CHAPTER 2

OBJECT ORIENTED SYSTEMS

Object-orientation is increasingly being used in several areas. Some of the related terms are interpreted differently in different context in which they are used. The goal of this chapter is not to give a complete presentation of the object-oriented technology, nor to unify the different object models, but to define the object-oriented framework in which our research is settled. First, the vocabulary used in this thesis has been defined and then use of these concepts has been clarified. This terminology is adapted from the object management group terminology [125]. Our main thrust in this chapter is on defining a common vocabulary. Special attention has been paid to the features of procedural object-oriented languages with particular attention to the topics that affect the testability of the software directly or indirectly.

2.1 THE OBJECT-ORIENTED PARADIGM

The basic principle on which most of the object-oriented paradigm is based, is the assumption (or intuition) that most of the real world applications can be developed by specifying, designing, and developing a software system in terms of objects. This assumption is justified by the fact that computer programs model real world entities together with their interactions, and users generally view their environment in terms of objects. It may be generalized that, we apply the object-oriented paradigm every time we think about software systems in terms of objects and interactions between them. It is very difficult to draw a borderline between what can be considered object technology and what cannot. There exist different degrees of object-orientation, and different classifications have been proposed to reflect how much a system implements the object-oriented paradigm. The presented classifications differ in the concepts they are based upon. We have presented the classification of Shriver and Wegner [39], which is based on features. Shriver and Wegner define three levels to which a language can belong, according to its features:
2.1.1 Object Based

These type of systems support the notion of objects in which the system is represented as encapsulated entities and states of the system are altered by the application of methods. The languages for these systems support features for defining objects, which are entities characterized by an encapsulated state and functionalities for accessing and/or modifying it.

2.1.2 Class Based

The object-based systems where each object is an instance of a class, a template from which objects with unified behaviour can be created. The languages for these systems support features for defining classes: classes are implementations of abstract data types (ADTs) and represent object templates. i.e., objects are instances of classes.

2.1.3 Object Oriented

This category consists of the languages which are class based languages and provide the possibility of incrementally defining new classes from earlier defined classes. The concepts of polymorphism and dynamic binding are mainly supported by these languages.

This taxonomy is arguable because it is based on the features of programming languages. It is challenged by many, and recent studies have been trying to capture the essence of object-oriented systems in terms of key concepts rather than language features.

It is also possible to identify a set of basic features provided by object-oriented systems, called, data abstraction, encapsulation, information hiding, inheritance, polymorphism, and dynamic binding.

Data abstraction refers to the possibility of defining objects as implementations of abstract data types.

Encapsulation refers to the ability of enclosing related data, routines, and definitions into a single entity. Object oriented systems integrate both the structural (data) and behavioural (procedural) aspects of a program, while structured systems force an artificial separation of the structure from behaviour. The separate treatment of structure and behaviour complicates design, specification, modification, and semantic
Object Oriented Systems

integrity analysis. Encapsulation allows programmers to modify the implementation of an object while avoiding the creation of unwanted side effects in other objects by hiding the implementation detail behind a public interface (protocol). This reduces complexity by ensuring that changes to the internal operations of an object, i.e., those that do not modify the public interface, are contained within that object.

Information hiding refers to the possibility for the programmer of specifying whether one feature, encapsulated into some module, is visible to the client modules or not: it allows for clearly distinguishing between the module interface and its implementation.

By supporting both encapsulation and information hiding, a language allows for defining and implementing abstract data types (ADTs).

Inheritance mechanism is considered important due to the potential for reuse. Inheritance allows programmers to define objects incrementally by reusing previously defined objects as the basis for new objects. There have been many different types of inheritance mechanisms associated with object-oriented systems. The most common inheritance mechanisms include different forms of single and multiple inheritance. Single inheritance allows a subclass to have only a single parent class. The subclass extends the parent's definition. For example, when defining a new type of student, e.g., UG student, the new student type can inherit the common characteristics of being an student from a generic type of student. Using this approach, the programmer only needs to be concerned with the difference between UG student and generic student.

Existing programming languages and object-oriented methodologies permit extension. Multiple inheritance occurs when a subclass may inherit from more than one parent class. In this situation, the types of inheritance conflicts are multiplied. In addition to the possibility of having an inheritance conflict between the subclass and one (or more) of its parent classes, it is now possible to have conflicts between two (or more) parent classes. Inheritance conflicts increase the difficulty of understanding an inheritance structure and individual objects in the structure. Thus, there is a risk of increasing the complexity of an object-oriented system instead of decreasing it through the use of inheritance. The programmers must be aware of the effects of the modification not only in the super class, but in each subclass that inherits the modification. Inheritance also allows for defining ADTs which derives some of their features from existing ones: in general, there exist different inheritance schemas, but...
for the languages under consideration, we may safely consider that the inheriting ADT is only allowed to add new features and/or redefine some features of the parent ADT.

**Polymorphism** allows program entities to refer to objects of more than one type at run-time. Polymorphism means having the ability to take several forms. In object-oriented systems, polymorphism allows the implementation of a given operation to be dependent on the object that "contains" the operation. For example, a "compute-dues" operation can be implemented differently based on the student's object type, e.g., under-graduate, post-graduate or doctorate. When a new type of student is created, e.g., student, the programmer simply creates a new type of student object and a new "compute-dues" operation in the new object. The "compute-dues" operations of the other types of students are not affected by the dues calculation operation implementation required for the new type of student. This reduces complexity by isolating the effect of changes and providing highly consistent semantics across the interfaces to all employee objects. Thus, polymorphism is a mechanism that can be used to control the complexity of an object-oriented system. In contrast, structured systems often have all compute dues operations contained in one program. The program must be capable of differentiating between the different types of students and applying the appropriate operation. Adding new types of students may require existing code to be changed.

The use of polymorphism also can increase the complexity of an OO system. For example, if the "compute-dues" operation in one type of student objects is implemented to print student descriptive information or some other function, i.e., not compute the pay of the student, then semantic consistency across the interfaces of the student objects no longer exists. The programmer may no longer assume that all operations with the same name perform the same generic function. The semantics of each individual implementation of an operation must be determined. This increases the difficulty of understanding the student objects. When used this way, polymorphism can lead to the same type of software engineering problems created by the unconstrained use of goto statements. The key to controlling an OO system's complexity through the use of polymorphism is to ensure that all operations with the same name are semantically consistent.
Dynamic binding is the ability of resolving at run-time the identity of an operation called on a polymorphic entity. In a language featuring dynamic binding the result of applying an operation to a polymorphic variable (i.e., the operation actually called) depends on the actual type of the object the variable is referring to at the time of the call.

2.2 OBJECTS

As implied by its name, the main components of an object-oriented system are objects. An object-oriented system is a set of objects which interact with each other in many ways. The scope of the term object is very broad: each entity in a system can be considered as an object [41]. Generally, an object is an item which represents a concept that is either abstract, or depicts an entity of the real world [2].

An object defines an abstraction. An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer [42]. Thus an object is mainly characterized by the meaning it has to its users, i.e. by the services that its client can request, independently of its data structure or its implementation. This process of singularizing out the elements of an abstraction in a public contractual interface and a private implementation is called encapsulation. The benefit of isolating clients from the details of an object’s implementation is to allow modifying the implementation of an object without having to change the clients.

An object is usually made up of three properties: services, a state and an identity.

**Services:** The services of an object represent the functionalities provided by the object. Services are commonly called methods, and can be used for either altering or inspecting the state of the object they belong to. They include the subprograms and its methods. The methods represent its behaviour and can give information on an object or alter its state.

**State:** The state of an object is characterized by the values associated to its attributes. Attributes can be either primitive types or references to other objects. From a theoretical viewpoint, the state of an object should be modifiable and inspected only through the services it provides. The state of an object remembers the effects of the services and consists of a set of attributes. In pure object-oriented systems, which do
not admit other entities than objects. The attributes are all connected or embedded objects. In hybrid object-oriented models, which also admit the existence of entities without identity such as natural numbers or booleans, the attributes can also be values.

**Identity:** The identity is an intrinsic property of objects, which assures that two different instantiations of a class are always distinguishable, even if their states are identical. It allows identifying an object independently of its state. Objects are the core of object-oriented systems. They represent the entities composing the system, whose interactions and characteristics define the behaviour of the system itself. Intuitively, an object can be seen as an abstraction representing a real-world entity or a conceptual element.

Objects are characterized by their attributes and methods which are denoted as its features, members or properties.

Objects don’t have a specific name. They are referenced through special entities called references. Here the term reference is used in a general way, without considering how such reference is actually implemented (e.g., pointers in C++ [44] or actual references in Java [43]).

The different phases of the life cycle of an object can be easily identified. All the resources which are needed by an object are allocated to it at the time of its creation and its initial state is defined. Then, the state of the object is altered by its methods or member functions. After the invocation of each method, the object reaches a new state (may be same as the original state). When the object is not required anymore, it is usually destroyed and the resources it is using are released.

### 2.3 CLASSES

In object-oriented methodology the programmers are provided with specific constructs which allow, to define abstract data types and to classify set of object sharing both a common structure and the same behaviour. Terminologically this type of construct is called class. It is provided by most object oriented languages, like Java, C++, Eiffel.

Classes may also be considered as typed modular templates. The definition of a class implies the definition of a new type. The definition of a class includes the number and type of attributes along with the methods which operate upon them.
As an example, Figure 2.1 shows the definition of a Java class representing a generic student, characterized by its rollno and its avgmarks.

```java
class student {
    int rollno;
    float avgmarks;

    student () {
        rollno=0;
        avgmarks=0.0;
    }

    student (int r, float m) {
        rollno=r;
        avgmarks=m;
    }

    void setrollno (int r) {
        rollno=r;
    }

    void setavgmarks (float m) {
        avgmarks=m;
    }

    int getrollno() {
        return rollno;
    }

    float getavgmarks() {
        return avgmarks;
    }
}
```

This class has two attributes of type integer and float which represent the rollno and avgmarks of a student. It also provides six operations, namely, student(), student(), setrollno, setavgmarks, getrollno(), and getavgmarks().

With the help of classes both encapsulation and information hiding are enforced. Encapsulation is achieved by tracing a crisp line between interface and implementation. The clients of a given class only need to know its interface, which is represented by the operations they can invoke on that class, while they are in general not concerned with the way such operations are implemented. Information hiding is achieved by separating the private part of a class from its public part. Programmers
Object Oriented Systems

are provided with constructs that allow them to declare some members of the class as only accessible by objects belonging to such class, and some other members as publicly accessible. Different object-oriented languages follow different rules of information hiding. In general there exist many different intermediate levels of visibility for the members of a class.

2.4 METHODS

In a class methods represent operations of ADTs. They are characterized by their name and their signature. Where the signature of a method is represented by the number and the type of its formal arguments and by the type of the value returned by the method (if any).

In some languages, pre and post-conditions for methods can be provided. And these conditions must be satisfied prior to or after the execution of the method, respectively. This feature is supported by Eiffel.

Methods may be categorised in different categories, depending on their purpose:

2.4.1 Constructors

They provide the specific instance of the class they are invoked upon, i.e., the newly created object, with meaningful initial values. There are different kinds of constructors: default constructors are constructors that take no arguments and usually initialize the object with default values (if not provided, they are usually automatically generated when a class is defined); copy constructors define how to initialize an object starting from another object of the same class and they take a reference to the object they must “copy” from: all the other constructors have a variable number of arguments, whose values are used to initialize the newly created object. In the Java language, constructors are methods having the same name as the class. For example, the class Student in Figure 2.1 provides two constructors, Student () and Student (int r , float m ) where the former is the user defined default constructor and the latter is a generic constructor taking two values it uses for initializing attributes rollno and avgmarks.
2.4.2 Observers

They provide their caller with information on the state of the object they are invoked upon. As their name suggest, these methods do not modify the state of the object. Methods getrollno and getavgmarks are observers of class student.

2.4.3 Modifiers

Their purpose is to allow for modifying the state of the object they are invoked upon by changing the value of its attributes. Methods setrollno and setavgmarks shown in Figure 2.1 are modifiers of class student, since they allow for setting the value of attributes rollno and avgmarks respectively. A method can be an observer and a modifier at the same time, when it allows for both modifying and inspecting the state of an object.

2.4.4 Destructors

Destructor is a special method which is invoked when the object of the class is no more required. They are in charge of performing specific actions when an object is no longer needed and it is thus eliminated. Usually, destructors are used for freeing resources allocated to the object being destroyed. In the case of languages without garbage collection, where the programmer is in charge of both allocating and deallocating memory, destructors are usually devoted to memory management. In the Java language a destructor is a method identified by the name finalize and it is invoked automatically when the object is no longer required.

2.5 ATTRIBUTES

Attributes, or instance variables, define together with their values the state of a class. As an example, the state of an object of class student, shown in Figure 2.1, is defined by the value of attributes rollno and avgmarks.

Attributes should always be given controlled access to enforce information hiding. They should never be directly accessible by client classes. Methods should be provided for both inspecting and defining attributes when needed, as done for class student of Figure 2.1.
In this case, the attributes of the class student are represented by pre-defined, or scalar, types of the language. In the general case, an attribute can also be a reference to another object. In such a situation, the state of the object is given by both the value of its scalar attributes and the state of the objects it references.

2.6 RELATIONSHIPS

Classes don’t exist in isolation; they mutually collaborate in several different ways. All object-oriented systems are characterised by relationships among classes. Relationships define the nature of the interactions among classes, and allow for identifying subsets of the systems which are closely related, the so-called clusters.

Relationships can be classified with respect to the role played by the involved classes. A first classification distinguishes between client-supplier and hierarchical relationships. It is possible to further classify relationships belonging to the main categories. We present a classification similar to the one proposed by Booch, Jacobson, and Rumbaugh [13].

2.6.1 Client-Supplier Relationship

Client-supplier relationships are ordered relations between two objects, such that the first object (the client) uses the services provided by the second object (the supplier) to operate. In a client-supplier relationship the client usually owns a reference to the supplier. This kind of relations are characterized by the two roles played by the classes involved in the relation, and by two cardinalities which indicate how many objects can participate in the relation on each side. Client-supplier relationships may be further categorized under four categories:

Association

They define generic relations between two classes. The relationship between a class teacher and a class student is an example of association. In this example, on the teacher side the role is teach and the cardinality is 1-n (teacher teach students), while on the student side the role is learn and the cardinality (students learn from professors).
Aggregation

Also known as has-a or part-of relations. This relation holds between two classes when an object of the first class can include one or more objects of the second class. An example of aggregation can be the relation between a company and its employees.

Composition

This is a stronger version of aggregation, characterized by objects of the client class being “physically” composed of one or more objects of the supplier class. Moreover, the contained objects can belong to only one whole and do not have a proper existence outside their container. The relation between a car and its wheels can represent a typical example of composition.

Generalization

This relationship ("is a"). it indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other (the super type) and superclass is considered as 'Generalization [85]' of subclass. In practice, this means that any instance of the subtype is also an instance of the superclass. An exemplary tree of generalizations of this form is found in binomial nomenclature: human beings are a subclass of simian, which are a subclass of mammal, and so on. The relationship is most easily understood by the phrase 'an A is a B' (a human is a mammal, a mammal is an animal).

2.6.2 Hierarchical Relationships

Hierarchical relationships reflect a fundamental aspect of object oriented languages, i.e., the possibility of defining the implementation of an ADT starting from an existing one. This feature, typical of this kind of systems is known as inheritance. Inheritance is an ordered relation with well established roles. When two classes participate in an inheritance relation, the inheriting class is defined as a subclass, a specialized class, a heir, or a descendant, while the inherited class is defined as a superclass, a parent class, or an ancestor. This kind of relation is also known as generalization or specialization relation. In the most popular object-oriented languages, inheritance can be used for two different purposes. It is a way of re-using code, and it is also a way of defining a type hierarchy.
To avoid ambiguity, we distinguish between these two different kinds of hierarchical relationships:

**Subclassing:**

It is more a syntactic than a semantic relation, which reflects the idea of inheritance as a means for supporting reuse. With subclassing, it is possible to take advantage of the features provided by one (single inheritance) or more (multiple inheritance) existing classes and to specialize them for specific purposes. Specialization is performed by either adding new properties to the class, or redefining (overriding) existing properties (in some languages it is also possible to remove properties).

**Subtyping:**

It is also indicated as is-a relationship. It is a semantic relationship between classes seen as types, which indicate a conformance of behaviour between the two classes involved in the relation. Subtyping is a relationship between specifications, while subclassing is a relationship between modules. Intuitively, a class is a subtype of another class when it can be used in any place where such class is used, i.e., when it can be substituted to it. As for subclassing, subtyping can be either single or multiple. Starting from subtyping relations it is possible to organize classes within a hierarchical tree, where each type has its subtypes as direct descendant. The root of such hierarchical tree represents the unique (if any) higher class in the type hierarchy. If the type hierarchy is semantically well formed, a subtype can be used in any context where one of its supertypes is expected. The concept of substitution has been rigorously expressed by Liskov [45] in two forms, namely, weak and strong.

### 2.6.3 Abstract Methods and Abstract Classes

In some cases the partial definition of a class is provided. In these circumstances, an interface may be appropriate, since it does not allow for defining attributes and methods, but only to declare method interfaces. Most modern object oriented languages support the feature of defining abstract methods and abstract classes. An abstract method is a method which is only declared, but not implemented (just like it happens with interfaces). A class providing one or more abstract methods is an abstract class. Abstraction is an useful mechanism to provide a common interface for a set of classes in mutual hierarchical relation, and such classes share a sub set of their behaviour. In this case the abstract class only provides the common behaviour and
implementation of abstract functionalities is provided by its heirs. Abstract classes are only partially implemented, so they may be referenced only, but they cannot be instantiated.

2.7 COUPLINGS OR DEPENDENCIES

Large systems are generally built by following modular approach. The smaller units make the system easier to understand, manage the complexity, to define development tasks and to assign them to different teams. Definitely, breaking a system into smaller unit will lead to dependencies between these units. Many of the software units within a large system depend on other units to provide their functionality.

Not only software modules are dependent on one another, but also other entities like requirements document, specification document and design document may be dependent on one another. These kinds of dependencies among different modules and other entities affect the testability of the system as whole.

2.7.1 Dependency

It may be defined as relationship where two entities are related to one another if changes in one entity affect other entity also. This type of dependency is sometimes called as semantic dependency between two entities.

Two objects or two classes may also depend on one another syntactically, when one has syntactic reference to other. It is not necessary that a semantic dependency lead to a syntactic dependency in all cases.

The dependency of entities on each other (direct or indirect) affects the testability of the software. A deep understanding of these effects is necessary to compute and improve the testability of the software under consideration.

2.7.2 Dependency Cycle

The dependency cycles should be avoided as far as possible. A dependency cycle severely affects the testing and the response to the dependency cycle should be quick and the cycle should be broken. If dependency cycle is not broken, it will be very difficult to test the classes individually and they will have to be tested together. Additional stubs will be required if the cycle is broken.
2.7.3 Effect of Dependencies on Testing

2.7.3.1 Effect of Dependency on Test Order

Test cases when executed in a proper order minimize the number of stubs and drivers during unit and integration testing. The test order to be followed is dependent on test the inter-class dependencies and the integration strategy.

If classes and dependencies can be arranged in the form of a tree, the more appropriate test order may be identified. Otherwise there is vast variety of ways in which the test order may be selected. Introduction of more dependencies reduces the choices in test order execution. These additional dependencies may introduce dependency cycles and may make it difficult to find an optimum test order [17] [18].

2.7.3.2 Effect of Dependency on Test Cases

Test cases are designed to uncover quickly as many errors as possible. Test case should exercise the code by using and producing the inputs and outputs which are correct and incorrect. For object oriented software under test, each test case should describe the initial set of classes, initial state of the objects, input arguments, local and global variables, the method invocations and the possible outputs like states of objects, values of local and global variables and exceptions thrown. It is next to impossible to design test cases to cover all possible input combinations. So, generally a small but sufficient set of test cases is selected which has a high probability to find most of the errors. If a new dependency is added between child class and parent class, the behaviour of the child class becomes more unpredictable and more number of black-box test cases are required for testing two classes.

2.7.3.3 Effect of Dependency on Stubs and Drivers

Stubs and drivers are used as substitutes for classes (or interfaces). Stubs allow to test a class in isolation or to substitute classes which haven’t been tested yet. Drivers are used to control the class under test directly the need to implement stubs and drivers depends on the integration strategy chosen. With the introduction of new dependencies the number of stubs required to test classes in isolation also increases. Dependencies introducing dependency cycles require the implementation of stubs if the strategy is to break the cycles during testing.
2.7.3.4 Effect of Dependency on Objects Initialization and Instantiation

Objects of the classes involved in a test are instantiated and usually initialized during test setup. Initialization may include the establishment of appropriate object links. Introduction of a new dependency to a supplier class often increases the number of objects that have to be instantiated and initialized. For example, if an instance of the supplier class has to be provided as a parameter object to the method under test, the test code has to create this supplier instance. If the class under test creates the instance of the supplier class itself, the test code is freed from this responsibility, but such direct creation of supplier instances can cause other test problems.

2.8 COHESION

Cohesion may be defined as an attribute of a module that refers to the relatedness of its components. A software module with the highest degree of cohesiveness is one which has only one function, which is indivisible further. It is not possible to further divide a cohesive module into further modules. In software engineering cohesion is always known as an important attribute of the design quality. As a member of the software development team we should be able to develop the software modules which are highly cohesive in nature. In fact there should be some mechanism to quantify the cohesiveness of the different modules being designed by the developers. These type of measures may help the developers to design the system in such a way that they are able to design highly cohesive modules which in turn help to improve the testability of the software being developed. Module cohesion has many forms, some of which are described briefly below:

1. Coincidental
2. Logical
3. Temporal
4. Procedural
5. Communicational
6. Sequential
7. Functional

All the above mentioned categories of the cohesion may not be highly desirable in all the projects. According to some researchers the functional cohesion is most important and coincidental cohesion is least important for the developers.
2.9 POLYMORPHISM

The objects in the object oriented languages may belong to multiple classes, this property is called polymorphism. The presence of polymorphic methods poses a great challenge for the testing team. Due to presence of polymorphic methods, it is not possible to find statically which method will be called by a particular object in response to a message. Polymorphism also affects the number of test cases required for a particular module.

The software under consideration may fail during test runs due to the special combinations of the bindings, along a particular execution path. To exercise all interactions among polymorphic objects is really a tedious task. So, need is generally felt to have some measurement tool to measure the degree of polymorphism among modules. This unit is Polymorphism Factor (PF).

The object-oriented programming language which supports polymorphism, objects can belong to more than one type and methods can accept as formal parameters actual parameters of more than one type [14].

Cardelli and Wegner [15] have proposed some different kinds and levels of polymorphism, which are shown below:

![Diagram of Types of Polymorphism](image)

**Figure 2.2** Types of Polymorphism
2.9.1 Ad Hoc Polymorphism

In adhoc polymorphism, the data types to be substituted are finite and small in number and there are no behavioural constraints on subtypes. It may be considered only syntactic polymorphism. Adhoc polymorphism can be broadly categorised as, overloading and Coercion.

2.9.1.1 Overloading

It may be defined as the simplest form of polymorphism in which an identifier can identify different functions performing same task syntactically, on different kinds of object, but with different semantics. A “multiply” function operating on integer numbers and real numbers is a typical example of overloading. It is possible to think the overloaded functions as a set of monomorphic functions, rather than a unique polymorphic function.

2.9.1.2 Coercion

It is an operation in which the data types of the function or operation are converted, at a specific place in a program, according to the function invoked at that point of time. Coercion may be invoked implicitly both statically and dynamically, depending on the type system. Coercion can be illustrated by the example of the “multiply” function mentioned above: if we multiply an integer value with a real value, the integer value is converted to a real value and then the version of “multiply” defined for real numbers is invoked.

2.9.2 Universal Polymorphism

It is considered the true form of polymorphism. There are two major forms of universal polymorphism.

2.9.2.1 Inclusion Polymorphism

It is the form of polymorphism which corresponds to the case of a set of different implementations of a method in a type hierarchy. It is called as inclusion polymorphism due to the fact that an object of a given type belongs to all the super types of that type. This object can be substituted in all the situations where an object of one of the supertypes is expected. In the case of a method invocation on a polymorphic reference to a supertype attached to an object of a subtype, the method actually called would be the one provided by that object. This mechanism is called
Object Oriented Systems

dynamic binding, which can be expressed as the ability of choosing the method to be actually executed according to the dynamic type of the object rather than to its static type. The static type of a reference is the one provided in its declaration, while its dynamic type is the actual type of the object attached to it.

If a method has several different implementations within a type hierarchy, it is impossible in the general case to statically determine which actual binding will occur as a consequence of an invocation of such method on a polymorphic entity during execution. The actual binding in such cases is chosen at run-time by examining the type of the actual object the method is invoked upon.

2.9.2.2 Parametric Polymorphism

In parametric polymorphism "the same object or function can be used uniformly in different type contexts without change, coercion, or any kind of polymorphism, run-time test or special encoding of representation" [15]. There are two types of this polymorphism, called syntactic and semantic.

2.9.2.3 Syntactic Polymorphism

It is equivalent to generic units, i.e., units which are parametrized with respect to one or more types. This polymorphism is called syntactic polymorphism because to use generic units, they must be instantiated with actual parameters that are known at compile-time. In such a situation, there is no "dynamic" polymorphism and every invocation in a generic unit can be statically bound. The set of instantiations of the generic units can be considered as a set of monomorphic units, but with a difference with respect to overloading: in this case all the units in the set share a common semantics.

2.9.2.4 Semantic Polymorphism

It is the most genuine form of polymorphism. It does not provide polymorphism through a set of different implementations selected at run-time or at compile-time. On the contrary, it provides a unique implementation which can be uniformly invoked on several different objects. Among the ones presented above, the most common form of polymorphism provided by modern object-oriented languages is inclusion polymorphism, which is the one we address in this thesis.