Introduction to Design Patterns

CSE 332 Spring 2019

Jon Shidal

Object-Oriented Software: Why is it hard?

Must identify important **objects**

Must identify the **structural relationships between objects** (composition vs inheritance, aggregate vs acquaintance)

Must identify the **interface objects present**

Design should be flexible, extensible, and reusable:

- Allow features to be added easily to address future requirements
- Reuse existing classes to implement new features
- Avoid redesign

Design Patterns

Provide a **vocabulary** to easily document and discuss complex OO software

Each pattern gives a reusable core solution to a common design problem

- Solution can be used over and over in many different applications
- **Identify objects** that are not obvious based on analysis of the problem
- Identify **structural/behavioral relationships** between objects

Each pattern description provides information on:

- What problem does the pattern solve?
- When is it applicable?
- What classes/objects are the participants in the solution?
- What are the consequences of using the design pattern?
- Implementation details and example code

Organization:

3 types of design patterns:

1. **Creational Patterns** - describe how objects are created and composed
    a. Builder, abstract factory, factory, singleton, prototype
2. **Structural Patterns** - how objects are combined to create larger, more complex structures. Concerned with the structural relationships between objects
    a. Composite, adapter, facade, decorator, ...
3. **Behavioral Patterns** - concerned with communication between objects, responsibilities of each object
    a. Chain of responsibility, observer, iterator, visitor, **strategy**, ...

Example: The Strategy Pattern (from **[GHJV]**)

Intent: Define a family of algorithms that are interchangeable at run-time. Decouples client from the specific algorithms it uses.

Applicability: Use the strategy pattern when:

- Many related classes differ only in their behavior - create a strategy for the behavior, configure a class with a given strategy
- You need different variants of an algorithm, client can choose between variants at run-time
- An algorithm uses data a client shouldn't know about. Encapsulate algorithm specific data within a strategy

Example: Strategy pattern(from **[GHJV]**)

Participants:

- **Strategy** - declares the interface common to all algorithms
- **ConcreteStrategy** - concrete class defining the interface for a specific algorithm
- **Context**(the client):
  - Maintains a reference to a Strategy object
  - Configured to refer to a ConcreteStrategy object
  - May define an interface that lets Strategy access its data (friendship in C++)

Consider our Calculator class

Each operator requires two int arguments. The operators vary only in how they compute a result(behavior).

Participants:

- Strategy**(Calculatable) -** declares the common interface required by all operators
- ConcreteStrategy**(addable, subtractable, ...)** - concrete classes defining how the result should be calculated for a specific operator
- Context**(Calculator)** - maintains a reference to a Strategy(or in our case several strategies)

Benefits:

1. Calculator class decoupled from concrete classes it uses
    a. Can easily extend calculator functionality by adding new operators(ConcreteStrategy classes)
    b. Can easily modify ConcreteStrategy classes without needing to update the Calculator class
2. Creates a family of related algorithms that can be easily reused and extended
3. Can configure clients at run-time to use different concrete strategies