GOALS

Today you will learn

• what design patterns are

• how to use them
DESIgn Patterns

• What are design patterns
• Some patterns in detail
• Pattern catalogue
• How to use patterns
DESIGN PATTERNS
DESIGN PATTERNS

• Well-design solution to common problem.

• Collected into catalogues (called pattern languages)

• Follow stylised presentation format

• Have accompanying code!
THE GANG OF FOUR

The original book that started it all. Applies Christopher Anderson’s architectural design patterns to software.
Figure 1.1: Design pattern relationships
ESSENTIALS OF A DESIGN PATTERN

• A **pattern name** by which we can call it – let’s us talk about design at a higher level of abstraction.

• The **problem** to which the pattern applies.

• The **solution**, consisting of elements which make up the design, their relationships, responsibilities and collaborations.

• The **consequences**, namely the results and trade-offs of applying the pattern.
DESIGN PATTERN INGREDIENTS

- Pattern Name and Classification
- Intent
- Also known as
- Motivation
- Applicability
- Structure
- Participants

- Collaborations – how participants carry out their responsibilities
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns
WHERE VARIATION OCCURS

- how composite gets created
- (sub)class that is instantiated
- number of instances of object
- interface to object
- implementation of object
- structure and composition of object
- responsibilities of object
- interface to subsystem
- storage costs of objects
- how object is accessed
- an algorithm
- state of an object
- steps of an algorithm
- ....
DESIGN PATTERNS IN DETAIL
ABSTRACT FACTORY
ABSTRACT FACTORY

An interface for creating families of related dependent objects.
EXAMPLE PROBLEM

An application needs to run using two (or more) different window managers – one for Android and one for desktop.

Want the application to be the same, regardless of look-and-feel of widget set.

Need to separate core functionality from creation of actual widgets.
EXAMPLE

**WidgetFactory**
- CreateScrollBar()
- CreateWindow()

**MotifWidgetFactory**
- CreateScrollBar()
- CreateWindow()

**PMWidgetFactory**
- CreateScrollBar()
- CreateWindow()

**Client**

**Window**

**ScrollBar**

**PMWindow**
**MotifWindow**

**PMScrollBar**
**MotifScrollBar**
APPLICABILITY OF ABSTRACT FACTORY

system is independent of how its products are created, composed and represented.

a system should be configured with one of multiple families of products.

a family of related products is designed to be used together and you want to enforce this constraint.

you want to provide a class library of products, and you want to reveal their interfaces, not their implementation.
ABSTRACT FACTORY

AbstractFactory
CreateProductA()
CreateProductB()

ConcreteFactory1
CreateProductA()
CreateProductB()

ConcreteFactory2
CreateProductA()
CreateProductB()

Client

AbstractProductA
ConcreteProductA1
ConcreteProductA2

AbstractProductB
ConcreteProductB1
ConcreteProductB2
interface MazeFactory {
    Maze makeMaze();
    Wall makeWall();
    Room makeRoom(int n);
    Door makeDoor(Room r1, Room r2);
}

Maze createMaze(MazeFactory factory) {
    Maze aMaze = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door aDoor = factory.makeDoor(r1, r2);

    aMaze.addRoom(r1); aMaze.addRoom(r2);

    ...
    return aMaze();
}
abstract class EnchantedMaze implements MazeFactory {
    Room makeRoom(int n) {
        return new EnchangedRoom(n, castSpell());
    }

    Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1, r2);  
    }  

    private Spell castSpell() {
        ...
    }
}


CONSEQUENCES

+ *It isolates concrete classes* – their names don’t appear in client code.

+ *It makes exchanging product families easy.*

+ *It promotes consistency among products* – ensuring different products from the same family are used together.

- *Supporting new kinds of products is difficult* – the AbstractFactory interface fixes the products that can be created.
DECORATOR
DECORATOR

Attach additional responsibilities to an object dynamically.

Decorators provide a flexible alternative for extending functionality.
DECORATOR EXAMPLE

```
DECORATOR EXAMPLE

aBorderDecorator → :aScrollDecorator → :aTextView

Enter more text to see scrollbars
Ok, here is some more text that will run past the width of the
```
DEDECORATOR PATTERN

Component
Operation()

ConcreteComponentA
Operation()

Decorator
Operation()

ConcreteDecoratorA
addedState
Operation()

BorderDecorator
Operation()
AddedBehaviour()

Component.Operation()

super.Operation()
AddedBehaviour()
class VisualComponent {
    void draw() { .. }
    void resize() { .. }
}

abstract class Decorator extends VisualComponent {
    private VisualComponent _component;

    Decorator(VisualComponent component) {
        _component = component;
    }

    void draw() {
        _component.draw();
    }

    void resize() {
        _component.resize();
    }
}
class BorderDecorator extends VisualComponent {
    private int _width;

    BorderDecorator(VisualComponent component, int borderWidth) {
        super(component);
    }

    void drawBorder(int width) {
        ...
    }

    void draw() {
        super.draw();
        drawBorder(_width);
    }
}
CONSEQUENCES

+ More flexibility than static inheritance – avoids classes like BorderedScrollableTextView. Can also add a property twice.

+ Avoids feature-laden classes high up in the hierarchy – objects contain only feature you need.

– A decorator and its component aren’t identical – cannot rely on object identity when using decorators.

– Lots of little objects – behaviour in interconnections, harder to debug.
VISITOR
VISITOR

Represents an operation to be performed on the elements of an object structure.

Visitor lets you define a new operation without changing the classes of the elements on which it operations.
VISITOR EXAMPLE

Visitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

TypeCheckingVisitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

CodeGeneratingVisitor
- VisitAssignment(AssignmentNode)
- VisitVariableRef(VariableRefNode)

Program

Node
- Accept(NodeVisitor)

VariableRefNode
- Accept(NodeVisitor v)
- v.VisitAssignment(this)

AssignmentNode
- Accept(NodeVisitor v)
- v.VisitRefNode(this)
VISITOR
CLASS DIAGRAM

Client

Visitor
  VisitConcreteElementA(ConcreteElementA)
  VisitConcreteElementV(ConcreteElementB)

ConcreteVisitor1
  VisitConcreteElementA(ConcreteElementA)
  VisitConcreteElementV(ConcreteElementB)

ConcreteVisitor2
  VisitConcreteElementA(ConcreteElementA)
  VisitConcreteElementV(ConcreteElementB)

ObjectStructure

Element
  Accept(Visitor)

ConcreteElementA
  Accept(Visitor v)
  OperationA()

ConcreteElementB
  Accept(Visitor v)
  OperationB()

v.VisitConcreteElementA(this)

v.VisitConcreteElementB(this)
VISITOR INTERACTION DIAGRAM

:anObjectStructure

Accept(aVisitor)

:aConcreteElementA

OperationA()

VisitConcreteElementA(aConcreteElementA)

:aConcreteElementB

OperationB()

VisitConcreteElementB(aConcreteElementB)

:aConcreteVisitor

Accept(aVisitor)
interface Visitor {
    void visitElementA(ElementA);
    void visitElementB(ElementB);
    void visitCompositeElement(CompositeElement);
}

class PrintingVisitor implements Visitor{
    void visitElementA(ElementA a) {
        // do stuff to print a
    }

    void visitElementB(ElementB b) {
        // do stuff to print a
    }

    void visitCompositeElement(CompositeElement ce) {
        // do stuff to print ce
        // Note: visitor has visited children already
    }
}
abstract class Element {
    void accept(Visitor);
}

class ElementA {
    void accept(Visitor v) {
        v.visitElementB(this);
    }
}

class ElementB {
    void accept(Visitor v) {
        v.visitElementB(this);
    }
}

class CompositeElement {
    private List<Element> _children;
    void accept(Visitor v) {
        for (Element e : _children) {
            e.accept(v);
        }
        v.visitCompositeElement(v);
    }
}
class PrintingVisitor implements Visitor{
    void visitElementA(ElementA a) {
        // do stuff to print a
    }

    void visitElementB(ElementB b) {
        // do stuff to print a
    }

    void visitCompositeElement(CompositeElement ce) {
        // do stuff to print ce
        // Note: visitor has visited children already
    }
}
CONSEQUENCES

+ *Visitor makes adding new operations easy* – simply add a new visitor

+ *A visitor gathers related operations and separates unrelated ones*

+ *Visiting across class hierarchies* – visitor does not require visited classes to have common type

+ *Accumulating state* – visitor can accumulate state of traversal which would otherwise need to be threaded as objects

- *Added new ConcreteElement classes is hard* – each visitor would need to be modified

- *Breaking encapsulation* – ConcreteElement interface needs to be powerful enough to let visitors do their jobs, which often require exposing internals.
STRATEGY
STRATEGY

A family of interchangeable algorithms that can varying independently from the client.
Algorithms for breaking a stream of text into lines.

Different algorithms appropriate at different times.

Avoid mingling all such algorithms with main code.

Allow new algorithms to be added.
STRATEGY PATTERN

Context may have an interface to allow Strategy to access its data.
Before Strategy

```java
void repair() {
    switch (_breakingStrategy) {

    case SimpleStrategy:
        composeWithSimpleCompositor();
        break;

    case TexStrategy:
        composeWithTexCompositor();
        break;
    // ..
    }
}
```

After Strategy

```java
void repair() {
    _compositor.compose();
}
```
class Composition {
    private Compositor _compositor; // Strategy instance
    Composition(Compositor compositor) {
        _compositor = compositor;
    }
    ...
    
    void repair() {
        ...
        int breakcount = _compositor.compose(natural,
                                             stretchability,
                                             shrinkability,
                                             componentCount,
                                             lineWidth,
                                             breaks);
        ...
        }
    }
    
    // Context information passed in
    
    // Class parameterised by strategy
    // (could change dynamically)
abstract class Compositor {
    void compose(Coord[] natural, Coord[] stretch, Coord[] shrink,
                 int componentCount, int lineWidth, int[] breaks);
}

class SimpleCompositor extends Compositor {
    void compose(Coord[] natural, Coord[] stretch, Coord[] shrink,
                 int componentCount, int lineWidth, int[] breaks) {
        // do the stuff
    }
}

Instantiating with a particular strategy.
Composition quick = new Composition(new SimpleCompositor());
CONSEQUENCES

+ **Families of related algorithms** – inheritance can help factor out commonality.

+ **Alternative to subclassing** – but allows algorithm to be changed dynamically.

+ **Strategies eliminate conditional statements** – too many conditionals indicates that a Strategy is required.

+ **A choice of implementations** – of same behaviour with different space-time trade-off

- **Clients must be aware of different strategies** – clients need to know of strategies to understand class, which exposes implementation details.

+ **Communication overhead between Strategy and Context** – not all of interface is used by all ConcreteStrategies

- **Increased number of objects**
CREATIONAL PATTERNS

**Builder** – Separate construction process from representation, allowing same construction to build different representations.

**Factory Method** – An method for creating objects, specialised in subclasses.

**Singleton** – A class with only one instance and a global point of access.

**Prototype** – Specify object by creating an instance that will be copied.
STRUCTURAL PATTERNS

**Adaptor** – Convert the interface of a class into another interface clients expect.

**Bridge** – Decouple abstraction and implementation to allow them to vary independently.

**Composite** – represents part-whole hierarchies, treating individual objects and compositions uniformly.

**Facade** – Provide a unified interface to a set of interfaces in a subsystem.

**Flyweight** – uses sharing to support large numbers of fine-grained objects.

**Proxy** – a placeholder for another object to control access to it.
BEHAVIOURAL PATTERNS I

*Chain of Responsibility* – decouple sender of a request from receiver: create chain of objects to handle request.

*Command* – encapsulate request in object (e.g., queue, log requests). Supports undoable actions.

*Interpreter* – a representation of the grammar of a language and interpreter for it.

*Iterator* – access elements of an aggregate without exposing its underlying representation.

*Mediator* – encapsulate the interaction between objects as an object. Objects do not refer to each other explicitly and interaction can vary independently.
BEHAVIOURAL PATTERNS II

*Memento* – Capture object’s internal state without violating encapsulation so that it can be restored later.

*Observer* – Many objects are notified when one object’s internals state changes.

*State* – Allow an object to alter its behaviour when its internal state changes.

*Template Method* – The skeleton of an algorithm in an operation, deferring some steps to subclasses.
USING DESIGN PATTERNS
GUIDELINES

Understand how patterns solve design problems.

Scan intent sections.

Study relationship between patterns.

Study patterns of like purpose.

Examine causes for redesign – often indicates place where pattern could have been applied.

Consider what should be variable in your design – encapsulate what varies.
LEARN
THE GANG OF FOUR
PATTERNS
GOALS REVISITED

Today we learnt about

• what design patterns are

• how to use them