CISC 322
Software Architecture

Lecture 14:
Design Patterns

Emad Shihab

Material drawn from [Gamma95, Coplien95]
Slides adapted from Spiros Mancoridis and Ahmed E. Hassan
Motivation

- Good designers know not to solve every problem from first principles. They reuse solutions.
- Practitioners do not do a good job of recording experience in software design for others to use.
What is a Design Pattern

- A **Design Pattern** systematically names, explains, and evaluates an important and recurring design.

- “descriptions of communicating objects and classes that are customized to solve a general problem in a particular context”
Classifying Design Patterns

- **Structural**: concern the process of assembling objects and classes

- **Behavioral**: concern the interaction between classes or objects

- **Creational**: concern the process of object creation
Design Patterns Covered

- **Structural**
  - Adapter
  - Façade
  - Composite

- **Behavioral**
  - Iterator
  - Template
  - Observer
  - Master-Slave

- **Creational**
  - Abstract Factory
For Each Pattern ....

- **Motivation** – the problem we want to solve using the design pattern
- **Intent** – the intended solution the design pattern proposes
- **Structure** – How the design pattern is implemented
- **Participants** – the components of the design pattern
Terminology

■ **Objects** package both data and the procedures that operate on that data.

■ Procedures are typically called **methods** or **operations**.

■ An object performs an operation when it receives a **request** (or **message**) from a **client**.
Terminology

- An object’s implementation is defined by its **class**. The class specifies
  - Object’s internal data and representation
  - Operations that that object can perform

- An **abstract class** is one whose main purpose is to define a common interface for its subclass
Terminology

- The set of signatures defined by an object’s operations or methods is called the **interface**
Adapter Pattern - Intent

- Convert the interface of a class into another interface clients expect.

- Adapter lets classes work together that otherwise couldn’t because of incompatible interfaces
When we want to reuse classes in an application that expects classes with a different interface, we do not want (and often cannot) to change the reusable classes to suit our application.
Adapter

Editor

Shape

BoundingBox()
CreateManipulator()

Interface for graphical object

LineShape

BoundingBox()
CreateManipulator()

Subclass of shape defined by editor for lines

Define TextShape to adapt TextView interface to Shape’s

TextShape

BoundingBox()
CreateManipulator()

Can change TextView class so it conforms to Shape interface ... would need source code of TextView. Too much work!

Text

textView

GetExtent()

Return text -> GetExtent()

Return new Text Manipulator

Allows objects to be ‘dragged’ interactively

OTS UI toolkit. Provides sophisticated class for displaying and editing text
Adapter Pattern Structure

Collaborates with objects conforming to the target interface

Defines the application-specific interface that clients use

Defines an existing interface that needs adapting

Client -> Target

Request()

Adaptee

SpecificRequest()

Adapts the interface of the adaptee to the target interface

Adapter

Request()

SpecificRequest()
Façade Pattern Intent

- Provide a unified interface to a set of interfaces in a subsystem.

- Facade defines a higher-level interface that makes the subsystem easier to use.
Façade Pattern Motivation

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- Use a façade object to provide a single, simplified interface to the more general facilities of a subsystem.
Façade Example – Programming Environment

- Programming environment that provides access to its compiler
- Contains many classes (e.g. scanner, parser)
- Most clients don’t care about details like parsing and code generation…just compile my code!
- The low-level interfaces just complicate their task
Façade Example – Programming Environment

- Higher-level interface (i.e., Compiler class) shields clients from low level classes
- Compiler class defines a unified interface to the compiler’s functionality
- Compiler class acts as a Façade. It offers clients a simple interface to the compiler subsystem
Façade Pattern Structure

Client Classes

Facade

Subsystem Classes
Participants of Façade Pattern

- Façade (compiler)
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects

- Subsystem classes (Scanner, Parser, etc..)
  - Implements subsystem functionality
  - Handles work assigned by the façade object
Use a façade when

- To provide a simple interface to a complex subsystem
- To decouple clients and implementation classes
- To define an entry point to a layered subsystem
Façade Pattern Collaborations

- Clients communicate with the subsystem by sending requests to façade, which then forwards requests to the appropriate subsystems.

- Clients that use the façade don’t have access to its subsystem objects directly. However, clients can access subsystem classes if they need to.
Composite Pattern Intent

- Lets clients treat individual objects and compositions of objects uniformly
If the composite pattern is not used, client code must treat primitive and container classes differently, making the application more complex than necessary.
Graphic applications allow users to build complex diagrams out of simple components.

Users group components to form larger components.

Primitive graphical objects:
- Line
  - Draw()
- Rect.
  - Draw()
- Text
  - Draw()
- Picture
  - Draw()
  - Add(Graphic)
  - Remove(Graphic)
  - GetChild(int)

forall g in graphics
  g.Draw()
A simple implementation defines classes for graphical primitives (e.g. Text and lines) plus other classes that act as containers for these primitives.

The problem is user must treat primitive and container objects differently.

Having to distinguish these objects makes applications more complex.

forall g in graphics g.Draw()
Key is an abstract class that represents both primitives and their containers.

Graphic declares operations such as draw that are specific to graphical objects.

Also operations for accessing and managing children.

forall g in graphics
g.Draw()
Structure of Composite Pattern

Client

Component

Leaf

Composite

Operation()
Add(Component)
Remove(Component)
GetChild(int)

Declares interface for objects and child components

Manipulates objects in the composition through Component interface

Defines behavior for primitive objects. Leafs have no children

Defines behavior for components having children. Implements child-related operations

forall g in children
g.Operation()