

Lecture 11.1

A very small selection of some topics in Object-Oriented Design

Systems development

The development of any software system involves several stages:

- Each one of these stages is based on a significant body of knowledge and practice
- Requirements analysis:
 - Data
 - Processes
 - Design
 - Implementation
 - Testing
 - Deployment
 - Maintenance
- Introduced in this course
- Systems development is not computer science. It's a discipline that intersects CS, management science, operations research, systems analysis, and engineering.
 - How can you learn systems development?
 - Other courses in CS / IT / management programs
 - On-the-job workshops
 - Hands-on experience
 - Common sense.

Testing

How can we ensure that a (large-scale) program does what it's supposed to do?

- Correctness proof (ranges from the impractical to the impossible)
- Testing:
 - Unit-testing
 - Designing for testing
 - Test planning
 - Regression testing
- **Test suite:** a well-documented collection of test-cases, each describing a particular input, user action, etc., and a desired output or outcome

Black-box testing

- Uses the system's API only
- Inputs are separated into *equivalence categories*
- The borders of the category must be carefully tested , to catch "one off" errors

Example (taken from some API):

```
// Returns true if x is in the range 0 to 99, inclusive
public boolean isCent(int x)
```

The test suite of this method should be something like:

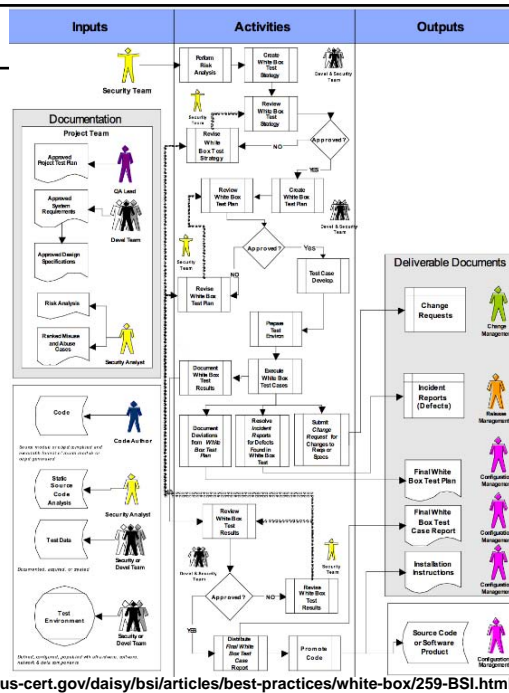
-500, -1, 0, 1, 50, 98, 99, 100, 500.

White-box testing

- Analyzes the code itself by trying to visit every possible path that the program flow may take ("statement coverage")
- Used to ensure correctness and prevent hacking

Software testing is a big business:

- Software tools
- Best practices
- Consulting companies
- 4 million hits on Googling "software testing"



<https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/white-box/259-BSI.html>

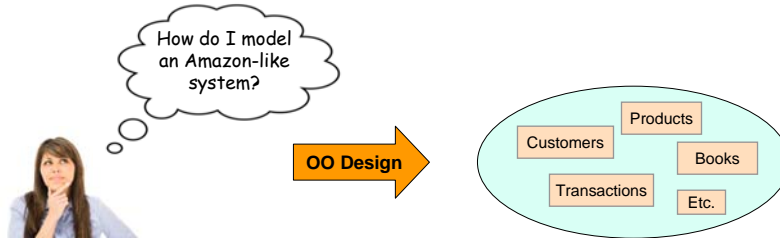
Systems development

The development of any software system involves several stages:

- Requirements analysis:
 - Data
 - Processes
- Design
- Implementation
- Testing
- Deployment
- Maintenance

OO

OO Design: class relationships



When we identify classes (part of system design), we typically try to map the classes on existing *design patterns*. The OO literature describes dozens of such design patterns, and OO designers are well advised to consult them

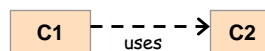
Typically, the classes that we will end up identifying and documenting are related to each other in three basic ways:

- Dependency: class C1 *uses* class C2. Example: Transactions **uses** Products
- Aggregation: class C1 *has-a* class C2. Example: Customers **has-a** Customer
- Inheritance: class C1 *is-a* class C2. Example: Book **is-a** Product

Dependency

When class C1 uses the services of class C2 in some way, we say that "C1 uses C2" or "C1 depends on C2".

Examples:



Class c

```
x = Math.sqrt(y)
```

The call to the static method creates a "C uses Math" dependency. This dependency is so basic that we don't bother to note it

Class c

```
if (file.isDirectory())
```

The file object was either created by c or passed to one of its methods as a parameter. Either way creates a "C uses File" dependency

Class c

```
act1.transferTo(400, act)
```

Same as in previous case: "C uses BankAccount" dependency

Class Set

```
this.insertSet(s2)
```

One object interacting with another object from the same class, creating a "Set uses Set" dependency.

Dependency example

```
public class Customer {
    private String firstName, lastName;
    private Address homeAddress, shippingAddress;

    // Constructs a customer
    public Customer (String firstName,
                    String lastName,
                    Address homeAddress,
                    Address shippingAddress) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.homeAddress = homeAddress;
        this.shippingAddress = shippingAddress;
    }

    // A textual rep. of this customer.
    public String toString() {
        String result;
        result = firstName + " " +
                lastName + "\n";
        result += "home Address: \n" +
                homeAddress + "\n";
        result += "Shipping Address: \n" +
                shippingAddress;
        return result;
    }
}
```

- customer **uses** Address
- The Address class can be used by other classes, simplifying their definitions.

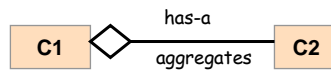
```
// Represents an address
public class Address {
    private String streetAddress, city;
    private long zipCode;

    // Constructs an address
    public Address (String streetAddress,
                  String city,
                  long zipCode) {
        this.streetAddress = streetAddress;
        this.city = city;
        this.zipCode = zipCode;
    }

    // Returns this address as text
    public String toString() {
        return streetAddress + "\n"
                + city + ", " + zipCode;
    }
}
```

Aggregation

If an object of class C1 is some collection of objects of class C2, we say that "C1 aggregates C2", or "C1 has-a C2"



Aggregation example

```

public class Customers {
    public static void main (String[] args) {
        Address homeAddress;
        Address shippingAddress = new Address ("10E 14th Street", "New York", 20010);
        Customer[] customers = new Customer[2]; // In reality the array will be bigger ...

        homeAddress = new Address ("15 Dekel Street", "Raanaana", 24551);
        customers[0] = new Customer ("Ron", "Tami r", homeAddress, shippingAddress);

        homeAddress = new Address ("10 Negev Street", "Beer Sheva", 44132);
        customers[1] = new Customer ("Mi chal", "Shami r", homeAddress, shippingAddress);

        for (int i=0 ; i < customers.length ; i++) {
            System.out.println (customers[i]);
            System.out.println ();
        }
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
D:\demo>java Customers
Ron Tamir
home Address:
15 Dekel Street
Raanaana, 24551
Shipping Address:
10E 14th Street
New York, 20010

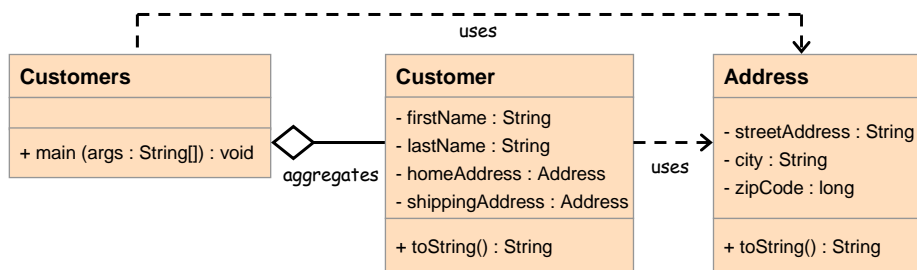
Michal Shamir
home Address:
10 Negev Street
Beer Sheva, 44132
Shipping Address:
10E 14th Street
New York, 20010

```

Aggregation relationship:

- Customers **aggregates** customer

UML class diagram



Unified Modeling Language (UML): a popular graphical notation for OO design

- Useful tool for visualizing the design of complex software architectures
- UML is Language independent
- Each class is represented by a block with three elements: class name, attributes, operations
- Relationships between classes are represented by arrows.

Outline

- Systems development
- Testing
- Class relationships
 - Dependency
 - Aggregation
 - Inheritance
- ➔ ■ Interfaces
- Modularity
- Enumarted types revisited

Interface

```
// Represents a musical instrument
interface Instrument {
    void play();
    void mute();
}
```

We wish to create several classes, each representing a musical instrument

We want to force all these classes to implement a set of behaviors that every musical instrument must have

This design goal can be achieved using an interface

- An interface says: "This is what classes that *implement* this particular interface should look like"
- Each class that implements an interface says: "I conform to the interface that I implement".

```
public class Guitar implements Instrument{

    // Constructs a guitar
    public Guitar (...) {}

    // Various guitar methods

    public void play() {
        // Code that plays this guitar
    }

    public void mute() {
        // Code that mutes this guitar
    }
}
```

```
public class Flute implements Instrument {
    // Similar, must implement play() and mute()
}
```

```
public class Flute implements Instrument {
    // Similar, must implement play() and mute()
}
```

Interface: the rules of the game

The interface:

- Interface = a collection of constants and abstract methods
- An interface file is a compilation unit, just like a class files
- An interface cannot be instantiated (no `new`)
- All the methods of an interface are, by default, public and abstract

```
// Represents a musical instrument
interface Instrument {
    void play();
    void mute();
}
```

```
class Guitar implements Instrument {
    // Must implement play() and mute()
}
```

The implementing class:

- A class can implement 0, 1, or more interfaces
- The implementing class must provide implementations for all the methods mentioned in all the interfaces it is implementing; failure to do so causes a compilation error
- Multiple classes can implement the same interface.

The Java standard class library includes many interfaces.

Example: the Comparable interface

- Part of the `java.lang` package
- Includes a single abstract method: `compareTo()`
- Implemented by numerous classes, including `String`

```
String word1 = scan.nextLine();
String word2 = scan.nextLine();
// ...
if (s1.compareTo(s2) < 0)
    System.out.println ("word1 preceded word2");
```

Interface Comparable API

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's `compareTo` method is referred to as its natural comparison method.

Lists and arrays of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

public int `compareTo`(Object o)

Parameters: o - the Object to be compared.

Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws: [ClassCastException](#) - if the specified object's type prevents it from being compared to this Object.

Example: the Iterator interface

- Part of the `java.util` package
- Describes a "standard way" to move through a collection of objects, one object at a time
- Implemented by many iterator classes in Java

Method Summary

<code>boolean</code>	<code>hasNext ()</code> Returns <code>true</code> if the iteration has more elements.
<code>Object</code>	<code>next ()</code> Returns the next element in the iteration.
<code>void</code>	<code>remove ()</code> Removes from the underlying collection the last element returned by the iterator (optional operation).

Outline

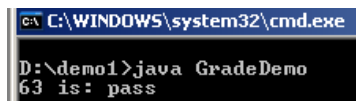
- Systems development
- Testing
- Class relationships
 - Dependency
 - Aggregation
 - Inheritance
- Interfaces
- ➔ ■ Modularity
- Enumarted types revisited

End comment: Modularity

- Perhaps the most important feature of well designed systems
- The best way to tame complexity is to divide the system into small, well-defined, manageable, unit-tested modules
- A modular design also implies sovereignty: each module should manage its own functionality
- We will now see an example that ...
 - Illustrates modular design
 - Revisits Java's "enumerated type" concept.

Enumerated types revisited

```
public class GradeDemo {  
    enum Grade {good, pass, fail};  
  
    public static void main (String[] args) {  
  
        int x = 63;  
        Grade g = null;  
  
        if (x >= 90) {g = Grade.good;}  
        if ((x >= 60) && (x < 90)) {g = Grade.pass;}  
        if (x < 60) {g = Grade.fail;}  
  
        System.out.println(x + " is: " + g);  
    }  
}
```



```
cmd C:\WINDOWS\system32\cmd.exe  
D:\demo1>java GradeDemo  
63 is: pass
```

Modular design: the grade logic should ideally be delegated to a stand-alone `Grade` class.

The hidden life of enum:

- `Grade` is a class (behind the scene)
- `Grade.high`, `Grade.pass`, `Grade.fail` are instances of the `Grade` class
- More precisely, they are references to `Grade` objects stored in `public static` variables within the `Grade` class
- Since an `enum` is a class, we can extend its definition at will.

Enumerated types revisited, Take 2

```
public enum Grade {
    good (90, 100),
    pass (75, 89),
    fail (64, 74);

    private int low;
    private int high;

    Grade (int low, int high) {
        this.low = low;
        this.high = high;
    }

    public int getLow() { return low; }
    public int getHigh() { return high; }

    public Boolean withinRange (int x) {
        return ((x >= low) && (x <= high));
    }

    public static Grade toVerbalGrade (int x) {
        for (Grade g : Grade.values())
            if (g.withinRange(x))
                return g;
        return null;
    }
}
```

```
public class GradeDemo {

    public static void main (String[] args) {
        for (Grade g : Grade.values())
            System.out.println (g + "\t" +
                g.getLow() + "\t" +
                g.getHigh());

        System.out.println("78 in verbal grade: " +
            Grade.toVerbalGrade(78));
    }
}
```

```
cmd C:\WINDOWS\system32\cmd.exe
```

```
D:\demo>java GradeDemo
good    90    100
pass    75    89
fail    64    74
78 in verbal grade: pass
```