SE 350 - Object-Oriented Software Development

Software Design Patterns: Intro

Instructor: Stefan Mitsch

Learning Objectives

- Understand some common design flaws
- Understand the role of design patterns
- Learn to communicate in terms of patterns
- Identify problems and their solutions

Software Design Patterns

- What is a Design Pattern?
  - A solution to commonly occurring problems
  - A customizable blueprint
  - Not a library of data structures and algorithms

Pattern

- Architecture/design-centric
- High-level description, not executable
- Blueprint of features and results

Algorithm

- Execution-centric
- Clear set of actions
- "Cooking recipe", steps towards a goal

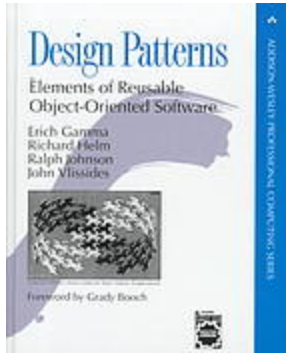
The Structure of Design Patterns

Software Design Patterns

- **Name:** concise handle of the problem and its solution
- **Problem:** when to apply the pattern; describes the problem and its context
- **Solution:** the elements that make up the design, their relationships, responsibilities, and collaborations
- **Consequences:** results and trade-offs of applying a pattern

GoF Design Patterns

- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software
- Gang of Four (GoF)
- **Program to an interface**, not to an implementation (emphasizes open-closed principle and dependency inversion principle)
- **Favor composition** (black-box reuse) over inheritance (white-box reuse)



GoF Pattern Details

- **Intent:** problem that the pattern addresses
- **Also Known As:** other names for the pattern
- **Motivation:** an illustrating scenario that illustrates the problem
- **Applicability:** how to recognize poor design and situations where pattern applies
- **Structure:** graphical represention of the pattern elements
- **Participants:** classes and objects participating in the pattern
- **Collaborations:** how participants carry out their responsibilities
- **Implementation:** pitfalls, hints, and implementation techniques
- **Sample Code:** code fragments that illustrate the pattern
- **Known Uses:** examples of pattern use in real systems
- **Related Patterns:** what other patterns are similar and what are their differences

GoF Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

Process of creating objects

- Problem: `new` is inflexible for specifying what is created when
- Intent: increase the flexibility of creating objects
- Idea: hide details about object creation and about composing objects

- **Singleton** restricts object creation for a class to one instance
- **Builder** separates object construction from representation
- **Factory Method** creates objects without specifying the exact class
- **Abstract Factory** groups object factories into themes
- **Prototype** creates objects by cloning

Structural Patterns

Composition of objects

- Problem: inflexible object and class structures
- Intent: build large systems that remain flexible
- Idea: use composition of multiple objects to achieve functionality

- **Adapter** lets classes with incompatible interfaces work together
- **Bridge** decouples abstraction from implementation
- **Composite** composes similar objects into one
- **Facade** provides a simplified interface to a large module
- **Decorator** dynamically adds or changes behavior
- **Proxy** provides a placeholder for another object
- **Flyweight** reduces the cost of creating a large numbre of similar objects

Behavioral Patterns

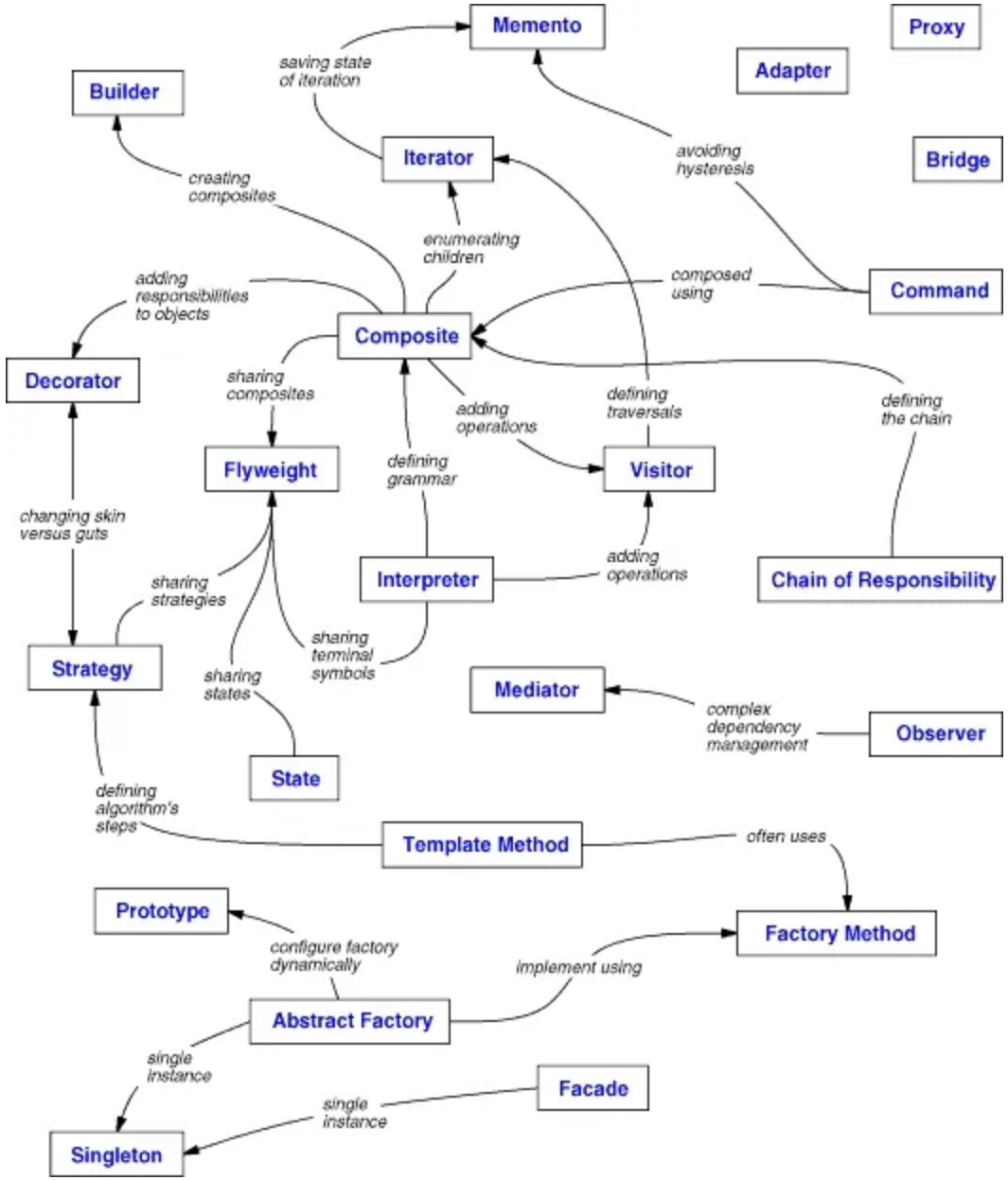
Ways of object interaction

- **Problem:** algorithms and control flow can be difficult to follow
- **Intent:** use composition to describe communication and complex control flow between objects
- **Idea:** shift focus from control flow to object composition

- **Interpreter** implements a language
- **Template Method** skeleton of an algorithm
- **Chain of Responsibility** chain of handlers
- **Command** encapsulates actions and parameters
- **Iterator** access elements sequentially
- **Mediator** enables loose coupling of objects
- **Memento** provides "undo" functionality
- **Observer** publish/subscribe to events
- **State** change behavior when state changes
- **Strategy** select one from a family of algorithms
- **Visitor** separates algorithm from object structure

Choosing a Design Pattern

- What is the problem? Read the motivation and intent section of a pattern
- What other patterns? Read relationships and comparisons between patterns
- Is a complete redesign needed? Read causes for redesign
- Analyze: what should be flexible in my design, what can be static?



Summary: Features of Good Design

Change is Inevitable

- E. Gamma on Flexibility and Reuse
- Code Reuse: reduce development cost and time
- Extensibility: prepare for additional features and library upgrades
- Early time-to-market: flexible design enables short time-to-market (with basic functionality), but prepares for additional functionality
- Communication: well-designed system architectures can be communicated in terms of patterns
- Development: well-designed systems simplify software development process
- Testing: components of well-designed systems can be tested separately