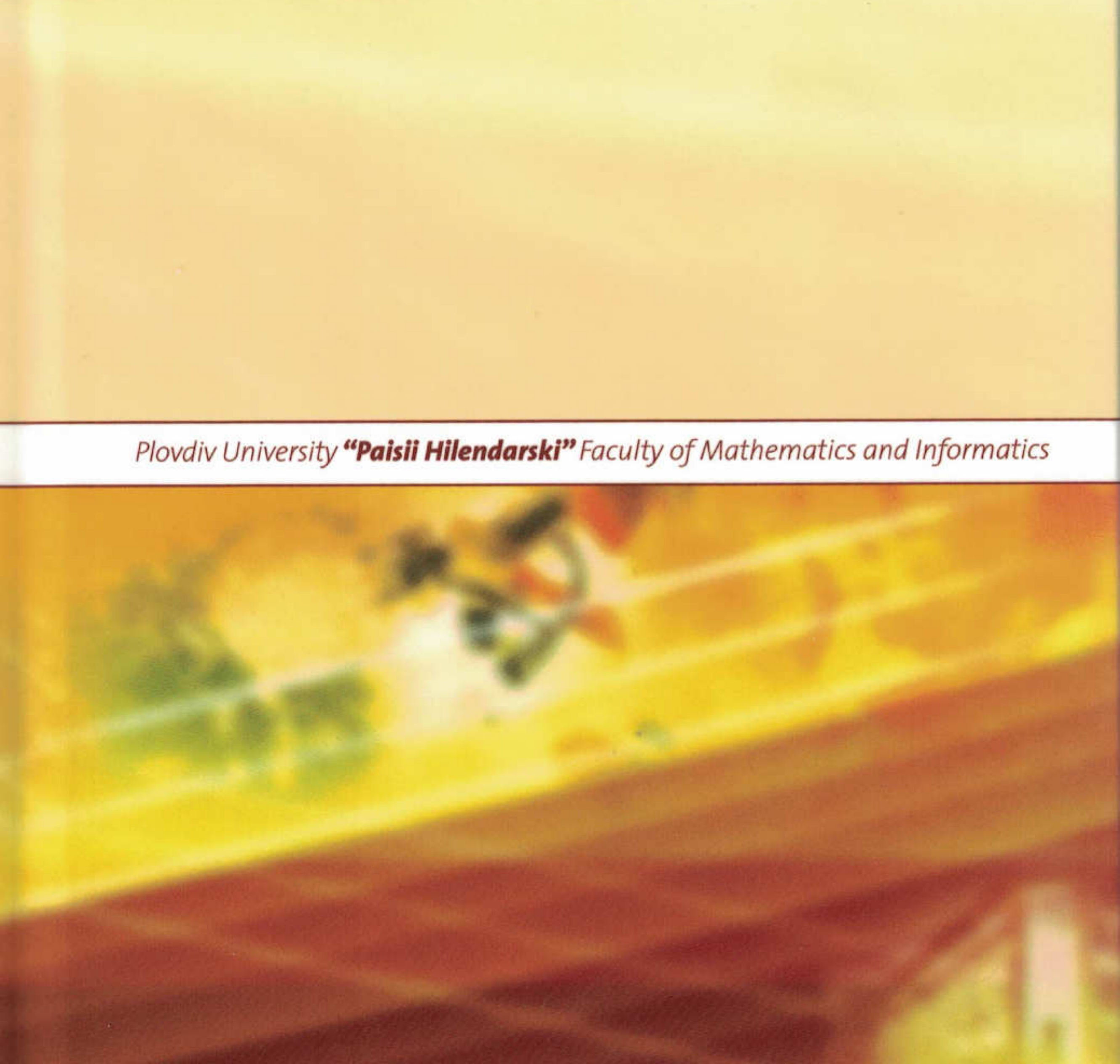
Object Oriented Programming

Asen Rahnev, Nikolay Pavlov, Nikola Valchanov, Todorka Terzieva

Reviewers: Anton Iliev, Angel Golev

Plovdiv University **"Paisii Hilendarski"** Faculty of Mathematics and Informatics



Object Oriented Programming

Asen Rahnev, Nikolay Pavlov, Nikola Valchanov, Todorka Terzieva

Plovdiv University "Paisii Hilendarski"
Faculty of Mathematics and Informatics

WWEDU World Wide Education, Wels

CEUS Center for European University Studies, Wels (Austria)

Object Oriented Programming

Asen Rahnev, Nikolay Pavlov, Nikola Valchanov,
Todorka Terzieva

Reviewers: Anton Iliev, Angel Golev

2013

ISBN 978-3-99034-202-2

Preface

The paradigm of object-oriented programming dates back to 1967, when Ole-Johan Dahl and Kristen Nygaar create the programming language Simula 67. Their work served as a foundation to Alan Key, who created SmallTalk – the first programming language to officially introduce the term *object-oriented programming*. Since then, many existing programming languages gradually adopted object-oriented features.

Recently, a number of object-oriented programming languages have been created, among which the most important are C#, Java, and Visual Basic.NET.

This book is an introduction to object-oriented programming using the C# language. Proper object-oriented design practices are emphasized throughout the book. Readers learn to be object users first, then learn to be class designers. The chapters are planned in a systematic way. Each concept is explained in an easy-to-understand manner supported with numerous worked-out examples and programs. The readers can run the solved programs, see the output, and enjoy the concepts of object-oriented programming.

This book is divided into 15 chapters:

- Chapter 1 describes the basic terms of object-orientated programming such as object, state, methods, attributes, and class.
- Chapter 2 presents encapsulation which is one of the fundaments of the object-oriented concept. Examples illustrate the data hiding.
- Chapters 3 through 7 introduce key features of the object-oriented paradigm such as inheritance, polymorphism, composition, abstraction, abstract classes, abstract user interfaces, constructors, and destructors. Each of the chapters is well structured and illustrates the application of the features by appropriate examples.
- Chapter 8 focuses on tools for building abstractions and implementing complex hierarchical structures.

- Chapter 9 clarifies the scope of variables, objects and classes.
- Chapter 10 makes an overview of exception mechanisms.
- Chapters 11 and 12 emphasize on the indispensable steps and considerations required for completing a good design of classes and systems.
- Chapter 13 illustrates the process of designed object-oriented solution to a small problem by the game Black Jack.
- Chapter 14 and 15 are dedicated to creating object models with UML. Conceptual foundations are provided of class diagrams and relationships in UML. Basic characteristics are covered such as inheritance, association, aggregation, composition, dependency, and interface.

Acknowledgements

We would like to give many thanks to the reviewers of this book provided invaluable feedback and removed many errors from this book.

Съдържание

1. Introduction to Object-Oriented programming

As computers increase in processing power, the software they execute becomes more complex. This increased complexity comes at a cost of large programs with huge codebases that can quickly become difficult to understand, maintain and keep bug-free.

Programming, at a high level, is the process of solving a problem in an abstract fashion, then writing that solution out in code. Object-Orientated Programming (OOP) and Procedure Oriented Programming are basically two different paradigms for writing code; more fundamentally, they are two different ways of thinking about and modeling the problem's solution. In a nutshell, Object Oriented programming deals with the elemental basic parts or building blocks of the problem, whereas Procedural programming focuses on the steps required to produce the desired outcome.

1.1. Procedural versus Object-Oriented programming

Procedural Programming is a methodology for modeling the real world of the problem being solved, by determining the steps and the order of those steps that must be followed so that the desired outcome or specific program state is reached. The building blocks of procedural programming are the procedures also known as routines, subroutines or methods. They are pieces of code that can be called multiple times within our program. When procedures are well designed, they can be reusable thus accelerating the development process, facilitating system modifications and reducing codebase where defects might occur. There are two problems with procedural programming.

The first problem is readability. When the codebase reaches a certain level of complexity it gets hard to read and understand. That is due to the fact that in procedural programming the functionality is global while the data is local. This means that any procedure can be called from anywhere in the code while the data

TEST 3

Inheritance in object-oriented programming:

- ☒ is a technique for defining relationships between objects
- ☐ is a technique for defining the relation between the functions and attributes within a single class
- ☐ leads to code duplication
- ☐ leads to unstructured code

If the class A inherits from class B then:

- ☒ A will inherit all fields and methods from B
- ☐ B will inherit all fields and methods from A
- ☐ A needs to redefine all fields and methods of B
- ☐ nothing will happen because inheritance in object-oriented programming does not work for classes

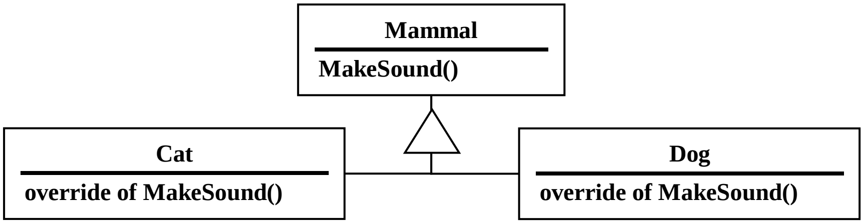
The base class of a hierarchy:

- ☒ usually defines an abstract concept that is later made specific in the leafs of the hierarchy tree
- ☐ defines a concrete concept
- ☐ can’t have any instances
- ☐ inherits fields and methods from the inheritant class

4. Polymorphism

The relationships that inheritance defines between objects allow them to behave polymorphically. The word “Polymorphism” comes from Greek and literally translates to “many shapes”. Polymorphism is tightly coupled with inheritance and is basically the ability of different types of objects to expose the same interface but behave differently when it’s accessed.

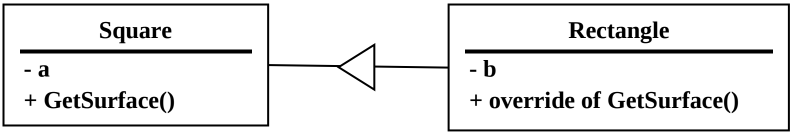
In the standard inheritance model the child inherits all methods that are defined in the base class. Let’s take a modification of our “mammals” example:



Since all mammals can make a certain sound the “MakeSound” method is extracted in the base class as it is a common behavior for all mammals. The tricky part comes here – the cat meows while the dog barks.

Polymorphism is a technique that allows us to replace an implementation of a method defined in the parent class. The process of replacing the implementation of a method defined in the parent class is called “method overriding”. This way the object of a concrete implementation behaves differently when the inherited method is called. In our case the cat will meow and the dog will bark when we call the “MakeSound” method that was originally from the “Mammal” class.

Let’s have another example a hierarchy of shapes:



Say we need to work with squares and rectangles. What we’ll first do is analyze their common behavior. Both shapes have calculable surface so they will both have a “GetSurface” method. The square has four sides of the same length. This can be modeled by an attribute – in our case we have named this attribute “a”. The rectangle has two sides of the same length – “a” and another two sides of the same length – “b”. It is obvious that the square is a subset of the rectangle therefore it can be used as a base class for it.

Knowing this we can start implementing the Square class. We first define the attribute “a” and the method “GetSurface” that calculates the surfaces based on the formula “a*a”. Then we create the second class – rectangle where we add another attribute “b”. The “Rectangle” class inherits “a” and “GetSurface” from “Square”. While we can use the attribute “a” here the “GetSurface” that we inherit “knows” how to calculate the surface of a square – “a*a”. In order to make the rectangle calculate its surface correctly we need to override the “GetSurface” method from the base so that it would calculate the surface using the formula “a*b”.

4.1. Polymorphism and abstraction

The main benefit of polymorphism is the ability to handle different objects in the same manner. Say we have a collection of type “Mammal”. Some of the items in the collection are of type “Cat” and others of type “Dog”. If we iterate through the collection and call the method “MakeSound” of each instance some of them will meow and others will bark depending on the type of the instance.

The same goes for shapes. Say we have a collection of type “Square”. Some of the items in the collection are squares and others are rectangles. When we iterate through the collection and call the “GetSurface” method of each item we will get the correctly calculates surface of the respective shape – square or rectangle.

4.2. Polymorphism in C#

In order to illustrate polymorphism better let's start with a simple example and later extend it to a complex one. Say we want to have a hierarchy of shapes consisting of a "Square" class and a "Rectangle" class. And we need a function that accepts either a "Square" or a "Rectangle" instance and displays the surface of the shape to the standard output.

In the code below we first declare the "Square" class. This will be our base class for hierarchy. Just like in the previous chapter our "Rectangle" class inherits from "Square". The only difference here is that instead of method hiding we're using method overriding to manage the different implementations of "GetSurface". We will later explain why through example.

```
using System;
namespace Polymorphism
{
    public class Square
    {
        public int a;
        public virtual int GetSurface()
        {
            return a * a;
        }
    }
}
```

As in the previous example the "Square" class has one field – "a" (the length of its side) and one method – "GetSurface" that calculates its surface.

Note that the "GetSurface" method is marked as "virtual". The "virtual" keyword practically says that this method is overridden. We will talk more about virtual methods in latter chapters.

In the "Rectangle" class we inherit the "Square" class, extend it with the "b" field and override the "GetSurface" method implementing the formula for rectangle surface calculation.

6. Abstract thinking when designing interfaces

In the object-oriented programming one of the first thought process skills you must master is how to actually think in terms of objects. In the real world, you do thinking in terms of objects and this method of thinking may be easier than you suppose. In this way the software development paradigm is to think in terms of objects, but no thinking of a program as simply code and data.

Object-oriented programming allows for the reuse of classes by implementing inheritance and polymorphism. Reusable classes tend to have interfaces that are more abstract than concrete. An abstract super class can be used by several sub classes. Each of the sub classes can inherit all of the attributes and methods of the super class. The other alternative involves the use of several/many “concrete” sub classes with duplicated code in each sub class. Concrete interfaces tend to be very specific, while abstract interfaces are more general. In actuality abstract interface is more useful than a concrete interface, nevertheless often is not always the case.

As OO designers we should be looking for opportunities to use abstract super classes that contain all of the duplicated code. Note that an abstract class cannot be instantiated and an abstract method in a super class must be implemented in all of its sub classes.

Our purpose is to design abstract, highly reusable classes, and to do this we will design highly abstract user interfaces.

To illustrate the difference between an abstract and a concrete interface we create a taxi object. The example using the taxi class is an excellent demo for the use of abstract classes vs. “concrete” classes. Note that from the abstract class with the “Take me to the Airport” abstract method provides the possibility of sub classes such as SofiaTaxi, RadioTaxi, PlovdivTaxi, etc. In each case the sub class would have its’ own implementation of the abstract method and the end user could get into a taxi in any city and call the Take.me.to.the.Airport method!

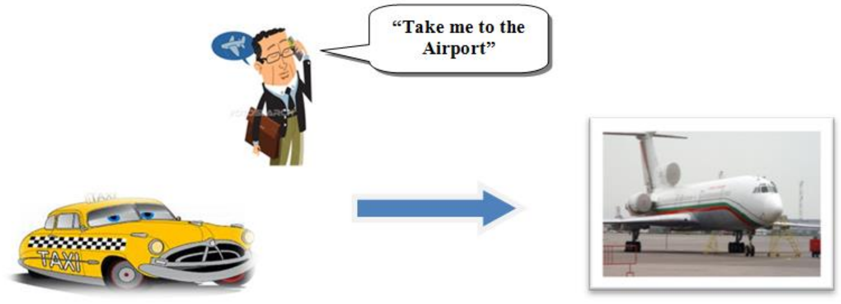


Figure 1. An abstract interface

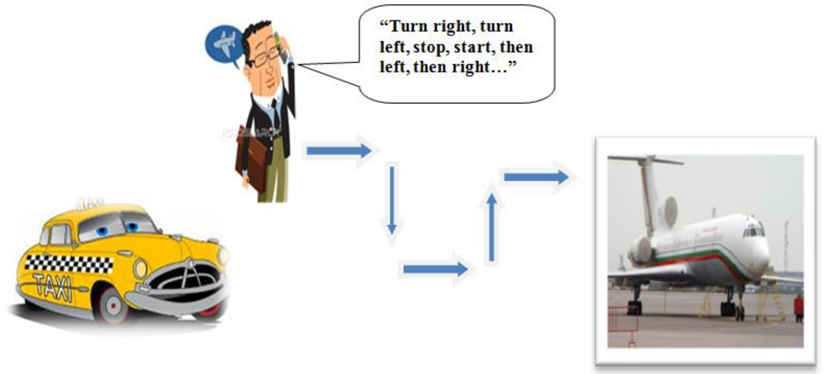

"Turn right, turn left, stop, start, then left, then right…"

Figure 2. A not-so-abstract interface

You come out from your hotel in Sofia, get in the taxi and the cabbie will turn to you and ask, "Where do you want to go?" You reply, "Please take me to the airport." (This assumes, of course, that there is only one major airport in the city. In Plovdiv you would have to say, "Please take me to Sofia Airport" or "Please take me to Plovdiv Airport.") You might not even know how to get to the airport yourself, and even if you did, you wouldn't want to have to tell the cabbie when to turn and which direction to turn, as illustrated in Figure 2. How the cabbie implements the actual drive is of no concern to you, the passenger. Ask yourself which of these

TEST 10

An exception is:

☒ an anomaly that occurs during the execution of a program
☐ a failure caused by the operating system
☐ a failure caused by the hardware
☐ returned by functions that don't have a return type

An exception:

☒ can be handled in such a way that the program can continue working properly
☐ always forces the program to close
☐ crashes the operating system
☐ forces restart of the hardware

In a try/catch/finally construct we can:

☒ have multiple catch sections
☐ have multiple try section
☐ have multiple finally section
☐ have only one try, one catch and one finally sections

11. Class design guidelines

Object oriented design goal is to encapsulate both data and behavior into objects. Then, objects can interact with each other.

In this lesson we are emphasizing on the indispensable steps and considerations required for completing a good class design. Observe that class design represents the real behaviors of the object.

11.1. Modeling real word systems

One of the basic goals of object-oriented (OO) programming is to model real-world systems in ways identical to the ways in which people in reality think. The designing class is the object-oriented method to create these models, which encapsulates the data and behavior into objects that interact with each other.

Again to explain the cabbie example from previous lessons. The `Cab class` and the `Cabbie class model` a real-world entity. As illustrated in Figure 5, the `Cab` and the `Cabbie objects` encapsulate their data and behavior, and they interact through each other's public interfaces.

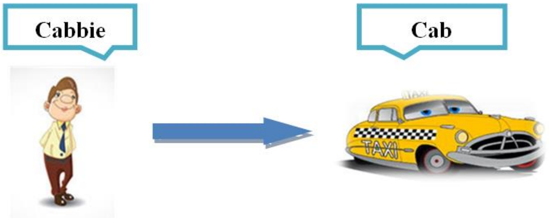


Figure 5. A cabbie and a cab are real word objects

When designing classes, there are three different design errors you can allow. An object is not a repository for data that the rest of your program will use. Your objects should contain both data and the methods that work on that data. An object is not just a collection of methods that you pass data to. If a class has no data elements, and if most of the methods in the class are static, then what you have is a procedural module. An object contains both data and the operations that are performed on the data. OO programming supports the idea of making classes that are complete packages, encapsulating the data and behavior of a single entity. An object must should model one abstraction. Consequently, a class should represent a logical component, such as a taxicab. Farther this lesson presents several guidelines for designing solid classes.

Identifying the Public Interfaces

Probably the most considerably requirement when designing a class is to keep the public interface to a minimum. The entire purpose of building a class is to provide something useful and concise. In evaluating the public interface of your class, ask yourself the following questions: Does the class do everything it needs to do? If programs that use your objects consists of `get()` and `set() methods`, may be your interface is not complete. If there are methods that are not used, don't overburden your interface with them. Providing the minimum public interface makes the class as concise as possible. Does the interface represent a single abstraction? Classes are not collections of autonomous methods. Make sure each method contributes to the expectation behavior of your object. Does each method represent a single operation? The purpose is to provide the user with the exact interface to do the job right. If there is missing behavior, the public interface is incomplete. If the user has access to behavior that is needless or even dangerous the public interface is not properly limited and even trouble with system integrity. Creating a class is a business proposition, and as with all steps in the design process, it is very important that the users are involved with the design right from the start and through the testing phase. In this way, the utility of the class, as well as the proper interfaces, will be assured. The public interface determines how the objects interact.

Look at the cabbie example once again. If other objects in the system need to get the name of a cabbie, the `Cabbie class` must provide a public interface to return its name; this is the `getName() method`. Thus, if a `Supervisor object` needs a name from a `Cabbie object`, it must invoke the `getName() method` from the `Cabbie object`. In effect, the supervisor is asking the cabbie for its name (see Figure 6).

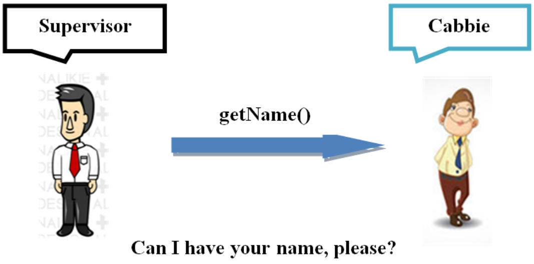


Figure 6. The public interface specifies how the objects interact

Users of your code need to know nothing about its internal workings. All they need to know is how to create and use the object. Give them a way to get in, but hide the details.

Hiding the Implementation

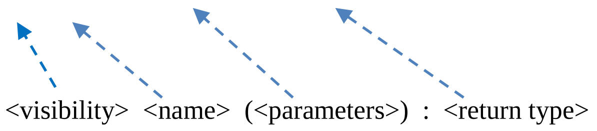
In the previous lessons the necessity for hiding the implementation has been considered in particularity. The implementation must provide the services that the user needs, but how these services are actually performed should not be made apparent to the user. A class is most useful if the implementation can change without affecting the users. In short, a change to the implementation should not necessitate a change in application code. In the cabbie example, the `Cabbie class` might have behavior belong to how it eats breakfast. However, the cabbie's supervisor does not need to know what the cabbie has for breakfast. Thus, this behavior is part of the implementation of the `Cabbie object` and should not be available to other objects in this system. One of the basic rules of encapsulation is that "all fields shall be private". In this way, none of the fields in a class is accessible from other objects.

The most important points in the modeling real-world systems are next:

```
// Attributes declaration
- number: int;
- name: string;
# faculty: string;
```

A class defines a set of objects that have state and behavior.

```
+ GetName() : string
+ GiveNumber(get ; set ;) : int
+ GiveName (get ; set ;) : string
```



14.3. Relationships in UML

There are different kinds of static relationships in UML:

- Inheritance is a Specialization/Generalization relationship between objects of the specialized elements (child) and objects of the generalized elements (parent).
- Association – technically an association is any relationship between two object classes, however association to really mean a dependency on one class to another. Associations are simply services provided between classes.
- Aggregation is a special kind of association. An aggregation is a whole or part-of relationship.
- Composition is aggregation where components cannot exist without the aggregate (parent).
- Dependency expresses the weakest coupling between two classes. Objects work briefly with object of another class.
- Interface/Realization is a semantic relationship that is encountered between interfaces and the classes or components that realize them and between use cases and the collaboration that realize them.

14.4. Inheritance

Inheritance permits software reusability. New classes can be created from existing classes; moreover the attributes (data) and behavior (methods) of the old class are available to the new class. New attributes (fields) and methods can be added to the new class also. A very important concept in object-oriented design, **inheritance**, refers to the ability of one class (child class) to **inherit** the identical functionality of another class (super class), and then add new functionality of its own. Inheritance allows you to write key pieces of code once and use it over and over. Unlike other OO languages C# only allows a class one base class. It does not support multiple inheritance, probably because you can get into a complete muddle with multiple inheritance.

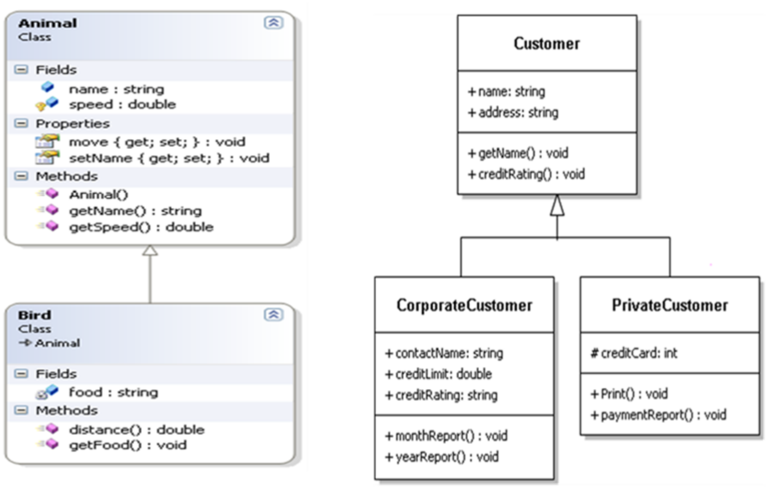


Figure 23. UML class diagrams representing generalizations

To model inheritance on a class diagram, a solid line is drawn from the child class (the class inheriting the behavior) with a closed, unfilled arrowhead (or triangle) pointing to the super class. Figure 23 shows two examples with UML hierarchy. First is Animal hierarchy with one child class Bird, second example shows how both CorporateCustomer and PrivateCustomer classes inherit from the base class Customer.

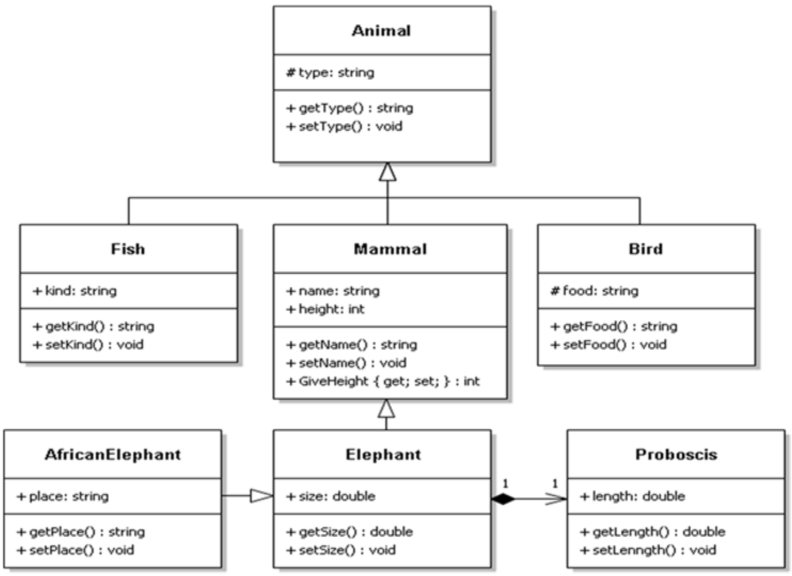


Figure 24. An expanded UML class diagrams representing inheritance - generalizations

The example in Figure 24 illustrates two concepts when modeling an inheritance tree. First, a superclass can have more than one subclass. Second, the inheritance tree can extend for more than one level. The example shows four levels of class Animal.

You are free to use however many levels you need, keeping in mind that three of the most important aspects of generalization are:

- The class diagram be as close to representing the real-world information system as possible.
- The class diagram facilitates correct communication between the user and the systems analyst.

References

Ambler, S. The Elements of UML Style. Cambridge University Press, 2003.

Larman, C. “What the UML Is – and Isn’t”. Java Report, 4(5): 20-24, May, 1999.

Booch, G., I. Jacobson, and J. Rumbaugh. The UML User’s Guide. Addison-Wesley, 1998.

Booch, G., I. Jacobson, and J. Rumbagh. The UML Users Guide, 2nd ed. Addison-Wesley, Boston, MA, 2005.

Lee, R. and W. Tepfenhart. Practical Object-Oriented Development with UML and Java. Prentice Hall, Upper Saddle River, New Jersey, 2003.

Schmuller, J. Sams Teach Yourself UML in 24 Hours, 3rd ed. Sams Publishing, 2006.

Indianapolis, IN. ower, Martin. UML Distilled, 3rd ed. Addison-Wesley Longman, Boston, MA, 2003.

www.uml.com

Lightning Source UK Ltd.
Milton Keynes UK
UKOW06n0756101013

218797UK00001B/2/P

9 783990 342022

Plovdiv University ***"Paisii Hilendarski"***
Faculty of Mathematics and Informatics

CEUS *Center for European University Studies, Wels (Austria)*

WWEDU *World Wide Education, Wels (Austria)*

Plovdiv University ***"Paisii Hilendarski"*** *Faculty of Mathematics and Informatics*

ISBN 978-3-99034-202-2

9 783990 342022

Version: *1.0 / 2013*