

COMP2100/6442

Software Design Methodologies / Software Construction

Overview

Software Construction




Bernardo Pereira Nunes

Outline

- Overview Software Construction
- UML
 - Class Diagram
 - Inheritance
 - Polymorphism
- Design Patterns
 - Factory
 - Singleton
 - Observer

Intro to Software Construction [1]

You know what “construction” means when it’s used outside software development. “Construction” is the work “construction workers” do when they build a house, a school, or a skyscraper. When you were younger, you built things out of “construction paper.” In common usage, “construction” refers to the process of building. The construction process might include some aspects of planning, designing, and checking your work, but mostly “construction” refers to the hands-on part of creating something.



It is not only coding, i.e., a mechanical translation (design to a programming language), it also requires: creativity, judgement and knowledge.

Intro to Software Construction

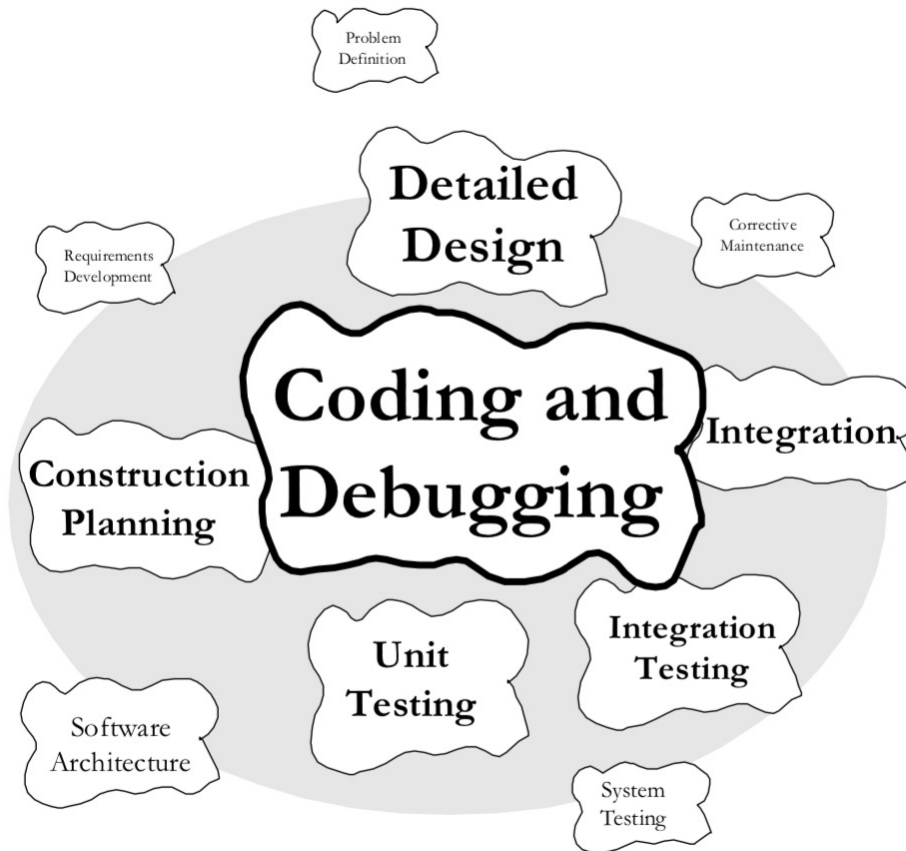


Fig 1. High-level views of construction activities.

Intro to Software Construction

Software Construction is related to all areas in Software Engineering, but it is mostly related to Software Design and Testing.



Software Construction Fundamentals^[2]

- Minimizing Complexity
- Anticipating change
- Constructing for verification
- Reuse
- Standards in construction

[2] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0 (3rd. ed.). IEEE Computer Society Press, Washington, DC, USA..

Intro to Software Construction

Some specific tasks^[1]:

- Determining how your code will be tested
- Designing and writing classes and routines
- Creating and naming variables and named constants
- Selecting control structures and organizing blocks of statements
- Unit testing, integration testing, and debugging your own code
- Reviewing other team members' low-level designs and code and having them review yours
- Polishing code by carefully formatting and commenting it
- Integrating software components that were created separately
- Tuning code to make it smaller and faster

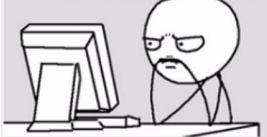
[1] Steve McConnell. 2004. Code Complete, Second Edition. Microsoft Press, USA.

Intro to Software Construction

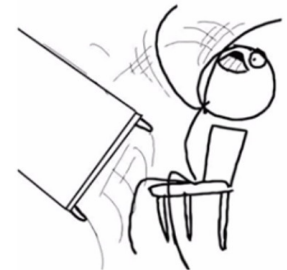
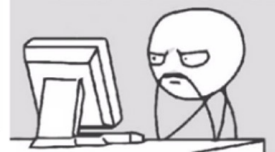
- Minimizing Complexity:
 - Code is simple and readable
 - Use of standards
 - Modular design
 - ...
- Anticipating Change:
 - Learn it now: **software changes overtime!**
 - Be prepared for it!
 - Develop a software that is easy to extend and changes will not break the underlying structure of the software

The fun of learning programming

CH 1: A BUNCH
OF COMPLEX CODE



CH 2: WHY YOU DON'T
NEED TO KNOW CH 1



Cem Paya
@randomoracle

Pro-tip: If colleagues complain your C code is "unreadable" try using these variable names:

- char broiled;
- double burger;
- short cake;
- float icecream;
- long story;
- signed sincerely;

6:41 PM · 08 Jun 19 · [Twitter Web App](#)

62 Retweets 246 Likes

Intro to Software Construction

- Constructing for Verification:
 - “Constructing for verification means building software in such a way that faults can be readily found by the software engineers writing the software as well as by the testers and users during independent testing and operational activities.”^[2]
 - Unit testing
 - Automated testing
 - Minimizing complexity to help verification
 - ...
- Reuse: (construction for reuse and construction with reuse)
 - The systematic use of existing assets (libraries, components, source code, ...) to build new software
 - It usually involves: parameterization/generics, design patterns, encapsulation, documentation
 - Enable productivity, quality and possibly reduce construction costs

Intro to Software Construction

- Standards in construction:
 - External or Internal Standards
 - Quality, efficiency, reduce cost, and improve security
 - Language standards
 - Coding standards (naming conventions, indentation, layout)
 - Platforms (interface standards)
 - Tools (for modelling/diagrams, such as UML)

Personal comment: Be careful, I heard once: Standards are so good that there are many! Try to use W3C, ISO, IEEE, ... and other known standards.

Planning is also important in software construction as it will help you to decide when and **how parts of the system will be built, integrated, tested**. It will help you decide **which standards to use, allocate tasks to each member of the software construction team, and reduce the costs**.

Intro to Software Construction

- Software Construction Tools:
 - Integrated Development Environments (IDE)
 - Help developers during software construction
 - It may affect quality and efficiency of software construction
 - It offers a number of tools to basic code editing, compilation and error detection, integration with source code control, test, debugging, refactoring, ...
 - Unit testing tools
 - Profiling and Performance Analysis Tools
 - Support code tuning
 - Each statement is executed or how much time the program spends on each statement or execution path.

Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system. [3]

What is UML?

- Language for modeling systems
- Easy (“natural”) to understand and use
- Used for visualizing, specifying, constructing and documenting artifacts of software
- It allows understanding a system from various perspectives
- UML is also used for documenting (e.g., requirements, design, architecture, ...)
- UML **does not** provide algorithmic details

Sometimes used before coding (**forward design**)

Sometimes used after coding (documentation purposes) (**backward design**)



[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. 2005. Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). Addison-Wesley Professional.

Unified Modeling Language (UML)

UML – Building Blocks

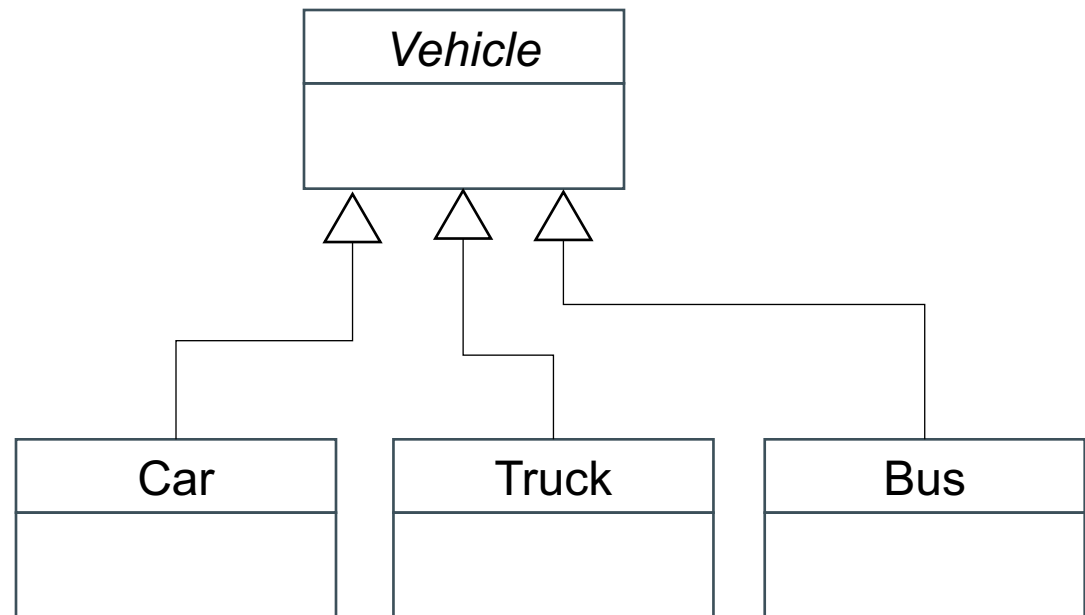
- Things (abstraction/elements that will be modelled)
 - Relationships (relate things/tie them together)
 - Diagrams (graphical representation of a set of elements (things and relationships))
-
- Diagrams:
 - **Class Diagram**
 - Sequence Diagram
 - Use Case Diagram
 - ... many others!



Class Diagram

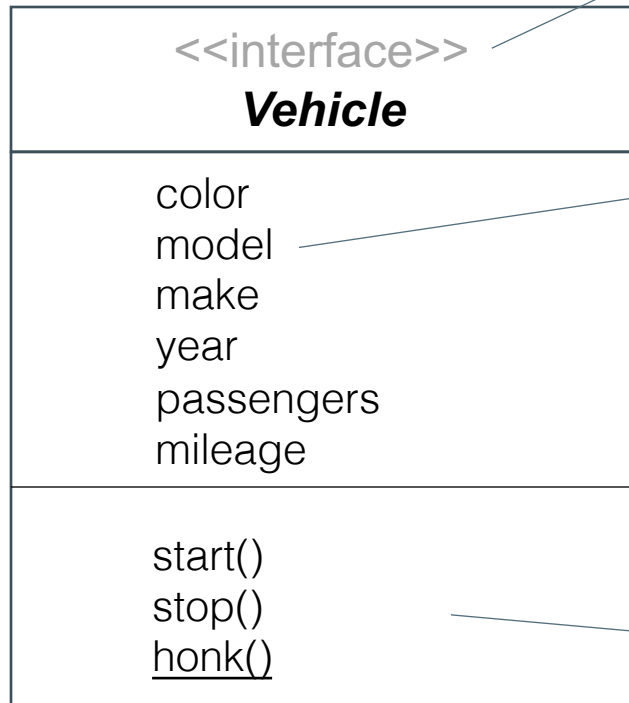
Object-Oriented Approach – Abstraction / Real-World Entities

“A class diagram shows a set of classes, interfaces, and collaborations and their relationships.”^[3]



Extra reference: <https://www.uml-diagrams.org/class-diagrams-overview.html>

Class Diagram



Name (bold, capitalized, centered)
If it is an interface, add <<interface>>
if it is an abstract class, use italics (e.g., *Vehicle*)

Attributes (not bold, lowercase, left-aligned)
It is optional.

[visibility] attributeName [[multiplicity]] [:type] [=initial value][{property}]

Examples:

+ year : Integer
- make : String {read only (final)}
- passengers : Person [0..N]
+ mileage : Double = 0.0

initial value

Operations

[visibility] operationName [(parameter-list)] [:return type] [{property}]
Use underlined for static methods (and attributes)

Examples:

+ start() : Boolean
- honk(level : Int) : void

*It presents some relevant items only.

Class Diagram

Visibility

	Modifier	Class	Package	Subclass	World
+	public	yes	yes	yes	yes
#	protected	yes	yes	yes	no
~	package	yes	yes	no	no
-	private	yes	no	no	no

*if no modifier is set, the default is **package**

Relationships

Generalizations (inheritance) – indicates a relationship between a more generalized class to more-specialized classes (**is-a; is-like-a**). The specialized class inherits all attributes, operations, and relationships defined in the more generalized class (parent-child relationship).

Associations – indicates **structural relationships** between instances. For example, an **employee works for an organization**. Each end of the relationship has properties (e.g., multiplicity).

- **Dependencies** – indicates runtime relation between classes (an object affects another object); **one depends on another**; a class uses objects (parameters) of another class (**uses-a**).
- **Aggregation** – indicates **is-part-of / has-a** relationships; object **can exist outside the whole**.
- **Composition** – indicates **whole-part** relations (**is-entirely-made-of**); the **parts lives and dies with the whole** (stronger form of aggregation)

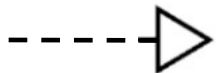
Relationships: Inheritance

Generalizations (inheritance) – indicates a relationship between a more generalized class to more-specialized classes (is-a; is-like-a). The specialized class inherits all attributes, operations, and relationships defined in the more generalized class (parent-child relationship).

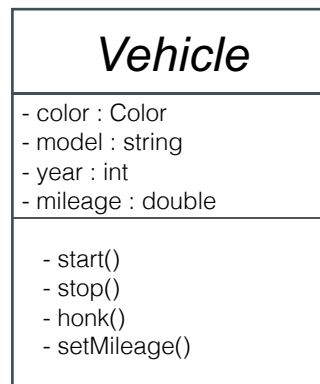
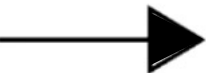
(Abstract class)



(Interface class)

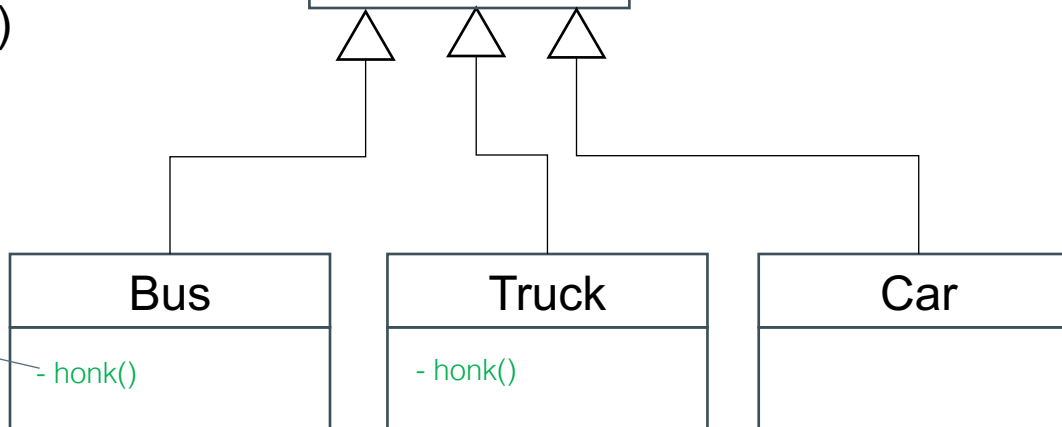


(Another class)



Superclass or
Parent Class

Overriding

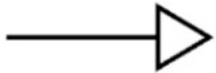


Subclass or
Child class

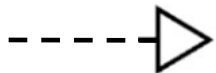
Relationships: Inheritance

Generalizations (inheritance) – indicates a relationship between a more generalized class to more-specialized classes (is-a; is-like-a). The specialized class inherits all attributes, operations, and relationships defined in the more generalized class (parent-child relationship).

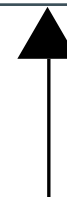
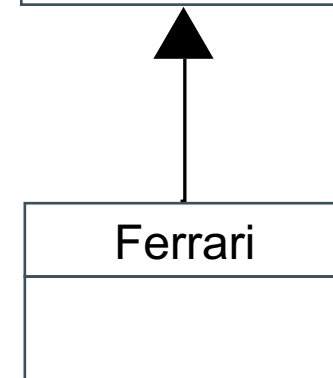
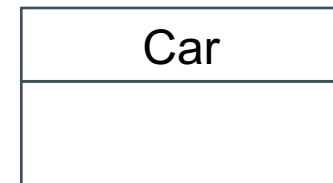
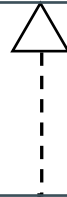
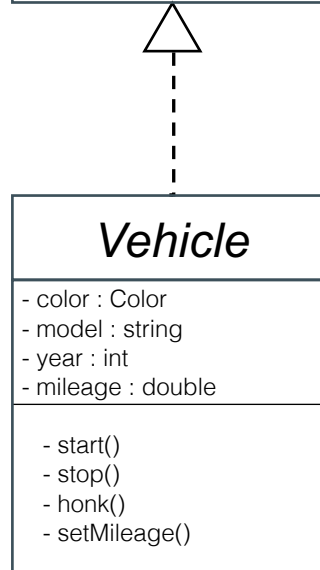
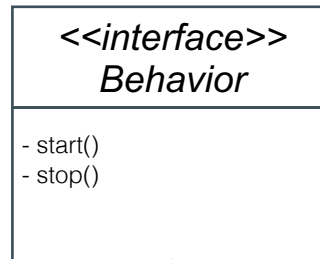
(Abstract class)



(Interface class)



(Another class)

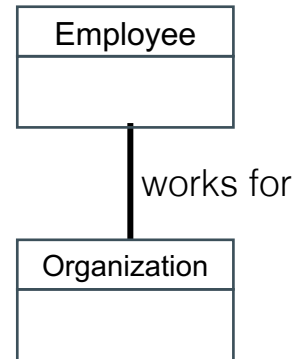


Relationships

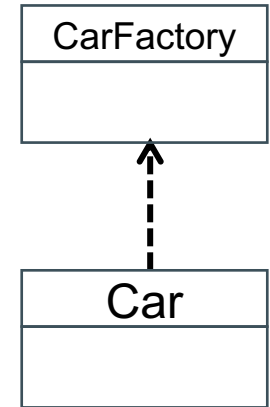
Associations – indicates structural relationships between instances. For example, an employee works for an organization. Each end of the relationship has properties (e.g., multiplicity).

- **Dependencies** – indicates runtime relation between classes; one depends on another; some class uses objects (parameters) of another class (**uses-a/depends-on/instantiate**).
- **Aggregation** – indicates **is-part-of** relationships; object can exist outside the whole.
- **Composition** – indicates **whole-part** relations (**is-entirely-made-of**); the parts lives and dies with the whole.

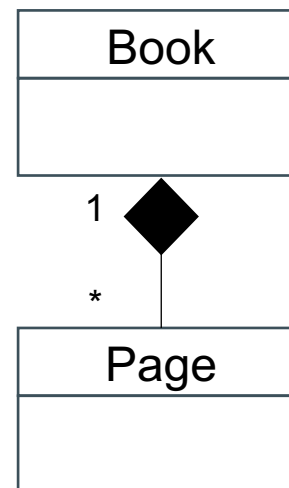
Association: Employee works for Organization



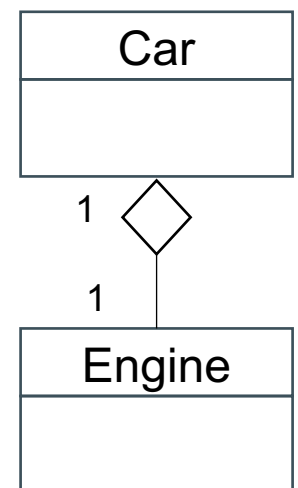
Dependency: CarFactory depends-on Car



Composition: Book is-entirely-made-of Pages



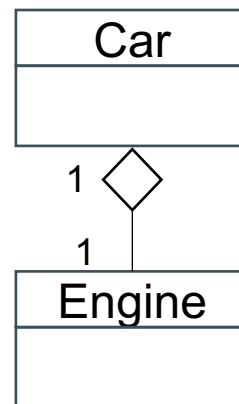
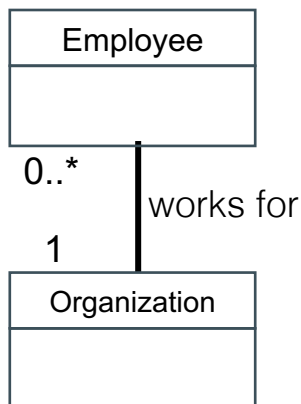
Aggregation: Engine is-part-of Car



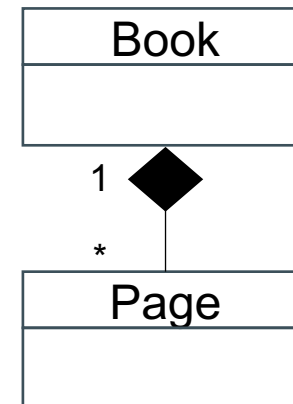
Multiplicity

Associational relationships

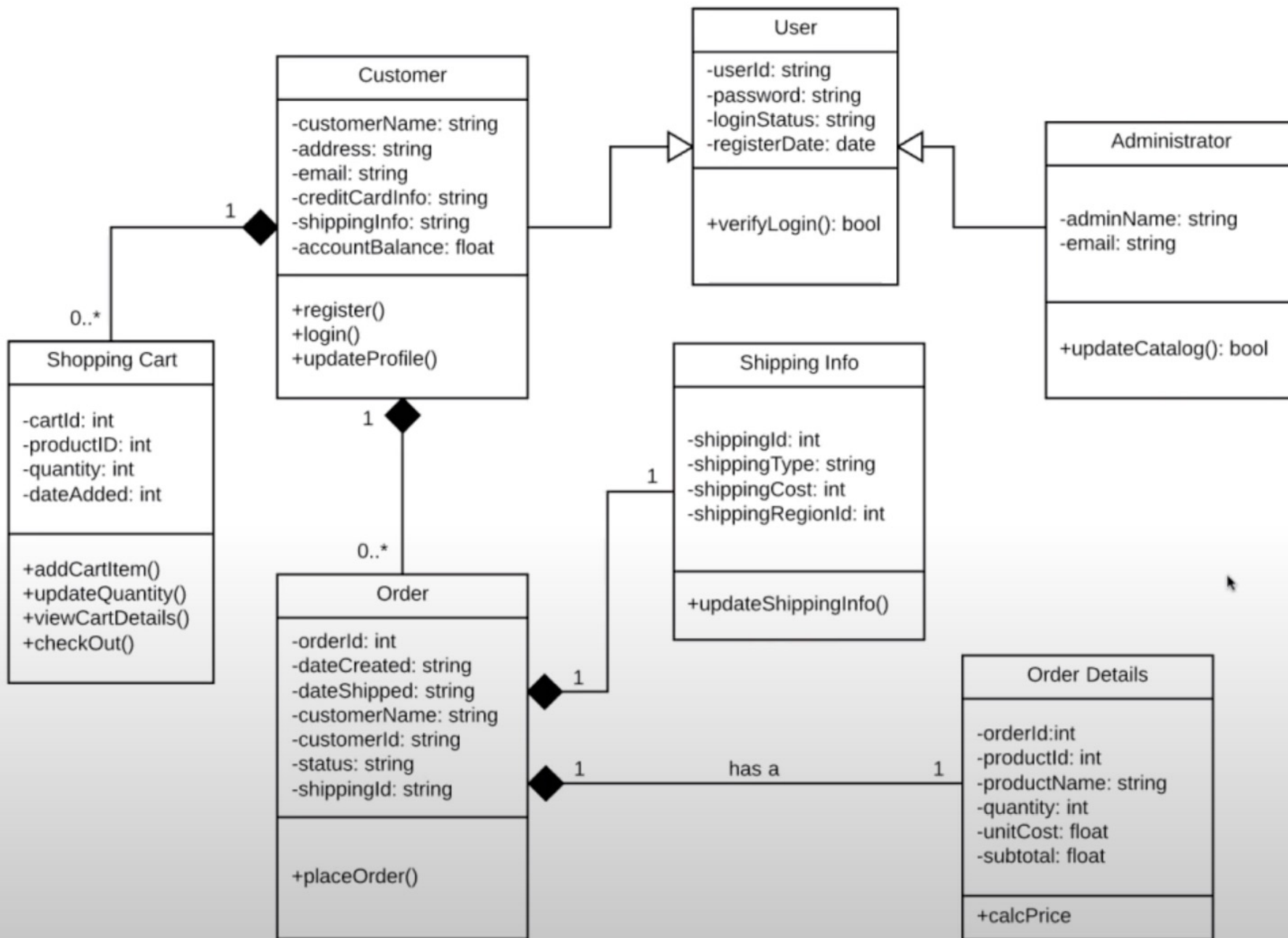
Multiplicity	Meaning
0..1	zero or one instance. The notation n.. m indicates n to m instances.
0..* or *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance



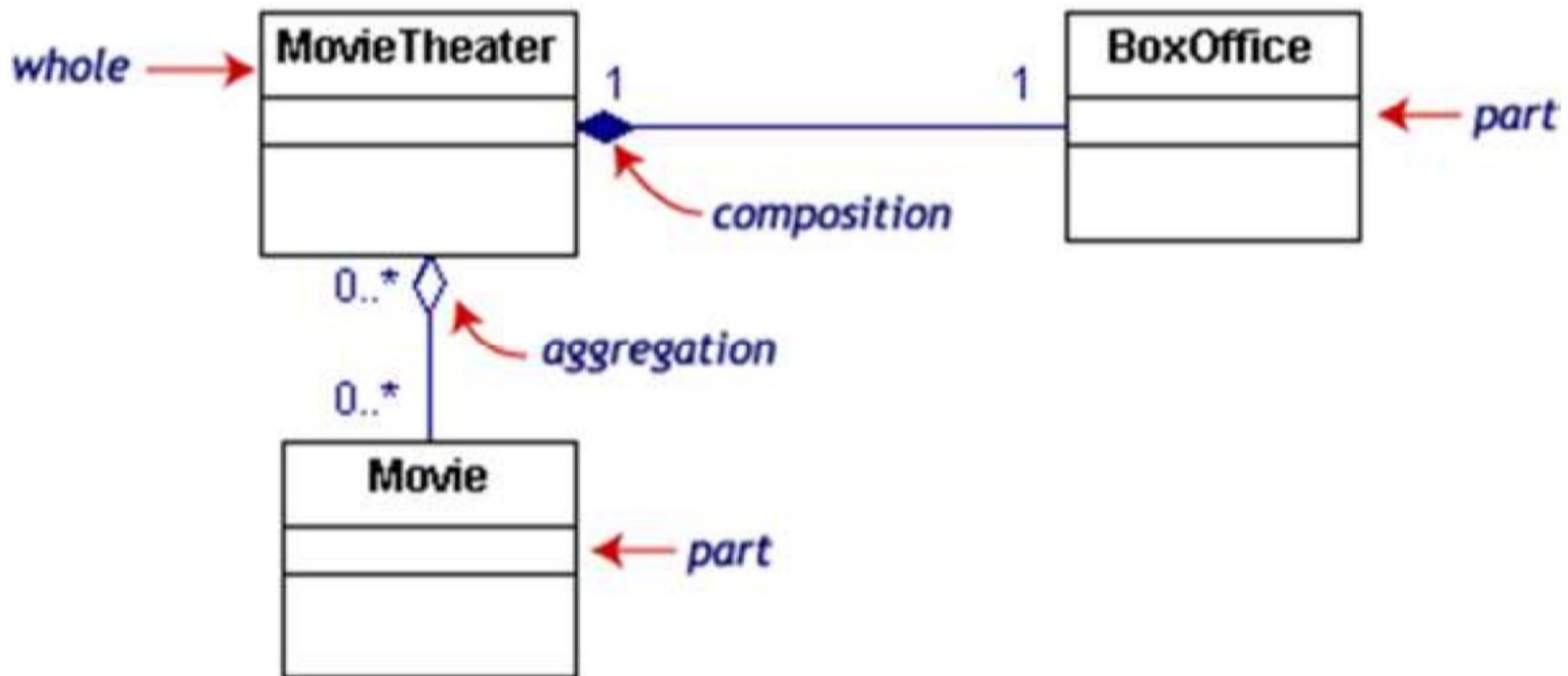
one-to-one



one-to-many



Example



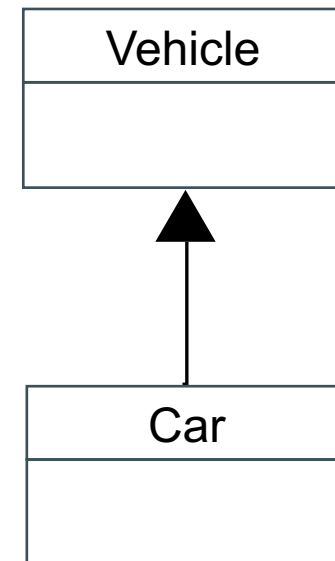
MovieTheater is destroyed, BoxOffice is also destroyed
Movie still exists

Java Inheritance

- Subclass inherits features (attributes and operations) from Superclass
- Reusability (reuse instead of rewriting operations and attributes in a subclass)
- Subclass can add new attributes and methods (extend)
- “IS-A” or Parent-Child relationship

```
class Vehicle {  
    //TODO  
}
```

```
class Car extends Vehicle{  
    //TODO  
}
```



Java Inheritance

- A common example using inheritance:

```
class Calc { //superclass
    double result;

    public void add(double x, double y) {
        result = x + y;
        System.out.println("ADD result:" + result);
    }

    public void mult(double x, double y) {
        result = x * y;
        System.out.println("MULT result:" + result);
    }
}
```

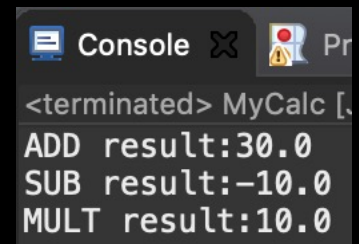
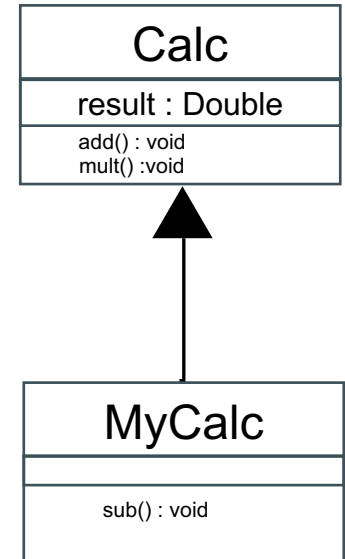

Java Inheritance

- A common example using inheritance:

```
public class MyCalc extends Calc { // subclass
    public void sub(int x, int y) {
        result = x - y;
        System.out.println("SUB result:" + result);
    }
}
```

```
/* driver */
public static void main(String args[]) {
```

```
    MyCalc mc = new MyCalc();
    mc.add(10, 20); //mc inherits add
    mc.sub(10, 20);
    mc.mult(2, 5); //mc inherits mult
}
```



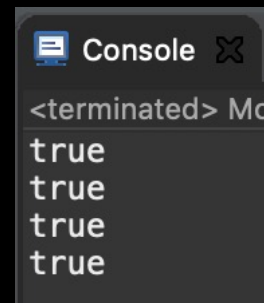
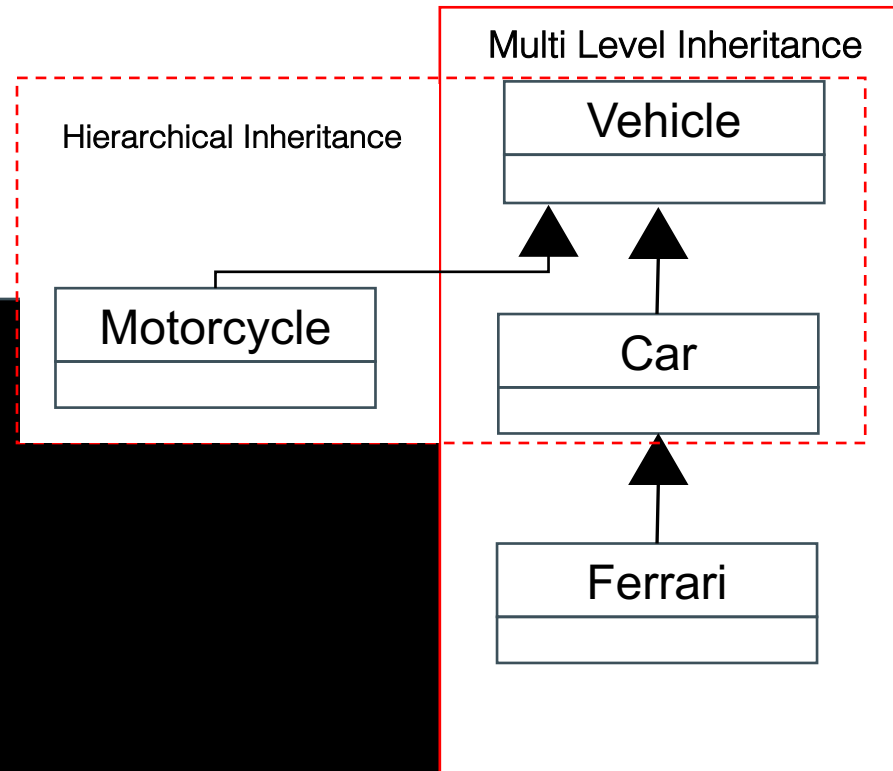
The screenshot shows a console window with the following output:

```
<terminated> MyCalc [.
ADD result:30.0
SUB result:-10.0
MULT result:10.0
```

Java Inheritance

- Another example using inheritance:

```
class Vehicle {  
}  
  
class Car extends Vehicle {  
}  
  
class Ferrari extends Car {  
}  
  
public class Motorcycle extends Vehicle {  
  
/* driver */  
public static void main(String args[]) {  
    Vehicle    v = new Vehicle();  
    Car        c = new Car();  
    Ferrari    f = new Ferrari();  
    Motorcycle m = new Motorcycle();  
  
    System.out.println(v instanceof Vehicle);  
    System.out.println(c instanceof Car);  
    System.out.println(f instanceof Ferrari);  
    System.out.println(m instanceof Motorcycle);  
    //m instanceof Vehicle?  
  
}  
}
```



```
Console X  
<terminated> Mot  
true  
true  
true  
true
```

Java Overriding

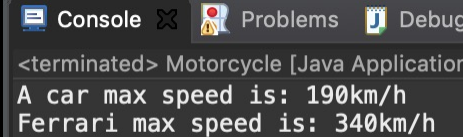
- Inheritance allows the subclass to override an existing superclass method (*unless it is final*)
- It redefines superclass behavior to meet subclass requirements

```
class Car extends Vehicle {
    void maxSpeed() {
        System.out.println("A car max speed is: 190km/h");
    }
}

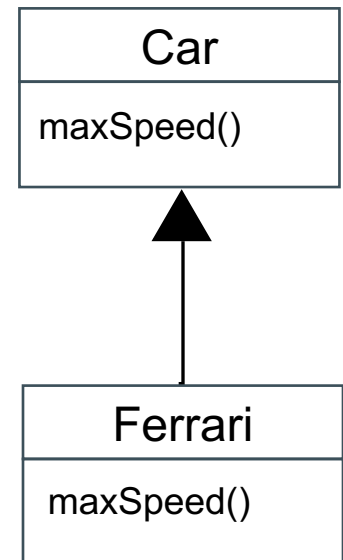
class Ferrari extends Car {
    void maxSpeed() {
        System.out.println("Ferrari max speed is: 340km/h");
    }
}

//driver method
public static void main(String args[]) {
    Car c1 = new Car();
    Car f1 = new Ferrari();

    c1.maxSpeed();
    f1.maxSpeed();
    //compile-time, check on reference type (Car has a maxSpeed() method)
    //runtime, check on the object type (execute Ferrari's maxSpeed())
}
```



```
<terminated> Motorcycle [Java Application]
A car max speed is: 190km/h
Ferrari max speed is: 340km/h
```



Polymorphism

“Polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages.” **poly** -> many / **morphs**-> forms

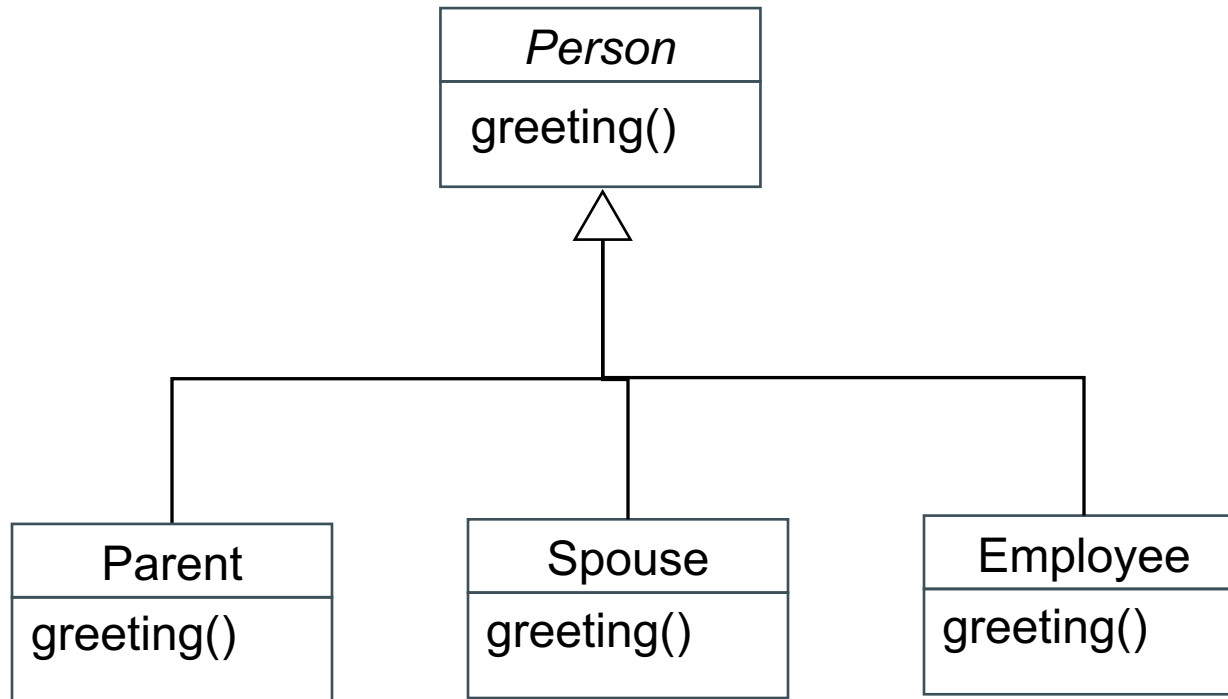


Copyright Brian Freiermuth, insituexsitu.com

All frogs are of the **same specie**
– *strawberry poison dart frog*
(color polymorphism).

Polymorphism

Different behaviors in different situations



(*UML does not represent polymorphism. The diagram represents inheritance only).

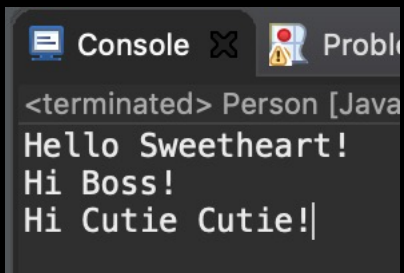
Polymorphism

```
import java.util.ArrayList;

public abstract class Person {
    void greeting() {}

    public static void main(String args[]) {
        ArrayList<Person> ap = new ArrayList<Person>();
        ap.add(new Spouse()); //upcasting
        ap.add(new Employee()); //upcasting
        ap.add(new Parent()); //upcasting

        for (Person p : ap)
            p.greeting();
    }
}
```



Console

```
<terminated> Person [Java
Hello Sweetheart!
Hi Boss!
Hi Cutie Cutie!|
```

```
class Spouse extends Person{
    void greeting(){
        System.out.println("Hello Sweetheart!");
    }
}

class Employee extends Person{
    void greeting(){
        System.out.println("Hi Boss!");
    }
}

class Parent extends Person{
    void greeting(){
        System.out.println("Hi Cutie Cutie!");
    }
}
```

Inheritance vs Polymorphism

Some differences that may help to understand both concepts:

Inheritance

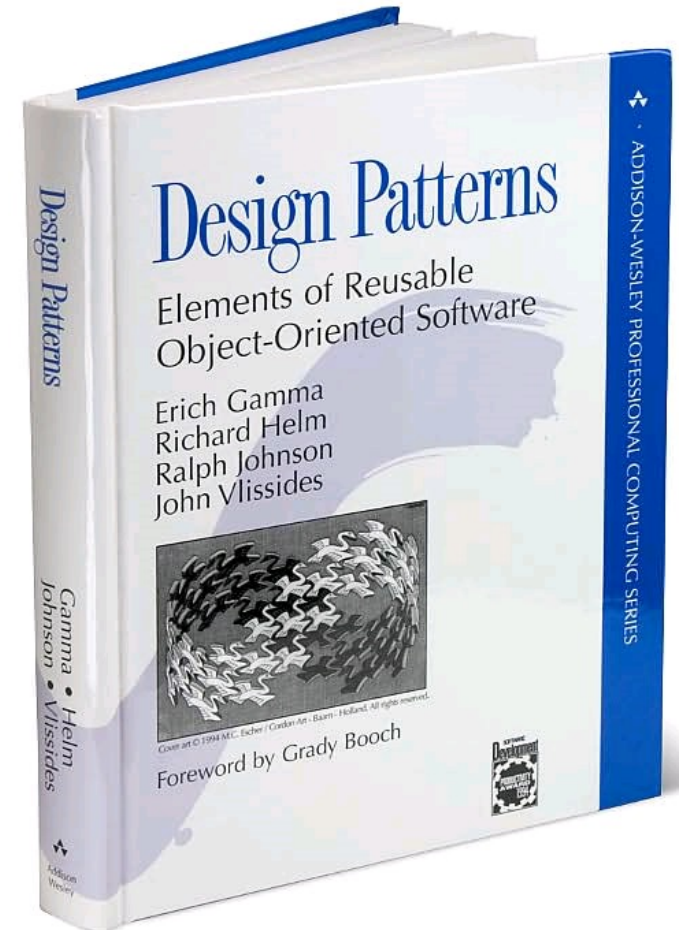
- Reusability (inherit and reuse parent class features [operations and attributes])
- Classes (new classes are created based on the parent class)

Polymorphism

- Object decides which form to take (at compile-time or at runtime)
- Applied to operations (can take different behaviors/forms)

Design Patterns (Gang of Four [GoF])

“Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. **Your design should be specific to the problem at hand but also general enough to address future problems and requirements.** You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a **reusable and flexible design is difficult if not impossible to get "right" the first time.** Before a design is finished, they usually try to reuse it several times, modifying it each time.” [GoF]



[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., USA.

Design Patterns

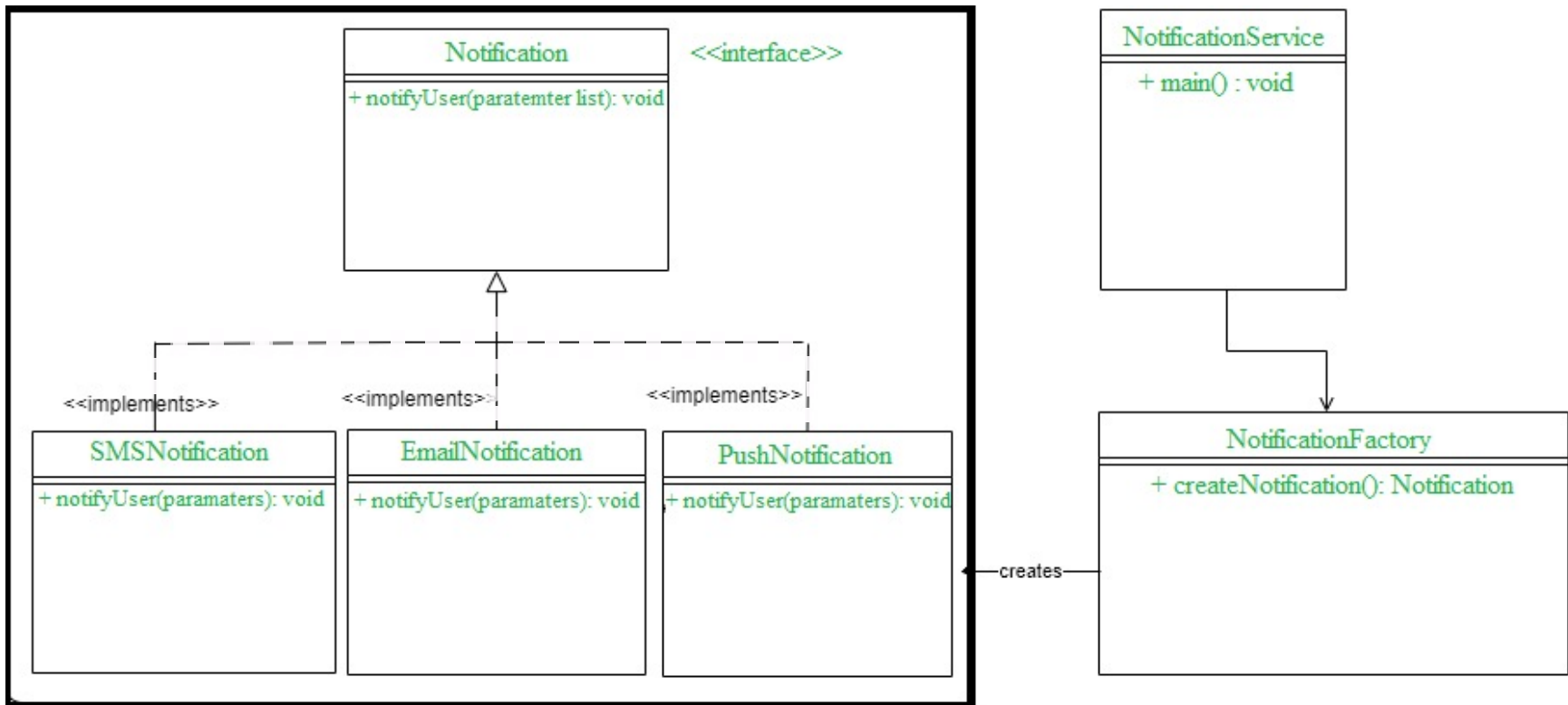
- [GoF] “Reuse solutions that have worked ... in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts.”
- [GoF] “goal is to capture design experience in a form that people can use effectively.”

Design Patterns: Factory Method [GoF]

■ Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.
- Use the Factory Method pattern when:
 - a class can't anticipate the class of objects it must create
 - a class wants its subclasses to specify the objects it creates
 - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Design Patterns: Factory Method



Design Patterns: Factory Method

```
public interface Notification {  
    void notifyUser();  
}  
  
public class SMSNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending an SMS Notification!");  
    }  
}  
  
public class PushNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending a Push Notification!");  
    }  
}  
  
public class EmailNotification implements Notification{  
    @Override  
    public void notifyUser() {  
        System.out.println("Sending an E-Mail Notification!");  
    }  
}
```

Design Patterns: Factory Method

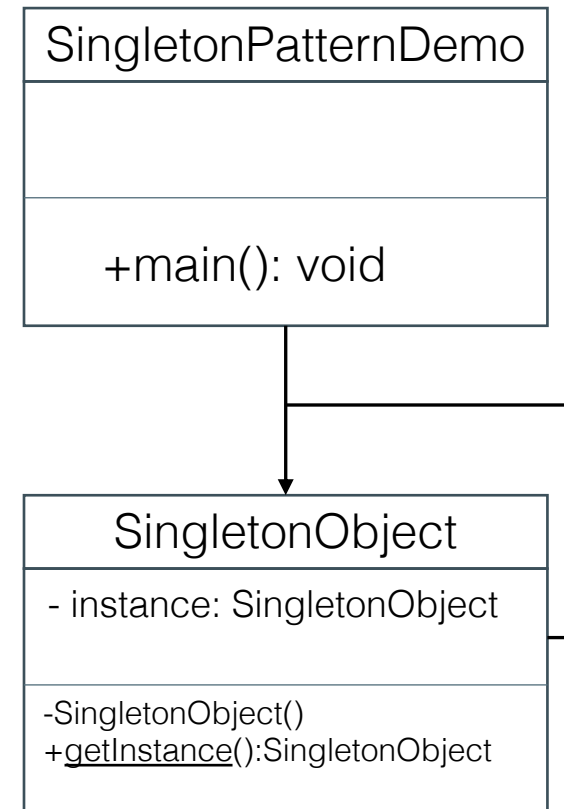
```
public class NotificationFactory {  
    public Notification createNotification(String msg, String c)  
    {  
        String channel = c;  
        //if notification type is not defined, randomly choose one  
        //Another Example: if it was a game, we could balance the number of enemies  
        //or depending on the difficulty level, create different enemies, etc.  
        if (channel == null || channel.isEmpty()) {  
            List<String> l = Arrays.asList("SMS", "EMAIL", "PUSH");  
            Random rand = new Random();  
            channel = l.get(rand.nextInt(l.size()));  
        }  
  
        if ("SMS".equalsIgnoreCase(channel)) {  
            return new SMSNotification();  
        }  
        else if ("EMAIL".equalsIgnoreCase(channel)) {  
            return new EmailNotification();  
        }  
        else if ("PUSH".equalsIgnoreCase(channel)) {  
            return new PushNotification();  
        }  
  
        return null;  
    }  
}
```

Design Patterns: Factory Method

```
public class NotificationService {  
  
    public static void main(String[] args)  
    {  
        NotificationFactory notificationFactory = new NotificationFactory();  
        Notification notification = notificationFactory.createNotification("Hi,  
this is my notification!", "");  
  
        notification.notifyUser();  
    }  
}
```

Design Patterns: Singleton [GoF]

- Intent
 - Ensure a class only has one instance, and provide a global point of access to it.
- Use the Singleton pattern when:
 - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
 - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
 - Examples: File System, print queue, database connections, configuration settings...

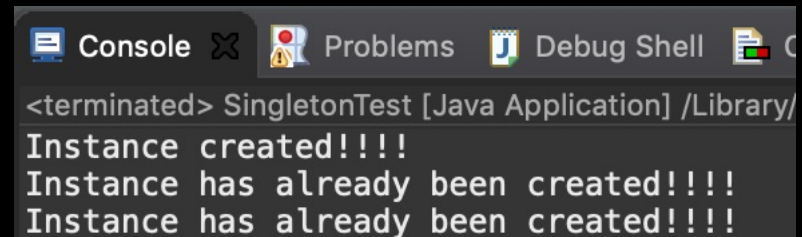


Design Patterns: Singleton

```
public class SingletonConnection {  
  
    //note that the constructor and instance variable are private  
    private static SingletonConnection instance = null;  
    private SingletonConnection(){};  
  
    //Note that this is the only method that can be accessed  
    public static SingletonConnection getInstance(){  
        if(instance == null) {  
            System.out.println("Instance created!!!!");  
            instance = new SingletonConnection();  
        }  
        else  
            System.out.println("Instance has already been created!!!!");  
  
        return instance;  
    }  
}
```


Design Patterns: Singleton

```
public class SingletonTest {  
  
    public static void main(String args[]) {  
        //We cannot instantiate it!  
        //SingletonConnection db = new SingletonConnection();  
        SingletonConnection.getInstance();  
        SingletonConnection.getInstance();  
        SingletonConnection.getInstance();  
    }  
}
```



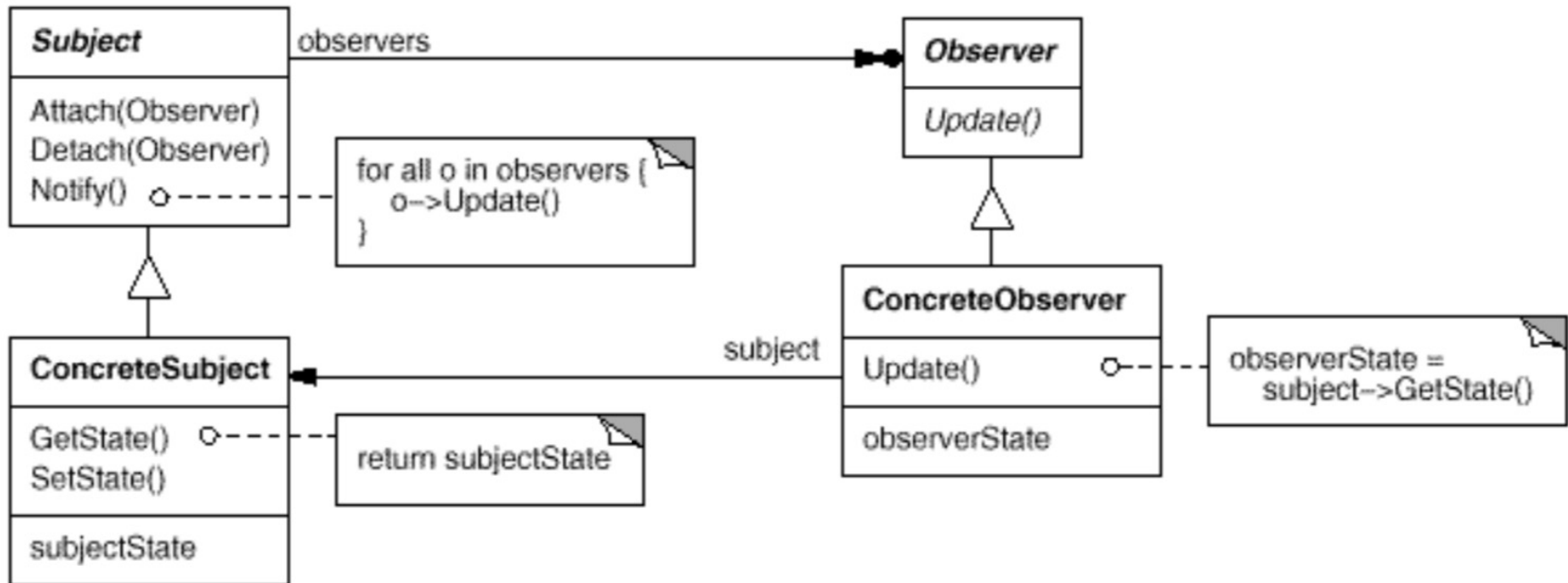
Console Problems Debug Shell

<terminated> SingletonTest [Java Application] /Library/
Instance created!!!!
Instance has already been created!!!!
Instance has already been created!!!!

Design Patterns: Observer [GoF]

- Intent
 - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Use the Observer pattern when:
 - an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
 - a change to one object requires changing others, and you don't know how many objects need to be changed.
 - an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Design Patterns: Observer [GoF]





```
public interface Subject { //interface Subject
    public void attach(Observer observer);
    public void detach(Observer observer);
    public void notifyAllObservers();
}

import java.util.ArrayList;
public class Place implements Subject{//class Place
    private ArrayList<Observer> observers;
    private String name;

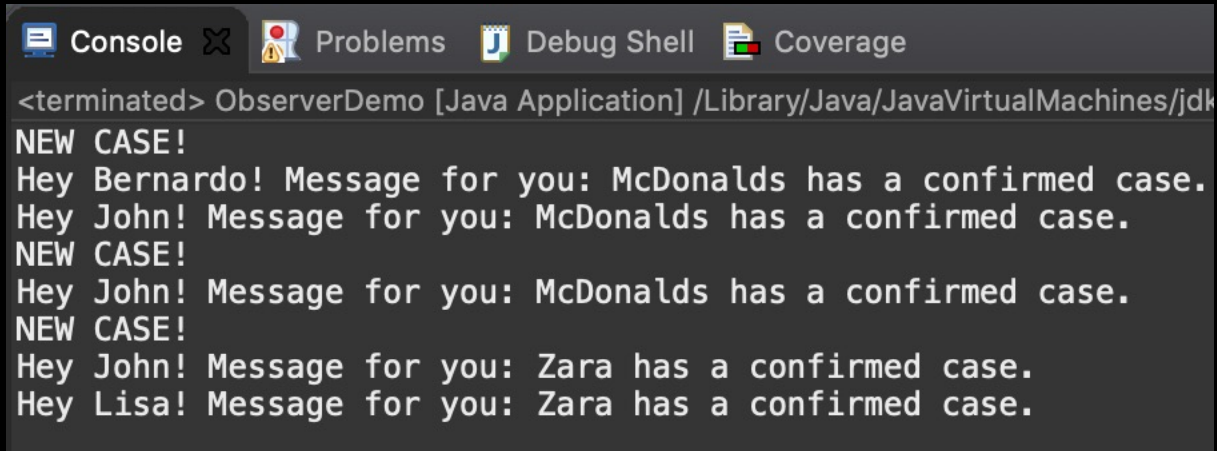
    public Place(String name) {
        this.name = name;
        observers = new ArrayList<Observer>();
    }
    public void setCorona() {//you can add some conditions here (boolean corona)
        notifyAllObservers();
    }
    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyAllObservers() {
        for(Observer obs : this.observers)
            obs.update(this.name + " has a confirmed case. ");
    }
}
```



```
public interface Observer {  
    public void update(String msg);  
}
```

```
public class Customer implements Observer{  
    private String name;  
  
    public Customer(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public void update(String msg) {  
        System.out.println("Hey " + this.name + "! Message for you: " + msg );  
    }  
}
```

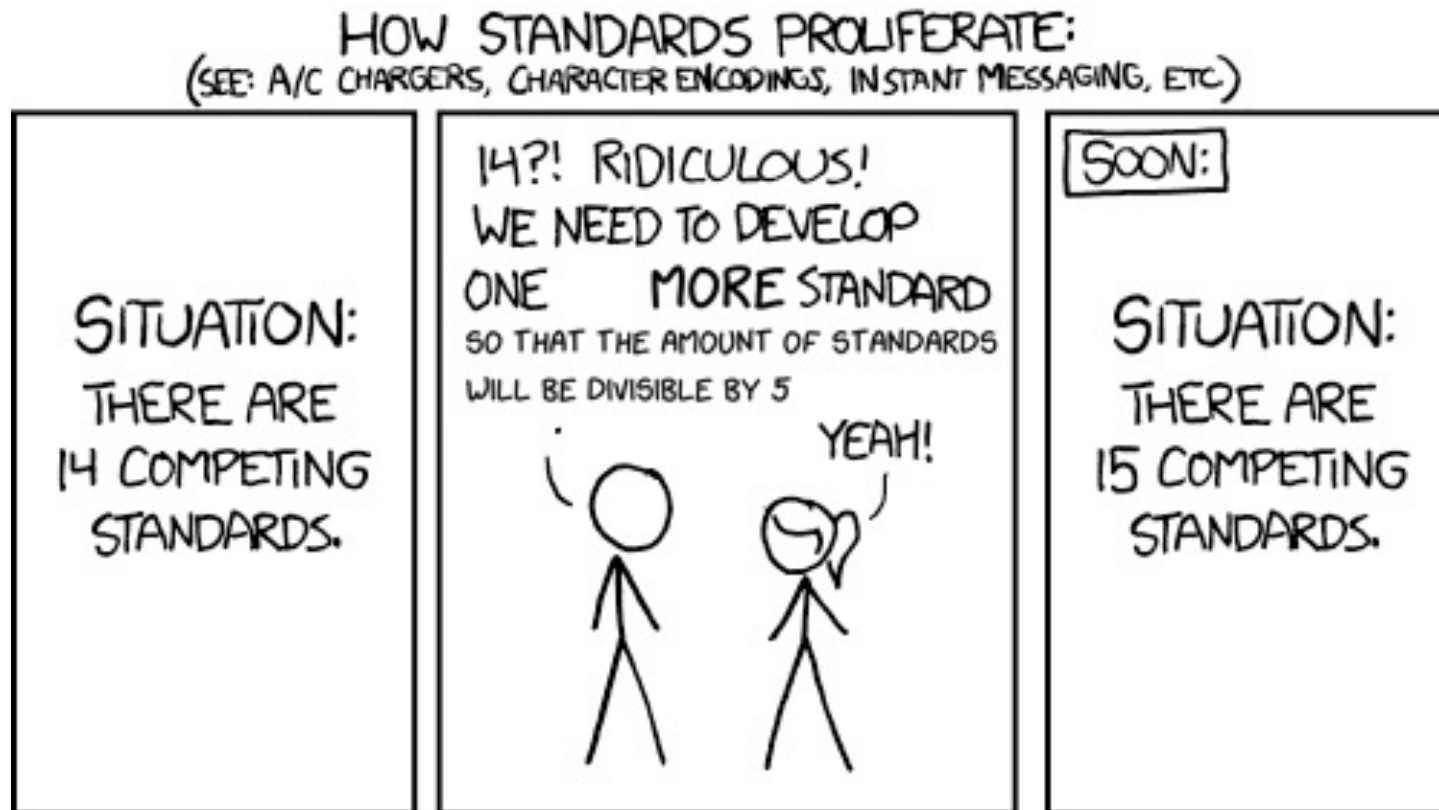
```
public class ObserverDemo {  
    public static void main(String[] args) {  
        Customer c1 = new Customer("Bernardo");  
        Customer c2 = new Customer("John");  
        Customer c3 = new Customer("Lisa");  
        Place p1 = new Place("McDonalds");  
        Place p2 = new Place("Zara");  
        p1.attach(c1);  
        p1.attach(c2);  
        p2.attach(c2);  
        p2.attach(c3);  
        System.out.println("NEW CASE!");  
        p1.setCorona();  
        p1.detach(c1);  
        System.out.println("NEW CASE!");  
        p1.setCorona();  
        System.out.println("NEW CASE!");  
        p2.setCorona();  
    }  
}
```



The screenshot shows an IDE interface with four tabs: Console, Problems, Debug Shell, and Coverage. The Console tab is active, displaying the output of the ObserverDemo application. The output shows the sequence of events: a new case is confirmed at McDonalds, which triggers messages to Bernardo and John. Then, a new case is confirmed at Zara, which triggers messages to John and Lisa.

```
<terminated> ObserverDemo [Java Application] /Library/Java/JavaVirtualMachines/jdk  
NEW CASE!  
Hey Bernardo! Message for you: McDonalds has a confirmed case.  
Hey John! Message for you: McDonalds has a confirmed case.  
NEW CASE!  
Hey John! Message for you: McDonalds has a confirmed case.  
NEW CASE!  
Hey John! Message for you: Zara has a confirmed case.  
Hey Lisa! Message for you: Zara has a confirmed case.
```

Meme for today's lecture! Keep practicing!



Main References

- [1] Steve McConnell. 2004. Code Complete, Second Edition. Microsoft Press, USA.
- [2] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. 2014. Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0 (3rd. ed.). IEEE Computer Society Press, Washington, DC, USA..
- [3] Grady Booch, James Rumbaugh, and Ivar Jacobson. 2005. Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). Addison-Wesley Professional.
- [4] UML Specification: <https://www.omg.org/spec/UML/2.5.1/PDF>
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., USA.