FDA149
Software Engineering

Introduction to Design Patterns

Peter Bunus
Dept of Computer and Information Science
Linköping University, Sweden
petbu@ida.liu.se
Top 10 Reasons to take a Design Pattern Course

1. Amy Diamond took this course but she is still wondering "What’s in it for me?". Maybe I will get it and explain it to her.
2. I could get some easy points.
3. Everybody is talking about it so it must be cool.
4. If I master this I can add it to my CV.
5. Increase my salary at the company.
6. Applying patterns is easier than thinking.
7. A great place to pick up ideas to plagiarize.
8. I bought this lousy T-Shirt and I would like to understand the joke.
9. I thought that course is about Dating Design Patterns.
10. I failed the course last year so I’m trying again.
Seven Layers of Architecture

- Enterprise-Architecture
- Global-Architecture
- System-Architecture
- OO Architecture
- Application-Architecture
- Subsystem
- Macro-Architecture
- Frameworks
- Micro-Architecture
- Design-Patterns
- Objects
- OO Programming
A Brief History of Design Patterns


  3-D computer modeling
  visual simulations
  computer aided design (CAD)
  virtual reality
  OO Programming

- 1970... - the window and desktop metaphors (conceptual patterns) are discovered by the Smalltalk group in Xerox Parc, Palo Alto
A Brief History of Design Patterns

- 1978/79: Goldberg and Reenskaug develop the MVC pattern for user Smalltalk interfaces at Xerox Parc

- 1979 Cristopher Alexander publishes: ”The Timeless Way of Buildings”

Introduces the notion of pattern and a pattern language

It is a architecture book and not a software book

Alexander sought to define step-by-step rules for solving common engineering problems relevant to the creation of buildings and communities.
A Brief History of Design Patterns

- 1987 OOPSLA - *Kent Beck and Ward Cunningham* at the OOPSLA-87 workshop on the Specification and Design for Object-Oriented Programming publish the paper: *Using Pattern Languages for Object-Oriented Programs*
  - Discovered Alexander's work for software engineers by applying 5 patterns in Smalltalk

- 1991 Erich Gamma came up with an idea for a Ph.D. thesis about patterns, and by 1992, he had started collaborating with the other GOF members (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) on expanding this idea.
  - Erik come up with the idea while working on an object oriented application framework in C++ called "ET++".

- Bruce Anderson gives first Patterns Workshop at OOPSLA
A Brief History of Design Patterns

- 1993 GOF submitted a catalog of object-oriented software design patterns to the European Conference of Object-Oriented Programming (ECOOP) in 1993
  
  *E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 97 LNCS 707, Springer, 1993*

- 1993 Kent Beck and Grady Booch sponsor the first meeting of what is now known as the Hillside Group

- 1994 - First Pattern Languages of Programs (PLoP) conference
A Brief History of Design Patterns

1995 – GOF publishes: Design Patterns. Elements of Reusable Object-Oriented Software

the most popular computer book ever published
1 million copies sold
Are you bored?

Let’s do some programming!!!!
The Job

Joe works at a company that produces a simulation game called *SimUDuck*. He is an OO Programmer and his duty is to implement the necessary functionality for the game.

The game should have the following specifications:
- A variety of different ducks should be integrated into the game
- The ducks should swim
- The duck should quake
A First Design for the Duck Simulator Game

All ducks `quack()` and `swim()`. The superclass takes care of the implementation.

The `display()` method is abstract since all the duck subtypes look different.

Each duck subtype is responsible for implementing its own `display()` behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck type.
Joe, at the shareholders meeting we decided that we need to crush the competition. From now on our ducks need to fly.

All subclasses inherit `fly()`.
But Something Went Wrong

Joe, I'm at the shareholder’s meeting. They just gave a demo and there were rubber duckies flying around the screen. Is this a joke or what?

OK, so there’s a slight flaw in my design. I don’t see why they can’t just call it a “feature”. It’s kind of cute.

By putting fly() in the superclass Joe gave flying ability to all ducks including those that shouldn’t.
void Duck::quack()
{
    cout << "quack, quack" << endl;
}

void RubberDuck::quack()
{
    cout << "squick, squick" << endl;
}

We can override the \texttt{fly()} method in the rubber duck in a similar way that we override the \texttt{quack()} method

void Duck::fly()
{
    \hspace{1cm} // fly implementation
}

void RubberDuck::fly()
{
    \hspace{1cm} // do nothing
}
How About an Interface

We can take the `fly()` out of the Duck superclass and make a Flyable interface with a method `fly()`. Each duck that is supposed to fly will implement that interface.

```
Flyable
+fly()

Quackable
+quack()

Duck
+quack()
+swim()
+display()
+fly()

MallardDuck
+display()
+fly()
+quack()

RedHeadDuck
+display()

RubberDuck
+display()
+quack()

DecoyDuck
+quack()
```

Really? I don't think so!

Brilliant
Yet Another Duck is Added to the Application

```cpp
void DecoyDuck::quack(){
  // do nothing;
}

void DecoyDuck::fly(){
  // do nothing
}
```
Embracing Change

In SOFTWARE projects you can count on one thing that is constant: **CHANGE**

**Solution**
- Deal with it.
  - Make CHANGE part of your design.
  - Identify what vary and separate from the rest.

- Let’s shoot some ducks!
Encapsulate that vary
The Constitution of Software Architects

- Encapsulate that vary.
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
Embracing Change in Ducks

- `fly()` and `quack()` are the parts that vary
- We create a new set of classes to represent each behavior

```cpp
void fly() = 0;

void quack() = 0;
```

```cpp
virtual void fly() = 0;

virtual void quack() = 0;
```

```cpp
void FlyWithWings::fly()
{   cout << "I'm flying!" << endl;
}

void FlyNoWay::fly()
{   cout << "I can't fly." << endl;
}

void Quack::quack()
{   cout << "Quack" << endl;
}

void Squeak::quack()
{   cout << "Squeak" << endl;
}

void MuteQuack::quack()
{   cout << "....." << endl;
}
```
Program to an interface not to an implementation
The Constitution of Software Architects

- Encapsulate that vary.
- Program to an interface not to an implementation.

- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
- ?????????
Integrating the Duck Behavior

The behavior variables are declared as the behavior interface type:

```
Duck
- flyBehavior : FlyBehavior
- quackBehavior : QuackBehavior
+ performQuack()
+ swim()
+ display()
+ performFly()
```

These methods replace `fly()` and `quack()`

```
class Duck{
    public:
        FlyBehavior *flyBehavior;
        QuackBehavior *quackBehavior;
        ...
        void performFly();
        void performQuack();
        ...
};
```

```
void Duck::performFly()
{
    flyBehavior->fly();
}
void Duck::performQuack()
{
    quackBehavior->quack();
}
```

MallardDuck:
```
MallardDuck()
{
    flyBehavior = new FlyWithWings();
    quackBehavior = new Quack();
}
```

RedHeadDuck:
```
RedHeadDuck()
{
    flyBehavior = new FlyWithWings();
    quackBehavior = new Quack();
}
```

RubberDuck:
```
RubberDuck()
{
    flyBehavior = new FlyNoWay();
    quackBehavior = new Squick();
}
```

DecoyDuck:
```
DecoyDuck()
{
    flyBehavior = new FlyWithWings();
    quackBehavior = new Quack();
}
```
<table>
<thead>
<tr>
<th>Duck</th>
</tr>
</thead>
<tbody>
<tr>
<td>-flyBehavior : FlyBehavior</td>
</tr>
<tr>
<td>-quackBehavior : QuackBehavior</td>
</tr>
<tr>
<td>+performQuack()</td>
</tr>
<tr>
<td>+swim()</td>
</tr>
<tr>
<td>+display()</td>
</tr>
<tr>
<td>+performFly()</td>
</tr>
</tbody>
</table>

Each Duck **HAS A** FlyingBehavior and a QuackBehavior to which it delegates flying and quacking.

**Composition**

Instead of inheriting behavior, the duck get their behavior by being composed with the right behavior object.
Design Principle

Favor Composition over Inheritance
Encapsulate that vary.

Program to an interface not to an implementation.

Favor Composition over Inheritance.
int main(){
    cout << "Testing the Duck Simulator" << endl << endl;
    Duck *mallard = new MallardDuck();
    mallard->display();
    mallard->swim();
    mallard->performFly();
    mallard->performQuack();
    cout << endl;
    Duck *rubberduck = new RubberDuck();
    rubberduck->display();
    rubberduck->swim();
    rubberduck->performFly();
    rubberduck->performQuack();
    cout << endl;
    return 0;
}

The mallard duck inherited `performQuack()` method which delegates to the object `QuackBehavior` (calls `quack()`) on the duck’s inherited `quackBehavior` reference.
Joe, I'm at the shareholder's meeting. The competitors are ahead us. They just released a new version of DOOM. Do something! It should be possible to shoot those damned ducks.

No problem boss. I can fix this. I will transform our Simulator into a duck shooting game.
Shooting Ducks Dynamically

```cpp
void Duck::setFlyBehavior(FlyBehavior *fb){
    flyBehavior = fb;
}
void Duck::setQuackBehavior(QuackBehavior *qb){
    quackBehavior = qb;
}

int main(){
    Duck *mallard = new MallardDuck();
mallard->display();
mallard->swim();
mallard->performFly();
mallard->performQuack();

cout << endl;
mallard->setFlyBehavior(new FlyNoWay());
mallard->setQuackBehavior(new MuteQuack());
mallard->performFly();
mallard->performQuack();

    return 0;
}
```
The Big Picture

Duck
- flyBehavior : FlyBehavior
- quackBehavior : QuackBehavior
+ performQuack()
+ swim()
+ display()
+ performFly()
+ setFlyBehavior()
+ setQuakBehavior()

1

MallardDuck
+ display()

RedHeadDuck
+ display()

RubberDuck
+ display()

DecoyDuck
+ display()

FlyBehavior
+ fly()

FlyWithWings
+ fly()

FlyNoWay
+ fly()

QuackBehavior
+ quack()

Quack
+ quack()

Squick
+ quack()

MuteQuack
+ quack()
Yet another Change

Joe, I'm at the movie theater. I just saw Star Wars. Great movie. I was just thinking maybe we should put some nice planes into our simulator. This will destroy our competitors.

OK, Ok... I will do that. But I need a raise.
Behavior Reuse
Congratulations !!!
This is your first pattern called STRATEGY
Strategy – defines a family of algorithms, encapsulate each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
Strategy – Non Software Example

TransportationToAirport

GotoAirport(Time, Cost)

Strategies (Options)

- Personal Car
- Taxi Cab
- Limousine
- City Bus
What are Patterns

- A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. “(D. Riehle/H. Zullighoven)

- The pattern is at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. (R. Gabriel)

- A pattern involves a general description of a recurring solution to a recurring problem with various goals and constraints. It identifies more than a solution, it also explains why the solution is needed. “(J. Coplien)

- ... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [Alexander]
Design Pattern Space

- **Creational patterns**
  - Deal with initializing and configuring of classes and objects

- **Structural patterns**
  - Deal with decoupling interface and implementation of classes and objects

- **Behavioral patterns**
  - Deal with dynamic interactions among societies of classes and objects
# Design Pattern Space

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter</td>
<td>Interperter</td>
</tr>
<tr>
<td>Scope</td>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td>Object</td>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facade</td>
<td>Momento</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flyweight</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>

- **Creational Patterns** include: Factory Method, Abstract Factory, Builder, Prototype, Singleton.
- **Structural Patterns** include: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- **Behavioral Patterns** include: Interperter, Chain of Responsibility, Command, Iterator, Mediator, Momento, Observer, State, Strategy, Visitor.
The Sacred Elements of the Faith

<table>
<thead>
<tr>
<th>the holy origins</th>
<th>the holy behaviors</th>
<th>the holy structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FM</strong> Factory Method</td>
<td><strong>CR</strong> Chain of Responsibility</td>
<td><strong>A</strong> Adapter</td>
</tr>
<tr>
<td><strong>PT</strong> Prototype</td>
<td><strong>CP</strong> Composite</td>
<td><strong>D</strong> Decorator</td>
</tr>
<tr>
<td><strong>S</strong> Singleton</td>
<td><strong>O</strong> Observer</td>
<td><strong>FA</strong> Façade</td>
</tr>
<tr>
<td><strong>AF</strong> Abstract Factory</td>
<td><strong>IN</strong> Interpreter</td>
<td><strong>PX</strong> Proxy</td>
</tr>
<tr>
<td><strong>TM</strong> Template Method</td>
<td><strong>CD</strong> Command</td>
<td><strong>V</strong> Visitor</td>
</tr>
<tr>
<td><strong>MD</strong> Mediator</td>
<td><strong>MD</strong> Observer</td>
<td><strong>FL</strong> Flyweight</td>
</tr>
<tr>
<td><strong>O</strong> Observer</td>
<td><strong>IN</strong> Interpreter</td>
<td><strong>V</strong> Visitor</td>
</tr>
<tr>
<td><strong>BU</strong> Builder</td>
<td><strong>ST</strong> State</td>
<td><strong>FL</strong> Flyweight</td>
</tr>
<tr>
<td><strong>SR</strong> Strategy</td>
<td><strong>IT</strong> Iterator</td>
<td><strong>BR</strong> Bridge</td>
</tr>
<tr>
<td><strong>MM</strong> Memento</td>
<td><strong>IT</strong> Iterator</td>
<td><strong>FL</strong> Flyweight</td>
</tr>
</tbody>
</table>
What’s In a Design Pattern--1994

- The GoF book describes a pattern using the following four attributes:
  - The **name** to describes the pattern, its solutions and consequences in a word or two
  - The **problem** describes when to apply the pattern
  - The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The **consequences** are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk
Closing remarks

- No Real Ducks have been harmed during this lecture.

Copyright © 2002 United Feature Syndicate, Inc.