Software Engineering
Design Patterns

Lecture 8: Introduction and Singleton

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

Outline

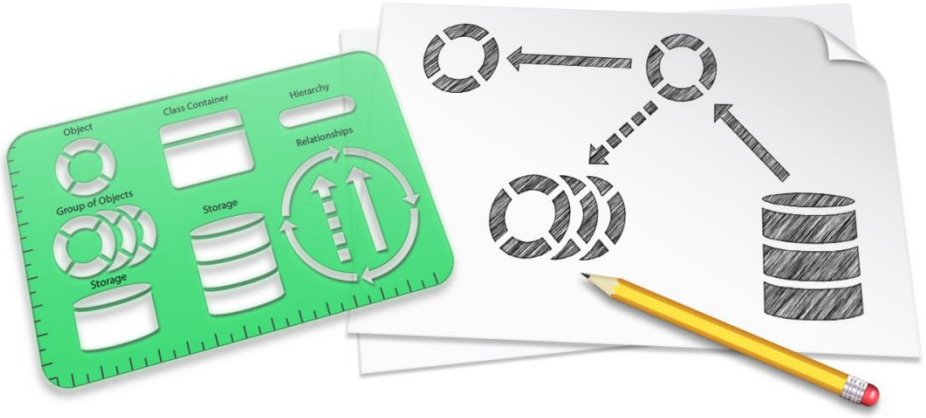
- *Introduction to Design Patterns*
  - What is a Design Pattern?
  - History of Design Patterns
  - The Gang of Four
  - Tangent: Unit Testing
- *The Singleton Pattern*
  - Logger Example
  - Lazy Instantiation
  - Singleton vs. Static Variables
  - Threading: Simple

What is a Design Pattern?

- *A problem that someone has already solved.*
- *A model or design to use as a guide*
- *More formally: "A proven solution to a common problem in a specified context."*

Real World Examples

- *Blueprint for a house*
- *Manufacturing*



Why Study Design Patterns?

- *Provides software developers a toolkit for handling problems that have already been solved.*
- *Provides a vocabulary that can be used amongst software developers.*
  - The Pattern Name itself helps establish a vocabulary
- *Helps you think about how to solve a software problem.*

History of Design Patterns

- *Christopher Alexander (Civil Engineer) in 1977 wrote*
  - "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*."

History (continued)

- *Each pattern has the same elements*
  - ***Pattern Name*** – helps develop a catalog of common problems
  - ***Problem*** – describes when to apply the pattern. Describes problem and its context.
  - ***Solution*** – Elements that make up the design, their relationships, responsibilities, and collaborations.
  - ***Consequences*** – Results and trade-offs of applying the pattern

History (continued)

- *In 1995, the principles of Alexander applied to software design and architecture. The result was the book:*

*"**Design Patterns: Elements of Reusable Object-Oriented Software**" by **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.***

*Also commonly known as "**The Gang of Four**".*

The Gang of Four

- *Defines a Catalog of different design patterns.*

- *Three different types*
  - ***Creational*** – "creating objects in a manner suitable for the situation"
  - ***Structural*** – "ease the design by identifying a simple way to realize relationships between entities"
  - ***Behavioral*** – "common communication patterns between objects"

The Gang of Four: Pattern Catalog

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	• Factory Method	• Adapter	• Interperter
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Momento • Observer • State • Strategy • Vistor

Reality

- *Problems with design early on*
  - It is sometimes very hard to “see” a design pattern.
  - Not all requirements are known.
  - A design that is applicable early on becomes obsolete.

- *Question: How do you moderate the fact that you won’t have all of the design figured out?*

Tangent: Unit Testing

- *create unit tests early in the development cycle*
  - it will be easier to refactor later on when more requirements are known.
  - As a developer, you will have more confidence to make good design adjustments.
- *What happens if you do not have Unit Tests early on? These statements may be heard:*
  - " I am afraid to break something."
  - "I know the right thing to do….but I am not going to do it because the system may become unstable."

Unit Testing (cont)

- *Unit Testing leads to easier Refactoring*
- *With easier Refactoring, you can take the risk of applying Design Patterns, even if it means changing a lot of code.*
- *Applying Design Patterns can improve the maintainability and extendibility of your system.*

Therefore…it pays to Unit Test!

Unit Testing: Final Thoughts

- *Make unit testing part of the project culture.*
- *When creating a schedule, include unit testing in your estimates.*
- *Create your unit tests before you write the code.*
  - Helps you think about how software needs to be layered…it may actually lead to more refactoring!

Common Pitfall

- *"I just learned about Design Pattern XYZ. Let's use it!"*
- *Reality: If you are going to use a Design Pattern, you should have a reason to do so.*
- *The software requirements should really drive why you are going to use (or not use) a Design Pattern.*

Outline

- *Introduction to Design Patterns*
  - What is a Design Pattern?
  - History of Design Patterns
  - The Gang of Four
  - Tangent: Unit Testing
- *The Singleton Pattern*
  - Logger Example
  - Lazy Instantiation
  - Singleton vs. Static Variables
  - Threading: Simple, Double-Checked, Eager Initialization

Example: Logger

What is wrong with this code?

```
public class Logger
    {
        public Logger() { }

        public void LogMessage() {
            //Open File "log.txt"
            //Write Message
            //Close File
        }
    }
```

Example: Logger (cont)

- *Since there is an external Shared Resource ("log.txt"), we want to closely control how we communicate with it.*

- *We shouldn't have to create the Logger class every time we want to access this Shared Resource. Is there any reason to?*

- *We need ONE.*

Singleton

- *GoF Definition: "The Singleton Pattern ensures a class has only **one instance**, and provides a global point of access to it."*
- *Best Uses*
  - Logging
  - Caches
  - Registry Settings
  - Access External Resources
    - *Printer, Device Driver, Database*

Logger – as a Singleton

See Chapter 5 in Head First Design Pattern Book

```
public class Logger
{
    private Logger(){}

    private static Logger uniqueInstance;

    public static Logger getInstance()
    {

      if (uniqueInstance == null)
            uniqueInstance = new Logger();

          return uniqueInstance;

    }
}
```

Lazy Instantiation

- *Objects are only created when it is needed*
- *Helps control that we've created the Singleton just once.*
- *If it is resource intensive to set up, we want to do it once.*

Singleton vs. Static Variables

- *What if we had **not** created a Singleton for the Logger class??*

- *Let's pretend the Logger() constructor did a lot of setup.*

In our main program file, we had this code:

```
public static Logger MyGlobalLogger = new Logger();
```

- *All of the Logger setup will occur regardless if we ever need to log or not.*

Threading

```
public class Singleton
{
   private Singleton() {}

   private static Singleton uniqueInstance;
   public static Singleton getInstance()
   {

     if (uniqueInstance == null)
           uniqueInstance = new Singleton();

        return uniqueInstance;

   }
}
```

What would happen if two different threads accessed this line at the same time?

Simple Locking (Expensive)

```
public class Singleton
{
   private Singleton() {}

   private static Singleton uniqueInstance;

   public static synchronized Singleton getInstance()
   {
        if (uniqueInstance == null)
                    uniqueInstance = new Singleton();

         return uniqueInstance;
   }
}
```

SUMMARY

- ***Pattern Name*** *– Singleton*
- ***Problem*** *– Ensures one instance of an object and global access to it.*
- ***Solution***
  - Hide the constructor
  - Use static method to return one instance of the object
- ***Consequences***
  - Lazy Instantiation
  - Threading
  - Difficult unit testing