

# Design Patterns

Jing Dong  
CS6362 Software Architecture and Design

1

## Outline

- Motivation for design patterns
- Overview of design patterns
- How to learn design patterns
- What is (not) a design pattern
- Sorting example
- More patterns
- Benefits and drawbacks of patterns

2

## Pre-requisite

- Object-oriented modeling techniques
  - Concepts (encapsulation, specialization, delegation, polymorphism, etc.)
  - Notations (UML/OMT)
- Object-oriented programming languages
  - C++
  - Java
- Software engineering background

3

## Motivation for Patterns

- Developing software is hard
- Developing reusable software is even harder
- Goals of design patterns
  - Reuse design experience
  - Improve communication
  - Capture design decisions
  - Record design tradeoffs

4

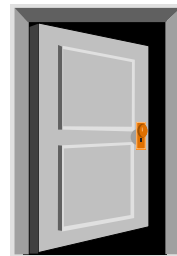
## Overview of Patterns

- Patterns support reuse of software architecture and design
  - Patterns capture the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain
- Patterns help to improve software quality and reduce development time
  - Reusability, extensibility, modularity, performance

5

## Where Does the Idea Come From?

- The idea came from architecture



6

## Learning Patterns

- Successful solutions to many areas of human endeavor are invaluable experience
- Transmitting experience documented in patterns from generation to generation is an important goal of education
- Learning to be a good designer is similar to learning to be a good architect, chess player

7

## To Be A Chess Master

- First, one must learn rules and physical requirements
  - Name of pieces, legal movements, chess board geometry and orientation, etc.
- Later, one must learn principles
  - Relative value of certain pieces, strategic value of center squares, power of a threat, etc.
- Finally, one must study the games of other masters to become a master of chess.
  - These games contain **patterns** that must be understood, memorized and applied repeatedly.
  - There are many **patterns** of chess playing

8

## To Be A Software Design Master

- First, one must learn rules
  - Data structures, algorithms, languages, etc.
- Later, one must learn principles
  - Structural programming, modular programming, object-oriented programming, generic programming.
- Finally, one must study the designs of other masters to truly master software design
  - These designs contain **patterns** that must be understood, memorized and applied repeatedly.
  - There are many **patterns** of software design

9

## Object-Oriented Principles

- Separate interface from implementation
  - Subtype versus inheritance
- Programming to interface not implementation
- Inheritance versus composition
  - White-box versus black-box reuse
- Polymorphism

10

## Object-Oriented Principles

- In summary, identify the stable and volatile parts, and encapsulate them separately, let them vary independently
- Design for change
  - Change is intrinsic to software
    - Requirements, technology, platforms
  - Robustness to change determines
    - Ease of evolution
    - Subsequent maintenance costs
    - Ultimate reusability

11

## What Is A Design pattern?

- Design patterns represent **solutions** to **problems** that arise when developing software within a particular **context**
  - **Pattern = problem/solution pairs in a context**
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs.

12

## What Makes it a Pattern?

- Solve a problem
  - It must be useful
- Have a context
  - It must describe where the solution can be used
- Recur
  - It must be relevant in other situations
- Communication
  - It must provide sufficient understanding to tailor the solution.
- Have a name
  - It must be referential consistent

13

## Common Design Vocabulary

- Design language beyond technology
  - Abstractions about problem not implementation
- “Let’s use an Observer here”
  - Increased design velocity and culture
- Shared vocabulary
  - Within/across teams, up/down management
  - Can exclude those not involved

14

## What Is NOT A Pattern

- Patterns are independent of programming language or implementation details although they are often described by them
- Patterns are not unusual programming features
- Patterns are not amazing programming tricks
- Patterns are not designs and must be instantiated
  - Evaluate tradeoffs and consequences
  - Make design and implementation decisions
  - Implement and combine with other patterns
- Patterns are not frameworks

15

## When to Use Patterns

- Solutions to problems recurring with variations
  - No need for reuse if the problem only arises in one context
- Solutions that require several steps
  - Not all problems need all steps
  - Patterns can be overkill if solution is simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
  - Patterns leave out too much to be useful to someone who really understand

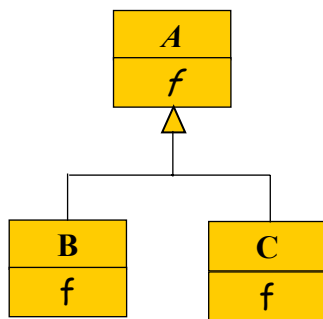
16

## Pattern Instantiation and Implementation

- Patterns are generic design pieces that need to be instantiated before uses
- It is not difficult to implement a pattern
- But, you still have to write functionality
  - A common mistake is to think patterns solve all your problems

17

## Polymorphism



```
class A {
    virtual void f() = 0;
}
class B {
    void f() { ... }
}
class C {
    void f() { ... }
}
```

```
A a;
a = new B();
a.f();
```

```
void g(A a) { ... }
b = new B();
g(b);
```

18