Comp-303 : Programming Techniques
Lecture 17

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- The purposes, goals and process of design.
- Between design and implementation, a systematic design review should take place.
  - Procedures
  - Types
  - Modules
- Before the implementation, several big decision must be made : top-down vs. bottom-up.

Origins of Design Patterns

- Years ago, an architect named Christopher Alexander asked himself: "Is quality objective?"
- This architect was looking to measure if an architectural design is good.
- In other word, he wanted to find a way where beauty could be measured through an objective basis.

From Anthropology to Architecture

- Within a culture, individuals will agree to a large extend on what is considered to be a good design.
- These behavior of a culture can be broken down in patterns.
- Thus, we only need to determine the patterns that people consider good design.

What are patterns?

- Christopher Alexander tried separating patterns into categories of architectural structures.
- However, two structures (i.e porches) may appear structurally different, yet both be of high quality.
- Instead, he looked at separating structure structures depending on the problem they solve.
- By narrowing his focus this way, he discovered that he discern similarities between designs that were high quality.
- He defined a pattern as *a solution to a problem in context.*

From Architecture to Software

- In the early 1990s, some developers happened upon Alexander's work in patterns.
- They wondered if what was true for architectural patterns was also true for software design.
- The answer was yes.

The Design Pattern Bible

- In the 1990s, Gamma, Helm, Johnson and Vlissides published what is known today as the bible of Design Patterns.
- Because of the success of this work, these four authors are know as *the gang of four*.
- The book is a catalogue of 23 design patterns.
- It also provides a look into the usefulness of design patterns.
- The book also provides a standard of describing design patterns.

References

- For the next 6 or 7 classes, I will be relying heavily on these three books:
  - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
  - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesly; 2000
  - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002

Why do we use Design Patterns?

- Design Patterns represent proven solutions to well known problems.
- Design Patterns also allow us to have a common terminology.
- Design Patterns give you a higher level view of problems and designs, thus allowing you solve them without immediately going into details.

Describing a Design Pattern

The description of a design pattern can be broken down into the following sections:

- *Pattern Name and Classification* : The name of the pattern and it's classification.
  - Creational: Patterns that deal with creating objects
  - Structural: Patterns that deal with how we order/organize our objects/classes
  - Behavioral: Patterns that deal with how objects interact
- *Intent* : What does the pattern do? What problem does it address?

Describing a Design Pattern

- *Also Known As* : Some patterns have other well-known name.
- *Motivation* : A scenario that describes the problem and shows how it is solved.
- *Applicability* : When must we apply the pattern? How do I recognize these situation.
- *Structure* : UML diagram of the pattern.
- *Participants* : Classes/objects participating in the design pattern and their responsibilities.
- *Collaborations* : How participants collaborate.

Describing a Design Pattern

- *Consequences* : What are the trade-offs and results of using this pattern.
- *Implementation* : How do we build it?
- *Sample Code* : Example code on how to build the pattern.
- *Known Uses* : Examples of where the pattern can be found.
- *Related Patterns* : How are the other patterns related to this one?

Which pattern will we look at?

- Previously: Iterator
- Today: Singleton
- March 16th: Factory / Abstract Factory
- March 18th: Adapter and Bridge
- March 23rd: Flyweight
- March 25th: Chain of responsibility
- March 30th: Command
- April 1st: Haven't decided yet

Singleton

- *Pattern Name* : Singleton
- *Classification* : Creational
- *Intent* : Ensure a class only has one instance, and provide a global point of access to it.

Motivation

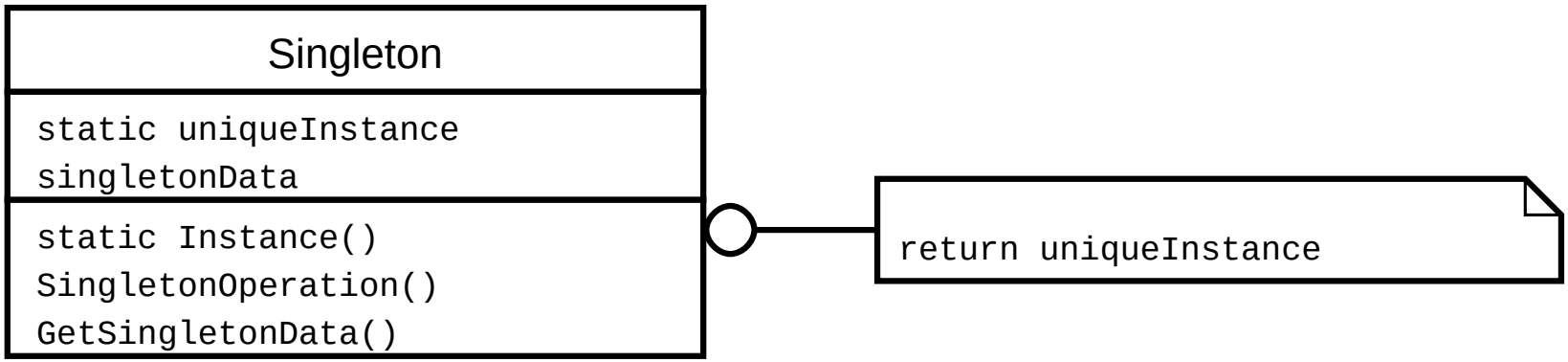
- Some classes should have only one instances.
- For example, a computer should only have one print spooler, one window manager, etc.
- But how do we make sure that only one instance of a class can be created.
- Globals are ugly, so we want to avoid them. Singletons can actually replace global variables in many situations.

Applicability

The Singleton pattern should be used when ...

- ... there must be exactly one instance of a class and it must be accessible to clients from a well-known access point.
- ... the sole instance should extensible by subclassing, and clients should be able to use an extended instance without modifying there code.

Structure


Singleton
static uniqueInstance
singletonData
static Instance()
SingletonOperation()
GetSingletonData()
return uniqueInstance

Participants

- The Singleton Object itself
  - Defines an operation (*Instance()* ?) which returns an instance of itself.
  - May be responsible for it's own creation.

Collaborations

- Other objects can only use the single instance of that object.

Consequences

- Controlled access to sole instance: Since Singleton is the sole instance of that class, the programmer has complete control on how and when the clients use it.
- Reduce name space: The use of singleton allows us to avoid globals, which are ugly.
- Permits refinement of operations and representation: The singleton can be subclassed.
- Permits a variable number of instance: You can easy modify a singleton to allow a limited number of instance to be created. Of course, you would need to modify the *Instance()* method.
- More flexible than class operations: In other words, they can be implemented with static members.

Implementation

- When implementing a singleton, the first thing we need to do is to prevent the user from creating multiple instances.
- One strategy is to prevent the user from creating the object himself.
- We can block of the default constructor by making it private.
- Then, we only need to provide a static method to allow the user to retrieve the singleton.

First Example

```
public class MySingleton{
  private static MySingleton singleInstance;

  private MySingleton() {
    //Empty Constructor
  }

  public static MySingleton instance(){
    if (singleInstance == null) {
     singleInstance = new MySingleton();
    }
    return singleInstance;
  }

  public int methodX() P
   // do something here
  }
}
```

Using the First Example

```
...

MySingleton singleObject = MySingleton.instance();

int i = singleObject.methodX();

...
```

Good, but not perfect

- The previous example is fairly simple and will only allow one instance of the class.
- If the user tries to create a MySingleton manually, it will fail since all constructors are private.
- However, when the user calls *instance()*, he has no way of knowing if the singleton already existed.
- This method also doesn't survive subclassing unless you force the user to re-implement the *instance()* method.

Second Example

```
public class MySingleton{
  static boolean instanceFlag = false;

  public MySingleton() {
    if (instanceFlag == true) {
      throw new SingleException("Only one of this type allowed");
    } else {
      instanceFlag = true;
    }
  }

  public int methodX() P
   // do something here
  }
}
```

Still not perfect

- This implementation is a little bit easier to subclass. However, the subclass *must* call the constructor of the superclass.
- The second example also provides more flexibility with the constructor.
- However, the user had no global visibility of the existing object.

Known Uses

- The *StringManager* Class in Tomcat is a singleton.
- Each package in Tomcat has a *StringManager* object to manage error messages.
- When an error message needs to be printed, the program must get the handle for the *StringManager* of that package.
- It would be wasteful to create two *StringManager* for the same package.
- Instead, if the *StringManager* object is already create, it will be returned (instead of creating a new object).

Summary

- The origins of Design Patterns can be traced back to Architecture.
- By using Design Patterns, we are re-using solutions to well know problems.
- The Singleton pattern should be used when only one copy of an object should be created.
- There are two ways to implement singletons in Java, each with their own advantages.

Tool of the day: Me and the T.A.

- Office Hours:
  - Teacher : Tuesday & Thursday: 1h00 - 2h30
  - T.A. : Monday & Wednesday: 12h00 - 13h30
- We are receiving very little visitors during our office hours.
- Only two teams have asked advice on the architecture of their project.
- If you don't ask us questions, we can't help out.
- The correction for the project is very harsh and very competitive.
- The only exception ... I don't know Swing or AWT.

Announcements

- Please be in class on times.
- The courses on design patterns are shorter, so getting to class on time is important.
- Interview week for the project will be held after midterm 2.