2Capítulo 1.

O que é o Software Design?

- O Design preenche a lacuna entre saber o que é necessário **(fase de especificação de requisitos de software)** e inserir o que código que faz o software funcionar **(fase de construção)**.

O Design é necessário em diferentes níveis de detalhes de um sistema:

- No próprio sistema
- Nos subsistemas ou em packages: interface do utilizador, data storage, gráficos, ...
- Classes dentro de pacotes, relacionamentos de classe, interface de cada classe, métodos públicos, ...
- Atributos, métodos privados, classes internas, ...
- Métodos de implementação de código fonte

Um design pobre torna o programa mais difícil de entender e de modificar, para além de que, quanto mais extenso é o programa, piores são as consequências de um design pobre. O principal **propósito do design é controlar a complexidade** de um sistema.

Existem 2 tipos de complexidade num sistema: as complexidades essenciais (inerentes ao problema) e complexidades acidentais (são artefatos da solução). A complexidade total do sistema é a soma dos dois.

Modularidade - subdivide a solução em partes mais pequenas para ser mais fácil gerenciar os outros componentes
Abstração - usar abstração para esconder detalhes em lugares que eles não são necessários
Escondendo informações - esconder os detalhes e a complexidade por detrás de simples interfaces
Herança - reutilizar componentes gerais para definir elementos mais específicos
Composição - reutilizar outros componentes para construir uma nova solução

Características de um software design:

- **Não determinístico** - dois designs provavelmente não produzem o mesmo output.
- **Heurística** - as técnicas de design baseiam-se em heurísticas e regras práticas em vez de processos repetíveis
- **Emergente** - o design final evolui com a experiência e feedback, sendo um processo iterativo e incremental.

Processo de design:

- Perceber o problema (requisitos do software) e construir um modelo de solução (fase de construção)
- Procurar por soluções já existentes que nos ajudem a resolver alguns problemas da nossa solução
- Construir protótipos e documentação, bem como reviews do design
- Iterar (ciclo)

Características desejáveis:

- Complexidade mínima (manter o design simples)
- Acoplamento solto (minimizar a dependência entre módulos)
- Manutenção fácil
- Extensibilidade (programar o design pensando no futuro)
- Reutilização
- Portabilidade (tem que trabalhar em diferentes ambientes)
- Leanness (se for possível, evite adicionar, o custo de adicionar uma linha é mais do que os minutos que usamos para digitá-la)
- Estratificação (organizar em camadas)
- Usar técnicas padrão (às vezes evitar experimentar técnicas incertas não é boa ideia)

Padrões - são soluções reutilizáveis para problemas de design que surjam. São adaptáveis e têm vários níveis: padrões de arquitetura, padrões de design e idiomas de programação.

Capítulo 2. - Princípios de GRASP

GRASP - General Responsibility Assignment Software Patterns (descreve os princípios fundamentais do design e responsabilidades)

GRASP Principles

- **Creator**
  Quem cria a instância A?
  Atribuímos à classe B a responsabilidade de criar a instância A se um destes for verdade: B contem ou agrega A, B regista A, B usa A, B tem os dados de inicialização para A.
  Promove o baixo acoplamento, uma vez que as classes são responsáveis pela criação de objetos que elas precisam de referenciar.

- **Information Expert**
  Como atribuímos a responsabilidade a objetos?
  Atribuímos responsabilidade à classe que possui as informações necessárias para cumpri-la.
  Promove também o baixo acoplamento e facilita o encapsulamento de informação, as classes usam as próprias informações para realizar as tarefas e o código é simples de perceber. No entanto, uma classe pode se tornar demasiado complexa.

- **Low Coupling**
  Como reduzimos o impacto de mudanças e encorajamos a reutilização?
  Atribuímos a responsabilidade para que o acoplamento permaneça baixo evitando que uma classe tenha que saber sobre muitas outras.
  Classes com um forte acoplamento sofrem mudanças relacionadas a outras classes, são mais difíceis de perceber e mais difíceis de reutilizar. No entanto, o acoplamento é necessário para que as classes comuniquem entre si, desta forma duas classes devem acoplar se: classe A tem um atributo que refere a uma instância da classe B, classe A tem um método que refere a uma instância da classe B, classe A é diretamente ou indiretamente uma subclasse da classe B, classe B é uma interface e a classe A implementa-a.
  Vantagens: classes mais fáceis de entender isoladamente e que não são afetadas por mudanças de outras classes, sendo mais fáceis de reutilizar.

- **High Cohesion**
  Como manter as classes focadas e gerenciáveis?
  Atribuímos responsabilidades para que a coesão se mantenha alta (e distribuímos para que não fique apenas numa classe).
  Vantagens: fácil de entender e de fazer a manutenção e complementa o low coupling.

- **Controller**

  Quem deve ser responsável pelos eventos UI?
  Atribuímos a responsabilidade de lidar com uma mensagem de evento do sistema a uma classe.
  Vantagens: aumenta o potencial de reutilização.

- **Polymorphism**

  Como lidar com o comportamento de uma classe sem uma instrução if-else/switch?
  Usamos a chamada a um método polimórfico (dar o mesmo nome a serviços diferentes em classes diferentes, serviços esses implementados por métodos) para selecionar o comportamento em vez de utilizar as instruções if-else/switch.

- **Pure Fabrication**

  Qual objeto deve assumir a responsabilidade quando nenhuma classe pode fazê-lo sem violar a alta coesão ou o baixo acoplamento?
  Atribuímos um conjunto altamente coeso de responsabilidades a uma classe artificial.

- **Indirection**

  Como evitar o acoplamento direto?
  Como desacoplar objetos de modo a que o baixo acoplamento seja suportado e o potencial de reutilização permaneça alto?
  Atribuímos a responsabilidade a um objeto intermediário para mediar entre outros de modo a que eles não sejam diretamente acoplados.

- **Protected Variations**

  Como desenhar objetos, subsistemas e sistemas de forma a que variações dos elementos não tenham um impacto indesejável noutros elementos?
  Identificamos os pontos de variação e atribuímos responsabilidades para criar uma interface estável em torno deles.
  Mecanismos motivados pelo protected variations:
  - Core PV mechanisms
  - Data driven designs
  - ....
  - Liskov Substitution Principle (LSP)
  - Structure-hiding designs (Law of Demeter - don't talk to strangers)

  Liskov Substitution Principle

  Uma subclasse B de A deve ser substituível por uma superclasse A, ou seja, B deve ser um verdadeiro subtipo de A, isto é, B não deve remover métodos de A.

Law of Demeter

Como evitar saber sobre a estrutura de objetos indiretos?

Se duas classes não tiverem outro motivo para estarem diretamente cientes uma da outra, as classes não devem interagir.

Capítulo 3. - Padrões de Design

Padrões são princípios e soluções codificadas numa estrutura formatada, que constituem um conjunto de regras que descrevem como realizar certas tarefas no domínio do desenvolvimento de software.
Tipos de padrões:

- Padrões de arquitetura: Expressa uma organização estrutural fundamental ou esquema para sistemas de software.
- Padrões de design: Fornece um esquema para refinar os subsistemas ou componentes de um sistema de software ou os relacionamentos entre eles.
- Idiomas: Um idioma descreve como implementar aspectos específicos de componentes ou os relacionamentos entre eles usando os recursos de uma determinada linguagem.

Grupos de padrões:

- Creational: padrões relacionados ao processo de criação do objeto
- Structural: padrões relacionados com a composição da classe/objeto
- Behavioral: padrões que caraterizam a maneira como as classes e os objetos interagem entre si e que distribuem responsabilidades.

3.1 Creational Patterns

Problema: construtores no java são inflexíveis, pois sempre retornam um objeto novo (não permitindo a reutilização) e não permitem retornar um subtipo do tipo a que eles pertencem.

Factory Method

Intenção: defina uma interface para criar um objeto mas deixe as subclasses decidirem qual classe instanciar, permitindo assim que uma classe adie a instanciação para as subclasses; defina um construtor virtual e a operação *new* é prejudicial.

Permite criar um objeto sem mostrar a sua lógica de criação. (Ex.: formas geométricas -> shapefactory > shape > circle - square - ....)

- **Abstract Factory**
  Permite produzir famílias de objetos relacionados sem especificar as classes concretas. Retorna o produto imediatamente. (Uma interface é responsável por criar outros objetos sem expor as classes).
  Solução: criar interfaces que possam ser implementadas posteriormente pelos diferentes subtipos.

- **Builder**
  Permite construir objetos complexos passo a passo, permitindo produzir diferentes tipos e representações de um objeto usando o mesmo código de construção.
  Solução: extrair o código de construção do objeto da classe e mover para construtores separados. Há uma classe diretora que diz a ordem dos passos. Evitamos então ter 300000 construtores dependendo do número de argumentos passados.

- **Singleton**
  Permite garantir que uma classe tem apenas uma instância ao mesmo tempo que fornece um ponto de acesso global para essa instancia.
  Resolve 2 problemas: certifica-se que uma classe tem apenas uma única instância, assim, se criarmos um objeto e depois o quisermos modificar, não criamos um novo e sim modificamos o que já existe; providencia um ponto global de acesso à instância, protegendo assim a instância de ser overwritten.
  Definimos então o construtor como private ou protected e definimos uma private static reference para a classe do objeto, definindo também um método de acesso à instância.

- **Prototype**
  Permite copiar objetos já existentes sem tornar o código dependente da classe do objeto copiado.
  Declaramos uma comum interface para os objetos que podem ser clonados (objetos com attr privados não podem ser clonados)

- **Object Pool**

Quadro Resumo:

- Abstract Factory
  Cria uma instância de várias famílias de classes
- Builder
  Separa a construção de objetos pelo que eles representam
- Factory Method
  Cria uma instância de várias classes derivadas
- Singleton
  Uma classe da qual apenas uma única instância pode existir
- Prototype
  Copia/clona a inicialização inteira de uma instância
- Object pool
  Evite a aquisição cara e liberação de recursos reciclando objetos que não estão mais em uso.

3.2 Structural Patterns

Adapter

Permite que objetos com interfaces incompatíveis colaborem, convertendo interfaces de um objeto para outro de forma a serem compatíveis.

- **Bridge**
  Permite dividir uma grande classe ou um conjunto de classes muito relacionadas em duas hierarquias separadas: abstração e implementação, que podem ser desenvolvidas independentemente uma da outra.
- **Composite**
  Permite compor objetos em estruturas de árvore e trabalhar com essas estruturas como se fossem objetos individuais.
- **Decorator**
  Permite anexar novos comportamentos a objetos. (new dentro de new)
- **Façade**
  Fornece uma interface simplificada para uma biblioteca, uma estrutura ou outro conjunto complexo de classes.
- **Flyweight**
  Permite ajustar mais objetos na quantidade de RAM disponível, compartilhando partes comuns de estado entre vários objetos.
- **Proxy**
  Permite fornecer um substituto ou espaço reservado para outro objeto. Controla o acesso ao objeto original, gerenciando a execução de algo antes ou depois que a solicitação chega ao objeto original.

Quadro Resumo:

- Adapter
  Correspondência entre interfaces de diferentes classes
- Bridge
  Separa interfaces dos objetos para a implementação
- Composite
  Usa uma estrutura de árvore
- Decorator
  Adiciona responsabilidades aos objetos dinamicamente
- Facade
  Um única classe pode representar um subsistema inteiro
- Flyweight
  Compartilhamento de dados entre objetos
- Proxy
  Um objeto represa outro objeto

3.3 Behaviour Patterns

- **Chain of Responsibility**
  Permite passar solicitações ao longo de uma cadeia de manipuladores, ao receber a solicitação cada manipulador decide se a processa ou se a passa ao próximo manipulador da cadeia.
- **Command**
  Transforma uma solicitação num objeto independente que contém todas as informações sobre a solicitação.
- **Iterator**
  Permite percorrer os elementos de uma coleção sem expor sua representação subjacente (lista, pilha, árvore etc.).
- **Mediator**
  Permite reduzir dependências caóticas entre objetos. O padrão restringe as comunicações diretas entre os objetos e os força a colaborar apenas por meio de um objeto mediador.
- **Memento**
  Permite salvar e restaurar o estado anterior de um objeto sem revelar os detalhes de sua implementação.
- **Null Object**
  Return new NullRequest()
- **Observer**
  Permite definir um mecanismo de inscrição para notificar vários objetos sobre quaisquer eventos que acontecem ao objeto que eles estão observando.
- **State**
  Permite que um objeto altere seu comportamento quando seu estado interno muda, dando a parecer que o objeto mudou a sua classe.
- **Strategy**
  Permite definir uma família de algoritmos, colocar cada um deles em uma classe separada e tornar seus objetos intercambiáveis.
- **Template Method**
  Define o esqueleto de um algoritmo na superclasse, mas permite que as subclasses substituam etapas específicas do algoritmo sem alterar sua estrutura.
- **Visitor**
  Permite separar algoritmos dos objetos nos quais operam.

Quadro Resumo:

- Chain of responsibility
  Uma maneira de passar uma solicitação entre objetos de uma cadeia.
- Command
  Encapsular uma solicitação de um comando como um objeto.
- Interpreter
  Uma maneira de incluir elementos de linguagem num programa

- Iterator
  Acesso sequencial a elementos de uma *collection*.
- Mediator
  Definir uma forma de comunicação simples entre classes.
- Memento
  Capture e restaure o estado interno de um objeto.
- Null Object
  Projetado para atuar como um valor padrão de um objeto.
- Observer
  Uma maneira de notificar as mudanças para um número de classes.
- State
  Alterar o comportamento dos objetos quando o estado muda.
- Strategy
  Encapsular um algoritmo dentro de uma classe.
- Template method
  Adiar as etapas exatas de um algoritmo para uma subclasse.
- Visitor
  Definir uma new operação a uma classe sem a mudar.

Capítulo 4. - Padrões de Arquitetura

Layered Architecture

O conceito de **camadas de isolamento** significa que as alterações feitas em uma camada da arquitetura geralmente não impactam ou afetam os componentes em outras camadas.

A tela do cliente é responsável por aceitar a solicitação e exibir as informações.

O objeto cliente é responsável por agregar todas as informações necessárias para a solicitação de negócios.