Object-Oriented Design Patterns

David Janzen
EECS 816 Object-Oriented Software Development
University of Kansas
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
What are Design Patterns?

• In its simplest form, a pattern is a solution to a recurring problem in a given context

• Patterns are not created, but discovered or identified

• Some patterns will be familiar?
  – If you’ve been designing and programming for long, you’ve probably seen some of the patterns we will discuss
  – If you use Java Foundation Classes (Swing), you have certainly used some design patterns
Design Patterns Definition

• Each pattern is a three-part rule, which expresses a relation between
  – a certain context,
  – a certain system of forces which occurs repeatedly in that context, and
  – a certain software configuration which allows these forces to resolve themselves

A Good Pattern

• *Solves a problem*:
  – Patterns capture solutions, not just abstract principles or strategies.

• *Is a proven concept*:
  – Patterns capture solutions with a track record, not theories or speculation

A Good Pattern

• *The solution isn't obvious:*
  – Many problem-solving techniques (such as software design paradigms or methods) try to derive solutions from first principles. The best patterns *generate* a solution to a problem indirectly—a necessary approach for the most difficult problems of design.

• *It describes a relationship:*
  – Patterns don't just describe modules, but describe deeper system structures and mechanisms.
A Good Pattern

• *The pattern has a significant human component (minimize human intervention).*
  – All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Strategy Pattern

• Intent:
  – Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

• Basic Idea:
  – Make an algorithm an object so you can switch it
  – common pattern to replace case statements
  – client chooses which strategy to use

• Also Known As Policy

• Examples
  – several ways to find line breaks in text (simple, TeX, array)
  – many ways to sort a list, same interface, space/time trade-offs
Strategy Pattern
General Structure

Client

Context
+setConcreteStrategy()

Strategy
+doAlgorithm()

ConcreteStrategyA
+doAlgorithm()

ConcreteStrategyB
+doAlgorithm()

ConcreteStrategyC
+doAlgorithm()
Strategy Pattern: Rock, Paper, Scissors

RPSGame

+ RPSGame(aPlayer : RPSChooser*)
+ play() : void
- toString( : RPS) : string
- winner( : RPS, : RPS) : string
- toRPS( : char) : RPS

<<enum>>

RPS

+ ROCK
+ PAPER
+ SCISSORS

<<abstract>> + getChoice() : RPS

RPSChooser

- compPlayer

RotatingRPSChooser

- counter : int
+ RotatingRPSChooser()
+ getChoice() : RPS

StuckRPSChooser

+ getChoice() : RPS
enum RPS {ROCK, PAPER, SCISSORS };  
class RPSChooser  
{
    public:  
        virtual RPS getChoice() = 0;
    
};
class RotatingRPSChooser : public RPSChooser  
{
    public:  
        RotatingRPSChooser() : counter(0) {}  
        RPS getChoice() { return (++counter % 3 == 0)?ROCK:  
            (counter%3==1)?PAPER:SCISSORS; }  
    
private:  
        int counter;
    
};
class StuckRPSChooser : public RPSChooser  
{
    public:  
        RPS getChoice() { return SCISSORS; }  
    
};
class RPSGame
{
public:
    RPSGame(RPSChooser* aPlayer) { compPlayer = aPlayer; }
...}

void RPSGame::play()
{
    char choice;
    do {
    cout << "Enter your choice (R)ock, (P)aper, (S)cissor...
    cin >> choice;
    if (choice!='Q') {
    RPS compChoice = compPlayer->getChoice();
    ...
    }
int main() { ...
    RPSGame rps(choice=='S'?new RotatingRPSChooser:
    new StuckRPSChooser);
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Why Patterns?

• Code Reuse is good
  – Software developers generally recognize the value of reusing code
    • reduces maintenance
    • reduces defect rate (if reusing good code)
    • reduces development time
  – Code Reuse Pitfalls
    • Reusing code blindly (out of context)
    • Reusing unproven or untested code
Why Patterns?

• Design Reuse is better
  – Similar benefits and pitfalls to Code Reuse
  – Identifying reuse early in the development process can save even more time
  – Copying proven solutions
  – Solutions can be analyzed visually with UML without complexities of the code
Why Patterns?

• Good designs reduce dependencies/coupling
  – Many patterns focus on reducing dependencies/coupling
  – Strongly coupled code
    • hard to reuse
    • changes have wide effects
  – Loosely coupled code
    • objects can be reused
    • changes have isolated effects
Why Patterns?\textsuperscript{1}

• Common vocabulary and language
  – Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them together.
  – Patterns help create a shared language for communicating insight and experience about recurring problems and their solutions.

1. http://hillside.net/patterns
Why Patterns?

• Common vocabulary and language
  – Sunday’s Game Analogy:
    • Call an option left from the line when in the Pro Set-formation and the defense shows blitz.
    • Pattern Name: Option on Blitz
    • Context: Offense in Pro Set-formation, Defense shows Blitz
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Design Patterns History

• Christopher Alexander (architect)
  “A Pattern Language”, 1977
  – quality architectural designs can be
    • described
    • reused
    • objectively agreed to be good
  – structures solve problems
  – multiple structures can solve the same problem
  – similarities in these structures can form a pattern

* see www.greatbuildings.com/architects/Christopher_Alexander.html and http://c2.com/cgi/wiki?ChristopherAlexander
Design Patterns History

• ESPRIT consortium in late 1980’s developed a pattern-based design methodology inspired by Alexander’s work

• OOPSLA’87 Kent Beck and Ward Cunningham introduced idea of identifying patterns in software engineering
  – Well known for work in Smalltalk, CRC Cards, xUnit testing framework, eXtreme Programming, ...
Design Patterns Resources

• “Design Patterns: Elements of Reusable Object-Oriented Software”
  – by Gamma, Helm, Johnson, and Vlissides
  – Gang of Four (GoF)
  – 1994 Software Productivity Award

• “Java Design Patterns” by James Cooper
  – Some examples used in presentation

• http://hillside.net/patterns/patterns.html
• http://patterndigest.com
• http://wiki.cs.uiuc.edu/PatternStories/
Design Patterns

• GoF grouped patterns into three areas:
  – Creational Patterns
    • Abstract Factory, Builder, Factory Method, Prototype, Singleton
  – Structural Patterns
    • Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
  – Behavioral Patterns
    • Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor
Design Patterns

- GoF book preceded commercial release of Java
- Java language designers read GoF
- Hence Java libraries implement many GoF

Design Patterns

- Iterator on Vector or ArrayList
- Observer (interface) and Observable (class)
- WindowAdapter
- Stack

- Similarly, C++ STL implements many patterns
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Creational Patterns

• Better ways to create an instance
• Abstract the instantiation process
  – hide how class instances are created and combined
  – Traditional creation:
    • MyType t = new MyType();
      – client code is now dependent on MyType
      – changing MyType means changing all clients
      – clients must know of different ways to create MyType
        » (e.g. constructor parameters)

• GoF Creational Patterns
  – Factory Method
  – Simple Factory, Abstract Factory
  – Builder, Prototype, Singleton
Factory Method

• Intent:
  – Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

• Also Known As: Virtual Constructor

• Basic Idea:
  – a client creates a product through an abstract class
  – i.e. a concrete subclass knows how to create the product, the client only knows the parent abstract class and the factory method to call to get the product
Factory Method
General Structure

Client

Product
+doSomething()

ConcreteProductB
+doSomething()

ConcreteProductA
+doSomething()

ConcreteCreatorA
+FactoryMethod()

ConcreteCreatorB
+FactoryMethod()

Creator
+AnOperation()
+FactoryMethod()
Factory Method
General Behavior

: Client

1: : ConcreteCreatorB

2: FactoryMethod()

3: : ConcreteProductB

4: return

5: doSomething()
GoF Maze prior to Patterns

```
Maze
- Rooms : Room[]
+ AddRoom() : void
+ RoomNo() : void

MazeGame
+ CreateMaze() : Maze*

<<enumeration>>
Direction
North
South
East
West

MapSite
+ Enter() : void

Room
- roomNumber : int
- sides : MapSite* [4]
+ GetSide( direction : Direction ) : MapSite*
<<<constructor>>>
+ Room( roomNum : int )
+ SetSide( direction : Direction, site : MapSite ) : void

Wall

Door
- isOpen : boolean
<<<constructor>>>
+ Door( room1 : Room*, room2 : Room* )
+ OtherSideFrom( room : Room* ) : Room*
```

Janzen. All rights reserved.
Maze* MazeGame::CreateMaze( )
{
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}

//Client code
int main()
{
    MazeGame* mg;
    mg = new MazeGame();
    Maze* m=mg.CreateMaze();
    //do something with m

    return 0;
}
Maze* MazeGame::CreateMaze( )
{
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);
    return aMaze;
}
Problems with GoF Maze

Maze* MazeGame::CreateMaze(char mazeType)
{
    Maze* aMaze = new Maze;
    switch(mazeType) {
        case 'N' : Room* r1 = new Room(1);
                   Room* r2 = new Room(2); break;
        case 'B' : Room* r1 = new RoomWithABomb(1);
                   Room* r2 = new RoomWithABomb(2); break;
        case 'E' :
                   Room* r1 = new EnchantedRoom(1,CastSpell());
                   Room* r2 = new EnchantedRoom(2,CastSpell()); break;
    }
    if (mazeType == 'E')
        Door* theDoor = new DoorNeedingSpell(r1, r2);
    else
        Door* theDoor = new Door(r1, r2);
    ...

Bad Approach: uses multiple if/then/else.
GoF Maze with Factory Method

Maze
- Rooms : Room[]
+ AddRoom() : void
+ RoomNo() : void

MazeGame
+ CreateMaze() : Maze*
+ MakeDoor() : Door*
+ MakeMaze() : Maze*
+ MakeRoom() : Room*
+ MakeWall() : Wall*

MapSite
+ Enter() : void

Room
- roomNumber : int
- sides : MapSite* [4]
+ GetSide( direction : Direction ) : MapSite*
<<constructor>>+ Room( roomNum : int )
+ SetSide( direction : Direction, site : MapSite ) : void

Wall
- isOpen : boolean

<<constructor>>+ Door( room1 : Room*, room2 : Room* )
+ OtherSideFrom( room : Room* ) : Room*

Door

Factory Methods
GoF Maze with Factory Methods

Maze* MazeGame::CreateMaze( )
{
    Maze* aMaze = MakeMaze();
    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());
    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);
    return aMaze;
}

Client calls Factory Methods.

virtual Maze* MakeMaze()
{
    return new Maze;
}

virtual Wall* MakeWall()
{
    return new Wall;
}

Now we can inherit from MazeGame and specialize parts of the maze.
Extending Factory Method GoF Maze

---

```
Maze
- Rooms : Room[*]
+ AddRoom() : void
+ RoomNo() : void

Door
+ Constructor
  + Door(room1 : Room*, room2 : Room* )
  + OtherSideFrom(room : Room* ) : Room*

- isOpen : boolean
+ SetSide(direction : Direction, site : MapSite) : void
+ GetSide(direction : Direction) : MapSite[*]

Room
- roomNumber : int
- sides : MapSite[*] [4]
+ GetSide(direction : Direction) : MapSite*
+ Constructor
  + Room(roomNum : int )
+ SetSide(direction : Direction, site : MapSite) : void

Maze
+ CreateMaze() : Maze*
+ MakeDoor() : Door*
+ MakeMaze() : Maze*
+ MakeRoom() : Room*
+ MakeWall() : Wall*

EnchantedMazeGame
# CastSpell() : Spell*
+ MakeDoor(room1 : Room*, room2 : Room* ) : Door*
+ MakeRoom(n : int) : Room*

BombedMazeGame
+ MakeRoom() : Room*
+ MakeWall() : Wall*

BombedWall

RoomWithABomb

EnchantedRoom

DoorNeedingSpell

MapSite
+ Enter() : void

Wall
- isOpen : boolean
++ Constructor
+ Door(room1 : Room*, room2 : Room* )
+ OtherSideFrom(room : Room*) : Room*

Extending Factory Method GoF Maze
```
Extending Factory Method GoF Maze

class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();
    virtual Wall* MakeWall() const
    {
        return new BombedWall;
    }
    virtual Room* MakeRoom(int n) const
    {
        return new RoomWithABomb(n);
    }
};

//Client code
int main()
{
    MazeGame* mg;
    mg = new BombedMazeGame();
    Maze* m=mg.CreateMaze();
    //do something with m
}

Only one line changes in client.
Extending Factory Method GoF Maze

class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();
    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); } 
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1,r2); } 
protected:
    Spell* CastSpell() const;
};

//Client code
int main()
{
    MazeGame* mg;
    mg = new EnchantedMazeGame();
    Maze* m=mg.CreateMaze();
    //do something with m
}

Only one line changes in client.
(Simple) Factory Pattern
General Structure

Product
  +doSomething()

ProductA
  +doSomething()

ProductB
  +doSomething()

ConcreteFactory
  +getProduct()

Client
  +doSomething()
(Simple) Factory Pattern
General Behavior

: Client

: ConcreteFactory

: ProductB

1: 

2: getProduct()

3: 

4: return

5: doSomething()
Abstract Factory Pattern

• Intent:
  – Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

• Also Known As: Kit

• Basic Idea:
  – a client object refers to a family of related objects using only abstract classes
Abstract Factory Vehicle Rentals

• Suppose we have a system for making rental car reservations

![UML Diagram]

WebClient
+ MakeReservation()

Reservation
+ Reservation()

PrimaryDriver

Vehicle Inventory
+ getAvailVehicle(numPassengers : Integer)

Vehicle
Abstract Factory Hotel Rentals

• Now suppose we want to make reservations for not only rental cars, but also hotels

– Looks familiar, huh?
Abstract Factory Pattern

- We want to create families of reservations
- WebClient can be independent of the kind of reservation
Abstract Factory Pattern Exercise

• We can even add a ReservationFactoryChooser which is a Factory Method that pulls the Factory choice out of WebClient

```
Customer
+ Customer(name : String, numGuests : Integer)
+ getName() : String
+ getNumCustomers() : Integer

PrimaryGuest

PrimaryDriver

Unit
+ Unit(id : Integer, capacity : Integer)
+ getUnitNum() : Integer
+ getCapacity() : Integer

UnitInventory
+ getAvailUnit() : Unit

Room

Vehicle

Property

VehicleInventory

Reservation_AbstractFactory
+ createProductUnit() : Unit
+ createProductCustomer() : Customer
+ createProductUnitInventory() : UnitInventory

RoomReservationFactory

VehicleReservationFactory

ReservationFactoryChooser
+ getReservationFactory() : Reservation_AbstractFactory

WebClient
+ MakeReservation()

Copyright © 2006 by David S. Janzen. All rights reserved.
Abstract Factory Pattern
General Structure

AbstractFactory
+CreateProductA()
+CreateProductB()

ConcreteFactory1
+CreateProductA()
+CreateProductB()

ConcreteFactory2
+CreateProductA()
+CreateProductB()

AbstractProductB

ConcreteProductB1

ConcreteProductB2

AbstractProductA

ConcreteProductA1

ConcreteProductA2

Client
Abstract Factory Pattern
General Behavior

: Client

1: : ConcreteFactory2

2: CreateProductA()

3: : ConcreteProductA2

4: return

5: doSomething()

6: CreateProductB()

7:

8: return

: ConcreteProductB2

Copyright © 2006 by David S. Janzen. All rights reserved.
Factory Pattern Differences

• Simple Factory
  – client asks a concrete factory class for a concrete object that will do what client wants
  – factory only returns one product

• When to use
  – when factory only returns one product
  - one product may have multiple variations all with perhaps some common functionality
Factory Pattern Differences

• Factory Method
  – client instantiates a concrete factory class (child of a common abstract class) and invokes a method on it that selects a concrete product class

• When to use
  – when client can’t anticipate the class of objects it must create
  – factory wants concrete factories to decide what concrete product to use
Factory Pattern Differences

• Abstract Factory
  – client creates a concrete factory class that can produce several concrete products
  – client only references abstract factory and abstract products

• When to use
  – when client needs a family of products, but wants to be independent of them
Singleton Pattern

• Intent:
  – Ensure a class only has one instance, and provide a global point of access to it

• Singleton should be considered only if all three of the following criteria are satisfied:
  – Ownership of the single instance cannot be reasonably assigned to some other class
  – Lazy initialization is desirable
  – Global access is not otherwise provided for
Singleton Implementation Options

• Many technical issues dealing with inheritance, multithreading and destruction

• Java
  – final class with static methods
  – class with a static variable, static method, and private constructor
  – class with a static variable and public constructor that throws an exception on subsequent calls

• C++
  – Static method, non-public constructor
  – Non-public copy constructor
  – Return reference instead of pointer so can’t be deleted
public class PrintSpooler {
    // a prototype for a spooler class
    // such that only one instance can ever exist
    private static PrintSpooler spooler;
    private PrintSpooler() {  // privatized
    }
    public static synchronized PrintSpooler getSpooler() {
        if (spooler == null)
            spooler = new PrintSpooler();
        return spooler;  // always returns the spooler
    }

    public void print(String s) {
        System.out.println(s);
    }
}

Singleton Pattern
Example: Print Spooler
Singleton Pattern

General Structure: static method, static member, private constructor

PrintSpooler

+getSpooler() : PrintSpooler
+print(s : String) : void
-PrintSpooler()
public class Spooler {
    // this is a prototype for a printer-spooler class
    // such that only one instance can ever exist
    static boolean instance_flag = false;  // true if
                                            // one instance

    public Spooler() throws SingletonException
    {
        if (instance_flag)
            throw new SingletonException("Only one printer allowed");
        else
            instance_flag=true;     // set flag for one instance
        System.out.println("printer opened");
    }
}
Singleton Pattern
General Structure: static member, public constructor, throws exception

<table>
<thead>
<tr>
<th>SingletonException</th>
<th>Spooler</th>
</tr>
</thead>
<tbody>
<tr>
<td>+SingletonException()</td>
<td>~instance_flag : boolean = false</td>
</tr>
<tr>
<td>+SingletonException(s : String)</td>
<td>+Spooler()</td>
</tr>
</tbody>
</table>
Builder Pattern

• Intent:
  – Separate the construction of a complex object from its representation so that the same construction process can create different representations

• Basic Idea:
  – Create a separate class that constructs each of several complex objects. That way the construction process is separate from its representation and it can produce different representations.
Builder Pattern

Client

Director

Builder

ConcreteBuilder

Product

Client

Director

Builder

ConcreteBuilder

Product
Builder Pattern

: Client

1: : ConcreteBuilder

2: : Director

3: BuildPart()

4: BuildPart()

5: GetProduct()
Builder Pattern: Benefits

• Isolates code for construction and representation
  – ConcreteBuilder’s can be reused by multiple Directors

• Lets you vary a product’s internal representation
  – Construction process is hidden from clients

• Gives you finer control over the construction process
  – Good for multi-step construction processes
Builder Pattern:
When to Use

• When the algorithm for creating a complex object should be independent of the parts that make up the object and how they’re assembled

• When the construction process must allow different representations for the object that is constructed
Prototype Pattern

• Intent:
  – Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

• Example:
  – Music score editor clones typical objects such as staff, note, rest, …

• Similar benefits to Abstract Factory and Builder
  – Hides concrete product classes from client

• Basic Idea:
  – Create an object that you will copy (clone) and modify as needed
Prototype Pattern

Client

Prototype

+Clone()

ConcretePrototypeA

+Clone()

ConcretePrototypeB

+Clone()
Prototype Pattern: Benefits

- Alternative to Abstract Factory and Builder
- Can allow products to be added and removed at run-time
- May reduce subclassing
  - Don’t need all the creator classes needed in Factory Method
- May be able to create new objects
  - by varying the values given to the prototype
  - Or by varying the objects composing the prototype
Prototype Pattern: When to Use

• When you want to decouple a client from the products that it creates
• When the classes to instantiate are specified at run-time
• When instances of a class can have one of only a few different combinations of state
A Creational Patterns Metaphor
Memory Aid

• Simple Factory:  Trabant
  – East German car with a 7-year wait, one model depending on year of delivery, factory determines which model is given depending on which year your Trabant is delivered

• Factory Method:  Honda
  – Client asks an AccordFactory or a CivicFactory to make an Accord or a Civic
A Creational Patterns Metaphor Memory Aid

• Abstract Factory: General Motors
  – client asks for a PontiacFactory to create a GrandPrix and a TransAm
  – client asks for a GMCFactory to create a half-ton pickup and a Yukon

• Builder: Kit Car
  – client gives director a Kit Car kit and asks it to create a kit car for it
A Creational Patterns Metaphor
Memory Aid

• Singleton:  Uncle Randy’s Dune Buggy
  – create a one-of-a-kind vehicle from scratch
• Prototype:  Copy my favorite car
  – make a copy of my favorite car so I can drive the
    original until it is worn out, and still have another
    one to enjoy when I’m done with the first
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Structural Patterns

• Ways to combine classes and objects into larger structures

• Structural *Class* Patterns use inheritance to compose interfaces or implementations
  – Multiple Inheritance, Adapter

• Structural *Object* Patterns compose objects to realize new functionality
  – Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
Adapter Pattern

• Intent:
  – Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

• Basic Idea:
  – Create a new class that provides a new interface for an existing class.

• Two types:
  – class adapter uses inheritance
  – object adapter uses composition

• Also known as Wrapper

• Silly Example:
  – Coffee server responds to regular or decaf
  – Client wants to say regular or unleaded
Adapter Pattern Example

- Suppose you have written a large application that uses a queue and calls enqueue and dequeue to insert and remove items from the queue.

```cpp
class Queue {
public:
    Queue() : length(0) {} 
    void enqueue(Item i) { q[length++] = i; }
    Item dequeue() { Item tmp = q[0]; 
        for(int i=0;i<length;i++)
            q[i] = q[i+1];
        length--;
        return tmp;
    }
private:
    Item q[100];
    int length;
};
```
int main() {
    Queue requests;
    Item request;
    char choice;
    do
    {
        cout << "Do you want to (E)nter or (R)etrieve a request?" << endl;
        cin >> choice;
        if (choice == 'E')
        {
            cout << "Enter your one character request" << endl;
            cin >> request;
            requests.enqueue(request);
        }
        else if (choice == 'R')
        {
            cout << "The next request is " << requests.dequeue() << endl;
        }
    } while (choice != 'Q');
    //many more calls to enqueue and dequeue...
}
Adapter Pattern Example

• Now you discover that the STL contains a queue class that is more efficient than yours
  – The only problem is that the STL queue uses push and pop instead of enqueue and dequeue
  – What are your choices?
    • Replace all calls to enqueue and dequeue with push and pop respectively
      – No, you have A LOT of code to change
    • Define an adapter that makes the STL queue look like yours
The Adapter Pattern Example

**Object Adapter Approach**

```cpp
class qAdapter{
public:
    qAdapter() {}  
    void enqueue(Item i) { q.push(i); }  
    Item dequeue() {
        Item tmp = q.front();
        q.pop();
        return tmp;  
    }
private:
    queue<Item> q;
};
```
### Adapter Pattern Example

**Class Adapter Approach**

```cpp
class qAdapter : public queue<Item>
{
    public:
        qAdapter() {}
        void enqueue(Item i) { push(i); } 
        Item dequeue() { Item tmp = front(); pop(); return tmp; }
};
```
Adapter Pattern
General Structure for Class Adapter

Client

Target
+Request()

Adaptee
+SpecificRequest()

Adapter
+Request()
Adapter Pattern

- **Target:**
  - defines the interface desired by the client

- **Client:**
  - application class that desires a particular interface

- **Adaptee:**
  - defines an existing interface that needs adapting

- **Adapter:**
  - adapts the interface of Adaptee to the Target interface
Adapter Pattern
General Structure for Object Adapter

Client

Target

Adapter

Adaptee

+SpecificRequest()

+Request()

+Request()
Adapter Pattern

• C++ STL Adapters
  – Container Adapters
    • stack adapts a vector, list, or deque by adding push, pop, top
    • queue adapts a list or deque by adding push, pop, front, back
    • priority_queue adapts a vector by adding push, pop, and top

```cpp
stack<string, vector<string> > str_stack;
str_stack.push(“This is a string”);
while (!str_stack.empty()) {
    cout << str_stack.top();
    str_stack.pop();
}
```

– Iterator Adapters
  • Reverse iterators
    – Begin returns the end of a regular iterator
Adapter Pattern

• Object vs. Class Adapter Approach
  – i.e. when to use inheritance vs. composition adapters
  – Object Adapter
    • good to use when little or no overlap between adapter and adaptee
    • one adapter can work with many adaptees (adaptee or any of its subclasses)
    • adapter can easily add functionality to all adaptees at once
  – Class Adapter
    • good to use when adaptee already supplies much of the desired interface, but not all (inherit some, override some, add some methods)
    • Two-way adapters
      – Class adapter allows the client to use the interface of either the Adapter or the Adaptee
Adapter Pattern Exercise

- C++ STL includes a double-ended queue (deque) that allows you to push and pop on both ends
- Create a Deque in Java by adapting an ArrayList
  - add methods
    - push_front(Object)
    - push_back(Object)
    - pop_front()
    - pop_back()
Decorator Pattern

• Intent:
  – Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

• In my words:
  – Dynamically modify the behavior of individual objects without having to create a new derived class.
  – Multiple decorators can be added to the same object

• Also known as Wrapper
Decorator Pattern
Example: String Formatting

• Suppose we want a tool that will format strings in a variety of ways
  – Convert first chars of sentences to upper case
  – Convert all chars to lower case
  – Convert proper I’s such as in “for i will go” to upper case

• We want to choose different combinations of formatters
Decorator Pattern
Example: String Formatting

//no changes
Orig:   this is a string.  i wonder if i want to change it?
New:   this is a string.  i wonder if i want to change it?

//make beginning of sentences upper case
Orig:   this is a string.  i wonder if i want to change it?
New:   This is a string.  I wonder if i want to change it?

//make all lower case
Orig:   THIS is a STRING.  i WONDER if i want to change it?
New:   this is a string.  i wonder if i want to change it?

//make all lower case and first char of sentences upper case
Orig:   THIS is a STRING.  i WONDER if i want to change it?
New:   This is a string.  I wonder if i want to change it?

//do above and make proper i upper case
Orig:   THIS is a STRING.  i WONDER if i want to change it?
New:   This is a string.  I wonder if I want to change it?
Decorator Pattern
Example: String Formatting

• Bad Approach:
  – Make each combination a new class

```
StringFormatter
+ getString()
```

```
  CapFirstSF  LowerAllSF  CapSF
  |__________|__________|__________|
  |__________|__________|__________|
  |__________|__________|__________|
  |__________|__________|__________|
CapFirstLowerAllSF  CapFirstCapISF  LowerAllCapISF  CapFirstLowerAllCapISF
```
Decorator Pattern
Example: String Formatting

• Better Approach:
  – Make each approach a decorator that can be dynamically combined by the client

```java
StringDecorator
+ StringDecorator()
+ StringDecorator(c : StringDecorator*)
<<virtual>> + getString(s : string) : string
```

```java
CapFirstSD
+ CapFirstSD()
+ CapFirstSD(c : StringDecorator*)
<<virtual>> + getString(s : string) : string
```

```java
LowerAllSD
+ LowerAllSD()
+ LowerAllSD(c : StringDecorator*)
<<virtual>> + getString(s : string) : string
```

```java
CapISD
+ CapISD()
+ CapISD(c : StringDecorator*)
<<virtual>> + getString(s : string) : string
```
Decorator Pattern
Example: String Formatting

class StringDecorator {
public:
    StringDecorator() : comp(0) {}  
    StringDecorator(StringDecorator* c) { comp = c; }
    virtual string getString(string s) {
        if (comp == 0) {  
            return s;
        } else {
            return comp->getString(s);
        }
    }
protected:
    StringDecorator* comp;
};
Decorator Pattern
Example: String Formatting

class CapFirstSD : public StringDecorator {
public:
    CapFirstSD() {}
    CapFirstSD(StringDecorator* c) : StringDecorator(c) {}
virtual string getString(string s) {
        s = comp->getString(s);
        bool punctFound = false;
        s[0] = toupper(s[0]);
        for (int i=0; i < s.length(); i++) {
            if (punctFound && (s[i] != ' ')) {
                s[i] = toupper(s[i]);
                punctFound = false;
            } else if (s[i] == '.')  punctFound = true;
        }
        if (s[i] == '.')  punctFound = true;
        return s;
    }
};
StringDecorator* sd;
string s;
sd = new CapFirstSD(new StringDecorator);
s = "this is a string. i wonder if i want to change it?";
cout << "Orig: " << s << endl;
cout << "   New: " << sd->getString(s) << endl;</string>
delete sd;

sd = new CapFirstSD(new LowerAllSD(new StringDecorator));
s = "THIS is a STRING. i WONDER if i want to change it?";
cout << "Orig: " << s << endl;
cout << "   New: " << sd->getString(s) << endl;</string>
Decorator Pattern
General Structure

\begin{diagram}
\textbf{Component} & \text{method}\()
\end{diagram}

\begin{diagram}
\textbf{ConcreteComponent} & \text{method}\()
\textbf{Decorator} & \text{method}\()
\textbf{ConcreteDecoratorA} & \text{anotherMethod}\(); \text{method}\()
\textbf{ConcreteDecoratorB} & \text{anotherMethod}\(); \text{method}\()
\end{diagram}

\textit{Decorator::method(); anotherMethod(); component.method();}
Decorator Pattern

• Java I/O example
  – In java.io, FilterInputStream is a Decorator of an InputStream
  – Children of FilterInputStream such as BufferedInputStream and PushbackInputStream can be nested (applied repeatedly to an InputStream)

```java
PushbackInputStream in = new PushbackInputStream(
    new InputStreamReader(System.in));
int i;
i = in.read();
in.unread(i);
```
Decorator and Adapter Patterns

- Note the similarities/differences between Decorator and Adapter
  - Both add functionality to an object
    - Adapter to a class
    - Decorator to a single object
  - Adapter changes the interface of a class
  - Decorator adds responsibilities to an object but leaves its interface intact
Proxy Pattern

• Intent:
  – Provide a surrogate or placeholder for another object to control access to it.

• In my own words:
  – Create an object that represents another object

• Also known as Surrogate

• Non-Software Examples:
  – A government ambassador
  – Email forwarding
    • reservations@thecampuscottage.com forwards to david@simexusa.com
    • Enables changes to actual recipient without having to notify all potential customers
Proxy Pattern

• Some types of Proxies
  – virtual proxy
    • standing in for an object that is slow to load, or
    • delaying instantiation until an object is actually needed
  – remote proxy
    • standing in for a remote object
  – protection proxy
    • validating access rights for users
  – smart references
    • smart pointers do reference counting and enable garbage collection
    • load a persistent object when first referenced
    • lock an object before allowing changes
  – copy-on-write
Proxy Pattern

- Virtual Proxy Example:
  - Application needs to load a large image from disk
  - An image proxy is created that draws a rectangle with the image’s size while loading the actual image
Proxy Pattern
Example: ProxyDisplay

**JxFrame**

- JxFrame( title : String )
- setCloseClick() : void
- setLF() : void

**ProxyDisplay**

- main( argv : String[] ) : void
- ProxyDisplay()

**ImageProxy**

- height : int
- width : int
- getPreferredSize() : Dimension
- ImageProxy( filename : String, w : int, h : int )
- paint( g : Graphics ) : void
- run() : void

**Image**

Copyright © 2006 by David S. Janzen. All rights reserved.
Proxy Pattern
Example: ProxyDisplay

: ProxyDisplay

1: : ImageProxy

2: : Image

3: : MediaTracker

4: addImage()

5: : Thread

6: run()

7: checkID()

8: true := return

9: paint( g )
Proxy Pattern
General Structure

Client

Subject
+Request()

RealSubject
+Request()

Proxy
+Request()
Proxy Pattern

• Remote Proxy Example:
  – An application needs to communicate with an object that may be local or may be remote
  – A proxy is created that imitates the desired object, but transports communication over the network to the actual object
Proxy Pattern
Example: NetGame

Game
+play()

PlayerInterface
+getMove()

PlayerProxy
-socket
+getMove()
+receive()
+send()

LocalPlayer
+getMove()

PlayerServer
-socket
+receive()
+send()

Game
+play()
Proxy Pattern

Example: NetGame

1: : LocalPlayer
2: : PlayerProxy
3: receive()
4: play()
5: getMove()
6: return
7: getMove()
8: send()
9: receive()
10: getMove()
11: return
12: send()
13: return
Proxy Pattern

- Smart references in C++:
  - By overloading ->, *, and = you can create a proxy to the real subject that provides additional value such as:
    - Ownership management
      - Reference counting for automatic garbage collection
      - Destroy original when a copy is made?
    - Loading the real subject into memory when first referenced
    - Locking the real subject before using it
  - see “Modern C++ Design: Generic Programming and Design Patterns Applied” by Andrei Alexandrescu for more information on smart pointers
//see Labs/Proxy/smartReferences.cpp

```cpp
template <class T>
class SmartPtr
{
public:
    explicit SmartPtr(T* realSubject) : subject(realSubject) {}
    SmartPtr(SmartPtr& src);
    SmartPtr& operator=(const SmartPtr& rhs);
    ~SmartPtr();
    T& operator*() const
    {
        ...
        return *subject;
    }
    T* operator->() const
    {
        ...
        return subject;
    }
private:
    T* subject;
};
```

**Proxy Pattern**

Add a reference

Remove a reference

Lazy initialization, lock subject, copy-on-write?
Façade Pattern

• Intent:
  – Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

• In my own words:
  – Create a new class that provides a simple interface for a complex subsystem

• Example:
  – A compiler is a simple interface to a complex system
Façade Pattern

General Structure without a Façade
Façade Pattern
General Structure with a Façade
Façade Pattern

• Façade:
  – knows which subsystem classes are responsible for a request
  – Delegates client requests to appropriate subsystem objects

• Subsystem classes:
  – Implement subsystem functionality
  – Handle work assigned by the Façade object
  – Have no knowledge of the façade; that is, they keep no references to it
Façade Pattern

• When to use:
  – When you have a complex system that you can provide a simple interface to
    • Clients needing all the complexities can bypass the façade
  – When you want to decouple the client from the many subsystem classes
    • Changes in the subsystem may only require changes in the façade, not all the clients
  – When you want to layer your subsystems
    • Façades serve as entry points to each layer

• Implementation ideas:
  – Subsystem classes could be placed in namespaces to “hide” them from clients
  – Façades are often Singletons as we only need one of them
Outline

• Introduction
  – Design Patterns Overview
  – Strategy as an Early Example
  – Motivation for Creating and Using Design Patterns
  – History of Design Patterns

• Gang of Four (GoF) Patterns
  – Creational Patterns
  – Structural Patterns
  – Behavioral Patterns
Behavioral Patterns

• Ways to deal with algorithms and have objects communicate

• Behavioral *Class* Patterns use inheritance to distribute behavior between classes
  – Template Method, Interpreter

• Behavioral *Object* Patterns compose objects to distribute behavior
  – Mediator, Chain of Responsibility, Observer, Strategy, Command, State, Visitor, Iterator, Memento
Chain of Responsibility Pattern

• Intent:
  – Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

• Basic Idea:
  – Decouple a request sender from its receiver by allowing multiple objects the chance to handle a request. Each object only knows the next object in chain

• Game:
  – variation on hot potato
  – always pass objects in same order
  – each person knows what kind of object to keep
    • e.g. candyeater eats candy, jackspinner, ballbouncer,…
Chain of Responsibility Pattern
General Structure

- **Chain**
  - +addChain()
  - +getChain()
  - +sendToChain()

- **Sender**
  - +addChain()
  - +getChain()
  - +sendToChain()

- **ConcreteChainA**
  - +addChain()
  - +getChain()
  - +sendToChain()

- **ConcreteChainB**
  - +addChain()
  - +getChain()
  - +sendToChain()
Chain of Responsibility Pattern
Chain of Responsibility Pattern

• Example
  – JTutorial\ChainOfResp\ImageChain\java Chainer
    • enter “Mandrill” and Mandrill.jpg will load
    • enter “blue” and box will be blue
    • enter “FileList” and FileList.class will be highlighted
public interface Chain
{
public abstract void addChain(Chain c);
    //make this class “point” to c
public abstract void sendToChain(String msg);
    //send request to next object in chain
public Chain getChain();
    //return the next object in the chain
}
public class Imager extends JPanel implements Chain {
    private Chain nextChain;
    //------------------------------------------
    public void addChain(Chain c) {
        nextChain = c; //next in chain of resp
    }
    //------------------------------------------
    public void sendToChain(String mesg) {
        //if there is a JPEG file with this root name
        //load it and display it.
        if (findImage(mesg))
            loadImage(mesg + "\.jpg");
        else
            //Otherwise, pass request along chain
            nextChain.sendToChain(mesg);
    }
    //------------------------------------------
    public Chain getChain() {
        return nextChain;
    }
}
public class Chainer extends JxFrame
{
  //list of chain members
  Sender sender;       //gets commands
  Imager imager;       //displays images
  FileList fileList;   //highlights file names
  ColorImage colorImage;  //shows colors
  RestList restList;   //shows rest of list

  public Chainer() {
    sender = new Sender();
    imager = new Imager();    //add all these to the Frame
    fileList = new FileList();
    colorImage = new ColorImage();
    restList = new RestList();

    //set up the chain of responsibility
    sender.addChain(imager);
    imager.addChain(colorImage);
    colorImage.addChain(fileList);
    fileList.addChain(restList);
  }
}

Chain of Responsibility Pattern
public class Sender extends JPanel implements Chain, ActionListener {
    private Chain nextChain;
    private JTextField tx;
    private JButton Send;

    public Sender() {
        ... //set up GUI
        Send = new JButton("Send");
        Send.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        String file = tx.getText();
        if ((file.length() > 0) && (nextChain != null))
            nextChain.sendToChain(file);
    }
}

Chain of Responsibility Pattern
Chain of Responsibility Pattern
Chain of Responsibility Pattern

dynamically add in the middle of the chain: ConcreteChainA

1: addChain()
2: addChain()
3: addChain()
4: addChain()
5: addChain()
6: addChain()
7: addChain()
8: getChain()
9: addChain()
10: addChain()
Chain of Responsibility Pattern

• When to use:
  – When more than one object may handle a request and we don’t know the handler beforehand
  – When we want to decouple a request sender from its request receivers
  – When we want to dynamically change the objects that might handle a request

• Consequences
  – A request might not get handled
  – Reduced coupling; neither sender nor receiver know of the other
Chain of Responsibility Pattern
Exercise

• Rearrange the order of the chain in the Imager application so that if a filename has the same name as a valid color, the file is handled, rather than setting the color
Mediator Pattern

• **Intent:**
  – Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

• **In my own words:**
  – reduce dependencies between objects that have a lot of interaction by making the interaction a separate object
  – the mediator object contains the business rules
  – the mediator object is probably not reusable, but the other objects are

• **Non-software example**
  – A control tower at an airport is a Mediator
Mediator Pattern

Example: Robot with one motor and a front and back touch sensor
Mediator Pattern

- Each Widget must know about the mediator:

```cpp
class Widget {
public:
    Widget(Mediator*);
    virtual void Changed();
    virtual void HandleEvent(Event& event);
    Widget();
private:
    Mediator* _mediator;
};
Widget::Widget(Mediator* m) {
    _mediator = m;
}
void Widget::Changed () {
    cout << "The widget is changed." << endl;
    _mediator->WidgetChanged(this);
}
```
class Mediator {
public:
  virtual ~Mediator();
  virtual void ShowDialog();
  virtual void WidgetChanged(Widget*) = 0;
protected:
  Mediator();
  virtual void CreateWidgets() = 0;
};
class BumpMediator : public Mediator{
  //...
  void BumpMediator::WidgetChanged(Widget* theChangedWidget) {
    if (theChangedWidget == _front) {
      cout << "front" << endl;
      _motor->SetDirection("backward");
    } else if (theChangedWidget == _back) {
      cout << "back" << endl;
      _motor->SetDirection("forward");
    }
  }
};
Mediator Pattern
Another Example: No Mediator

- KTextField
  +KTextField( md : Mediator )

- ClearButton
  +ClearButton( act : ActionListener, md : Mediator )
  +Execute() : void

- CopyButton
  +CopyButton( fr : ActionListener, md : Mediator )
  +Execute() : void

- KidList
  -fillKidList() : void
  +KidList( md : Mediator )
  +valueChanged( ls : ListSelectionEvent ) : void

- PickedKidsList
  +PickedKidsList( md : Mediator )
Mediator Pattern
Another Example: With Mediator

CopyButton
+copyButton( fr : ActionListener, md : Mediator )
+Execute() : void

ClearButton
+ClearButton( act : ActionListener, md : Mediator )
+Execute() : void

CopyButton
+ClearButton( fr : ActionListener, md : Mediator )
+Execute() : void

KidList
+KidList( md : Mediator )
+valueChanged( ls : ListSelectionEvent ) : void

Mediator
+Clear() : void
+init() : void
+Move() : void
+registerClear( cb : ClearButton ) : void
+registerCopy( mv : CopyButton ) : void
+registerKidList( kl : KidList ) : void
+registerPicked( pl : PickedKidsList ) : void
+registerText( tx : KTextField ) : void
+select() : void

KTextField
+KTextField( md : Mediator )

PickedKidsList
+PickedKidsList( md : Mediator )

KidList
+KidList( md : Mediator )
+valueChanged( ls : ListSelectionEvent ) : void
Mediator Pattern
General Structure
Mediator Pattern
General Structure

Client

1: :ConcreteMediator

2: :ConcreteColleagueA

3: register()

4: :ConcreteColleagueB

5: register()

6:

7:

8:
Web Application Patterns

• Action Controller
  – Each web page has a controller object that knows which model and view to use

• Front Controller
  – A single object handles incoming requests and passes them to other objects
Web Application Patterns

• Data Mapper
  – Each Domain Object has a Mapper object that knows how to store and retrieve it from the DB

• See more at:
  – http://martinfowler.com/isa/

• For a framework that contains many patterns:
  – See Expresso at http://www.jcorporate.com/
Summary

• Ask what changes?
  – if algorithm changes: use Strategy
  – if only parts of an algorithm change: use Template Method
  – if interaction changes: use Mediator
  – if subsystem changes: use Façade
  – if interface changes: use Adapter
  – if number and type of dependents changes: use Observer
  – if state causes behavior changes: use State
  – if object to be instantiated changes: use Factory
  – if the action of an object is requested to perform changes: use Command