**History**

- **The Gang of Four (GoF)**
  - Their book, Design Patterns: Elements of Reusable Object-Oriented Software, was released at OOPSLA 1994.
  - The book and the authors have won several awards including the Jolt productivity award, the Software Development productivity award, and the SIGSOFT Outstanding Research award.

**Pre-history: Christopher Alexander**

“...At the core... is the idea that people should design for themselves their own houses, streets and communities. This idea... comes simply from the observation that most of the wonderful places of the world were not made by architects but by the people.”

—Christopher Alexander, *A Pattern Language*

*bing “design patterns” on 5/2/2011*

65.5M hits including…

- .NET Design Patterns in C# and VB.NET
- Yahoo! Design Pattern Library
- Azure Design Patterns
- ASP.NET Wiki: Architecture; Design Patterns
- SOA patterns
- Design Patterns for Building Flexible and Maintainable J2EE Applications
- Design Patterns and Refactoring
- Design Patterns in Ruby
- Ajax Patterns
- CSS Design Patterns
- PHP Design Patterns
- Train the Trainer Design Patterns Design Patterns Training
- … and millions more!
What are design patterns?

- Solutions to commonly arising object-oriented design problems – solutions actually used multiple times by multiple people over time
- Stylized descriptions that include (in part)
  - a motivation (the problem and the context),
  - a design-level description (in terms of interfaces and interconnections),
  - one or more example implementations in a well-known programming language
- "a ‘well-proven generic scheme’ for solving a recurring design problem”
- Often overcoming limitations of OO hierarchies
- Idioms intended to be “simple and elegant solutions to specific problems in object-oriented software design”
- Patterns are a collection of “mini-architectures” that combine structure and behavior
- Gabriel: “Alexander could have written a 1-sentence definition of what a pattern is, or an essay, but instead he wrote a 550-page book to do it. Because the concept is hard.”

Similar in motivation to PL constructs
Simple example: singleton pattern
Only one object of the given type exists

```java
class Bank {
    private static Bank theBank;

    // private constructor
    private Bank() { ... }

    // factory method
    public static Bank getBank() {
        if (theBank == null) {
            theBank = new Bank();
        }
        return theBank;
    }
    ...
}
```

- Pattern isn't explicit in code
- Could be defined by a programming language
- Would appear differently in different languages

Interning pattern

- Reuse existing objects instead of creating new ones
  - Less space
  - May compare with == instead of equals()
- Permitted only for immutable objects

Interning mechanism

- Maintain a collection of all objects
- If an object already appears, return that instead
  ```java
  HashMap<String, String> segnames;
  String canonicalName(String n) {
      if (segnames.containsKey(n)) {
          return segnames.get(n);
      } else {
          segnames.put(n, n);
          return n;
      }
  }
  ```
- Java builds this in for strings: String.intern()
- Two approaches
  - create the object, but perhaps discard it and return another
  - check against the arguments before creating the new object
**java.lang.Boolean does not use the Interning pattern**

```java
public class Boolean {
    private final boolean value;

    // constructor
    public Boolean(boolean value) {
        this.value = value;
    }

    public static Boolean FALSE = new Boolean(false);
    public static Boolean TRUE = new Boolean(true);

    // factory method that uses interning
    public static Boolean valueOf(boolean value) {
        if (value) {
            return TRUE;
        } else {
            return FALSE;
        }
    }
}
```

**Recognition of the problem**

- Javadoc for Boolean constructor includes...
  - Allocates a Boolean object representing the value argument
  - “Note: It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.”

- Josh Bloch (JavaWorld, January 4, 2004):
  - “The Boolean type should not have had public constructors. There’s really no great advantage to allow multiple trues or multiple falses, and I’ve seen programs that produce millions of trues and millions of falses, creating needless work for the garbage collector.”
  - “So, in the case of immutables, I think factory methods are great.”

**My eye-opening experience**

- At Dagstuhl
- With Griswold
- And with Helms and Vlissides
- In the bar
- “How to rethink our design for a program restructuring system?”

**Patterns are an example of chunking**

- Advanced chess players are in part superior because they don’t see each piece individually
  - Instead, they chunk groups of them together
  - This reduces the search space they need to assess in deciding on a move

- This notion of chunking happens in almost every human endeavor
- Such chunking can lead to the use of idioms
  - As it has in programming languages
  - The following slides show some parts of a particular pattern: flyweight
  - I won’t go through the slides in detail, but they give a feel for people who haven’t seen more concrete information on patterns
Example: flyweight pattern

- What happens when you try to represent lots of small elements as full-fledged objects?
- It's often too expensive
- And it's pretty common

An alternative approach

- Use sharing to support many fine-grained objects efficiently
  - Fixed domain of objects
  - Maybe other constraints
  - Similar to interning

Flyweight structure

Participants

- Flyweight (glyph in text example)
  - Interface through which flyweights can receive and act on extrinsic state
- ConcreteFlyweight (character)
  - Implements flyweight interface, shareable, only intrinsic state (independent of context)
- UnsharedConcreteFlyweight (row, column)
- FlyweightFactory
  - Creates and manages flyweight objects
### Sample code

```cpp
class Glyph {
public:
    virtual ~Glyph();
    virtual void Draw(...);
    virtual void SetFont(...);

private:
    static Glyph* _single_instance;
};
```

```cpp
class Character : public Glyph {
    Character(char c);
    virtual void Draw(...);

private:
    char _charcode;
};
```

- The code itself is in the domain (glyphs, rows, etc.)
- But it's structured based on the pattern
- The client interacts with Glyph, Character

### A little more code

```cpp
Character* GlyphFactory::CreateCharacter(char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }
    return _character[c];
}
```

- Explicit code for each of the elements in the flyweight structure

### Classes of patterns: structural

- These provide ways to compose interfaces and define ways to compose objects to obtain new functionality
  - Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
  - Bridge decouples an abstraction from its implementation so that the two can vary independently
  - Composite composes zero-or-more similar objects so that they can be manipulated as one object
  - Decorator dynamically adds/overrides behavior in an existing method of an object
  - Facade provides a simplified interface to a large body of code
  - Flyweight reduces the cost of creating and manipulating a large number of similar objects
  - Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity

### Classes of patterns: creational

- For instantiating classes and objects
  - Class-creation patterns tend to exploit inheritance
  - Object-creation patterns tend to exploit delegation
  - Abstract Factory groups object factories that have a common theme
  - Builder constructs complex objects by separating construction and representation
  - Factory Method creates objects without specifying the exact class to create
  - Prototype creates objects by cloning an existing object
  - Singleton restricts object creation for a class to only one instance
These patterns are concerned with communication between objects.

- Chain of responsibility delegates commands to a chain of processing objects
- Command creates objects which encapsulate actions and parameters
- Interpreter implements a specialized language
- Iterator accesses the elements of an object sequentially without exposing its underlying representation
- Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods
- Memento provides the ability to restore an object to its previous state (undo)
- Observer is a publish/subscribe pattern which allows a number of observer objects to see an event
- State allows an object to alter its behavior when its internal state changes
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior
- Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object

Patterns: ergo, anti-patterns

- Rarely clear to me how they are actionable
- But they have lots of cute names
- God object
- Object cesspool
- Object orgy
- Poltergeists
- Yo-yo problem
- Big ball of mud
- Gold plating
- Magic pushbutton

Organizational/management anti-patterns

- Analysis paralysis
- Cash cow
- Design by committee
- Escalation of commitment
- Management by perkele
- Moral hazard
- Mushroom management
- Stovepipe or Silos
- Death march
- Groupthink
- Smoke and mirrors
- Software bloat

Design patterns: not a silver bullet...

- ..but they are impressive, important and worthy of attention and study
- I think that some of the patterns have and more will become part and parcel of designers' vocabularies
- This will improve communication and over time improve the designs we produce
- The relatively disciplined structure of the pattern descriptions may be a plus
Show trial

- “Indeed, this notorious cabal will soon be brought to justice at OOPSLA ’99 during a panel entitled the ShowTrialOfTheGangOfFour for crimes against computer science.” [http://c2.com/cgi/wiki?GangOfFour]
- “The Accused, by making it possible to design object-oriented programs in C++, have inhibited the rightful growth of competing object-oriented languages such as SmalltalkLanguage, CommonLisp, and JavaLanguage.”
- “The Accused have engaged in an usurpation of perfectly good English words and well-known technical terms for the purpose of establishing an arcane argot known only to a narrow circle of GoF initiates.”
- “The Accused, by cataloging mere experience, rather than conducting novel research, have displayed an utter disregard for traditional standards of academic originality.”
- ...

Discussion?

- Questions might include
  - What do patterns say, if anything, about the correspondence between names and invokes relations?
  - Should patterns be turned into PL constructs? If so, why and when? If not, why not?
  - Does AOP save the day with patterns (at least with Observer)?
  - What’s on your mind about patterns?