Comp-304 : Intro to Design Patterns / Singleton
Lecture 21

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2007
In the 1970s, an architect named Christopher Alexander started to question himself about design.

- How do I know if an architecture design is good?

Alexander proposed that there was an objective way of measuring quality of design.

He studied the architecture of many cities (buildings, streets, parks, etc).

He discovered that, although each architecture is different, they can still be considered high quality.
Front Porch
A solution to a problem

- Two porches may appear structurally different, and yet they may still be considered of high quality.
  - One porch might be a simple transition from the front yard to the front door.
  - Another might also be a resting area.
- However, they both solve a common problem of transition.
- By comparing two items that solve a common problem, one can identify similarities between the designs that are of high quality.
- Alexander called these similarities Patterns.
In 1987, Kent Beck and Ward Cunningham began experimenting with Design Patterns.

They believed that this idea of patterns as solutions to common problems could be used with software.

In 1994, Erich Gamme, Richard Helm, Ralph Johnson and John Vlissides published Design Patterns: Elements of Reusable Object-Oriented Software.

This book, also known as the Design Pattern bible, helped Design Patterns gain popularity with the Computer Science community.

In recognition for their important work, the four authors are known as the gang of four.
What are Patterns?

"A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." [Buschmann].

- Patterns are a solution to a problem in a context.
- Patterns are not invented, they are derived from practical experience.
- Patterns are construction blocks, to be used to solve complex problems.
- Patterns can be used as a vocabulary to communicate.
Why use Patterns?

- Because Patterns are well tested and well proven solution to common problems.
  - They have been successfully used in the past.
  - They are a form of code reuse.
- Patterns are to Design what Libraries are to Software.
Components of a Design Pattern

- **Name**
  - Each pattern has an assigned name so it can be easily recognized.
  - This gives us the vocabulary we can use to discuss design.

- **Problem**
  - Each pattern is designed to address a specific problem.
  - Some also have conditions before the pattern can be used.

- **Solution**
  - Each pattern provides a solution to a problem.
  - Components of that solution are also known as Participants.

- **Consequence**
  - They are the results and trade-off of using design patterns.
Types of Design Patterns

- Creational Patterns
  - These patterns abstract the instantiation process.
  - They make the system independent of how objects are created, composed and represented.

- Structural Patterns
  - These patterns are concerned with how classes and objects are composed to form larger structures.
  - These structures are use to provide new functionalities

- Behavioral Patterns
  - These patterns are concerned with algorithms and the assignment of responsibility between objects.
  - They describe the communication between objects.
The Design Patterns book is a catalog of design patterns.

When faced with a design pattern, one should:

- Browse the catalog to determine if a particular design pattern solves this pattern.
- If so, before implementing the solution,
  - Carefully identify the various participants of the problems.
  - Study thoroughly the appropriate section in the book, particularly the consequence and implementation section.
The material you will see in Design Patterns is not new.
Some of you might have been using this stuff for years.
That's the whole point.
It's a catalog of good design.
If you have already been using a Pattern, then
  • You now have an official name for it.
  • You know it good design.
  • You might gain a few new incites on how to use it.
Just a few example?

- Command
- Adapter
- Proxy
- Composite
- Observer
- Template Method
- Visitor
- Factory
Mammoth is a massively multiplayer game research framework.

The world of Mammoth is a 2D environment viewed from a 2D perspective.

The world contains a fixed number of game objects, some of which can be controlled by humans (players).

A player can move around in the game, examine objects, pick them up, and drop them again.
Each object in the world (player, items, grass, etc) has a unique Id associated to it.

How do I distribute Ids, making sure that I never distribute a duplicate one?
Mammoth uses unique identifiers (ID) to identify all the Game objects in the world.

These IDs are distributed by a single object.
- If more than one distributor were used, duplicate IDs could be distributed.

The application needs global access to this distributor.
- It would be very complicated/ugly to pass around the reference to the distributor all around the application.
- We need to make sure that only one instance of a class can be created.
- We want that instance to be easy to access anywhere in the application.
Singleton

- Ensure a class only has one instance, and provide a global point of access to it.
<table>
<thead>
<tr>
<th>Class Diagram</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance: Singleton</td>
</tr>
<tr>
<td>-constructor()</td>
</tr>
<tr>
<td>+getInstance(): Singleton { return instance; }</td>
</tr>
</tbody>
</table>
public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton getInstance() {
        return Singleton.instance;
    }
}

Consequences

- You are assured that only one instance can be created.
  - Global access to that instance without the use of a global variable (less pollution)
- Can be modified to allow a fix number of instances.
- Singletons can be sub-classed.
public class IdDistributor {

    private static IdDistributor instance = new IdDistributor();
    private long lastId;

    private IdDistributor() {
        this.lastId = -1;
    }

    public static IdDistributor getInstance() {
        return IdDistributor.instance;
    }

    public long getId() {
        this.lastId++;
        return this.lastId;
    }
}
public class Singleton {

    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (Singleton.instance == null) {
            Singleton.instance = new Singleton();
        }
        return Singleton.instance;
    }

    return Singleton.instance;
}
}
public class Singleton {

    private static Singleton instance;

    private Singleton() {
    }

    public static synchronized Singleton getInstance() {
        if (Singleton.instance == null) {
            Singleton.instance = new Singleton()
        }
        return Singleton.instance;
    }

    return Singleton.instance;
}
}