Software Engineering Design Patterns (1)

Erik Fredericks // frederer@gvsu.edu

Adapted from materials provided by Byron DeVries, Jagadeesh Nandigam

Outline

What is a Design Pattern?
Why study Design Patterns?
Describing Design Patterns

Catalog of Design Patterns

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

Design Pattern Examples

- Creational: Singleton
- Structural: Facade
- Behavioral: Observer

Where can you find more?
Why do Design Patterns really exist?

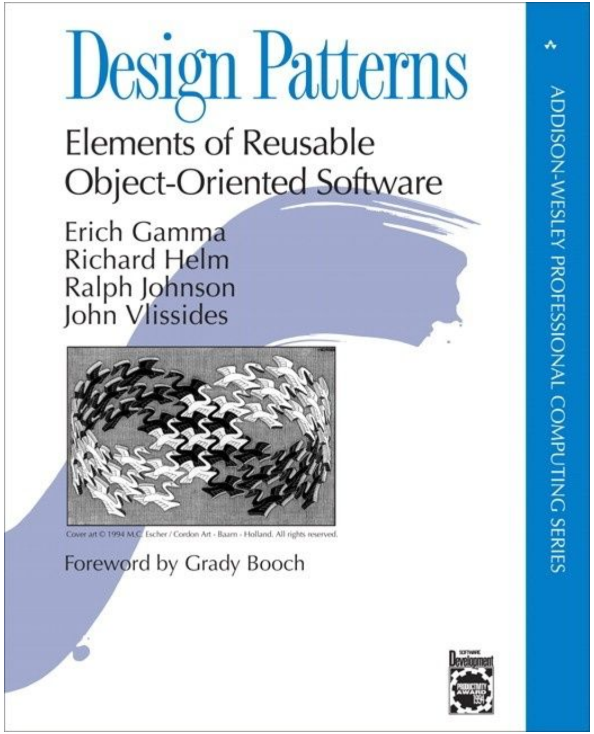
What *is* a design pattern?

A design pattern "…names, abstracts, and identifies the key aspects of a common design structural that make it useful for creating a reusable object-oriented design."*

A design pattern is a **proven** solution to a recurrent problem in a context.

An effective, reusable, proven structure/**communication** solution for a given object-oriented design problem.

What do we mean by proven?
How does communication fit in?


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

*From the book pictured

Why study them (the design patterns)

Reuse existing, high-quality solutions to commonly recurring problems

Establish common terminology to improve communications within teams

- Shifts the level of thinking to a higher perspective.

Improve team communication and individual learning

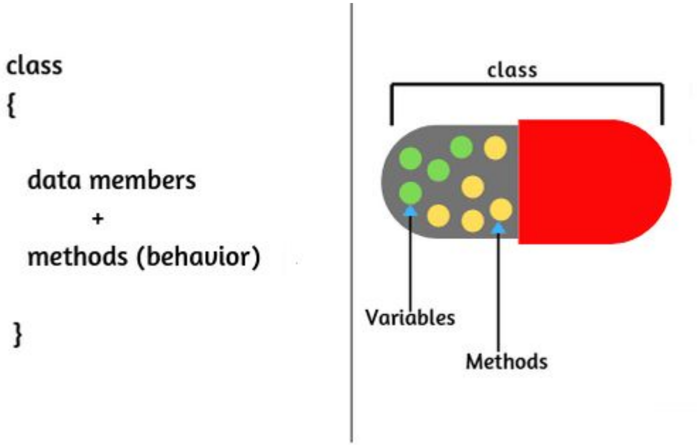
Improve modifiability and maintainability of code

- Design patterns are time-tested solutions (i.e., “proven”)

Why study them (the design patterns)

Adoption of improved object-oriented design strategies

- Encapsulation and information hiding
- Design to interfaces
- Favor composition over inheritance



Describing design patterns

Pattern Name and Classification

Intent: Design issue/problem being addressed

Structure: A graphical representation (in UML) of the classes in the pattern

Participants: Classes participating in the pattern and their responsibilities.

Collaborations: How participants collaborate to carry out their responsibilities

Consequences: Trade-offs and results of using the pattern

Implementation: Techniques, hints, or pitfalls to be aware of when implementing as well as sample code

Catalog of design patterns

The ‘**Gang of Four**’ (Gamma, Helm, Johnson, and Vlissides) did the early / seminal work on design patterns.

Described a structure within which to catalog and describe design patterns.

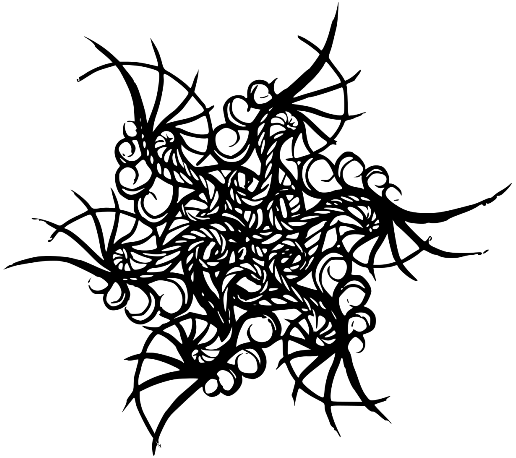
Cataloged 23 patterns using this structure.

Note: The ‘Gang of Four’ did not create these patterns, they identified patterns that already existed in high-quality designs.

Catalog of design patterns

Creational	Structural	Behavioral
Factory Method	Adapter	Chain of Responsibility
Abstract Factory	Bridge	Command
Builder	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	**Facade**	Mediator
	Flyweight	Memento
	Proxy	**Observer**
		State
		Strategy
		Template Method
		Visitor

Catalog of design patterns



Creational Patterns

- Abstract the instantiation process
- Define classes to handle object creation

Structural Patterns

- Concerned with how classes and objects are composed to form larger structures
- Describe ways to compose objects to realize new functionality

Behavioral Patterns

- Concerned with algorithms, flow of control, and assignment of responsibilities between objects
- Describe how a group of objects cooperate to perform a task

Singleton pattern

Pattern Category: **Creational**

Intent:

- Ensure a class only has one instance, and provide a global point of access to it.

Problem addressed:

- Ensuring that a class is instantiated only once, and that the resulting object is readily accessible.

Solution:

- Make the class itself responsible for instantiation and knowing whether it has been instantiated.

What is meant by
“controlled subclasses”?

Singleton pattern

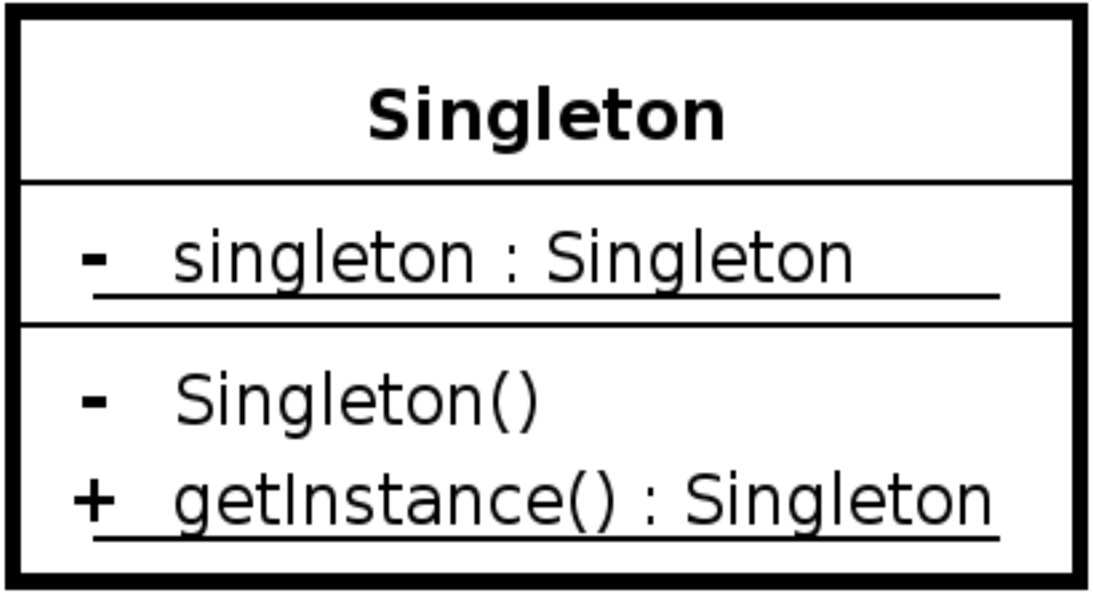
Implementation:

- Add a **private static member** of the class that refers to the desired object (initially, it is null).
- Add a **public static method** that instantiates this class if this member is null (and sets this member’s value) and then returns the value of this member.
- Sets the **constructor’s visibility to private or protected** so that no one can directly instantiate this class (except a child-class if protected) and bypass the static constructor mechanism (i.e., the public static method).

Consequences:

- Controlled access to sole instance.
- Easily adapted to permit a fixed number of instances greater than one.
- Easily extended to provide a family of such **access-controlled subclasses** with different functionality.

Singleton pattern (UML)



Singleton

- singleton : Singleton

- Singleton()

+ getInstance() : Singleton

Singleton pattern: Java example

```java
public class Singleton {

    private static Singleton instance = null;

    private Singleton() { }

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

        return instance;
    }
}
```

Singleton pattern: Java example

```java
public class Singleton {

    private static Singleton instance = null;

    protected Singleton() { }

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

        return instance;
    }
}
```

Private vs. protected?

Private:

- Cannot be accessed/viewed from anywhere **outside class**

Protected:

- Cannot be accessed/viewed from anywhere **outside class**
    - With the exception of child classes (derived classes)

Public:

- Accessible from anywhere within program

Singleton pattern: Java example

```java
public class SingletonChild1 extends Singleton { }

public class SingletonChild2 extends Singleton { }

public class SingletonTest {

    @Test
    public void test() {

        Singleton child1 = SingletonChild1.getInstance();
        Singleton child2 = SingletonChild2.getInstance();

        assertEquals(child1, child2);
    }
}
```

Singleton pattern: Python example

```
class Singleton:
   __instance = None

   @staticmethod
   def getInstance():
      """ Static access method. """
      if Singleton.__instance == None:
         Singleton()
      return Singleton.__instance
```

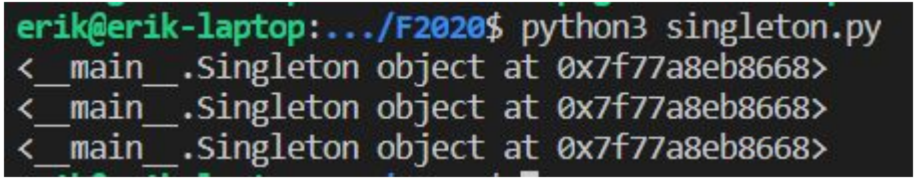
https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_singleton.htm

```
   def __init__(self):
      """ Virtually private constructor. """
      if Singleton.__instance != None:
         raise Exception("This class is a singleton!")
      else:
         Singleton.__instance = self

s = Singleton()
print(s)

s = Singleton.getInstance()
print(s)

s = Singleton.getInstance()
print(s)
```



```
erik@erik-laptop:.../F2020$ python3 singleton.py
<__main__.Singleton object at 0x7f77a8eb8668>
<__main__.Singleton object at 0x7f77a8eb8668>
<__main__.Singleton object at 0x7f77a8eb8668>
```

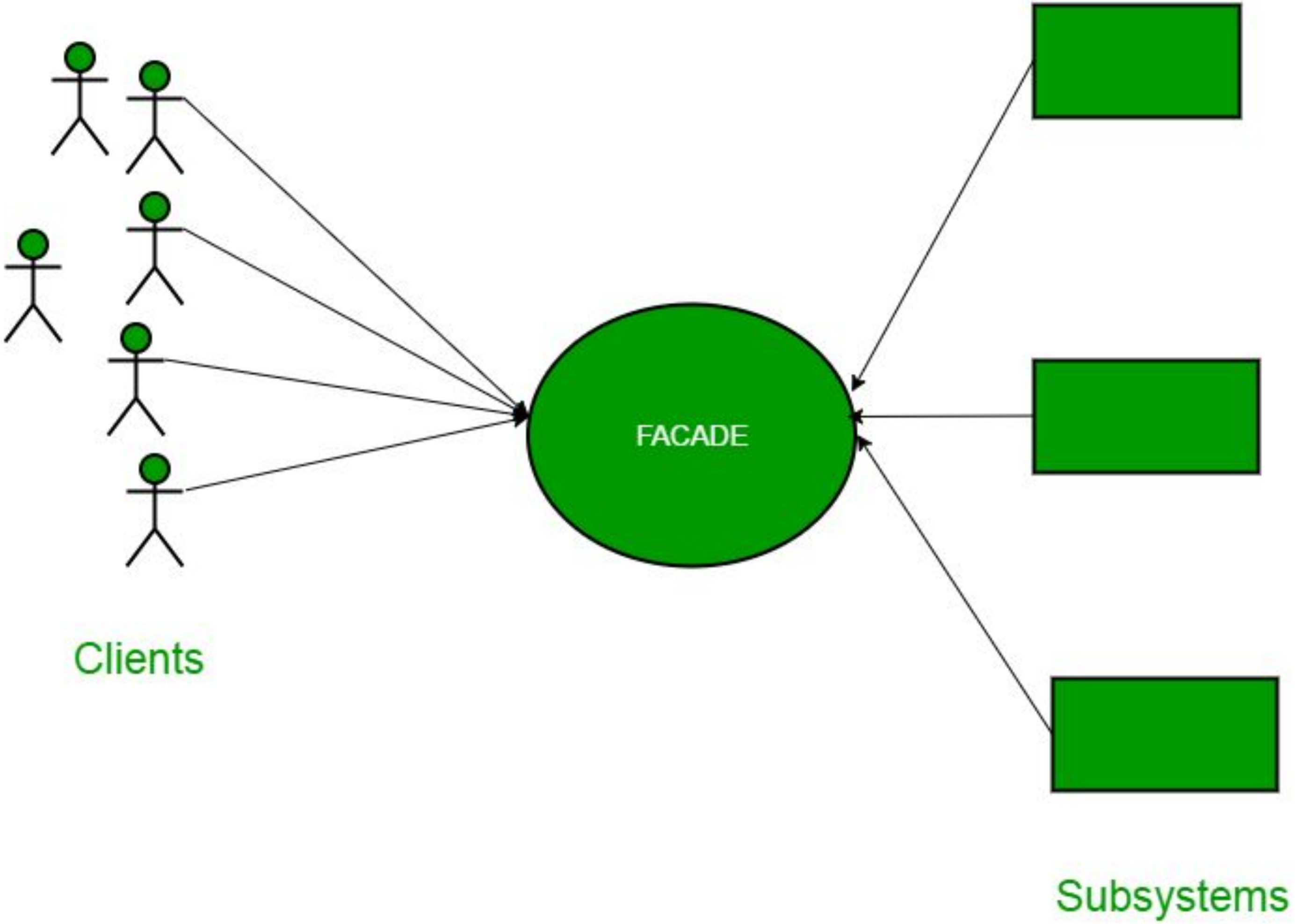
In-class activity

In your **project groups:**

1. Identify what you could use the Singleton Design Pattern for within your project.

2. Be prepared to discuss the Pros and Cons

3. Select a single spokesperson to describe your project and where the Singleton could be used **and describe it briefly to the class.**


FACADE
Clients
Subsystems

Facade pattern

Pattern Category: **Structural**

Intent:

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a unified higher-level interface that makes the subsystems easier to use.

Problem addressed:

- Using design patterns often leads to a complex system of many small components which may be daunting for the casual user. It would be nice if there were a way to provide a simple interface for the basic functionality that is needed most often.

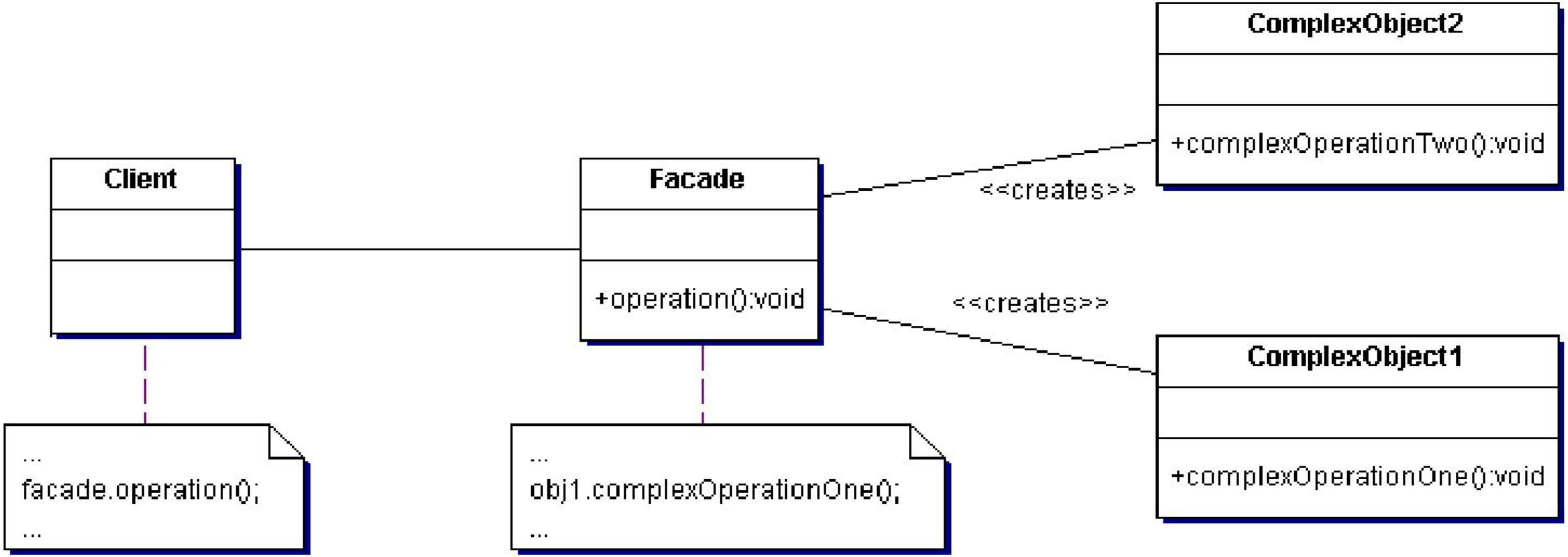
Solution:

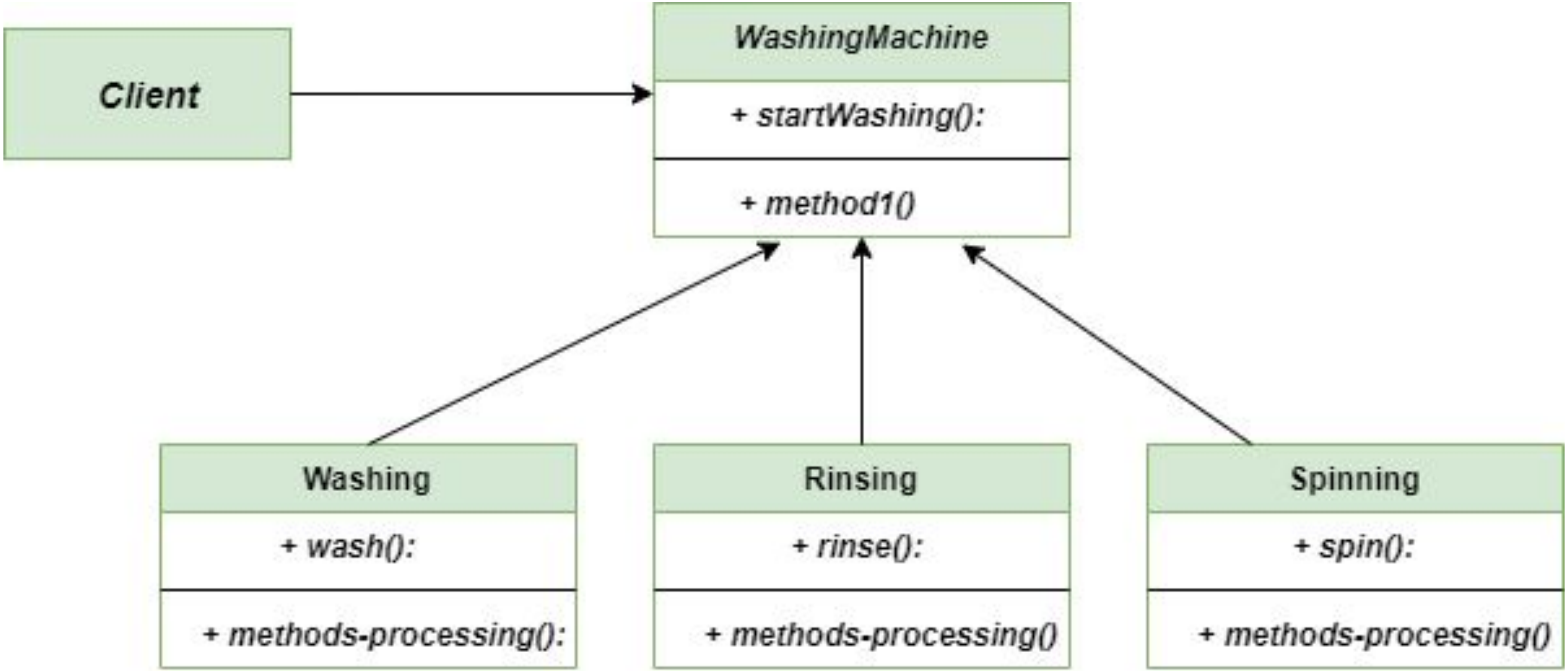
- Create a Facade class that encapsulates the basic functionality of the system by bundling together common operations

When else would a *Facade* class be useful?

When else would a Facade class be useful?

Facade pattern: UML




Client
WashingMachine
+ startWashing():
+ method1()
Washing
+ wash():
+ methods-processing():
Rinsing
+ rinse():
+ methods-processing()
Spinning
+ spin():
+ methods-processing()

Facade pattern

Consequences:

- Simplifies use of the system:
  - Shields users from subsystems components, limiting the number of objects that the user must deal with
- Does not limit “power users”:
  - Serves as a convenience layer for general users, without limiting access to subsystem components for those that desire or need more customizable uses

Facade pattern can be applied when:

- You do not need to use all the functionality of a complex system
- You want to encapsulate or hide the original system
- You want to extend the functionality of the original system
- The cost of writing the new class (i.e., Facade) is less than the cost of everybody learning how to use the original system

Facade pattern (complex parts)

```
class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}
```

Facade pattern:
Facade part

```
class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
        processor.jump(BOOT_ADDRESS);
        processor.execute();
    }
}
```

Facade pattern: Client part

```
class Client {
    public static void main(String[] args) {
     ComputerFacade computer = new ComputerFacade();
     computer.start();
    }
}
```

PYTHON

```
"""Facade pattern with an example of
WashingMachine"""
class Washing:
    '''Subsystem # 1'''
    def wash(self):
        print("Washing...")


class Rinsing:
    '''Subsystem # 2'''
    def rinse(self):
        print("Rinsing...")

class Spinning:
    '''Subsystem # 3'''
    def spin(self):
        print("Spinning...")
```

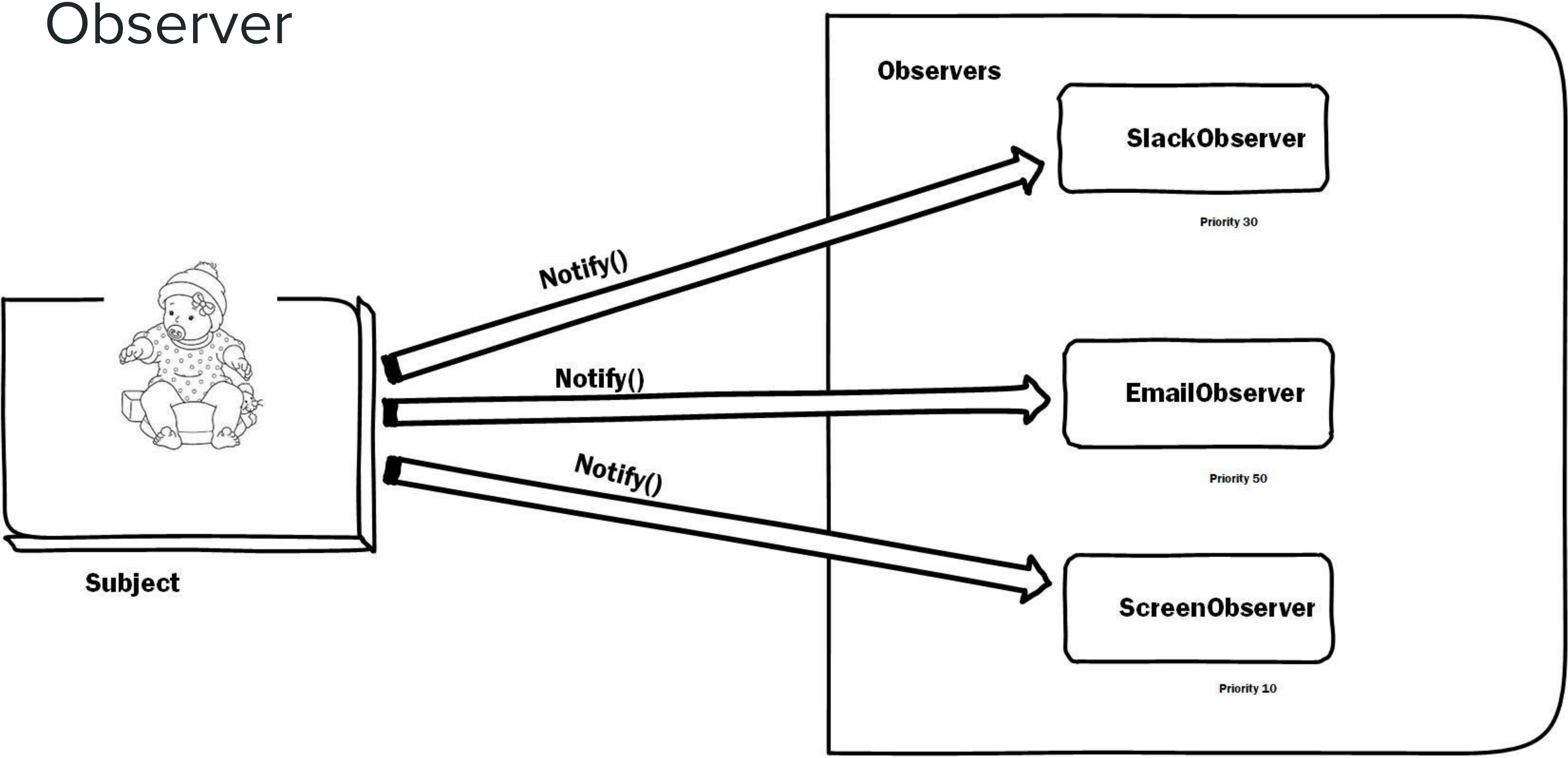
```
class WashingMachine:
    '''Facade'''
    def __init__(self):
        self.washing = Washing()
        self.rinsing = Rinsing()
        self.spinning = Spinning()

    def startWashing(self):
        self.washing.wash()
        self.rinsing.rinse()
        self.spinning.spin()
```

```
""" main method """
if __name__ == "__main__":
    washingMachine = WashingMachine()
    washingMachine.startWashing()
```

How might you use a Facade pattern in your term projects?

Observer


Observers
SlackObserver
Priority 30
Notify()
Notify()
EmailObserver
Notify()
Priority 50
ScreenObserver
Priority 10
Subject

Observer pattern

Pattern Category: **Behavioral**

Intent:

- Define a one-to-many dependency between objects so that when one object (subject) changes state, all its dependencies (observers) are notified and updated automatically.

Problem:

- You need to notify a **varying** list of objects that an event has occurred.

Solution:

- Observers delegate the responsibility for monitoring for an event to a central object – the subject itself

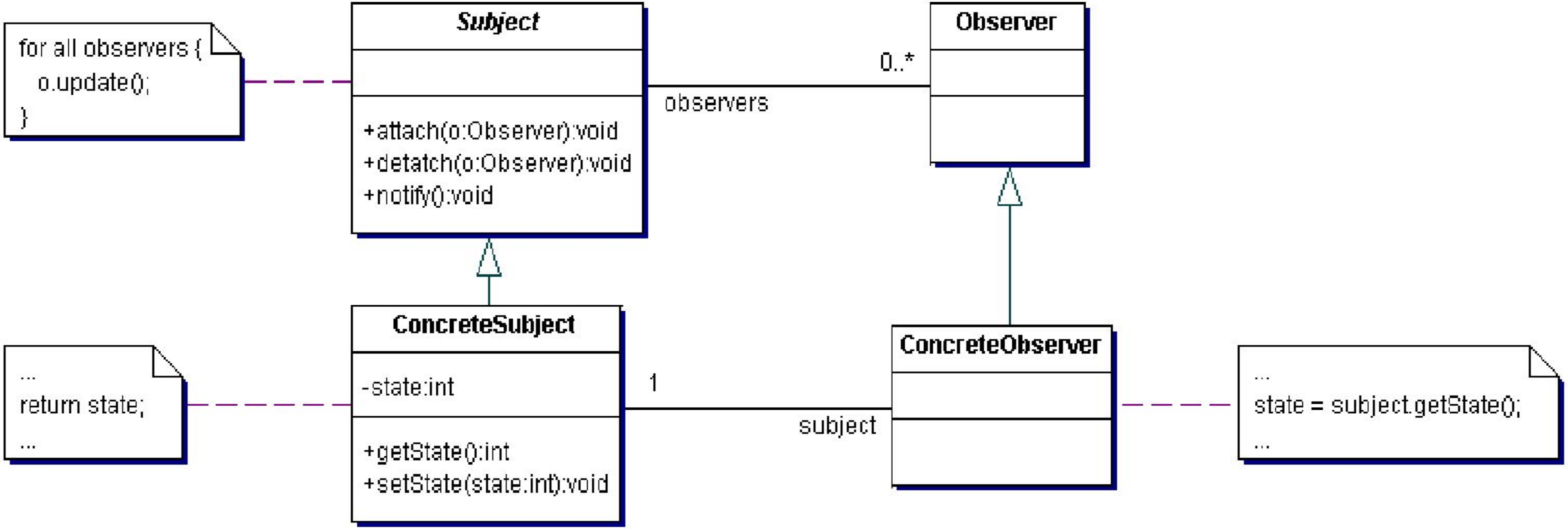
Observer pattern

Use the Observer pattern when the:

- List of objects that need to be notified of an event changes or is somehow conditional

- Subject cannot anticipate every object that might need to know about the event

Observer pattern: UML



Observer pattern: Implementation

Implementation:

- Step 1: Make the observers behave in the same way (i.e., implement the same interface)
    - `update()`

- Step 2: Observers are responsible for knowing what they are to watch for (i.e., register themselves with the subject)
    - `attach(observer)`
    - `detach(observer)`

Observer pattern: Implementation

- Step 3: The subject notifies the observers when the event occurs
    - `notify()`

- Step 4: Observers get the information from the subject

The Observer pattern aids flexibility and keeps things decoupled.

Observer pattern: Java

Support for observer pattern in Java:

```
java.util.Observable class
java.util.Observer interface

public class Observable {
  public void addObserver(Observer o);
  public void deleteObserver(Observer o);
  public void notifyObservers();
  public void notifyObservers(Object arg);
  public void deleteObservers();
  protected void setChanged();
  protected void clearChanged();
  public boolean hasChanged();
  public int countObservers()
}
```

Observer pattern: Java

Support for observer pattern in Java:

```
java.util.Observable class
java.util.Observer interface

public interface Observer {
  void update(Observable o, Object arg);
}
```

When would we use the Observer Pattern?

Observer pattern: Observable

```
import java.util.Observable;
import java.util.Scanner;

public class EventSource extends Observable implements Runnable {

    @Override
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
              setChanged();
            notifyObservers(response);
        }
    }
}
```

Observer pattern: Observer

```java
import java.util.Observable;
import java.util.Observer;

public class MyApp {

    public static void main(String[] args) {
        System.out.println("Enter Text: ");
        EventSource eventSource = new EventSource();

        eventSource.addObserver(new Observer() {
            @Override
            public void update(Observable obj, Object arg) {
                System.out.println("Received response: " + arg);
            }
        });

        new Thread(eventSource).start();
    }
}
```

Observer pattern: Python

```
import threading
import time
import pdb

class Downloader(threading.Thread):
  def run(self):
    print('downloading')
    for i in range(1,5):
      self.i = i
      time.sleep(2)

    print('unfunf')
    return 'hello world'
```

https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_observer.htm

```
class Worker(threading.Thread):
   def run(self):
      for i in range(1,5):
         print('worker running: %i (%i)' % (i, t.i))
         time.sleep(1)
         t.join()

         print('done')

t = Downloader()
t.start()

time.sleep(1)

t1 = Worker()
t1.start()

t2 = Worker()
t2.start()

t3 = Worker()
t3.start()
```

How might you use an Observer in your term projects?

Where can you find more?

https://en.wikipedia.org/wiki/Software_design_pattern

https://www.tutorialspoint.com/python_design_patterns/index.htm