SOFTWARE ENGINEERING
(Week-14)

USAMA MUSHARAF

MS-CS (Software Engineering)

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

AGENDA OF WEEK # 14

SOLID Principles (Cont…)

Software Design Patterns

LISKOV SUBSTITUTION PRINCIPLE

LISKOV SUBSTITUTION PRINCIPLE

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

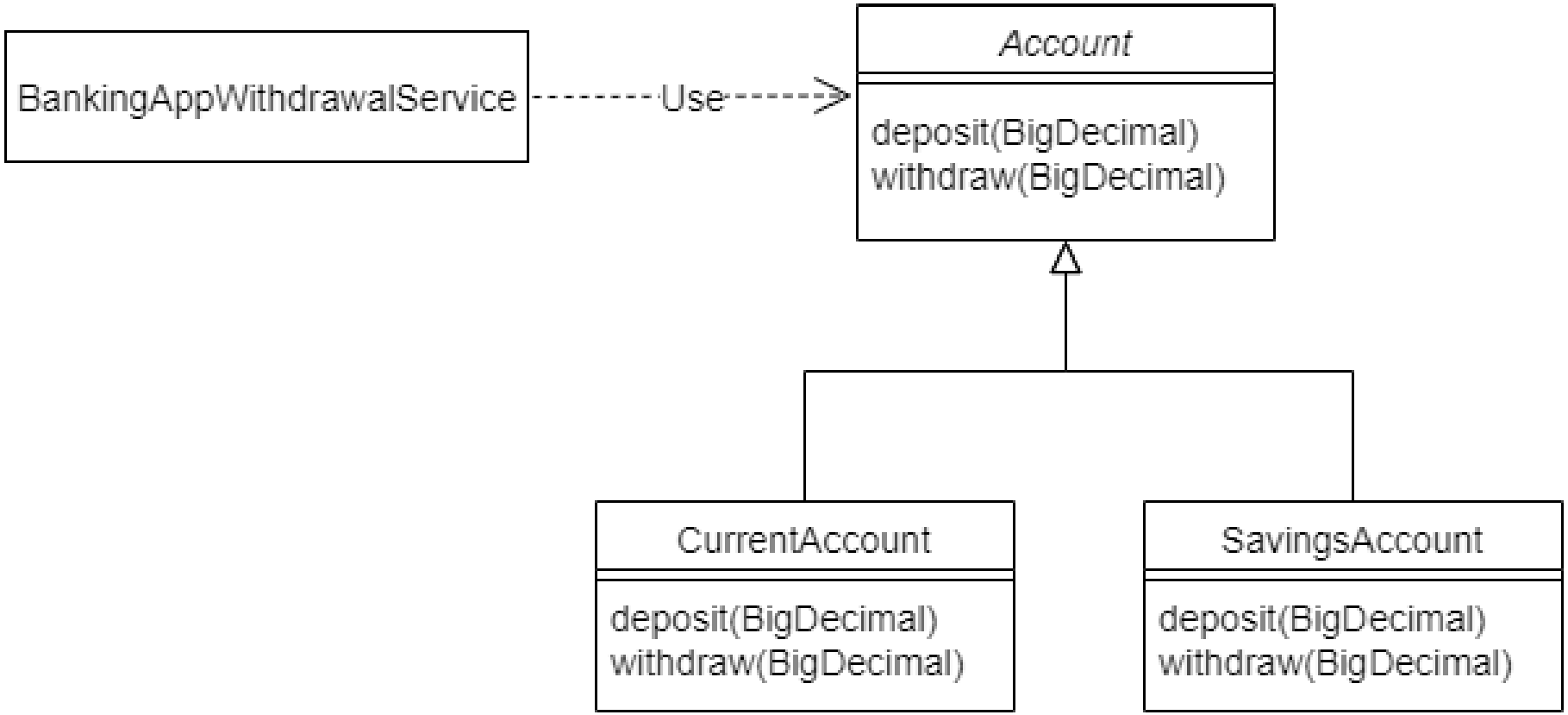
If substituting a superclass object with a subclass object changes the program behavior in unexpected ways, the LSP is violated.

The LSP is applicable when there's a super-type sub-type inheritance relationship by either extending a class or implementing an interface.

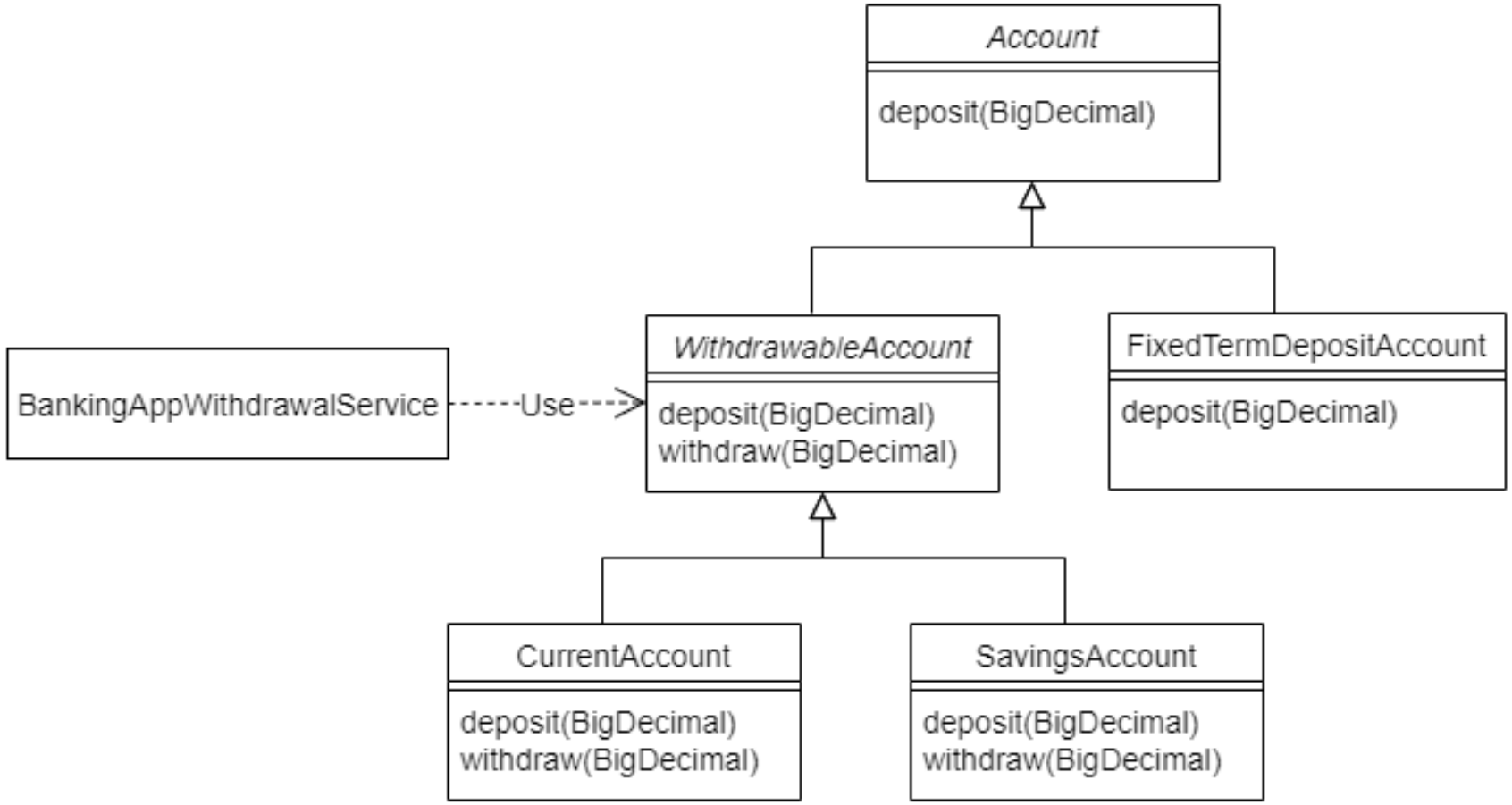
LSP IMPORTANCE

Liskov Substitution Principle actually teaches the exact meaning of inheritance.

EXAMPLE


BankingAppWithdrawalService
Use
Account
deposit(BigDecimal)
withdraw(BigDecimal)
CurrentAccount
deposit(BigDecimal)
withdraw(BigDecimal)
SavingsAccount
deposit(BigDecimal)
withdraw(BigDecimal)

EXAMPLE


Account
deposit(BigDecimal)
WithdrawableAccount
deposit(BigDecimal)
withdraw(BigDecimal)
FixedTermDepositAccount
deposit(BigDecimal)
BankingAppWithdrawalService
Use
CurrentAccount
deposit(BigDecimal)
withdraw(BigDecimal)
SavingsAccount
deposit(BigDecimal)
withdraw(BigDecimal)

INTERFACE SEGREGATION PRINCIPLE

INTERFACE SEGREGATION PRINCIPLE

“Clients should not be forced to depend upon interfaces that they do not use.”

INTERFACE SEGREGATION PRINCIPLE

If you have an abstract class or an interface, then the implementers should not be forced to implement parts that they don’t care about.

In programming, the ISP states that no client should be forced to depend on methods it does not use.

INTERFACE SEGREGATION PRINCIPLE

Don’t add additional functionality to an existing interface by adding new methods.

Instead create a new interface and let your class implement multiple interfaces if needed.

INTERFACE SEGREGATION PRINCIPLE

ISP deals with non cohesive interfaces and it reduces coupling in a system.

ISP (EXAMPLE)

Violation of ISP

```
Interface iPost {
Void CreatPost();
Void ReadPost();
}
```

```
Interface iPostCreate {
Void CreatPost();
}
Interface iPostRead {
Void ReadPost();
}
```

ISP (EXAMPLE)

Violation of ISP

```
Interface ISmartDevice{
void Print();
void Fax();
void Scan();
}
```

```
class AllinOnePrinter implements ISmartDevice{
public void Print() {
    }
public void Fax() {
    }
public void Scan() {
    }
}
```

ISP EXAMPLE

Now suppose we need to handle a new device (EconomicPrinter class) that can only print. We're forced to implement the Whole interface.

```
class EconomicPrinter implements Iprinter{
    public void Print()   {
        }
    public void Fax()   {
    throw new NotSupportedException();
        }
    public void Scan() {
    throw new NotSupportedException();
        }
}
```

SOLUTION

```
interface Iprinter {
void Print();
}
interface Ifax {
void Fax();
}
interface Iscanner{
void Scan();
}
```

```
class EconomicPrinter implements ISmartDevice{
   public void Print()   {
      }
}
```

SOLUTION

```
class EconomicPrinter implements
Iprinter, Ifax, Iscanner{
    public void Print()   {
        }
    public void Fax()   {
        }
    public void Scan() {
        }
}
```

ISP

The ISP guides us to create many small interfaces with coherent functionalities instead of a few big interfaces with lots of different methods.

When we apply the ISP, class and their dependencies communicate using focused interfaces, minimizing dependencies.

Smaller interfaces are easier to implement, improving flexibility and the possibility of reuse.

DEPENDENCY INVERSION PRINCIPLE

DEPENDENCY INVERSION PRINCIPLE

1. High Level Modules should not depend on low level modules. Both should depend on abstractions.

2. Abstractions should not depend upon details. Details should depend upon abstractions.

DEPENDENCY INVERSION PRINCIPLE

In programming., the dependency inversion principle is a way to decouple software modules.

OCP, LSP, and even SRP lead to the dependency inversion principle.

DEPENDENCY INJECTION

Dependency Injection

Injecting any dependencies of a class through a class constructor as an input parameter.

Injection basically converts composition to association.

Strong coupling to weak coupling.

DIP (EXAMPLE)

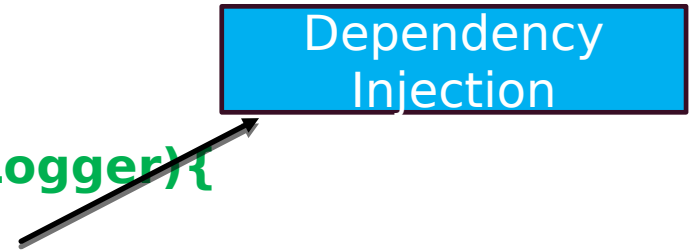
```
Class Post{
Private ErrorLogger errorlLogger = new ErrorLogger();
void CreatePost (Database db, string postMessage) {
try{
        db.Add(postMessage);
        }
catch (Exception ex)
        {
        errorLogger.log(ex.ToString());
        }
   }
}
```

Dependency

```
Class ErrorLogger {
        void log(string error)
        {
        db.LogError("An error occurred:", error);
        File.WriteAllText("\LocalErrors.txt", error);
          }
}
```

SOLUTION

```
Class Post{
Private Logger _logger;
Public Post (Logger injectedLogger){
    logger = injectedLogger;
    }
void CreatePost (Database db, string postMessage) {
        {
        .....
        .....
        }
}
```



SOFTWARE DESIGN PATTERNS

INTRODUCTION TO DESIGN PATTERNS

A good design is more than just knowing and applying OO concepts like abstraction, inheritance, and polymorphism.

Designers focuses on creating flexible designs that are more maintainable and that can cope with changes easily.

HOW PATTERNS ARE USED?

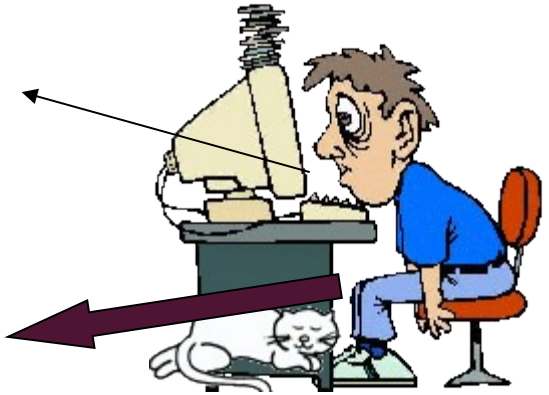
Designer

Design Problem.

Solution.

Implementation details.

Programmer



Reduce gap

Design

Implementation

PATTERN

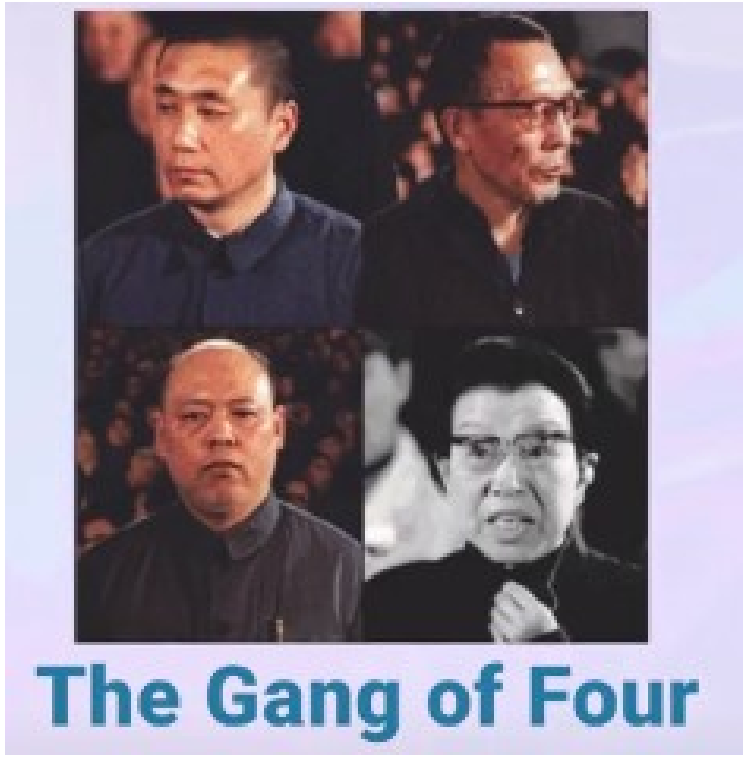
"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice."

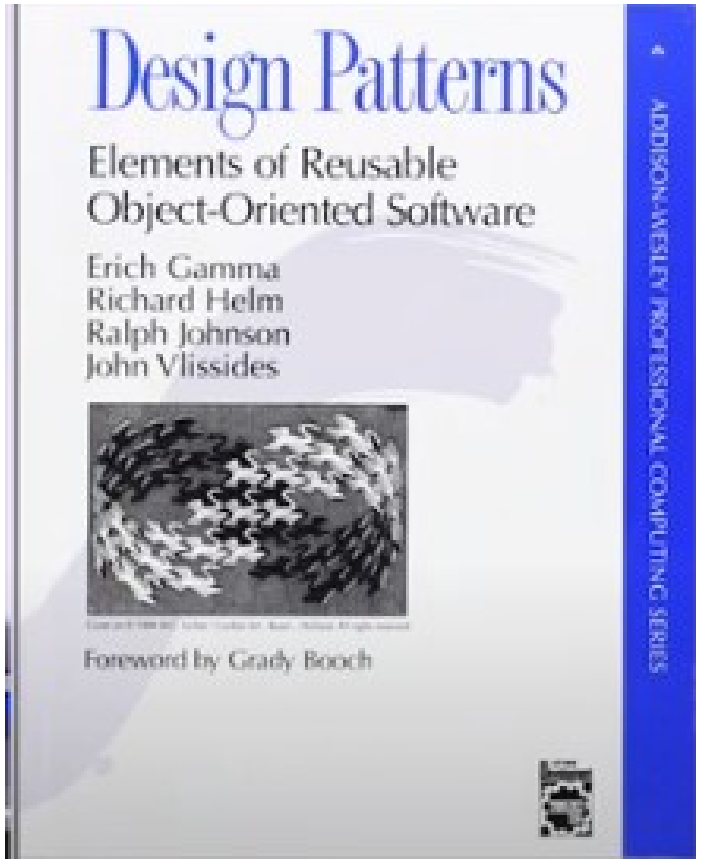
PATTERN

A Pattern must:

- Solve a problem and be useful.
- Have a context and can describe where the solution can be used.
- Recur in relevant situations.
- Provide sufficient understanding to tailor the solution.

GANG OF FOUR


The Gang of Four


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

ELEMENTS OF DESIGN PATTERN

ELEMENTS OF DESIGN PATTERN

Design patterns have four essential elements:

- Pattern name
- Problem
- Solution
- Consequences

DESIGN PATTERN VS FRAMEWORK

Pattern	Framework
Design patterns are recurring solutions to the problems that arise during the life of a software application in a particular context.	A frame work is a group of components that cooperate with each other to provide a reusable architecture.
Primary Goal	Primary Goal
• Improves quality of the software in terms of the software being reusable, maintainable, extensible etc. • Reduces development time	• Improves quality of the software in terms of the software being reusable, maintainable, extensible etc. • Reduces development time

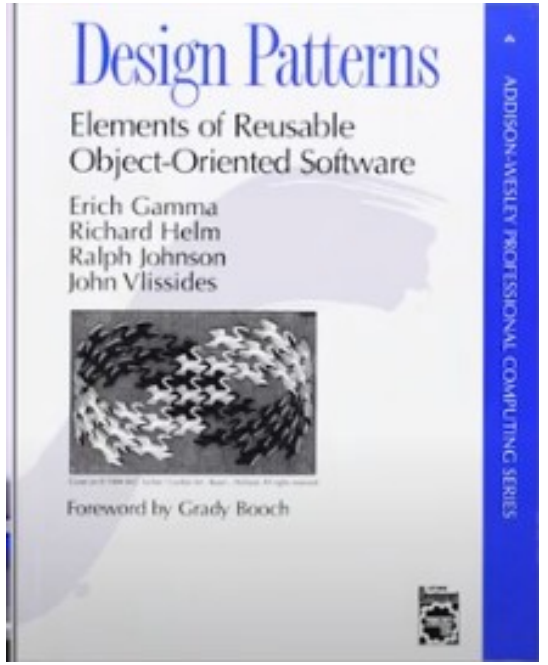
DESIGN PATTERN VS FRAMEWORK

Pattern	Framework
Patterns are logical in nature.	Frameworks are more physical in nature as they exist in the form of software.
Independent of programming language and implementation.	Implementation Specific.
Patterns provide a way to do good design and are used to help design frameworks.	Design patterns may be used in the design and implementation of a framework.

CATEGORIES OF DESIGN PATTERN

CATEGORIES OF DESIGN PATTERN

This book defined 23 patterns in three categories



Creational patterns deal with the process of initializing and configuring of classes and objects (5)

Structural patterns, deal primarily with the static composition and structure of classes and objects (7)

Behavioral patterns, which deal primarily with dynamic interaction among classes and objects (11)

- How they distribute responsibility

GOF PATTERNS

Creational Patterns

- Abstract Factory
- Builder
- **Factory Method**
- Prototype
- **Singleton**

Structural Patterns

- Adapter
- Bridge
- **Composite**
- Decorator
- **Façade**
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- **State**
- **Strategy**
- Template Method
- Visitor

LIMITATIONS OF DESIGN PATTERN

1. *Patterns* do not lead to direct code reuse.
2. Patterns are deceptively simple.
3. Patterns are validated by experienced rather than by automated testing.
4. Integrating patterns into a software development process is a human intensive activity.

SINGLETON DESIGN PATTERN
(CREATIONAL PATTERN)

SINGLETON DESIGN PATTERN

Sometimes there may be a need to have one and only one instance of a given class during the lifetime of an application.

Eg. Database Connection

Singleton Design Pattern ensures that there is only one instance of a class and provides global point of access to it.

CODE FOR SINGLETON DESIGN PATTERN

```
public class Singleton
{
private static Singleton instance;
private Singleton()     // Private Constructor
    {   }
public static Singleton getInstance()
        {
        if (instance == null)
        { instance = new Singleton(); }
    return instance;
         }
}
```

```
public class testsingleton
 {
public static void main (String args[])
      {
     Singleton.getInstance();  // call to static
method
      }
 }
```

SINGLETON DESIGN PATTERN

Problem Statement:

In Chocolate manufacturing industry, there are computer controlled chocolate boilers. The job of boiler is to take in milk and chocolate, bring them to boil and then pass it on to the next phase of chocolate manufacturing process.

We have to make sure that bad things don’t happen like filling the filled boiler or boiling empty boiler or draining out unboiled mixture.

CODE

```
public class ChocolateBoiler {
private boolean empty;
private boolean boiled;
private static ChocolateBoiler uniqueins;
private ChocolateBoiler()
{
empty=true;
boiled=false;
}
```

```
public static ChocolateBoiler getInstance()
{
    if(uniqueins==null)
    {
     uniqueins=new ChocolateBoiler();
     getInstance().fill();
     getInstance().boil();
     getInstance().drain();
    }
 return uniqueins;
}
```

HAVE A GOO DAY!