An Introduction to Object-Oriented Analysis and Design and the Unified Process

“Applying UML and Patterns, 3rd ed.” – Craig Larman, pp. 197 – 319

Kakarontzas George
gkakaron@teilar.gr
What is Logical Architecture

- The logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers. It's called the logical architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the deployment architecture).

- A layer is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. Also, layers are organized such that "higher" layers (such as the UI layer) call upon services of "lower" layers, but not normally vice versa. Typically layers in an OO system include:
  - **User Interface**.
  - **Application Logic and Domain Objects** software objects representing domain concepts (for example, a software class Sale) that fulfill application requirements, such as calculating a sale total.
  - **Technical Services** general purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.
Strict and Relaxed Layered Architectures

- In a **strict layered architecture**, a layer only calls upon the services of the layer directly below it.

- This design is common in network protocol stacks, but not in information systems, which usually have a **relaxed layered architecture**, in which a higher layer calls upon several lower layers. For example, the UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.
An Example Logical Architecture

UI

- Swing
- Web

not the Java Swing libraries, but our GUI classes based on Swing

Domain

- Sales
- Payments
- Taxes

Technical Services

- Persistence
- Logging
- RulesEngine
What is Software Architecture?

“An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition.” – Booch, G., Rumbaugh, J, and Jacobson, I. 1999. The Unified Modeling Language User Guide. Reading, MA.: Addison-Wesley
UML Package Diagrams

- UML package diagrams are often used to illustrate the logical architecture of a system – the layers, subsystems, packages (in the Java sense), etc. A layer can be modeled as a UML package; for example, the UI layer modeled as a package named UI.

- A UML package diagram provides a way to group elements. A UML package can group anything: classes, other packages, use cases, and so on. Nesting packages is very common. A UML package is a more general concept than simply a Java package or .NET namespace, though a UML package can represent those – and more.
Package Dependencies and Fully-Qualified Class Names

- It is common to want to show dependency (a coupling) between packages so that developers can see the large-scale coupling in the system. The UML dependency line is used for this, a dashed arrowed line with the arrow pointing towards the depended-on package.

- A UML package represents a namespace so that, for example, a Date class may be defined in two packages. If you need to provide fully-qualified names, the UML notation is, for example, java::util::Date in the case that there was an outer package named "java" with a nested package named "util" with a Date class.
The UML provides alternate notations to illustrate outer and inner nested packages.
Designing Systems with Layers

The essential ideas of using layers are simple:

- Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.

- Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided.

Problems Addressed by the Layers Pattern

The problems addressed by the *Layers Pattern* include the following:

- Source code changes are rippling throughout the system – many parts of the systems are highly coupled.
- Application logic is intertwined with the user interface, so it cannot be reused with a different interface or distributed to another processing node.
- Potentially general technical services or business logic is intertwined with more application-specific logic, so it cannot be reused, distributed to another node, or easily replaced with a different implementation.
- There is high coupling across different areas of concern. It is thus difficult to divide the work along clear boundaries for different developers.
Common Layers in Information System Logical Architecture

- **UI** (AKA Presentation, View)
  - GUI windows
  - reports
  - speech interface
  - HTML, XML, XSLT, JSP, Javascript, ...

- **Application** (AKA Workflow, Process, Mediation, App Controller)
  - handles presentation layer requests
  - workflow
  - session state
  - window/page transitions
  - consolidation/transformation of disparate data for presentation

- **Domain** (AKA Business, Application Logic, Model)
  - handles application layer requests
  - implementation of domain rules
  - domain services (POS, Inventory)
  - services may be used by just one application, but there is also the possibility of multi-application services

- **Business Infrastructure** (AKA Low-level Business Services)
  - very general low-level business services used in many business domains
  - CurrencyConverter

- **Technical Services** (AKA Technical Infrastructure, High-level Technical Services)
  - (relatively) high-level technical services and frameworks
  - Persistence, Security

- **Foundation** (AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
  - low-level technical services, utilities, and frameworks
  - data structures, threads, math, file, DB, and network I/O

Width implies range of applicability
Benefits of Using Layers

- In general, there is a separation of concerns, a separation of high from low-level services, and of application-specific from general services. This reduces coupling and dependencies, improves cohesion, increases reuse potential, and increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations. This is generally not possible for lower-level Technical Service or Foundation layers (e.g., java.util), but may be possible for UI, Application, and Domain layers.
- Lower layers contain reusable functions.
- Some layers (primarily the Domain and Technical Services) can be distributed.
- Development by teams is aided because of the logical segmentation.
Domain Objects and the Domain Layer

- The recommended approach is to create software objects with names and information similar to the real-world domain, and assign application logic responsibilities to them. For example, in the real world of POS, there are sales and payments. So, in software, we create a Sale and Payment class, and give them application logic responsibilities. This kind of software object is called a **domain object**. It represents a thing in the problem domain space, and has related application or business logic, for example, a Sale object being able to calculate its total.

- Designing objects this way leads to the application logic layer being more accurately called the **domain layer** of the architecture – the layer that contains domain objects to handle application logic work.
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter. This reduces the representational gap.

This is one of the big ideas in object technology.
Tiers, Layers and Partitions

- **Tier**: The word has become widely used to mean a physical processing node (or cluster of nodes), such as the "client tier" (the client computer).

- **Layers**: They represent the vertical slices of the system.

- **Partitions**: They represent a horizontal division of relatively parallel subsystems of a layer. For example, the Technical Services layer may be divided into partitions such as Security and Reporting.

```
<table>
<thead>
<tr>
<th>Vertical Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
</tr>
<tr>
<td>- POS</td>
</tr>
<tr>
<td>- Inventory</td>
</tr>
<tr>
<td>- Tax</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Horizontal Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technical Services</td>
</tr>
<tr>
<td>- Persistence</td>
</tr>
<tr>
<td>- Security</td>
</tr>
<tr>
<td>- Logging</td>
</tr>
</tbody>
</table>
```
Model-View Separation Principle

- This principle has at least two parts:
  - Do not connect or couple non-UI objects directly to UI objects. For example, don't let a Sale software object (a non-UI "domain" object) have a reference to a Java Swing JFrame window object. Why? Because the windows are related to a particular application, while (ideally) the non-windowing objects may be reused in new applications or attached to a new interface.
  - Do not put application logic (such as a tax calculation) in the UI object methods. UI objects should only initialize UI elements, receive UI events (such as a mouse click on a button), and delegate requests for application logic on to non-UI objects (such as domain objects).

- Model = Domain Layer
- View = User Interface Layer
Benefits of Model-View Separation

- Supports cohesive model definitions that focus on the domain processes, rather than on user interfaces.
- Allows separate development of the model and user interface layers.
- Minimizes the impact of requirements changes in the interface upon the domain layer.
- Allows new views to be easily connected to an existing domain layer, without affecting the domain layer.
- Allows multiple simultaneous views on the same model object, such as both a tabular and business chart view of sales information.
- Allows execution of the model layer independent of the user interface layer, such as in a message-processing or batch-mode system.
- Allows easy porting of the model layer to another user interface framework.
Connection Between SSDs, System Operations and Layers

- In a well-designed layered architecture, the UI layer objects will forward or delegate the requests from the UI layer (system operations) onto the domain layer for handling.

- The messages sent from the UI layer to the domain layer will be the messages illustrated on the SSDs, such as enterItem.
the system operations handled by the system in an SSD represent the operation calls on the Application or Domain layer from the UI layer.
Designing Objects: Static and Dynamic Modeling

- There are two kinds of object models: dynamic and static.
  - Dynamic models, such as UML interaction diagrams (sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies. They tend to be the more interesting, difficult, important diagrams to create.
  - Static models, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).

- There is a relationship between static and dynamic modeling and the agile modeling practice of create models in parallel: Spend a short period of time on interaction diagrams (dynamics), then switch to a wall of related class diagrams (statics).
Designing Objects: Static and Dynamic Modeling (cont.)

- **UML Class Diagram**
  - **DiceGame**
    - ...
    - `play()`
  - **Die**
    - `faceValue`
    - `roll()`

- **UML Sequence Diagram**
  - **play**
  - **roll**

- **Static Model**
- **Dynamic Model**
  - `::DiceGame`
  - `::Die`
UML Class Diagrams
Class

- Classes are the basis of OO systems. They encapsulate data and the methods acting on them.
- UML symbol for a class is a rectangle with three compartments. The top compartment contains the name of the class, the middle contains the attributes of the class, and the third contains the operations of the class.

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- balance : Money</td>
</tr>
<tr>
<td>+ withdraw(amount : Money)</td>
</tr>
<tr>
<td>+ deposit(amount : Money)</td>
</tr>
</tbody>
</table>
Public and Private Features

- A usual practice is that properties are *private*, whereas operations are *public*.
  - Private are the features of a class (properties or operations) that are not accessible from outside the class (i.e. from other classes).
  - Public are those features that are accessible from other classes.

- Data Hiding Principle:
  - Easy to change the internal representation of data
  - Application of control policies to data access.
The symbol “–” is used for private visibility
‘+’ means public visibility
Visibility symbol “#” is protected visibility, and it means that the property or operation is accessible from the class and possible subclasses.
Package visibility is indicated with the “~” symbol and it means that the property or operation is accessible from within the class in which it is declared and the classes within the same package with that class.
Finally, not using a symbol has the meaning that the visibility indicator is missing: it doesn’t mean that there is no access or that access is public.
UML Property Syntax

The full syntax for properties in UML is:

- Visibility property-name: data-type [multiplicity order] = initial-value {property string}

Few examples:

- students : string[* ordered]
- interestRate : double = 0.035 {frozen}
UML operation syntax

- The full syntax for operations in UML is:
  - visibility name(parameter-list):return-type {property-string}

- Parameter-list syntax:
  - [in | out | inout] parameter-name : data-type = default-value

- Few examples:
  - + withdraw(in amount : Money) : boolean
  - + getBalance():Money {query}
Features with class scope

- Some operations or properties have class scope not object scope.
- Suppose that we want to be able to create Account objects, by reading the information of each such object from a database. We could have an operation with the following syntax:
  
  `+createAccount(AccountID : String) : Account`
Class Associations

- An association between two classes depicts a static relationship between these classes.
- Associations are depicted with a line connecting the two classes.
Rolenames

- At each association end we can place a name that stands for the role of this class in the association (a rolename).
- Rolenames are necessary when two classes are connected with more than one associations.
Multiplicity

- Multiplicity is placed at the end of an association and denotes the possible number of connected objects at any moment during program execution.
- In the figure, an account has one or more holders and a customer might have one or more accounts.
- Other values for the multiplicity include the following: * (zero or more), 0..1 (zero or one), some specific number (e.g. 11), or a specific range of values (e.g. 2..4).
Navigability

- Navigability in an association is denoted with an arrow at the end of an association, and it means navigation capability only towards the direction of the arrow.
- It indicates the capability to obtain objects of the other class in an association.
- In the example in the figure, we can navigate from the “Department” class to the “Teacher” class, but not the opposite (i.e. having a department object allows retrieval of the associated teacher objects, but not the opposite).
In the following code example we have a “Teacher” class, that represents the “Teacher” domain concept. Each teacher has a name (a String). Since teachers don’t have the obligation to know their departments (i.e. no navigability from Teacher to Department), Teacher class does not refer to the Department class.
public class Teacher {
    private String name;
    public Teacher(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public boolean equals(Object o) {
        Teacher t = (Teacher) o;
        return t.name.equals(this.name);
    }
    public int hashCode() {
        return name.hashCode();
    }
}
...import statements
public class Department {
    private Teacher president;
    private Set<Teacher> teachers;
    private String name;
    public Department(String name) {
        this.name = name;
        this.teachers = new HashSet<Teacher>();
    }
    public boolean addTeacher(Teacher t) {
        return teachers.add(t);
    }
    public boolean removeTeacher(Teacher t) {
        return teachers.remove(t);
    }
}
public boolean findTeacher(Teacher t) {
    return teachers.contains(t);
}

public void setPresident(Teacher t) {
    this.president = t;
}

public Teacher getPresident() {
    return president;
}

public void printTeachers() {
    Iterator<Teacher> iterator = teachers.iterator();
    while (iterator.hasNext()) {
        Teacher t = iterator.next();
        System.out.println(t.getName());
    }
}
public static void main(String args[]) {
    Teacher george = new Teacher("G. Kakarontzas");
    Teacher george1 = new Teacher("G. Kakarontzas");
    Department dept = new Department("XYZ");
    dept.addTeacher(george);
    //this insertion will fail!
    boolean ok = dept.addTeacher(george1);
    if (!ok) {
        System.out.println("Insertion failed");
    }
    System.out.println("Dept. teachers:");
    dept.printTeachers();
    dept.setPresident(george);
    System.out.println("President:");
    "+dept.getPresident().getName());
}
Generalization or Inheritance

- Generalization is a special association type, in which a general class is the base for the declaration of one or more, specialized in some sense, subclasses.
- The general class is called super-class, whereas the more specialized classes that inherit from it are called sub-classes.
- Inheritance or generalization is indicated with an empty-head arrow pointing from the specialized class to the base class.

<table>
<thead>
<tr>
<th>Person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- name : String</td>
<td></td>
</tr>
<tr>
<td>- age : int</td>
<td></td>
</tr>
<tr>
<td>+ setName(name : String)</td>
<td></td>
</tr>
<tr>
<td>+ getName() : String</td>
<td></td>
</tr>
<tr>
<td>+ setAge(age : int)</td>
<td></td>
</tr>
<tr>
<td>+ getAge() : int</td>
<td></td>
</tr>
<tr>
<td>+ toString() : String</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>- fieldOfStudy : String</td>
<td></td>
</tr>
<tr>
<td>+ setFieldOfStudy(fieldOfStudy : String)</td>
<td></td>
</tr>
<tr>
<td>+ getFieldOfStudy() : String</td>
<td></td>
</tr>
<tr>
<td>+ toString() : String</td>
<td></td>
</tr>
</tbody>
</table>
An Inheritance Example in Java

- Classes “Person” and “Student” in Java are given in the following slides.

- Observe that the “Student” class is declares as a subclass of the “Person” class with the use of the “extends” keyword.

- Also observe that the “Student” class overrides the “toString” method of its base class “Person”
public class Person {
    protected String name;
    private int age;
    public Person() {}
    public void setName(String name) {this.name = name; }
    public void setAge(int age) {this.age = age;}
    public String getName() {return name;}
    public int getAge() {return age;}
    public String toString() {
        return "I am " + name + " and I’m " + age + " years old.";
    }
}
public class Student extends Person {
    private String fieldOfStudy;
    public void setFieldOfStudy(String fieldOfStudy) {
        this.fieldOfStudy = fieldOfStudy;
    }
    public String getField() {
        return fieldOfStudy;
    }
    // toString student method overrides the base class method
    public String toString() {
        return "I am " + name + 
            " and I’m " + getAge() + " years old. " + " My field of study is" + fieldOfStudy;
    }
}

Student Class

public class Student extends Person {
    private String fieldOfStudy;
    public void setFieldOfStudy(String fieldOfStudy) {
        this.fieldOfStudy = fieldOfStudy;
    }
    public String getField() {
        return fieldOfStudy;
    }
    // toString student method overrides the base class method
    public String toString() {
        return "I am " + name + 
            " and I’m " + getAge() + " years old. " + " My field of study is" + fieldOfStudy;
    }
}

Student Class

public class Student extends Person {
    private String fieldOfStudy;
    public void setFieldOfStudy(String fieldOfStudy) {
        this.fieldOfStudy = fieldOfStudy;
    }
    public String getField() {
        return fieldOfStudy;
    }
    // toString student method overrides the base class method
    public String toString() {
        return "I am " + name + 
            " and I’m " + getAge() + " years old. " + " My field of study is" + fieldOfStudy;
    }
}
Abstract Classes and Concrete Subclasses

- Some classes may be declared as abstract. These classes provide the implementation for some operations, while leaving other operations unimplemented (abstract).

- Some subclasses of these abstract classes will provide the implementation for the unimplemented methods. These subclasses are called concrete.
UML Notation for Abstract Classes

- To indicate that a class is abstract you write its name in *Italics*.
- Also in *Italics*, you write any abstract operations that the class might have.
- In the example figure the class Shape is abstract and has an abstract operation: draw. Class square implements draw and is a concrete subclass of class Shape.
Interfaces

- An interface is a type declaration, in which all operations are abstract.
- An interface doesn’t have any data (state). It also doesn’t have associations with navigability to the opposite end.
  - The lack of navigability stems from the fact that interfaces have no state, because navigability implies the obligation to know the objects associated with an object.
Interface Realization

- A class realizes an interface when it implements all the abstract operations of the interface.
- The “benefit” for the implementing class, is that its objects are now considered also objects of the interface type, and can use services offered by other objects, to objects of this type.
An interface in UML is denoted with a class symbol with the stereotype `<<interface>>` over the class name.

Property compartment may be absent or otherwise it will be empty.

An alternative notation is a small empty circle with the name of the interface below it.
UML Notation for Interface Realization

- A class realizing an interface, is associated with the interface with the realization symbol which is similar to the inheritance symbol, except that the line is doted.

- Alternatively we can use the lollipop notation
A class can realize more than one interface.

Also classes can depend on different interfaces provided by the same class.

In the figure, class Baby provides two interfaces: CryingObject and CuteObject. Class Speaker depends on CryingObject, while class Camera depends on CuteObject.

Dependency is indicated with a dashed arrow from the client of the dependency to the source of the dependency.
Advantages of Using Interfaces

Partitioning operations of a class to separate interfaces, and the controlled way that other classes depend only on selected interfaces, is one of the fundamental techniques that can be used to manage the complexity of modern software systems.

The basic reason for that, is that since a class only uses a well defined subset of operations of a provider class (those provided by the specific interfaces that it uses), the provider class implementation can change as long as the interface and the semantics of operation usage (operation contracts) remains stable.
Example Using Interfaces

- In the figure a class Person implements the Java Comparable interface.
- PriorityQueue, another Java class, has a method “offer” for putting objects in the queue.
- As you can see PriorityQueue only depends on Comparable and no the specific Person class. Objects of class Person can be inserted in the queue, because the Person class implements Comparable.
Person Class in Java

import java.util.PriorityQueue;
import java.util.Iterator;

public class Person implements Comparable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // A possible implementation of compareTo in which persons are ordered in ascending
    // order, and persons with the same age are ordered in alphabetical order
    public int compareTo(Object obj) {
        Person other = (Person) obj;
        if (other.age == this.age) {
            return name.compareTo(other.name);
        }
        return this.age - other.age;
    }

    ...
public static void main(String[] args) {
    //A priority queue of Persons
    PriorityQueue<Person> pq = new PriorityQueue<Person>();
    //Inserting three persons in the queue
    Person p3 = new Person("Natalia", 39);
    Person p1 = new Person("George", 39);
    Person p2 = new Person("Anastasia", 2);
    pq.offer(p1);
    pq.offer(p2);
    pq.offer(p3);
    //Displaying the objects in the queue
    Iterator<Person> i = pq.iterator();
    while (i.hasNext()) {
        Person p = i.next();
        System.out.println(p);
    }
}

The result of the program execution is:
Age: 2, Name: Anastasia
Age: 39, Name: George
Age: 39, Name: Natalia
Template (or Parametric) Classes

- Sometimes we might want to define a class in which a type of its properties will be a parameter of the class.
- A classic example of this, is different kinds of container classes, such as lists, sets and queues.
- For example, if we were defining a priority queue, we would probably like to do it without committing to the specific type of elements that will be inserted in the queue.
- Advantages of doing so include the following:
  - Reuse of the same container class with elements of different types
  - Strong type checking during compilation
UML Notation for Template Classes

- UML notation for template classes is a small dashed rectangle on the top left corner of the class box, where we put the parameters of the class.

- If the data type of a parameter is omitted, it is considered as a data type of the system or the programming language that will be used. If this is not the case then we must write the data type.

```
T
N:int

Array
- elements[1..N] : T
+ add(element : T)
```
Bound Elements

- Template class can’t be used for the creation of object instances. The parameters of the template class, must be bound first to specific data types and values, in order to become concrete classes from which objects can be created.

- We can’t create, for example, an array of objects of type T. You have to create an array with elements of a specific type (e.g. Person). These concrete classes are called bound elements.
UML Bound Element Notation

Unnamed Bound Element

Named Bound Element

Array
- elements[1..N] : T
+ add(element : T)

(Person, 50)

PersonArray
- elements[1..50] : Person
+ add()
Aggregation

- Aggregation is an association of a class with another class that is a part of the first: a whole-part relationship.
- Aggregation has only one additional constraint compared to a casual association: the cyclic association of the part back to the whole is disallowed.
- Aggregation is depicted with a white diamond on the side of the whole.

«Η συναρμολόγηση είναι αυστηρά χωρίς σημασία. Σαν αποτέλεσμα σας συνιστώ να την αγνοήσετε όταν σχεδιάζετε τα δικά σας διαγράμματα»
Martin Fowler
Composition

- Composition is a much stronger whole-part relationship than aggregation, with two additional semantic elements:
  - The whole contains its parts exclusively (no sharing)
  - There is a life-and-death relationship between the whole and its parts (parts are created after the whole has been created and are destroyed with it).

- UML symbol for the composition association is a black diamond on the whole side.
Qualified Associations

- A qualified association uses a qualifier in one side of the association, to determine the objects participating in this association, to the other side.

- In the figure the object Student is the qualifier and for each student there are 0 or 1 lines in the Class Register. We call the association between the Class Register and its Lines a qualified association, with the qualifier being objects of type Student.
Object Diagrams

- An object or instance diagram, is a snapshot of the system at some particular moment, in which we depict the objects as well as the links between them.

- It is an example of the arrangement of objects during runtime.
  - The term “Object Diagram” is not an official term of the UML, but it is widely used.
Valid object diagram

Invalid object diagram
Dependencies

- Dependencies are depicted with dashed arrows connecting a classifier from which they start and is called the client of the dependency, with another classifier called the source of the dependency.
- If the source changes then this might cause changes to the client, which depends on the source.
- There are several predefined types of dependencies in UML. If, for example, the classifiers are packages, and we want to say that the client imports the source, we use the «import» dependency instead of the more general «use» dependency. If the client is a class calling a method on the source, we use a «call» dependency.
- Dependencies ARE NOT transitive.
Dependency Example

- Class Course depends on Student, because it calls the getName method.

- To depict the fact that the dependency exists because the client calls a method on the source, we put the «call» keyword on the dependency line.
Interaction and State Diagrams
What are the Interaction Diagrams?

- Interaction diagrams are used for the visual representation of the different use case scenarios of a system.
- They depict a scenario as a set of object instances, interacting with one another by exchanging messages.
Types of Interaction Diagrams

- There are two types of interaction diagrams: the sequence diagram and the collaboration diagram (or communication diagram in UML 2.0)
- Sequence diagrams put emphasis on the time sequence of the different messages.
- Collaboration diagrams depict the links between the objects, and in order to provide the time sequence they use numbering on message lines.
Objects, Lifelines and Stimuli

- Sequence diagrams visualize the collaboration between objects. Objects are depicted as boxes in which we write the name of the object followed by the class name. The two names are separated with a colon. We may omit the name of the object, in which case we still put the colon in front of the class name.

- Underneath each object a dashed line is extended, called the lifeline of the object. If we need to show the termination of the object we use the symbol ×.

- Objects are written from left to right on the horizontal axis.

- Objects exchange messages, called stimuli.
Types of Stimuli

- **Operation call**: when an object calls an operation on another object.
- **Signal**: when an object sends a message asynchronously on another object.
- **Object creation**: a stimuli that results on the creation of a new object. This message ends on the box of the object being created.
- **Call return**: a dashed arrow depicting the return from a previous operation call.
- **Object destruction**: a stimuli ending on the termination symbol of another object.
A sequence diagram example

- Secretary
  - programming: Course
- CourseRecord
- Assignment
- bestRec: CourseRecord
- s: Student

getBestName() --> [*[για όλα τα course record] average()]

name := getName()

s := getStudent()

average

mark

* [για όλα τα assignments] getMark()

πλαίσιο Ενεργοποίησης

κλήση Μεθόδου

επανάληψη και συνθήκη επανάληψης

επιστροφή με διακεκομμένο βέλος

επιστροφή με πιστροφής από τη κλήση της μεθόδου
Remarks on Sequence Diagrams (1)

- Sequence diagram does not depict the logic of an operation (its algorithm)
- Activation frames are important since they depict the depict program flow from object to object, with method call executions included in other executing methods.
- Each object calling a method in another object, must be linked to that other object, either permanently (with an association of the respective class on the class diagram), or have a transient relationship (e.g. the called object might be a local variable or parameter)
Remarks on Sequence Diagrams (2)

- Operations being called on the different objects, must be operations of the respective object classes of the class diagram.

- In relation to the previous comment, we must say that assigning responsibilities to objects is a key issue in object-oriented design.
  - We will discuss later some empirical rules for object responsibility assignment called GRASP (General Responsibility Assignment Software Patterns).
Object Creation Notation

The creation of a new object is depicted with a message being sent to the created object, that ends on the box of the new object being created.
Object Termination

- Object termination is depicted with an \( \times \) at the end of its lifeline.

- If the termination of the object is caused by another object, we can show this with a signal sent by this other object and that ends to the \( \times \) symbol.
Self-call

- When an object calls a method on itself, this is depicted with a call that starts and ends on the same object.

- The activation frame of the self-call in this case, is contained entirely on the activation frame of the operation that did the self-call.
Conditional Messages

- Sometimes it is useful to say that a message will only be sent if a condition holds. In these cases the message is preceded by the condition inside brackets.

- In a conditional message, if we also want to show an alternative path (what happens if the condition is false), we use two conditional messages starting from the same point and we put the [else] keyword on the alternative message. The two messages are mutually exclusive.
Collaboration Diagrams

- Collaboration diagrams are semantically equivalent to the sequence diagrams. The difference is that with collaboration diagrams, objects can be placed freely within the available space (they don’t have to be placed from left to right).
- Message exchanges must be numbered in order to depict their order.
- To further clarify which calls are included in the execution of other calls, we use a dotted decimal numbering scheme, such as in 1.2.3, which means that the third call is included in the second, which in turn is included in the first.
Links and Messages

- Collaboration diagrams also show the links between the objects, including transient links.
- Messages are shown as stimuli lines on top or near the links through which they are sent.
Collaboration Diagram Example

1. getBestName()

1.1. *[για όλα τα course record] average()

1.1.1. *[για όλα τα assignments] getMark()

1.2. s:=getStudent()

1.3. name:=getName()

programming : Course

bestRec : CourseRecord

: Student

CourseRecord

: Assignment

1.1.2. average

1.1.1.1. mark
A state diagram shows the dynamic behaviour of the objects of a class and the way that their state is changing as a reaction to events.

State diagrams are typically used to model the behaviour of the instances of one class.

We don’t usually draw state diagrams for all the classes of a system. We only use them for these classes that have a very dynamic behaviour. State diagrams help on clarifying this behaviour.
State Diagram Example

Display Time
- switchOn / hrs=0, mins=0
- do/ displayTime
- exit/ saveTime

- set pressed / resetTime
- cancel pressed / restoreTime

Set Time
- + pressed / hrs++
- - pressed / hrs--

Set Hours
- entry/ blinkHours
- set pressed

Set Minutes
- entry/ BlinkMinutes
- set pressed
- - pressed / mins--
- + pressed / mins++
Start and End States

- You can use a filled circle pseudo-state to depict the start state, and a filled circle included in an empty circle for the end state.
- In the figure we can see the not-so interesting life of a human object (we have omitted many details of the composite state living!)
Concurrent states

- In some cases we want to show concurrent state diagrams, to which an object is in two or more states at the same time.
- For this we can split one state in two or more parts with a dashed line, and show the concurrent parts in these two different segments.