UNIVERSITY OF NOVI SAD

FACULTY OF SCIENCES

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Objects versus Components

Term paper from course Component-based Development

Professor:
Dr. Miloš Racković

Student:
Robert Pap

Date:
January 24, 2011 Novi Sad

Table of Contents

Abstract

This term paper is about objects and components. Today, both philosophies, object-oriented development and component-based development, are very popular. The idea of component-based development is to construct programs from prebuilt software components. Because reuse is so important nowadays, it was expected that components will spread very fast, and even rule out objects. However, the delicate nature of software resulted instead in a very slow acceptance.

As *Szyperski* said, *"components are for composition"*. Composition enables prefabricated items to be reused by rearranging them in new composites. A component is a unit of independently deployable third-party composition, and has no observable state. All components have interfaces: an interface about the services a particular component provides, and an interface about the services it needs to function. Components are important, because they provide a higher level of abstraction, with goals such as conquering complexity, managing change and reuse.

The arrival of software components resulted in the spread of component software and component markets. This is in direct relation with the outsourcing of software. For component software, the key factor to success lies somewhere in between zero (called custom-made software) and maximum outsourcing (called standard software). At the other hand, for component markets to develop, component standards must also exist.

The goal of this term paper is to present a clear understanding about components themselves, but also to try to bridge the gap between objects and components by placing both concepts face to face. This direct comparison will be performed from various viewpoints. First, in a market viewpoint, a more business-specific approach will be presented, explaining what are the units of deployment for objects, and what for components. An important question about why did object technology not succeed in establishing significant object markets, will be also answered. Next, objects and components will be compared from a client's point of view. In this section, some component success stories will be revealed, together with an interesting example about component deployment and usage from a simple computer user's point of view. In the next section, objects and components will be analyzed and compared in the little details, most notably on the implementation level, pointing out that components don't need to be object-oriented in the inside at all. On the abstract level, the size of objects and components will be compared, leading to an important statement about their roles. On the technical level, a very direct and tabular comparison will be given. Finally, an attempt will be made to bridge the gap between these two concepts by proposing a guideline to a software engineering process which tries to harness the best from both worlds. The suggested guideline will be also illustrated on a simple but practical example.

This paper is organized as follows: after a brief introduction, two chapters will be presented. The first will introduce the reader to the world of components, defining components, but objects, too. The direct comparison between objects and components from various viewpoints will be given in the second chapter. The paper concludes with a brief conclusion.

All diagrams in this paper were created using *BOUML v4.23 patch 2*, the freeware *UML 2* modeling tool, and *OpenOffice.org Draw v3.2.1*, the free and open drawing tool. The homepages of these projects can be found at <http://bouml.free.fr/> and <http://www.openoffice.org/>, respectively.

Introduction

Programming is an activity of constructing computer programs that are a sequence of instructions which describe, how to perform certain tasks with computers [Wang & Qian 2005]. This activity can be described in terms of the major techniques or concepts used for constructing these programs. Different computers require different kinds of programming. For example, *Pascal*'s machine build in 1642 by *Blaise Pascal* could only be programmed by operating gears and cranks. Of course, on modern electronic computers, a programming language must be used for programming (or developing). If a specific technique, for example *X*, is being used in the development, then it is called *X-oriented programming* or *X-based development*. *Gear-oriented programming* used gears, cogs and wheels. By arranging these items, a specific computation could be performed. Besides *Pascal*'s machine, *Charles Babbage*'s difference engine (1822) used this kind of programming. *Switch-oriented programming* used switches. By presetting these switches and rewiring the system, a new calculation could be performed. *ENIAC* (abbr. **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer), the first electronic computer, used this type of programming.

When the high-level programming languages have arrived, the programmers have been liberated from manipulating hardware details. At first, *procedure-oriented programming* was the most popular technique, which required for the programmers to think in procedures or functions. This was highlighted by structured programming, a top-down design and stepwise refinement design approach. Later, a new development technique called *object-oriented programming* was introduced, encouraging programmers to think in data types. This paradigm used objects and classes as building blocks. It was only a matter of time, when a new philosophy will approach. This new paradigm was called *component-oriented programming* or *component-based development*. The idea behind this technique was the construction of programs from prebuilt software components. With reuse as the main motivator, it was evangelized as the solution to the so-called software crisis. Components were well established in all other engineering disciples, so it was natural to think that the same will happen in the world of software. But it didn't lift off – until the 1990s. The reasons behind this failure can be linked to the particular nature of software. Even today, software components are said to be delicate in nature.

Software components will be described in greater detail in the next chapter.

The World of Components

This chapter will describe the key concepts of *CBD* (abbr. **C**omponent-**B**ased **D**evelopment) or *COP* (abbr. **C**omponent-**O**riented **P**rogramming). First, a brief summary will be given about components in general: how were they introduced in the industry, how they thrive in other engineering fields, and the reasons behind the hardships of components in the world of software. The chapter will advance with more technical concepts, as important definitions will be presented to such terms as components, objects, interfaces and explicit context dependencies. In the next section, a brief journey will be presented with emphasis on component software: what is custom-made software, what is standard software, and where lies component software compared to these two notions. The chapter will close with some remarks about the importance of components, about component standards and about component forms. This chapter is mostly based on chapters one, three and four of [Szyperski, Gruntz & Murer 2002], and chapter one of [Wang & Qian 2005], with help of [Petre 2000] and [Brown & Short 1997].

Components in Other Engineering Fields

As it was already described in the previous section, *CBD* deals with the construction of programs from prebuilt software components. The focus is therefore on the development of software by assembling components. It could be said, that software components have a lot in common with components used in other engineering fields. These other disciplines have started to use components as a natural way of system building. For example, the automotive industry develops extremely complex vehicles using components of every size from a tiny screw to complex subsystems such as engines. Therefore, a modern automotive factory has become more of a system integrator than a manufacturer. This practice has led to the development of more and more complex products without affecting overall speed of delivery. Of course, it's easy to name other engineering industries where components are used as units of assembly (e.g. the hardware industry, which is the closest to the world of software).

It is said, that it was the *Industry Revolution* that dramatically changed the nature of production in which machines replaced men, and the massive and complex products made by these machines replaced handmade items. In the software industry, however, the products are still mainly handmade items. This means, that the productivity is low and the projects are often overrun. People have started to talk about the so-called *software crisis*. Thanks to great advancements in the hardware industry, the cost of hardware started to decrease, but the cost of software remained still. It seamed that everybody has found it's path to success, only the software industry was left alone. A newly created field named software engineering tried to address the problem of creating quality software effectively. Because so many industry fields had success stories with components, it seamed that the whole world, except the world of software, was already component-oriented! So it was only natural to follow this ideology, and introduce components to software as well. These software components were initially considered to be analogous to hardware components, especially to integrated circuits. Because of this, the term *software IC* (abbreviation of **I**ntegrated **C**ircuit) became popular. Components were seen by software engineers as a key solution to the software crisis. They thought that the Industrial Revolution of software will come through component-based software engineering. However, this didn't happen, and for a good reason. Yet the question remains: Why? What is so different in the world of software?

The answer lies in the fact that software's true nature is different than the nature of other engineering fields. Software is not the same as the products of these other engineering disciplines. That's because delivery of software means delivering the blueprints for products, not the product

itself. Computers can accept these blueprints and instantiate them. In other words, software can be seen only as a *metaproduct*. When acquiring software, it is these metaproducts that are actually deployed. The same holds true for software components, too. It is as important to distinguish between software and its instances as it is to distinguish between blueprints and products, between plans and a building. Plans can be parametrized, scaled and instantiated any number of times, but none of this is possible with actual instances. By using different parametrizations of the plan, instances with different "shapes" can be delivered. In conclusion, software is a generic metaproduct that can be used to create entire families of instances.

Terms and Definitions

In this section, the main terms will be described and defined. Because it seams that components and objects go hand in hand, both concepts will be explained together with other notions such as interfaces and context dependencies.

Components

As it was already mentioned, software components were introduced during the software crisis circa four decades ago as the solution to this crisis. They were imagined like the components from other engineering fields with features like increased flexibility and robustness, as well as shorter time to market. It can be said, that the most obvious fact about components is that *"components are for composition"* [Szyperski, Gruntz & Murer 2002]. Composition enables prefabricated items to be reused by rearranging them in new composites. Programming or developing software components and assembling new software systems from them is called *COP* (abbr. **C**omponent-**O**riented **P**rogramming) or *CBD* (abbr. **C**omponent-**B**ased **D**evelopment). According to [D'Souza & Wills 1998], *CBD* is *"an approach to software development in which all artifacts – from executable code to interface specifications, architectures and business models; and scaling from complete applications and systems down to small parts – can be built by assembling, adapting and "wiring" together existing components into a variety of configurations"*.

One way to capture the intuitive meaning of a term is to enumerate its characteristic properties. For a **component**, these characteristic properties are that it:

- is a unit of independent deployment (*cp1*),
- is a unit of third-party composition (*cp2*),
- has no (externally) observable state (*cp3*).

For a component to be independently deployable, it must be separated from its environment and other components. To achieve this, the component must encapsulate its constituent features. Also, because it is a *unit* of deployment, a component will not be deployed partially. A component is not so interesting when considered alone, but when it "communicates" with other components. As the second property (*cp2*) states, components are developed by different parties, so it's only natural to expect that a component will need to collaborate with components developed by other parties. This implies an important fact: a component developer cannot expect to have access to the construction details of all the components involved. The other implication is that a component should come with clear specifications of what it requires and provides. In other words, it needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces.

The third property (*cp3*) needs further discussion. To distinguish between the deployable unit and the instances it supports, a component shouldn't have any (externally) observable state – it is required that the component cannot be distinguished from copies of its own. A component can be loaded into and activated in a particular system. However, due to the stateless nature of components, it makes little sense to have multiple copies in the same operating system process as these would be mutually indistinguishable anyway. In other words, in any given process, there will be only one copy of a particular component. Indeed, in practice, components are usually heavyweight units with exactly one copy in a system. For example, consider a database server as a component. Let's suppose that there is only one database maintained by this server. This often leads to confusion, because this database server together with the database might be seen as a component

with an observable state. But according to the property *cp3*, this "instance" of the database concept cannot be a component. Instead, the component is the static database server program, which supports a single instance – the database "object". In other words, the database data is the instance. Again, as it was mentioned earlier, it all comes down to the separation of the immutable "plan" from the mutable "instances".

We have enough knowledge now to give formal definitions of components. Many have tried to define what a software component is, usually observing the term from different points of view. In the remainder of this section, some well-known definitions of components will be presented. The more important definitions will be extended with a brief discussion to deepen the understanding of that particular definition.

According to *Philippe Krutchen* from *Rational Software*, *"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces."* [Brown & Wallnau 1998]

According to the *Gartner Group*, *"A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime."* [Brown & Wallnau 1998]

Alan W. Brown and *Keith Short* have defined a component *"as an independently deliverable set of reusable services"* [Brown & Short 1997]. This characterization is rather rough and short, but holds a lot of meaning. First, the *"reusable services"* phrase implies that a component provides capabilities that many other components may wish to access. Thus, an important principle of *CBD* is that a component should have a **specification** which describes what a component does, and how it behaves when its services are used. This way, a potential client who tries to use these services can focus on the overall solution without concern for how these services are actually implemented. The developer, who provides the **implementation** for the component in some programming language, guarantees to meet the specification (at least in theory). A particular component can be replaced by another, as long as both guarantee to implement the same specification. The second phase, *"independent delivery"*, refers to the issue of context awareness of components. This implication is practically the same as the property *cp1*. Indeed, components are expected to collaborate with one another, not interfere, because they are usually unaware of the contexts in which they may be used. One consequence of this is that two components cannot share a common data structure directly. For instance, two components could not access the same columns in a relational database table[1]. Of course, this doesn't mean that components cannot have dependencies on one another. For example, a spell checking component may require the services of a dictionary manager component. In practice, both components are needed, but any of the two components can be swapped with a more appropriate equivalent.

Andy Ju An Wang and *Kai Qian* have formulated a more technical definition. According to them, *"A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface"* [Wang & Qian 2005]. From this definition, a component is a program or a collection of programs that can be compiled and made executable. Because it is self-contained, it provides coherent functionality. Because it is self-deployable, it can be installed and executed in an end user's environment. It can be assembled with other components, so it can be reused as a unit in different contexts. The integration is through the interface of the component, which means that the internal implementation of the component is usually hidden from the user.

However, probably the most popular definition of software components comes from

1 They may do so if a third component took the responsibility for updating that same column, unbeknown to the first two components.

Clemens Szyperski: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties"*. [Szyperski, Gruntz & Murer 2002] From this definition, it can be said, that components are specified by one party, implemented by another party, and finally used by a possible third party. The third party uses a component according to the specification of its interfaces and has no knowledge about the actual implementation of the component itself. The specification of an interface is contractual, i.e. it binds together a user of the interface and a provider of an implementation of the interface in a balanced manner. This contract should be precise enough to specify the required functionality, but also non-deterministic in order to allow the provider of the implementation to choose its own solution. In addition, a component may need other services from other components. These requirements must be clearly specified in a contractual manner as context dependencies. The final result is a component-based software system, which is more robust, as it is composed of smaller components, already tested (reused) by the market. The system is also more flexible, because old and less efficient implementations of components can be replaced by faster or more adequate ones, providing a more seamless evolution of the system. Finally, the time to market is shortened, since only the specification of the required services and the assembly of the components implementing them has to be performed.

Objects

Terms like instantiation, identity or encapsulation led to the notion of objects. The object-oriented style of programming consists in developing applications built of objects that communicate with each other. In contrast to the properties describing components, the characteristic properties of an **object** are that it:

- is a unit of instantiation, it has a unique identity (*op1*),
- may have state and this can be externally observable (*op2*),
- encapsulates its state and behavior (*op3*).

Because an object is a unit of instantiation, it cannot be partially instantiated. Since an object has individual state, it also has a unique identity that is needed to identify the object. This means, that an object can change its state during its lifetime, but the object's identity must need to remain the same. As objects are instantiated, there needs to be a construction plan that describes the state space, initial state and behavior of a new object. This plan must exist before the object can come into existence. Such a plan may be explicitly available and is called a **class**. When this instantiation is complete, this newly instantiated object needs to be set to an **initial state**. This initial state needs to be a valid state of the constructed object, but it may also depend on parameters specified by the client asking for the new object. The code to control object creation and initialization can be a static procedure residing inside the class itself, and it's usually called a **constructor**.

An object is an abstraction from a real-world entity, that can have associated items of information and can be manipulated by a set of operations. More precisely, an object is an abstraction of a data item characterized by a unique and invariant identifier, a class to which it belongs and a state, represented by a simple or more complex value. The class defines the behavior of the object. A class is an abstract data type characterized by a set of properties (*attributes* and *operations*) common to its objects, together with the means for creating objects with these properties. The state of an object cannot change spontaneously but only as a consequence of operations performed on it. The set of all objects created by a class form the class extent. A class (*subclass*) *inherits* from another class (*superclass* or base class) if the subclass extent is a subset of

the superclass extent. This concept is called **inheritance**. The objects in the subclass extent must support all the operations that are supported by the objects in the superclass extent. However, the objects in the subclass extent may carry out these operations in a specific way and they may also have additional operations. Objects can interact by using each others operations. They are not allowed to directly access other object states. The implementation of object operations is also hidden from other objects.

Interfaces

A component specification often includes a set of services which naturally group into meaningful clusters. These clusters of service specifications are known as **interfaces**. An interface summarizes how a client should interact with a component, but still hides underlying implementation details. In fact, an interface may exist separately from components which try to implement it. This means that an interface can exist even before an implementation exists. This is because an interface summarizes a part of behavior and responsibilities for some situation.

For example, consider the management of names and addresses as a useful and potential part of behavior. This is defined by operations like: “add new address” for a person, “how long” has a person been at an address, or “where” was a person living three years ago. Let's call this interface as *ISimpleAddressManagement*. Suppose that two competing companies, *XSoft* and *YSoft*, decide to implement this interface by building components. Suppose that *ClientA* needs management capabilities of names and addresses. By browsing a catalog of interfaces, the client realizes, that he needs the *ISimpleAddressManagement* interface, and there are two companies offering an implementation to this particular interface. The client decides that he will buy from *YSoft*, as it's cheaper. At the same time, another client, *ClientB*, realizes that he also needs a name and address management capability. By searching the interface catalog, he finds *ISimpleAddressManagement*. After considering the offers by the two companies, *ClientB* decides to go for *XSoft*'s solution, which is more expensive, but the offered component also implements the *IAdvancedAddressManagement* interface, which offers more advanced management of names and addresses. This simple example shows that an interface also has an important role in the component market.

To summarize, interfaces are the primary means for clients of components to make decisions on whether a particular component is suitable for their needs. Because of this, interfaces must be defined precisely and unambiguously and be easy to search. Also, a component can offer multiple interfaces, leaving to the clients the possibility to choose whether to use all interfaces or only a portion of them. Lastly, well-defined interfaces encourage competition among component implementers, allowing clients to choose among them based on non-functional attributes like cost, quality of service, reliability, etc.

Explicit Context Dependencies

Besides the specification of the interface about what the component provides (often called **provides** or **provided interface**), a component must also specify what it requires so that the component can function properly (often called **required** or **requires interface**). This latter interface provides information about what the component needs from its environment. These requirements are called **context dependencies**, referring to the context of composition and deployment. They include the component model that defines the rules of composition and the component platform that defines the rules of deployment, installation and activation of components. For example, a mail merge component would specify that it needs a file system interface.

Today, there are many so-called “component worlds” that partially coexist, partially compete and partially conflict with each other, for example: *Oracle*'s *Java*, *OMG*'s *CORBA* or *Microsoft*'s

.NET. Because of this, a component's specification of context dependencies must include its required interfaces as well as the component world (or worlds) that it has been prepared for. More information about component worlds will be presented later in the section “Standards”.

The Journey to Component Software

When we read news and articles about the IT world, we often come across headlines such as a company has decided to develop an entirely new revision of software from scratch, or another company has decided to outsource parts of a software to foreign companies to ease the development. This is not a coincidence, because traditional software development can be divided into two main camps. At one extreme, the project is developed entirely from scratch, often called as custom-made software. At the other end, everything is outsourced – in other words, standard software is bought from the shelf and parametrized to provide a solution that is close enough to what is needed.

Full **custom-made software** has a significant advantage: it is highly adapted to customer needs, and it can take advantage of any in-house proprietary knowledge. Hence, it can possess various benefits compared to similar solutions from the competition. But 100% custom-made software also has severe disadvantages. First of all, it's a very risky undertaking, because developing from zero is very expensive. Also, as a software product tends to take from various aspects of knowledge to form a coherent whole, there is a possibility that the finished product will have many "weak" points. This is understandable, because a particular software company has its own areas of expertise, but that same company cannot possibly excel in all fields and technologies. Lastly, because the product is developed in-house, stakeholders often demand for perfectionism. As a result, many large custom-made software products fail in some way, usually because they are too late.

With all these flaws, it seams that the outsourced software is the way to go. Production of custom-made software is *outsourced* under fixed-price contracts to limit the financial risk. To cover the time-to-market risk, **standard software** is bought – software that is only slightly adjusted to actual needs. Maintenance, product evolution and interoperability is left to the vendor of the standard package. But this type of development also has its problems. First of all, standard software may necessitate the reorganization of the business processes affected. This undertaking in itself is not bad when it's done for the sake of the company's evolution, but it could be troublesome when the only reason of the undertaking is to adapt for the standard software. The second problem is that standard software is standard, which means that the competition has it as well, so it's harder to prove the benefits of that particular solution. Lastly, as standard software is not under local control, the speed of conformance to changing needs is up to the vendor itself.

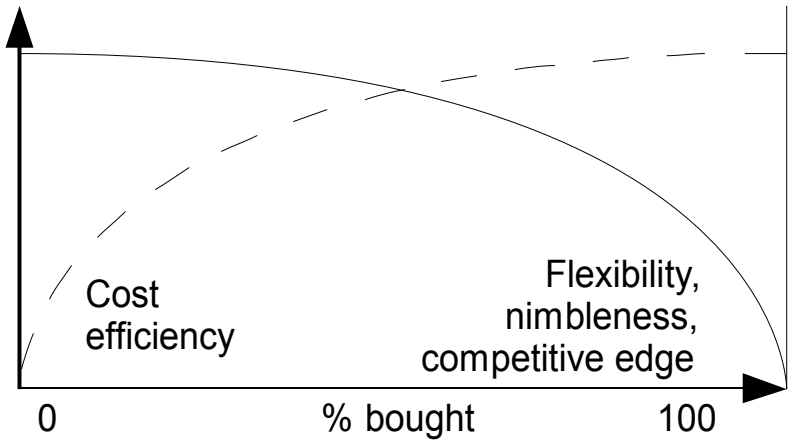

Image 1: Spectrum between make-all and buy-all

When these two extremes are observed, it's clear that outsourcing is a better decision, but the goal here is to minimize the disadvantages. The idea is to take the best of both extremes, so the ideal solution lies somewhere between them. The concept of **component software** represents that middle path. Although each bought component is a standardized product, with all the attached advantages, the process of component assembly allows for significant customization. If there is a competition between components, the company can choose between similar components based on the company's priorities, such as performance, resource-friendliness, robustness, cost, etc. Moreover, some components (usually the most important ones) can be developed in-house as custom-made components to suit specific requirements. Image 1 illustrates this problem.

That's how **Component-Based Software Engineering** (*CBSE*) emerged, and has become one of the major disciplines of software engineering. The major goals of *CBSE* were [Teiniker 2007] :

- to provide support for development of software systems as assemblies of components;
- to support development of software components as reusable entities;
- to facilitate the maintenance and upgrade of systems by customizing and replacing their components.

Unfortunately, developing excellent component software based on component technology is not enough to establish a successful market. The component approach, as any other, needs *critical mass* to take off. This critical mass is attained when the offered products are of sufficient variety and quality, there is an obvious benefit of using the technology, and the offering is backed by sufficiently strong sources. Once this critical mass is reached in a market segment, the use of that particular approach (in this case, the use of components) becomes inevitable, forming a “whirlpool” that pulls in everything in its path, including the traditional solutions. Developers using traditional solutions will soon feel that opposing to this pulling force will be more and more difficult. Indeed, when a new approach becomes popular, one of its great benefits is that solutions made by this particular approach improve in quality much faster than the rest. At the end, many developers will decide to accept this new approach. Of course, developers can also choose to escape from the whirlpool, but it also eliminates the impulse that can be gained from the mighty pull of this whirlpool.

Preparedness for an emerging new approach can be the deciding success factor for a company. Moreover, being ready for a new approach before the competition means great advantages. The first organization to get ready for the new approach can set standards and shape the then emerging market to its own advantage. This way, instead of waiting for others or claiming that it is unlikely that the market of the new approach will ever form, these organizations will take the lead and conquer the most important parts of that particular market.

In the previous paragraphs of this section, we've talked about the outsourcing of components, in other words, about the reuse of components themselves. However, there is also a need to discuss about outsourcing of parts of a component, about reuse of component parts. It's obvious, that a component would be the most useful, if it would offer everything the client needs, and it would require nothing in return. In other words, if it would offer perfect provides interfaces, but no context dependencies at all. Actually, in theory, it could be technically possible to construct such “magical” components by bundling in all required software, but that would clearly defeat the purpose of using components in the first place.

Instead of building a self-sufficient component with everything bundled in, a developer can decide to build with “maximum reuse” in mind. In this case, the developer should outsource all

secondary parts of the component to avoid redundant implementations, and implement only the prime portions of the component, which represent the main "selling" functionality of the component. There are many advantages to this "maximum reuse" ideology, but unfortunately, it has one serious issue: the explosion of context dependencies. If component worlds weren't so changing across time, then this wouldn't be a problem. After all, it's not that hard to satisfy the context dependencies only once, and forget it for the rest of time, knowing that the environment of that particular component will never change. However, thinking that there is no change in the world of software, would be naive at best. On the contrary, there is a constant evolution in the world of software: components evolve, the platform evolves, component worlds evolve. With this fact in mind, it's clear that a component with that many context dependencies would not survive, because it would be cumbersome to update it every time a change happens. To summarize: *"Maximizing reuse minimizes use."* [Szyperski, Gruntz & Murer 2002]

In practice, component developers should strive for a balance between these two extremes. Increasing the context dependencies usually leads to "leaner" components, but also to smaller markets. Moreover, when a change happens in the component itself or in the environment, it's harder to update the component to satisfy the context dependencies, resulting in a more "vulnerable" component. At the other hand, increasing the degree of self-containedness reduces context dependencies, increases the market, and makes the component more robust, but also leads to "fatter" components. Image 2 illustrates this problem. Note that it's not a coincidence that Image 1 looks so similar to Image 2. Image 1 illustrates the problem of outsourcing parts of the system (that is, components), and Image 2 shows the problem of outsourcing parts of components.

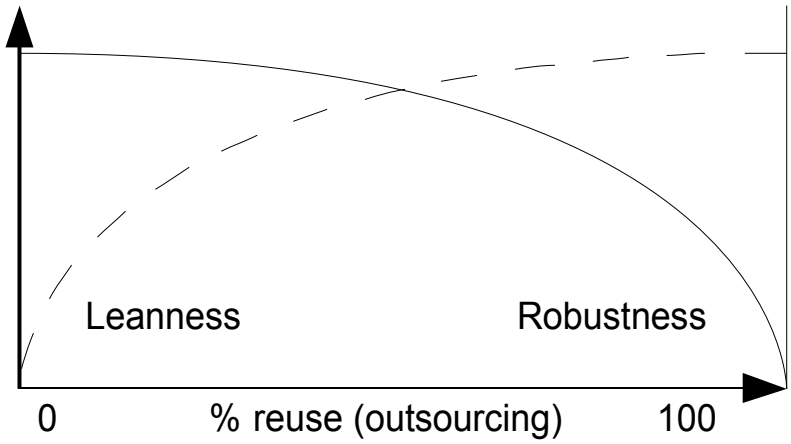

Image 2: Degree of reuse within a component which controls the robustness (limited context dependence) and leanness (limited "fat")

The Importance and Goals of Components

In the previous sections of this chapter, an abundance of information was presented about components and *CBD*. Yet, we've never answered a particular question: Why are components and *CBD* so important? Of course, some pieces of the answer were hinted at different places, but it is now time to sum up these pieces to form a coherent answer. Components are important, because they provide a higher level of abstraction. There are a large number of reusable component libraries that assist in the development of applications for various domains.

There are three major goals in *CBD*:

1. **Conquering complexity**

   We are living in a complex world at information explosion age. It is well known, that the world creates millions and millions of gigabytes per year. In computer science, the size and complexity of software have been increasing significantly. For example, the *Linux* kernel had 10.000 lines of source code in 1991, but in 1999, this number ballooned up to a million. The *Solaris* kernel had 850.000 lines of source code in 1993, but over two million in 2000. Similarly, the *Windows NT* kernel had six million lines of source code in 1993, but in 2000, this number had reached 29 million. [Tanenbaum 2001] *CBD* provides an effective way to deal with the complexity of software. This principle is often called as the "divide and conquer" principle.

2. **Managing change**

   It is said that change is inherent: user requirements change, specifications change, human resources change, the budget changes, the technology changes, etc. One of the most important principles in software engineering is the significance of *change management*. Primary emphasis should be placed on the dependencies between components during architecture and design phases. *CBD* provides an effective way to follow this software engineering principle. Planning, designing and building for change is more seamless, because components are relatively easy to adapt to new and changing requirements. By building systems out of reusable components, dealing with change can become less troublesome.

3. **Reuse**

   Software reuse allows to design and implement something once and to use it over and over again in different contexts. With reuse, productivity increases and the quality is improved, resulting in a better and faster-to-market product.

   There are different levels of software reuse. The lowest level of reuse is *source code copy*. On the next level of reuse are the *procedural function libraries*, which are better, but they are not extensible. *Class libraries* are even better, because they are extensible. However, class reuse requires deep understanding of the class itself before utilization. Moreover, clients will be affected if internals of classes change. For example, in an *OO* language such as *Java*, derived classes are coupled to the base class implementation. Any change in the base class can break the derived classes. Besides, this level of reuse is language-specific: reuse across languages is not possible.

   CBD supports the highest level of reuse, because it support various types of reuse. The most important types are white-box, black-box and gray-box reuse. These types of reuses refer to the visibility of an implementation "behind" its interface. In a **white-box** abstraction, implementation of the particular element is fully available, and can be studied to enhance the understanding of what the abstraction does. However, this doesn't mean that clients can

freely manipulate with the abstraction.[2] In a **black-box** abstraction, clients know nothing beyond the interface and its specification, so a black-box reuse is based on the principle of information hiding, because it refers to the concept of reusing implementations without relying on anything but their interfaces and specifications. **Gray-box** reuse is somewhere in between white-box and black-box reuse. Gray-boxes reveal only a controlled part of their implementation.

As the size and complexity of software systems grows, the identification and management of interconnections between the pieces of the system becomes more and more difficult. *CBD* provides an acceptable solution to deal with the complexity of software, the constant changes of systems and the problem of software reuse. *CBD* is thus an adequate approach for developing large software systems.

2 Because of this, some authors further distinguish between *white-boxes* and *glass-boxes*, with a white-box allowing for manipulation of the implementation, and a glass-box only allowing study of the implementation.

Standards and Component Forms

For component markets to develop, **component standards** must also exist. Standards are useful for:

- establishing agreement on common models, enabling interoperation in principle;
- creating agreement on concrete interface specifications, enabling effective composition;
- ensuring agreement on overall architectures to assign components their place in a larger picture of composition and interoperation.

For a component to find a reasonable number of clients, it needs to have *requirements* that can be expected to be widely supported, and, of course, it must also provide *services* that can be expected to be widely needed. As the number of potential uses and clients grow, the possibility that a component could address all needs while being deployable in all environments decrease rapidly. The unavoidable middle ground that both clients and vendors need to seek is based on environment standards. There are no rules about how these standards are created. Sometimes, it is created by the first vendor or company arriving on a new market segment. At other times, it is created by a regular standardization body. Based on this idea, component standard approaches can be divided into two camps:

- **build working markets first, formulate the standards later.** At this side, the biggest player is *Microsoft* with *.NET* and *ActiveX/COM*. Another important player is *Sun Microsystems* (now owned by *Oracle*) with *Java*, *JavaBeans* and *EJB*.
- **formulate standards first, build the markets later.** The prime player in this camp is the *Object Management Group* or *OMG*, a huge industry consortium with *CORBA* (abbreviation of **C**ommon **O**bject **R**equest **B**roker **A**rchitecture).

The question, who will win the standardization race in component technology, is actually not that important. In fact, as long as the markets continue to thrive, multiple standards can coexist and compete. However, too many competing standards are certainly not useful.

It's obviously clear, that defining a software component generally isn't an easy task. We have given many definitions of components in earlier parts of this paper. One of the reasons of these hardships is that components practically live in the whole life cycle of the project, but in different forms. A person's view of components changes during the phases of a project. From requirements and specification, through design, to assembly, deployment and runtime, the characteristics a person wants from a component vary. A number of **component forms** can be identified, and each form reflects some aspect of a component during the development life cycle. [Cheesman & Daniels 2001] [Wang & Qian 2005] There are five component forms[3] also shown on Image 3:

1. **Component specification.** When we look for a component to plug in, it's important to know what that component does. For example, when a burnt-out light bulb should be replaced in a lamp, it's important to know the power rating of the replacement light bulb. All regular light bulbs have the same shape, so they can all be inserted into a regular lamp socket, but the power rating of these bulbs can range from 45W to 120W. If the maximum recommended power rating of the particular lamp is 60W, then inserting a 45W or 60W light bulb will

3 *Cheesman* and *Daniels* defined initially six component forms in [Cheesman & Daniels 2001], with *component standards* as the first form. However, it was dropped out by *Wang* and *Qian* in [Wang & Qian 2005], probably because a component standard is not a form, but an environment where components live.

solve the problem, but inserting a 100W bulb will destroy the lamp by fusing the lamp socket. So, the specification of what a component does must also be part of a valid definition. Technically speaking, a component specification represents the specification of a unit of software that describes the behavior of a set of *component objects* and defines a unit of implementation. Behavior is defined as a set of *interfaces*. A component specification is realized as a *component implementation*.

2. **Component interface.** A major part of a component specification is the definition of component interfaces. This form presents a definition of a set of behaviors that can be offered by a *component object*.
3. **Component implementation.** This form is a realization of *component specification*, which is independently deployable. This means that it can be installed and replaced independently of other components. However, it doesn't mean that it's independent of other components – it may have many dependencies. From an assembly perspective, the *component specification* is more important than the way that specification is realized or implemented. A particular component can be replaced with another (of an equivalent specification) without affecting the assembly. For example, a 60W light bulb can be replaced with another 60W bulb from a different manufacturer. What matters from an assembly point of view is the interdependency between the parts, not the way those parts work. This clear separation of *component specification* from component implementation is therefore another important characteristic of a component. The assembly itself should only depend on the specification. If there is any dependency on the implementation then the ability to replace that piece easily will be lost.
4. **Installed component.** The installed form of a component is an installed (or deployed) copy of a *component implementation*. Each time a *component implementation* is installed, an installed component is created (the installed version of the component, known to the environment). The *component implementation* is deployed by registering it with the runtime environment. This enables the runtime environment to identify the installed component to use when creating an instance of the component or when running one of its operations.
5. **Component object.** The existence of state or content of the component at runtime must also be considered. While the services provided by a component are important, so is the information or data managed by that particular component. Technically speaking, a component object is an instance of an *installed component*. It's a runtime concept. Similar to objects in the *OO* paradigm, a component object has its own data and a unique identity, which performs the implemented behavior. Only component objects can actually do anything. An *installed component* may have multiple component objects (which require explicit identification) or a single one.[4]

4 It should be clear by now, what is the difference between components and component objects, between the immutable plan and mutable instances. It was mentioned earlier, that components shouldn't have any observable state. Now it's clear, that component objects can have a state. This is because a component is an abstract concept, a blueprint, a plan, which tells what the component should do. The component “in action” is the instance of that particular component, and it is called a component object.

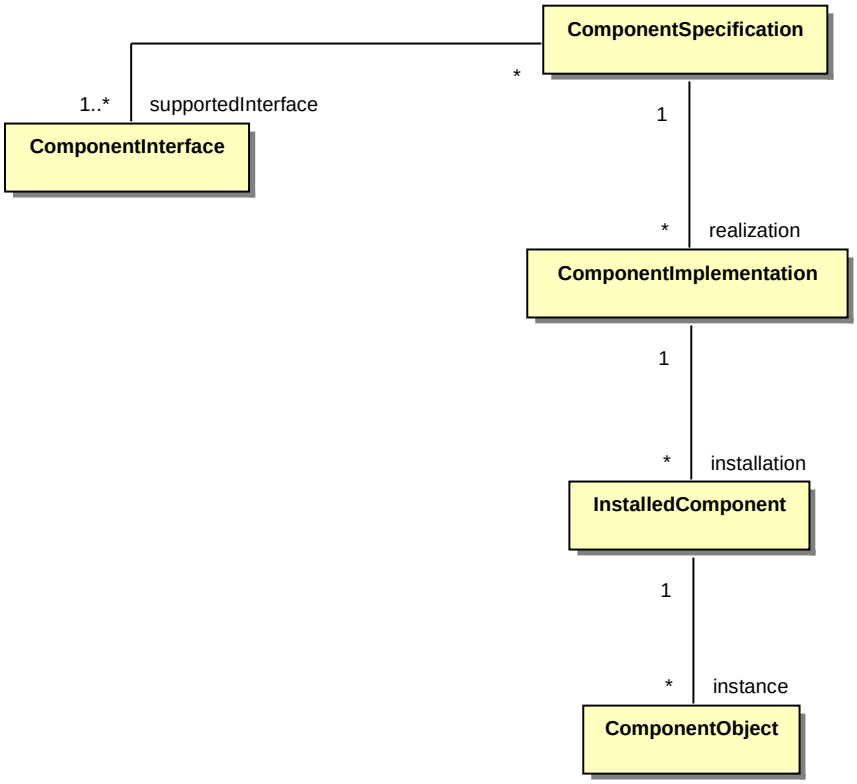

Image 3: Various component forms

A practical example of a component and how it can take all these forms will be presented in the section “Objects versus Components from the Client's Point of View”.

Objects and Components

Objects and components are often mentioned together in the literature. Some authors don't even try to differentiate the two concepts, calling even objects as components. At the other hand, others don't really consider the debate between components and objects necessary, claiming that the difference between them is “obvious”. However, there is a need to make a comparison between the two concepts, both technically and economically. In some segments, it is even tempting to compare the achievements of these concepts against each other. By making these comparisons, it will be easier to acknowledge their strengths and weaknesses, and even putting them to work together in order to utilize the best of both worlds.

In this chapter, the main emphasis will be on objects and components from different aspects of view: from market segments and the client's point of view to abstract and technical aspects. The last section will give some interesting remarks and a demonstration about using both objects and components in a software project. This chapter is mostly based on [Szyperski, Gruntz & Murer 2002] and [Petre 2000].

Objects versus Components in the Market

In the component approach, the actual units that are deployed are the components themselves. In the object approach, the units of deployment are classes, or more likely, a set or framework of classes, compiled and linked into a package. This statement is not in accordance with frequent claims, that objects are deployed, because in reality, objects are almost never sold, bought or deployed. Objects that logically form parts of component “instances” are instantiated as needed, based on the classes that have been deployed with a component. Although a component can be a single class, in reality, it is more likely to be a collection of classes (sometimes called a module[5]). Components as a whole are thus not normally instantiated. Also, a component could just as well use some totally different implementation technology, such as pure functions or assembly language, and look not at all object-oriented from the inside.

One can ask, if classes are so similar to components, why did object technology not succeed in establishing significant object markets? There are many reasons. First, the definition of objects is purely technical – briefly, encapsulation of state and behavior, polymorphism and inheritance. There are no notions of independence and late composition, and because of this, object technology is mostly used today to construct monolithic applications. Second, object technology tends largely to ignore the aspects of economies and markets and their technical consequences. At the beginning, people predicted object markets, places with catalogs full of objects, or, more likely, classes, class libraries and frameworks. In fact, the opposite has occurred. Today, the number of such sources are scarce. Even the available sources are mostly driven by vendors that provide such semi-finished software products to sell something else.

The misprediction of object markets is understandable. For technologists and academics,

5 Components aren't modules in the strict sense of the word, however, they are rather close. *Modules* were introduced by modular languages in the late 1970s by *Niklaus Wirth* and others. The most popular modular languages were *Modula-2* (by *Wirth*) and *Ada* (in *Ada*, modules were called packages). Unlike classes, modules can indeed be used to form minimal components. In fact, a component is a set of normally simultaneously deployed atomic components. This distinction implies that most atomic components will never be deployed individually (although they could). Instead, most atomic components belong to a family of components and a typical deployment will cover the entire family. It can be said, that an atomic component is a module and a set of resources. This means that a module is an atomic component with no separate resources. (It's important to know, that *Java* packages are not modules in this strict sense. The units of deployment in *Java* are class files, and a single package will be compiled into many class files.) A resource is a collection of typed items.

markets are often considered marginal, as something left for others to worry about once the technological problems have been solved. However, components are much about the potential of technology as they are about the technology. In other words, the economy of components is part of the component itself. The additional investment required to produce components, rather than fully specialized solutions, can only be justified if the return on investment follows. A component must have a sufficiently large number of uses, and therefore clients, for it to be viable. Repeated use is the main idea behind the notion of "reuse".

To summarize, there is no doubt that the vision of object markets didn't happen. On the contrary, most of the few early component success stories weren't even object-oriented, although some are. More recently, based on true component technologies, successful markets began forming. One of the most popular markets of components is *ComponentSource*, which can be found on <www.componentsource.com>. According to [ComponentSource 2010], *ComponentSource* was established in 1995 to provide software development products to developers. Today, their eCommerce open market counts over 10.000 *SKUs* (abbreviation of **S**tock-**K**eeping **U**nit), and the number of publishers is over 230. *ComponentSource* has over 115.000 customers from 175 countries, the websites are visited by 300.000 unique visitors per month, and the number of registered users is now over 1.000.000. *ComponentSource* has offices in the USA, United Kingdom and Japan, and their website is localized into seven languages. In terms of components, *ComponentSource* has almost 2.000 components. Most of them are *.NET* components (more precisely, 1277), but there are also *ActiveX/COM* components, *Java* components and *JavaScript/AJAX* components. The official logo of the company can be seen on Image 4.

Image 4: The logo of ComponentSource

Objects versus Components from the Client's Point of View

One can ask, how can a simple client, who isn't a professional programmer and maybe not even experienced in the *IT* world, tell the difference between objects and components. From the client's point of view, there is one aspect that is often overlooked, but it can nevertheless make a distinction between these two notions: components exist on a level of abstraction where they directly mean something to the deploying client.

Most objects have no direct meaning to clients who are not programmers. Class libraries and frameworks are typical developer tools and require highly qualified developers for their proper use. It's also clear that the construction of components should be left to professionals. However, for components to be successful, composition and integration – that is, component assembly – must not generally be confined to such a relatively small group. Components are often purchased from providers and deployed by clients. These customers greatly value products that are obviously useful, easy to use and can be safely mixed and matched[6]. Whether these products are object-oriented in the inside or not, is the least of their concerns.

Objects are rarely shaped to allow for mix-and-match composition by a third party (also known as *"plug and play"*). Configuring and integrating an individual object into a particular system is not normally possible. This is one of the reasons, that objects cannot be sold independently.

There are many successful component-oriented stories. Earlier, the most popular have been *Microsoft*'s *Visual Basic* (now suppressed by *Microsoft*'s *.NET*). Later, *Sun Microsystems*' (now *Oracle*'s) *Java* and *EJB* (abbreviation of **E**nterprise **J**ava **B**eans) followed. However, the oldest successful examples are all modern operating systems. In this case, for applications can be said, that they are **coarse-grained components** executing in the environment provided by the operating system. Other older component examples are the relational database engines. However, the most recent successes are the **fine-grained components**, in other words, the *plug-in* architectures. *Netscape*'s *Navigator*, a now defunct web browser, was one of the first examples to use this architecture so expansively. Other web browsers followed. Plug-ins, gadgets, widgets, and other synonyms are all examples of fine-grained components.

To demonstrate the use of components from a simple client's point of view, let's consider one of the most used application on computers: *Microsoft Word* [Cheesman & Daniels 2001]. *Microsoft Word*, as part of the *Office* package, is a component itself, and it has many other components, too. Although *Word* has many other components, let's consider only the two most obvious of them: the component representing the application itself, and the components representing the documents the user creates, opens, manipulates or closes. When the user buys the CD or DVD with *Microsoft Word* on it, the user won't find the *component specifications* on the disk for obvious reasons[7]. Instead, the user gets an executable file called `winword.exe` that packages the *component implementations*. These are implementations specific to the used component standard, for example *COM*. The installation process, simply speaking, involves copying the physical files (including `winword.exe`) into a chosen location and registering their contents with the runtime environment (in this case, *COM*). This creates two *installed components* – components, which are known to *COM*. Running *Microsoft Word* on the default settings initially creates two *component objects*: one representing the application object itself, which acts as a frame, and another as a default new document object. If the user wants to create further document objects, he/she will click the *New*

6 Multiple components from different sources can coexist in the same installation. However, an arbitrary or random combination of components can lead to misbehavior, but purposeful deployment rarely leads to this situation.

7 First, simple users wouldn't find the specifications useful, as these are rather technical. Second, *Microsoft Office* is a closed-source application family.

button on the application object, or choose the *File → New* command from the menu. This simplified example proves just how frequent components are, and shows the many forms a component can take (explained earlier in the "Standards and Component Forms" section). The diagram illustrating the component forms of *Microsoft Word* can be seen on Image 5.

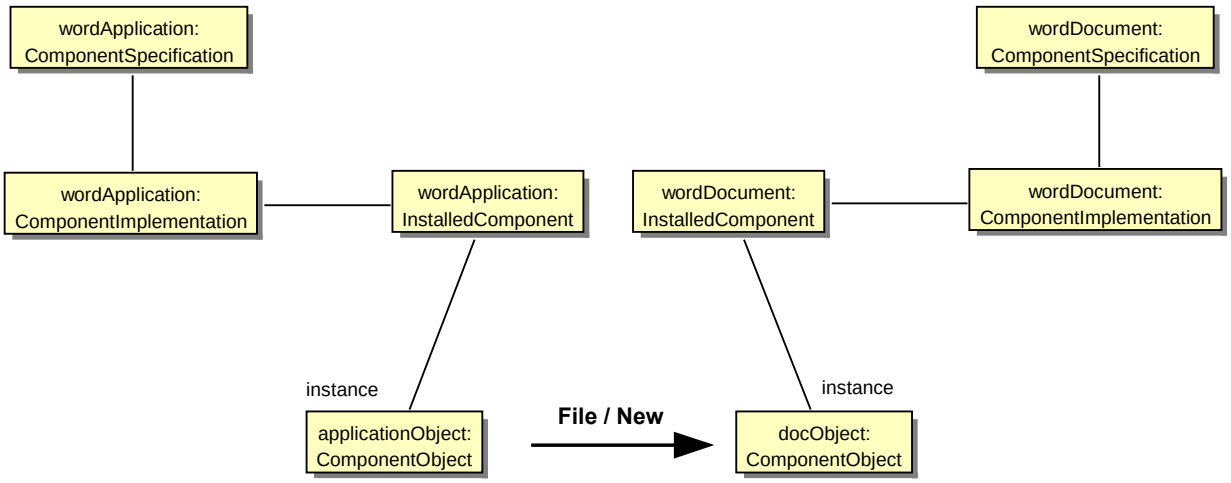

Image 5: The component forms of Microsoft Word

Objects versus Components – the Little Details

As it was mentioned in the previous section, customers aren't very interested in the innards of a component. But for developers, that is one of the most crucial parts. The functionality, assembly and overall quality of the component depends on the chosen implementation language. However, in the world of components, making the right decision is even harder.

Usually, a component is likely to act through objects and therefore would normally consist of one or more classes. In this case, a suitable *OO* language should be selected for development. For example, the developer could choose a language such as *C++*, but this language doesn't directly support a component concept (although it's possible), so management of dependencies could be rather difficult. Merely replacing *C++* with another, perhaps clearer *OO* language won't solve the problem. The component-oriented approach goes much deeper that simply picking the right language. It is said, that many fragilities of *C++* were addressed in *Java*, and in *C#* probably even more, but nevertheless, it's still easy to fail when the goal is to develop something component-oriented.

Let's consider classes and components on the implementation level. It would be tempting to see, what constitutes a minimal component. According to [D'Souza & Wills 1998], a class can be considered a component, if the explicit context dependencies, such as required and provided interfaces, are packaged, as well. Consider the following Java code:

```
class C1 implements I1, I2 {
     public T0 foo(T1 x);
     private T2 y;
}
```

In this case, the minimal component that could contain class `C1` would also have to include the specifications of `I1`, `I2`, `T0`, `T1` and `T2`. A package with a single class; the interfaces it implements; and the interfaces it requires of any other objects it deals with (input parameters, returned objects, factory objects it uses to instantiate other objects, etc.) would form a minimal component.

A component may contain multiple classes, but a class is necessarily confined to being part of a single component. Indeed, partial deployment of a class would not normally make sense. Of course, just as classes can depend on other classes using inheritance, components can depend on other components – this is an *import* relation. The superclass of a class do no need to reside in the same component as the class itself. Where a class has a superclass in another component, the inheritance relation between these two classes crosses component boundaries, forcing a corresponding import relationship between the two components.

At the other hand, there is absolutely no need for a component to contain classes only, or even to contain classes at all. In other words, great component-oriented software can be developed even without the help of object-orientation. A component can just as well use some fundamentally different implementation technology, such as pure functions, traditional procedures or even assembly language. To summarize, *"truly component-oriented languages have yet to arrive and even then they will not solve many of the intrinsic engineering tradeoffs that engineers of software components and component software have to face and address"* [Szyperski, Gruntz & Murer 2002]. However, although for large systems, it's fairly clear that components are a major step forward from objects (classes), that doesn't mean that objects should be avoided. On the contrary, object technology, if used wisely, is probably one of the best approaches to develop component software.

On the Abstract Level

To make a more abstract comparison between object and components, some intuitive grip can provide further insight. It can be said, that components are *"bigger"* entities than objects. Also, components appear to be more suitable entities for software architecture (an early software engineering phase), while objects tend to be more suitable in the implementation phase (late software engineering). However, both arguments are quite inaccurate, because they depend on the size of the project itself. At different granularity levels, objects could be used for architecture, and components for implementation. Similarly, saying that components are bigger than objects is a rather vague statement, and even if it's true, the question remains: how can we compare these notions, when we know that components don't even need to be object-oriented?

What is then the conceptual difference between objects and components on a more abstract level? According to [Petre 2000], the answer lies in their *roles*: while objects are suitable for describing real-world entities, components are suitable for describing the services of real-world entities. In other words, objects are suitable for describing the *problem domain* of a system, while components are suitable for describing its *functionality*. Observed from this perspective, objects of a system partition the *state space*, while components partition its *service space*.

By following this train of thought, it's clear, that using a service from a components means to specify the required service and to use *any* component providing an implementation of that service. Using a service from an object, however, means to specify *which* object is to be used and to use the service that particular object, with its particular state, can provide. To illustrate this, let's consider an email application that must have an address book import functionality. In a component-based project, that should be the *AddressBookImport* component. In fact, the project should require such a component that should be able to import address books from many already existing sources with great accuracy and with satisfactory performances. The identity of this component is not important for the project as a whole. However, in an object-based project there might be a hierarchy of classes specifying the different types of sources, with different degrees of accuracy and speed. In this case, it is the identity of the object, or the class to which extent the object belongs to, that determines what "services" a particular address book importer can perform.

The above information leads to an interesting remark: object-orientation can be considered as a more *mathematical* approach to software, and on the other hand, component-orientation can be seen as a more *engineering* approach. As it was already mentioned, software components have a particular nature that combines mathematical, engineering and even market aspects. The engineering dimension comes from the basic concept of the term itself – that of building systems out of already existing parts. Indeed, objects model the problem domain of a system similarly to mathematics offering abstract models of the world. By analogy, components model the functionality of the system similarly to engineering disciplines offering practical solutions to problems.

On the Technical Level

To make a more complete comparison between objects and components, their technical differences should be also considered. According to [Teiniker 2007] and [Wang & Qian 2005], the most noteworthy differences are:

- **Reuse.** Components are highly reusable, while objects support only lower levels of reuse.
- **Multiple interfaces per component.** Typically, an object implements a single class interface, which may be related to other classes by inheritance. At the other hand, a component can implement many interfaces which may not be related by inheritance. Components can provide navigation operations to move between different component

interfaces. Navigation in objects is limited to moving up or down the inheritance tree via cast or narrow operations.

- **Extensibility.** While objects are implemented in an *OO* language, in theory components can be implemented in any programming language. This is in accordance with the information presented in earlier parts of this work. Components can be viewed as providers of functionality that can be replaced with equivalent components written in another language. In this case, "equivalent" means that the replacement component must provide at least the services that the environment expects of the original, and must expect no more than the services the environment provides to the original.
- **Granularity.** By comparison, a component has a much larger granularity and therefore usually more responsibilities than an object. One reason that led to the introduction to components was to group objects to larger entities in order to reduce the overall complexity of a system.[8]
- **Improved communication.** Components have a more extensive set of intercommunication mechanisms than objects. While the most common communication mechanism between objects is the basic *OO* message, between components there are many other types, such as workflows and events.
- **Higher-level execution environment.** Component models define a run-time execution environment, called component container, that operates at a higher level of abstraction, than access via ordinary objects. The container provides additional levels of control for defining and enforcing policies on components at runtime.
- **Obedience.** Components are designed to obey rules of the underlying component framework. In contrast, objects are designed to obey the *OO* principles.

Table 1 gives another brief summary of commonalities and differences among objects and components [Wang & Qian 2005].

8 Careful readers may notice that there is a potential relatedness between the granularity of objects and components and their "size" mentioned a little earlier. However, while it was noted that it's too vague to state that components are bigger than objects, it's rather safe when they are compared by their granularity. Indeed, component are on a higher level of granularity than objects. However, it must be noted, that this statement holds only when the observed components are object-oriented on the inside.

Capabilities	Object-orientation	Component-orientation
Divide and conquer • manage complexity • break a large problem down into smaller pieces	Yes	Yes
Unification of data and functions • a software entity combines data and the functions processing those data • improves cohesion	Yes	Yes
Encapsulation • the client of a software entity is insulated from how that software entity's data is stored or how its functions are implemented • reduces coupling	Yes	Yes
Identity • each software entity has a unique identity	Yes	Yes
Interface • represents specification dependency • divides a component specification into interfaces • restricts inter-component dependency		Yes
Deployment • the abstraction unit can be deployed independently		Yes

Table 1: Comparison between object-orientation and component-orientation

Objects and Components Working Together

In the previous sections, in order to better understand the differences between objects and components, both notions were placed face to face. However, this doesn't mean that these concepts are exclusive, with one ruling the other out. On the contrary, according to *Luigia Petre* in [Petre 2000], a real-world entity can be modeled using both notions. Based on this, she proposed a guideline for a software engineering process that incorporates both notions and shows their real potential.

Object-Oriented versus Component-Based Approach: an Example

As an example, let's consider a mail delivery simulation. In the simulation, the system is required to send letters and packages to their destinations as well as to confirm their delivery. Suppose that we need to model it with both concepts.

In a component-based approach, three components were identified: *MDS* (abbr. **M**ail **D**elivery **S**ystem), *Driver* and *PostVehicle*, with *MDS* using the services of the other two. The *MDS* component provides two services: *SendPackage*, which sends letters and packages to their destinations; and *ConfirmDelivery*, which, as the name suggests, confirms the delivery of letters and packages. In addition, the *MDS* component requires three services: it needs someone to drive a post vehicle; it needs someone to load the letters and packages into the post vehicle; and it needs the post vehicle itself to do the transport of the deliverable items to their destinations. The first two required services (let's call them *Drive* and *Load*) will be implemented in the *Driver* component, and the third service (let's call it *Transport*) will be implemented in the *PostVehicle* component. The interacting components and their interfaces are represented on Image 6 using a *UML* component diagram.

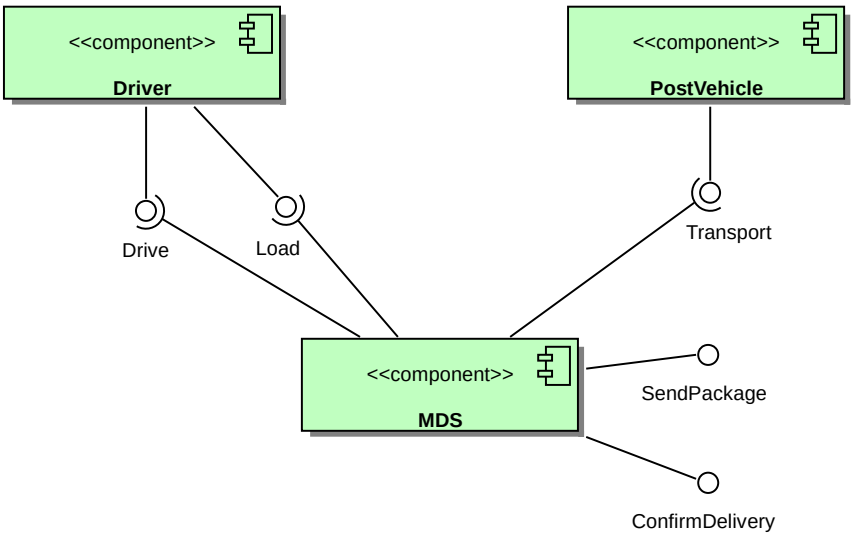

Image 6: Component view of the Mail Delivery System

Now let's consider the same system in an object-oriented approach. In an *OO* software, each object is concerned with its own responsibilities and is constrained by the properties of its corresponding class. Provided that a suitable class decomposition was found for the system under development, focusing on classes construction rather than on the system as a whole eases the software engineering process. Because objects are quite different from components, modeling the

same system both ways is challenging. In the *OO* approach, a number of classes were identified. The most important classes are *PostVehicle* and *Driver*. As a real-world vehicle consists of many parts, so the *PostVehicle* class, which is composed of three other classes: *Engine*, *Wheel* and *Door*[9]. These classes link to the *PostVehicle* class as compositions, with an aggregation relationship. In addition, a post vehicle can be specialized to different classes of vehicles, such as a *Bus*, *Truck* or a *Car*. These classes link to the *PostVehicle* class as generalizations or specializations, with an inheritance relationship. A post vehicle is driven by a driver. More precisely, a particular driver is allowed to drive several post vehicles. This relationship is represented as an association relationship between the *Driver* and *PostVehicle* classes. However, the *Driver* class has two more association relationships with other classes. Firstly, a particular driver can be on at most one leave at a time (for instance on holiday or on a sickness leave), so there is an association relationship between the *Driver* and *Outage* classes. Secondly, an instance of the *Driver* class can perform at most one task at a time, hence the relationship between the *Driver* and *Task* classes. These classes and their relationships are shown on Image 7 using a *UML* class diagram.

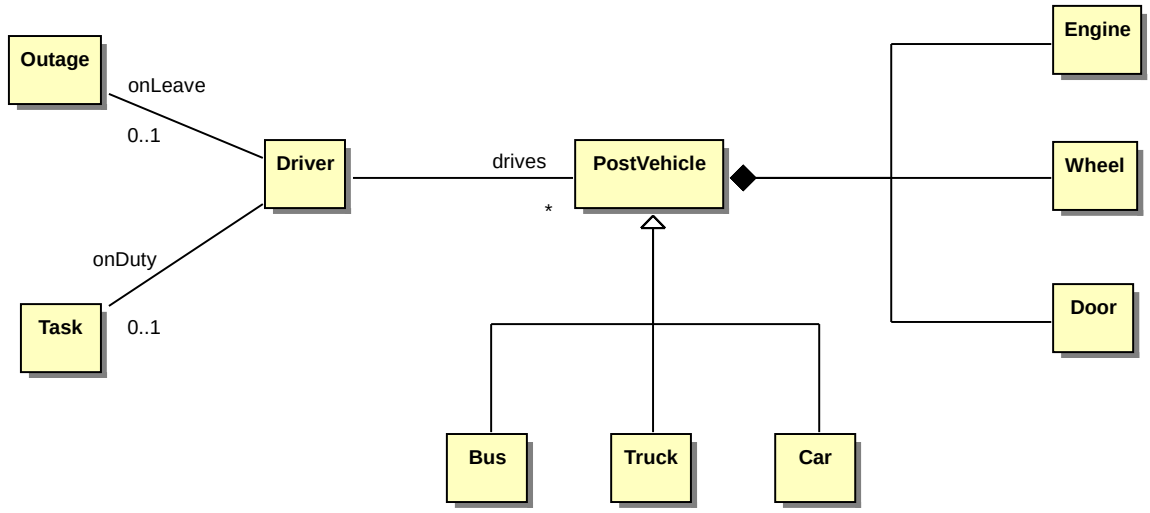

Image 7: Object view of the Mail Delivery System

By comparing the two approaches and the diagrams presented on Image 6 and Image 7, it can be seen that both objects and components are thought of being natural abstractions of real-world entities. Moreover, a real-world entity can be modeled using either notion. What is, then, the vital difference between these abstractions which can help us bring them closer? As it was mentioned earlier in the section "Object versus Components – the Little Details", the answer lies in their *roles*. While objects are best at describing the real-world entities themselves, components are best at describing the services of these entities. That is, objects should describe the problem domain of a system, and components should describe its functionality. This refers to the manner of using services from components and objects. Using a service from a component means to specify the required service and to use any component providing an implementation of that service. Using a service from an object means to specify which object is to be used and to use the service that particular object, with its particular state, can provide. As it was illustrated with the email application earlier, the same can be done with the mail delivery system presented above. Because there are many types of post vehicles, even different types of cars, the system would require, for example, a driver who is able to drive many types of cars, for a specified amount of time, and requires a salary of no more than *x*. In the component-based approach, the identity of this *Driver* component would be of no interest for the project as a whole. However, in the object-oriented

9 Of course, many other parts could be identified, but for the sake of this demonstration, three will suffice.

approach, a whole hierarchy of classes specifying drivers, some of them more skilled than others, would be needed. In this case, the identity of a particular object would determine what services that particular driver can offer for the project.

Using Both Components and Objects

Because components can be considered as service-oriented, describing the functionality of the system, and because objects can be considered as identity-oriented, describing the problem domain of the system, this information can be used to propose a way to develop software products that harness the best from both worlds.

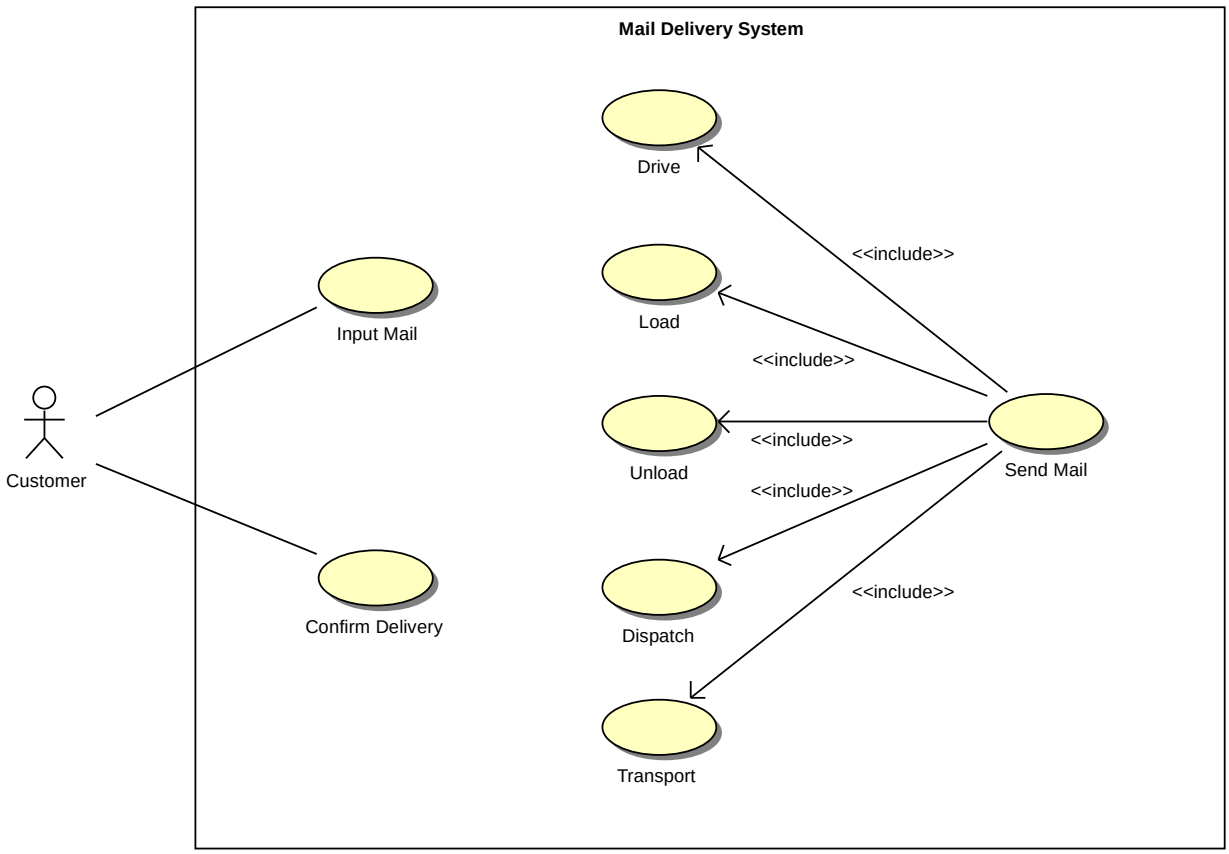

Image 8: Use case diagram of the Mail Delivery System

In the analysis phase of the project life-cycle, the specification of a software system comprises only its required functionality. Therefore, it's not easy, nor natural to define a suitable class decomposition of the problem domain. Instead, defining the services the system should provide would be more reasonable and natural. Because many of these services are usually related to each other, they can be grouped in a convenient way. This way, a set of components can be defined, each of them implementing a group of services. Some of these components need to be constructed, however, other components may already exist, which would mean that they are ready to be reused. By excluding the already existing components, we end up with a set of components to construct for which the services should be well-documented. The design of each component basically consists in determining the services to be used from other components and the services to be implemented by that component. In order to implement the services, a suitable object-oriented decomposition providing the required implementation has to be found. At this stage, the object-orientation is not mandatory, but this approach should be more intuitive and stable, withstanding the

evolution of the software system better. Moreover, finding a suitable class decomposition for a focused, smaller part of the initial system – in this case, a component – is much more easier.

This approach is in accordance with the statements presented earlier. Indeed, components are to be used as software architecture means, while objects are to be used at the implementation level. Of course, for a software project requiring only one or two components, the associated class decompositions can be seen as the architectural skeleton.

Let's demonstrate this with the example shown earlier. Suppose that the final mail delivery system should provide the following services:

- it should allow the customer to input mail into the system and also to receive a confirmation of the mail delivery, if required;
- it should send the mail to the desired destination. This can be performed using other services, like dispatching the mail, loading it into and unloading it from a post vehicle, transporting it, as well as driving this post vehicle.

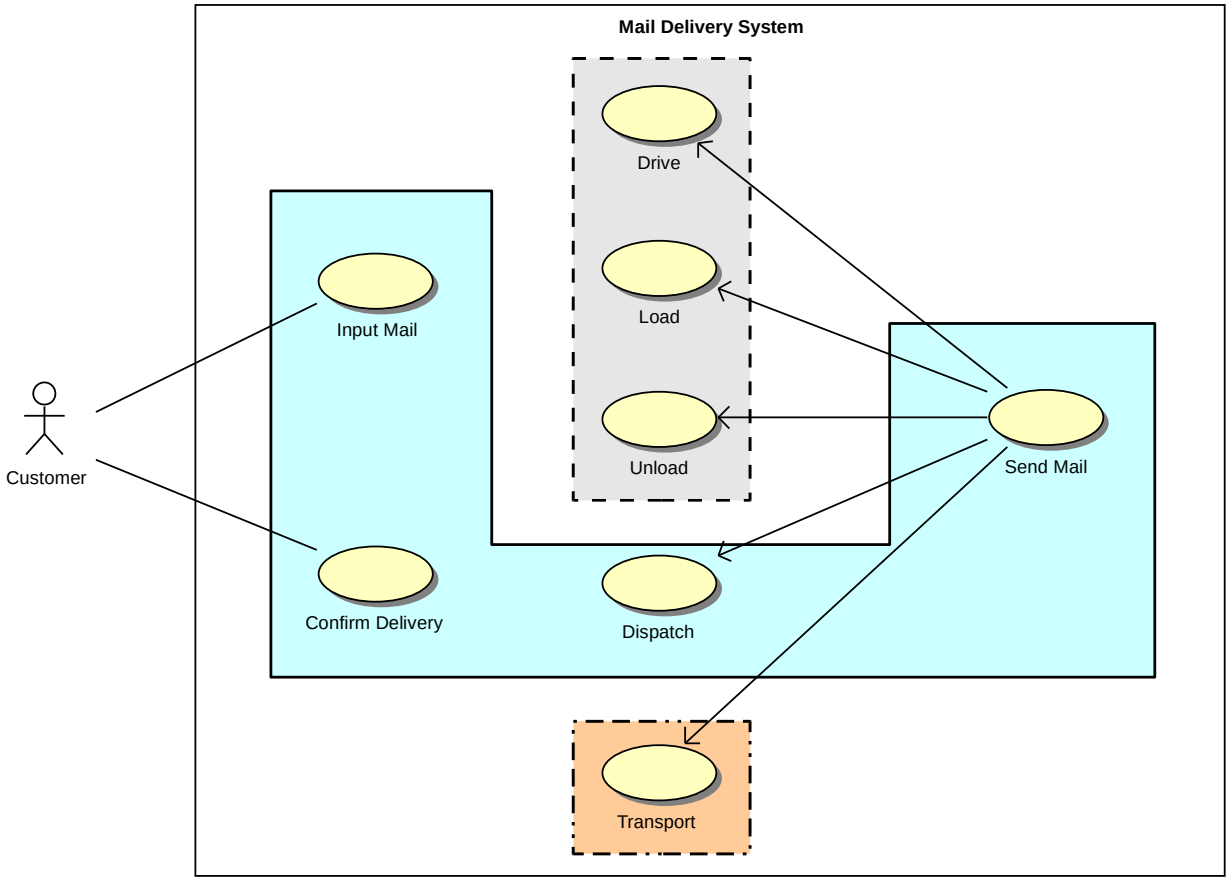

Image 9: Grouping together the component services on the use case diagram of the Mail Delivery System

These services can be represented with a *UML* use case diagram, as shown on Image 8. The actor called *Customer* is connected to use cases *Input Mail* and *Confirm Delivery*. The *Send Mail* use case is internal. This activity includes other use cases, too, in this case: *Drive*, *Load*, *Unload*, *Dispatch* and *Transport*. These use cases are connected to the *Send Mail* use case with *dependency* links, using the <<*include*>> stereotype.

After the services have been identified, the logically related ones can be grouped together, like in Image 9. As it can be seen, a set of three components providing the respective services have been formed, with the *Drive*, *Load* and Unload use cases belonging to the *Driver* component; *Input Mail*, *Confirm Delivery*, *Send Mail* and *Dispatch* use cases belonging to the *MDS* component; and the *Transport* use case belonging to the *PostVehicle* component. Also, they should communicate with each other in the way prescribed by the use case diagram. In fact, based on the diagram shown in Image 9, the system can now be represented with a more complete form of the diagram given in Image 6.

Each component can be further specified and implemented using an internal class decomposition. For example, the *PostVehicle* component can have a structure shown on Image 10. In this case, the *PostVehicle* component offers two services: *Move*, which moves the vehicle and will be used by the *Driver* component, and *Capacity*, which tells the capacity of the post vehicle, and will be used by the *MDS* component.

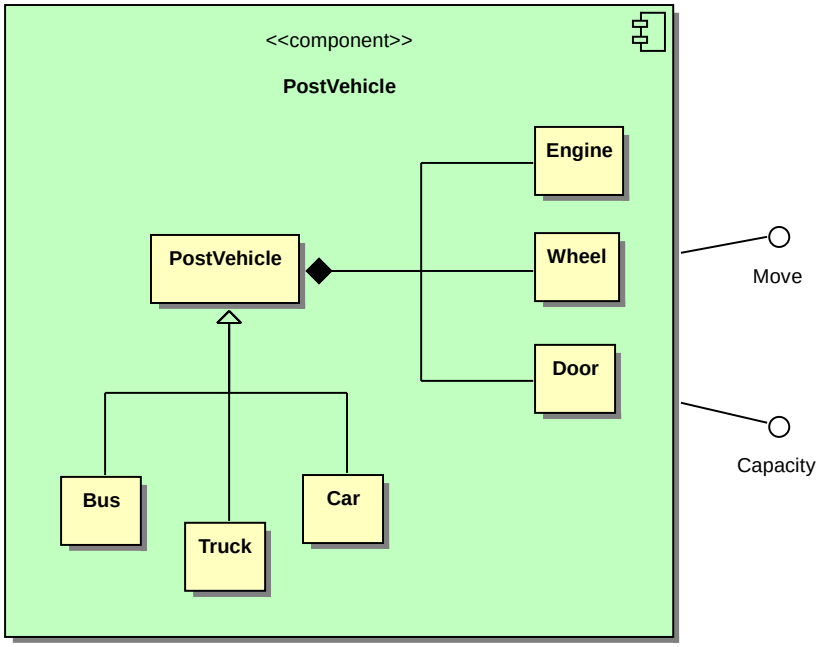

Image 10: The internal class decomposition of the PostVehicle component

The software engineering process described above gives a general guideline for constructing software. Although both objects and components can be used in all phases of a project life-cycle, they have different roles, and because of this, they are more useful in certain aspects of the process than in others. However, by using both concepts during the process as a whole, we can exploit their advantageous features better.

Conclusion

In the brief history of electronic computers and software development, various paradigms of programming appeared and faded. The goal was always the same: utilizing the full potential of the underlying hardware, but – at the same time – easing the development process for humans. This user-friendliness (or "human-friendliness") ranged from "write like humans", implying that the source code should be readable by humans, to "think like humans", meaning that the programming language should think in human-friendly entities, like objects. However, it seams that the goal now is to "do it like humans", implying that once a problem has been solved, it should be solved easily in case that the problem arises again. This is called reuse: why implement parts of an application again, if that was already implemented in an earlier project?

This term paper had two goals: to define components and to try to bridge the gap between objects and components by placing both concepts face to face. However, to properly understand components, it's also important to understand objects. What was the main idea to introduce components in the world of software? Why should people care about them? Answering these questions is so much easier when we know how the current object-oriented software world looks like and what it lacks. Of course, it's easy to say that "components are for composition", but the main moving force towards a change was better reuse, however, why was reuse even mentioned when for objects was also said to be also reusable? As it was explained, objects are usually used to create monolithic applications, which are hard to reuse again.

However, although components were even evangelized as the solution to the software crisis, at the end, the magic didn't happen. Software components are not to be compared to components from other engineering fields. That's the reason that software components struggled ultimately for decades before taking off. Nowadays, they are popular, but the literature about components is very slim compared to literature about objects. To make things even worse, there is no clear understanding about the concept itself, some people calling even objects as components. Two questions arise from these concerns: "Are they really that popular as they seam?" and "Will components rule out objects in the near future?" First of all, there is no doubt that components are here, and their popularity is a reality. However, component markets have a handful of players, from *Microsoft* and *Oracle* to *OMG*. Their realization of components is not yet standardized, with some players obeying their own rules, but their popularity is not questionable. Simple computer users don't even realize that they encounter various forms of components day by day, starting from operating systems and popular applications like the *Microsoft Office* package, to various plug-ins, gadgets and widgets. As for the second question, the statement that component will outcast objects in the future is a possible scenario, with some even saying that *"Object orientation has failed but component software is succeeding"* (Image 11) [Udell 1994], but it's unlikely that something like that will happen. It's clear that components don't need to to be object-oriented in the inside at all, allowing other paradigms like procedural, assembly or functional languages to breath inside a component, but the possibility that components will rule out objects is still low, because true component-based languages have yet to arrive. Moreover, as *Petre* suggested, both components and objects should work together in a project as a whole. Because their roles are different, they can be used in different parts of the project life-cycle, utilizing the best from both worlds.

At the end, one interesting question remains: what will be the next phase in "human-friendliness" in software development? According to some, it could very well be "change it like humans", implying that if a change in the project was elegantly solved, it should also be solved easily when it happens again. This is a real concern: change can occur in all phases in the project, ranging from requirements to the used technology, and there are various theories in software

engineering and development concerning this problem. Managing change was identified as one of the goals of components, but until the present, it was mainly overshadowed by reuse. However, it seams, that people started to realize that software components can be harnessed in a way, that should ease the management of change. Knowing this, it should be clear that components have a bright future ahead.

Image 11: The cover of the famous BYTE magazine (May issue, year 1994), which predicted the death of object-orientation

References

Wang, AJA & Qian, K 2005, Component-Oriented Programming, , John Wiley & Sons, Inc., Hoboken, New Jersey

Szyperski, C, Gruntz, D & Murer, S 2002, Component Software – Beyond Object-Oriented Programming, Second Edition, Addison-Wesley, Harlow, London

Petre, L 2000, 'Components vs. Objects', Technical Report, Turku Centre for Computer Science (TUCS), October, no. 370, viewed 26 February 2010, <http://crest.cs.abo.fi/publications/public/2000/TR370.pdf>

Brown, AW & Short, K 1997, 'On Components and Objects: The Foundations of Component-Based Development', 5th International Symposium on Assessment of Software Tools (SAST '97), pp. 112-121, viewed 19 October 2010, <http://doi.ieeecomputersociety.org/10.1109/AST.1997.599921>

D'Souza, DF & Wills, AC 1998, Objects, Components, and Frameworks with UML: The Catalysis Approach, , Addison Wesley, Reading, Massachusetts

Brown, AW & Wallnau, KC 1998, 'The Current State of CBSE', IEEE Software, vol. 15, no. 5, pp. 37-46, viewed 10 November 2010, <http://doi.ieeecomputersociety.org/10.1109/52.714622>

Teiniker, E 2007, 'CCM Tools', User's Manual, viewed 24 November 2010, <http://ccmtools.sourceforge.net/CcmtoolsManual.pdf>

Tanenbaum, AS 2001, Modern Operating Systems, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey

Cheesman, J & Daniels, J 2001, UML Components: A Simple Process for Specifying Component-Based Software, , Addison-Wesley Professional, Boston

About Us 2010, ComponentSource, viewed 27 November 2010, <http://www.componentsource.com/services/about-us/index.html>

Udell, J 1994, 'ComponentWare', BYTE Magazine, vol. 19, no. 5, pp. 46-56