Object Oriented
Analysis and Design

Oded Lachish
Department of Computer Science
and Information Systems
Birkbeck College, University of London

Email: oded@dcs.bbk.ac.uk
Web Page: http://www.dcs.bbk.ac.uk/~oded
Previously on OOAD

- GRASP design patterns
  - Information Expert
  - Creator
  - Low Coupling
  - High Cohesion
  - Controller

  (don’t think in patterns, think patterns)
Classification (GoF – Gang of Four)

- Creational Patterns
  - Abstract object instantiation

- Structural Patterns
  - Compose and organise objects into larger structures

- Behavioural Patterns
  - Algorithms, interactions and control flow between objects.
Previously on OOAD

- List of Advanced patterns we already discussed
  - **Strategy** (behavioural)
  - **Builder** (creational)
  - **Decorator** (structural)
  - Factory Method
  - Abstract Factory
  - Singleton
More

Advanced Patterns

Bridge
**Problem:** We want to decouple an abstraction from its implementation

Why can’t we settle for inheritance?
Bridge Pattern

**Solution:** delegate implementation

**Problem:** there maybe different AI’s, with added responsibilities

```cpp
void AI::travel() {
    ImpAI->drive();
    ImpAI->swim();
}
```
**Bridge Pattern**

**Solution:** Inheritance

- **Abstraction**
  - operationAlpha ()
  - operationBeta ()

- **Implementor**
  - impOperation1 ()
  - impOperation2 ()

- **RefinedAbst1**
  - operationAlpha ()
  - operationBeta ()
  - operationGamma ()

- **RefinedAbst2**
  - operationAlpha ()
  - operationBeta ()
  - operationDelta ()

- **ConcreteImplementorA**
  - impOperation1 ()
  - impOperation2 ()

- **ConcreteImplementorB**
  - impOperation1 ()
  - impOperation2 ()
Discussion

Why use?

- Avoiding permanent binding
- Abstraction and implementations are extensible
- Implementation changes have no impact on clients
- Implementation hidden
- Implementations can be shared

(what is the difference from strategy)
State
DriverAI

- We want to simulate an AI that among other things drives
- To make it feel “human” it should have moods
- The driving and only the driving should depend on the AI’s mood

One solution is to, give it a mood attribute.

What is the problem with this solution?
State Pattern

**Solution:** delegate mood dependent behaviors
State Pattern, Consequences

- Easy to add more states
- More classes
- Less code
- State transition appears explicitly in the DCD
State Pattern, What’s Missing?

Who is responsible for the state transitions?

- Context
- ConcreteState classes

What are the pros and cons of each option?
Discussion

Why Use?
- Objects behaviour depends on its state + messy code to implement this

Issues
- When are states created
- How to add more states
Back to GRASP
More GRASP patterns

- Information Expert, Creator, Low Coupling, High Cohesion, Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations
Polymorphism
Pattern Name: Polymorphism

- **Problem It Solves:** How to handle alternatives based on type? How to create pluggable software components?
- **Solution:** When related alternatives or behaviour vary by type, assign responsibilities for the behaviour using polymorphism

 Alternatives based on type – replacing code by classes and types  
Pluggable software components – viewing components in client server relationships, how can you replace one server component with another without affecting the client
Where have we seen Polymorphism

- GoF Design Patterns:
  - Strategy
  - State
  - Bridge
  - Abstract Factory
Indirection
Pattern Name: Indirection

- **Problem It Solves:** Where to assign a responsibility, to avoid direct coupling between two or more things
- **Solution:** Assign the responsibility to an intermediate object
Where have we seen Indirection

- GoF Design Patterns:
  - Strategy
  - State
Pure Fabrication
Pattern Name: Pure Fabrication

- **Problem It Solves:** What object should have the responsibilities, when you don’t want to violate high cohesion and low coupling, or other goals, but solutions offered by Expert (for example) are not appropriate.

- **Solution:** Assign a highly cohesive set of responsibilities to an artificial class that does not represent a problem domain concept (such a class is a fabrication of the imagination).
Where have we seen Pure Fabrication

- GoF Design Patterns:
  - Strategy
  - Abstract Factory
Protected Variation
**Pattern Name:** Protected Variation

- **Problem It Solves:** How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

- **Solution:** Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.
Where have we seen Protected Variation

- GoF Design Patterns:
  - Strategy
  - Decorator
  - Abstract Factory
  ...

...
Advanced Patterns Again

MVC
Example: Model-View-Controller (MVC)

Model-View-Controller group of classes used to build interfaces (originally in Smalltalk, Java Swing, EPOC uses MVC).

- Model - application object,
- View - presentation on screen,
- Controller - how user input controls the interface
Two views of the data
MVC

- Model must notify views of change.
- Views must keep themselves up to date.
- Several controllers e.g. command keys, pop-up menus can be used. Usually controllers organised in a class hierarchy -> sub-classing
MVC Advantages

- MVC decouples the model management (data) from the representation (views) and the reaction to user input (controllers). Increased flexibility.

- Allows to have multiple (synchronized views) on the same data.

- Allows to associate different controllers with each of the views if needed. Change the way in which the interface reacts without changing the interface.

- Views can be easily composed.
Observer
PassengerAI

- We want to simulate an AI that is the passenger in a car when the AI, from the state pattern is the driver.
- To make the new AI feel “human” it should change its behavior according to how the driver AI drives.

What is the problem with this solution?

Maybe we don’t even know how many passengers are in the care?

- DriverAI
  - mood
  - notify ()
- PassengerAI
PassengerAI

- Solution: “teach them the subject observer relation”
Discussion

Why Use?

• A change to one object requires change to another object
• One object can notify others without any dependency on their class

Issues

• Need some mechanism to say what changed
• Without maintenance, may get an unwanted cascade of updates
Adapter
Problem: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?

Solution: Convert the original interface of a component into another interface through an intermediate adapter object
Class Adapter

**How:** Inheritance

- **client**
- **Target**
  - request()
- **Adaptee**
  - specificRequest()
- **Adapter**
  - request()

All the Gluing code (implementation)
Object Adapter

**How:** Inheritance

client → Target → Adaptee

Adapter

request() → specificRequest()
Discussion

Why Use?

- Want to use other people code
- Not just use other peoples code but override some of it

Class Adapter advantages:
- Simpler

Class Adapter Issues
- Works only for one specific class and not its subclasses

Object Adapter Issues
- Harder to override adaptee behaviour
**Problem:** How to store the internal state of an object without violating encapsulation?

**Motivation:**

- Support undo
This doesn’t end here!

Only the originator should be able to access and store information in the memeto

How?
class State;
Class Originator {
public:
    Memento* createMemento* CreateMemento();
    void setMemento(const Memento*);
    //..
private:
    State* _state; // internal data structure
};
class Memento {
public:
    virtual Memento();
private:
    friend class Originator; // Private member accessible only to Originator
    Memento();
    void setState(State*);
    State* getState();
    //..
private:
    State* _state;
    //..
};
Discussion

Why Use?

• Want to store internal state without
• Not just use other peoples code but override some of it

Issues

• More classes
• Works in C++ what about other languages
• Cohesion
Chain of Responsibility
Problem: How to handle a request when it is not known which object should deal with it?

Motivation: Recall the monopoly game

- In the example (with the GRASP pattern) we had a controller.
- What if in order to avoid cohesion we need many controllers, one for each complex operation
- On top of that we want to be able to add more operations
Chain of Responsibility

- **Client**
  - **Handler**
    - `handleRequest()`
  - **ConcreteHandler1**
    - `handleRequest()`
  - **ConcreteHandler2**
    - `handleRequest()`
Class HelpHandler {
public:
    HelpHandler(HelpHandler* s) : _successor(s) {}  
    virtual void HandleHelp(int code);  
    //..
private:
    HelpHandler* _successor;
};

Void HelpHandler::HandleHelp() {
    if(_successor){
        _successor->HandleHelp(int code);
    }
}

Class controller1: public HelpHandler {
Public:
    virtual void HandleHelp(int code);  // overrides original function
}
Discussion

Why Use?

• Easy to add new request handlers
• No need to specify who handles the request

Issues

• Representing requests
• Not guaranteed that request will be answered