# Design Patterns For Fun And Profit

...

Nikolas Vourlakis

# Agenda

- What
- Evolution
- Benefits
- Problems
- Composite
- Builder

# Evolution

Each pattern 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.

Christopher Alexander - A Pattern language: Towns, Buildings, Construction (1977)

# Evolution

5 patterns for designing window-based user interfaces

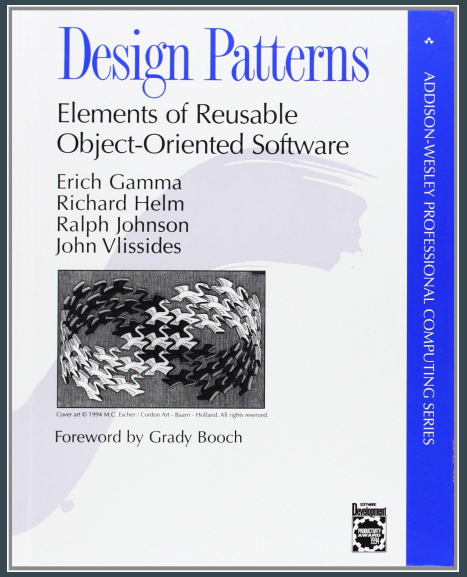
Kent Beck & Ward Cunningham at OOPSLA (1987)

http://c2.com/doc/oopsla87.html

# Evolution

*Program to an interface, not an implementation*

*Favor object composition over class inheritance*




# Evolution

Pattern Languages of Programs Conference

# Classification

- Point of view changes what is and isn't a pattern
- Choice of programming language is important
- Programming paradigm (Object-Oriented, Functional, Procedural, etc)

# Classification

- Creational
- Structural
- Behavioral
- Security
- Concurrency

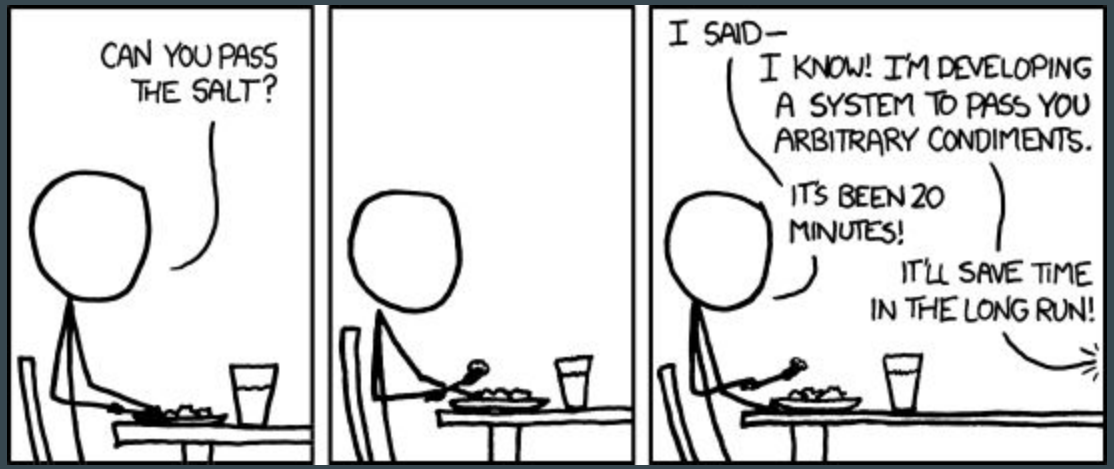
- Database/Sql
- User Interface
- Social
- Network
- Distributed

... and many more, the list is in constant growth

# What we Gain?

- Common vocabulary
- Identify less-obvious abstractions and the objects that can capture them
- Easier to understand existing systems ... sometimes
- Increase project code quality ... sometimes

# Over-engineer / Patterns Happy



# How to use then?

# Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Martin Fowler

# Design patterns provide targets for your refactorings !!!

# But How?

```
public class OrderSerializer {
   public String toXML(Order[] orders) {
      StringBuffer xml = new StringBuffer();
      xml.append("<orders>");
      for (Order order : orders) {
         xml.append("<order  id='"); xml.append(order.getId()); xml.append("'>");
         for (Product product : order.getProducts()) {
            xml.append("<product id='"); xml.append(product.getId()); xml.append("'>");
            xml.append("<name>"); xml.append(product.getName()); xml.append("</name>");
            xml.append("</product>");
         }
         xml.append("</order>");
      }
      xml.append("</orders>");
      return xml.toString();
   }
}
```

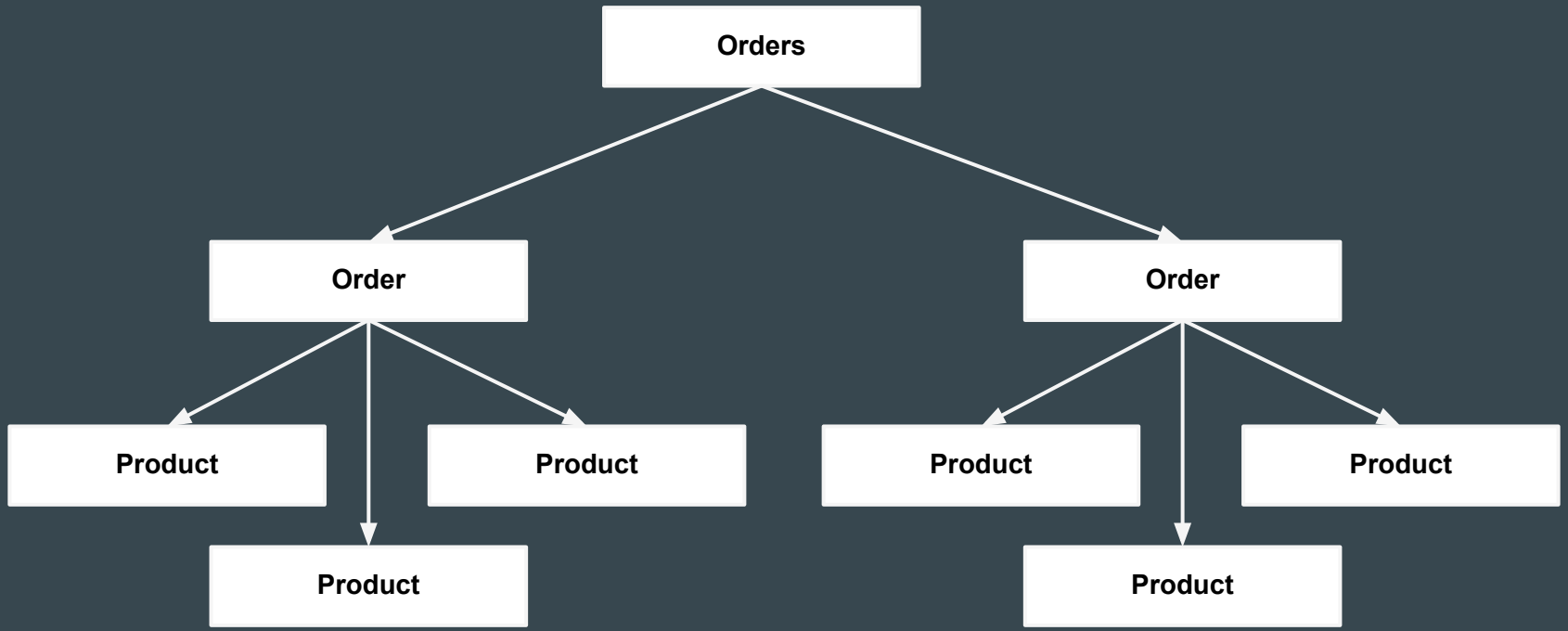
# Transform orders to XML

Taken from Refactoring to Patterns

# Problems

- Show 10 more properties of product?
- How about implementing ClientSerializer?
- Difficult to work with
- Similar logic all over the place
- Build & representation coupled together

# What do we have here?

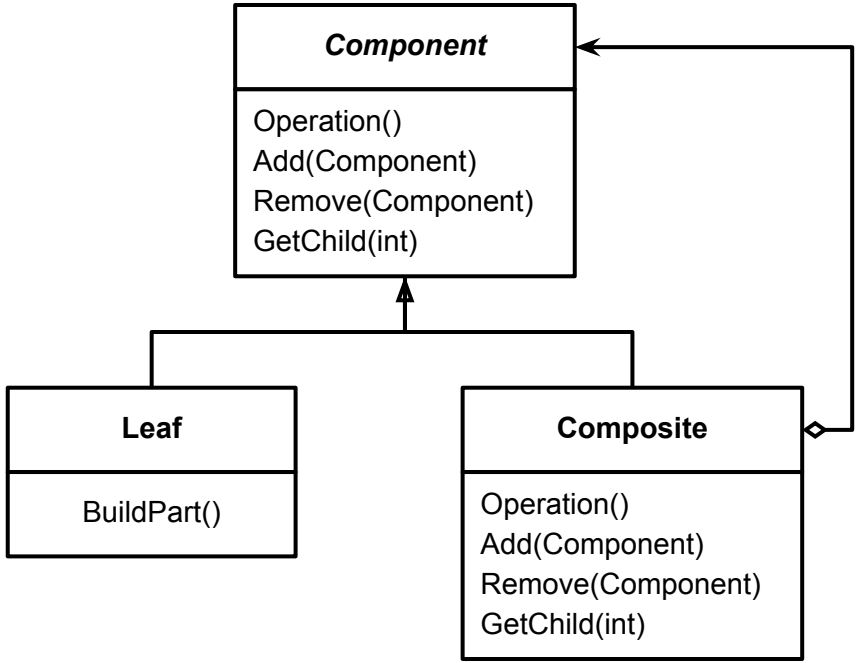

Orders
Order
Order
Product
Product
Product
Product
Product
Product

# A Tree!

# Composite

# Composite

- Structural pattern
- Represent part-whole hierarchies
- Treat these objects uniformly

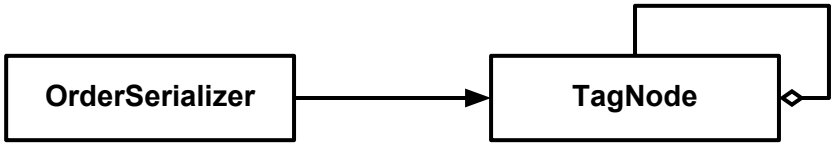


```
public class TagNode {
    /* private vars */
    public TagNode(String name) { /* ... */ }
    public void addValue(String value) { this.value = value; }
    public void addAttribute(String attribute, String value) {
        this.attributes.append(" "); this.attributes.append(attribute);
        this.attributes.append("="); this.attributes.append(value); this.attributes.append("");
    }
    public String getName() { /* ... */ }
    public void setParent(TagNode parent) { /* ... */ }
    public TagNode getParent() { /* ... */ }

    public void add(TagNode node) { /* ... */ }
    public String toString() {
        String result; result = "<" + name + attributes + ">";
        for (Object childNode : this.children) {
            TagNode node = (TagNode) childNode; result += node.toString();
        }
        result += value; result += "</" + name + ">";
        return result;
    }
}
```

```
public class OrderSerializer {
    public String toXML(Order[] orders) {
        TagNode ordersTag = new TagNode("orders");

        for (Order order : orders) {
            TagNode orderTag = new TagNode("order");
            orderTag.addAttribute("id", String.valueOf(order.getId()));

            for (Product product : order.getProducts()) {
                TagNode nameTag = new TagNode("name");
                nameTag.addValue(product.getName());
                TagNode productTag = new TagNode("product");
                productTag.addAttribute("id", String.valueOf(product.getId()));
                productTag.add(nameTag);
                orderTag.add(productTag);
            }
            ordersTag.add(orderTag);
        }
        return ordersTag.toString();
    }
}
```

# Consequences

+ Hide repetitive instructions
+ Generalized way to handle similar logic
+ Removes responsibilities from client
- Complicate design



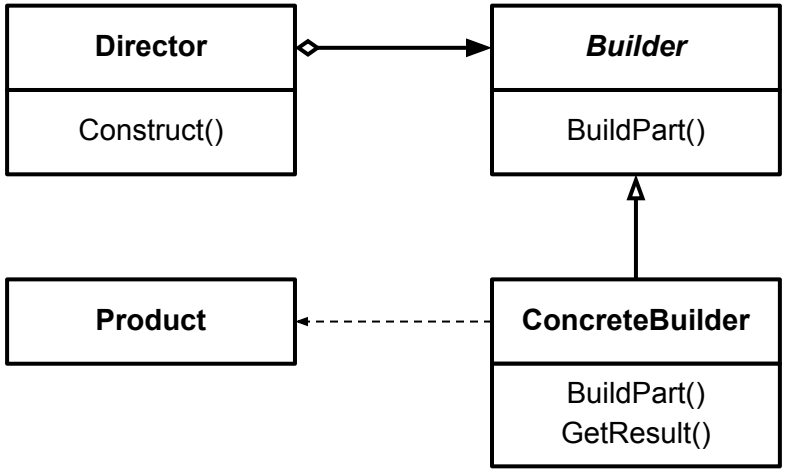
# New Problems

- Knowledge how a complex object is constructed
- Client coupled with composite
- Difficult if we need to swap TagNode with something else in the future

# Builder

# Builder

- Creational pattern
- Separate construction of a complex object from its internal representation
- Able to create different representations with the same construction process

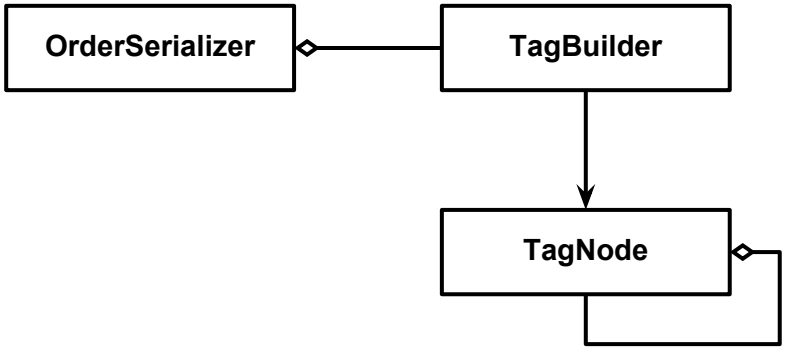


```
class TagBuilder {
    private final TagNode root;
    /* other private members */

    public TagBuilder(String name) {
        root = new TagNode(name);
    }

    public String toXml() {
        return root.toString();
    }

    public void addToParent(String parentName, String childName) { /* ... */ }
    public void addChild(String name) { /* ... */ }
    public void addSibling(String name) { /* ... */ }
    public void addAttribute(String attribute, String value) { /* ... */ }
    public void addValue(String value) { /* ... */ }
}
```

```
public class OrderSerializer {
    public String toXML(Order[] orders) {
        TagBuilder builder = new TagBuilder("orders");
        for (Order order : orders) {
            builder.addToParent("orders", "order");
            builder.addAttribute("id", "1");
            for (Product product : order.getProducts()) {
                builder.addToParent("order", "product");
                builder.addAttribute("id", String.valueOf(product.getId()));
                builder.addChild("name");
                builder.addValue(product.getName());
            }
        }
        return builder.toXml();
    }
}
```

# Consequences

+ Simplify client construction code
+ Reduces repetitive and error-prone creation process
+ Client is not aware of the composite
+ We are free to change composite whenever we want
- Complicate design



# What to Read?

- Head First Design Patterns
- Design Patterns, Elements of Reusable Object-Oriented Software
- Refactoring to Patterns
- Patterns of Enterprise Application Architecture

# Questions?