Design Patterns

Lecture Notes

Software Design (SE324)

(Prepared by Dr. Khaldoon Al-Zoubi)

Design Patterns

Creational design patterns are the ones that attempt to efficiently manage the creation or creational process of objects in a software system

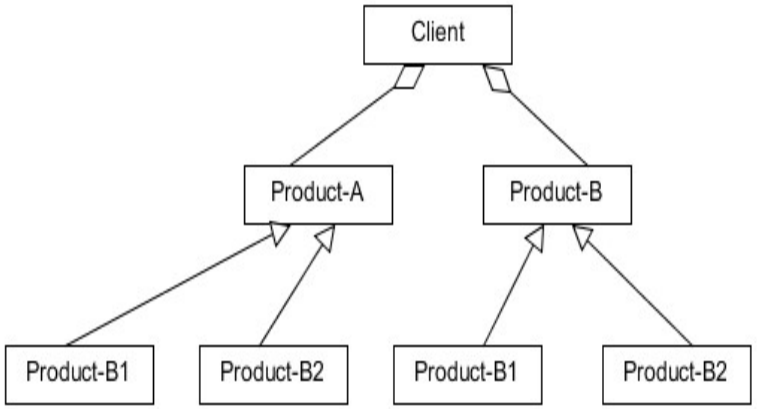
Structural design patterns are the ones that attempt to construct larger structures from the composition of existing classes, objects, or other structures.

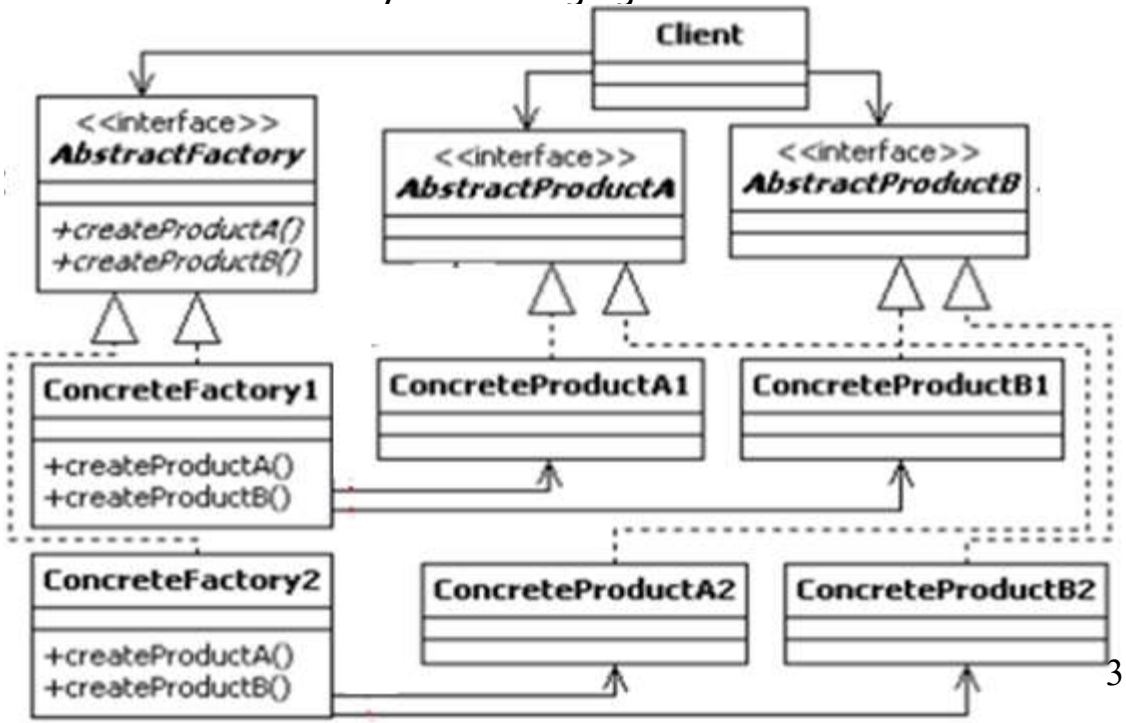
Behavioral design patterns are used to manage operations, relationships, and responsibilities between objects.

C	Abstract Factory	S	Decorator	C	Prototype
S	Adapter	S	Facade	S	Proxy
S	Bridge	C	Factory Method	B	Observer
C	Builder	S	Flyweight	C	Singleton
B	Chain of Responsibility	B	Interpreter	B	State
		B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor

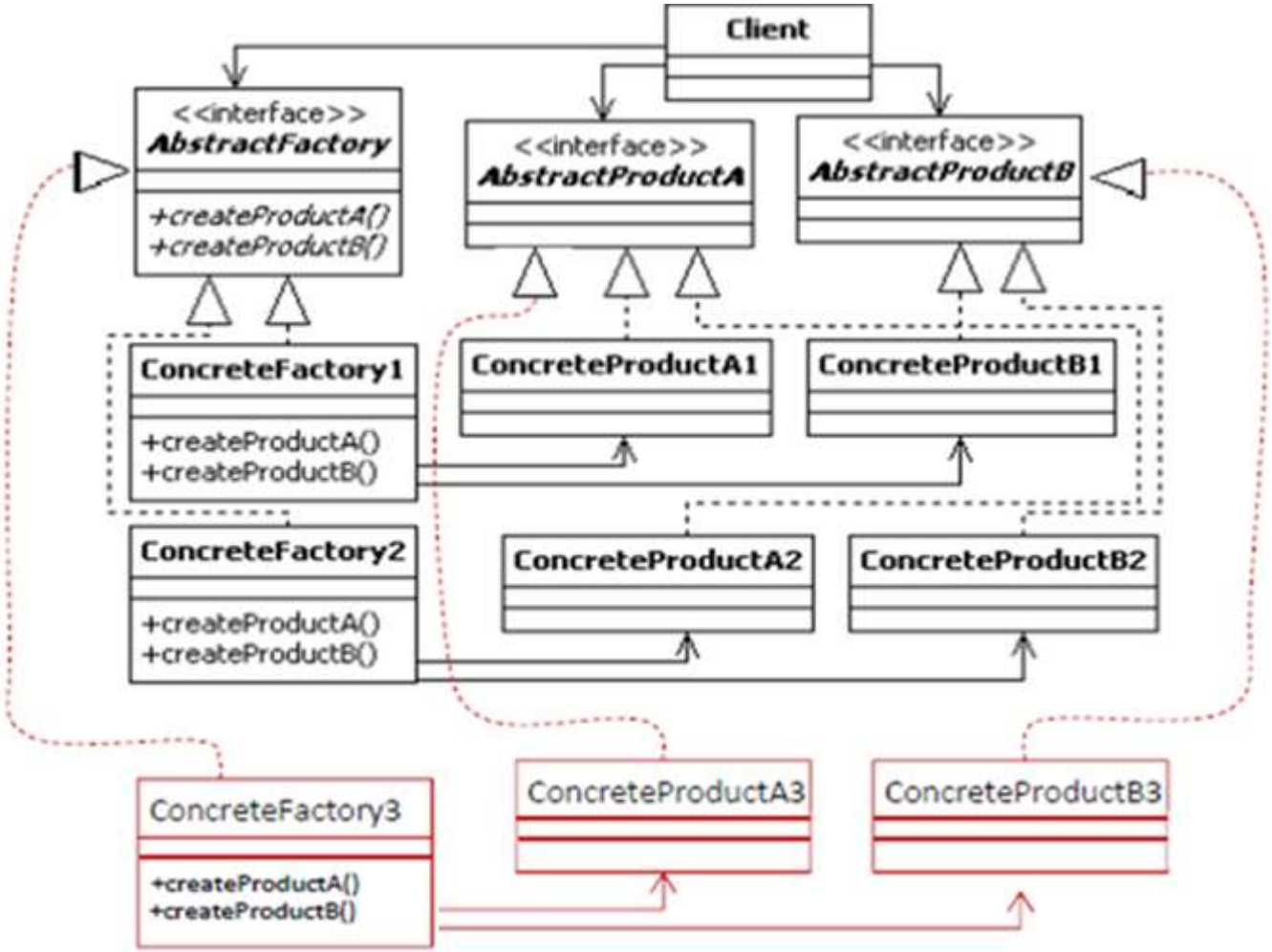
Abstract Factory (Creational Pattern)

- The abstract factory is an object creational design pattern intended to manage and encapsulate the creation of a set of objects that conceptually belong together and that represent a specific family of products.
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- In the abstract factory pattern, the terms *family of products* or *family of objects* are used to denote a group of objects that belong together and therefore must be created together. When designing software that uses a group of objects that need to be created and used together, problems can arise when there is no consistent way for managing the creation of these objects.

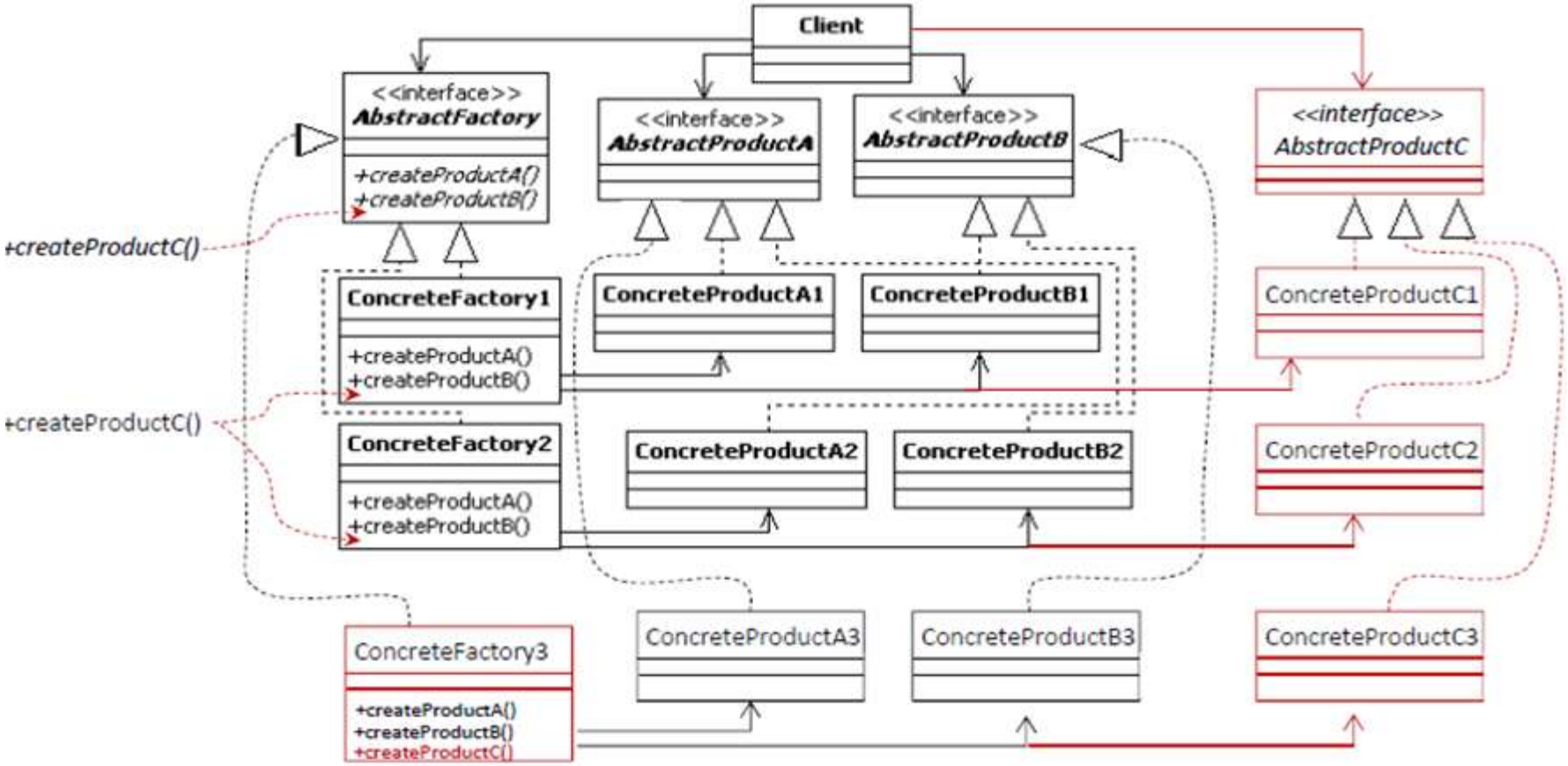




Abstract Factory
Adding a new family of products



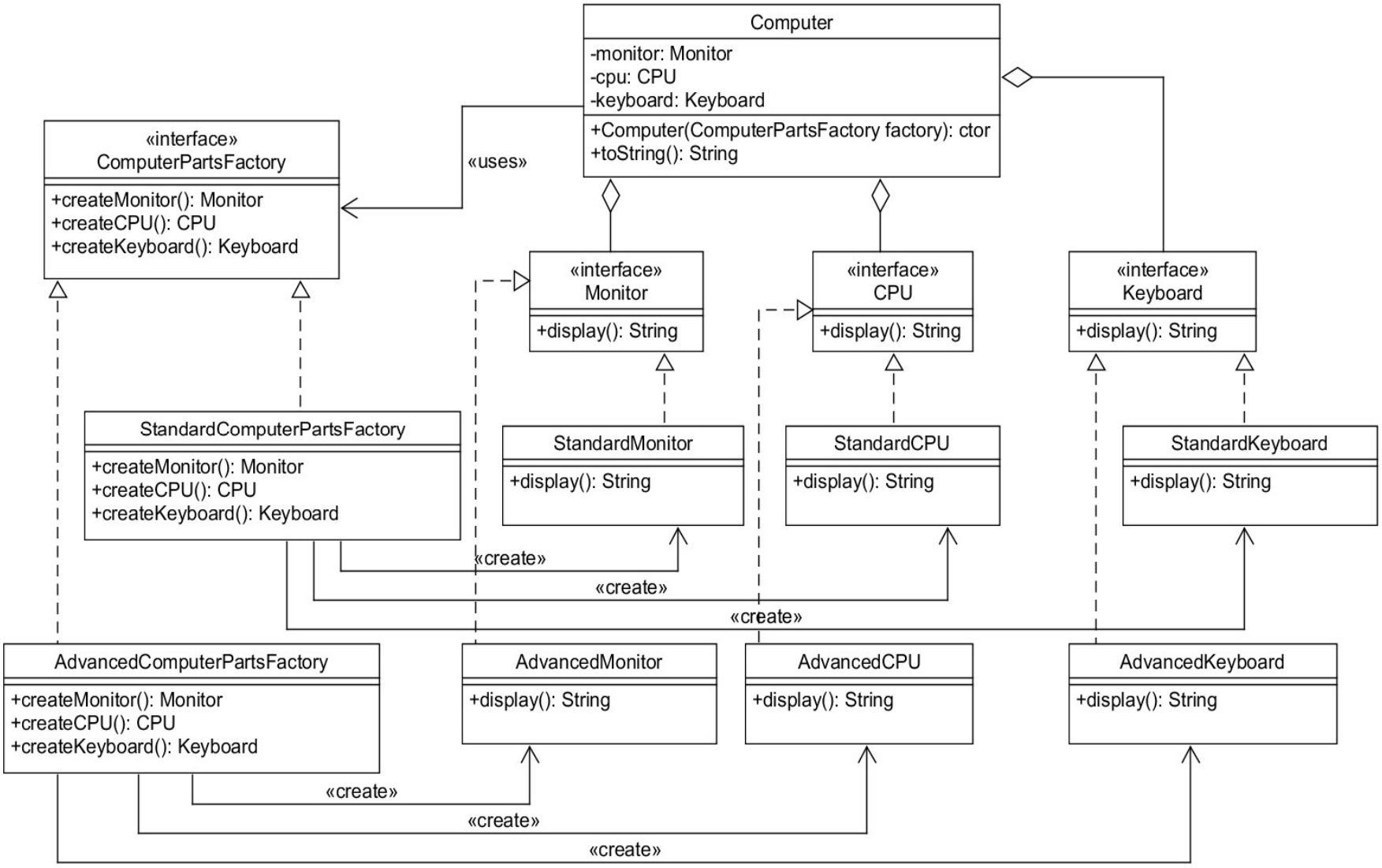
Abstract Factory
(Adding new Product)



Abstract Factory Computer Store Example

- Consider a software system for a computer store, where the store carries only two types of computers for sale:
  - Top of the line computer, we'll call these advanced computers
  - Inexpensive computers, we'll call these standard computers
  - Obviously, a computer store will need to carry more computers in the future!
- Advanced computers are made up of "advanced computer products," e.g. the latest multi-core CPU, wireless keyboard, advanced monitor (e.g., widescreen large 3D), advanced graphics & sound card, etc.
- Standard computers are made up of "standard computer products," e.g., single core CPU, wired keyboard, small screen monitor, low-grade graphics and sound, etc.
- The system is designed so that it searches remote information sources, for product information, such as:
  - Product reviews
  - Customer's & Manufacturers' comments

Abstract Factory
Computer Store Example



Abstract Factory
Computer Store Example

```
public interface CPU {
    public String display();
}
```

```
public interface Keyboard {
    public String display();
}
```

```
public interface Monitor {
    public String display();
}
```

```
public class StandardCPU implements CPU {

    @Override
    public String display() {
        return "Standard CPU";
    }
}
```

```
public class AdvancedCPU implements CPU {

    @Override
    public String display() {
        return "Advanced CPU";
    }
}
```

```
public class StandardKeyboard implements Keyboard {

    @Override
    public String display() {
        return "Standard Keyboard";
    }
}
```

```
public class AdvancedKeyboard implements Keyboard {

    @Override
    public String display() {
        return "Advanced Keyboard";
    }
}
```

```
public class StandardMonitor implements Monitor {

    @Override
    public String display() {
        return "Standard Monitor";
    }
}
```

```
public class AdvancedMonitor implements Monitor {

    @Override
    public String display() {
        return "Advanced Monitor";
    }
}
```

Abstract Factory
Computer Store Example

```
public interface ComputerPartsFactory {

    public Monitor createMonitor();

    public CPU createCPU();

    public Keyboard createKeyboard();
}
```

```
public class StandardComputerPartsFactory
                implements ComputerPartsFactory {

    @Override
    public Monitor createMonitor() {
        return new StandardMonitor();
    }

    @Override
    public CPU createCPU() {
        return new StandardCPU();
    }

    @Override
    public Keyboard createKeyboard() {
        return new StandardKeyboard();
    }
}
```

```
public class AdvancedComputerPartsFactory
                implements ComputerPartsFactory {

    @Override
    public Monitor createMonitor() {
        return new AdvancedMonitor();
    }

    @Override
    public CPU createCPU() {
        return new AdvancedCPU();
    }

    @Override
    public Keyboard createKeyboard() {
        return new AdvancedKeyboard();
    }
}
```

Abstract Factory
Computer Store Example

```java
public class Computer {

    private Monitor monitor;
    private CPU cpu;
    private Keyboard keyboard;

    public Computer(ComputerPartsFactory factory) {
        monitor = factory.createMonitor();
        cpu = factory.createCPU();
        keyboard = factory.createKeyboard();
    }

    public String getMonitorInfo() {
        return monitor.display();
    }

    public String getCpuInfo() {
        return cpu.display();
    }

    public String getKeyboard() {
        return keyboard.display();
    }

    @Override
    public String toString() {
        return monitor.display() + "," +
                cpu.display() + "," +
                   keyboard.display();
    }
}
```

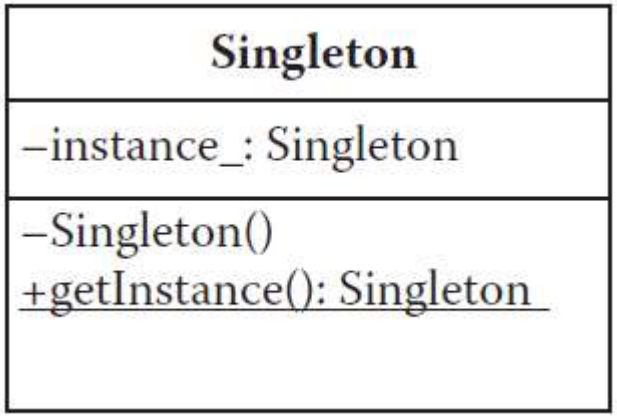
```java
public class Main {

    public static void main(String[] args) {

        // Standard Computer
        Computer computer = new Computer(new StandardComputerPartsFactory());
        System.out.println("Standard Computer: " + computer);

        // Advanced Computer
        computer = new Computer(new AdvancedComputerPartsFactory());
        System.out.println("Advanced Computer: " + computer);
    }
}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Main (2) [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (
Standard Computer: Standard Monitor,Standard CPU,Standard Keyboard
Advanced Computer: Advanced Monitor,Advanced CPU,Advanced Keyboard
```

Singleton design pattern (Creational Pattern)

The **singleton** design pattern is an object **creational design pattern** used to prevent objects from being instantiated more than once in a running program. It is meant to provide a design solution that enforces the conceptual representation of entities that must be singular within the problem domain. For example, consider the simulation of an operating system. In the operating systems domain, the simulation software may contain types (i.e., classes) for processes, threads, and so forth that can be instantiated multiple times to properly represent running programs within the simulation. In these cases, the multiple instantiation of these types is conceptually consistent with the problem domain. However, consider an entity type for singular items, such as the file system. In this case, for a typical operating system, it would be conceptually incorrect to have multiple file system object instances running within the simulation. Furthermore, if the simulation were to create two file systems by mistake, the results would not be reliable. In cases such as this one, the singleton design pattern can be used to enforce the policy that only one instance of the file system object is running at all times.

How can we ensure always one instance of an object?



Singleton design pattern

Example (Log Events):

Consider an application that requires event-logging capabilities. The application consists of many different objects that generate events to keep track of their actions, status of operations, errors, or any other information of interest.

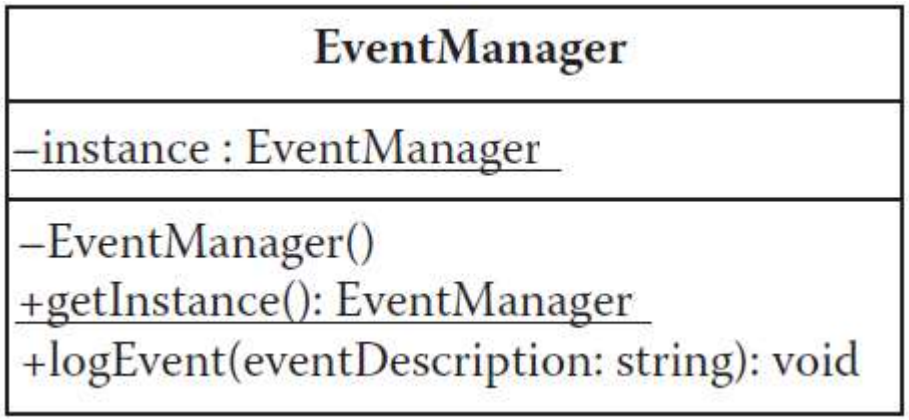
```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class EventManager {

    private static DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

    public void logEvent(String eventDescription) {
        System.out.println(dtf.format(LocalDateTime.now()) + " " + eventDescription);
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        EventManager ev = new EventManager();
        ev.logEvent("Hello");
    }
}
```

Singleton design pattern

Example (Log Events):

A decision is made to create an event manager that can be accessed by all objects and used to manage all events in the system. Upon instantiation, the event manager creates an event list that gets updated as events are logged. At specific points during the software system's operation, these events are written to a file. To prevent conflicts, it is desirable that at any given time there is only one instance of the event manager executing.



Singleton design pattern

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class EventManager {
    private static DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
    private static EventManager instance;

    private EventManager() {

    }

    public void logEvent(String eventDescription) {
        System.out.println(dtf.format(LocalDateTime.now()) + " " + eventDescription);
    }

    public static EventManager getInstance() {
        if (instance == null) {
            instance = new EventManager();
        }
        return instance;
    }
}
```

```java
public class Main {

    public static void main(String[] args) {
        EventManager.getInstance().logEvent("Hello");
    }
}
```

Is thread safe?

Singleton design pattern

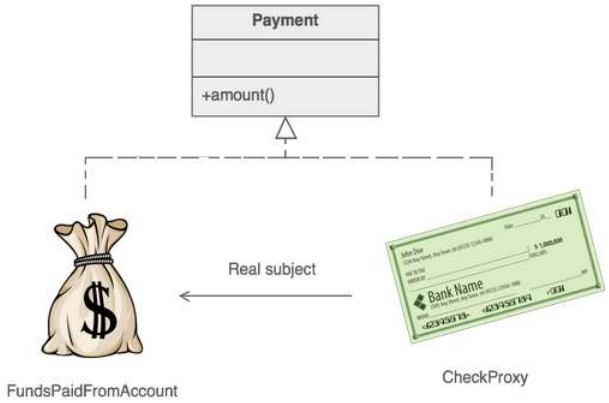
```java
public class EventManager {
    private static DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
    private static EventManager instance;
    private static Object lock = new Object();

    private EventManager() {

    }

    public void logEvent(String eventDescription) {
        System.out.println(dtf.format(LocalDateTime.now()) + " " + eventDescription);
    }

    public static EventManager getInstance() {
        if (instance == null) {
            synchronized (lock) {
                if (instance == null) {
                    instance = new EventManager();
                }
            }
        }
        return instance;
    }
}
```

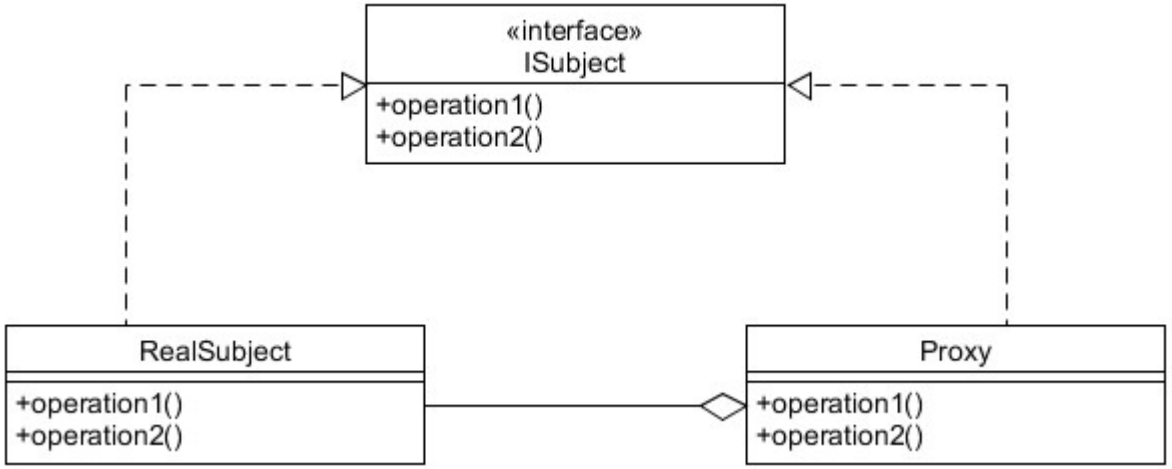
```java
public class Main {

    public static void main(String[] args) {
        EventManager.getInstance().logEvent("Hello");
    }
}
```

Is thread safe?

Proxy Pattern (Structural)

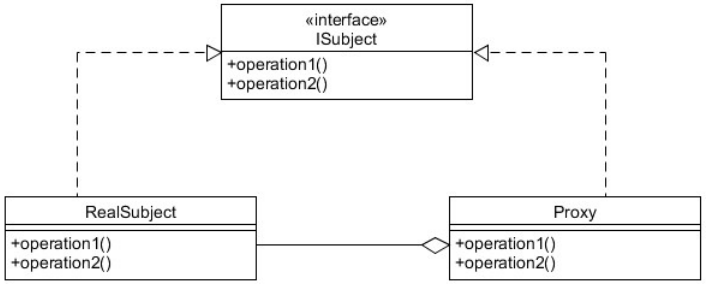
- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from unnecessary complexity.
- Applicable situations.
  - A virtual proxy is a placeholder for "expensive to create" objects. → create on first use
  - A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC provides.
  - A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.

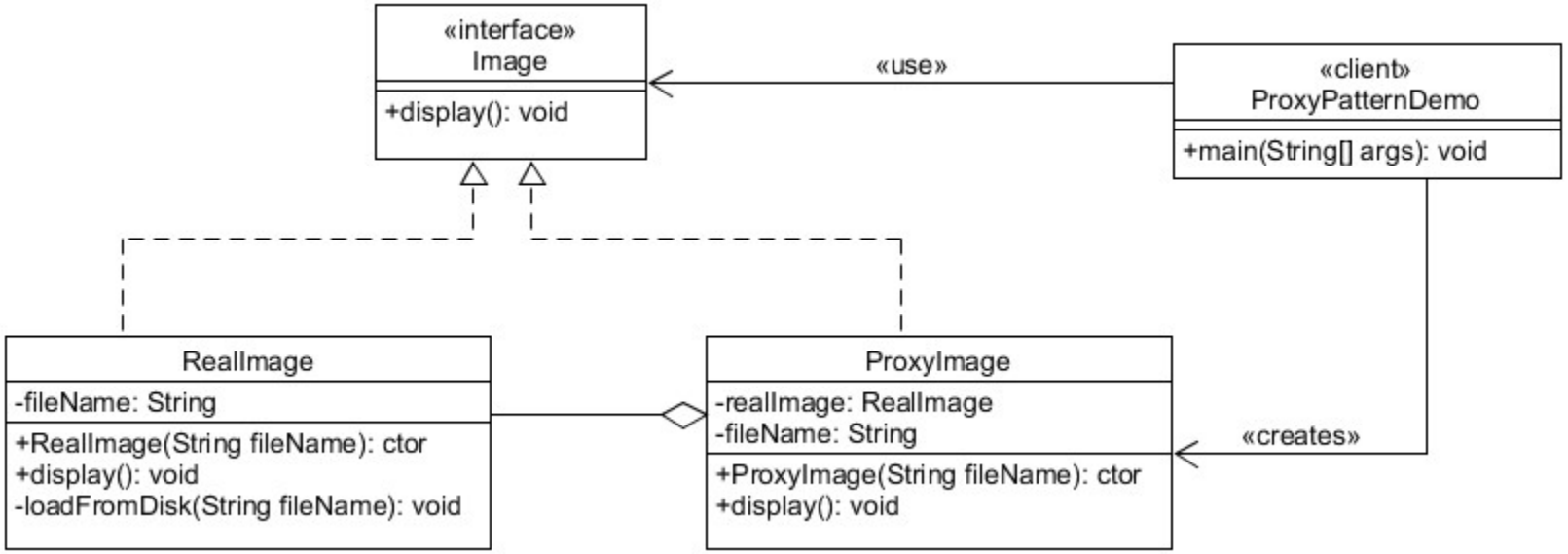




Proxy Pattern (Structural)

- We need to create proxy to display Real images





Proxy Pattern (Structural)

```java
public interface Image {
    public void display();
}
```

```java
public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName) {
        System.out.println("-->RealImage: Constructor");
        this.fileName = fileName;
        loadFromDisk(fileName);
    }

    @Override
    public void display() {
        System.out.println("-->RealImage: display()");
        System.out.println("Displaying " + fileName);
    }

    private void loadFromDisk(String fileName) {
        System.out.println("Loading " + fileName);
    }
}
```

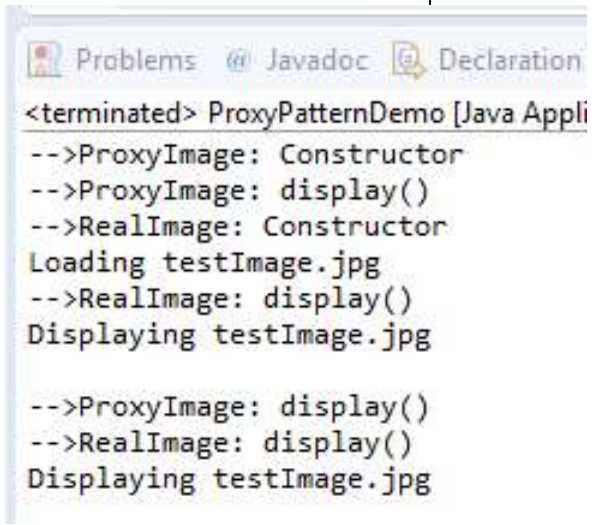
Proxy Pattern (Structural)

```java
public interface Image {
    public void display();
}
```

```java
public class ProxyImage implements Image {

    private RealImage realImage;
    private String fileName;

    public ProxyImage(String fileName) {
        System.out.println("-->ProxyImage: Constructor");
        this.fileName = fileName;
    }

    @Override
    public void display() {
        System.out.println("-->ProxyImage: display()");
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```

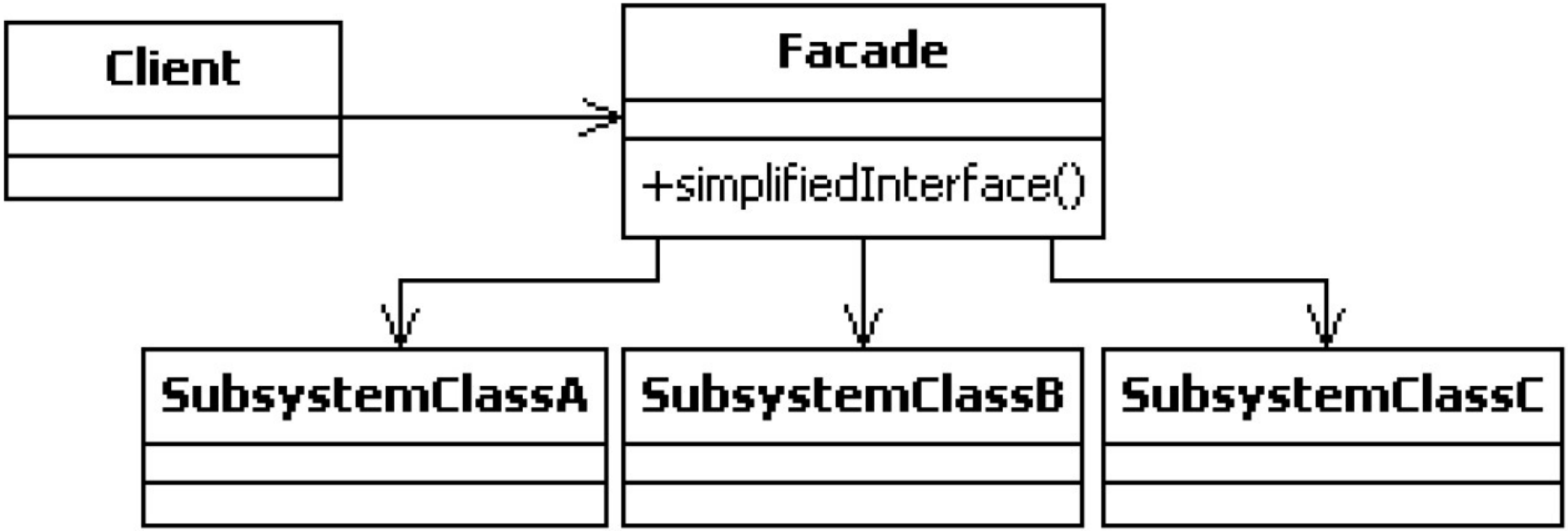
Proxy Pattern (Structural)

```
public class ProxyPatternDemo {

    public static void main(String[] args) {
        Image image = new ProxyImage("testImage.jpg");

        // image will be loaded from disk
        image.display();
        System.out.println("");

        // image will not be loaded from disk
        image.display();
    }
}
```



Facade Pattern (Structural)

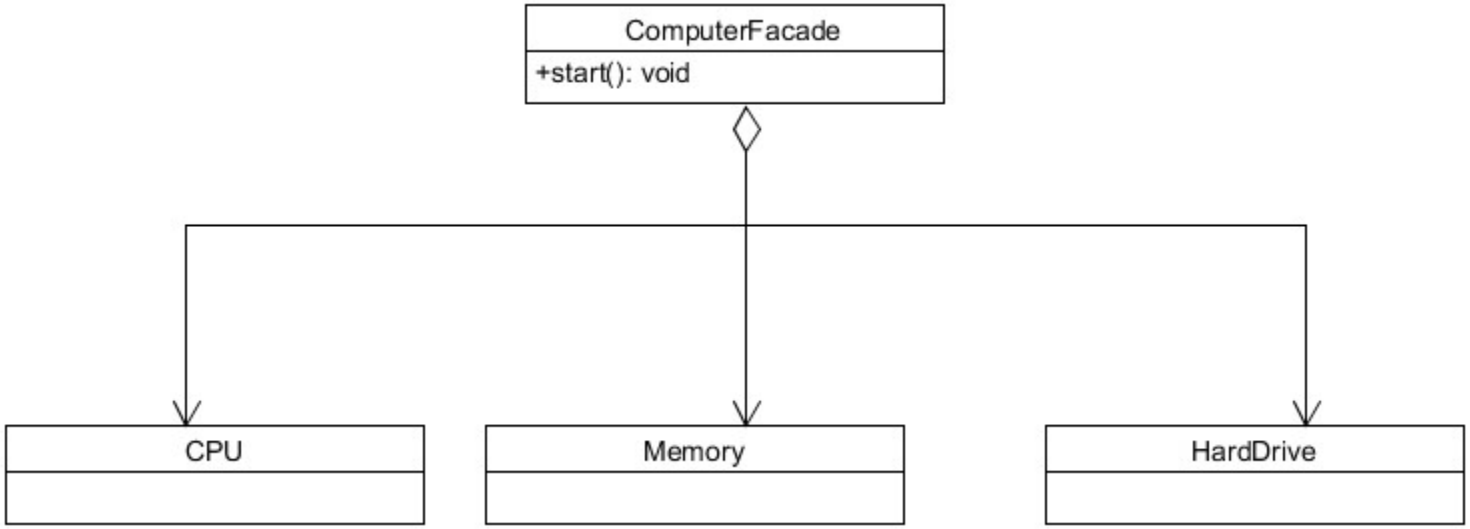
- The facade design pattern is an object structural pattern that provides a simplified interface to complex subsystems. By providing a simplified interface, the facade design pattern provides a higher level of abstraction that liberates clients from the responsibility of knowing the internal structure of various elements of the subsystem, which in turn reduces coupling and simplifies client code. Facade also shields clients from changes that occur in the subsystem; by having a standardized facade interface, the internal structure of the subsystem can vary without affecting clients.



Facade Pattern Example

```
/* Complex parts */

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

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

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

Facade Pattern Example



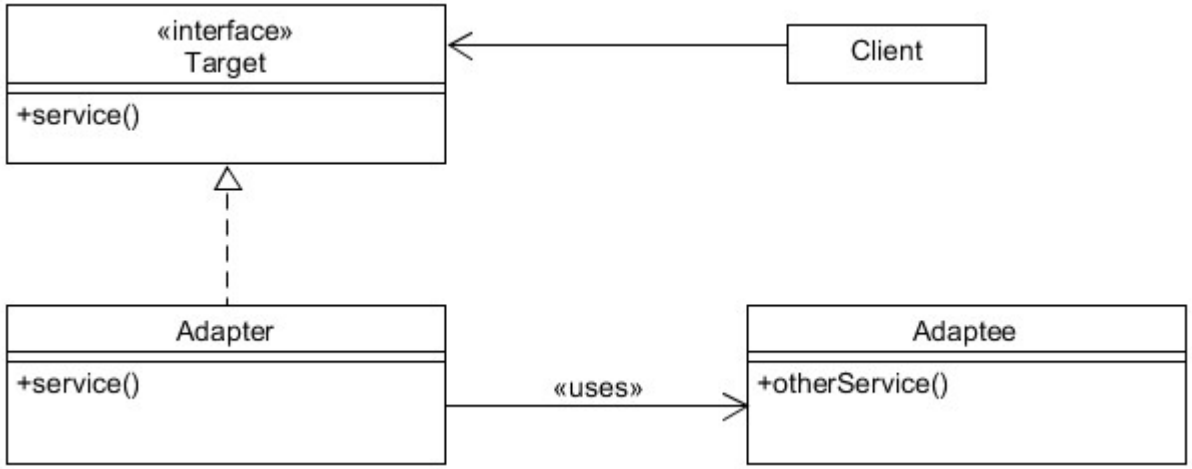
Facade Pattern Example

```java
/* Facade */

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();
    }
}
```

```java
/* Client */

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

Adapter Pattern (Structural)

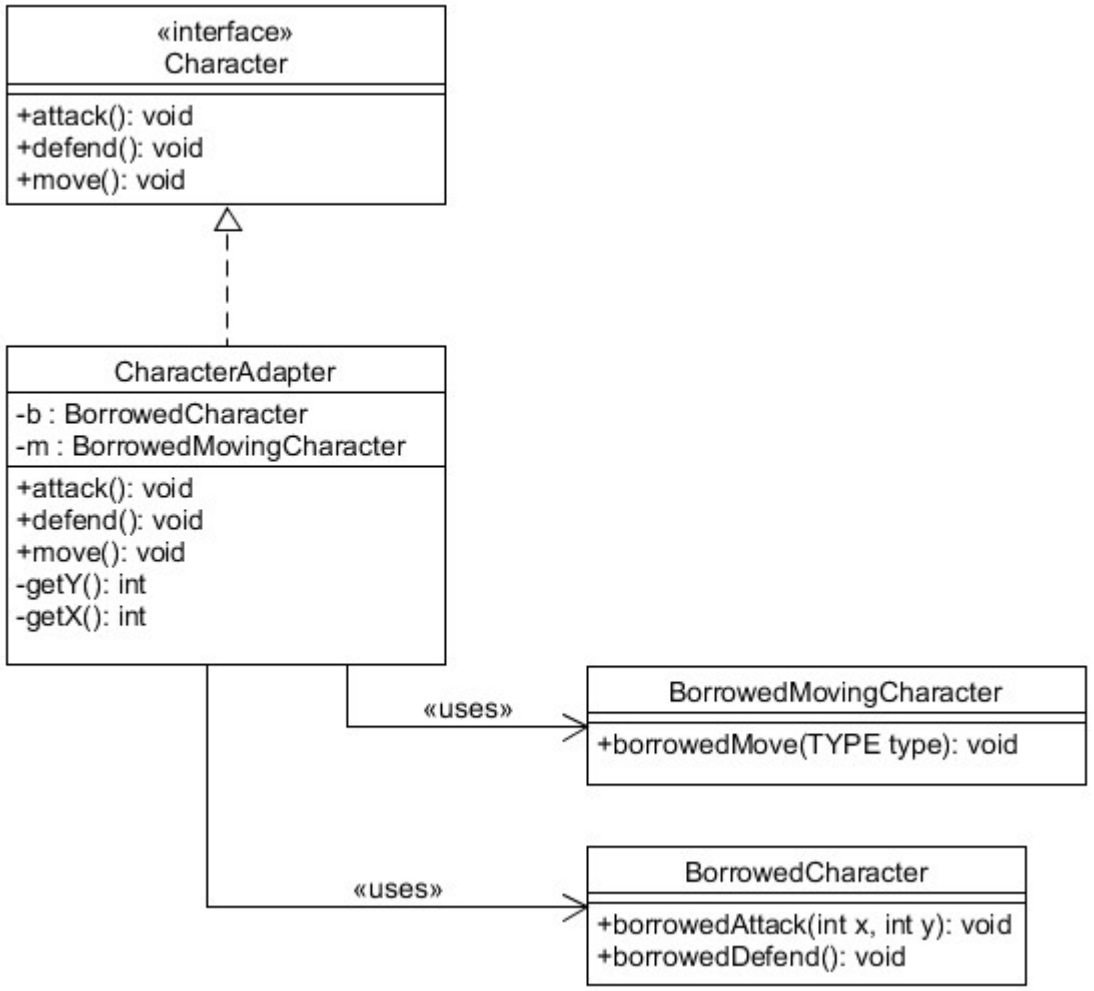
- **Purpose:** Allows incompatible classes to work together by converting the interface of one class into an interface expected by the clients.
  - Incompatible API
- Conceptually, adapters are used everywhere.
  - For example, electrical adapters can be used to connect devices with incompatible interfaces



Adapter Pattern

Consider the gaming system discussed previously. At each level, the core of the gaming system (i.e., GameEngine) uses the Character interface to add enemy characters to the game, making them move, defend, and attack using the move(), defend(), and attack() interface methods, respectively. An **online character developer** has created a special character that is compatible with the game development's application **programming interface (API) but not with the particular Character interface**; that is, the special character designed by the online developer. The special character is made available freely to the gaming community; however, the special character code can be **downloaded and incorporated into other gaming systems only as a binary compiled library**, which can be incorporated into the existing game.

Adapter Pattern



Adapter Pattern

```java
public interface Character {

    public void attack();

    public void defend();

    public void move();
}
```

```java
public class BorrowedCharacter {

    public void borrowedAttack(int x, int y) {
        System.out.println("borrowedAttack "
                + "starts at location ("
                + x + "," + y + ")");
    }

    public void borrowedDefend() {
        System.out.println("borrowedDefend starts..");
    }
}
```

```java
public class BorrowedMovingCharacter {

    public enum TYPE {
        FORWARD, BACKWARD
    };

    public void borrowedMove(TYPE type) {
        System.out.println("borrowedMove starts..");
    }
}
```

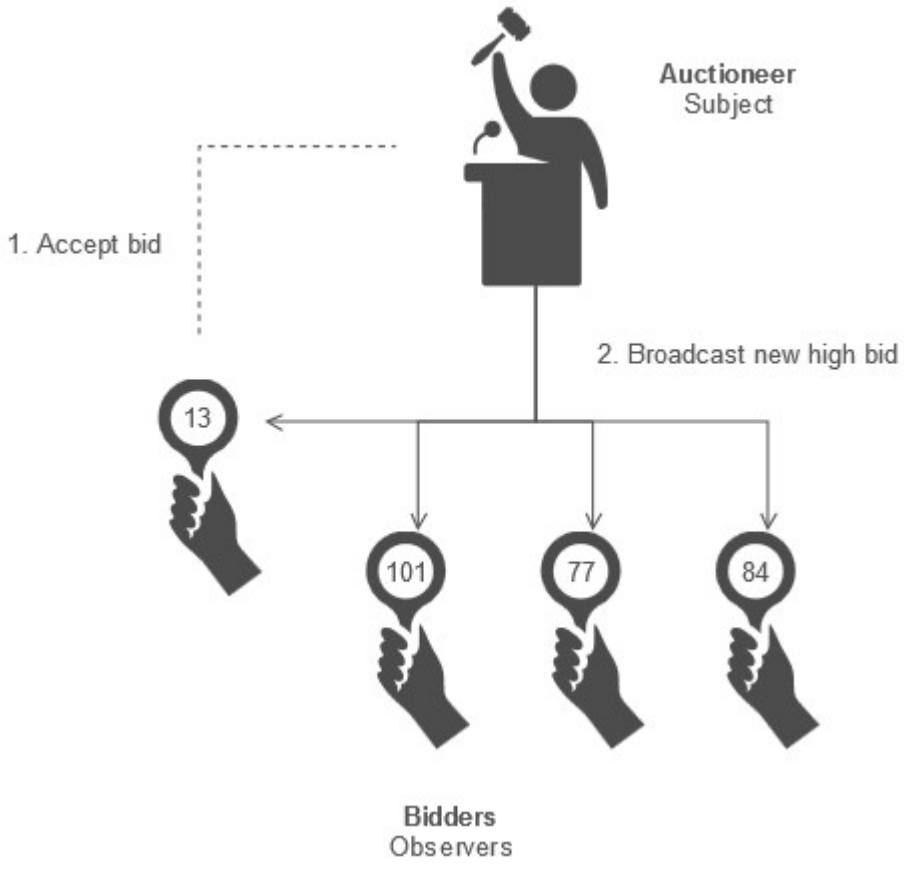
Adapter Pattern

```
public class CharacterAdapter implements Character {

    private BorrowedCharacter b = new BorrowedCharacter();
    private BorrowedMovingCharacter m = new BorrowedMovingCharacter();

    @Override
    public void attack() {
        b.borrowedAttack(getX(), getY());
    }

    @Override
    public void defend() {
        b.borrowedDefend();

    }

    @Override
    public void move() {
        m.borrowedMove(BorrowedMovingCharacter.TYPE.FORWARD);
    }

    private int getY() {
        // Get current X coordinate
        return 0;
    }

    private int getX() {
        // Get current Y coordinate
        return 0;
    }
}
```

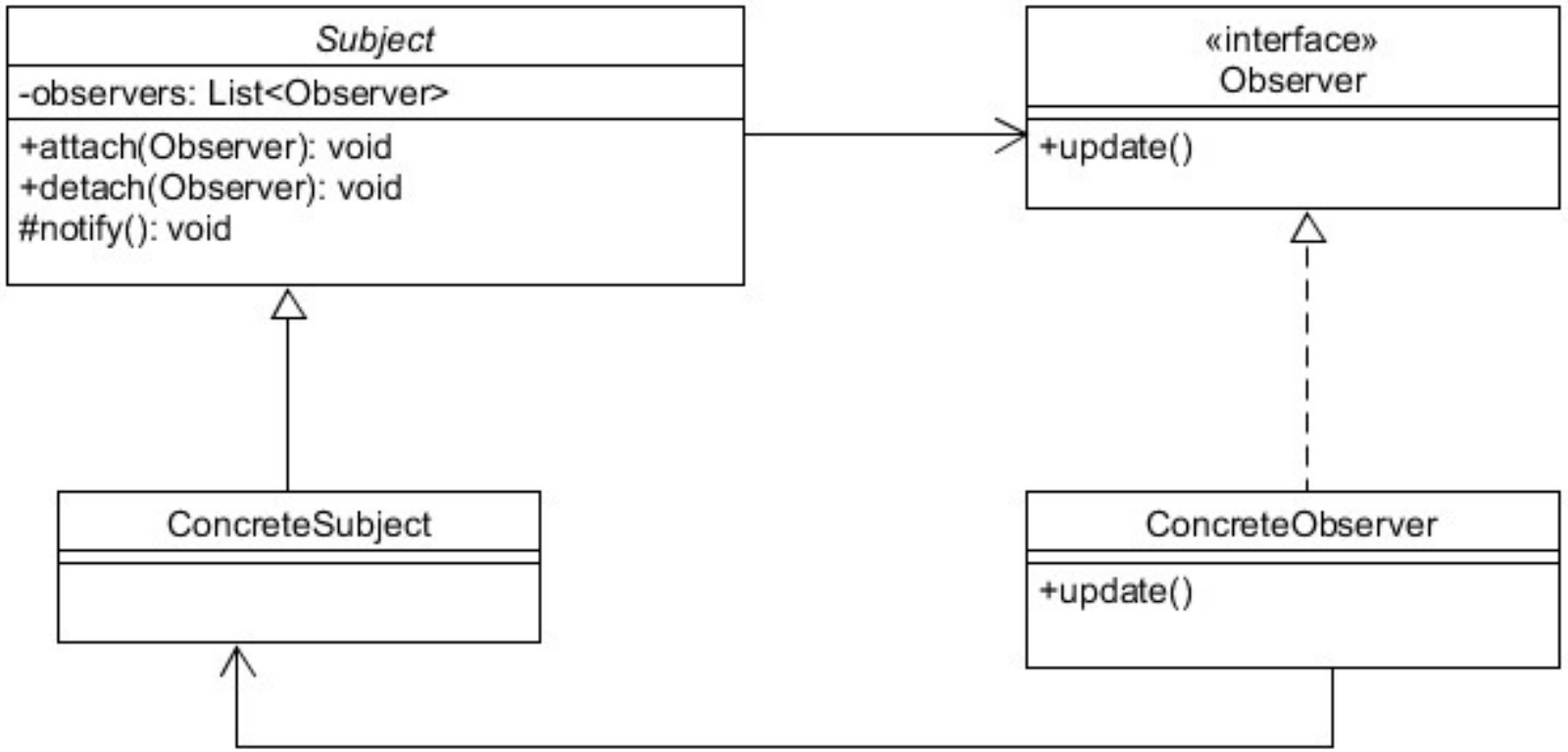
```
public static void main(String[] args) {
    Character ch = new CharacterAdapter();
    ch.attack();
    ch.defend();
    ch.move();
}
```

Observer Design Pattern (Behavioral)

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. In many practical applications, a common design structure is required to support interaction between objects that monitor a common data source so that when changes occur in the data source the objects react appropriately. Consider a detailed design that supports the model–view–controller (MVC) architectural pattern. In the MVC architectural pattern, once the data in the model component changes, all views must change as well. In cases such as the MVC, the observer design pattern provides the necessary structural interfaces to allow one or more views to register with the model component. Once registered, the observer design pattern provides the structural interfaces for executing a uniform change propagation mechanism for the model to notify all registered views of the recent changes.

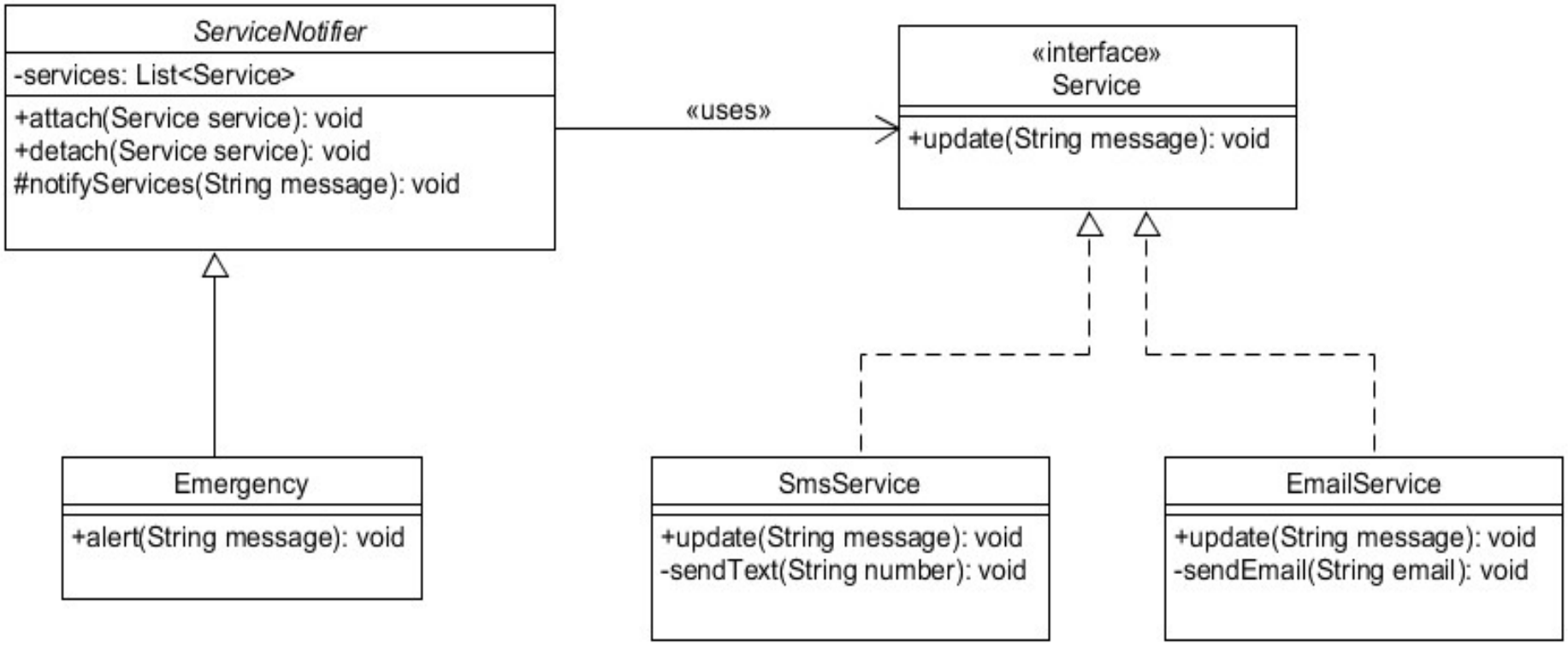


Observer Pattern



Observer Design Pattern Example

- A local university is designing a system for weather-alert notification that allows students, faculty, and staff to receive notifications of class cancellations (due to weather) via e-mail or SMS text messages.



Observer Design Pattern Example

```java
public interface Service {
    public void update(String message);
}
```

```java
public class SmsService implements Service {

    @Override
    public void update(String message) {
        sendSms(message);
    }

    private void sendSms(String message) {
        List<String> numbers = Database.getDatabase().getRegNumbers();
        for (String num : numbers) {
            sendText(message, num);
        }
    }

    private void sendText(String message, String number) {
        System.out.println("(" + message + ") is sent to : " + number);
    }
}
```

Observer Design Pattern Example

```java
public interface Service {
    public void update(String message);
}
```

```java
public class EmailService implements Service {

    @Override
    public void update(String message) {
        sendEmails(message);
    }

    private void sendEmails(String message) {
        List<String> emails = Database.getDatabase().getRegEmails();
        for (String email : emails) {
            sendEmail(message, email);
        }
    }

    private void sendEmail(String message, String email) {
        System.out.println("(" + message + ") is sent to : " + email);
    }
}
```

Observer Design Pattern Example

```
abstract public class ServiceNotifier {
    private List<Service> services =
            new ArrayList<Service>();

    public void attach(Service service) {
        services.add(service);
    }

    public void detach(Service service) {
        services.remove(service);
    }

    protected void notifyServices(String message) {
        if (services.isEmpty()) {
            System.out.println("No attached "
                    + "services...");
            return;
        }
        for (Service service : services) {
            service.update(message);
        }
    }
}
```

Observer Design Pattern

```java
public static void main(String[] args) {
    // Registered Contacts
    Database.getDatabase().addEmail("xxx@gmail.com");
    Database.getDatabase().addEmail("yyy@gmail.com");
    Database.getDatabase().addNumber("0796667723");
    Database.getDatabase().addNumber("0775557723");

    Emergency emergency = new Emergency();
    // Test case 1
    System.out.println("\nTest Case 1");
    emergency.alert("Emergency Msg 1");

    // Test case 2
    System.out.println("\nTest Case 2");
    System.out.println("About to attach Email service");
    EmailService emailService = new EmailService();
    emergency.attach(emailService);
    emergency.alert("Emergency Msg 2");

    // Test case 3
    System.out.println("\nTest Case 3");
    System.out.println("About to attach SMS service");
    SmsService smsService = new SmsService();
    emergency.attach(smsService);
    emergency.alert("Emergency Msg 3");

    // Test case 4
    System.out.println("\nTest Case 4");
    System.out.println("About to remove email service");
    emergency.detach(emailService);
    emergency.alert("Emergency Msg 4");
}
```

```java
public class Emergency extends ServiceNotifier {

    public void alert(String message) {
        System.out.println("***Alert All*** "
                + "msg: (" + message + ")");
        this.notifyServices(message);
    }
}
```

```
Test Case 1
***Alert All*** msg: (Emergency Msg 1)
No attached services...

Test Case 2
About to attach Email service
***Alert All*** msg: (Emergency Msg 2)
(Emergency Msg 2) is sent to : xxx@gmail.com
(Emergency Msg 2) is sent to : yyy@gmail.com

Test Case 3
About to attach SMS service
***Alert All*** msg: (Emergency Msg 3)
(Emergency Msg 3) is sent to : xxx@gmail.com
(Emergency Msg 3) is sent to : yyy@gmail.com
(Emergency Msg 3) is sent to : 0796667723
(Emergency Msg 3) is sent to : 0775557723

Test Case 4
About to remove email service
***Alert All*** msg: (Emergency Msg 4)
(Emergency Msg 4) is sent to : 0796667723
(Emergency Msg 4) is sent to : 0775557723
```

Observer Design Pattern

```
public class SmsService implements Service {

    @Override
    public void update(String message) {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                sendSms(message);
            }
        });
        t.start();
    }

    private void sendSms(String message) {
        List<String> numbers = Database.getDatabase().getRegNumbers();
        for (String num : numbers) {
            sendText(message, num);
        }
    }

    private void sendText(String message, String number) {
        System.out.println("(" + message + ") is sent to : " + number);
    }
}
```

Observer Design Pattern

```
public class EmailService implements Service {

    @Override
    public void update(String message) {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                sendEmails(message);
            }
        });
        t.start();
    }

    private void sendEmails(String message) {
        List<String> emails = Database.getDatabase().getRegEmails();
        for (String email : emails) {
            sendEmail(message, email);
        }
    }

    private void sendEmail(String message, String email) {
        System.out.println("(" + message + ") is sent to : " + email);
    }
}
```

Observer Design Pattern

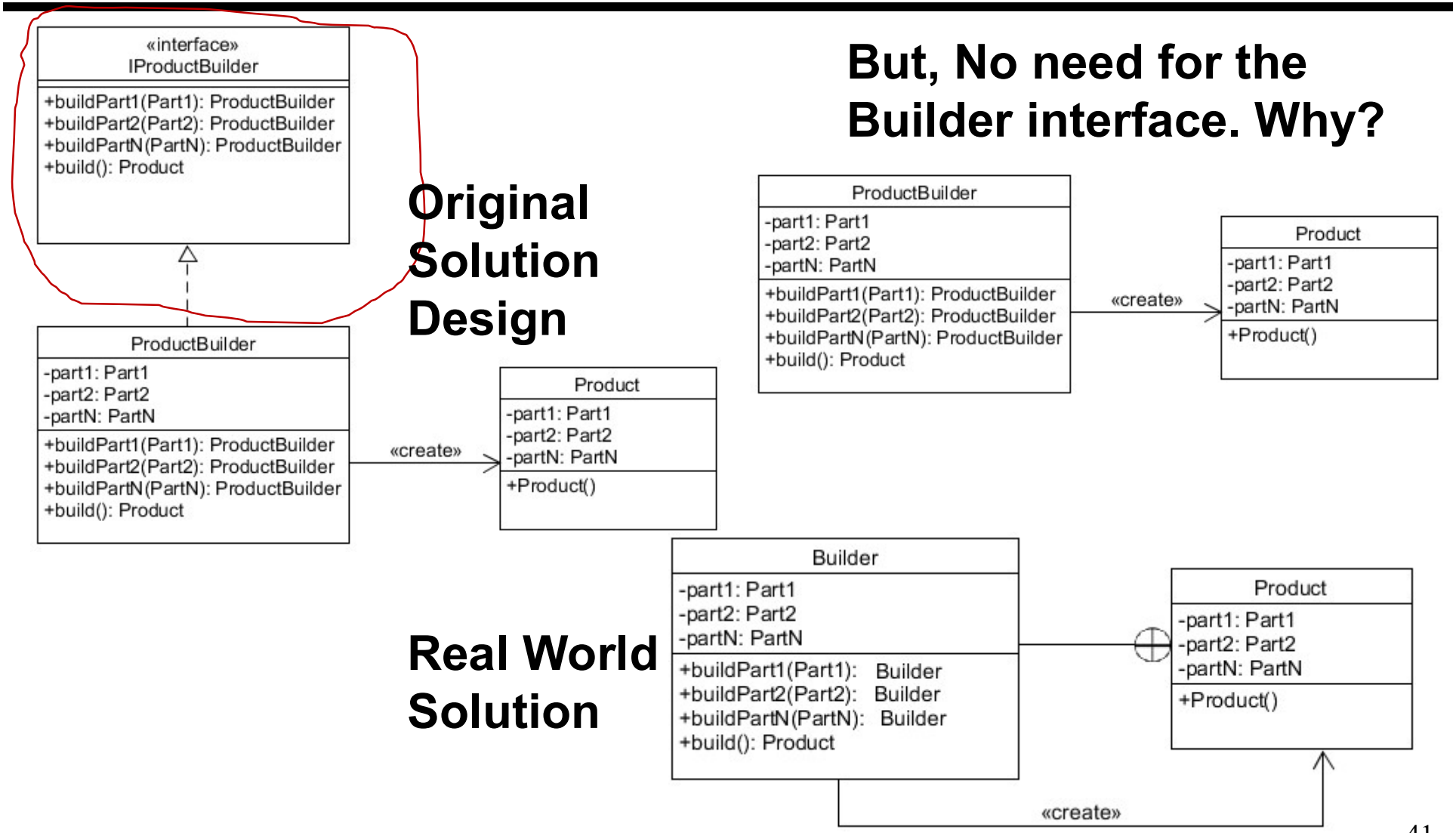
```
public static void main(String[] args) {
    // Registered Contacts
    Database.getDatabase().addEmail("xxx@gmail.com");
    Database.getDatabase().addEmail("yyy@gmail.com");
    Database.getDatabase().addNumber("0796667723");
    Database.getDatabase().addNumber("0775557723");

    Emergency emergency = new Emergency();
    EmailService emailService = new EmailService();
    emergency.attach(emailService);
    SmsService smsService = new SmsService();
    emergency.attach(smsService);
    emergency.alert("Emergency Msg");
}
```

```
***Alert All*** msg: (Emergency Msg)
(Emergency Msg) is sent to : 0796667723
(Emergency Msg) is sent to : xxx@gmail.com
(Emergency Msg) is sent to : 0775557723
(Emergency Msg) is sent to : yyy@gmail.com
```

Builder Pattern (Creational)

- The builder design pattern is an object creational pattern that encapsulates both the creational process and the representation of product objects.
  - Used to encapsulate the construction of a product and allow it to be constructed in steps
  - It Separates the construction of a complex object from its representation so that the **same construction** process can create different **representations**.
  - Unlike the abstract factory design pattern, in which various product objects are created all at once, the builder design pattern allows clients **to control the (multistep) creational** process of a **single product object**, allowing them to dictate the creation of individual parts of the object at discrete points throughout software operations.
- In practice: Builder pattern provides a solution to the telescoping constructor anti-pattern problem.
  - The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors

Builder Pattern Solution



Telescoping Constructor Anti-pattern

```java
public class Main {

    public static void main(String args[]) {
        Person p1 = new Person("Ahmad", "Ali");
        System.out.println("P1 = " + p1);
    }
}
```

How many possible constructor you need to write to handle all possible options?

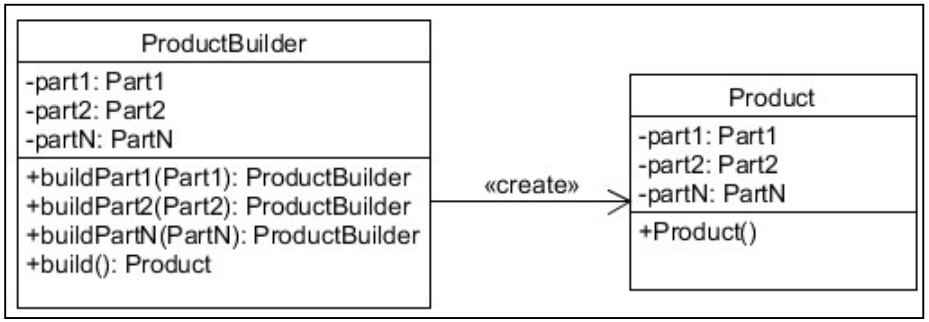
```java
public class Person {
    private String lastName;
    private String firstName;
    private String address;
    private String city;
    private String country;
    private boolean isEmployed;
    private boolean isHomeOwner;

    public Person(String lastName, String firstName, String address, String city,
                  String country, boolean isEmployed, boolean isHomeOwner) {
        this.lastName = lastName;
        this.firstName = firstName;
        this.address = address;
        this.city = city;
        this.country = country;
        this.isEmployed = isEmployed;
        this.isHomeOwner = isHomeOwner;
    }

    public Person(String lastName, String firstName) {
        this(lastName, firstName, null, null, null, false, false);
    }

    public Person(String lastName, String firstName, String address) {
        this(lastName, firstName, address, null, null, false, false);
    }

    public Person(String lastName, String firstName, boolean isEmployed) {
        this(lastName, firstName, null, null, null, isEmployed, false);
    }

    public Person(String lastName, String firstName, String city, String country) {
        this(lastName, firstName, null, null, null, false, false);
    }

    @Override
    public String toString() {
        return "[lastName: " + lastName + " firstName: " + firstName + " address: " + address + " city: " + city
                + " country: " + country + " isEmployed: " + isEmployed + " isHomeOwner: " + isHomeOwner + " ]";
    }
}
```

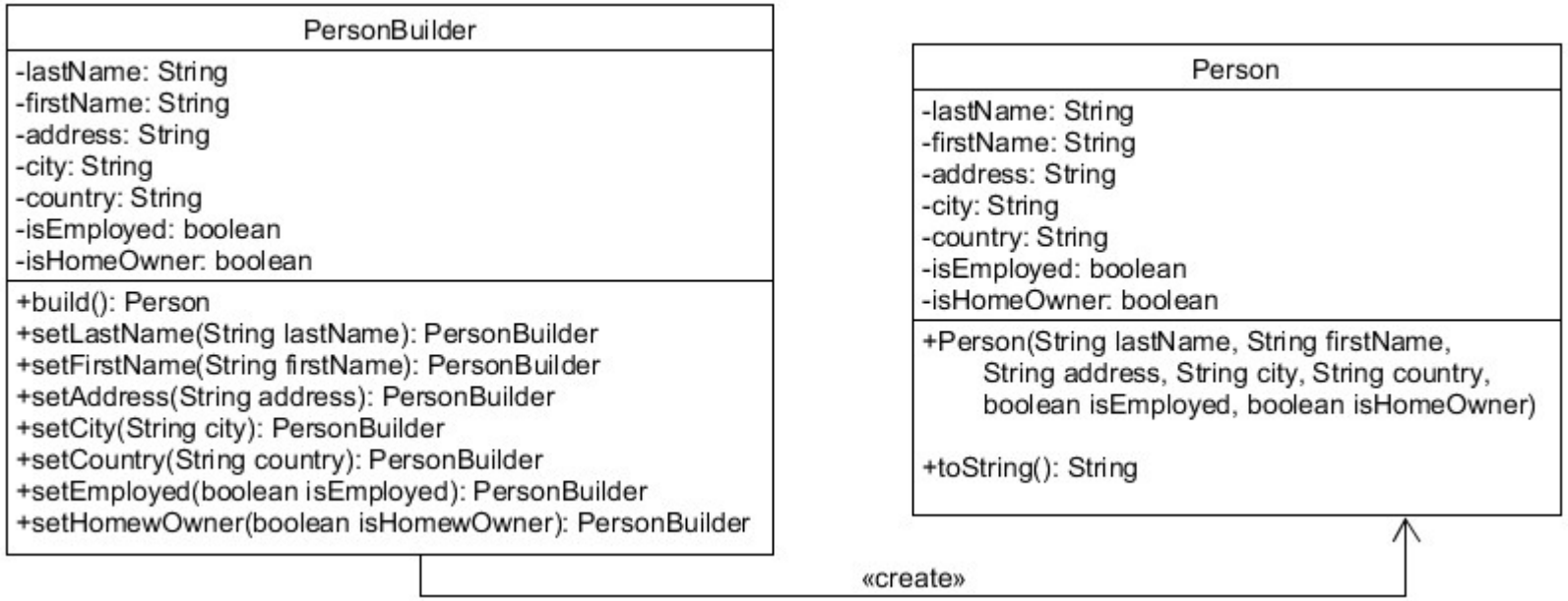
Problems Javadoc Declaration Console

```
<terminated> Main (4) [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (Mar 12, 2018, 3:54:42 PM)
P1 = [lastName: Ahmad firstName: Ali address: null city: null country: null isEmployed: false isHomeOwner: false ]
```

Builder Solution-1

General Solution





Builder Solution-1

```
public class Person {
    private String lastName;
    private String firstName;
    private String address;
    private String city;
    private String country;
    private boolean isEmployed;
    private boolean isHomeOwner;

    public Person(String lastName, String firstName, String address,
                  String city, String country, boolean isEmployed, boolean isHomeOwner) {
        this.lastName = lastName;
        this.firstName = firstName;
        this.address = address;
        this.city = city;
        this.country = country;
        this.isEmployed = isEmployed;
        this.isHomeOwner = isHomeOwner;
    }

    @Override
    public String toString() {
        return "[lastName: " + lastName + " firstName: " + firstName + " address: " + address +
               " city: " + city + " country: " + country + " isEmployed: " + isEmployed +
               " isHomeOwner: " + isHomeOwner + " ]";
    }

}
```

Builder Solution-1

```java
public class PersonBuilder {
    private String lastName;
    private String firstName;
    private String address;
    private String city;
    private String country;
    private boolean isEmployed;
    private boolean isHomeOwner;

    public Person build() {
        return new Person(lastName, firstName, address,
                          city, country, isEmployed, isHomeOwner);
    }

    public PersonBuilder setLastName(String lastName) {
        this.lastName = lastName;
        return this;
    }

    public PersonBuilder setFirstName(String firstName) {
        this.firstName = firstName;
        return this;
    }

    public PersonBuilder setAddress(String address) {
        this.address = address;
        return this;
    }
```

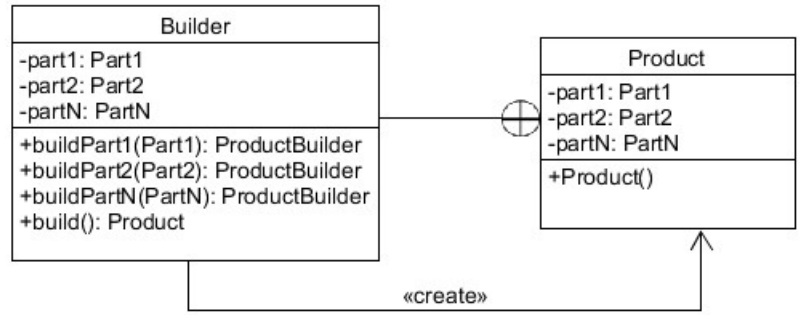
```java
    public PersonBuilder setAddress(String address) {
        this.address = address;
        return this;
    }

    public PersonBuilder setCity(String city) {
        this.city = city;
        return this;
    }

    public PersonBuilder setCountry(String country) {
        this.country = country;
        return this;
    }

    public PersonBuilder setEmployed(boolean isEmployed) {
        this.isEmployed = isEmployed;
        return this;
    }

    public PersonBuilder setHomewOwner(boolean isHomewOwner) {
        this.isHomeOwner = isHomewOwner;
        return this;
    }
}
```

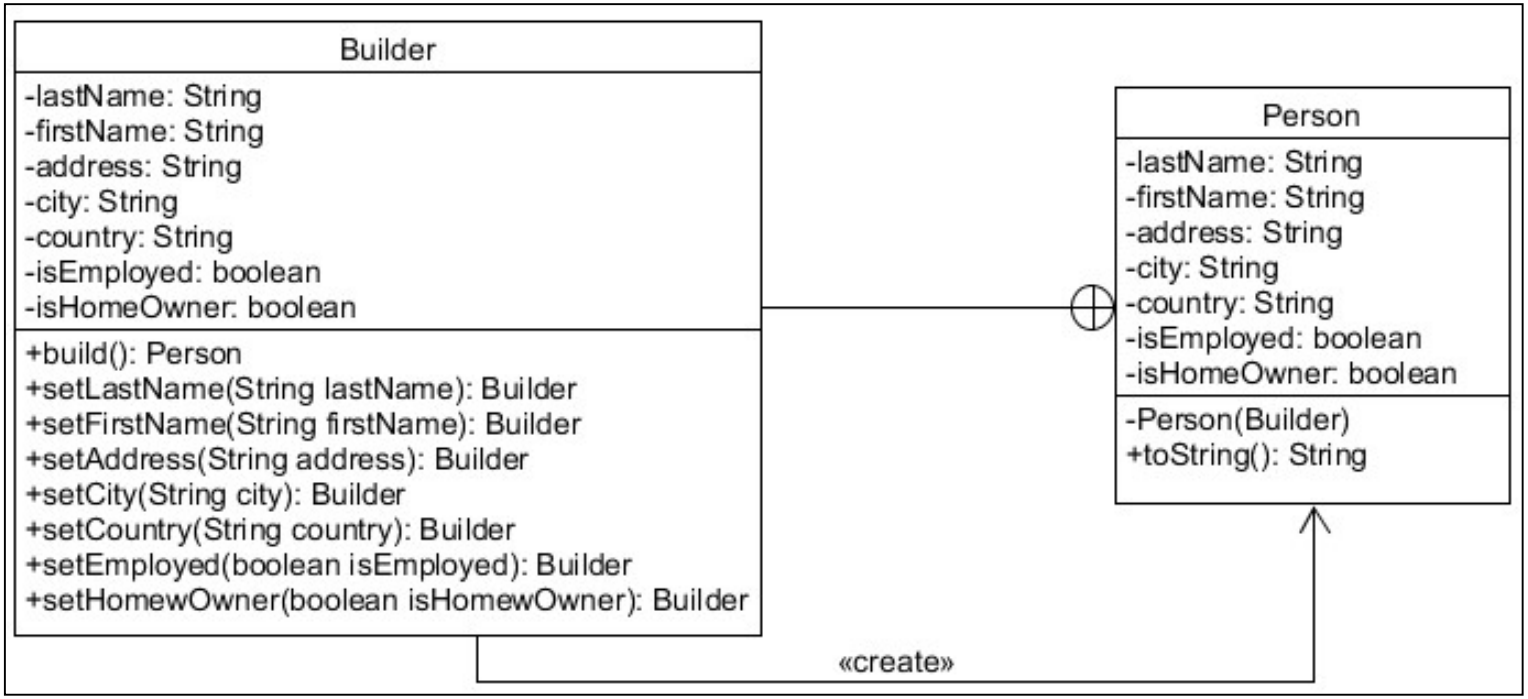
```java
public class Main {

    public static void main(String args[]) {

        Person p1 = new PersonBuilder().setLastName("Ahmad").setFirstName("Ali").build();
        System.out.println("P1 = " + p1);
    }
}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Main (4) [Java Application] C:\Program Files\Java\jre-9.0.1\bin\java
P1 = [lastName: Ahmad firstName: Ali address: null city: null
```

Builder Solution-2

General Solution





Builder Solution-2

```java
public class Person {
    private String lastName;
    private String firstName;
    private String address;
    private String city;
    private String country;
    private boolean isEmployed;
    private boolean isHomeOwner;

    private Person(Builder builder) {
        this.lastName = builder.lastName;
        this.firstName = builder.firstName;
        this.address = builder.address;
        this.city = builder.city;
        this.country = builder.country;
        this.isEmployed = builder.isEmployed;
        this.isHomeOwner = builder.isHomeOwner;
    }

    @Override
    public String toString() {
        return "[lastName: " + lastName + " firstName: " +
                firstName + " address: " + address + " city: " + city
               + " country: " + country + " isEmployed: " + isEmployed
               + " isHomeOwner: " + isHomeOwner + " ]";
    }

    static public class Builder {
        private String lastName;
        private String firstName;
        private String address;
        private String city;
        private String country;
        private boolean isEmployed;
        private boolean isHomeOwner;

        public Person build() {
            return new Person(this);
        }
        public Builder setLastName(String lastName) {
            this.lastName = lastName;
            return this;
        }
        public Builder setFirstName(String firstName) {
            this.firstName = firstName;
            return this;
        }
        public Builder setAddress(String address) {
            this.address = address;
            return this;
        }
        public Builder setCity(String city) {
            this.city = city;
            return this;
        }
        public Builder setCountry(String country) {
            this.country = country;
            return this;
        }
        public Builder setEmployed(boolean isEmployed) {
            this.isEmployed = isEmployed;
            return this;
        }
        public Builder setHomewOwner(boolean isHomewOwner) {
            this.isHomeOwner = isHomewOwner;
            return this;
        }
    } // PersonBuilder
} // Person
```

```java
public class Main {

    public static void main(String args[]) {

        Person p1 = new Person.Builder().setLastName("Ahmad").setFirstName("Ali").build();
        System.out.println("P1 = " + p1);
    }
}
```

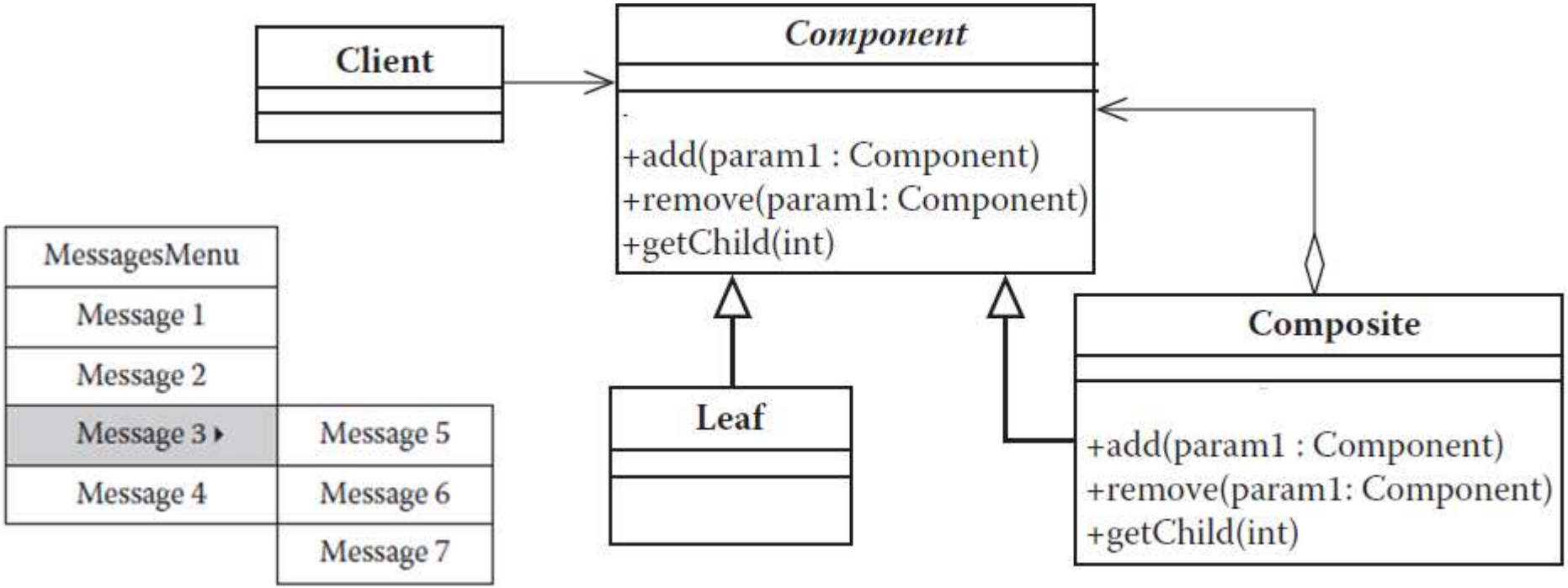
```
Problems  Javadoc  Declaration  Console
<terminated> Main (4) [Java Application] C:\Program Files\Java\jre-9
P1 = [lastName: Ahmad firstName: Ali address: null c
```

Builder Solution-2
(May put attributes together)

```java
public class Person {

    private Attributes attributes; // Person

    static private class Attributes {
        public String lastName;
        public String firstName;
        public String address;
        public String city;
        public String country;
        public boolean isEmployed;
        public boolean isHomeOwner;
    }

    private Person(Attributes builderAtts) {
        this.attributes = builderAtts;
    }

    @Override
    public String toString() {
        return "[lastName: " + attributes.lastName + " firstName: " +
                attributes.firstName + " address: "
               + attributes.address + " city: "
               + attributes.city + " country: "
               + attributes.country + " isEmployed: "
               + attributes.isEmployed + " isHomeOwner: "
               + attributes.isHomeOwner + " ]";
    }

    static public class Builder {
        private Attributes attributes = new Attributes(); // builder

        public Person build() {
            return new Person(attributes);
        }

        public Builder setLastName(String lastName) {
            this.attributes.lastName = lastName;
            return this;
        }

        public Builder setFirstName(String firstName) {
            this.attributes.firstName = firstName;
            return this;
        }

        public Builder setAddress(String address) {
            this.attributes.address = address;
            return this;
        }

        public Builder setCity(String city) {
            this.attributes.city = city;
            return this;
        }

        public Builder setCountry(String country) {
            this.attributes.country = country;
            return this;
        }

        public Builder setEmployed(boolean isEmployed) {
            this.attributes.isEmployed = isEmployed;
            return this;
        }

        public Builder setHomewOwner(boolean isHomewOwner) {
            this.attributes.isHomeOwner = isHomewOwner;
            return this;
        }
    } // PersonBuilder
} // Person
```

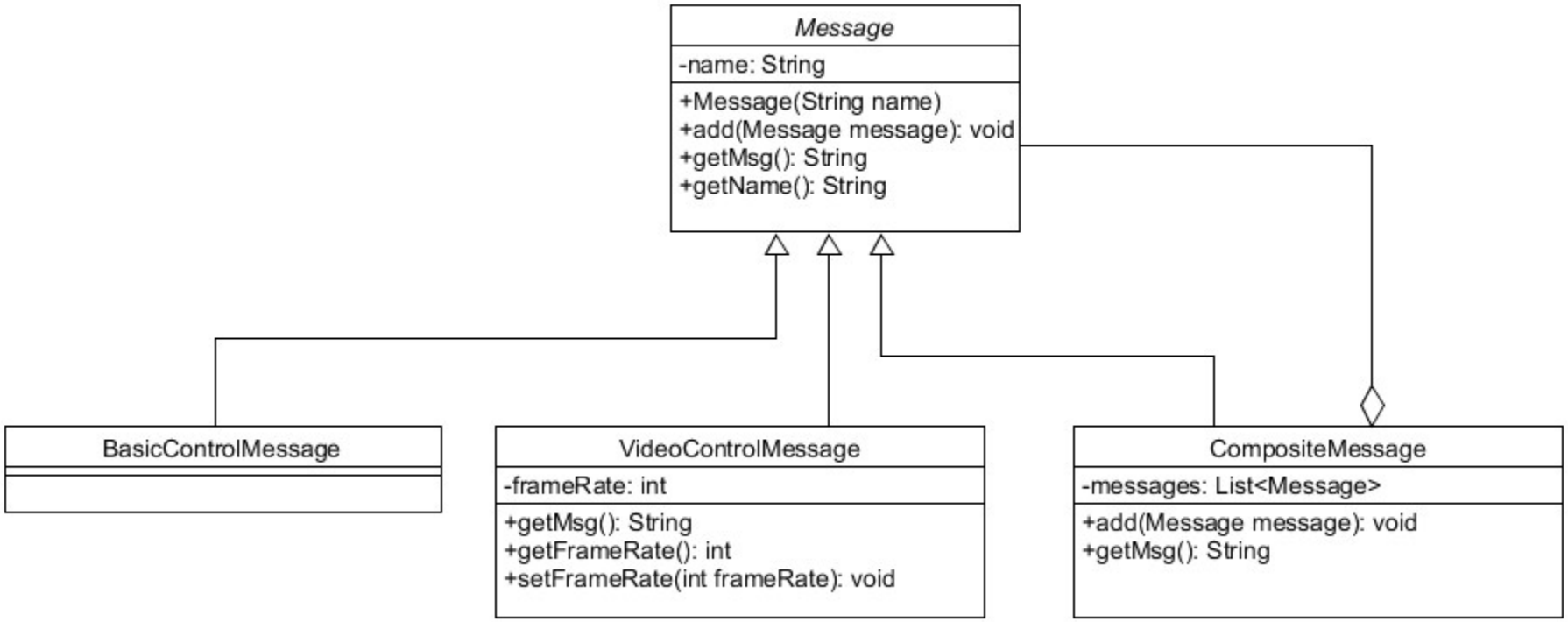
Composite Pattern (Structural)

The composite design pattern is an object structural pattern that allows designers to compose (large) tree-like design structures by strategically structuring objects that share a whole-part relationship. Whole-part relationships are those in which a larger entity (i.e., the whole) is created by the composition of smaller entities (i.e., the parts). The key advantage of using the composite design pattern is that it provides a design structure that allows both whole and part objects to be treated uniformly; therefore, operations that are common to both type of objects can be applied the same way to both types of objects.



Composite Pattern

- **Problem:** A wireless sensor system is remotely deployed to collect environmental information. The sensor system communicates via satellite to a central location, where a schedule of tasks (i.e., a mission plan) is created and sent over satellite communications. A mission plan is a composite message that contains one or more messages that command the sensor system to perform particular tasks. These messages contain information on how and when to perform particular tasks. Mission plan messages can be created with many different combinations of basic, video and composite messages.



Composite Pattern

```
abstract public class Message {
    private String name;

    public Message(String name) {
        this.name = name;
    }

    public void add(Message message) {
        System.out.println("Messages cannot be "
                + "added to Leaf objects!");
    }

    public String getMsg() {
        return "[" + getName() + "]";
    }

    public String getName() {
        return name;
    }
}
```

```
public class BasicControlMessage extends Message {

    public BasicControlMessage(String name) {
        super(name);
    }

}
```

```
public class VideoControlMessage extends Message {

    private int frameRate = 25;

    public VideoControlMessage(String name) {
        super(name);
    }

    @Override
    public String getMsg() {
        return "[" + getName() + "," + frameRate + "]";
    }

    public int getFrameRate() {
        return frameRate;
    }

    public void setFrameRate(int frameRate) {
        this.frameRate = frameRate;
    }

}
```

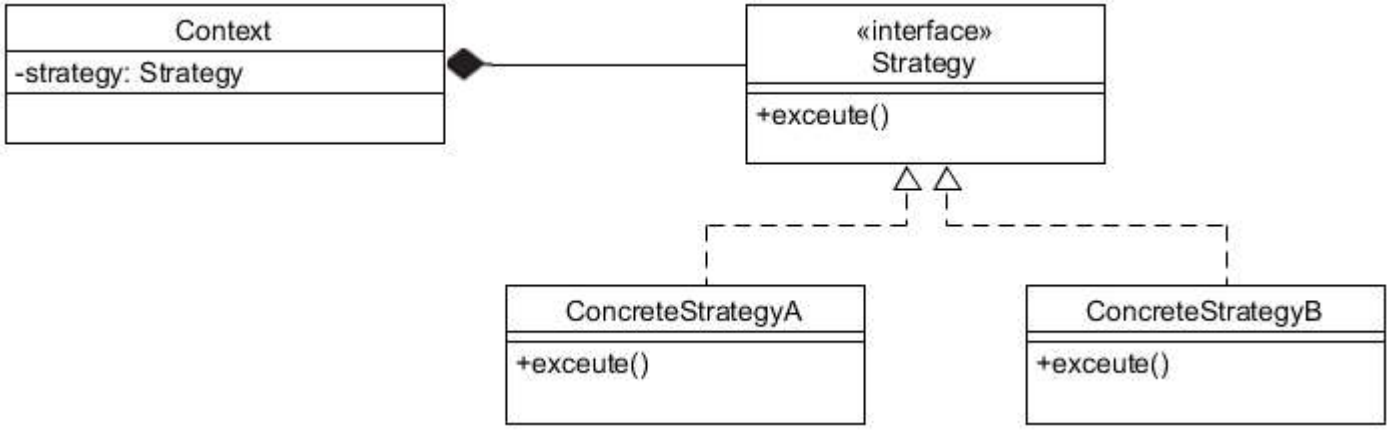
Composite Pattern

```java
public class CompositeMessage extends Message {
    public CompositeMessage(String name) {
        super(name);
    }

    private List<Message> messages = new ArrayList<Message>();

    @Override
    public void add(Message message) {
        messages.add(message);
    }

    @Override
    public String getMsg() {
        StringBuilder sb = new StringBuilder("{" + getName());
        for (Message msg : messages) {
            sb.append(msg.getMsg());
        }
        sb.append("}");
        return sb.toString();
    }
}
```

```java
public static void main(String[] args) {
    Message m1 = new CompositeMessage("m1");
    Message m2 = new CompositeMessage("m2");
    VideoControlMessage v1 = new VideoControlMessage("v1");
    v1.setFrameRate(10);
    BasicControlMessage b1 = new BasicControlMessage("b1");
    BasicControlMessage b2 = new BasicControlMessage("b2");
    m1.add(v1);
    m1.add(b1);
    m1.add(b2);
    System.out.println("print m1: " + m1.getMsg());
    m2.add(m1);
    System.out.println("print m2: " + m2.getMsg());
    VideoControlMessage v2 = new VideoControlMessage("v2");
    v1.setFrameRate(15);
    m2.add(v2);
    System.out.println("print m2: " + m2.getMsg());
}
```

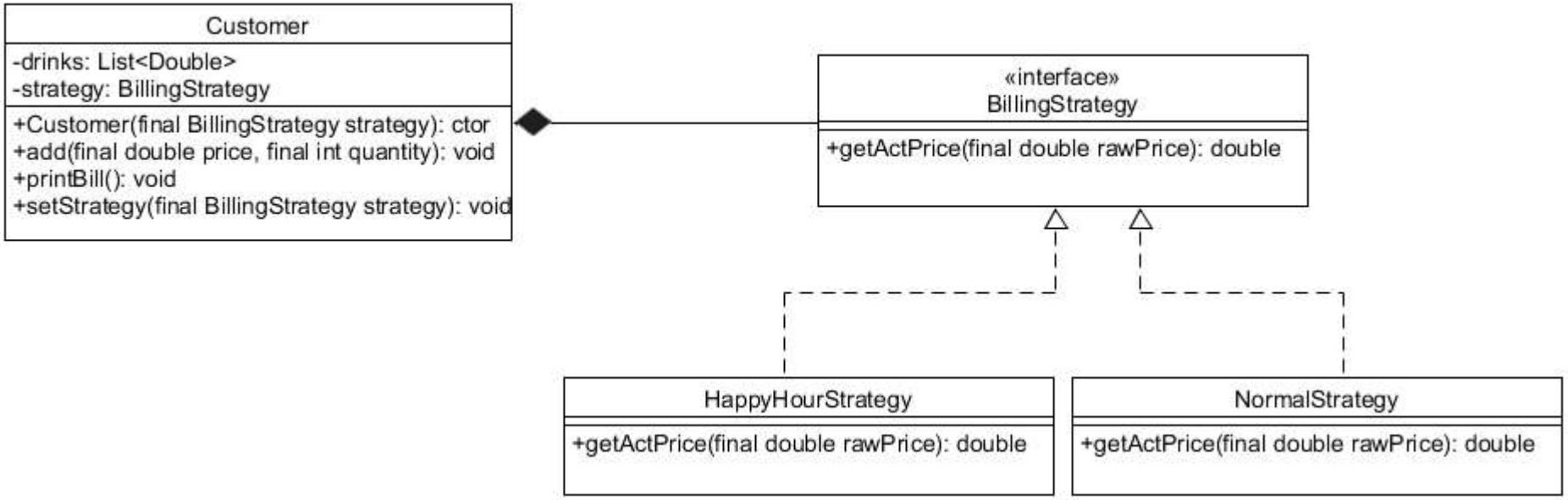
Strategy Design Pattern (Behavioral)

- Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior
  - Factory method pattern uses inheritance → to inject the object type in a behavior
  - Strategy pattern uses composition → to inject the behavior type in an object



- **When to Use:**
  - The only difference between many related classes is their behavior.
- Example: Restaurant Customers pay half price during happy hour. Otherwise pay full price

Strategy Pattern



Strategy Design Pattern (Strategy Types)

```java
public interface BillingStrategy {
    public double getActPrice(final double rawPrice);
}
```

```java
public // Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
        return rawPrice * 0.5;
    }

}
```

```java
//Normal billing strategy (unchanged price)
public class NormalStrategy implements BillingStrategy {

    @Override
    public double getActPrice(final double rawPrice) {
        return rawPrice;
    }

}
```

Strategy Design Pattern (Context)

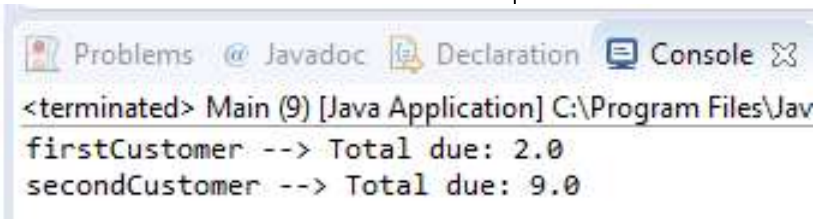
Context = Customer class

Which Composes the Strategy

```
class Customer {

    private List<Double> drinks;
    private BillingStrategy strategy;

    public Customer(final BillingStrategy strategy) {
        this.drinks = new ArrayList<Double>();
        this.strategy = strategy;
    }

    public void add(final double price, final int quantity) {
        drinks.add(strategy.getActPrice(price * quantity));
    }

    // Payment of bill
    public void printBill() {
        double sum = 0;
        for (Double i : drinks) {
            sum += i;
        }
        System.out.println("Total due: " + sum);
        drinks.clear();
    }

    // Set Strategy
    public void setStrategy(final BillingStrategy strategy) {
        this.strategy = strategy;
    }
}
```

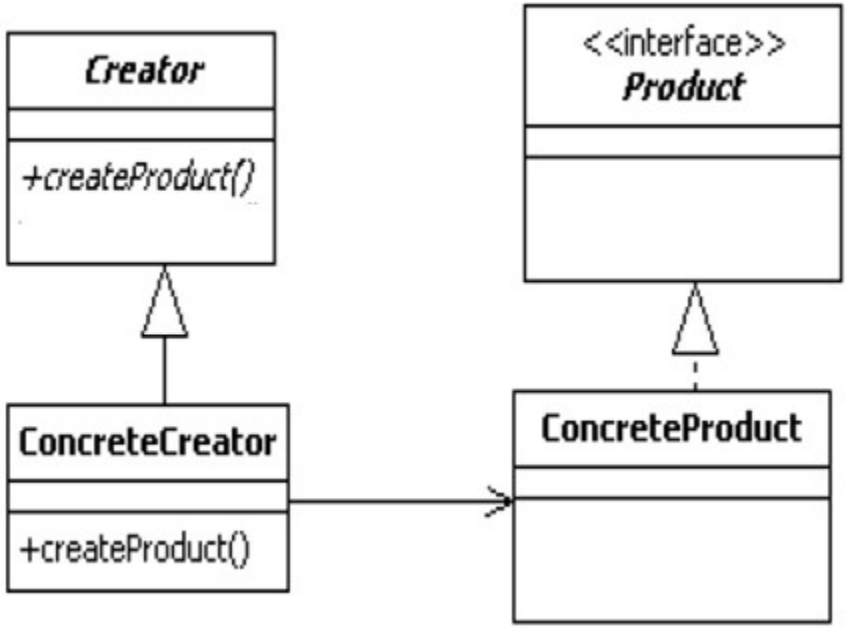
Strategy Design Pattern (Client)

```
public static void main(final String[] arguments) {
    Customer firstCustomer = new Customer(new NormalStrategy());
    firstCustomer.add(1.0, 1);
    // Start Happy Hour
    firstCustomer.setStrategy(new HappyHourStrategy());
    firstCustomer.add(1.0, 2);
    System.out.print("firstCustomer --> ");
    firstCustomer.printBill();

    // New Customer
    Customer secondCustomer = new Customer(new HappyHourStrategy());
    secondCustomer.add(1.0, 10);
    // End Happy Hour
    secondCustomer.setStrategy(new NormalStrategy());
    secondCustomer.add(2.0, 2);
    System.out.print("secondCustomer --> ");
    secondCustomer.printBill();
}
```



Factory Method (Creational Pattern)

The factory method design pattern is a class creational pattern used to encapsulate and defer object instantiation to derived classes. Structurally, the factory method can be modeled as a simplified version of the abstract factory design pattern, since both patterns require creator and product interfaces. However, unlike the abstract factory design pattern, in which the creator objects (i.e., factories) are responsible for instantiating a plurality of products that belong to a specific family type, creator objects in the factory method design pattern are responsible for the creation of a single **product** of specific type. Therefore, the creator interface for the factory method design pattern provides only one creational method, whereas the creator interface for the abstract factory design pattern provides two or more creational methods.



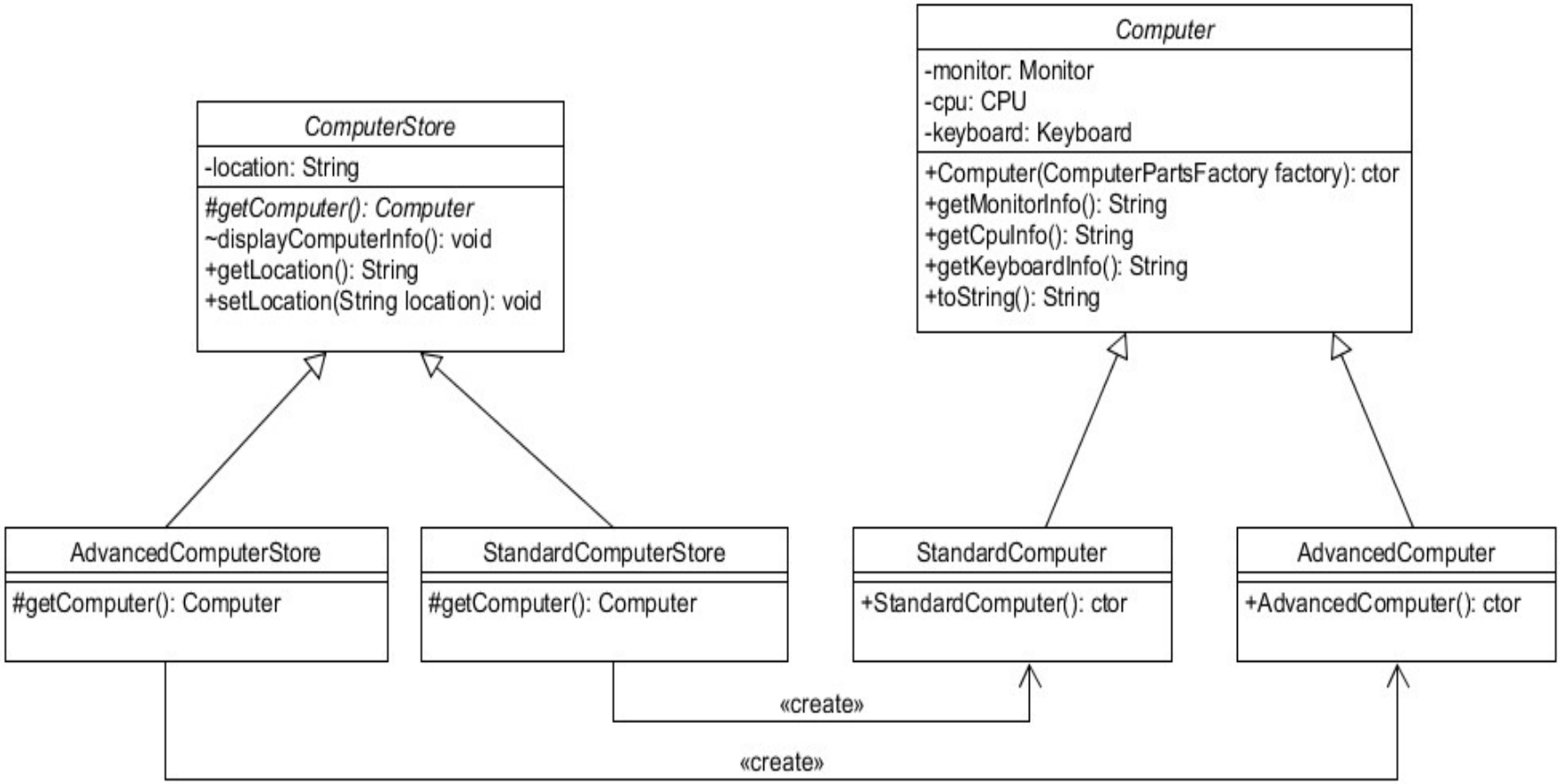
Factory Method

Suppose we have computer stores that want to display products information for Standard and Advanced computers.

How factory method pattern would change this code design?

```
public class ComputerStore {
    private String location;
    public enum TYPE {
        STANDARD, ADVANCED
    };

    void displayComputerInfo(TYPE type) {
        Computer computer;
        if (type == TYPE.STANDARD) {
            computer = new Computer(new StandardComputerPartsFactory());
        } else {
            computer = new Computer(new AdvancedComputerPartsFactory());
        }

        // display computer info
        System.out.println("CPU: " + computer.getCpuInfo());
        System.out.println("Keyboard: " + computer.getKeyboardInfo());
        System.out.println("Monitor: " + computer.getMonitorInfo());
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }
}
```

Factory Method (Solution)



Factory Method (Solution)

```
public abstract class ComputerStore {
    private String location;

    protected abstract Computer getComputer();

    void displayComputerInfo() {

        Computer computer = getComputer();

        // display computer info
        System.out.println("CPU: " + computer.getCpuInfo());
        System.out.println("Keyboard: " + computer.getKeyboardInfo());
        System.out.println("Monitor: " + computer.getMonitorInfo());
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }
}
```

```
public class StandardComputer extends Computer {
    public StandardComputer() {
        super(new StandardComputerPartsFactory());
    }
}
```

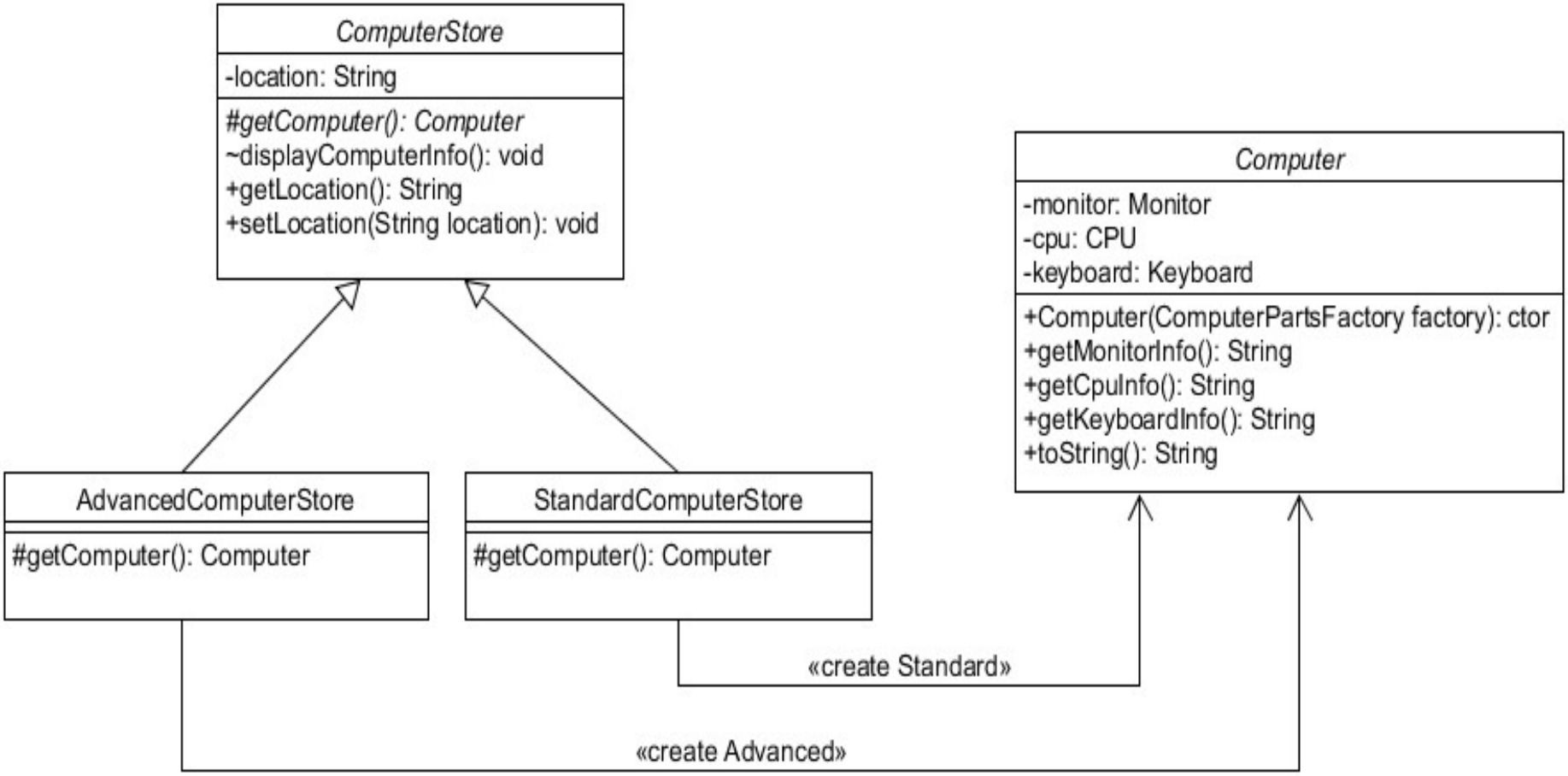
```
public class AdvancedComputer extends Computer {
    public AdvancedComputer() {
        super(new AdvancedComputerPartsFactory());
    }
}
```

```
public class StandardComputerStore extends ComputerStore {
    @Override
    protected Computer getComputer() {
        return new StandardComputer();
    }
}
```

```
public class AdvancedComputerStore extends ComputerStore {
    @Override
    protected Computer getComputer() {
        return new AdvancedComputer();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ComputerStore store = new AdvancedComputerStore();
        store.displayComputerInfo();
        System.out.println("====================");
        store = new StandardComputerStore();
        store.displayComputerInfo();
    }
}
```

Factory Method
(Another solution)



Factory Method (Another Solution)

```java
public abstract class ComputerStore {
    private String location;

    protected abstract Computer getComputer();

    void displayComputerInfo() {

        Computer computer = getComputer();

        // display computer info
        System.out.println("CPU: " + computer.getCpuInfo());
        System.out.println("Keyboard: " + computer.getKeyboardInfo());
        System.out.println("Monitor: " + computer.getMonitorInfo());
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }
}
```

```java
public class StandardComputerStore extends ComputerStore {
    @Override
    protected Computer getComputer() {
        return new Computer(new StandardComputerPartsFactory());
    }
}
```

```java
public class AdvancedComputerStore extends ComputerStore {
    @Override
    protected Computer getComputer() {
        return new Computer(new AdvancedComputerPartsFactory());
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        ComputerStore store = new AdvancedComputerStore();
        store.displayComputerInfo();
        System.out.println("====================");
        store = new StandardComputerStore();
        store.displayComputerInfo();
    }
}
```

Problems @ Javadoc Declaration Console

```
<terminated> Main (2) [Java Application] C:\Program Files\Java\
CPU: Advanced CPU
Keyboard: Advanced Keyboard
Monitor: Advanced Monitor
====================
CPU: Standard CPU
Keyboard: Standard Keyboard
Monitor: Standard Monitor
```

Decorator pattern (Structural)

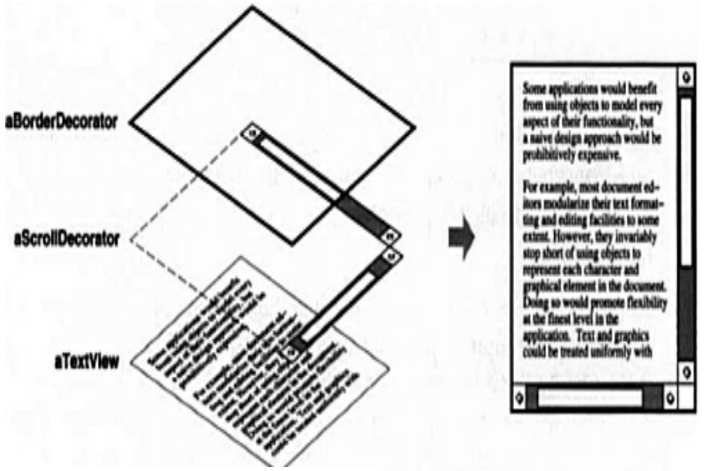
- **Exercise:** We want to design **new classes** to do more optional printing **based on the A class**:
- print AX
- print AY
- print AZ
- print AXY
- print AXYZ

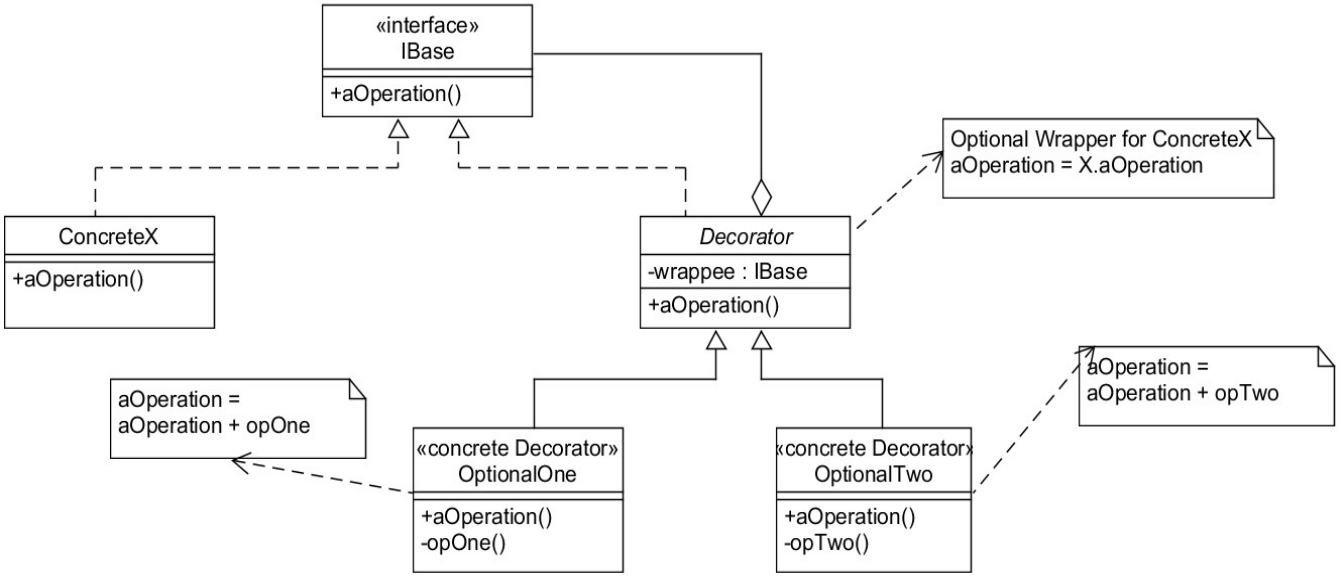
- What is the core functionality?
- What are the added options?

```
interface I {
    void doIt();
}

class A implements I {
    public void doIt() {
        System.out.print('A');
    }
}
```

Decorator pattern (Structural)

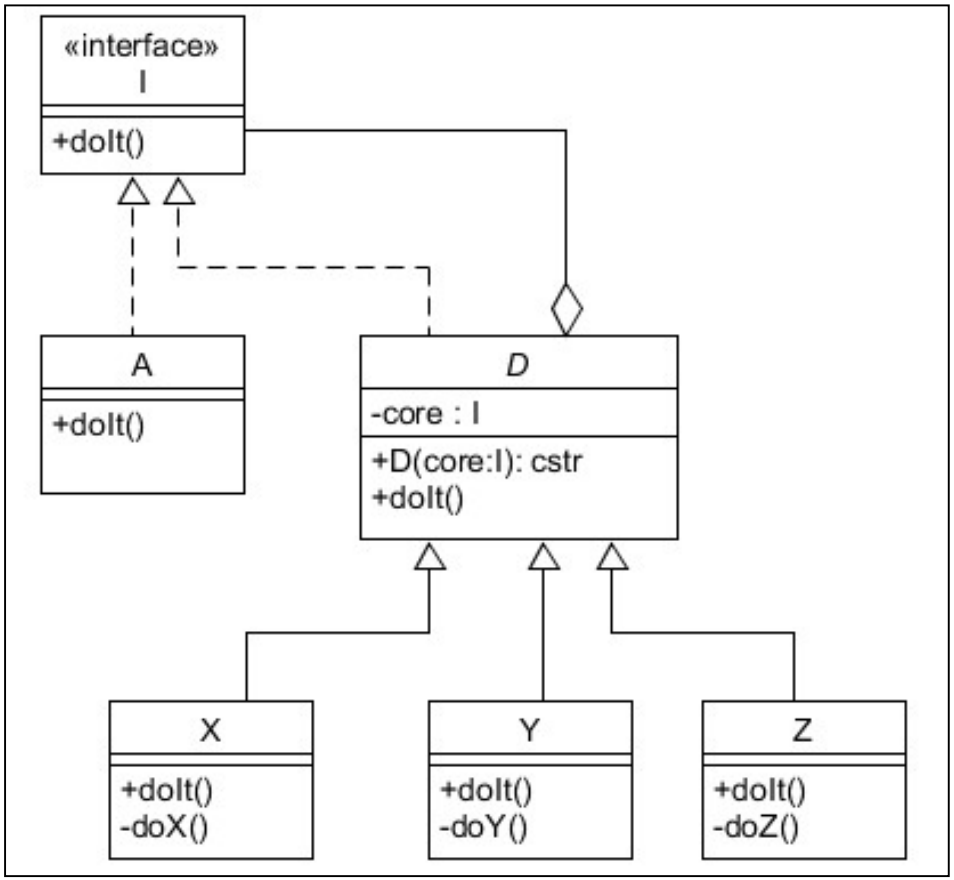
- Decorator is a structural pattern that composes objects recursively to allow an open-ended number of additional optional responsibilities.
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified exaggeration of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.





Decorator pattern (Structural)

```
abstract class D implements I {
    private I core;
    public D(I inner) {
        core = inner;
    }

    public void doIt() {
        core.doIt();
    }
}
```



Decorator pattern (Structural)

```
class X extends D {
    public X(I inner) {
        super(inner);
    }

    public void doIt() {
        super.doIt();
        doX();
    }

    private void doX() {
        System.out.print('X');
    }
}
```

```
class Y extends D {
    public Y(I inner) {
        super(inner);
    }

    public void doIt()  {
        super.doIt();
        doY();
    }

    private void doY() {
        System.out.print('Y');
    }
}
```

```
class Z extends D {
    public Z(I inner) {
        super(inner);
    }

    public void doIt()  {
        super.doIt();
        doZ();
    }

    private void doZ() {
        System.out.print('Z');
    }
}
```

```
public class DecoratorDemo {
    public static void main( String[] args ) {
        I[] array = {new X(new A()), new Y(new X(new A())),
                new Z(new Y(new X(new A())))};
        for (I anArray : array) {
            anArray.doIt();
            System.out.print("  ");
        }
    }
}
```

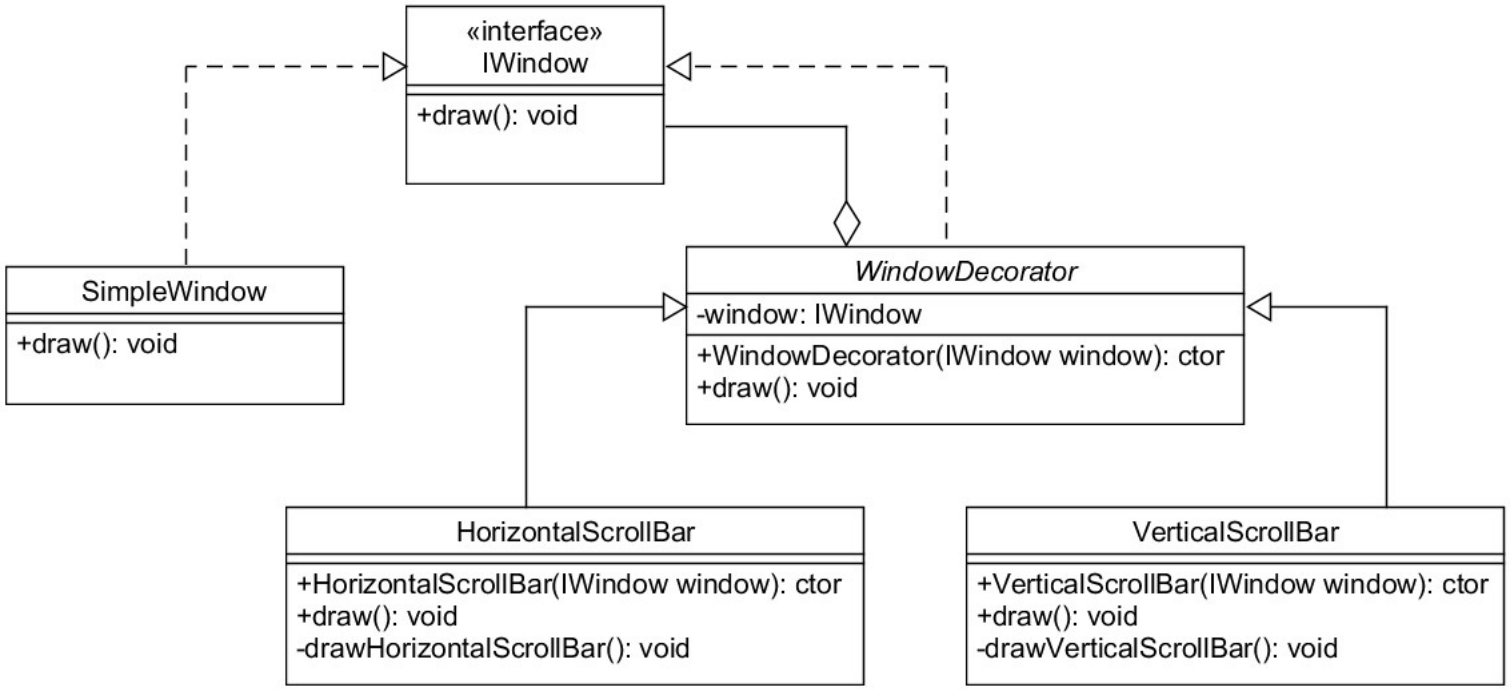
Output

```
AX  AXY  AXYZ
```

Decorator pattern (Structural)

- As another example, consider a window in a graphical tool. Windows need to allow scrolling of the window's contents, one may wish to add horizontal or vertical scrollbars to it, as appropriate.
  - We want to draw a window and/or (horizontal and/or vertical scrollbars)

Decorator pattern (Structural)



Decorator pattern (Structural)

```java
package example;

public interface IWindow {
    public void draw();
}
```

```java
public class SimpleWindow implements IWindow {

    @Override
    public void draw() {
        System.out.println("Draw SimpleWindow");

    }
}
```

```java
abstract public class WindowDecorator implements IWindow {
    private IWindow window;

    public WindowDecorator(IWindow window) {
        this.window = window;
    }

    @Override
    public void draw() {
        window.draw();
    }
}
```

Decorator pattern (Structural)

```java
class HorizontalScrollBar extends WindowDecorator {
    public HorizontalScrollBar(IWindow window) {
        super(window);
    }

    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }

    private void drawHorizontalScrollBar() {
        System.out.println("Draw HorizontalScrollBar");
    }
}
```

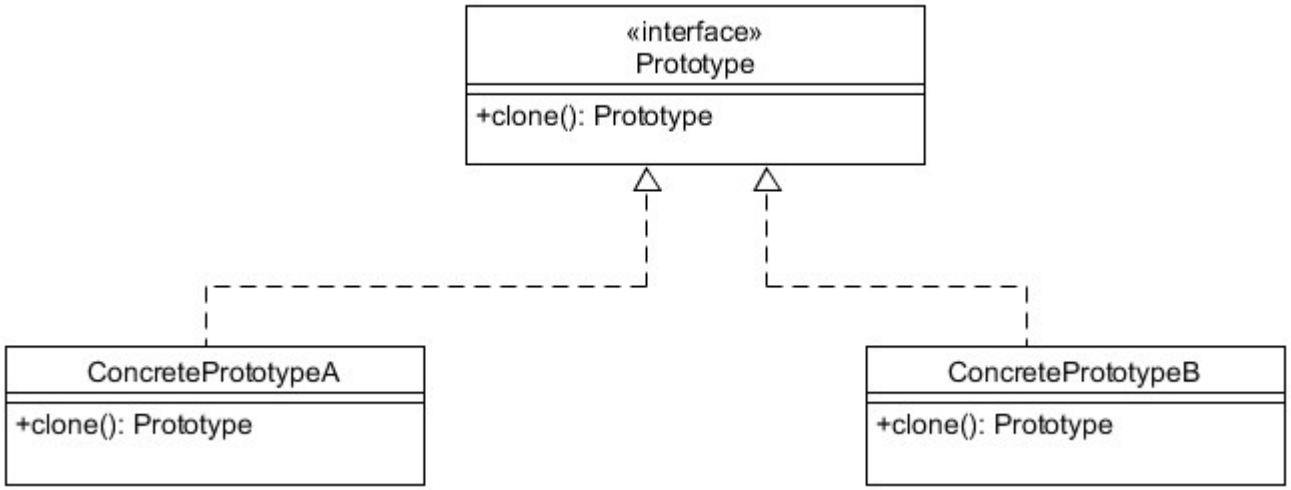
```java
class VerticalScrollBar extends WindowDecorator {
    public VerticalScrollBar(IWindow window) {
        super(window);
    }

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

    private void drawVerticalScrollBar() {
        System.out.println("Draw VerticalScrollBar");
    }

}
```

```
Problems  Javadoc  Declaration  Console
<terminated> Main (7) [Java Application] C:\Prograr
Draw SimpleWindow
Draw VerticalScrollBar
Draw HorizontalScrollBar
```

```java
public class Main {
    public static void main(String[] args) {
        IWindow window = new HorizontalScrollBar(
                                new VerticalScrollBar(
                                        new SimpleWindow()));
        window.draw();

    }
}
```

Prototype Pattern

- The Prototype design pattern is a class creational pattern that allows clients to create duplicates of prototype objects at run-time without knowing the objects' specific type.
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Example

- **Problem:** The game designers have identified the need to have each character provide a method for creating copies of themselves so that at any given point during the game a character clone can be made including identical profiles.

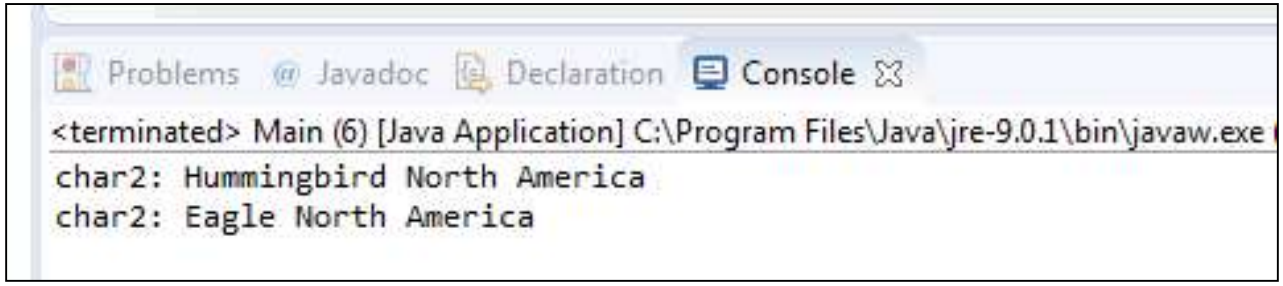
```
public interface Character {

    public String getName();

    public void setName(String name);

    public String getLocation();

    public void setLocation(String location);

}
```

```
public class TerrestrialEnemyCharacter implements Character {

    private Info info = new Info();

    public TerrestrialEnemyCharacter() {
    }

    public TerrestrialEnemyCharacter(TerrestrialEnemyCharacter
                                     aEnemyChar) {
        this.info = aEnemyChar.info;
    }

    public TerrestrialEnemyCharacter duplicateTerrestrial() {
        return new TerrestrialEnemyCharacter(this);
    }

    // Other methods.....
}
```

```
public class AerialEnemyCharacter implements Character {

    private Info info = new Info();

    public AerialEnemyCharacter() {
    }

    public AerialEnemyCharacter(AerialEnemyCharacter aEnemyChar) {
        this.info = aEnemyChar.info;
    }

    public AerialEnemyCharacter duplicateTerrestrial() {
        return new AerialEnemyCharacter(this);
    }

    @Override
    public String getName() {
        return info.getName();
    }

    @Override
    public void setName(String name) {
        this.info.setName(name);
    }

    @Override
    public String getLocation() {
        return this.info.getLocation();
    }

    @Override
    public void setLocation(String location) {
        this.info.setLocation(location);
    }
}
```

Example

```
public static void main(String[] args) {
    AerialEnemyCharacter char1 = new AerialEnemyCharacter();
    char1.setName("Hummingbird");
    char1.setLocation("North America");
    AerialEnemyCharacter char2 = char1.duplicateTerrestrial();
    System.out.println("char2: " + char2.getName() + " " + char2.getLocation());
    char1.setName("Eagle"); // *** Changed char1 **/
    System.out.println("char2: " + char2.getName() + " " + char2.getLocation());
}
```



What is wrong with the output?

Example

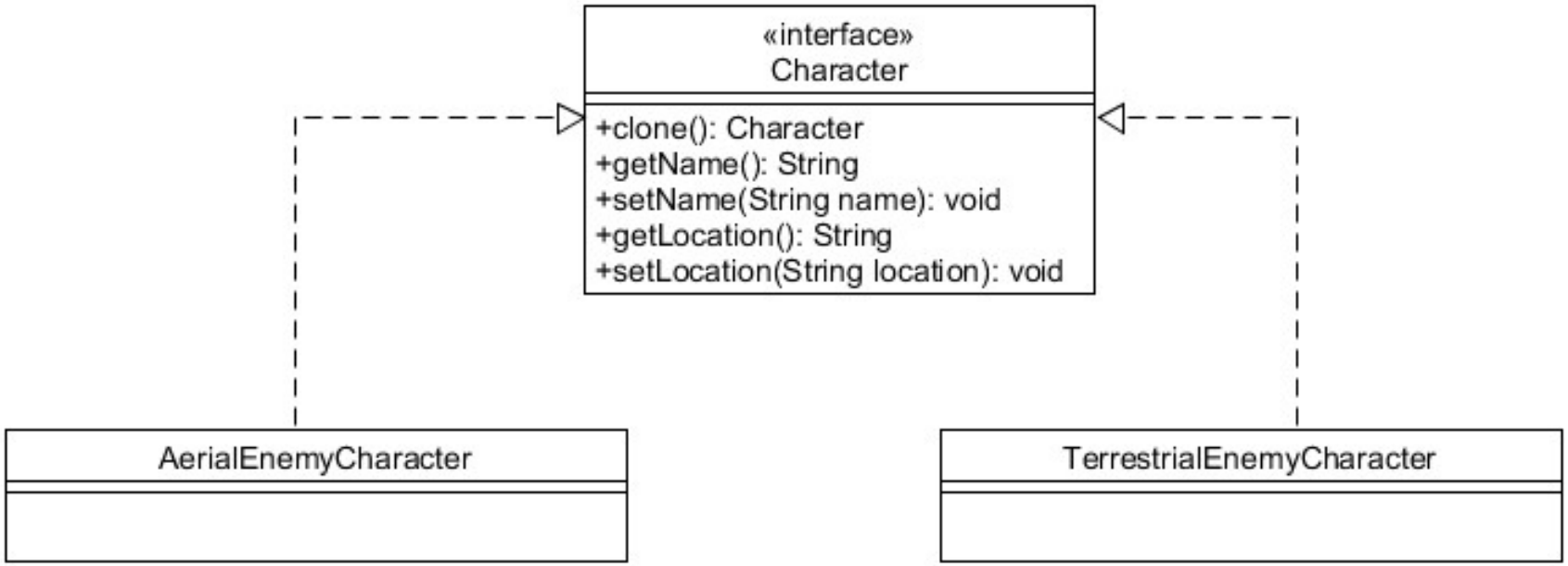
```
public static void main(String[] args) {
    AerialEnemyCharacter char1 = new AerialEnemyCharacter();
    char1.setName("Hummingbird");
    char1.setLocation("North America");
    AerialEnemyCharacter char2 = char1.duplicateTerrestrial();
    System.out.println("char2: " +
            char2.getName() + " " + char2.getLocation());
    char1.setName("Eagle"); // *** Changed char1 **/
    System.out.println("char2: " +
            char2.getName() + " " + char2.getLocation());
    System.out.println("char1: " +
            char1.getName() + " " + char1.getLocation());
}
```

Are we copying Correctly?

Can we use Character in the above "main" method?

```
public class AerialEnemyCharacter implements Character {

    private Info info = new Info();

    public AerialEnemyCharacter() {
    }

    public AerialEnemyCharacter(AerialEnemyCharacter aEnemyChar) {
        this.info.setName(aEnemyChar.info.getName());
        this.info.setLocation(aEnemyChar.info.getLocation());
    }

    public AerialEnemyCharacter duplicateTerrestrial() {
        return new AerialEnemyCharacter(this);
    }

    @Override
    public String getName() {
        return info.getName();
    }

    @Override
    public void setName(String name) {
        this.info.setName(name);
    }

    @Override
    public String getLocation() {
        return this.info.getLocation();
    }

    @Override
    public void setLocation(String location) {
        this.info.setLocation(location);
    }
}
```

Prototype Pattern Solution



Prototype Pattern Solution

```java
public interface Character {

    public Character clone();

    public String getName();

    public void setName(String name);

    public String getLocation();

    public void setLocation(String location);

}
```

```java
public class TerrestrialEnemyCharacter implements Character {
    private Info info = new Info();

    public TerrestrialEnemyCharacter() {
    }

    public TerrestrialEnemyCharacter(
            TerrestrialEnemyCharacter aEnemyChar) {
        this.info.setName(aEnemyChar.info.getName());
        this.info.setLocation(aEnemyChar.info.getLocation());
    }

    // Other methods.....
```

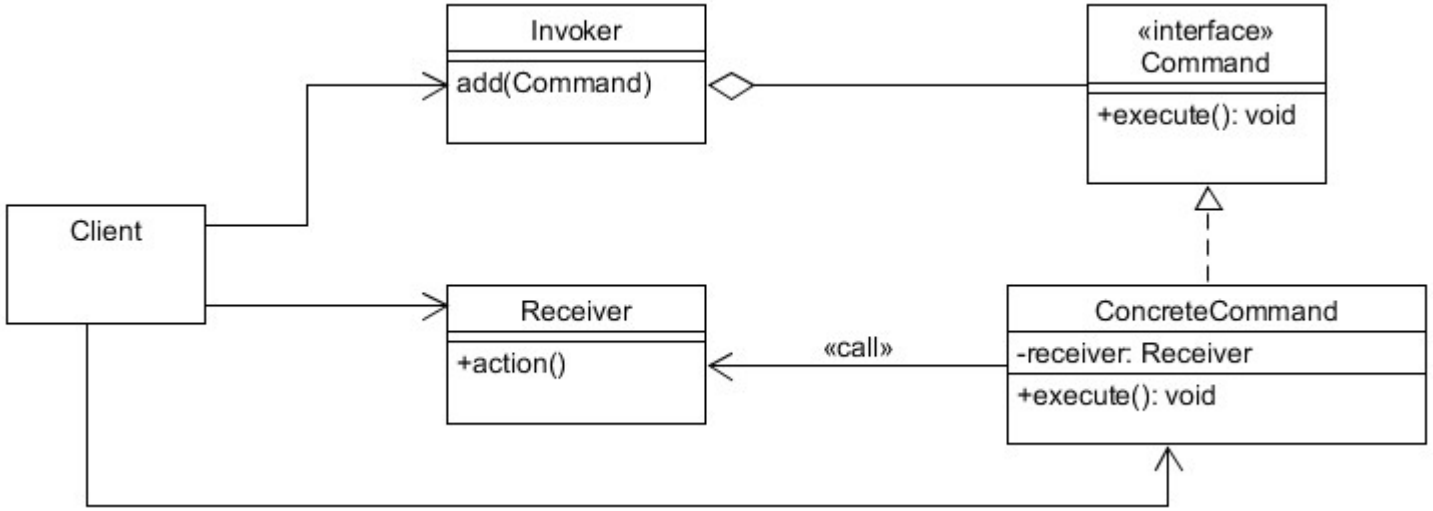
```java
public class AerialEnemyCharacter implements Character {
    private Info info = new Info();

    public AerialEnemyCharacter() {
    }

    public AerialEnemyCharacter(AerialEnemyCharacter aEnemyChar) {
        this.info.setName(aEnemyChar.info.getName());
        this.info.setLocation(aEnemyChar.info.getLocation());
    }

    @Override
    public Character clone() {
        return new AerialEnemyCharacter(this);
    }

    @Override
    public String getName() {
        return info.getName();
    }

    @Override
    public void setName(String name) {
        this.info.setName(name);
    }

    @Override
    public String getLocation() {
        return this.info.getLocation();
    }

    @Override
    public void setLocation(String location) {
        this.info.setLocation(location);
    }
}
```

Prototype Pattern Solution

```
public static void main(String[] args) {
    Character char1 = new AerialEnemyCharacter();
    char1.setName("Hummingbird");
    char1.setLocation("North America");
    System.out.println("char1: " +
            char1.getName() + " " + char1.getLocation());
    Character char2 = char1.clone();
    System.out.println("char2: " +
            char2.getName() + " " + char2.getLocation());
    char1.setName("Eagle"); // *** Changed char1 **/
    System.out.println("char2: " +
            char2.getName() + " " + char2.getLocation());
    System.out.println("char1: " +
            char1.getName() + " " + char1.getLocation());
}
```

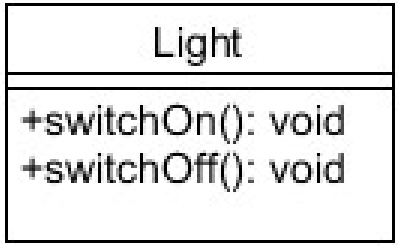
Command pattern (Behavioral)

- The Command pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways.
  - You need callback functionality.
  - Requests need to be handled at variant times or in variant orders.
  - A history of requests is needed.
  - The invoker should be decoupled from the object handling the invocation.
- Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

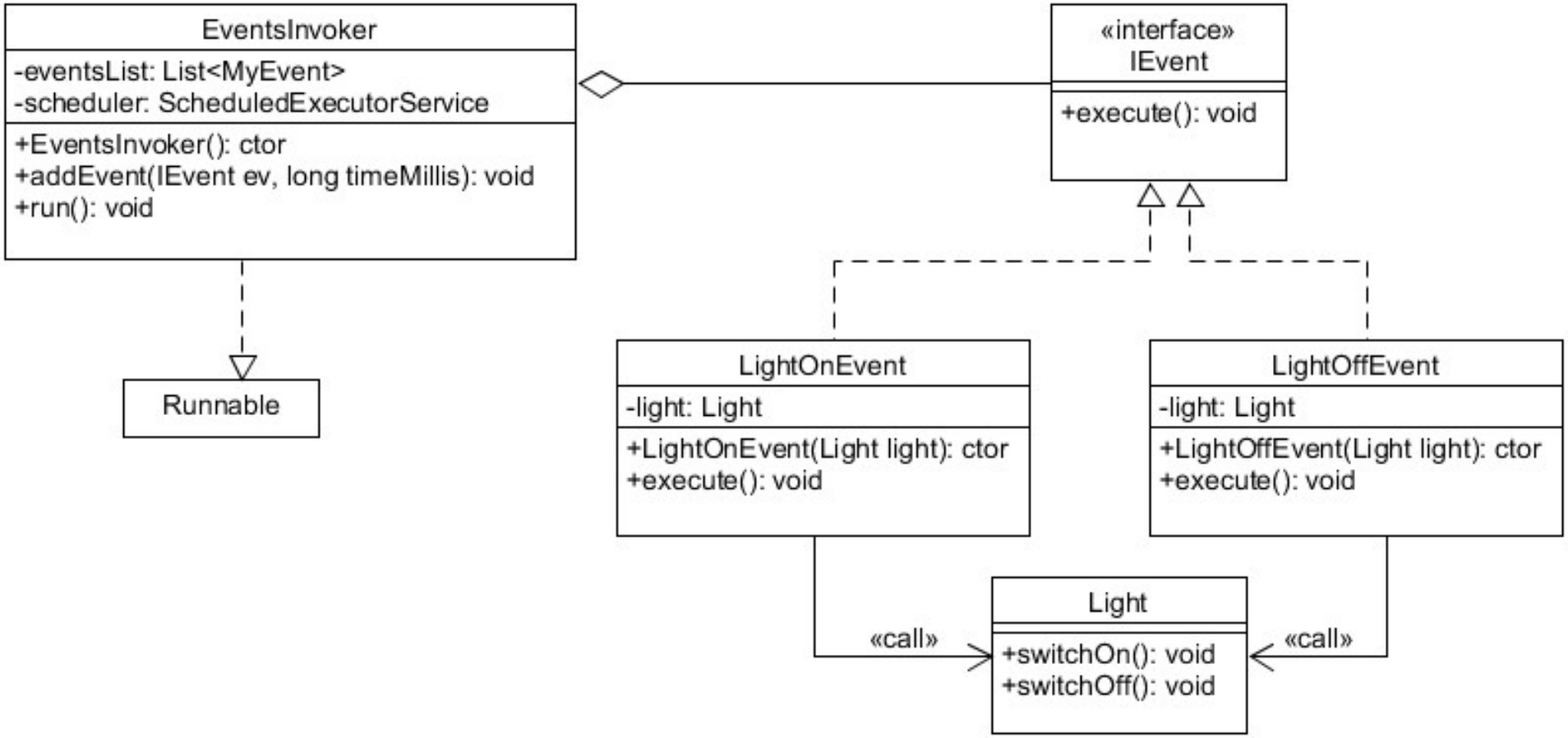


Command pattern (Behavioral)

- Design a system to implement commands at scheduled time. For now, we need to implement two commands one turn the light on while the other to turn the light off.
  - Assume the Light class will actually perform the real operations



Command pattern (Behavioral)



Command pattern (Behavioral)

```
public interface IEvent {
    public void execute();
}
```

```
public class Light {
    public void switchOn() {
        System.out.println("LightOn @ " +
                System.currentTimeMillis());
    }

    public void switchOff() {
        System.out.println("LightOff @ " +
                System.currentTimeMillis());
    }
}
```

```
public class LightOffEvent implements IEvent {
    private Light light;

    public LightOffEvent(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.switchOff();
    }
}
```

```
public class LightOnEvent implements IEvent {
    private Light light;

    public LightOnEvent(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.switchOn();
    }
}
```

Command pattern (EventsInvoker class)

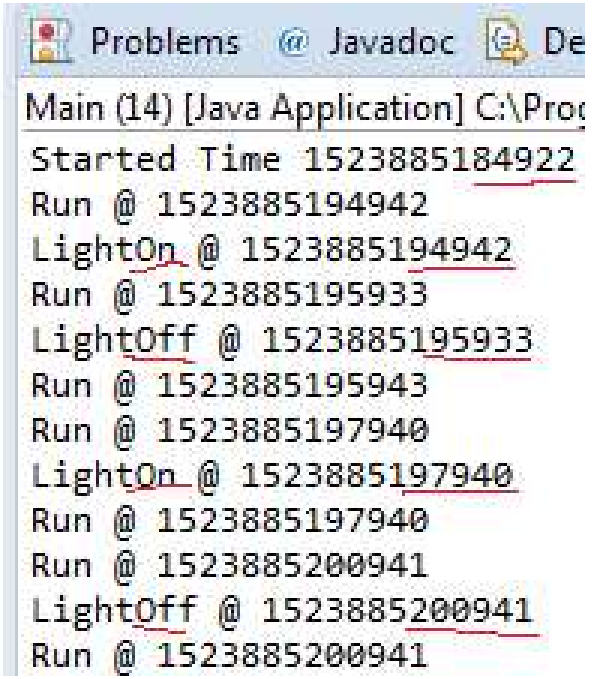
```
public class EventsInvoker implements Runnable {

    // data structure to store events internally
    private static class MyEvent implements Comparable<MyEvent> {
        public long timeMillis;
        public IEvent event;

        public MyEvent(IEvent ev, long timeMillis) {
            this.timeMillis = timeMillis;
            this.event = ev;
        }

        @Override
        public int compareTo(MyEvent other) {
            return (int) (this.timeMillis - other.timeMillis);
        }
    }

    // the events list
    private List<MyEvent> eventsList = new ArrayList<MyEvent>();
    private ScheduledExecutorService scheduler =
            Executors.newSingleThreadScheduledExecutor();

    public EventsInvoker() {
        System.out.println("Started Time " +
                           System.currentTimeMillis());
    }
```

Command pattern (EventsInvoker class Cont.)

```
    public void addEvent(IEvent ev, long timeMillis) {
        synchronized (eventsList) {
            MyEvent event = new MyEvent(ev,
                    System.currentTimeMillis() + timeMillis);
            eventsList.add(event);
            Collections.sort(eventsList);
            if (eventsList.get(0) == event) {
                // the added event is the first to execute
                scheduler.schedule(this,
                        timeMillis,
                        TimeUnit.MILLISECONDS);
            }
        }
    }
    @Override
    public void run() {
        System.out.println("Run @ " + System.currentTimeMillis());

        synchronized (eventsList) {
            Iterator<MyEvent> it = eventsList.iterator();
            while (it.hasNext()) {
                MyEvent e = (MyEvent) it.next();
                if (e.timeMillis > System.currentTimeMillis()) {
                    // All remaining events to happen in future
                    scheduler.schedule(this,
                            e.timeMillis - System.currentTimeMillis(),
                            TimeUnit.MILLISECONDS);
                    break;
                } else { // remove and execute this event
                    it.remove();
                    e.event.execute();
                }
            }
        }
    }
}
```

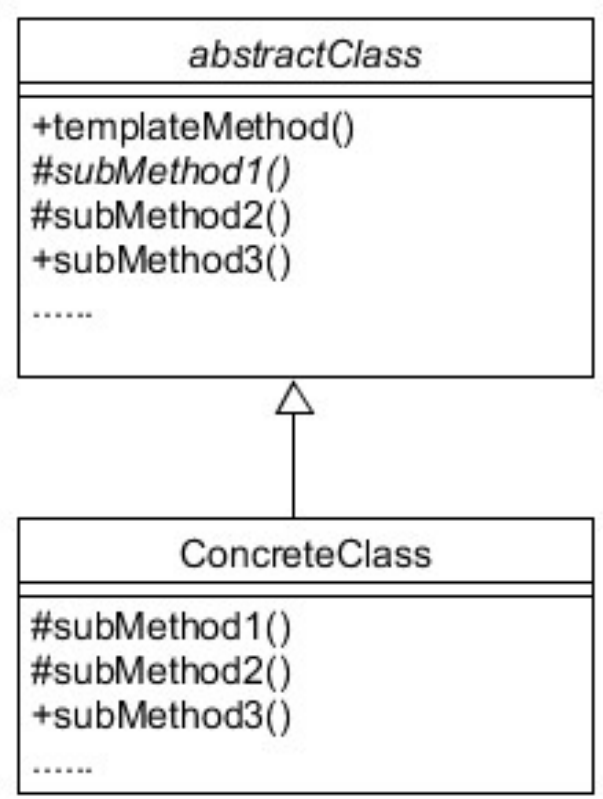
Command pattern (Behavioral)

```
public class Main {

    public static void main(String[] args) {
        EventsInvoker scheduler = new EventsInvoker();
        Light light = new Light();
        scheduler.addEvent(new LightOffEvent(light), 11000); // 2
        scheduler.addEvent(new LightOffEvent(light), 16000); // 4
        scheduler.addEvent(new LightOnEvent(light), 10000); // 1
        scheduler.addEvent(new LightOnEvent(light), 13000); // 3

        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```



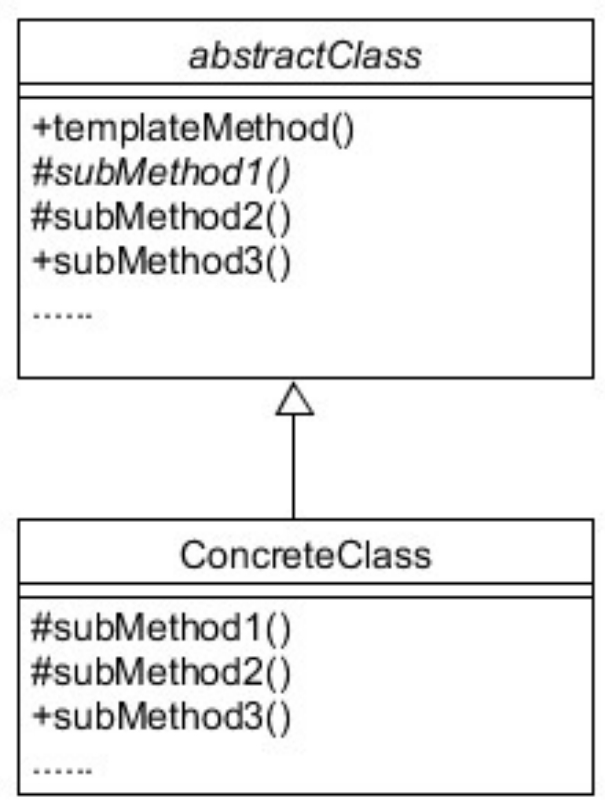
Template Method pattern (Behavioral)

- Template Method is a behavioral design pattern that allows you to defines a skeleton of an algorithm in a base class and let subclasses override the steps without changing the overall algorithm's structure.
- This pattern is used to create a template for an algorithm which is basically a method which specifically has a method that defines an algorithm as a set of steps. One or more of these steps is defined to be abstract and implemented by a subclass. This ensures the algorithms structure stays unchanged, while subclasses provide some part of the implementation.
- Template pattern is good when we have a use case where the structure of an algorithm is the same and we provide different implementations, we can use the template method design pattern and also its a good fit for creating frameworks.

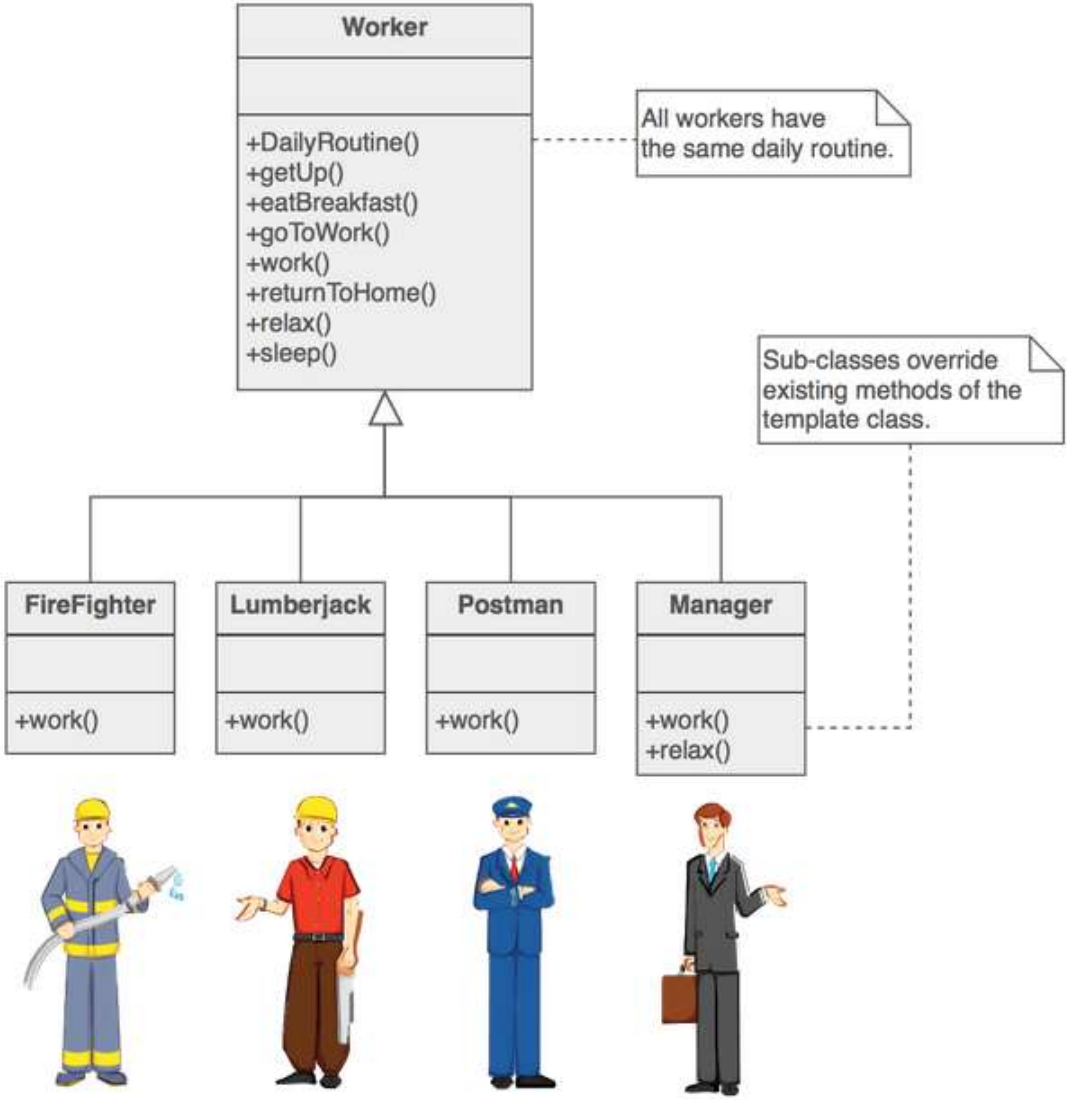


Template Method pattern (Behavioral)

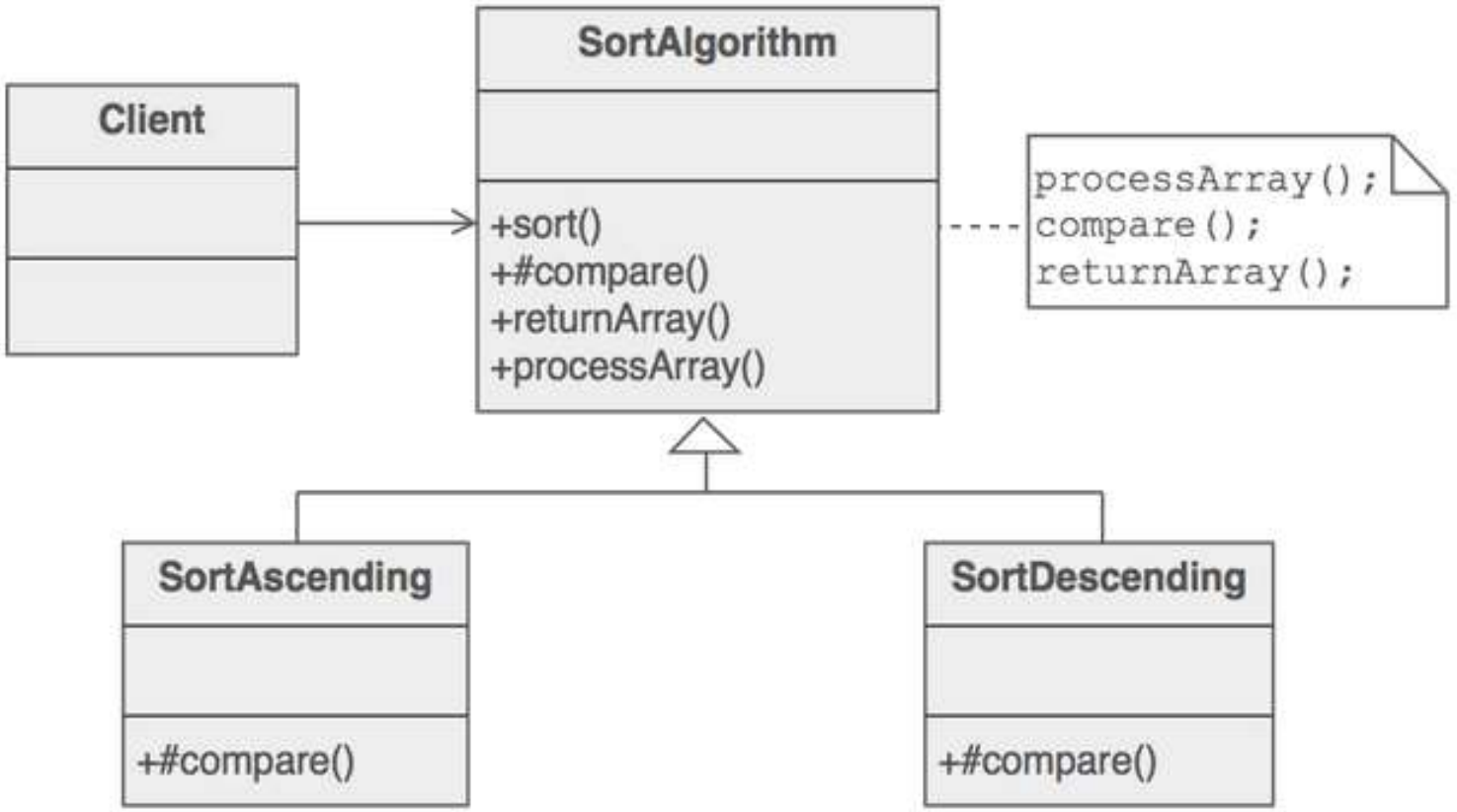
- When broken down, there are four different types of methods used in the parent class:
  - **Concrete methods:** Standard complete methods that are useful to the subclasses. These methods are usually utility methods.
  - **Abstract methods:** Methods containing no implementation that must be implemented in subclasses.
  - **Hook methods:** Methods containing a default implementation that may be overridden in some classes. Hook methods are intended to be overridden, concrete methods are not.
  - **Template methods:** A method that calls any of the methods listed above in order to describe the algorithm without needing to implement the details.



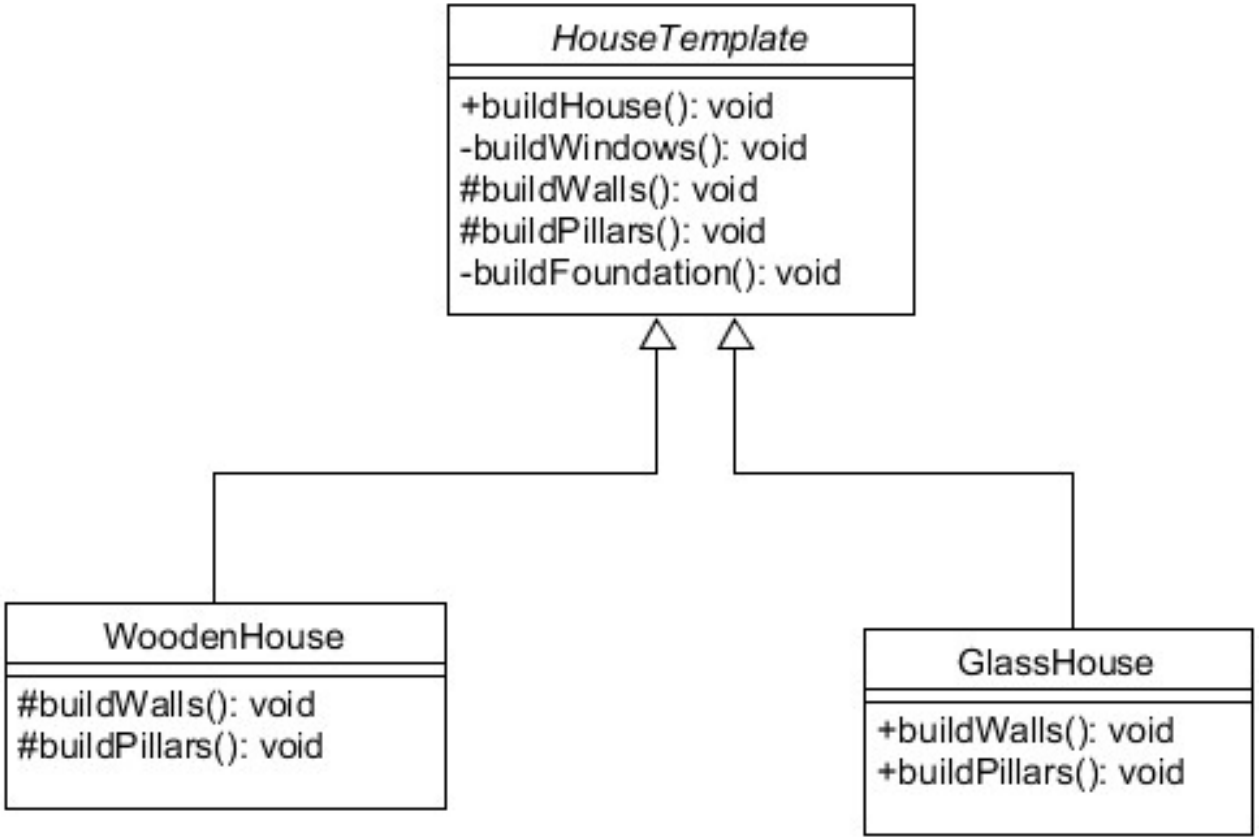
Template Method pattern (Example-1)



Template Method pattern (Example-2)



Template Method pattern (Example-3)



Template Method pattern (Example-3)

```
public abstract class HouseTemplate {

    // template method, final so subclasses can't override
    public final void buildHouse() {
        buildFoundation();
        buildPillars();
        buildWalls();
        buildWindows();
        System.out.println("House is built.");
    }

    // default implementation
    private void buildWindows() {
        System.out.println("Building Glass Windows");
    }

    // methods to be implemented by subclasses
    protected abstract void buildWalls();

    protected abstract void buildPillars();

    private void buildFoundation() {
        System.out.println("Building foundation with cement,iron rods and sand");
    }
}
```

Template Method pattern (Example-3)

```java
public class WoodenHouse extends HouseTemplate {

    @Override
    protected void buildWalls() {
        System.out.println("Building Wooden Walls");
    }

    @Override
    protected void buildPillars() {
        System.out.println("Building Pillars"
                + " with Wood coating");
    }

}
```

```java
public class GlassHouse extends HouseTemplate {

    @Override
    public void buildWalls() {
        System.out.println("Building Glass Walls");
    }

    @Override
    public void buildPillars() {
        System.out.println("Building Pillars"
                + " with glass coating");
    }

}
```

```java
public class HousingClient {

    public static void main(String[] args) {

        HouseTemplate houseType = new WoodenHouse();

        houseType.buildHouse();
        System.out.println("************");
        houseType = new GlassHouse();
        houseType.buildHouse();
    }

}
```