SOFTWARE RE-ENGINEERING

SE-409

Contents

- Object Oriented Re-Engineering Patterns
    - Design Patterns

Object Oriented Re-Engineering Patterns

- Object-Oriented Re-Engineering Patterns, also known as OORP, are a set of design patterns that are used to modify and improve existing object-oriented software systems.
- These patterns provide a way to analyze, refactor, and improve the design of existing software systems to make them more flexible, maintainable, and extensible.
- Design patterns play a significant role in Object-Oriented Re-Engineering Patterns (OORP).

Design Patterns

- A design pattern is a reusable solution to a common software design problem that has been proven to be effective over time.
- Design patterns provide a structured approach to solving problems in software development and can improve the quality, maintainability, and scalability of software systems.
- Design patterns are typically documented in a standardized format and can be applied to a wide range of programming languages and software systems.

Design Patterns

- They are often used to solve recurring problems in software development, such as how to create objects efficiently, how to maintain code modularity, or how to manage the communication and behavior between objects.

Benefits of using patterns

- Patterns give a design common vocabulary for software design:
  - Allows engineers to abstract a problem and talk about that abstraction in isolation from its implementation.
  - A culture; domain-specific patterns increase design speed.
  - Capture expertise and allow it to be communicated.
  - Promotes design reuse and avoid mistakes.
  - Makes it easier for other developers to understand a system.

Benefits of using patterns

- Efficiency
  - Design patterns can improve the efficiency of software development by providing a set of standard solutions to common problems.
  - This can help developers avoid time-consuming trial-and-error approaches and focus on building robust and high-quality software systems.
- Overall, using design patterns can help developers create more maintainable, and efficient software systems. By leveraging proven solutions to common software design problems, developers can save time and effort, improve collaboration and communication, and build higher-quality software systems.

Describing a pattern

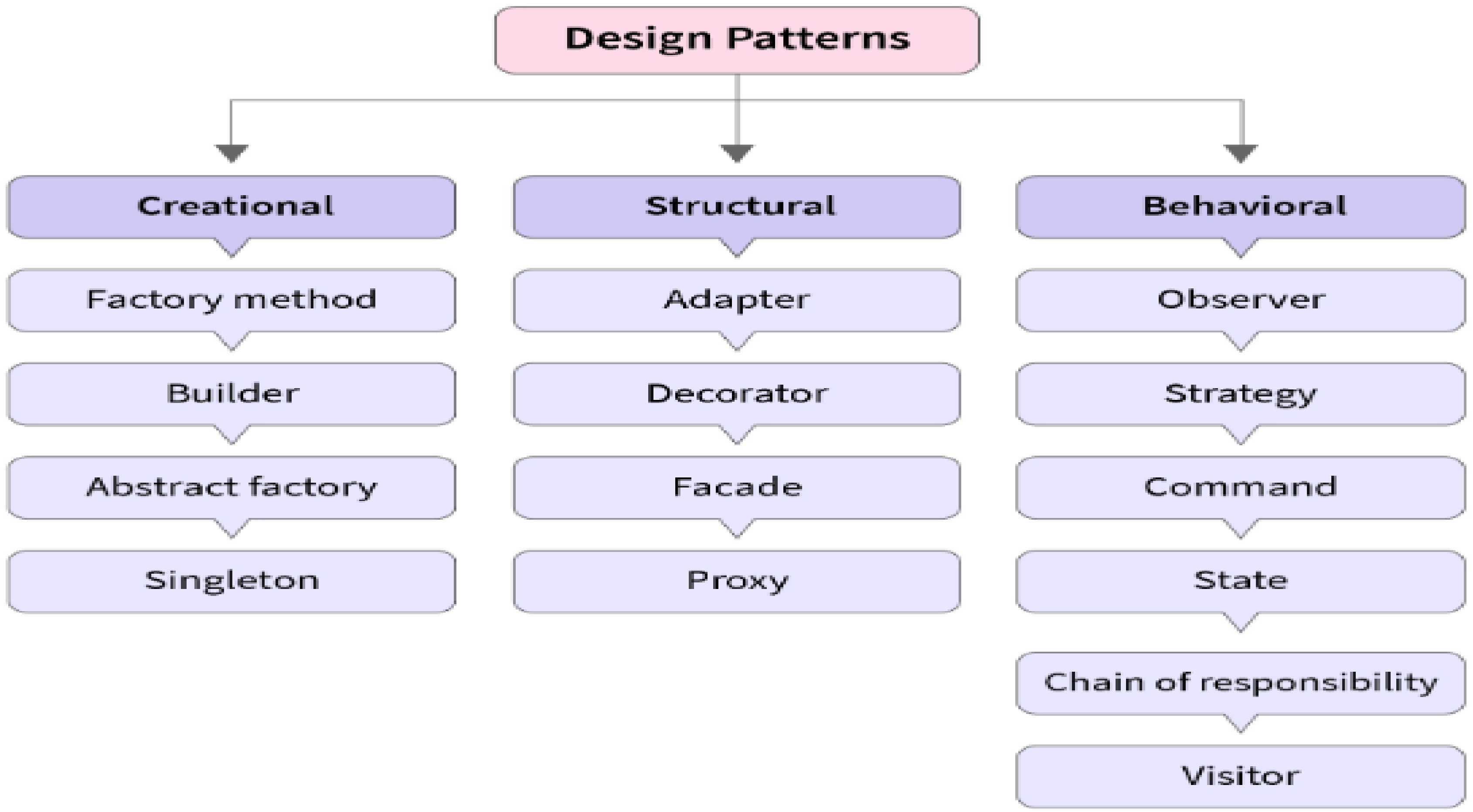
- **Problem:** In what situation should this pattern be used?
- **Solution:** What should you do? What is the pattern?
  - describe details of the objects/classes/structure needed
  - should be somewhat language-neutral
- **Advantages:** Why is this pattern useful?
- **Disadvantages:** Why might someone not want this pattern?

Categories of design patterns in software RE-engineering:

- There are three main categories of design patterns in software engineering or reengineering:

1. **<u>Creational Patterns:</u>** These patterns are concerned with object creation mechanisms, trying to create objects in a manner that is suitable to the situation.
2. **<u>Structural Patterns:</u>** These patterns are concerned with object composition and relationships between objects, helping to form large-scale structures from individual objects.
3. **<u>Behavioral Patterns:</u>** These patterns are concerned with communication and coordination between objects and classes, focusing on how objects distribute work and responsibilities.

Categories of design patterns in software RE-engineering:



Creational Patterns

- **Creational Patterns**: These patterns focus on object creation mechanisms, abstracting the instantiation process. Examples include:
  - **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
  - **Factory Method**: Defines an interface for creating an object, allowing subclasses to alter the type of objects that will be created.
  - **Prototype**: Creates new objects by copying an existing object, avoiding the need to create new instances from scratch.

Structural Patterns

- **Structural Patterns**: These patterns deal with the composition of classes or objects. They help in forming large structures from individual parts. Examples include:
  - **Adapter**: Allows incompatible interfaces to work together by providing a wrapper with a compatible interface.
  - **Decorator**: Adds new functionality to an object dynamically without altering its structure.
  - **Facade**: Provides a simplified interface to a complex system, hiding its complexities from clients.

Behavioral Patterns

- **Behavioral Patterns**: These patterns are concerned with algorithms and the assignment of responsibilities between objects. They describe communication between objects and how they collaborate to perform a task. Examples include:
  - **Observer**: Defines a one-to-many dependency between objects, ensuring that when one object changes state, all its dependents are notified and updated automatically.
  - **Strategy**: Defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable at runtime.
  - **Command**: Encapsulates a request as an object, allowing parameterization of clients with queues, requests, and operations.