

Modern technologies for modeling and development of process information systems

Prof. Dr. Ivica Crnković, M.Sc. Goran Mustapić, M.Sc. Mikael Åkerholm

Mälardalen University, Department of Computer Science and Engineering,
721 23 Västerås, Sweden, <http://www.idt.mdh.se/~icc>

ivica.crnkovic@mdh.se, goran.mustapic@mdh.se, mikael.akerholm@mdh.se

Table of Contents

1	Introduction.....	2
2	Basic Concepts of Object-Oriented Approach.....	5
2.1	Elements of Object-Oriented Approach.....	5
2.2	Object-oriented Languages.....	7
2.3	Conclusion.....	7
3	Unified Modeling Language.....	8
3.1	UML Language Architecture.....	8
3.1.1	BASIC BUILDING BLOCKS OF UML.....	9
3.1.2	RELATIONSHIPS.....	10
3.1.3	DIAGRAMS.....	11
3.2	UML Example.....	12
3.3	Conclusion.....	15
4	Component-Based Development.....	15
4.1	Component Specification.....	16
4.2	Life cycle of Component-based Systems.....	16
4.3	UML and Component-Based Systems Modeling.....	18
4.4	Interacting with Components.....	19
4.5	Component Models.....	20
4.5.1	COMPONENT OBJECT MODEL (COM).....	20
4.5.2	MICROSOFT .NET.....	21
4.5.3	ENTERPRISE JAVA BEANS (EJB).....	23
4.6	Common Object Request Broker Architecture (CORBA).....	23
5	References.....	24

Summary

This report gives an overview of the modern technologies used in software development in general, and in development of process information and process automation systems. As the systems became more complex and the software parts of them larger, old software development technologies based on structural type of programming had more and more difficulties to cope with this complexity. The new technologies emerged in recent ten-fifteen years have tried to solve these problems by focusing on a) reuse of software b) increasing the abstraction level in the development processes. All these technologies are based on object-oriented approach. We can see two clear directions of the development. One direction is in a relation to modeling of software systems. During the establishment of object-oriented languages, different methods for modeling object-oriented systems appeared which eventually resulted in Unified Modeling Language (UML). The second direction is related to the implementation part, i.e. to reuse of software components already implemented. This direction is characterized by component-based approach which includes development of methods and different implementations of component-based models and middlewares (for example COM, .NET, CORBA, JavaBeans, etc.). These trends are continuing. UML is grown up in a general methodology, not only used in object-oriented design, and component-based technologies are gaining ground in new domains.

This report gives an overview of basic concepts of object-oriented approach, UML and component-based technologies.

1 Introduction

In last ten years software is becoming the most important part in increasing number of products previously considered to be pure or to large extent hardware products. Examples of such are cars, washing machines, TV-sets, different control systems, etc. Even more, for such products the cost of developing the software components is a considerable proportion of the total development cost and is increasing. For example, the development cost of software in the industrial robot industry was one third of the total development cost ten years ago, today it is about two thirds, see Figure 1.1.

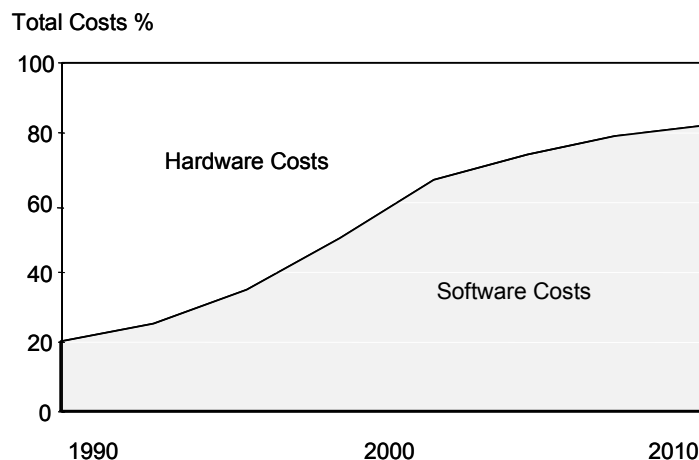


Figure 1.1 Software and hardware development costs for the industrial robots)

We observe similar trends in the automotive industry, in the telecommunication and in number of other domains.

Software is no longer marginal in technical systems but has now become a central factor in many fields. System features based on software functionality, rather than other characteristics, are becoming the most important factor in competing on the market. Increasing numbers of software users are non-experts. These trends place new demands on software. Usability, robustness, simple installation and integration become the most important features of software. As a consequence of the wider area of software utilization, the demand for the integration of different areas has increased.

A consequence of all this is that software is becoming increasingly large and complex. Traditionally, software development addressed challenges of increasing complexity and dependence on external software by focusing on one system at a time and on delivery deadlines and budgets, while ignoring the evolutionary needs of the system. This has led to a number of problems: the failure of the majority of projects to meet their deadline, budget, and quality requirements and the continued increase in the costs associated with software maintenance.

Software has indeed changed and improved and we can witness a continuous change of its paradigm. With each change of the paradigm the productivity has been increased. Figure 1.2 shows different software paradigms that have appeared during last twenty years.

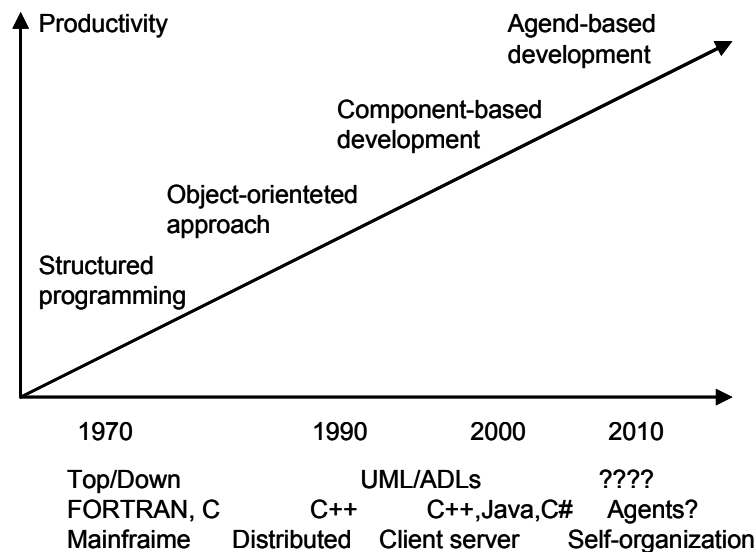


Figure 1.2. Software paradigm shift (2001 © ITEA Office Association)

We can distinguish different periods of software paradigms that can have been characterized by different approaches in design methods, different languages, different computer architectures and different system environments:

- **Structured programming** is characterized by a top/down design, and structured and imperative languages such as Pascal, FORTRAN and C. The computer systems have migrated from large mainframes to minicomputers.
- In late eighties and nineties the **object-oriented approach** has been established. The object-oriented approach comprises software analysis and design, and programming languages. The computer systems are dominated by workstations and PCs. Object-oriented analysis and design have included different modeling languages which eventually merged into the Unified Modeling Language (UML).
- The object-oriented design has been shown as successful in many application domains, but did not solve the increasing requirements for new functions and productivity. Long ago it was realized that the key in solving productivity problem can be solved by re-using software. Object-oriented approach has enabled it to some extent, but it was shown that the re-use level (i.e. re-use of classes and objects) is too low. The object-oriented approach has evolved into a **component-based approach**. Components are reusable parts that can be used in different systems, and not strictly related to objects. Still no new languages have been developed, but object-oriented languages have been used. Instead different component technologies have been developed which focused on re-use of binary components. UML has been further improved, but also different Architectural Definition Languages have been developed. Component-based approach is today a state of the art of software development.
- The recent trends point to a global integration of information. It is not only integration of information within particular domains, but dynamic and intelligent information exchange between different types of computer systems. The pervasive systems, intelligent embedded systems, ad-hoc communication systems, flexible and adjustable software are characteristics of the future systems. The basic technologies supporting these trends are based on **software agents** that can migrate between different systems and adjust their functionalities according to available resources, and different **middleware** solutions that provide support for established a high-level communication.

This report will give a short overview of some of the technologies widely used today. Section 2 presents some basic concepts of object-oriented approach, as the most of the technologies used today are based on object-oriented concept. Section 3 gives an overview of UML. Section 4 describes component-based and middleware technologies used today as standards and de-facto standards in software development. Section 5 will summarize the limits of these

technologies in particular for process information systems. Finally, section 6 concludes the report.

2 Basic Concepts of Object-Oriented Approach

Object-oriented approach includes analysis, design and programming in which the focus is set on 'things' rather than on operations or functions. A software program is not designed as a set of functions that interchange data through their parameters and through a shared memory or global variables; an object-oriented program consists of interacting objects. Objects maintain their own local state and define operations on that state information. They hide information about the representation of the state and hence limit access to it.

The characteristics of an object-oriented design are:

- In an object-oriented design a software system is designed as a set of interacting objects that manage their own private state and offer services to other objects. These services are often called methods, or operations.
- Objects are specified by object classes. An object is created by instantiating an object class.
- System functionality is expressed in terms of operations or services associated with each object. Objects interact by calling on the operations defined by other objects.
- There are no shared data areas. Objects communicate by calling on services offered by other objects rather than sharing variables. There is no possibility that a program component can be affected by modifications to shared information.

2.1 Elements of Object-Oriented Approach

The fundamental part of any object-oriented approach is a **class** and an **object**. While a class is a form (i.e. it identifies which attributes and operations it includes), an object is an instance of that form with concrete values of the attributes and which performs the operations. An example of a class is shown on Figure 2.1 in which we can see the class name (Rectangle), attributes (position, length, with and color) and operations (translate, rotate). The attributes and the operations belong to a particular type. For example, the position is a pair of integer value, length and with may be integer values, etc. The types may be primitive (such as integer, or Boolean) or complex (such as different structures or any other object). The operations may have input parameters (such as new coordinates) and they can return the result which is also of a pre-defined type. In that sense the operations do not differ from the functions in non-object-oriented languages.

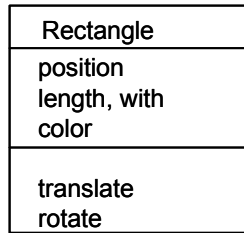


Figure 2.1 An example of a class

From the example shown above we can see directly that we can identify objects with many common characteristics, but some of them different. To efficiently manage these types of objects, object-oriented approach uses terms generalization (or inheritance). In our example we can identify a class Shape, that contains attributes such as position and color, and then classes Rectangle, Circle, Triangle, and so on, that all are derived from the Shape class, but have some specific operations. Figure 4 illustrates the “generalization” term, in which the arrow denotes “generalization”.

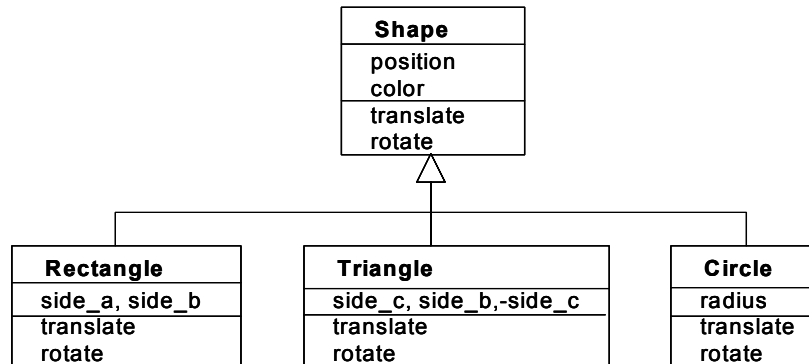


Figure 2.2 Class generalization

From Figure 2.2 we see that the class Shape includes two operations, translate and rotate. It is obvious that the rules for the translation are different for every particular shape. This is the reason why each sub-class has its own implementation of these operations. These operations in the Shape class (i.e. in the super-class) are only declared, but no implementation is specified. Their role here is only to show that every sub-class of the class Shape must implement these operations. This principle is very important as it completely specifies what the sub-classes must include. A special case of a super-class is a class which only includes declarations (specifications) and not implementations. Such classes are designated as virtual classes or interfaces. Although the separation of specification from implementation is an old principle already present in other languages, such as Pascal, Ada and even C, the full utilization of this principle has started with object-oriented technology and has been further developed in new component-based technologies. In a similar way as the methods from a sub-class can “overload” a method from a super-class, many object-oriented languages have mechanisms to overload basic operations (such as addition, subtraction, assignments, etc.), which provides more abstractions to them. For example, the operation “+” may denote addition of two integer numbers, or two matrices, depending on which objects are summed.

2.2 Object-oriented Languages

Object-oriented approach comprises several phases of the software development: analysis, design and implementation. The implementation part has a strong support from many programming languages, some of them clearly predominated used. We list here some of them, the most important in the history of object-oriented approach and now.

- **Smalltalk** is the first real object-oriented language and the entire development environment. Its basic characteristic is that literally every element in the language is defined as an object. This approach has made this language very efficient for programming, but unfortunately with very low performance. Due to various reasons, most of them of the marketing natures, Smalltalk has never reach the wide popularity.
- **C++** is the first object-oriented language that has reach enormous popularity and is today one of the most used programming languages. The reason of its popularity is its upward compatibility with C, its ability to be used for small and large systems, efficient compilers, and similar. C++ has also many drawbacks, such as complexity and a lot of inconsistencies, which makes it difficult to learn and read.
- **Java** is improved and simplified C++. All parts in C++ that are inherited from C are removed and some clearer concepts have been introduced. This made this language extremely popular, in particular in academia. Java uses a concept of a virtual machine which interprets the java code – so called byte code which is machine independent. By introducing virtual machine concept, Java code has become completely portable. For this reason Java is very suitable for applications executed on different platforms (computer systems), and became very popular in web- and internet-based applications. On the other hand the consequence of this concept is a poorer performance and a lack of support of direct use of services of particular operating systems.
- **C#** is a relatively new language developed by Microsoft, very similar to Java, both as a language and as the concept. The superb development environment (Visual Studio) and run time environment (.NET) makes this language, and the entire technology, increasingly popular.
- There exist many other object-oriented languages, or languages with object-oriented flavor. Some of them are: Eiffel, Objective C, MS Visual Basic, interpreter Python, etc. They support the object-oriented approach to different extent, including some of the principles. Some of them are also “full” object-oriented languages, but have never reached the popularity as the languages listed above.

2.3 Conclusion

During nineties the object-oriented approach has become dominated in software development. The main advantage of an object-oriented approach is the ability of mapping the real world with the programming models. An object (“a thing”) in

the real world defined an object in the object-oriented approach. The second main advantage is the data and operation encapsulation. An object is independent of the rest of the system, which makes such systems easier to maintain. Many object-oriented languages provide support for efficient implementation of object-oriented designs. However, the object-oriented approach has not solved all the problems, it was not sufficient. The levels of the abstractions can still be on the low level, and the reusability was not achieved to that extent as there were hopes.

3 Unified Modeling Language

UML is a result of the evolution of object-oriented modeling languages. It was developed by Rational Software Company, by unifying some of the leading object-oriented modeling methods: Booch (author: Grady Booch), OMT (Object Modeling Technique; author: Jim Raumbaugh) and OOSE (Object-Oriented Software Engineering; author: Ivar Jacobson). The unification work started in '94. UML 1.0 was submitted to OMG in '97 (Object Management Group) by a group called UML Partners which was founded by Rational Software. The current UML version is 1.4 (published in Sep 2001) and there is an ongoing work in OMG on a new major version 2.0.

UML is used for modeling software systems. The modeling includes a process of analysis and design. By an analysis the system is first described by a set of requirements, and then by identification of system parts. The design phase is tightly connected to the analysis phase; it starts from the identified system parts and continues with detailed specification of these parts and their interaction. For requirements identification UML provides a support for identifying and specifying *use cases*. System parts are identified as packages, components and finally as objects (which are represented by classes).

3.1 UML Language Architecture

To be able to read and create UML models, one needs to understand the conceptual model of UML language. The conceptual model of UML contains the following elements:





- UML basic building blocks
 - Things
 - Relationships
 - Diagrams
- Rules that dictate how building blocks can be used together
There are semantic rules for what well-formed UML models are. Those include: naming, scope, visibility, integrity, execution. However, during development, UML models are typically not well-formed, but tend to be incomplete and inconsistent.
- Common mechanisms that apply consistently throughout UML:
 - Specifications
 - UML's graphical notation is used to visualize the model, but UML's specification is used to state the model's details.




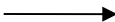

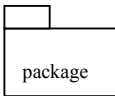
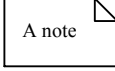
- Adornments. Many of the specification details can be rendered as graphical or textual adornments (meaning of the word adornments is similar to enhancements or decorations) to the basic notation. Every element in the UML's notation starts with a basic symbol. Variety of adornments can be added to this basic symbol.
- Common divisions. Almost every building block in UML has class/object concept. Graphically, UML uses the same symbol for class and object, but object's name is underlined. There is a separation between interface and implementation. Almost every building block in UML has this interface/implementation concept.
- Extensibility mechanisms (stereotypes, tagged values and constraints)

3.1.1 Basic building blocks of UML

Basic building blocks of UML are: Things, Relationships and Diagrams. Things and relationships are building blocks for diagrams.

Types of UML things are presented in the **Error! Reference source not found.** below.

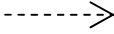
Types of UML Things	Subtypes of the Things			
	Name	Symbol	Description	Variations/other related elements
Structural <ul style="list-style-type: none"> • Nouns of UML models • Conceptual or physical 	Class		Description of a set of objects that share the same: attributes, operations, relationships and semantics.	- actors - signals - utilities
	Interface		A collection of operations that specify a service of a class or component.	
	Collaboration		An interaction and a society or roles and other elements that work together to provide some cooperative behavior that is bigger than the sum of all the elements. Represent implementation of patterns that make up the system.	
	Use Case		A description of set of sequence of actions that a system perform that produces an observable result of value to a particular actor. Used to structure behavioral things in the model.	

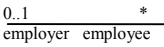
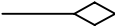

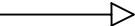
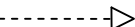
	Active class		A class whose objects own a process or execution thread and therefore can initiate a control activity on their own.	- processes - threads
	Component		A component is a physical and replacable part that conforms to and provides the realisation of a set of interfaces.	
	Node		A physical resource that exists in run time and represents a computational resource.	
Behavioral Dynamic parts of UML models	Interaction	message 	Set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose.	- messages - action sequences - links
	State machine		A behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.	- states - transitions - events - activities
Grouping Organizational parts of UML	Packages		General purpose mechanism of organizing elements into groups.	- frameworks - models - subsystems
Annotational Explanatory parts of UML	Note		A symbol for rendering notes and constraints attached to an element or a collection of elements.	

3.1.2 Relationships

Relationships are used to connect *things* into well-defined models.

The types of UML relationships are shown in the table below.

Types of UML Relationships	Subtypes of the Relationships		
	Symbol	Description	Specialization
Dependency		A semantic relationship between two things in which a change to one thing may affect the semantics of the dependent thing.	

Association		<p>Structural relationship that describes a set of links, where a link is a connection between objects.</p> <p>Aggregation and composition are “has-a” relationship. Aggregation (white diamond) is an association indicating that one object is temporarily subordinate or the other, while the composition (black diamond) indicate that an object is a subordinate of another through its lifetime.</p>	<p>Aggregation</p>  <p>Composition</p> 
Generalization		<p>Specialization/generalization relationship in which objects of the specialized element are substitutable for objects of the generalized element.</p>	
Realization		<p>Semantic relationship between two classifiers, where one or them specifies a contract and the other guaranties to carry out the contract. They are used between:</p> <ul style="list-style-type: none"> - interfaces and classes or components - use cases and collaborations that realize them 	

3.1.3 Diagrams

A diagram is a graphical representation of a set of elements. Following types of diagrams are defined by UML:

Types of UML <i>Diagrams</i>	Description
<i>Use case diagram</i>	Shows a set of use cases and actors and relationships between them.
<i>Class diagram</i>	Structural relationship that describes a set of links, where a link is a connection between objects.
<i>Object diagram</i>	a set of objects and their relationships. This is a snapshot of instances of the things found in the class diagrams.
<i>Statechart diagram</i>	Shows a state machine consisting of: states, transitions, events and activities.
<i>Activity diagram</i>	This is a special kind of statechart diagram, which shows the flow from activity to activity within a system.
Interaction	<p><i>Sequence diagram</i></p> <p>Interaction diagrams show set of objects and their relationships, including messages that may be dispatched between them. Sequence diagram emphasize the time ordering between messages.</p> <p><i>Collaboration diagram</i></p> <p>Collaboration diagram emphasize structural ordering of objects that send and receive messages.</p>
<i>Component diagram</i>	Shows organizations and dependencies among a set of components. These diagrams address static implementation view of the system.
<i>Deployment diagram</i>	Show the configuration of run-time processing nodes and components that live on them.

These are the diagrams most commonly found in the practice, but this is not a closed list. Tools may provide other kinds of diagrams. Also, as mentioned in [4], it is common to use combinations of these diagrams. A common example is to combine Component and Deployment diagrams.

3.2 UML Example

We shall illustrate use of UML by a simple example.

Imagine the software system at a library. The main task should be keeping track of all books and the status of each book (out of lone, in stock etc). In this example we are modelling and explaining the system with a sequence of UML diagrams.

The first step is the system analysis, and the input to the analysis is the specification of the requirements. In an object-oriented and UML approach the requirements are identified with help of identifying of cases of use of the system. This is done by UML use *case diagram*. The main goal of this part is to identify the most characteristics use cases, and the actors (i.e. people or other types of “users” of the system).

In figure 3.1, a UML use case diagram shows examples of how the system is intended to be used. To the left in the figure we have identified an actor “Librarian”, which is a librarian. The librarian can use the system in four ways. Firstly it is possible to add a new book to the system visualized by the “AddNewBook” use-case. Secondly, if a book is out of loan, it is possible for the librarian to make a reservation for a customer; this is showed by the “ReserveBook” use-case. It should also be possible to loan books, i.e. the “LoanBook” use-case. Finally the “ReturnBook” use case shows that a librarian should also be able to return a book to the system when a customer hands a previously borrowed book in.

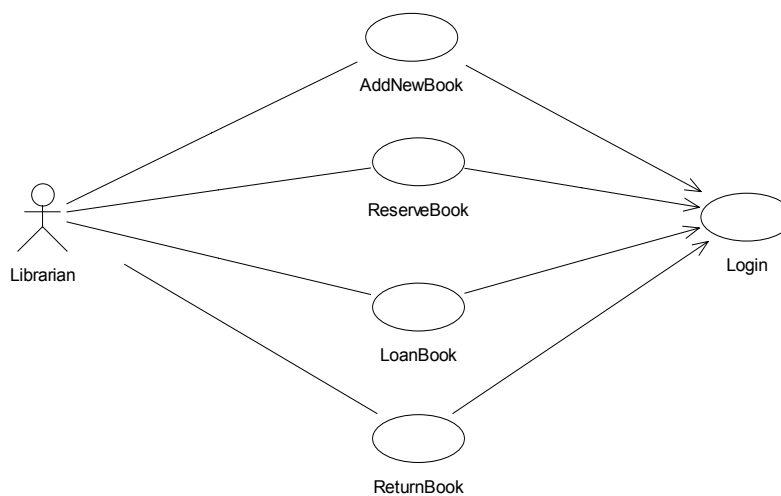


Figure 3.1, a UML use-case diagram, visualizing how a librarian can use a library booking system.

The next step in the analysis and design process is to identify the objects the system deal with. Form the problem description and use case diagrams it is easy to identify the objects involved in the system: The library, the books, and the librarian. In addition to this we also have a librarian system itself which we can specify as a set of services. We specify the objects by specifying the classes with their attributes and services (methods) they provide. In UML this is done by a *class diagram*. This diagram also includes specifications of relations between the objects.

In Figure 3.2, a UML class diagram shows the four classes “Book”, “Library”, “LoginService” and “Librarian” and how they are related to each others. Internal attributes and methods are also shown in the figure.

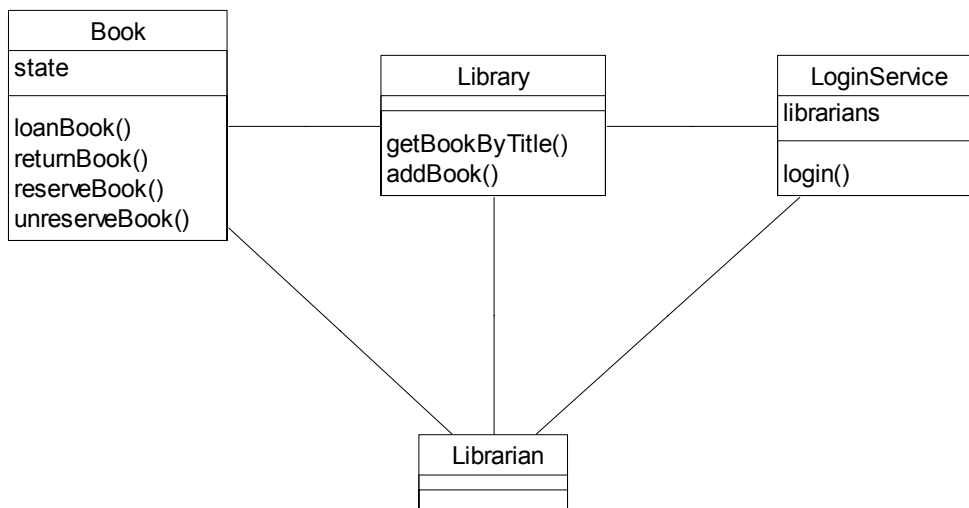


Figure 3.2, UML class diagram

The UML class diagram shows the static characteristics (i.e. the structure) of the system. The dynamic behavior of the system can be described by **chart diagrams** and **interaction diagrams**.

Chart diagrams are used to express the states of the systems and its transition from a state to a state triggered by a particular event. Chart diagrams are variations of finite-state machines, a standard method used in software design and programming.

Figure 3.3 shows UML state chart diagram showing the internal state of the class “Book”. The diagram shows that a book can be in one of the four states “In stock”, “Out of loan”, “Reserved and out of loan” and “Reserved and in stock”. The starting point for a new book is marked with the black dot to the left in the figure. When a new book is entered in the system it enters the “In stock” state, when the method “loanBook()” is executed the book enters the “Out of loan” state etc.

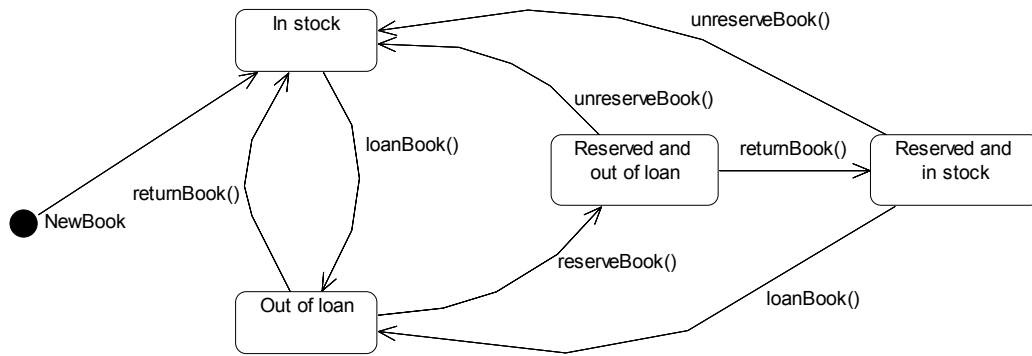


Figure 3.3, UML state-chart diagram for the class book

UML sequence diagram shows interaction between different objects in a time sequence. The vertical line denotes time, the rectangles appearance of the objects, and the arrows invocation of services of particular objects, or interaction between the objects.

In Figure 3.4, the “LoanBook” use-case is further developed with a UML interaction diagram, showing the sequence of interactions required to solve the use-case. Firstly the Librarian has to use the “login()” method provided by the “LoginService” class. Then a reference to the book is required, and the librarian has to utilize the “getBookByTitle()” method provided by the “Library” class. Finally the loaning process can be accomplished by the “loanBook” method provided by the class representing a book.

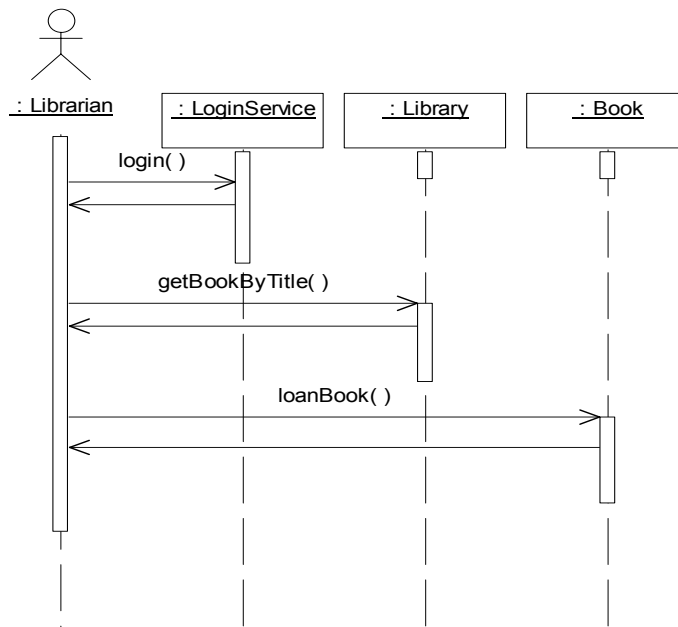


Figure 3.4, a UML sequence diagram, solving the loan book use-case

3.3 Conclusion

UML has intentionally been developed as a language for modeling object-oriented systems. Its use has however widely been spread out. Today UML is used for system specifications. In different domains UML (for example in distribution of electrical power) is used for specification and standardization of different systems or parts of the systems. This standardization makes it possible that different vendors produce products that comply with the standard specification.

From UML it is possible to automatically generate different types of descriptions (for example specifications in XML), or even automatic creation of software code. UML is becoming a standard tool for software and system engineers.

4 Component-Based Development

The concept of building software from components is not new. A “classical” design of complex software systems always begins with the identification of system parts designated subsystems or blocks, and on a lower level modules, classes, procedures and so on. The reuse approach to software development has been used for many years. However, the recent emergence of new technologies has significantly increased the possibilities of building systems and applications from reusable components. Both customers and suppliers have had great expectations from component-based development (CBD), but their expectations have not always been satisfied. Experience has shown that component-based development requires a systematic approach to and focus on the component aspects of software development. Traditional software engineering disciplines must be adjusted to the new approach, and new procedures must be developed. Component-based Software Engineering (CBSE) has become recognized as such a new sub-discipline of Software Engineering.

The major goals of CBSE are the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customizing and replacing their components. The building of systems from components and the building of components for different systems requires established methodologies and processes not only in relation to the development/maintenance aspects, but also to the entire component and system lifecycle including organizational, marketing, legal, and other aspects. In addition to specific CBSE objectives such as component specification or composition and technologies, there are a number of software engineering disciplines and processes which require specific methodologies for application in component-based development. Many of these methodologies are not yet established in practice, some are not even developed. The progress of software development in the near future will depend very much on the successful establishment of CBSE and this is recognized by both industry and academia.

4.1 Component Specification

For a common understanding of component-based development, the starting point is an agreement of what a component is and what it is not. As a generic term the concept is pretty clear – a component is a part of something – but this is too vague to be useful. The definition of a component has been widely discussed. However, we shall adopt Szyperski's definition, which is the most frequently used today:

A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parts.

The most important feature of a component is the separation of its interface from its implementation. We require that the integration of a component into an application should be independent of the component development lifecycle and that there should be no need to recompile or re-link the application when updating with a new component. Another important characteristic of the separation is that the component implementation is only visible through its interface. This is especially significant for components delivered by a third party. An implication of this is the requirement for a complete specification of a component including its functional interface, non-functional characteristics (performance, resources required, etc.), use cases, tests, etc. While current component-based technologies successfully manage functional interfaces, there is no satisfactory support for managing other parts of a component specification.

The component definition adopted above is focused on the use of components. It says little about how to design, implement and specify a component. There are however, other definitions which point to other aspects of component-based development. For example there is a strong relation between object-oriented programming (OOP) and components. Component models (also called component standards) COM/DCOM, .NET, Enterprise Java Beans (EJB)**Error! Reference source not found.**, and CORBA Component Model (CCM) **Error! Reference source not found.** relate Component Interface to Class Interface. Components adopt object principles of unification of functions and data encapsulation.

4.2 Life cycle of Component-based Systems

Development with components differs from traditional development through its focus on the identification of reusable entities and relations between them, starting from the system requirements. Different lifecycle models, established in software engineering, can be used in component-based development. These models will be modified to emphasize component-centric activities. Let us consider, for example, the waterfall model using an extreme component-based approach. The top half of Figure 4.1 shows the phases of the waterfall model. Underneath are shown the accompanying activities in component-based development.

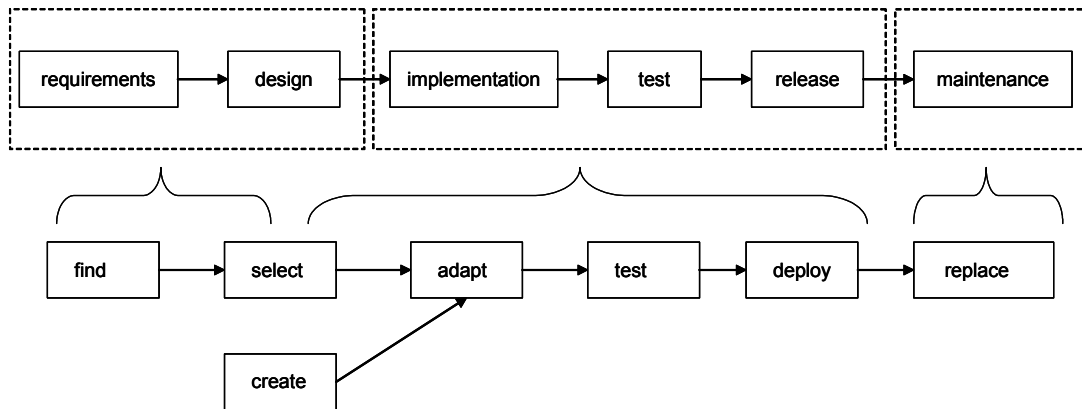


Figure 4.1. The development cycle compared with the waterfall model.

The characteristic features of component-based development are as follows.

The initial identification of requirements is performed as in traditional development. However in the component-based approach, the mapping between system and component requirements is important. As one of the goals of component-based design approach is to reuse existing components, during the system requirements elicitation the requirements for components should be identified.

The early design phase focuses on two essential steps:

- The logical view of the system is specified by system architecture in terms of components and their interaction. In this view, components are represented by specification of their interfaces, possibly including specification of relevant extra-functional properties (in real-time systems this includes timing properties)
- The structural view is specified by a system architecture consisting component implementations. Component implementations must conform to a particular component model which assumes a particular system architecture, component framework, and different technology-specific services. The component model and framework may have a significant impact on the design solution, thus in a CBD approach (where reuse of existing components and services is assumed) they must be considered in the early design phase.
- The implementation phase includes adapting, composing, and deploying components, using a framework for components.
- The verification (or test) phase performs system verification (e.g., by testing).
- The maintenance phase puts extra focus on the replacement and update of entire components, possibly during system operation.

4.3 UML and Component-Based Systems Modeling

UML can be used for both component and system modeling. Component-driven design concentrates on interface definitions and collaboration between the components through the interfaces. The design process continues with the modeling of the system with physical components, which do not necessarily match the logical structure. These may be pre-existing components, with interface already specified and possibly in need of wrappers. One logical component, identified in the first phase of design, may consist of several physical components. Finally, there is a deployment aspect, the components being executed on different computers in a distributed application. In a non-component-based approach the first, the design phase is important, while mapping between the conceptual and implementation level is a direct mapping, and the deployment phase is the same for the whole application. In principle, UML can be utilized to provide support for designing component-based systems covering all these aspects. Interfaces are presented as multiple subsystems (also multiple interfaces may be realized by a subsystem), which indicate the possibility of changing the implementation without replacing the interface. An interface can be presented in two ways (see Figure 4.2), the second alternative being the more common presentation.

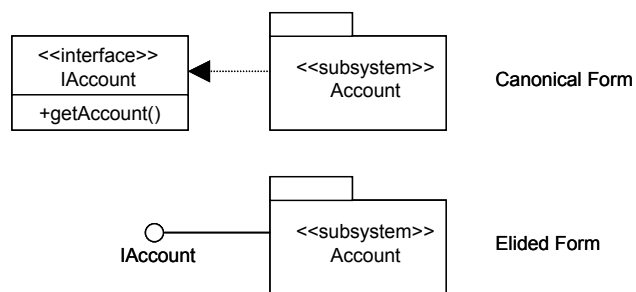


Figure 4.2 UML component

Figure 4.3 shows the three aspects of system architecture. The conceptual architecture is a result of a top-down system analysis and design and in at least the first step is not different from a “non-component-based” design. In the conceptual part the components are expressed by UML packages with the <<subsystems>> stereotype. In the implementation architecture part, the physical components are represented by UML components and the <<imp>> stereotype. Note that the implementation part is not necessary only refinement of the conceptual level, but also the structure can be changed. For example, different packages can include the same physical components. It may also happen that the component selection requires modifications of the conceptual architecture.

UML is however not specialized for CBD and certain extensions to standard UML (such as naming convention, or stereotypes) are required. The component interfaces cannot be described by UML at such a detailed level that they can be used directly.

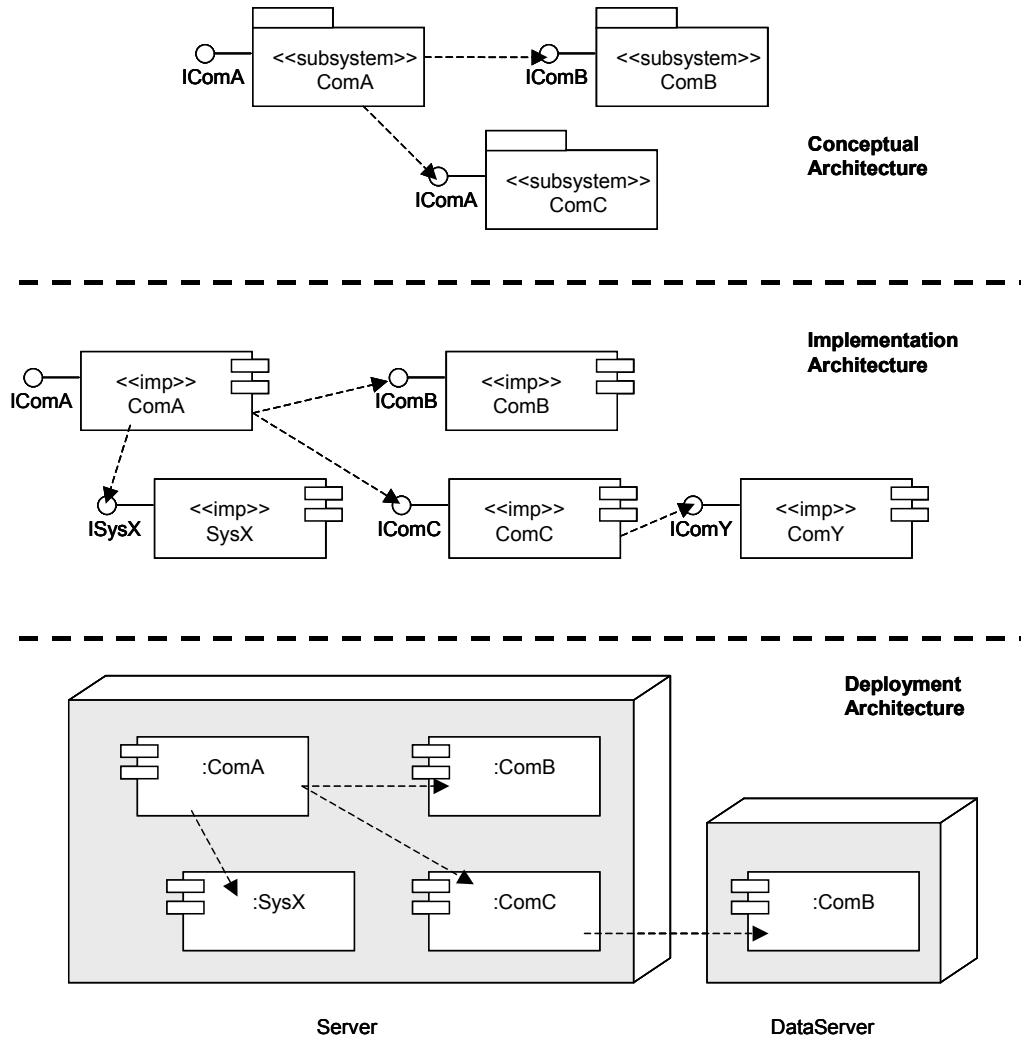


Figure 4.3 Examples of different aspects of component-based architecture

4.4 Interacting with Components

Components express themselves through interfaces. An interface is the connection to the user that will interact with a component. If an interface is changed the user needs to know that it has changed and how to use the new version of it.

Functions that are exposed to the user are usually called Application Programmable Interface (API). If there is a change to the API, the user has to recompile his code as well.

In an object-oriented world, an interface is a set of the public methods defined for an object.

Usually the object can be manipulated only through its interface. In C++ the user has to recompile the code only when an interface, referred from the code, is changed. There is also a drawback that the user of the class must use the same programming language throughout the whole development.

Separating the interface from the implementation is a way to avoid this tight coupling. This kind of separation is made with binary interfaces as done in CORBA and COM, the component models described in the next section. Binary interfaces are defined in an interface definition language (IDL) and an IDL compiler, which generates stubs and proxies, makes the applications location transparent.

An example of using the same interface but different implementations is shown in 0.4: By a separation between the interface and the implementation it is possible to run new clients together with old server components or vice versa. The word processor is called the client and the dictionary is called the server since it provides functionality to the word processor. It is possible to upgrade to new versions of the word processor and dictionary component independent of each other.

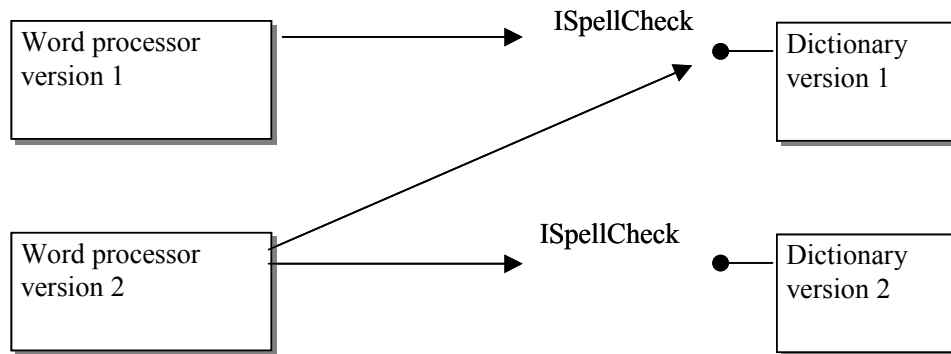


Figure 4.4 The possible combinations between old and new clients and their components.

Even if an interface has not been changed, its implementation can be changed. This increases flexibility of possible updates, but also introduces a possibility of having uncontrolled effects. For this reason, it is of interest to know if the implementation has been changed.

4.5 Component Models

The component models define the standards forms and standard interfaces between the components. They make it possible to components to being deployed and to communicate. The communication can be established between components on the same node (computer) or between different nodes. For the later we are talkies about component distribution.

4.5.1 Component Object Model (COM)

The Component Object Model provides a model for designing components that have multiple interfaces with dynamic binding to other components. COM is an open standard, which has been implemented on many different platforms, but the main platform is of course Microsoft Windows for which it was first developed. Components expose themselves through interfaces and only interfaces. The interfaces are binary which makes it possible to implement the component in a variety of programming languages such as C++, Visual Basic and Java. A COM component can implement and expose multiple interfaces. A client uses COM to locate the server components and then it queries for the wanted interfaces.

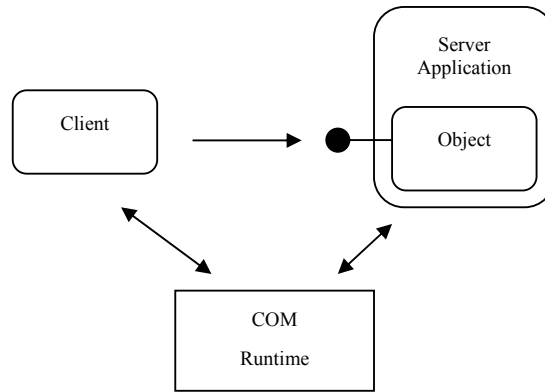


Figure 4.5 COM establishes the connection between client and server.

By defining interfaces as unchangeable units, COM solves the interface versioning problem. Each time a new version of the interface is created a new interface will be added instead of changing the older version. A basic COM rule is that you cannot change an interface when it has been released. This makes couplings between COM components very loose and it is easy to upgrade parts of the system indifferent from each other.

DCOM is the protocol that is used to make COM location transparent. A client talks to a proxy, which looks like the server and manages the real communication with the server.

COM+ is an extension to COM with technologies that support among others: transactions, directory service, load balancing and message queuing.

4.5.2 Microsoft .NET

.NET is a language-independent environment for developing software programs that will co-operate in a distributed environment. It is also a run-time platform with a number of services for interpolation and dynamic integration of software components. .NET a compiler translates the source code into an intermediate language called the Microsoft Intermediate Language (MSIL) very similar to the Java Byte Code. The common language runtime (CLR), very similar to a Java Virtual Machine then takes the intermediate language and, on the fly, converts it into machine-specific instructions.

.NET includes a number of services available during development and during run-time (see figure 4.6).

Interfaces and Assembly

.NET represents the programming language approach for component programming. It means that the program contains the information related to the relationships with other “components”, and that the compiler is responsible for generating the information needed at execution. This (proprietary) approach contrasts with the OMG (open) approach where separate formalisms (and files), are used to indicate component related information; languages and compilers being unchanged.

What most resembles a component is an assembly; the manifest is the component descriptor, it gathers in a single place all the information about an assembly: exported and imported methods and events, code, metadata and resources. Because of the programming language approach, the corresponding programming language, C#, which looks very much like Java, includes some features of a component model: (first class) events and extensible meta data information. The compiler not only produces MSIL

byte code but also generates, in the manifest, the interface description of the component (called assembly), in the form of a list of import and export types.

Framework

.NET relies on the traditional programming approach: the framework is seen as the language run-time support. Extra-functional aspects like distribution, security, confidentiality, and version control are delegated at execution to the OS and loader.

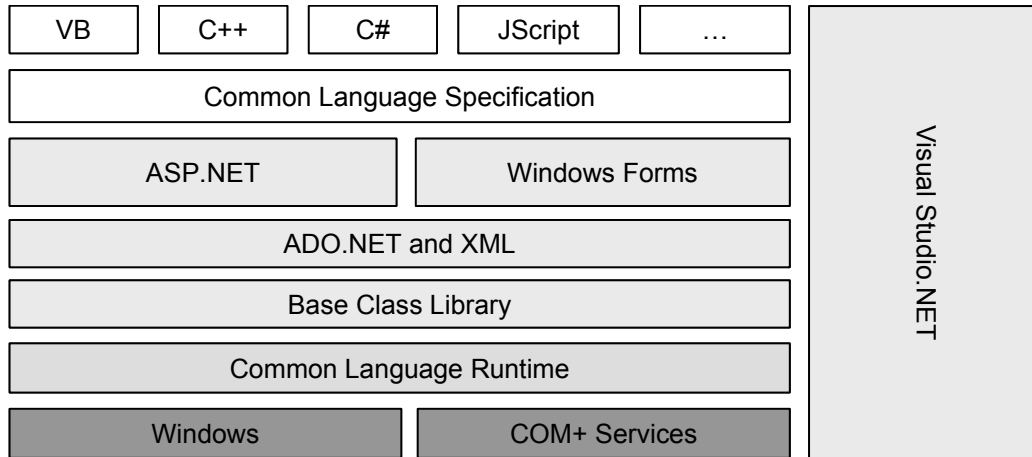


Figure 4.6 Components of .NET Framework

Lifecycle

Unlike when using traditional DLLs, the .NET model includes visibility control, which allows assemblies (and their modules) to be local to an application, and thus different DLLs with same name can run simultaneously. Further, each assembly has versioning information about itself and about the assemblies it depends on, provided either in the form of attributes in the code source or as command line switches when building the manifest. Version control is delegated to the dynamic loader, which selects the “right” version, local or distant, based on the assembly’s version information and on a set of default rules.

Extra-functional properties

.NET does not provide any support for analyzing extra-functional properties. The language enables use of meta data at run-time, which gives some possibilities for checking the properties at run-time. For example, contract-based interfaces with pre- and post-conditions may be implemented by using this feature. .NET does not provide any support for real-time applications. Further, the memory size and performance excludes it from do embedded systems domain so far.

Availability

.NET is used on Microsoft Windows 2000 and XP platforms. Some parts of it are ported to Windows CE, the Microsoft real-time systems. Mono initiative (<http://go-mono.com>) develop an open source implementation of the .NET Development Framework. Mono includes a compiler for the C# language, a runtime for the Common Language Infrastructure and a set of class libraries. The runtime can be embedded into the application.

4.5.3 Enterprise Java Beans (EJB)

Enterprise Java Beans is a component architecture for server-side components used to build distributed systems with multiple clients and servers. A Java Bean is a reusable component that support persistency and can inter-operate across all platforms supported by Java. EJB uses Java Beans but it is a lot more than a component model. EJB provides support for transactions and security over a neutral object communication protocol, which gives the user the benefit to implement the application on top of a protocol of choice. EJB is part of the Java 2 Platform Enterprise Edition (J2EE) which includes many other technologies remote method invocation (RMI), naming and directory interface (JNDI), database connectivity (JDBC), Server Pages (JSPs) and Messaging services (JMS).

To make a JavaBean an Enterprise bean the JavaBean has to conform to the specification of EJB by implementing and expose a few required methods. These methods allow the EJB container to manage beans in a uniform way for creation, transactions etc. A client to an enterprise bean can virtually be anything, for example a servlet, applet or another enterprise bean. Since enterprise beans may call each other then a complex bean task might be divided into smaller tasks and handled by a hierarchy of beans. This is a powerful way of “divide and conquer”.

There are two different kinds of enterprise beans: session and entity beans. Session beans live as long as the client code that calls it. Session beans represent the business process and are used to implement business logic, business rules and workflow.

EJB is designed so it can run together with CORBA and access CORBA objects easily.

4.6 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) is a standard that has been developed by the Object Management Group (OMG) in the beginning of the nineties. The OMG provides industry guidelines and object management specifications to supply a common framework for integrating application development. Primary requirements for these specifications are reusability, portability and interoperability of object based software components in a distributed environment. CORBA is part of the Object Management Architecture (OMA) which covers object services, common facilities and definitions of terms.

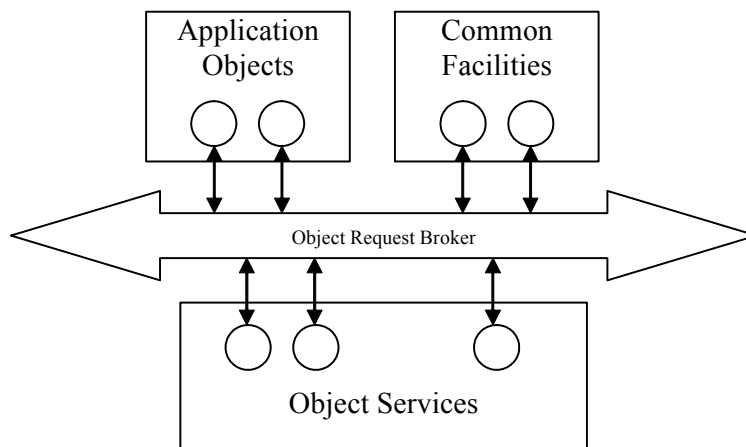


Figure 4.7 The parts of the Object Management Architecture.

Object services are for instance naming, persistency, events, transactions and relationships. These can be used when implementing applications. Common facilities provide general-purpose services like information, task and system management. All services and facilities are specified in IDL. An object request broker (ORB) provides the basic mechanism for transparently making requests and receiving responses from objects located locally or remotely. Requests can be made through the ORB without regard to the service location or implementation. Objects publish their interfaces using the Interface Definition Language (IDL) as defined in the CORBA specification.

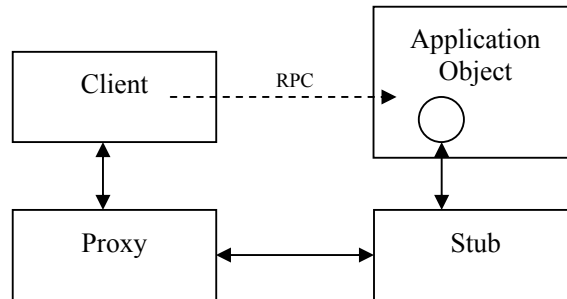


Figure 4.8 Clients communicate with RPC transparently with the server.

Objects are stored in an interface repository where they can be found and activated on demand from the clients. The stubs and proxies are generated from the IDL specification that each object provides for its interfaces.

5 References

- Somerville I., Software Engineering, Addison-Wesely
- Perdita Stevens with Rob Pooley, Using UML: Software Engineering with Objects and Components, Addison-Wesley
- Ivica Crnkovic and Magnus Larsson, Building Reliable Component-Based Software Systems