Unidade Curricular

“Padrões e Desenho de Software”

#03 - Patterns

António José Ribeiro Neves

an@ua.pt

https://www.ua.pt/pt/uc/12275

universidade de aveiro

ECIU university

deti

IEETA

LASI

Outline

GRASP principles

Design Patterns Classification

Design patterns

Quizz #03

Practical work

General Responsibility Assignment Software Patterns (GRASP)

- **15 minutes** to explore the topic and answer the questions in the link:

https://forms.gle/MFggy5tyfqikJzxR9

GRASP

GRASP is a set of guidelines and principles used in object-oriented design to determine the assignment of responsibilities to classes and objects.

Name chosen to suggest the importance of grasping fundamental principles to successfully design object-oriented software

Describe fundamental principles of object design and responsibility

The goal of GRASP is to promote better design practices by helping developers make informed decisions about the distribution of responsibilities within their software systems.

GRASP principles help to achieve high cohesion and low coupling in software design, which are essential for building maintainable, scalable, and flexible systems.

GRASP

By applying GRASP principles, developers can create software designs that are easier to understand, maintain, and extend, leading to higher quality and more robust systems.

It's important to note that GRASP principles are not strict rules but rather guidelines to inform design decisions. The application of GRASP principles should be adapted based on the specific requirements and context of each software project.

Developers should strive to understand and apply GRASP principles effectively to achieve better software design outcomes and improve the overall quality of their codebases.

Some key GRASP principles (there are more …)

Creator: Assign the responsibility for creating instances of a class to the class itself or to a related class.

Controller: Assign the responsibility for controlling and coordinating activities within a system to a class that acts as an intermediary between user interfaces and domain objects.

Information Expert: Assign a responsibility to the class with the most information required to fulfill it, promoting encapsulation and reducing dependencies.

High Cohesion: Assign responsibilities in a way that ensures that elements within a module or class are highly related and focused on a single purpose.

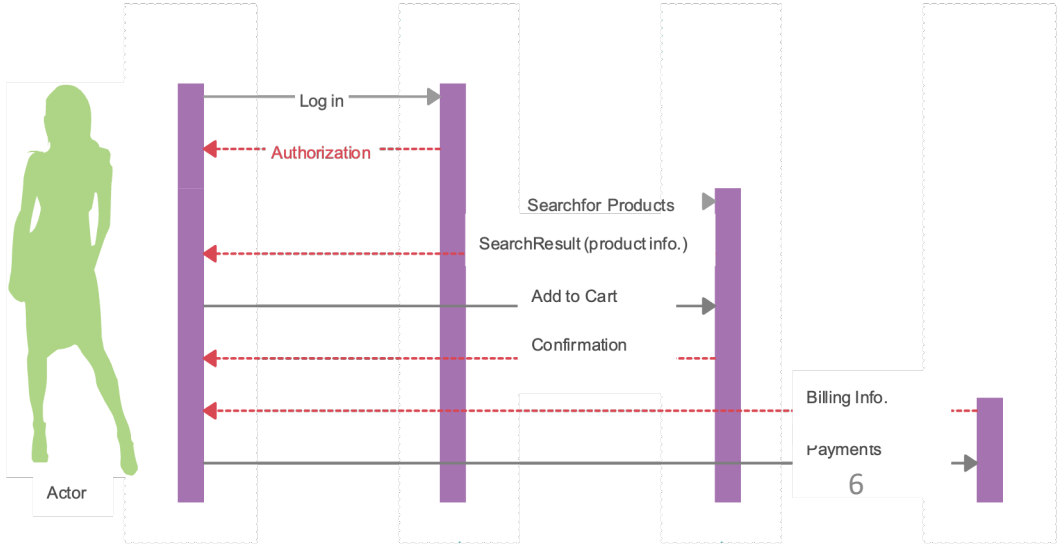
Low Coupling: Assign responsibilities in a way that minimizes dependencies between classes and modules, allowing for easier maintenance and changes.

Polymorphism: Assign responsibilities in a way that leverages polymorphism to enable interchangeable behaviors without modifying the client code.

Practical Example

- Online Shopping System
- Imagine we are tasked with designing the classes for an online shopping system. Our system should allow users to browse products, add items to their cart, and place orders.
- Explore the referred GRASP principles in this context, to gain a better understanding of how to design classes that are cohesive, loosely coupled, and well-structured, leading to more maintainable and scalable software solutions.
- **10 minutes** to think about this problem

Online Shopping System



Creator

Responsibility: Assign the responsibility for creating instances of classes to the class itself or to a related class.

Example: In our online shopping system, the ShoppingCart class could be responsible for creating instances of Product objects when users add items to their cart. Alternatively, the Order class could be responsible for creating instances of OrderLine objects when users place orders.

Promotes low coupling by making instances of a class responsible for creating objects they need to reference.

The Creator principle suggests that objects should create objects when it makes sense in terms of the responsibilities and relationships within the system:

- When the object has the required information to create the new object
- When the creation of the object is part of the object's responsibility
- When the relationship between the objects is aggregation or composition
- When there is a natural association between the creating object and the created object

Controller

Responsibility: Assign the responsibility for controlling and coordinating activities within a system to a class that acts as an intermediary between user interfaces and domain objects.

Example: The ShoppingCartController class could serve as the controller responsible for handling user interactions related to shopping cart operations, such as adding items to the cart or updating quantities. It would communicate with the ShoppingCart and Product classes to perform these actions.

If a program receive events from external sources other than its GUI, add an event class to decouple the event source(s) from the objects that actually handle the events.

Information Expert:

Responsibility: Assign a responsibility to the class with the most information required to fulfill it, promoting encapsulation and reducing dependencies.

Example: The Product class would be the information expert for retrieving details about a specific product, such as its name, price, and availability. This class encapsulates the product data and behavior related to individual products. Another example can be the ProductDatabase class.

Real-world analogy: workers in a business, bureaucracy, military. “Don’t do anything you can push off to someone else”.

Can cause a class to become excessively complex (eg. who is responsible to save data in a database?)

High Cohesion

Responsibility: Assign responsibilities in a way that ensures that elements within a module or class are highly related and focused on a single purpose.

Example: The ShoppingCart class should be highly cohesive, focusing on managing the items in the user's shopping cart. It should not be responsible for handling order processing or user authentication, as these are unrelated concerns.

Low Coupling:

- Responsibility: Assign responsibilities in a way that minimizes dependencies between classes and modules, allowing for easier maintenance and changes.
- Example: The ShoppingCart class should interact with the Product class through a well-defined interface, rather than directly accessing its internal details. This reduces coupling between the ShoppingCart and Product classes, making it easier to change the implementation of the Product class without affecting the ShoppingCart.
- In object-oriented languages, common forms of coupling include:
- Content coupling occurs when one class directly references the internal implementation details of another class
- Common coupling exists when multiple classes depend on a shared global variable or data structure
  - Control coupling occurs when one class controls the flow of execution in another class by passing control information through method parameters
  - External coupling arises when classes depend on external components, such as libraries, frameworks, or APIs
  - Data coupling occurs when classes communicate by passing data through method parameters or return values
  - Temporal coupling refers to the dependency between classes based on the timing or order of method invocations
  - …

There are more…

- An example of **pure fabrication** could be the introduction of a ShoppingCartValidator class. This class would be responsible for validating the contents of a shopping cart before proceeding with the checkout process.
- The ShoppingCartValidator class is a pure fabrication because it does not represent a real-world entity or concept in the domain of online shopping. Instead, it is created solely to fulfill the responsibility of validating shopping carts.
- An example of **polymorphism** could be the implementation of a PaymentProcessor interface with multiple concrete implementations for different payment methods.
- At runtime, depending on the payment method chosen by the user during checkout, the appropriate concrete implementation of the PaymentProcessor interface is instantiated and used to process the payment.

Let’s take a short break

10 Minutes

You are free to go grab
a coffee, water, etc.

But… 10 minutes **is 10 minutes** (600 seconds, **not 601 seconds!)**

10 minutes

Design patterns

Patterns play an important role in the design methods of today

A design pattern is a reusable solution to a commonly occurring design problem

A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations

Design patterns are adapted to the unique characteristics of the particular problem

Architecture Styles/Patterns

Design Patterns

Programming Idioms

Principles (SOLID, DRY, …)

More definitions

Design patterns are recurring solutions to common problems in software design. They are templates for solving problems and designing flexible and reusable object-oriented software.

Design patterns capture best practices and proven solutions to recurring problems encountered in software design. They provide a common language and vocabulary for software developers to communicate and share design solutions.

Design patterns are general, reusable solutions to common problems that arise during software development. They encapsulate expert knowledge and design expertise, enabling developers to leverage proven solutions and avoid reinventing the wheel.

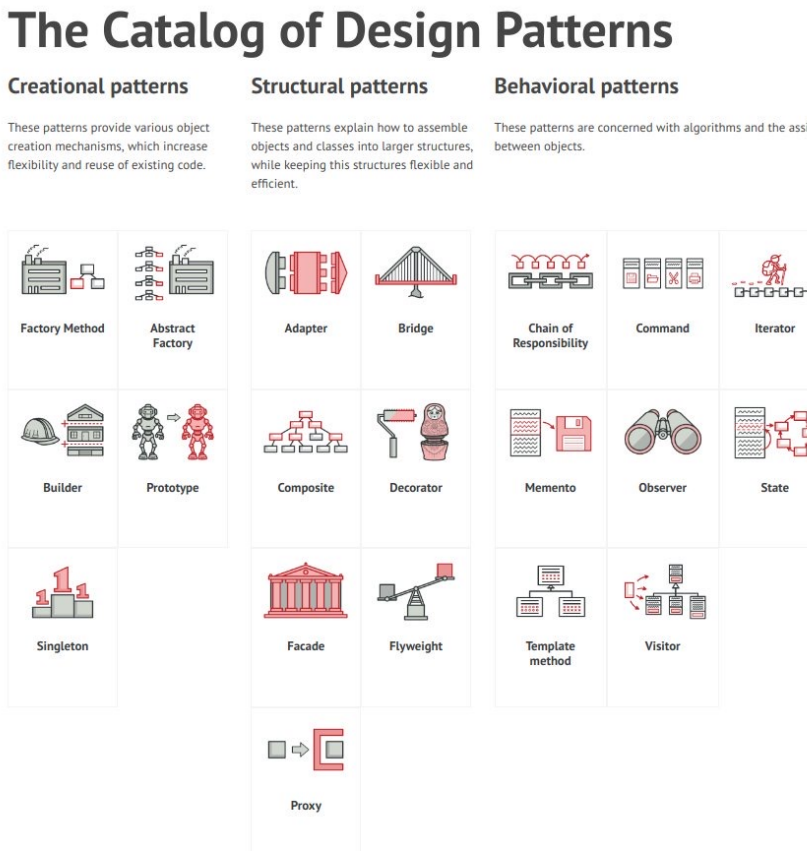
…

Patterns also exist in our daily life

- **Pattern Recognition:** Humans are adept at recognizing patterns in various aspects of life, from identifying familiar faces to discerning recurring behaviors and trends.
- **Problem Solving:** We often apply patterns to solve everyday problems efficiently. For example, we may follow a recipe (a cooking pattern) to prepare a meal or use a map (a navigation pattern) to reach a destination.
- **Routine and Habits:** Many of our daily routines and habits follow recognizable patterns. From morning rituals to bedtime routines, these patterns help structure our lives and provide a sense of order and predictability.
- **Communication:** Language itself is built on patterns, with words arranged in recognizable sequences to convey meaning. We also use patterns in non-verbal communication, such as gestures and facial expressions, to convey emotions and intentions.
- **Learning and Education:** Educational curricula often incorporate patterns to facilitate learning. Concepts are presented in a structured manner, with topics building upon one another in a logical sequence.
- **Creativity and Innovation:** Even in creative endeavors, patterns play a role. Artists and designers often draw inspiration from existing patterns or create new ones to convey their ideas effectively.
- **Social Interactions:** Social interactions follow recognizable patterns, such as greeting rituals, conversational turn-taking, and group dynamics. Understanding these patterns helps navigate social situations and build relationships.
- **Health and Well-being:** Patterns also influence our health and well-being. From sleep patterns to dietary habits, establishing healthy routines can have a significant impact on our physical and mental health.

Gang of Four (GoF) Patterns

- The Gang of Four (GoF) patterns refer to a collection of 23 design patterns documented in the book "Design Patterns: Elements of Reusable Object-Oriented Software," authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- The GoF patterns are classified into three main categories: Creational, Structural, and Behavioral. Each category addresses different aspects of software design and provides solutions to common design problems.
- The GoF patterns serve as a catalog of proven solutions to recurring problems in software design. They provide developers with a common language and vocabulary for discussing design solutions and offer reusable templates that can be adapted to different contexts.
- Follow https://refactoring.guru/design-patterns, for example.



Design Patterns Classification

- **15 minutes** to explore the topic and answer the questions in the link:

https://forms.gle/wEJh9cng5wg6TZik9