Chapter 8, Object Design: Introduction to Design Patterns

- During Object Modeling we do many transformations and changes to the object model.
- It is important to make sure the object design model stays simple!
- In the next two lectures we show how to use design patterns to keep system models simple.

Modeling Heuristics

- Modeling must address our mental limitations:
  - Our short-term memory has only limited capacity (7±2)
  - Good models deal with this limitation, because they...
    - ... do not tax the mind
    - A good model requires only a minimal mental effort to understand
    - ... reduce complexity
    - Turn complex tasks into easy ones (by good choice of representation)
    - Use of symmetries
    - ... use abstractions
    - taxonomies
    - ... have organizational structure:
      - Memory limitations are overcome with an appropriate representation ("natural model").

Finding Objects

- The hardest problems in object-oriented system development are:
  - Identifying objects
  - Decomposing the system into objects
- Requirements Analysis focuses on application domain:
  - Object identification
- System Design addresses both, application and implementation domain:
  - Subsystem Identification
  - Object Design focuses on implementation domain:
    - Additional solution objects

Techniques for Finding Objects

- Requirements Analysis
  - Start with Use Cases. Identify participating objects
  - Textual analysis of flow of events (find nouns, verbs, ...)
  - Extract application domain objects by interviewing client (application domain knowledge)
  - Find objects by using general knowledge
- System Design
  - Subsystem decomposition
  - Try to identify layers and partitions
- Object Design
  - Find additional objects by applying implementation domain knowledge

Another Source for Finding Objects: Design Patterns

- What are Design Patterns?
  - A design pattern describes a problem which occurs over and over again in our environment
  - Then it describes the core of the solution to that problem, in such a way that you can use the this solution a million times over, without ever doing it the same twice
What is common between these definitions?

- **Definition Software System**
  - A software system consists of subsystems which are either other subsystems or collection of classes

- **Definition Software Lifecycle**:
  - The software lifecycle consists of a set of development activities which are either other activities or collection of tasks

Introducing the Composite Pattern

- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly

What is common between these definitions?

- **Software System**:
  - Definition: A software system consists of subsystems which are either other subsystems or collection of classes
  - Composite: Subsystem (A software system consists of subsystems which consists of subsystems, which consists of subsystems, which...) Leaf node: Class

- **Software Lifecycle**:
  - Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
  - Composite: Activity (The software lifecycle consists of activities which consist of activities, which consist of activities, which....) Leaf node: Task.

Modeling a Software System with a Composite Pattern

- User
- Software System
- Class
- Subsystem

Modeling the Software Lifecycle with a Composite Pattern

- Manager
- Software Lifecycle
- Task
- Activity
- Children
Graphic Applications also use Composite Patterns

- The Graphic class represents both primitives (Line, Circle) and their containers (Picture)

```
Line Draw()
Circle Draw()
Picture Draw()
Add(Graphic g)
RemoveGraphic)
GetChild(int)
```

Reducing the Complexity of Models

- To communicate a complex model we use navigation and reduction of complexity
  - We do not simply use a picture from the CASE tool and dump it in front of the user
  - The key is navigate through the model so the user can follow it
- We start with a very simple model
  - Start with the key abstractions
  - Then decorate the model with additional classes
- To reduce the complexity of the model further, we
  - Look for inheritance (taxonomies)
    - If the model is still too complex, we show subclasses on a separate slide
  - Then we identify or introduce patterns in the model
  - We make sure to use the name of the patterns.

Example: A Complex Model

Exercise

- Redraw the complete model for Project from your memory using the following knowledge
  1. The key abstractions are task, schedule, and participant
  2. Work product, Task and Participant are modeled with composite patterns, for example

```
Work
Product
```

- There are taxonomies for each of the key abstractions
  You have 7 minutes!

Adapter Pattern (See Last Lecture)

```
Client

_CLIENTINTERFACE
Request()

LegacyClass

EXISTINGREQUEST()

adapter

Inheritance

Adapter

Request()

Delegation
```

The adapter pattern uses inheritance as well as delegation:
- Interface inheritance is used to specify the interface of the Adapter class.
- Delegation is used to bind the Adapter and the Adaptee
Adapter Pattern

- The adapter pattern lets classes work together that couldn’t otherwise because of incompatible interfaces
  - “Convert the interface of a class into another interface expected by a client class.”
  - Used to provide a new interface to existing legacy components (Interface engineering, reengineering).
- Two adapter patterns:
  - Class adapter:
    - Uses multiple inheritance to adapt one interface to another
  - Object adapter:
    - Uses single inheritance and delegation
- Object adapters are much more frequent.
- We cover only object adapters (and call them adapters).

Bridge Pattern

- Use a bridge to “decouple an abstraction from its implementation so that the two can vary independently” (From [Gamma et al 1995])
- Also know as a Handle/Body pattern
- Allows different implementations of an interface to be decided upon dynamically.

Why the Name Bridge Pattern?

Motivation for the Bridge Pattern

- Decouples an abstraction from its implementation so that the two can vary independently
- This allows to bind one from many different implementations of an interface to a client dynamically
- Design decision that can be realized any time during the runtime of the system
  - However, usually the binding occurs at start up time of the system (e.g., in the constructor of the interface class)

Using a Bridge

- The bridge pattern can be used to provide multiple implementations under the same interface
  - Interface to a component that is incomplete (only Stub code is available), not yet known or unavailable during testing
  - If seat data are required to be read, but the seat is not yet implemented (only stub code available), or only available by a simulation (AIM or SART), the bridge pattern can be used:

```
VIP
Seat (in Vehicle Subsystem)
  imp
  SelfPosition()
  GetPosition()

Stub Code
AIMSeat
SARTSeat

SeatImplementation
```
Seat Implementation

```java
public interface SeatImplementation {
    public int GetPosition();
    public void SetPosition(int newPosition);
}
```

```java
public class Stubcode implements SeatImplementation {
    public int GetPosition() {
        // stub code for GetPosition
    }
}
```

```java
public class AimSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the AIM simulation system
    }
}
```

```java
public class SARTSeat implements SeatImplementation {
    public int GetPosition() {
        // actual call to the SART seat simulator
    }
}
```

Another use of the Bridge Pattern:
Support multiple Database Vendors

Adapter vs Bridge

- **Similarities:**
  - Both are used to hide the details of the underlying implementation.
- **Difference:**
  - The adapter pattern is geared towards making unrelated components work together
  - Applied to systems after they're designed (reengineering, interface engineering).
  - "Inheritance followed by delegation"
  - A bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.
  - Green field engineering of an "extensible system"
  - New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.
  - "Delegation followed by inheritance"

Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)

Facade Pattern Diagram

Subsystem Design with Façade, Adapter, Bridge

- The ideal structure of a subsystem consists of
  - an interface object
  - a set of application domain objects (entity objects) modeling real entities or existing systems
  - Some of the application domain objects are interfaces to existing systems
  - one or more control objects
  - We can use design patterns to realize this subsystem structure
  - Realization of the Interface Object: Facade
  - Provides the interface to the subsystem
  - Interface to existing systems: Adapter or Bridge
  - Provides the interface to existing system (legacy system)
  - The existing system is not necessarily object-oriented
Realizing an Opaque Architecture with a Facade

• The subsystem decides exactly how it is accessed.
• No need to worry about misuse by callers.
• If a façade is used the subsystem can be used in an early integration test.
  • We need to write only a driver.

When should you use these Design Patterns?

• A façade should be offered by all subsystems in a software system who services.
  • The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed.
  • The adapter design pattern should be used to interface to existing components.
    • Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer.
  • The bridge design pattern should be used to interface to a set of objects.
    • where the full set of objects is not completely known at analysis or design time.
    • when a subsystem or component must be replaced later after the system has been deployed and client programs use it in the field.

Summary

• Design patterns are partial solutions to common problems such as
  • such as separating an interface from a number of alternate implementations
  • wrapping around a set of legacy classes
  • protecting a caller from changes associated with specific platforms
• A design pattern consists of a small number of classes
  • uses delegation and inheritance
  • provides a modifiable design solution
• These classes can be adapted and refined for the specific system under construction
  • Customization of the system
  • Reuse of existing solutions.

Summary II

• Composite Pattern:
  • Models trees with dynamic width and dynamic depth
• Facade Pattern:
  • Interface to a subsystem
  • Distinguish between closed vs open architecture
• Adapter Pattern:
  • Interface to reality
• Bridge Pattern:
  • Interface to reality and prepare for future