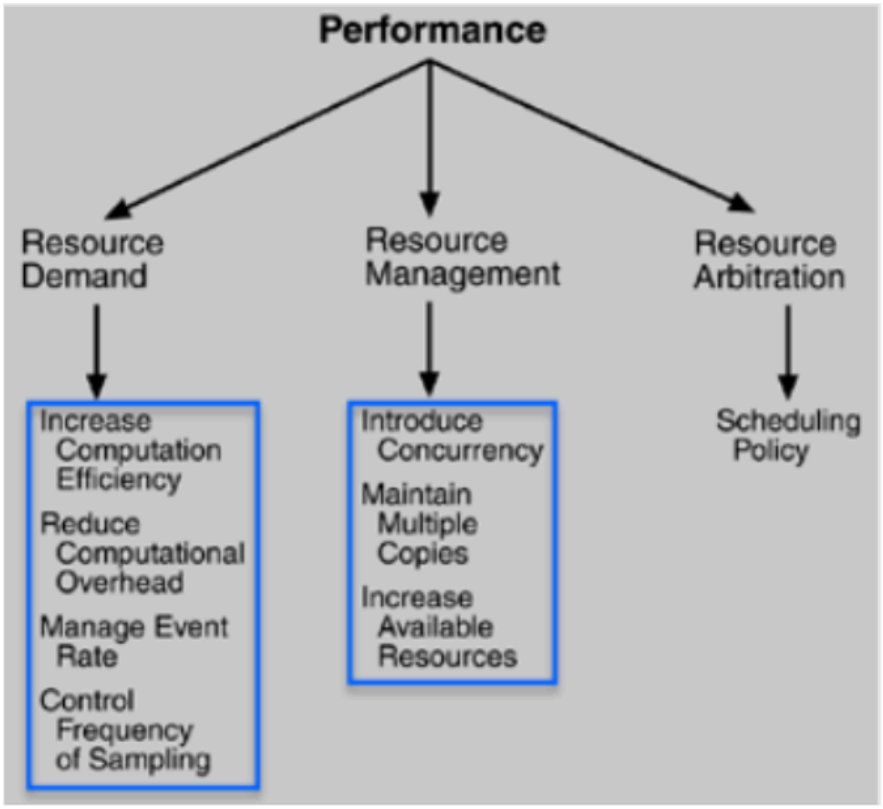
As táticas são usadas pela arquitetura para criar um design
A tática é uma decisão do design que impacta atributos de qualidade específicos.

Tipos :

- Disponibilidade -> quantifica a percentagem de tempo durante o qual um determinado sistema está operacional e a funcionar

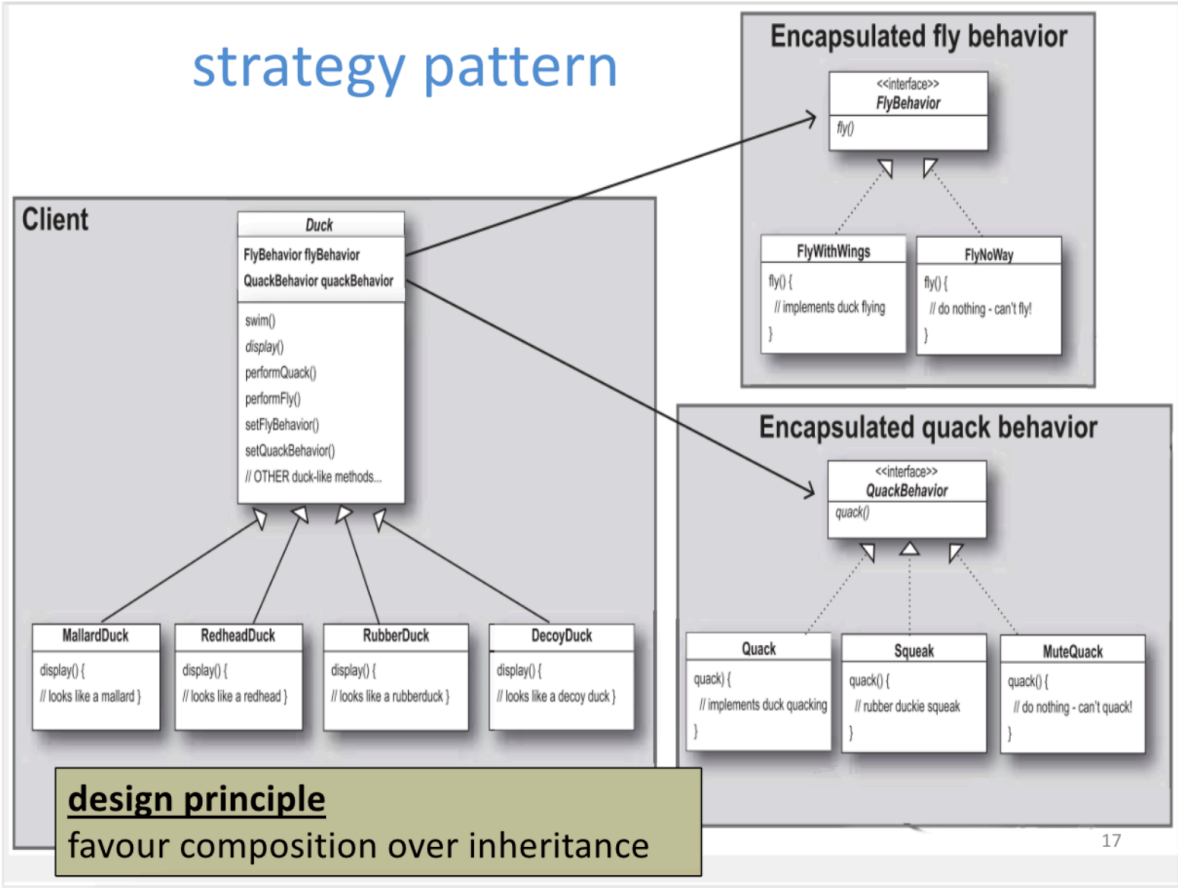


- Performance -> refere se à capacidade do sistema de responder a estímulos, ou seja, o tempo necessário para responder aos eventos

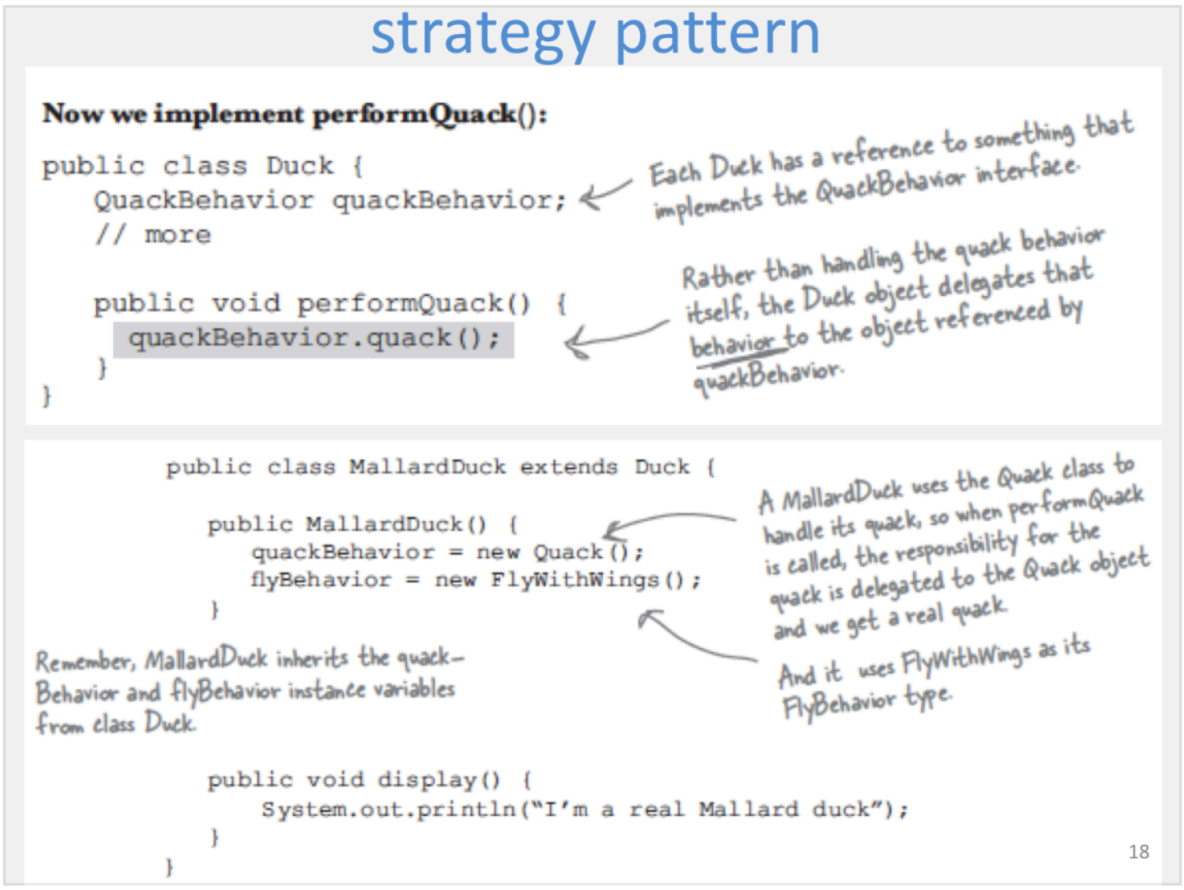


Capítulo 9 - Patterns(Padrões) de Projeto

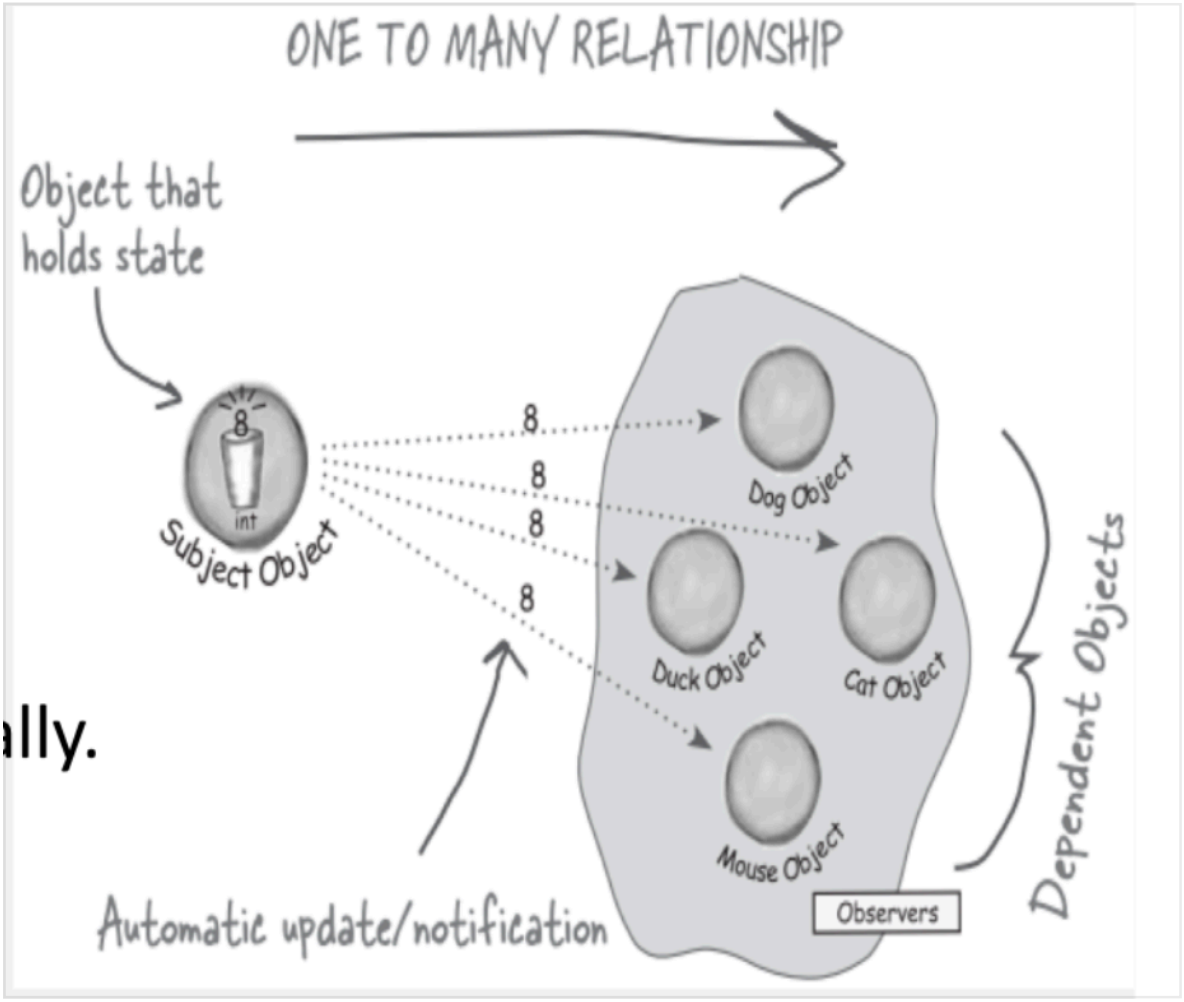
Os padrões constituem soluções comprovadas para problemas conhecidos e problemas comuns
Invés de código reutilizado, os padrões permitem a reutilização de experiências.
É uma solução reutilizável para um problema recorrente.
Há 3 tipos de design de padrões:

- Criacional
- Estrutural (estratégico)
- 

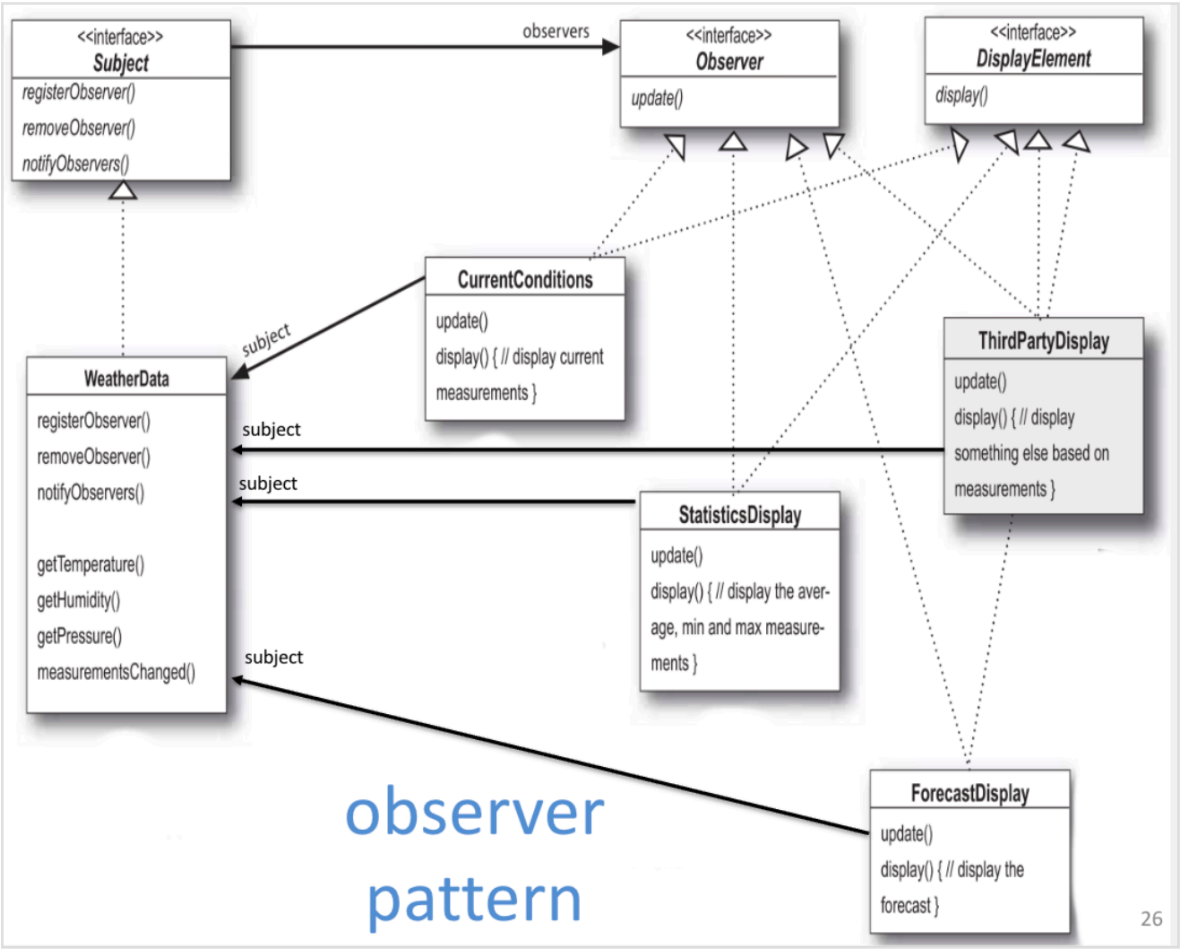
favorecer a composição em vez da herança



- Comportamental (observador)



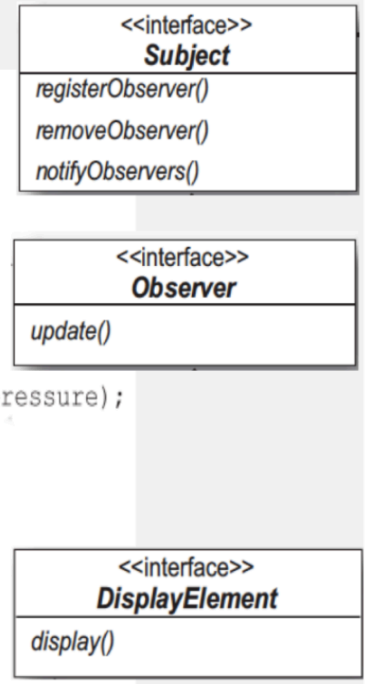
Estados dependentes são notificados quando há alterações de estados



observer pattern

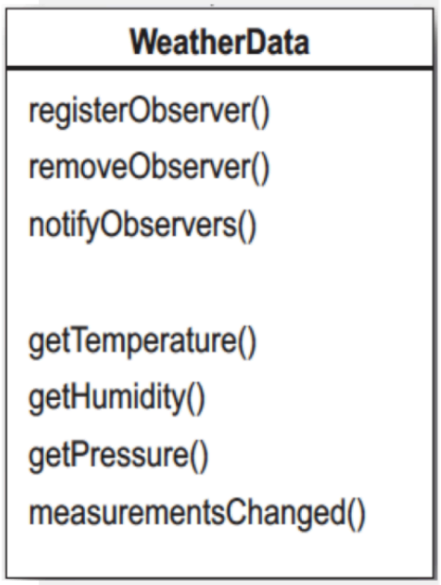
observer pattern

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}


public interface Observer {
    public void update(float temp, float humidity, float pressure);
}



public interface DisplayElement {
    public void display();
}
```



observer pattern

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
```



28

Capítulo 10 - Estilos Arquiteturais

catalog of styles

	Viewtype	Elements & Relations	Constraints / guide rails	Qualities Promoted
Layered	Module	Layers, uses relationship, callback channels	Can only use adjacent lower layers	Modifiabilty, portability, reusability
Big Ball of Mud	Module	None	None	None, but many inhibited
Pipe-and-Filter	Runtime	Pipe connector, filter component, read & write ports	Independent filters, incremental processing	Reconfigurability (modifiability), reusability
Batch-Sequential	Runtime	Stages (steps), jobs (batches)	Independent stages, non-incremental processing	Reusability, modifiability
Model-Centered (Shared Data)	Runtime	Model, view, and controller components; update and notify ports	Views and controllers interact only via the model	Modifiability, extensibility, concurrency
Publish-Subscribe	Runtime	Publish and subscribe ports, event bus connector	Event producers and consumers are oblivious	Maintainability, evolvability
Client-Server & N-Tier	Runtime	Client and server components, request-reply connectors	Asymmetrical relationship, server independence	Maintainability, evolvability, legacy integration
Peer-to-Peer	Runtime	Peer components, request-reply connectors	Egalitarian peer relationship, all nodes clients and servers	Availability, resiliency, scalability, extensibility
Map-Reduce	Runtime & allocation	Master, map, and reduce workers; local and global filesystem connectors	Divisible dataset amenable to map & reduce functions, allocation topology	Scalability, performance, availability
Mirrored, Farm, & Rack	Allocation	Varies	Varies	Varies: Performance, availability

6

Layered (Arquitetura em Camadas)

- **Explicação**: Neste estilo, o sistema é dividido em **camadas**, onde cada camada só pode interagir com a camada adjacente (superior ou inferior).

Exemplo Prático:

- Camada de Apresentação: interface do usuário (front-end).
- Camada de Lógica de Negócio: processamento de regras do sistema.
- Camada de Dados: acesso ao banco de dados.

Vantagem: Facilita a manutenção e a substituição de componentes individuais (ex: trocar um banco de dados sem alterar as outras camadas).
Desvantagem: Pode gerar um desempenho mais lento, pois cada requisição precisa passar por várias camadas.

Big Ball of Mud

- **Explicação**: É uma arquitetura **caótica**, sem estrutura ou organização.

Exemplo Prático: Um sistema legado que cresceu ao longo dos anos sem planejamento. Novas funcionalidades foram "coladas" no código existente sem considerar a organização.
Vantagem: Rápido de implementar no curto prazo.
Desvantagem: Difícil de manter e entender. Adicionar novas funcionalidades pode quebrar outras partes do sistema.

Pipe-and-Filter (Arquitetura de Filtros e Conexões)

- **Explicação**: O sistema é dividido em **filtros** independentes que processam dados e os enviam para o próximo filtro através de **pipes**.

Exemplo Prático:

Um compilador, onde o código-fonte passa por várias etapas de processamento: análise léxica → análise sintática → geração de código.
Sistemas de **streaming** que processam vídeos ou áudios em etapas.

```
cat "f.txt" | grep "^Braga" | cut -f 2-
```

Vantagem: Os filtros são independentes e podem ser reutilizados em outros sistemas.
Desvantagem: Pode ser ineficiente para grandes volumes de dados, já que cada etapa precisa terminar para passar ao próximo filtro.

Batch-Sequential (Processamento em Lotes)

- **Explicação**: Os dados são processados em **batches** (lotes) em etapas

independentes, mas o processamento **não é incremental** (um lote inteiro deve ser processado antes de ir para a próxima etapa). Idêntico ao pipe-and-filter

Exemplo Prático: Processamento de **folha de pagamento** de funcionários ou geração de relatórios financeiros.
Vantagem: Simples de implementar e eficiente para grandes conjuntos de dados.
Desvantagem: Não é adequado para sistemas em tempo real.

Model-Centered (Shared Data)

- **Explicação**: Os componentes **model**, **view** e **controller** interagem através de um **modelo compartilhado** (componente central de dados).

Exemplo Prático:
Padrão **MVC** usado em frameworks como Laravel, Django ou Spring.
A **View** exibe dados ao user, o **Controller** manipula a lógica, e o **Model** gere os dados.
Vantagem: Organiza o código e promove a separação de responsabilidades.
Desvantagem: Pode ter problemas de concorrência, pois vários componentes podem acessar o modelo ao mesmo tempo.

Publish-Subscribe (Publicação e Subscrição)

- **Explicação**: Neste estilo, os **produtores** de eventos publicam mensagens em um **canal** e os **consumidores** subscrevem e recebem as mensagens. Os dois lados são independentes.

Exemplo Prático:
Sistemas de notificação, como o **Firebase** para envio de mensagens push.
Aplicações baseadas em **eventos**, como sistemas de logs ou mensagens em tempo real.
Vantagem: Alta escalabilidade e desacoplamento entre componentes.
Desvantagem: Pode ser difícil rastrear e fazer debug o fluxo de mensagens

Client-Server & N-Tier (Cliente-Servidor e N-Camadas)

- **Explicação**: O sistema é dividido em **clientes**, que solicitam serviços, e **servidores**, que respondem às requisições. Em **N-Tier**, existem camadas adicionais (ex: serviços intermédios).

Exemplo Prático:
Aplicações **web**, onde o navegador é o cliente e o servidor web processa as requisições.
Vantagem: Permite escalabilidade e modularidade.
Desvantagem: Pode ter gargalos no servidor em sistemas com muitos clientes.

Peer-to-Peer (P2P)

- **Explicação**: Não há distinção entre clientes e servidores; todos os **peers** podem fornecer e consumir serviços.

Exemplo Prático:

Redes de compartilhamento de arquivos, como **BitTorrent**.
Sistemas blockchain, como o Bitcoin.
Vantagem: Alta disponibilidade e resiliência.
Desvantagem: Complexidade na coordenação e segurança.

Map-Reduce

- **Explicação**: Dados são divididos em **partes menores** para serem processados paralelamente (**Map**) e, em seguida, os resultados são combinados (**Reduce**).

Exemplo Prático: Processamento de grandes volumes de dados
Vantagem: Escalabilidade e performance em grandes volumes de dados.
Desvantagem: Não adequado para pequenos volumes de dados.

Mirrored, Farm & Rack

- **Explicação**: Este estilo foca em **alocação física de recursos** para melhorar desempenho e disponibilidade.

Exemplo Prático:
Mirrored: Dados replicados em vários servidores para redundância.
Farm: Distribuição de tarefas em um conjunto de servidores.
Rack: Organização física de servidores em datacenters.
Vantagem: Melhora desempenho e garante disponibilidade.
Desvantagem: Alto custo operacional e de manutenção.

Capítulo 11 - Refactoring

É uma serie de pequenos passos, cada um dos quais muda a estrutura interna do programa sem mudar o comportamento externo

Porque faze lo ?

- Código duplicado
- Rotina demasiado longa
- Loop muito longo ou muito aninhado
- Torna o software mais fácil de entender
- Ajuda a encontrar bugs

TIPS

- Quando for para adicionar alguma feature ao programa, e o código do programa não contem a estrutura conveniente para tal, primeiro fazer refactoring ao programa para o tornar mais fácil de adicionar a feature e depois sim, adiciona la.
- Antes de fazer o refactoring ter um conjunto de testes para ter a

certeza que o comportamento não é alterado.
- Fazer pequenas mudanças para ser mais fácil corrigir possível erros
- Se o método é simples como a função, trocar a chamada da função pelo o corpo da mesma
- Se uma expressão for complexa, por o resultado dessa expressão numa variável intuitiva

Bad code smells

- Change preventers smells - Mudanças num sitio do programa, implica mudanças noutros sítios
- Bloaters - Unidades de código que aumentaram e se tornaram difíceis de trabalhar
- Dispensable - algo inútil que pode ser removido