April 2022,
Course Java

Object Oriented Programming in Java

Trayan Iliev

tiliev@iproduct.org

http://iproduct.org

About me

Trayan Iliev

– CEO of IPT – Intellectual Products & Technologies

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6/7, TypeScript, Angular, React and Vue.js

– 15+ years IT trainer

– Voxxed Days, jPrime, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast (http://robolearn.org)

Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

https://github.com/iproduct/java-fundamentals-2022

Agenda for This Session

- OOP principles – Encapsulation, Inheritance and Polymorphism, Overriding / Overloading
- String Processing,
- Data Formatting, Resource Bundles, Regular Expressions
- java.util & java.math
- StringTokenizer, Date/Calendar,
- Locale, Random, Optional, Observable, Observable interface, BigDecimal

Basic Concepts in OOP and OOAD

- interface and implementation – we divide what remains constant (contractual interface) from what we would like to keep our freedom to change (hidden realization of this interface)
- interface = public
- implementation = private
- This separation allows the system to evolve while maintaining backward compatibility to already implemented solutions, enables parallel development of multiple teams
- programming based on contractual interfaces

Object-Oriented Approach to Programming

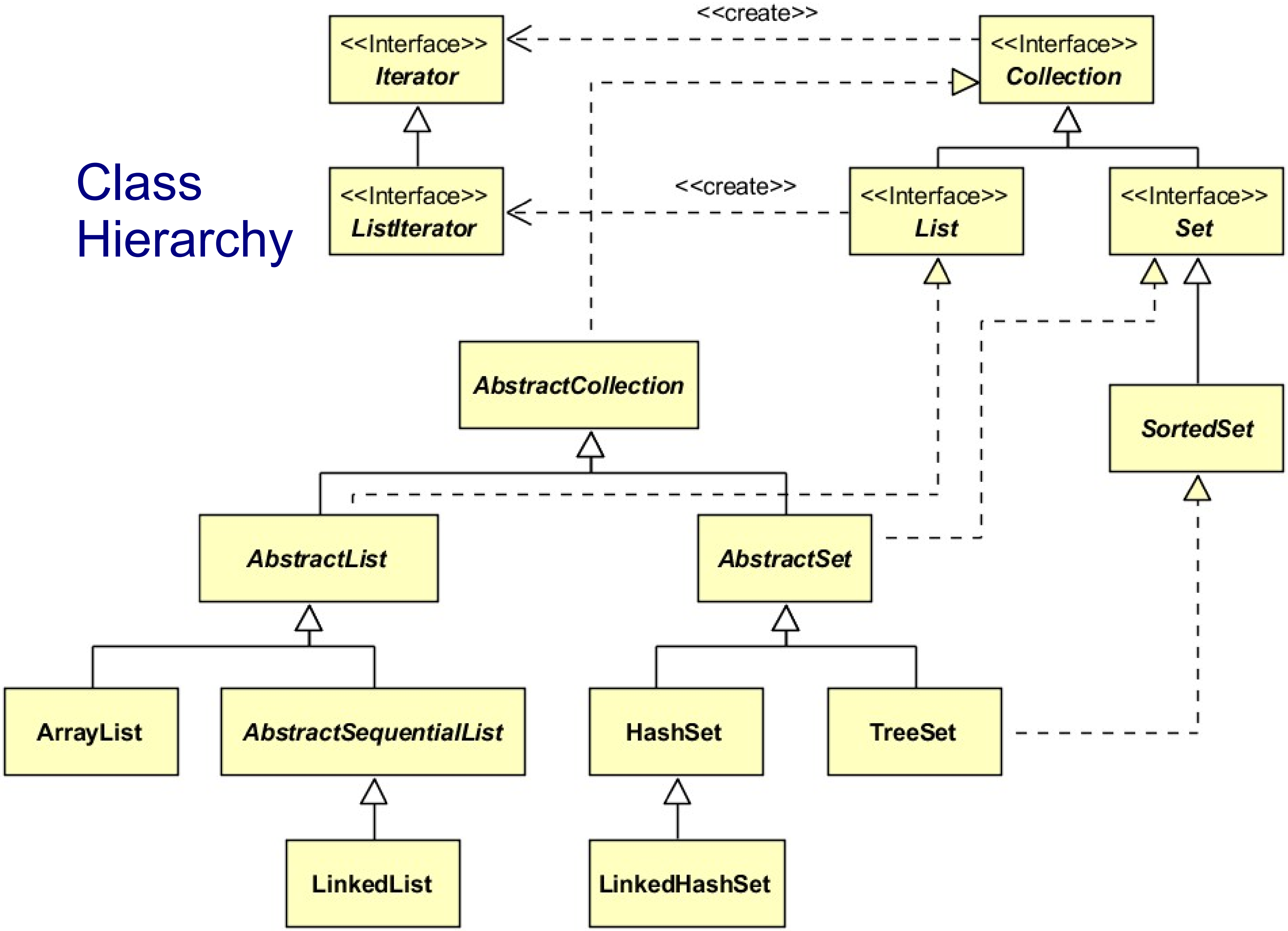
Key elements of the object model [Booch]:

- class, object, interface and implementation
- abstraction – basic distinguishing characteristics of an object
- capsulation – separating the elements of abstraction that make up its structure and behavior - interface and implementation
- modularity – decomposing the system into a plurality of components and loosely connected modules - principle: maximum coherence and the minimum connectivity
- hierarchy – class and object hierarchies

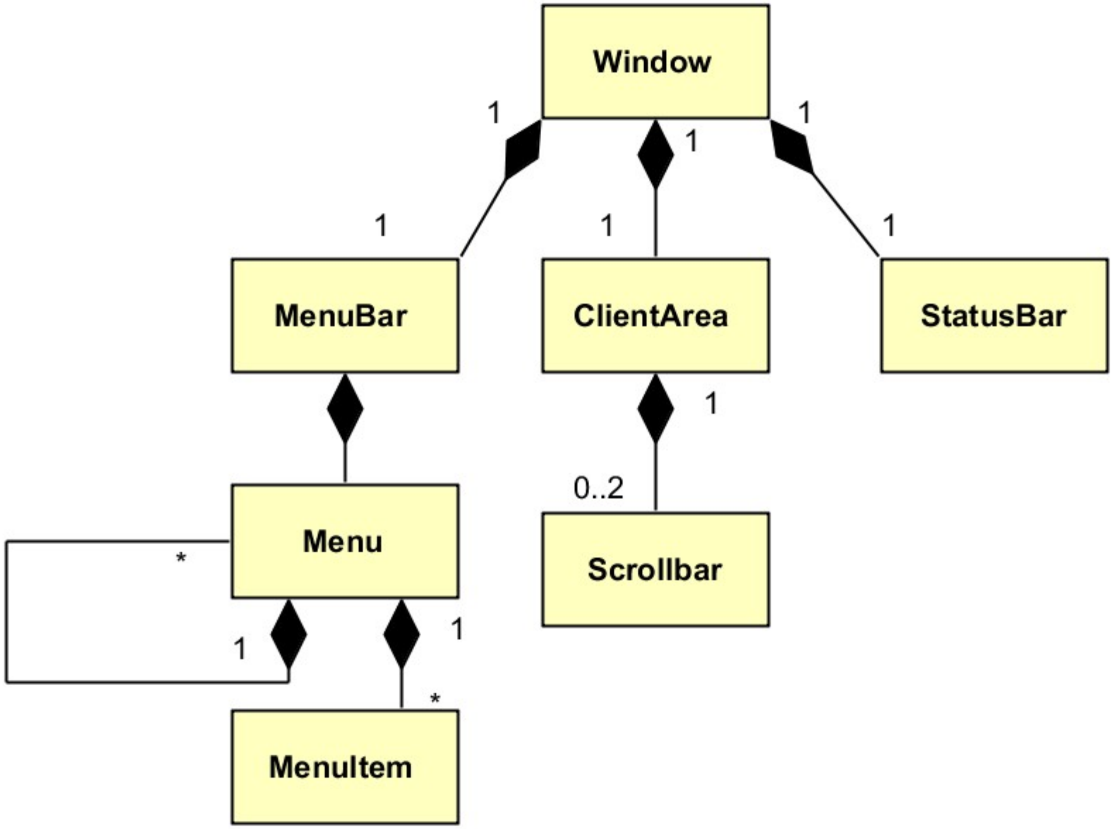
SOLID Design Principles of OOP

- Single responsibility principle - a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- Open–closed principle - software entities should be open for extension, but closed for modification.
- Liskov substitution principle - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- Interface segregation principle - Many client-specific interfaces are better than one general-purpose interface.
- Dependency inversion principle - depend upon abstractions, not concretions.

Class Hierarchy



Object Hierarchy



Object-Oriented Approach to Programming

Additional elements of the object model [Booch]:

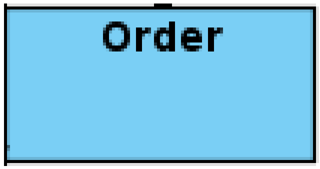
- typing – requirement for the class of an object such that objects of different types can not be replaced (or can in a strictly limited way)
  - static and dynamic binding
  - polymorphism
- concurrency – abstraction and synchronization of processes
- length of life – object-oriented databases

Classes

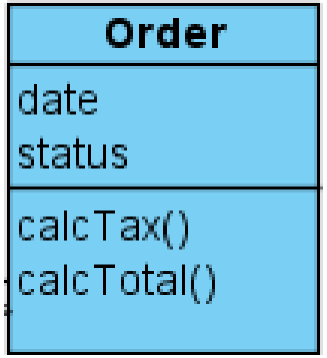
Class – describes a set of objects that share the same specifications of the characteristics (attributes and methods), constraints and semantics

- attributes – instances of properties in UML, they can provide end of association, object *structure*
- operations - behavioral characteristics of a classifier, specifying name, type, parameters and constraints for invoking definitely associated with the operation behavior

Classes - Graphical Notation in UML



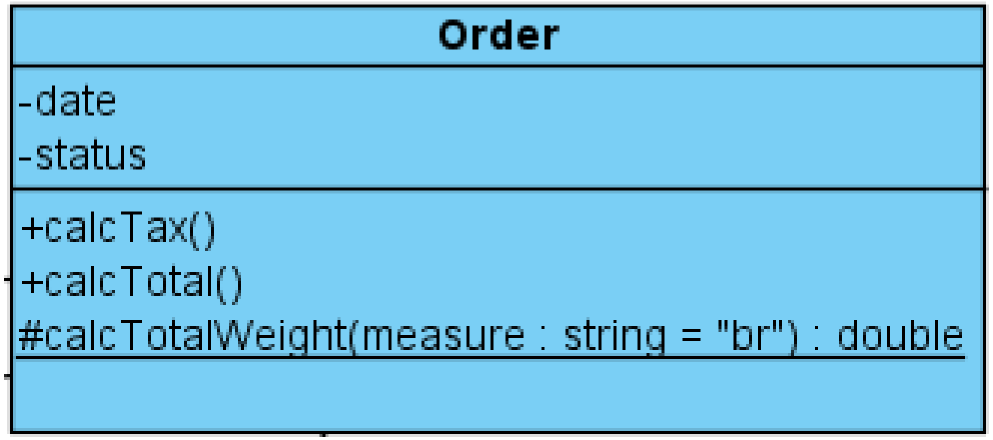
Order



Order

date
status

calcTax()
calcTotal()

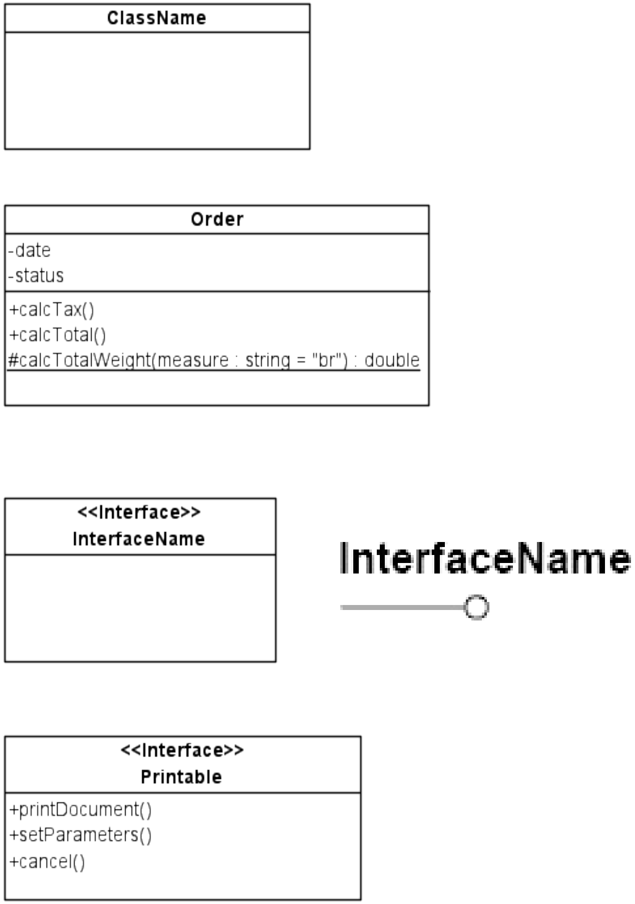


Order

-date
-status

+calcTax()
+calcTotal()
<u>#calcTotalWeight(measure : string = "br") : double</u>

Elements of Class Diagrams



ClassName

Order

-date
-status

+calcTax()
+calcTotal()
#calcTotalWeight(measure : string = "br") : double

<<Interface>>
InterfaceName

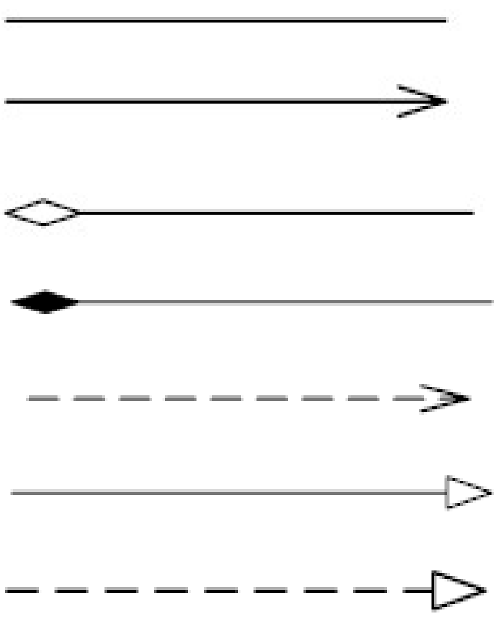
InterfaceName

<<Interface>>
Printable

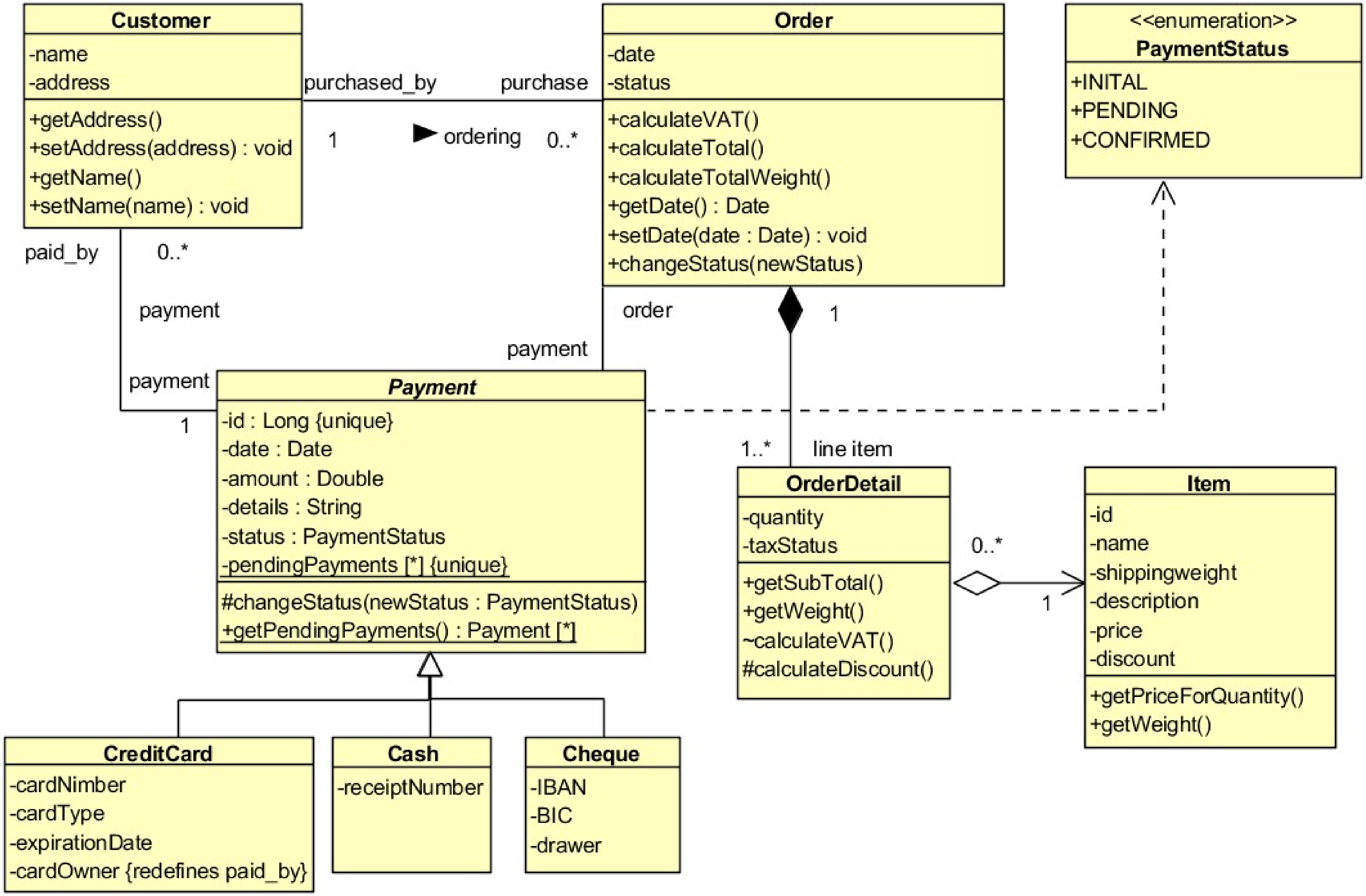
+printDocument()
+setParameters()
+cancel()

Types of connections:

- association
- aggregation
- composition
- dependence
- generalization
- realization



Class Diagram - 1



Customer

-name
-address

+getAddress()
+setAddress(address) : void
+getName()
+setName(name) : void

Order

-date
-status

+calculateVAT()
+calculateTotal()
+calculateTotalWeight()
+getDate() : Date
+setDate(date : Date) : void
+changeStatus(newStatus)

<<enumeration>>
PaymentStatus

+INITAL
+PENDING
+CONFIRMED

Payment

-id : Long {unique}
-date : Date
-amount : Double
-details : String
-status : PaymentStatus
-pendingPayments [*] {unique}

#changeStatus(newStatus : PaymentStatus)
+getPendingPayments() : Payment [*]

OrderDetail

-quantity
-taxStatus

+getSubTotal()
+getWeight()
~calculateVAT()
#calculateDiscount()

Item

-id
-name
-shippingweight
-description
-price
-discount

+getPriceForQuantity()
+getWeight()

CreditCard

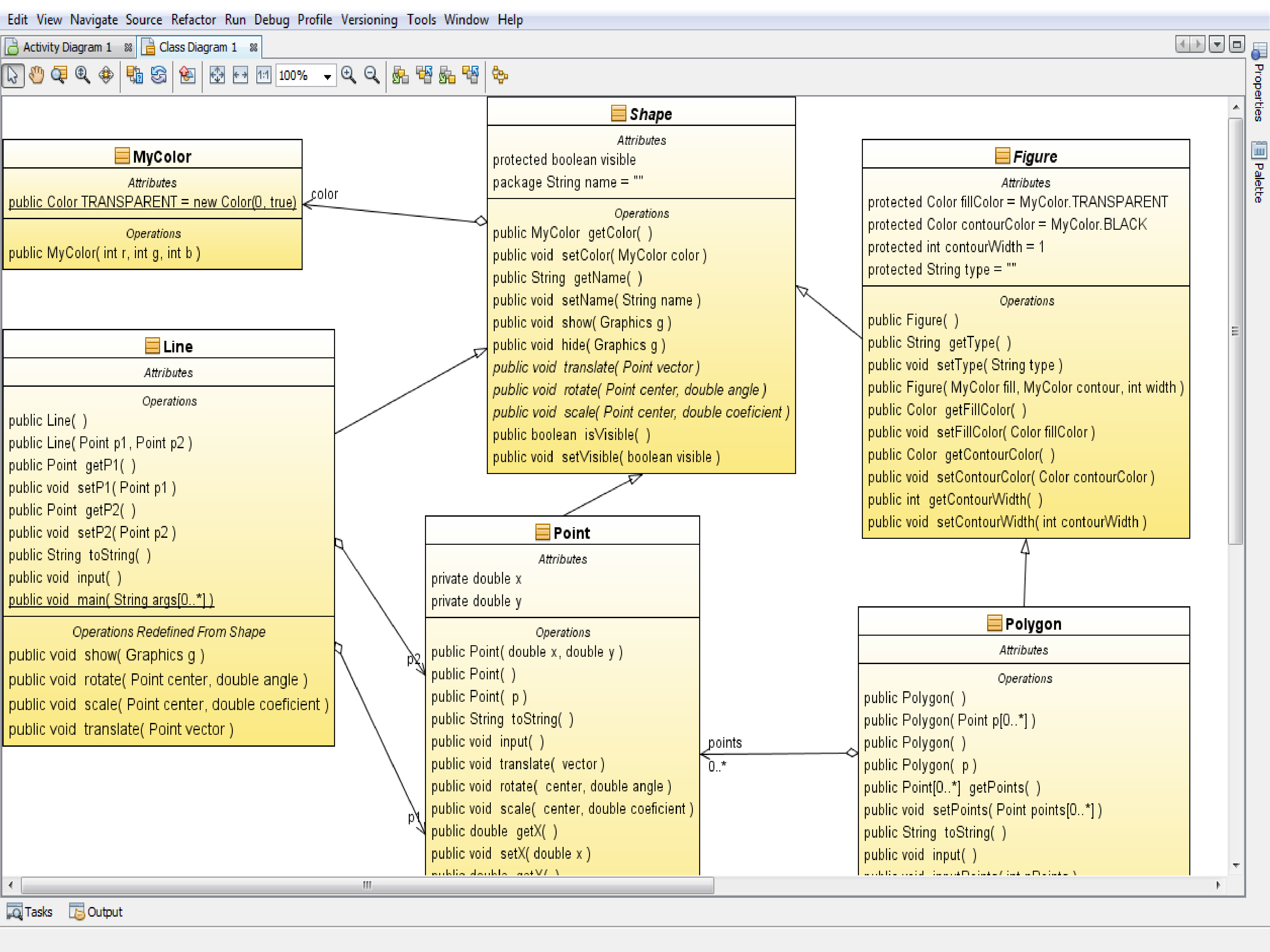
-cardNimber
-cardType
-expirationDate
-cardOwner {redefines paid_by}

Cash

-receiptNumber

Cheque

-IBAN
-BIC
-drawer



Shape

Attributes

protected boolean visible
package String name = ""

Operations

public MyColor getColor( )
public void setColor(MyColor color)
public String getName( )
public void setName(String name)
public void show(Graphics g)
public void hide(Graphics g)
public void translate(Point vector)
public void rotate(Point center, double angle)
public void scale(Point center, double coeficient)
public boolean isVisible( )
public void setVisible(boolean visible)

MyColor

Attributes

public Color TRANSPARENT = new Color(0, true)

Operations

public MyColor(int r, int g, int b)

Figure

Attributes

protected Color fillColor = MyColor.TRANSPARENT
protected Color contourColor = MyColor.BLACK
protected int contourWidth = 1
protected String type = ""

Operations

public Figure( )
public String getType( )
public void setType(String type)
public Figure(MyColor fill, MyColor contour, int width)
public Color getFillColor( )
public void setFillColor(Color fillColor)
public Color getContourColor( )
public void setContourColor(Color contourColor)
public int getContourWidth( )
public void setContourWidth(int contourWidth)

Line

Attributes

Operations

public Line( )
public Line(Point p1, Point p2)
public Point getP1( )
public void setP1(Point p1)
public Point getP2( )
public void setP2(Point p2)
public String toString( )
public void input( )
public void main(String args[0..*])

Operations Redefined From Shape

public void show(Graphics g)
public void rotate(Point center, double angle)
public void scale(Point center, double coeficient)
public void translate(Point vector)

Point

Attributes

private double x
private double y

Operations

public Point(double x, double y)
public Point( )
public Point(p)
public String toString( )
public void input( )
public void translate(vector)
public void rotate(center, double angle)
public void scale(center, double coeficient)
public double getX( )
public void setX(double x)
public double getY( )

Polygon

Attributes

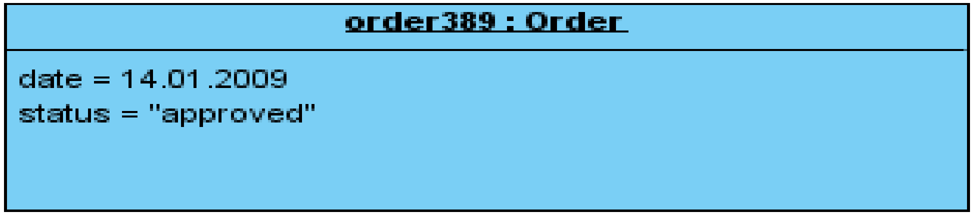
Operations

public Polygon( )
public Polygon(Point p[0..*])
public Polygon( )
public Polygon(p)
public Point[0..*] getPoints( )
public void setPoints(Point points[0..*])
public String toString( )
public void input( )
public void inputPoints(int nPoints)

Objects

Instance specification = Object – represents an instance of the modeled system, for example class -> object association -> link, property -> attribute, etc.

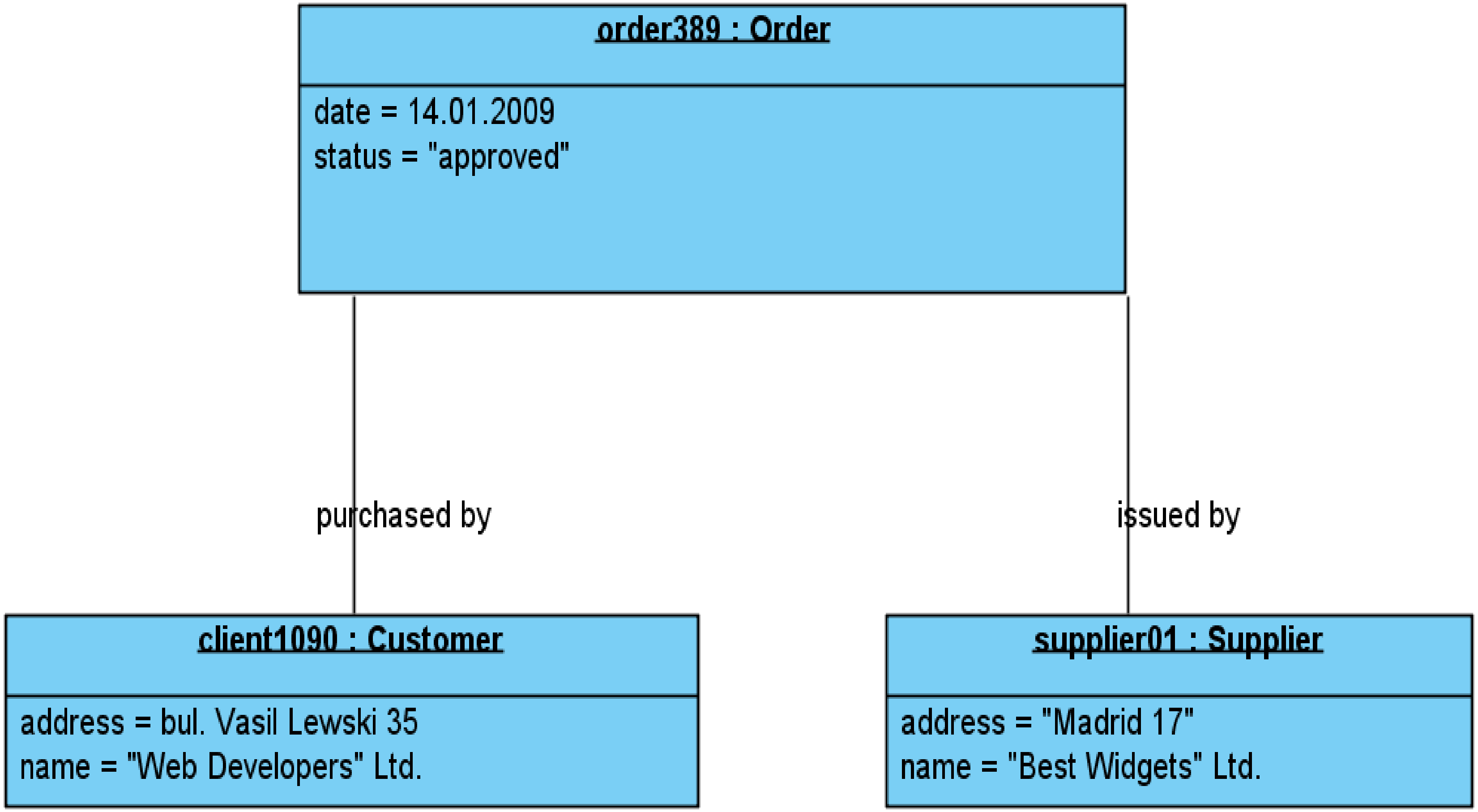
- can provide illustration or example of object
- describes the object in a particular moment of time
- may be uncomplete
- Example:



order389 : Order

date = 14.01.2009
status = "approved"

Object Diagram



order389 : Order

date = 14.01.2009

status = "approved"

purchased by

issued by

client1090 : Customer

address = bul. Vasil Lewski 35

name = "Web Developers" Ltd.

supplier01 : Supplier

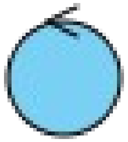
address = "Madrid 17"

name = "Best Widgets" Ltd.

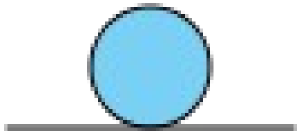
Analysis Classes Stereotypes

Analysis classes are used in the mapping and analysis of system architecture - they present rather different roles and responsibilities, than specific classes to be realized, and are independent of implementation technology:

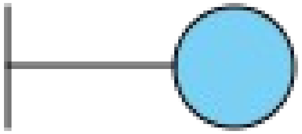
- <<controll>> - business logic
- <<entity>> - data
- <<boundary>> - user or system interface



Controlling Class



Class Unit



Border Class

Reusing Classes

- Advantages of code reuse
- Ways of implementation:
  - Objects composition
  - Inheritance of classes (object types)
- Building complex objects by composition
- Initializing the references:
  - on declaration of the site
  - in the constructor
  - before using (lazy initialization)

Class Inheritance - I

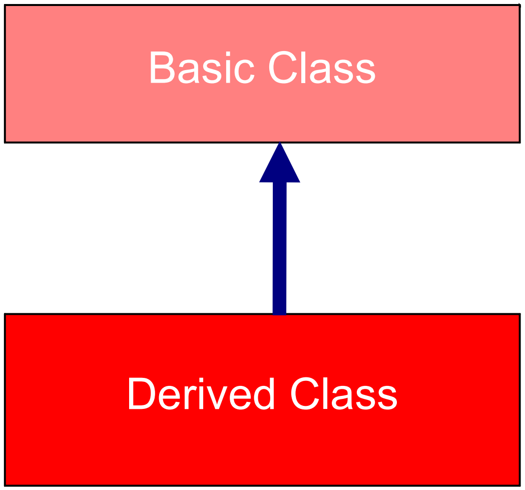
- ❖ Inheritance realization in Java™ language
  - – Keyword **extends**
  - – Keyword **super**
- ❖ Initialization of objects inheritance:
  1) base class; 2) inherited class
  - – Calling the default constructors
  - – Calling constructors with arguments
- ❖ Combining composition and inheritance

Class Inheritance - II

- Clearing of objects – realization in Java™
- Overloading and overriding methods of base class in derived classes
- When to use composition and when inheritance?
  - Do we need the interface of the base class?
  - Connection Type - „there is“ and „it is“?

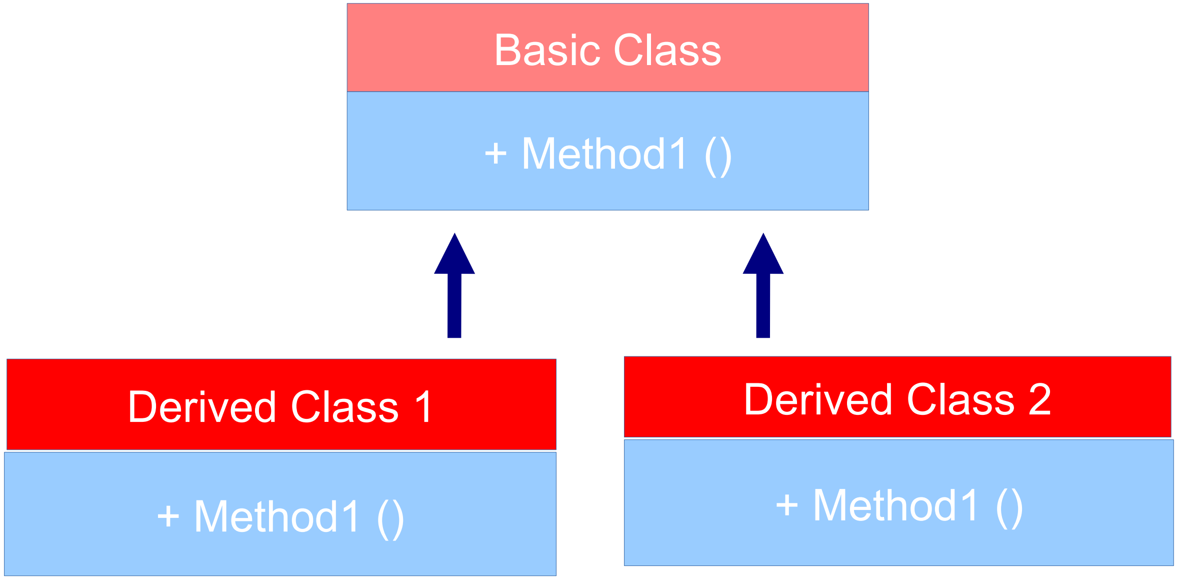
Class Inheritance - III

- Protected methods
- Upcasting
- Keyword final
  - Final data – defining constants
    - simple data type
    - objects
    - empty fields
    - arguments
  - Final methods
  - Final classes



Polymorphism - I

- Upcasting



- Abstract methods and classes – abstract
- Order of constructor calls
- Inheritance and expansion

Polymorphism - II

- Polymorphism – by default, unless the method is declared as static or final (private methods become automatically final)
- When constructing objects with inheritance each object cares about its attributes and delegate initialization of parental attributes on parental constructor or method
- Using polymorphic methods in constructor
- Covariance types of return (from Java SE 5)
- Composition <-> Inheritance - **State** Design Pattern

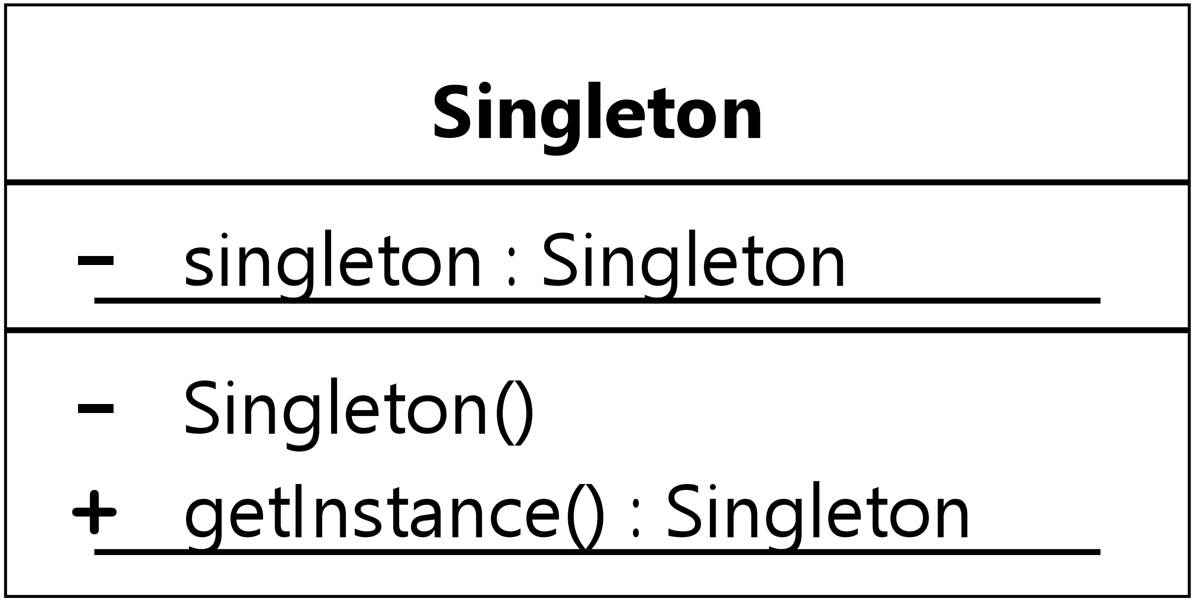
Interfaces and Multiple Inheritance

- Interfaces – keywords: interface, implements
- Multiple inheritance in Java
- Interface expansion through inheritance
- Constants (static final)
- Interface incorporation

Advantages of Using Interfaces

- Interfaces cleanly separate requirements type of the object from many possible implementations and make our code more universal and usable
- Reusable Design Pattern: Adapter – It allows to adapt existing realization interface that is required in our application
- Inheritance (expansion) of interfaces
- Reusable Design Pattern: Factory Method – creating reusable client code, isolated from the specifics of the particular server implementation

Singleton Design Pattern



Singleton

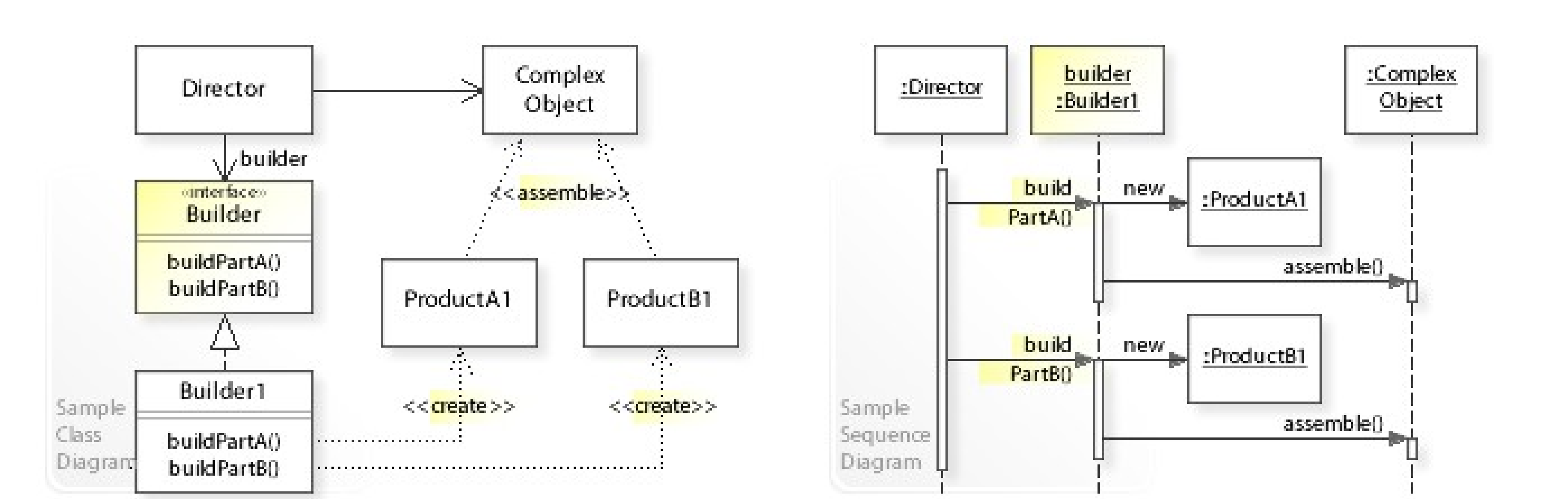
− singleton : Singleton

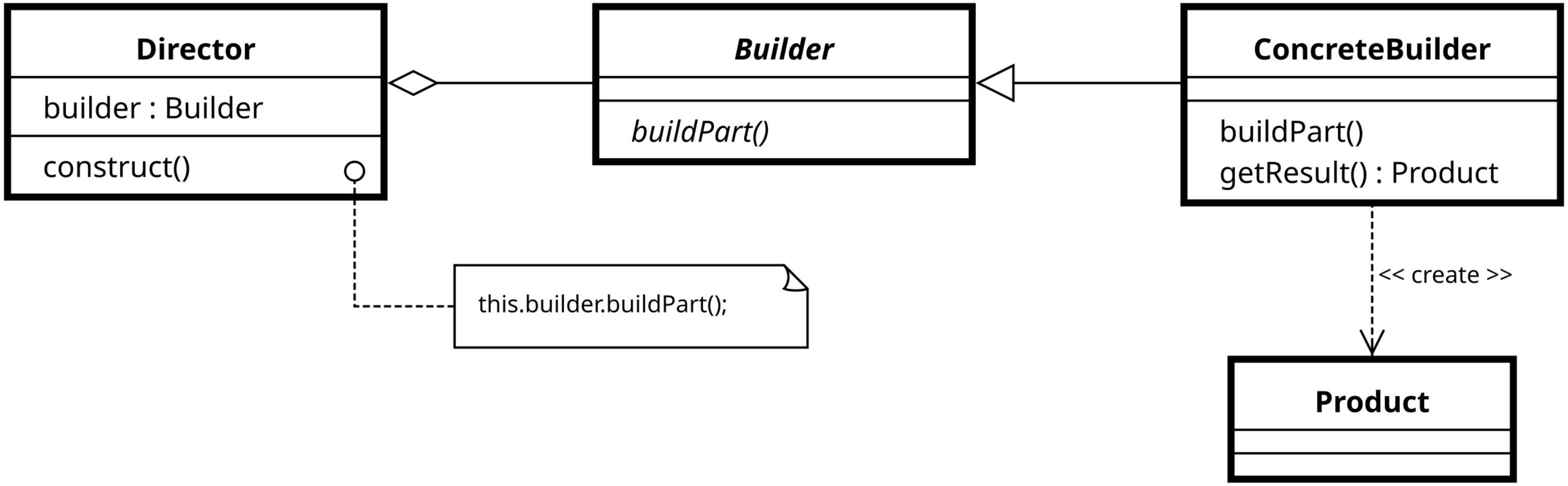
− Singleton()

+ getInstance() : Singleton

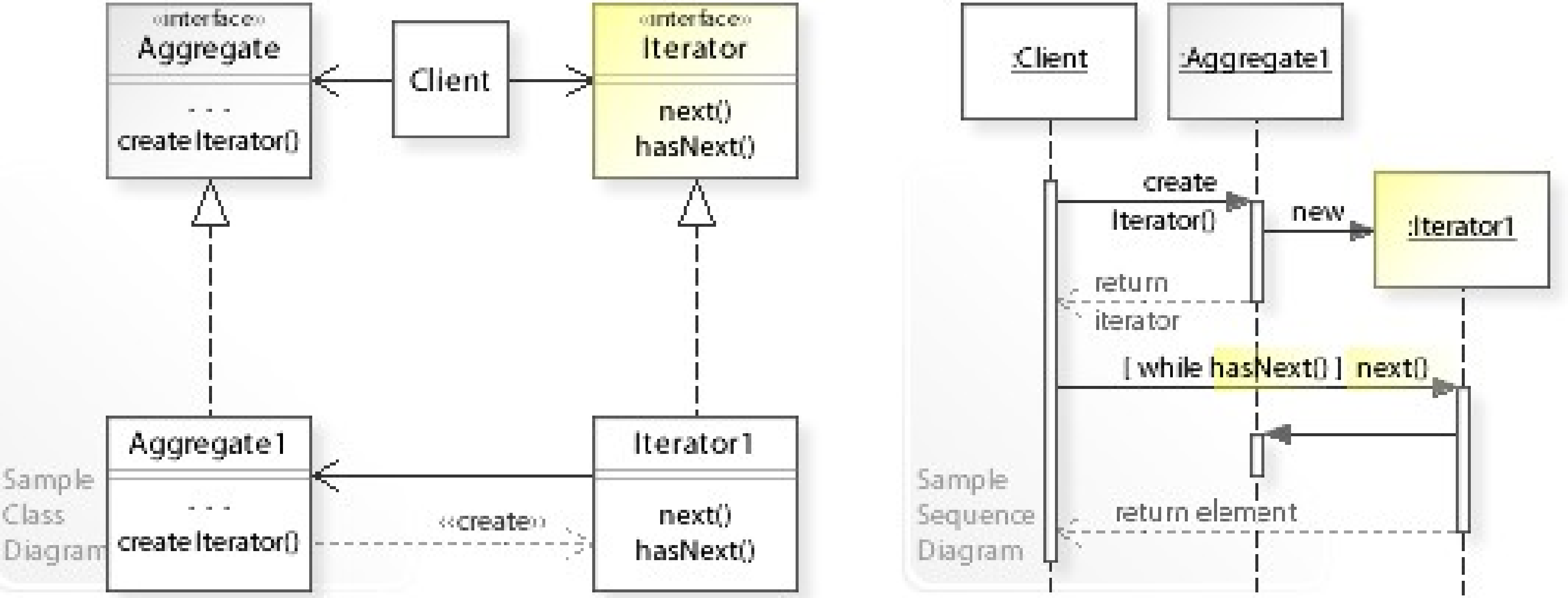
By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63061518

Builder Design Pattern





Iterator Design Pattern

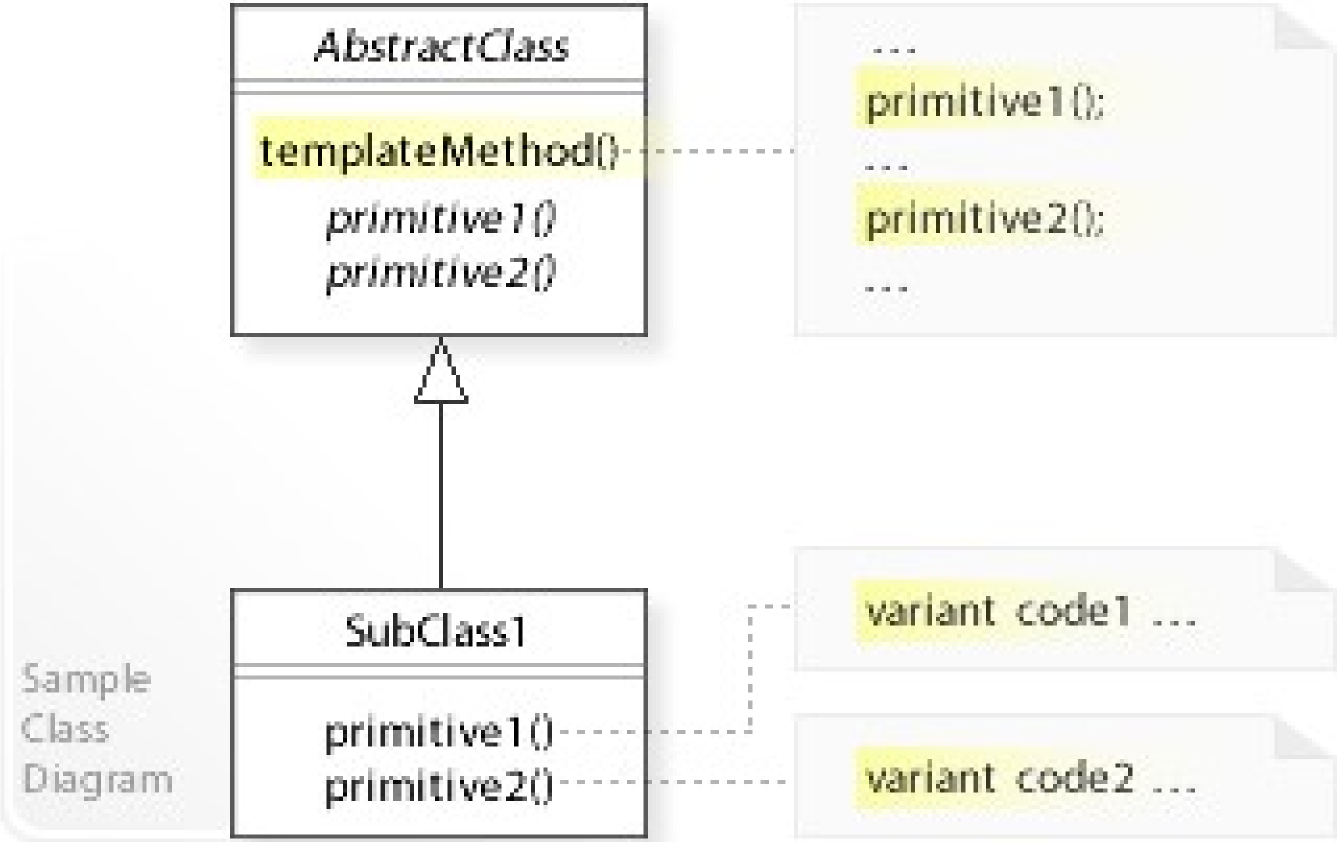


Iterator Design Pattern

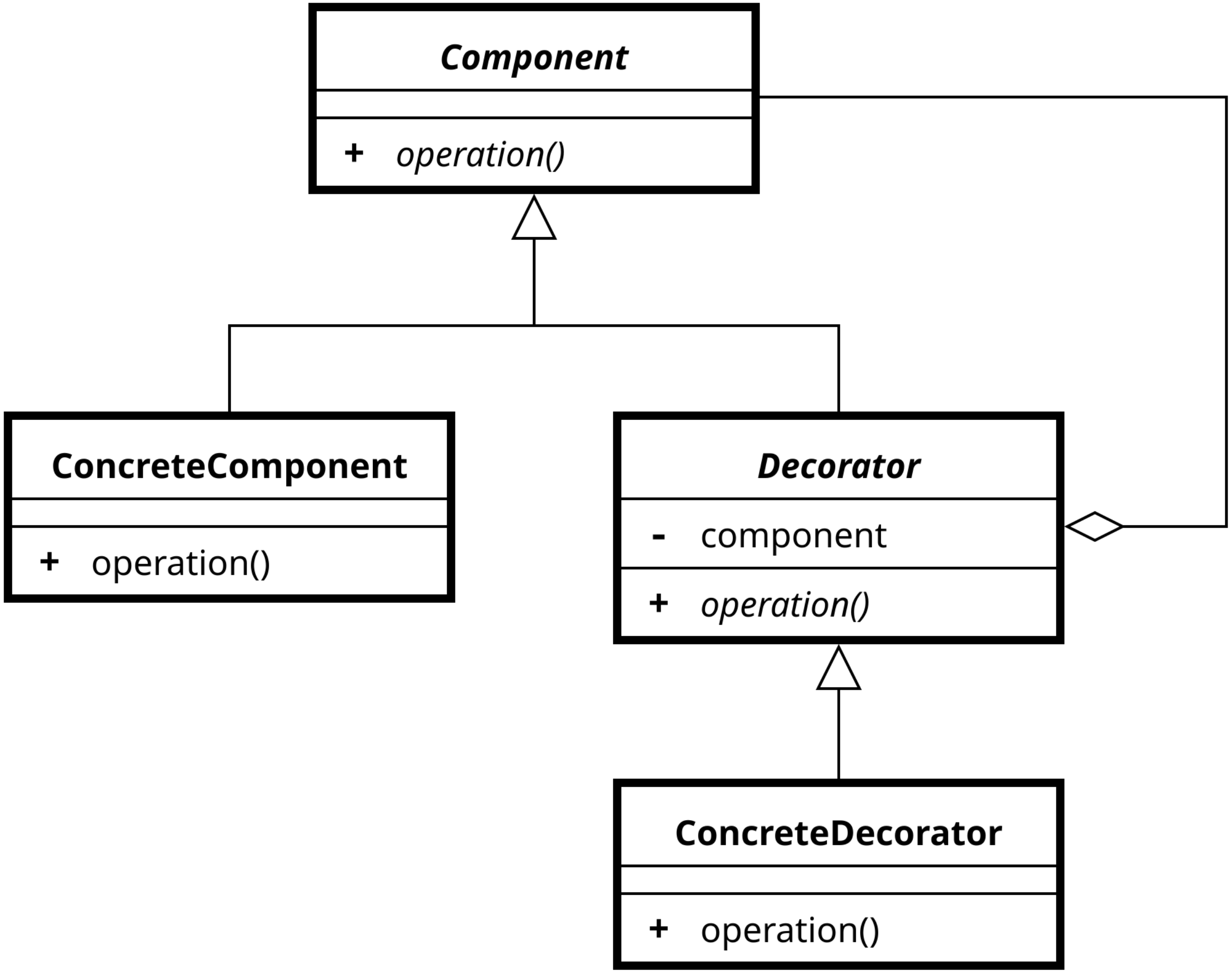
```
Iterator iter = list.iterator();
// Iterator<MyType> iter = list.iterator(); // in J2SE 5.0
while (iter.hasNext()) {
    System.out.print(iter.next());
    if (iter.hasNext())
        System.out.print(", ");
}
```

By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63061518

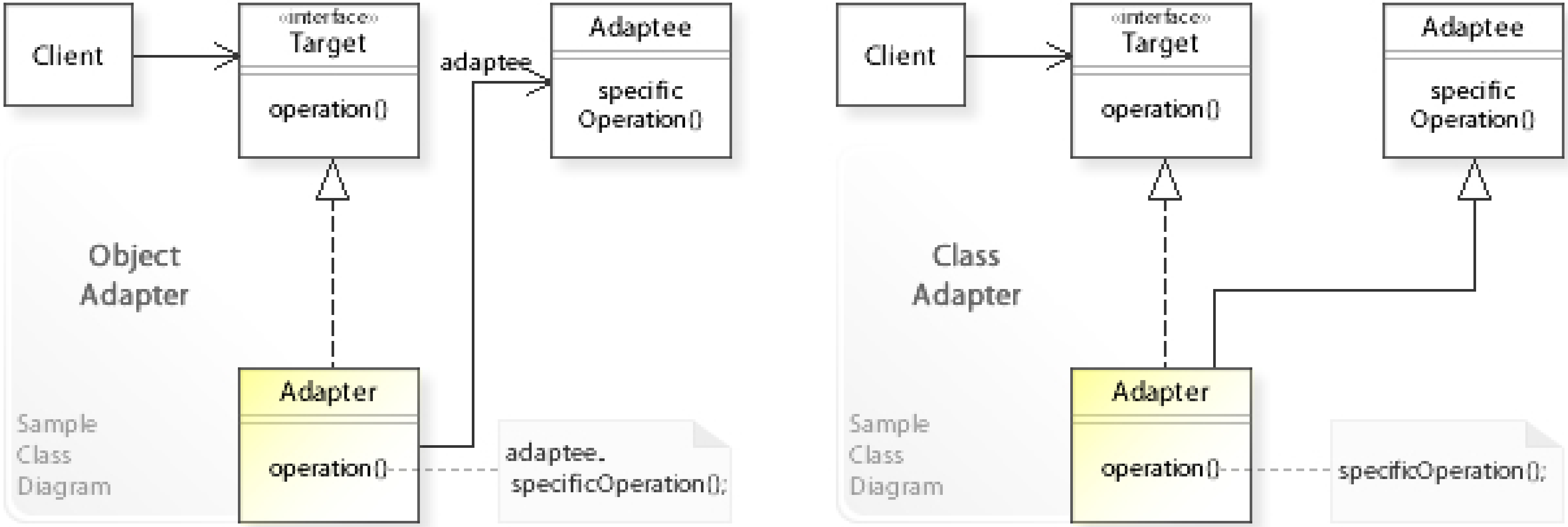
Template Method Design Pattern



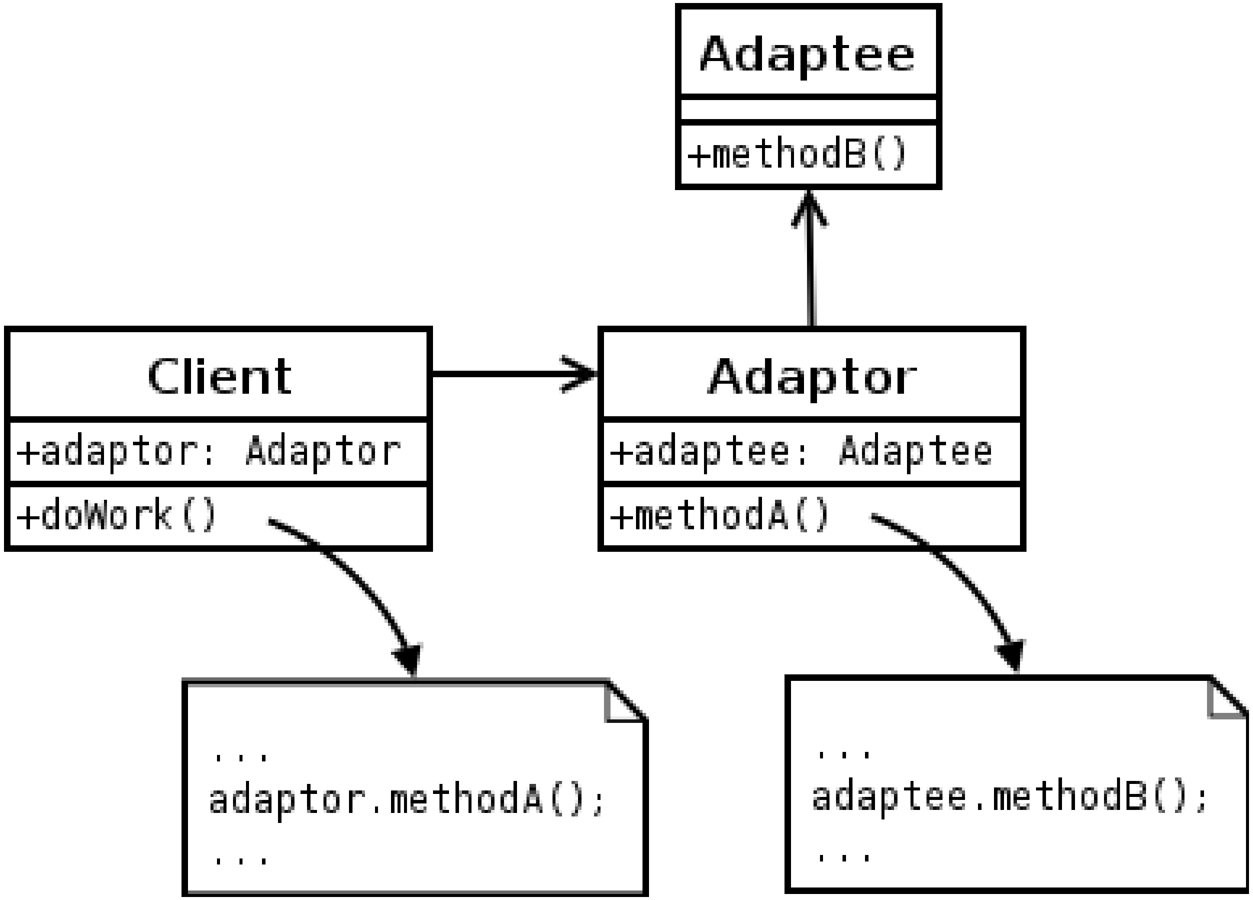
Decorator Design Pattern



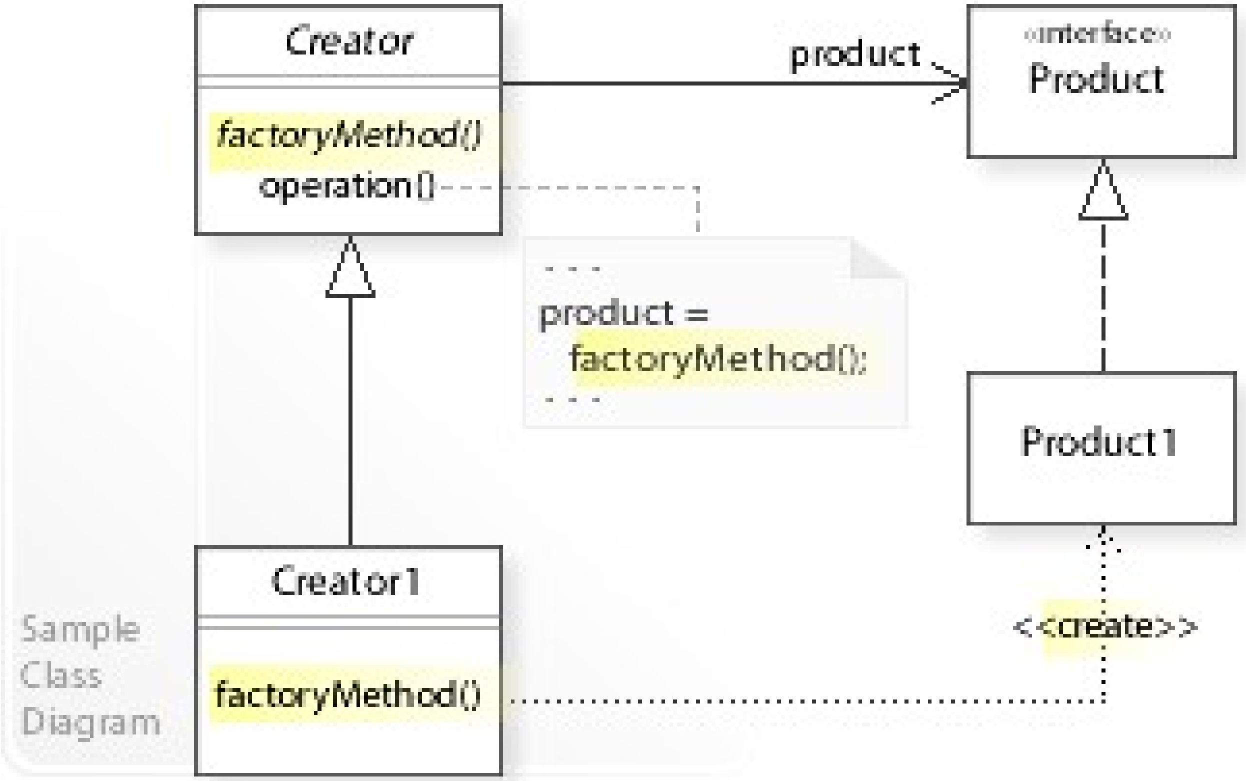
Adapter Design Pattern



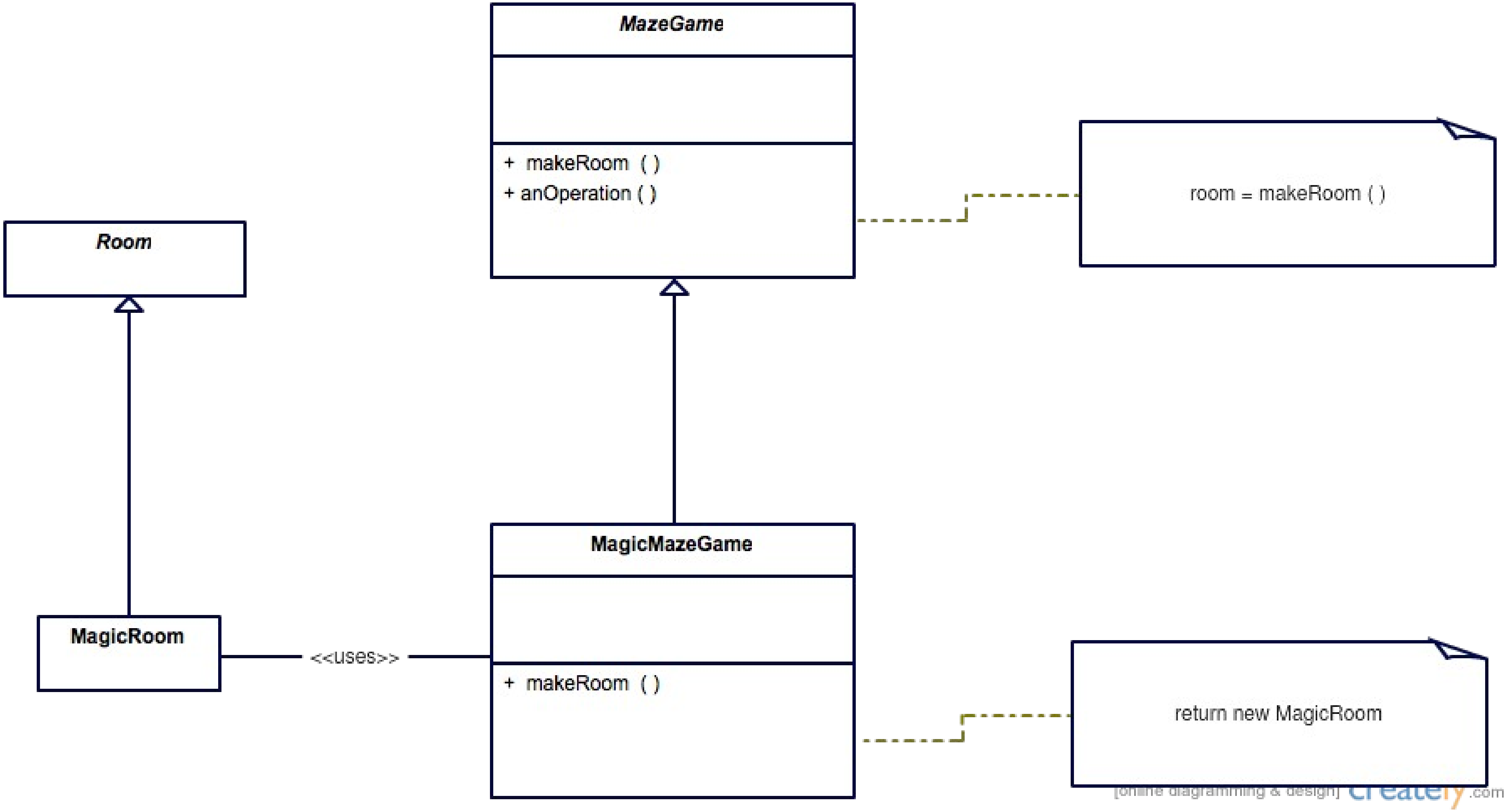
Adapter Design Pattern - II



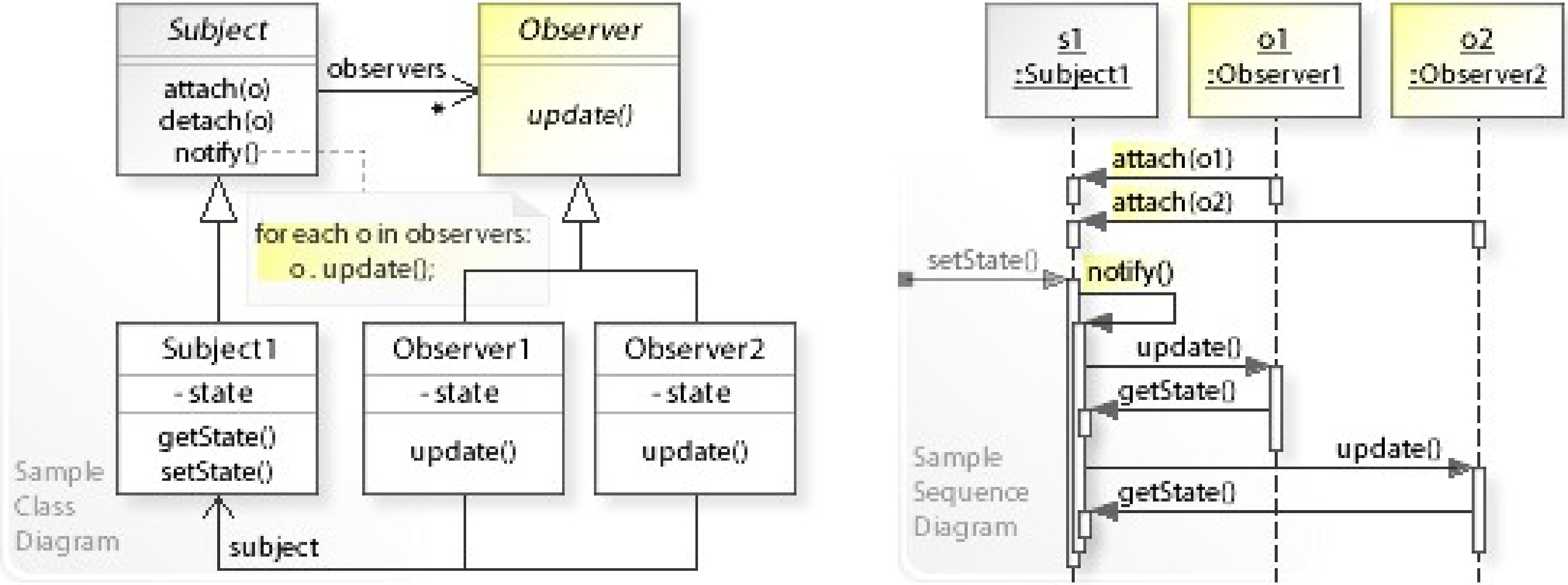
Factory Method Design Pattern



Factory Method Design Pattern - II



Observer Design Pattern



By Vanderjoe - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=63061518

Inner Classes - I

- Inner Classes group logically related classes and control their visibility
- Closures – internal class has a constant connection to containing outside class and can access all its attributes and even final arguments and local variables (if defined in the method or block)
- Inner classes can be anonymous if used once in the program. Construction.
- Reference to the object from an external class - .this and creating an object from internal class in the context of containing object of the outer class - .new

Inner Classes - II

- Inner Classes
  - defined in an external class
  - defined in method
  - defined in a block of operators
  - access to the attributes of the outer class and to the arguments of the method which are defined in
- Anonymous inner classes
  - realizing public interface
  - inheriting class
  - instance initialization
  - static inner classes

Enumeration Types

```
public class MyEnumeration {
  public enum InvoiceType { SIMPLE, VAT }
  public static void main(String[] args) {
    for(InvoiceType it : InvoiceType.values())
        System.out.println(it);
    }
}
```

Резултат: SIMPLE
VAT

Exceptions Handling in Java

- Mandatory processing of exceptions in Java → secure and reliable code
- Separating the business logic of the program from exception handling code
- Class **Throwable** → classes Error and Exception
  - Generating exceptions – keyword **throw**
  - Exceptions handling:
    - **try – catch – finally** block
    - delegating exception handling to caller - **throws**

Try-Catch-Finally Block

❖Operator **try** for executing unreliable code, multiple **catch** blocks for handling exceptions and **finally** for guaranteed clean-up at the end of processing:

```
try {
    // code that can generate exceptions Ex1, Ex2,  …
} catch(Ex1 ex) {  // only executed when Ex1
    // appropriate measures taken for solving problem 1
} catch(Ex2 ex) {  // only executed when Ex2
    // appropriate measures taken for solving problem 2
} finally {
    // executed always, independently of weather there is an
    // exception or not
}
```

Handling Exceptions in Java - 2

- Implementing custom exceptions
- Constructors with additional arguments
- Embedding and re-throwing excptions – Cause
- Specific handling of **RuntimeException** and its descendants – unchecked exceptions
- Guaranteed completion/cleanup using **finally**

Noveltions in Exception Handling in Java 7

- Handling multiple exceptions in same **catch** clause:

```
catch (Exception1|Exception2 ex) {
    ex.printStackTrace();
}
```

- Block **try-with-resources**

```
String readInvoiceNumber(String myfile) throws IOException {
      try (BufferedReader input = new BufferedReader(new
                FileReader(myfile))) {
                      return input.readLine();
                }
}
```

Thank's for Your Attention!

Trayan Iliev

CEO of IPT – Intellectual Products

& Technologies

http://iproduct.org/

http://robolearn.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD

https://plus.google.com/+IproductOrg