Design Patterns

V. Paúl Pauca

Department of Computer Science
Wake Forest University

CSC 331-631
Fall, 2013

Reusable Object-Oriented Design is Hard

From Design Patterns, Gamma et. al, 1995

Design challenges

- Finding objects, defining classes, interfaces, inheritance hierarchies, key associations, etc.
- Design should be specific to the problem but general enough to address future needs
- Design should be reusable and flexible

Reusable Object-Oriented Design is Hard

From Design Patterns, Gamma et. al, 1995

Design challenges

- Finding objects, defining classes, interfaces, inheritance hierarchies, key associations, etc.
- Design should be specific to the problem but general enough to address future needs
- Design should be reusable and flexible

Expert designers

- Know not to solve every problem from first principles
- Reuse previous successful solutions
- Evidence: recurring patterns of classes and communicating objects

Patterns

- **Design pattern**: a record of design experience

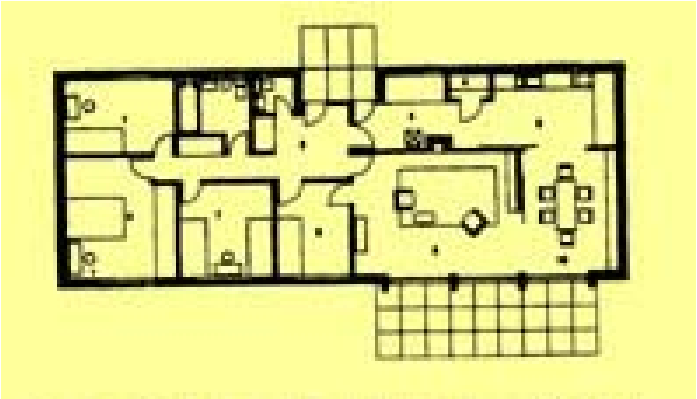
  "Each pattern describes a problem which occurs over and over again, and then describes the core of the solution to that problem, in such a way that you can use this solutoin a million times over, without ever doing it the same way twice" – C. Alexander, 1977, Architect, author of A Pattern Language

Patterns

- **Design pattern**: a record of design experience

  "Each pattern describes a problem which occurs over and over again, and then describes the core of the solution to that problem, in such a way that you can use this solutoin a million times over, without ever doing it the same way twice" – C. Alexander, 1977, Architect, author of A Pattern Language

He was talking about buildings and cities not software!



Principles of Design Patterns

1. Record of experience in software design
   Systematically name, explain, and evaluate a important and recurring OO design

Principles of Design Patterns

1. Record of experience in software design
   Systematically name, explain, and evaluate a important and recurring OO design
2. Enable reuse of successful designs and architectures
   Expressing proven techniques as design patterns makes them accessible to developers of new systems

Principles of Design Patterns

1. Record of experience in software design
   Systematically name, explain, and evaluate a important and recurring OO design
2. Enable reuse of successful designs and architectures
   Expressing proven techniques as design patterns makes them accessible to developers of new systems
3. Enable design decisions that make a system reusable

Principles of Design Patterns

1. Record of experience in software design
   Systematically name, explain, and evaluate a important and recurring OO design
2. Enable reuse of successful designs and architectures
   Expressing proven techniques as design patterns makes them accessible to developers of new systems
3. Enable design decisions that make a system reusable
4. Can also help with documentation and maintenance of existing systems

Design Patterns: Essential Elements

Four essential elements

1. **Pattern name**: Allows description of problem, solutions, and consequences in a word or two

Design Patterns: Essential Elements

Four essential elements

1. **Pattern name**: Allows description of problem, solutions, and consequences in a word or two
2. **Problem**: Situation/conditions in which the pattern applies. Describes the problem and its context

Design Patterns: Essential Elements

Four essential elements

1. **Pattern name**: Allows description of problem, solutions, and consequences in a word or two
2. **Problem**: Situation/conditions in which the pattern applies. Describes the problem and its context
3. **Solution**: Elements making up the design, relationships, responsibilities, and collaborations. No concrete implementation, rather general abstract design

Design Patterns: Essential Elements

Four essential elements

1. **Pattern name**: Allows description of problem, solutions, and consequences in a word or two
2. **Problem**: Situation/conditions in which the pattern applies. Describes the problem and its context
3. **Solution**: Elements making up the design, relationships, responsibilities, and collaborations. No concrete implementation, rather general abstract design
4. **Consequences**: Results and trade-offs of applying the pattern. Help provide a voice to important design decisions

What does a Design Pattern do?

- Names, abstracts, and identifies key aspects of a common design structure

What does a Design Pattern do?

- Names, abstracts, and identifies key aspects of a common design structure
- Identifies participating classes, objects, their roles, collaborations, and responsibilities

What does a Design Pattern do?

- Names, abstracts, and identifies key aspects of a common design structure
- Identifies participating classes, objects, their roles, collaborations, and responsibilities
- Focuses on particular OO design problem

What does a Design Pattern do?

- Names, abstracts, and identifies key aspects of a common design structure
- Identifies participating classes, objects, their roles, collaborations, and responsibilities
- Focuses on particular OO design problem
- Describes when pattern applies and under what constraints

What does a Design Pattern do?

- Names, abstracts, and identifies key aspects of a common design structure
- Identifies participating classes, objects, their roles, collaborations, and responsibilities
- Focuses on particular OO design problem
- Describes when pattern applies and under what constraints
- Describes the consequences and trade-offs of using it

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes
- Bridge: Decouple an abstraction from its implementation so that both can vary independently

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes
- Bridge: Decouple an abstraction from its implementation so that both can vary independently
- Composite: Allow a group of objects to be treated in the same way as a single instance of an object.

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes
- Bridge: Decouple an abstraction from its implementation so that both can vary independently
- Composite: Allow a group of objects to be treated in the same way as a single instance of an object.
- Facade: Provide a unified higher-level interface to a set of interfaces in a subsystem

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes
- Bridge: Decouple an abstraction from its implementation so that both can vary independently
- Composite: Allow a group of objects to be treated in the same way as a single instance of an object.
- Facade: Provide a unified higher-level interface to a set of interfaces in a subsystem
- Strategy: Define family of algorithms, encapsulate each one, and make them interchangeable

Gamma et. al Catalog of Design Patterns

Some samples:

- Abstract factory: Allow creation of families of related objects, without specifying their concrete classes
- Bridge: Decouple an abstraction from its implementation so that both can vary independently
- Composite: Allow a group of objects to be treated in the same way as a single instance of an object.
- Facade: Provide a unified higher-level interface to a set of interfaces in a subsystem
- Strategy: Define family of algorithms, encapsulate each one, and make them interchangeable
- Iterator: Prove a way to access the elements of an aggreate object sequentially without exposing its underlying representation

How Design Patterns Solve Design Problems

Finding appropriate objects

- Many conflicting factors to consider when decomposing a system into objects: encapsulation, granularity, dependency, etc.
- **How to decompose the system?**
  - Associate nouns with clases and verbs with methods
  - Look for and model collaborations and responsibilities
  - Model the real world directly
- The key is to find abstractions that make the design flexible
- Design patterns help identify less-obvious abstractions.

How Design Patterns Solve Design Problems

Determining object granularity

- How large should objects be in size and number? Objects can represent entire applications or represent individual hardware
- How to decide what should be an object?

- **Facade** describes how to represent complete subsystems as objects
- **Flyweight** describes how to support huge numbers of objects at the finest granularities
- Others describe ways to decompose an object into smaller objects

How Design Patterns Solve Design Problems

Specifying object interfaces

- Interfaces are fundamental in OO
  Objects are known only through their interfaces, their implementations are hidden
- Design patterns help identify what elements or data to include or not include in the interface
- Suppose you must be able to restore an object's data to a previous state
- **Memento** suggests defining two interfaces: 1) a restricted client interface to hold and copy *mementos*, and 2) a privileged interface for the original object to store and retrieve states
- **Decorator** and **Proxy** allow specific relationships between interfaces, requiring similar interfaces or placing constraints on the interfaces of some classes

Programming to an interface, not an implementation

- Class inheritance facilitates **implementation reuse**
  Let's a subclass obtain much of its implementation from existing classes

Programming to an interface, not an implementation

- Class inheritance facilitates **implementation reuse**
  Let's a subclass obtain much of its implementation from existing classes
- But it allows definition of objects with **identical interfaces**

Programming to an interface, not an implementation

- Class inheritance facilitates **implementation reuse**
  Let's a subclass obtain much of its implementation from existing classes
- But it allows definition of objects with **identical interfaces**

- Proper way to use inheritance:
  Derive from an **abstract class**, so that **all** subclass instances can respond to requests in the interface
  - Allows manipulation of objects based solely on their interface
  - Clients don't need to worry about the *class* of the object they use
  - Clients don't need to worry about the specific implementations

Programming to an interface, not an implementation

- Class inheritance facilitates **implementation reuse**
  Let's a subclass obtain much of its implementation from existing classes
- But it allows definition of objects with **identical interfaces**
- Proper way to use inheritance:
  Derive from an **abstract class**, so that **all** subclass instances can respond to requests in the interface
  - Allows manipulation of objects based solely on their interface
  - Clients don't need to worry about the *class* of the object they use
  - Clients don't need to worry about the specific implementations
- Key benefit: greatly reduced implementation dependencies

Favor object composition over class inheritance

Object composition, an alternative to class inheritance

New functionality obtained by assembling objects of simpler functionality, by objects acquiring references to other objects

Favor object composition over class inheritance

Object composition, an alternative to class inheritance

New functionality obtained by assembling objects of simpler functionality, by objects acquiring references to other objects

- Class inheritance
  Advantages
  - Defined statically at compile-time, easy to use
  - Easy to modify implementation being reused

Favor object composition over class inheritance

Object composition, an alternative to class inheritance

New functionality obtained by assembling objects of simpler functionality, by objects acquiring references to other objects

- Class inheritance

  Advantages

  - Defined statically at compile-time, easy to use
  - Easy to modify implementation being reused

  Disadvantages

  - Can't change implementation of inherited operations at run-time
  - Implementation dependency between parent and subclasses (breaks encapsulation)
    A change in parent's implementation often forces change in subclass implementation

Favor object composition over class inheritance

- Object composition

  Advantages

  - Defined dynamically at run-time through object references
  - Object must respect each other's interfaces (doesn't break encapsulation)
  - Objects of same type can be replaced at run-time
  - Fewer implementation dependencies

Favor object composition over class inheritance

- Object composition

  Advantages

  - Defined dynamically at run-time through object references
  - Object must respect each other's interfaces (doesn't break encapsulation)
  - Objects of same type can be replaced at run-time
  - Fewer implementation dependencies

  Disadvantages

  - Interfaces must be designed carefully
  - Can lead to system design with more objects

Favoring object composition over class inheritance helps keep classes encapsulated and focused on one task

Design based on object composition keeps the system's behavior dependent on object interrelationships

Delegation I

- Can make composition as powerful for reuse as inheritance
- Example:

- Instead of `Window` being a `Rectangle` (inheritance), `Window` has a `Rectangle`
- Delegates rectangle behavior to it, e.g. `[rectangle area];`

Delegation II

- Why use delegation instead of inheritance?
  - Easy to compose behavior at run-time (could make window circular if desired, as long as `Circle` and `Rectangle` have same type)
- Disadvantages
  - Harder to understand than static code
  - Run-time inefficiencies

Use only when it simplifies more than it complicates