OVERVIEW

What are design patterns

Applying some patterns
DESIGN PATTERNS
DESIGN PATTERNS

Well-designed 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.
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 PROBLEMS
Two kinds of board, but otherwise play is the same. How to specify board construction in a modular way.
ABSTRACT FACTORY, FACTORY METHOD, BUILDER

Abstract Factory – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Factor Method – Define an interface for creating an object, but let subclasses decide which class to instantiate.

Builder – separate the construction of a complex objects from its representation so that the same construction process can create different representations.
ABSTRACT FACTORY

CreateProductA()
CreateProductB()

AbstractFactory

ConcreteFactory1
CreateProductA()
CreateProductB()

ConcreteFactory2
CreateProductA()
CreateProductB()

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();
}
class EnchantedMazeFactory implements MazeFactory {
    Room makeRoom(int n) {
        return new EnchantedRoom(n, castSpell());
    }

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

    private Spell castSpell() {
        ...
    }
}
abstract public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }
    abstract Room makeRoom();
}

public class OrdinaryMazeGame extends MazeGame {
    Room makeRoom() {
        return new OrdinaryRoom();
    }
}

public class MagicMazeGame extends MazeGame {
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
for each element read
  switch element.type
    case PARAGRAPH
      converter.makeParagraph(element)
    case LIST
      converter.makeList(element)
    case TABLE
      converter.makeTable(element)
PROBLEM

Separate creation of objects from the objects they depend on
Dependency Injection – Move dependency of clients on services out of client.

Pass in service as parameter to constructor or via setter.
BEFORE DEPENDENCY INJECTION

class Game {
    Board board;

    Game() { // constructor
        board = new Board(); // class of board fixed
    }

    game = new Game();
}
class Game {
    Board board;

    Game(Board board) { // constructor
        this.board = board;
    }
}

// client chooses class of board
game = new Game(new RandomBoard());
PROBLEM

Connect to Facebook to access list of friends, but need to remain protected from changes in Facebook’s API.
Adaptor – convert the interface of a class into another interface classes expect.

Bridge – decouple an abstraction from its implementation so that the two can vary independently.
ADAPTOR

Client \rightarrow \textit{FriendListI}
\begin{enumerate}
\item getFriends()
\end{enumerate}
\rightarrow \textit{FriendList Impl}
\rightarrow \textit{FacebookAPI}
\begin{enumerate}
\item getNextFriend()
\item noMoreFriends()
\end{enumerate}

\textit{Client} \rightarrow \textit{FriendListI} \rightarrow \textit{FriendList Impl} \rightarrow \textit{FacebookAPI}
BRIDGE

FriendList
getFriends()  

FacebookAPI
getNextFriend()
nomoreFriends()

l = new List()
while (!impl.noMoreFriends) {
    l.add(impl.getNextFriend())
} 
return l

impl
BRIDGE

FriendList
getFriends()

FacebookAPI Interface
getNextFriend()
noMoreFriends()

Refined FriendList

FacebookAPI Impl

AltFacebook APIImpl
DIFFERENCES

- Bridge separates an interface from its implementation.
- Adaptor provides a new interface to an existing implementation.
- Implementation is essentially the same.
PROBLEM

Client holds copy of board – real board resides on server.
Proxy – provide a surrogate or placeholder for another object to control access to it.
PROBLEM

Board construction is expensive – reuse existing ones from finished games.
Object pool uses a set of initialised objects kept ready for use in a pool rather than allocating and destroying them on demand.

```java
public class Board implements PooledObject {
    Board() {
        // do expensive operations to set up board
    }

    void cleanUp() {
        // clear tiles from board
    }
}
```
public static class BoardPool {
    private static List<Board> _available = new List<Board>();
    private static List<Board> _inUse = new List<Board>();

    public static Board getBoard() {
        if (_available.count != 0) {
            Board po = _available[0];
            _inUse.Add(po);
            _available.RemoveAt(0);
            return po;
        } else {
            Board po = new Board();
            _inUse.Add(po);
            return po;
        }
    }

    public static void releaseObject(Board po) {
        po.cleanUp(); _available.Add(po); _inUse.Remove(po);
    }
}
Avoid creating lots of little objects (tiles).
FLYWEIGHT

- **FlyweightFactory**
  - `getFlyweight(flyweightKey : Key) : Flyweight`

- **Flyweight**
  - `doOperation(state : ExtrinsicState) : void`

- **ConcreteFlyweight**
  - `state : IntrinsicState`
  - `doOperation(state : ExtrinsicState) : void`

- **Client**
PROBLEM

How to deal with semantics of empty square without having an if statement.
null object

Code below is part of functionality determining bonuses – empty squares contain null.

```java
if (square[n][m] == null) {
    wordMultiplier = 1;
    letterMultiplier = 1;
} else {
    wordMultiplier = square[n][m].wordMultiplier();
    letterMultiplier = square[n][m].letterMultiplier();
}
```
NULL OBJECT

A better solution is to replace null with an object that delivers the desired functionality.

class EmptySquare extends Square {
    int wordMultiplier() { return 1; }
    int letterMultiplier() { return 1; }
}

Code on previous slide becomes:

wordMultiplier = square[n][m].wordMultiplier();
letterMultiplier = square[n][m].letterMultiplier();
Ensure that there is just one statistics object, potentially accessible from everywhere
**SINGLETON**

**Singleton** – Ensure a class has only one instance, and provide a global point of access to it.

class Statistics {
  static private Statistics theStatistics = new Statistics();

  static public getStatistics() {
    return theStatistics;
  }

  // private constructor
  private Statistics() { ... }
}
Problem

Propagate updates to the board (by other player) to all relevant parties/views
Observer – define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.
**OBSERVER**

```plaintext
Subject
- attach(Observer)
- detach(Observer)
- notify()

Observer
- update()

call update on all observers

Board
- subjectState
- getState()
- updateState()

BoardObserver
- observerState
- update()

observerState = subject.getState()
```
Behaviour of game varies depending on which player’s move it is.
State – Allow an object to alter its behaviour when its internal state changes.

Strategy – Define a family of algorithms, encapsulate each one, and make them interchangeable.

Template Method – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
class PlayController {
    PlayState _state;

    void play() {
        _state.play(...);
    }
    void enterYourTurnState() {
        _state = YourTurnState.instance();
    }
}

class MyTurnState extends PlayState {
    void play(...) {
        // do the play
        controller.enterYourTurnState();
    }
}

class YourTurnState extends PlayState {
    void play(...) {
        // cannot play
    }
}

A new class for each possible state.

State-based functionality is encapsulated into another object.

A singleton
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.
Context may have an interface to allow Strategy to access its data.
BEFORE STRATEGY

class Composition {
    void repair() {
        switch (_breakingStrategy) {
            case SimpleStrategy:
                composeWithSimpleCompositor();
                break;
            case TexStrategy:
                composeWithTexCompositor();
                break;
            // …
        }
    }
}

Strategies handled within case statement.
Restricted to hardcoded strategies
class Composition {
    CompositionStrategy _compositor; // set in constructor
    void repair() {
        _compositor.compose(...);
    }
}

class SimpleStrategy extends CompositionStrategy {
    void compose(Context ctx) {
        // ...
    }
}

class TexStrategy extends CompositionStrategy {
    void compose(Context ctx) {
        // ...
    }
}
class View {
    abstract void doDisplay(Canvas cs);

    // generic algorithm
    void display(Canvas cs) {
        setFocus();
        doDisplay(cs);
        resetFocus();
    }
}

class MyView extends View {
    void doDisplay(Canvas cs) {
        // render view’s contents on cs
    }
}
COMPARISON

Key differences are:

**State** pattern encompasses all possible states and the current state can be changed by either the context or a state object.

**Strategy** is defined outside the object and is preserved throughout the object lifetime.

**Template Method** uses inheritance to vary part of the algorithm, whereas **Strategy** uses delegation.
PROBLEM

make it flexible to permit different game variations, such as varying the rules that are checked
Decorator – Attach additional responsibilities to an object dynamically.

Chain of Responsibility – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
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 view.
DECORATOR

VisualComponent
Draw()

TextView
Draw()

Decorator
Draw()

ScrollDecorator
scrollPosition
Draw()
ScrollTo()

BorderDecorator
borderWidth
Draw()
DrawBorder()

component

component.Draw()

super.Draw()
DrawBorder()
DECORATOR PATTERN

Component
Operation()

ConcreteComponentA
Operation()

Decorator
Operation()

ConcreteDecoratorA
addedState
Operation()

BorderDecorator
Operation()
AddedBehaviour()

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

Abstract base class of all visual components

abstract class Decorator extends VisualComponent {
    private VisualComponent _component;

    Decorator(VisualComponent component) {
        _component = component;
    }

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

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

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

    void drawBorder(int width) {
        ...
    }

    void draw() {
        super.draw();
        drawBorder(_width);
    }
}
CHAIN OF RESPONSIBILITY

- Client
- Request
- Processing element
- Processing element
- Processing element
- Processing element
CHAIN OF RESPONSIBILITY

Handler
- super
- nextHandler
+ handle()

nextHandler.handle();

If (I can handle request;  // handle it  
else  
  super.handle();

HandlerOne

HandlerTwo
PROBLEM

After placing move, player presses clear. Then board and hand return to previous state.
MEMENTO

Memento – Without violating encapsulation, capture and externalise an object’s internal state so that the object can be restored later to this state.
MEMENTO

```java
return new Memento(state);
```

```java
state = m->getState();
```
Traverse through list of ongoing games without being able to modify list, for example, to render to screen
Iterator – Providing a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
ITERATOR

```
return new ListTraversal(this);
```
Supporting adaptable rules language.
COMPOSITE + INTERPRETER

Composite – Compose objects into tree structures to represent part-whole hierarchies.

Interpreter – Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences from the language.

The abstract syntax tree is an instance of the composite pattern.
composite

// Container functionality:
// for each element
elements[i].doThis();
INTERPRETER

Client

Context

TerminalExpression

AbstractExpression

+solve(inout Context)

CompoundExpression

Perform "parent" functionality then delegate to each "child" element "Context" is data structure for holding input and output
PROBLEM

Copying (random) board rather than rebuilding it from scratch.
Prototype – specify the kinds of objects to create using a prototypical instance, and create new objects by copying this instance.

Implementation – essentially give objects a clone method.

Applicability – when classes are specified at run-time (cannot call new)

– when instances have only a few combinations and it is easier to clone and adjust than create afresh.
PROBLEM

Flexibly allow various (future) operations on the board, squares and tiles.
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 operates.
PROBLEM

```
PROBLEM

Node
| TypeCheck() |
| GenerateCode() |
| PrettyPrint() |

VariableRefNode
| TypeCheck() |
| GenerateCode() |
| PrettyPrint() |

AssignmentNode
| TypeCheck() |
| GenerateCode() |
| PrettyPrint() |
```
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)

AssignmentNode
Accept(NodeVisitor v)

v.VisitAssignment(this)

v.VisitVariableRef(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

:VisitConcreteElementA(aConcreteElementA)

:OperationA()

:aConcreteElementB

:VisitConcreteElementB(aConcreteElementB)

:OperationB()

: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 b
    }

    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.visitElementA(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
    }
}
PROBLEM

Separate a request/operation from where the operation is called – that is, allow callbacks
COMMAND

Command – Encapsulates a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.
COMMAND

Client

«interface» CallbackInterface
+execute()

CallbackOne

CallbackTwo

Receiver

+doThis()
+doThat()

targetObject = receiverObject;
targetMethod = methodPointer;

CallbackInterface token = new CallbackTwo(newReceiver(), "doThis");
// the token object is passed to another // object and that object calls
// token.execute();</p>

// use Java reflection or // C++ or Delphi pointer // to member function
targetObject.targetMethod();
MEDIATOR

Mediator – define an object that encapsulates how a set of objects interact.
Facade — 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.
In this lecture and the previous one we learn a little about a lot of design patterns.

We glossed over a lot of details – these can be found in one of many patterns books or online.