Oppgavens tekst:

Programming is a very complex task. Many different tools and techniques have been introduced in order to increase quality and reduce complexity of code. Object-oriented technology is very popular at present time, but the progress of software development does not stay on the same place. This exercise is to analyze object-oriented approach and give an overview of new techniques in software development area. One of such technique is a multidimensional separation of concerns, which is central in aspect-oriented and subject-oriented programming approaches.

Oppgaven gitt: 1 september.
Oppgaven levert: 28 november.
Utført ved: NTNU, Gløshaugen, Trondheim
Veileder: Svein Hallsteinsen (SINTEF)
Abstract

Object-oriented programming is the most popular technology in software development today, but it is still in development and there are many open questions. The object-oriented approach uses objects as main abstractions. This helps describe software systems in very natural and intuitive way, but it does not help to make development of software easy. This is because in general it is very difficult to find the right set of abstractions, define relation, and implement them in such a way that system will work.

Multidimensional separations of concerns is a new approach in software development. It concentrates attention on possibility to describe software systems with different set of abstractions, which possibly can overlap each other. This tends to reduce complexity and increase maintainability of software and also has many other positive sides.

The report discusses importance of overlapping abstractions in system descriptions, points some problems of object-oriented programming and gives an overview of new technologies in software development.
Preface.

This is a report written in the course TDT4735 System development, in the final year of study at the Department of Computer and Information Science at the Norwegian University of Science and Technology in Trondheim. The report is written with the intention to cover the area of source code complexity and separation of concerns in software development. The result is an analysis of object-oriented technique and overview of new approaches in software development, such as aspect-oriented and subject-oriented programming. Beside from covering these subjects I have given some comments to them.

The idea for writing about software complexity came as a result of my previously knowledge of the subject and a desire for learning more about it. Software development techniques are an interesting area, and the problem of semantics complexity of programming code remains topical today.

Teaching supervisor for this project has been Svein Hallsteinsen, researcher at SINTEF Tele and Data department in Trondheim. I would like to thank him for all contributions and guidance.

Roman Terekhov

November 23, 2001
Trondheim, Norway.
# Table of contents.

Abstract..................................................................................................................................ii  
Preface. .................................................................................................................................. iii  
Table of contents. .................................................................................................................. iv  
1. Introduction. ................................................................................................................... 1  
   1.1 Document Structure .............................................................................................. 1  
2. Fundamental concepts of software description. ............................................................... 3  
   2.1 Complexity. ........................................................................................................... 3  
   2.2 Discrete systems.................................................................................................... 4  
   2.3 Peoples and information. ...................................................................................... 5  
   2.4 Abstraction............................................................................................................ 5  
   2.5 Hierarchy. ............................................................................................................. 6  
   2.6 Hierarchies and dependencies. ............................................................................ 8  
   2.7 Computer architecture and the world we live in. ............................................... 8  
   2.8 Objects of real world and programming objects..................................................... 9  
   2.9 Classification. ...................................................................................................... 9  
   2.10 Programming languages. ................................................................................... 10  
3. Object-oriented paradigm.............................................................................................. 12  
   3.1 The structure of objects. ..................................................................................... 12  
   3.2 Objects and methods. ........................................................................................... 13  
      3.2.1 Static methods............................................................................................... 14  
      3.2.2 Logic distribution......................................................................................... 14  
   3.3 Abstractions of messages and methods. ................................................................ 15  
      3.3.1 Representation of messages as objects......................................................... 15  
      3.3.2 Write and Read messages. ......................................................................... 16  
   3.4 Interaction between objects. ................................................................................ 17  
      3.4.1 Semantic differences between messages and events................................. 17  
   3.5 Relation between class and object. ...................................................................... 18  
   3.6 Inheritance and Polymorphism. ......................................................................... 19  
   3.7 Reuse in OOP. .................................................................................................... 20  
   3.8 Object-oriented design. ....................................................................................... 20  
4. Multi-dimensional separation of concerns..................................................................... 22  
   4.1 Concerns............................................................................................................... 22  
   4.2 Dimensions of concerns. ..................................................................................... 22  
   4.3 Crosscutting concern problems............................................................................ 23  
   4.4 Concerns in the modern programming languages............................................... 24  
   4.5 Challenges of multi-dimensional separation of concerns.................................... 25  
5. Overview of the new software technologies. ................................................................. 27  
   5.1 Aspect-oriented programming. ............................................................................. 27  
      5.1.1 AOP benefits................................................................................................. 28  
   5.2 Subject-oriented programming. ......................................................................... 28  
      5.2.1 Subject-Oriented Composition.................................................................... 29  
6. Conclusion..................................................................................................................... 31  
Bibliography......................................................................................................................... 32
1. Introduction.

In present-days software systems play very important role, but development of such systems is not an easy task. Writing of large applications is a long and intensive process, with many people involved. Program techniques, used in development, define the way how software systems are constructed as well as how they should be developed. There are many different programming approaches, but today the most popular is the object-oriented approach. Object-oriented approach uses objects as main abstractions. This helps describe software systems in very natural and intuitive way, but it does not help to make developing of software easy. This is because in general it is very difficult to find the right set of abstractions, define relation, and implement them in such a way that system will work.

The most difficult problems of software development are: complexity of program code and reuse of previous written code. Object-oriented approach reduces complexity and increase maintainability of software systems, but it has some restrictions as well. The complexity of program code increase affords need to maintain and change already written systems. It also results in necessity to produce additional descriptions of the code in for of different diagrams and documentations. Difficulties of code reuse result in necessity to write a lot of similar code. The reuse problem increase importance of software design. It changes the design purpose from descriptive to reserving making it more difficult and inflexible.

Multidimensional separations of concerns [9] is a new approach in software development. It concentrates attention on possibility to describe software systems with different set of abstractions, which possibly can overlap each other. This tends to reduce complexity and increase maintainability of software and also has many other positive sides. Multidimensional separation of concerns allows existing of many models of the same system simultaneously. This makes program code more meaningful and easy to maintain, as well as reduce necessity to pre-design software systems.

I believe that the most important property of the approach is possibility to define overlapping abstractions. The rapport discusses importance of overlapping abstractions in system descriptions, points some problems of object-oriented programming and gives an overview of new technologies in software development.

1.1 Document Structure

This report consists of fore six chapters:

Chapter 1: “Introduction”. Introduction to the problems considered in the report and document structure.

Chapter 2: “Fundamental concepts of software description”. Overview of the main concepts used in report. The chapter considers concepts like: abstraction, complexity, hierarchy and so on. The purpose of the chapter is to show importance of overlapping abstractions in knowledge representation.

Chapter 3: “Object-oriented paradigm”. Here I describe some problems of the object-oriented programming (OOP). I am trying to show that OOP is not the last word in the programming technologies and it has many significant disadvantages. The main problems are: complexity of the object-oriented design and inefficient reuse of program code.

Chapter 4: “Multidimensional separation of concerns”. The chapter describes the new technique in software development which is known as “Multi-dimensional separation of
concerns”. This can be considered as answer to the problems of OOP. This technique brings many new capabilities in software development, increase reuse of code and reduce its complexity.

Chapter 5: “Overview of the new software technologies”. The chapter contains description of the most known new technologies based on multi-dimensional separation of concerns. These are: aspect-oriented programming and subject oriented programming. The technologies are under development, but the practical use of them is already possible.

Chapter 6: “Conclusion”. Here is the conclusion of the report and a little look in the future.
2. Fundamental concepts of software description.

This chapter contains description of the main concepts such as: complexity, abstraction, hierarchies and so on. I show the relation between the concepts and the human, and introduce a concept of overlapping abstractions. The important conclusion of the chapter is that the phenomenon of overlapping abstractions is very natural. In application to software development it can reduce complexity of systems description and increase reuse of previously written code.

2.1 Complexity.

Most of software systems are complex and the grade of their complexity differs widely. Because of software complexity it is very difficult, or even impossible, for the individual developer to understand its design in all details. We can say that the software systems complexity exceeds the human intellectual capacity. But the complexity we speak about seems to be an essential property of all large software systems. This means that we can master it, but we can never make it go away.

It could be interesting to study the nature of complexity in order to find some techniques to master it. When we talk about complexity in software development area many different types of complexity are usually mentioned. For example we can talk about complexity of developing process, coordination complexity among developers or complexity that user has to deal with when he begins to use a new software system. We may discuss complexity of software systems according with their size, number of functions it gives or based on technologies that was used in a system. But the complexity that I will discuss in this document can be called program code complexity.

If we look around with hope to find issues of code complexity, we observe that this complexity derives from different aspects of software development. At first, the problems, we are trying to solve in software, often contain complex and sometimes contradictory requirements. The functionality of software systems is difficult enough to implement, but in addition we always have nonfunctional requirements such as usability, performance, cost, and reliability. Combination of functional and nonfunctional requirements increases total complexity of software systems. A further complication is that the requirements of a system often change during its development, largely because the existence of a software development project alters the rules of the problem.

We deal with program code complexity during development of software, but development includes many different processes and therefore I will try to specify exactly when the code complexity appears. It is obvious that we have to work with code in order to feel its complexity. Two different processes may be discovered when we work with program code. We either read or change the code. In order to be able to change the code we have to understand it first. Changing of code without understanding it is not a good thing to do. I don't say that we have to understand the code in all of its details, sometimes it is even impossible, but we need to understand the parts we deal with.

During changing of the program code the problem of its complexity is not so critical, because we already have some kind of knowledge about the system in our mind and we also see the relation between the code and the structure of the system.

The other process is when we read some code and hope to understand it. Now the problem of code complexity is as critical as ever. The situation, when we are able to write some code in
such a way that in the future it is very difficult to understand how it works, is very common. Another example is when one developer needs to change a code written by some other developer.

Often it is not enough to study the source code and we need to use some kind of external information. It may be code comments, tutorials, different diagrams and so on. The external information usually describes the system from different views and shows relation between the views and the code. Such descriptions are called design or architecture and are very popular at present time. UML is an industrial standard modeling language used for describing software design.

While design makes it a little bit easier to understand the code, it is not a part of the code. Because of this we get another problem: to keep design information in synchronization with the code. Many different programmer tools have been created to solve this problem, but using them is difficult and inefficient. Not all aspects of design can be synchronized with code, and also implementing language (technologies) puts some unique restrictions to design. A more natural way to solve the problem is to develop a programming language that will describe programming systems in a way more suitable for humans then for computers.

While existing programming languages uses different approaches to reduce complexity and increase readability of the code, this is not their main goal. The following techniques are often used to increase readability of programs: using of descriptive names for variables and methods, structuring program code, construction of abstractions from problem domain and so on. All these techniques help to manage with code complexity, but they do not solve all the problems. I may say that until now there is no any satisfactory technique that can solve the problems of code complexity.

2.2 Discrete systems.

Because we execute our software on digital computers, we can consider software systems as some kind of discrete systems. Discrete systems by their nature have a finite number of possible states. In large systems, there is a combinatorial explosion that makes the number of states very large. There may be hundreds or even thousands of variables as well as more than one thread of control. The entire collection of these variables, their current values, and the current address and calling stack of each process within the system constitute the present state of the application.

By contrast with analog systems, where small changes in inputs will always cause correspondingly small changes in outputs, discrete systems are more unpredictable. Each event external to such system has the potential of placing the system in a new state and the
mapping from state to state is not always deterministic. In the worst circumstances, an external event may corrupt the state of a system, because its designers failed to take into account certain interactions among events. This is the primary motivation for vigorous testing of software systems. In general exhaustive testing of software is impossible, with exception for the most trivial systems.

Discrete systems can be described as a set of states together with transitions from one state to another. Such systems are always in some state and go from one state to another during its runtime. The system with all possible transitions between all its states may be called as chaos system. The opposite is static system with just one state and no transitions. Chaos and static systems are trivial discrete systems and therefore are not interesting. The interesting ones lie between them. The challenge here is to describe all possible states of a system as well as all possible transitions in such a way that the description is easy to understand and change for peoples.

We do no describe states and transitions one by one because the number of them can be very large and it is impossible to put any meaning in such description. The possibility to put meaning in system description is very important for peoples because of our nature to work with information. We describe software systems with help of discrete abstractions. A discrete abstraction is a group of states and transitions with some meaning. It is much more easy to operate with abstraction then with distinct states and transitions. As result, we have a description of a software system in form of some combination of abstractions.

2.3 Peoples and information.

Peoples cannot work with information in the same way as computers do. The reason for this is the fundamental limitations of the human capacity for dealing with complexity. Experiments by psychologists suggest that the maximum number of chunks of information that an individual can simultaneously comprehend is on the order of seven, plus or minus two. This channel capacity seems to be related to the capacity of short-term memory. In addition to this the processing speed is a limiting factor as well. It takes the mind about five seconds to accept a new chunk of information [9].

We are thus faced with a fundamental dilemma: how people deal with large amount of information together with the limitations they have. The answer to this question is abstraction. We can say that abstraction is a complexity mastering technique. People organize the stimulus input of information simultaneously into several dimensions and successively into a sequence of chunks. In this way we manage to break the informational bottleneck.

The deeply rooted characteristic of the human mind is that we continuously try to organize and explain our impressions of the world around us. It seems to be an interpreter in our brain, which tries to make sense out of all our varied experiences. We say that our mind creates a mental model of the world around us, which we use to explain how the world works and to predict the future. We do not conceive all the information that we get, but abstract from it. Unable to master the entirety of a complex object, we choose to ignore its inessential details, dealing instead with the generalized, idealized model of the object.

2.4 Abstraction.

The word abstraction is used to describe both a process and an entity. Abstraction, as a process, denotes the extracting of the essential details about an item, or a group of items,
while ignoring the inessential details. Abstraction, as an entity, denotes a model, a view, or some other focused representation for knowledge.

People deal with many different types of abstraction. The most fundamental of them are object abstraction and abstraction of process. Abstraction of object is very common, because we live in three-dimensional world with lot of physical objects around us. Really, there are no objects at all, but we (humans) conceive the world via objects. The other fundamental abstraction is a process. Everything is changing continually and the abstraction that we use to describe changing of nature is a process.

Abstractions are building blocks of models. We create a lot of mental models in order to help us understand phenomena of interest and to master them. The choice of phenomena and the questions we ask about them depend on our interests, the modeling paradigms we are comfortable with, and the tools we use for expressing our thoughts.

Abstractions of the same model are distinct from each other. However, the boundaries that distinguish one object from another are often quite fuzzy. For example, look at your leg. Where does your knee begin, and where does it end? In recognizing human speech, how do we know that certain sounds connect to form a word, and are not instead a part of any surrounding words? Consider also the design of a word processing system. Do characters constitute a class, or are whole words a better choice? How do we treat arbitrary, noncontiguous selections of text? Also, what about sentences, paragraphs, or even whole documents: are these classes of objects relevant to our problem?

The property of abstractions, that they are distinct from each other, is often called separation of concerns. Here abstractions of a model play the role of concerns. Separation of concerns is very important property of a model, because otherwise we are not able to understand it. People cannot understand abstractions with vague or overlapped boarders. Boarders of the abstractions in a model have to be precisely specified in order to reason about the model. While peoples often define abstractions, which can overlap each other, they always belong to different models.

A model is created for a purpose and we cannot say whether any model is correct or not, only that it is more or less suited for the study of specific phenomena. In general, a model is never complete. We create models to simplify and generalize, and for this purpose we ignore more than we include. We think in multiple models, always trying to choose the best model for our purpose. Model is an abstraction, but it consists of other abstraction in itself. Thus we tend to think about levels of abstractions or hierarchical models.

Some models can be combined together resulting in more complex models. Combining more and more models together we additionally increase the complexity of the resulting model. Because of humans restrictions in working with information the complexity can overflow our ability to manage it. Therefore it is advised to construct many simple models instead of one big and complex. While there are models that can be combined together, there are also some that can never be combined. Such models often contain overlapping abstractions and therefore the representation of them together is impossible to understand for humans.

### 2.5 Hierarchy

If we consider complex systems, we see a hierarchic nature in them. A system consists of some subsystems, which themselves consists of other subsystems and so on. For example, a personal computer is composed of a CPU, some amount of memory, a monitor, a keyboard, and some sort of secondary storage device. We may take any one of these parts and further
decompose it. A CPU contains cache memory and an arithmetic/logic unit (ALU). Each of these parts may in turn be further decomposed: an ALU may be divided into registers and random control logic, which themselves are constructed from even more primitive elements, such as NAND gates, inverters, and so on [10].

At each level of abstraction, we find a collection of devices that collaborate to provide services to higher layers. We choose a given level of abstraction to suit our particular needs. For instance, if we were trying to track down a timing problem in the primary memory, we might properly look at the gate-level architecture of the computer, but this level of abstraction would be inappropriate if we were trying to find the source of a problem in a spreadsheet application.

We live in three-dimensional world and this gives us a very natural way to represent hierarchies. Actually, the number of dimensions is not so important, but the presence of them is important. We often decompose our systems according to space structure. We call such decomposition physical. There are also other decompositions, which have nothing to do with space. We call them often logical. We can find many logical decompositions of a system, but the physical is only one. Because of this the physical decomposition looks easier and more natural than logical ones. All the decompositions, where abstractions are part of a system, are called structured decompositions.

Most of systems do not embody a single structured hierarchy. Often, we find many different hierarchies present within the same complex system. For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or "part of" hierarchy. Another structural hierarchical decomposition can take place as well. For example we can decompose aircraft in wings, motor, undercarriage and so on. Actually, all these different structural decompositions are models of an aircraft. Every model shows dependencies between components of an aircraft, but from different view.

Alternatively, we can cut across the system in an entirely orthogonal way. For example, a turbofan engine is a specific kind of jet engine, and a Pratt and Whitney TF30 is a specific kind of turbofan engine. Stated in another way, a jet engine represents a generalization of the properties common to every kind of jet engine; a turbofan engine is simply a specialized kind of jet engine, with properties that distinguish it, for example, from ramjet engines. This kind of hierarchy represents an "is a" hierarchy. Our experience shows that we have found it essential to view a system from both perspectives, studying its "is a" hierarchy as well as its "part of hierarchy. Object-oriented approach calls these hierarchies as the class structure and the object structure, respectively.

The abstractions in structured hierarchies are physical parts of a system, while abstractions of "is a" hierarchy are some kind of descriptions or knowledge. The division between physical parts and respectively knowledge parts is very important. This division can be sometimes difficult to see, but it is always possible to find it. Relation between physical object and its description is not constant and can be very complicated in real world.

While construction of hierarchies of abstraction is a natural way to organize abstractions, it is not always the case. Sometimes there are many different alternatives of organizing abstractions in a hierarchy and the choice is not obvious. For physical abstractions of real world hierarchical composition is very natural. Here the choice of decomposition is often based on space dimensions. In this case we can say what is a part and what is a whole. When we decompose a system according with some logical property of its part it is not so easy to say what is part of what. Here many possible hierarchies can be suitable.
Hierarchies of knowledge abstractions are to a higher degree artificial with many possible alternatives. Here the most general (minimal) descriptions are on the top of a hierarchy, while more precise descriptions are under them. The hierarchy of knowledge abstractions is correct when all descriptions of parent abstraction can be applied to its children abstractions. While it is possible to find examples of such situation, it is not a general rule. In general, knowledge abstractions can overlap each other in any possible combinations. Sometime, several hierarchical decompositions can be composed in a graph of abstractions.

2.6 Hierarchies and dependencies.

Hierarchies give us possibility to realize about dependencies between components. For example, if we have some problems with the monitor we do not blame the hard drive disk, because a hard disk is not a part of monitor. We consider hierarchies as a dependency contract between components of a system. If component A is a part of component B we conclude that B may depend on A, but not vice versa. In reality dependencies between components are not hierarchical and therefore a hierarchy is not enough to show all necessary dependencies. We often define explicit relations (other from "part of" and "is a") between components in order to show additional dependencies.

Program code also has hierarchies. In OOP we define classes, which contain methods and data, while they self are located inside packages. Method descriptions contain data and call to other methods. I think that dependencies play the most important role in the problem of understanding program code. The number of explicitly defined relations between software components is very large. Dependencies created by these relations crosscut the hierarchical structure of program code and make it difficult to grasp the meaning of it.

2.7 Computer architecture and the world we live in.

It could be interesting to notice that model of the real world and computer architecture have much in common. We can think about the memory as space and the CPU can represent time. The content of the memory changes during the time and software systems represent some kind of program worlds. A matter of a program world is information and it changes over the time like changes the matter of our world. This likeness can explain why object-oriented approach in programming is so popular.

While the computer architecture can be compared with model of the world, there are many differences as well. The programming world is much simpler then the world we live in. For example, in comparison with our world, the programming world often has no dimensions. Of course we can say that physical memory structure is one-dimensional, but programs rarely focus attention on it. So for most software systems we can find many logical models, while we have only one physical decomposition in the real world. Programming world is discrete, while the real one is not and so on. All these differences have to be taken in consideration during software development.

The strong side of the object-oriented paradigm (OOP) is that we can describe solution of a problem with help of abstractions used in the problem description. OOP gives us possibility to define necessary abstractions and then use them in our systems. All abstractions in OOP are some kind of objects, but the objects are very different from physical ones and we have to keep that in mind.
The programming world is different in the way we can manipulate with it. We have no possibility to change the rules of the real world. We can just study it and hope to find more roles or to specify already discovered ones deeper. The difference is that in the program world we are able to do everything here. We create the programming worlds by defining the roles for it. Actually programs are set of such roles and programmers play role of Gods for their programs. The challenge is to find some way to describe the roles in such a way that we are able to predict the future behavior of the software system we write.

### 2.8 Objects of real world and programming objects.

It can be interesting to compare objects of real world and the ones we use in programming of software systems. If we look around we can see many different objects. These can be table, chair, TV-set and many others. All objects are different, but some of them have common properties. For example table and chair can be of the same color or there can be many different tables in a room. The nature of people is that we interpret the world as set of objects. The first thing we do is trying to pick out an interesting thing by defining of its borders. Then we call it an object and can study it further. The reason of an object abstraction is to set the border and give us possibility to say, whether this is a part of the object or not. It is not always easy to do this (define borders). The ability to see border is based on human’s knowledge, memory and sensibility. This process is complex and I will not try to describe it here. The important thing is that we define borders anyway, without it we are not able to study things. The feature knowledge about the object is kept together with the previously defined borders. Most of the objects around us are three-dimensional and have some form, color, weight and other attributes. These attributes are common for all physical objects. The interesting thing is that all what we say about an object have nothing to do with it. It is just our perception of the object. For physical objects we define some attributes, which are common for them, but for other object the set of attributes can be different.

While studying of some object another object is building in our head. We can call it knowledge object. The knowledge object contains all we know about the object under study. It is very important to see the difference between an object we manipulate and the corresponding knowledge object. We can say that these two objects have a human between them, and he defines their relation.

So we can talk about two different worlds. The first one is the world, which we manipulate, and the other is world of knowledge. The knowledge world lives with its own rules, which are not well understood. But one interesting thing is that our brain is continually trying to reduce number of different objects in knowledge world, by finding similarity between them. This process is independent of study process.

Relations between study objects and knowledge objects are not direct. This means that knowledge about one study object usually contains in many knowledge objects and vice versa.

### 2.9 Classification.

Classification is the means whereby we order knowledge. Recognizing the similarities among things allows us to expose the commonality within key abstractions and mechanisms. Unfortunately, there is no golden path to classification. The discovery of an order is no easy
task. The best software designs look simple, but as experience shows, it takes a lot of hard work to design a simple architecture.

Why is classification so hard? We suggest that there are two important reasons. First, there is no such thing as a "perfect" classification, although certainly some classifications are better than others. There are potentially at least as many ways of dividing up the world into object systems as there are scientists to undertake the task. Any classification is relative to the perspective of the observer doing the classification. For example, the United Kingdom could be seen as an economy by economists, a society by sociologists, a threatened chunk of nature by conservationists, a tourist attraction by some Americans, a military threat by rulers of the Soviet Union, and the green, green grass of home to the more romantic of Britons. Second, intelligent classification requires a tremendous amount of creative insight. This fact recalls the riddle, "Why is a laser beam like a goldfish? Because neither one can whistle". Only a creative mind can find similarities among such otherwise unrelated things.

Classification can be considered as a process of structuring knowledge. We classify abstractions according to some their properties by including them in groups. Groups can be separate from each other or one group can include another. In general, boundaries of one group can cross boundaries of another group. Overlapping classification is very common and this is a reason for "difficult classification". Classification is difficult only when we are trying to find classifying groups that do not overlap.

Actually, classification groups are also abstractions. This means that the difference between our previous definition of abstraction and the notion of classification group is very poor. A classification group is an abstraction for many different abstractions, while under the normal abstraction we understand group of physical objects.

### 2.10 Programming languages.

Most new industrial-strength software systems are larger and more complex than their predecessors were even just a few years ago. The development of more expressive programming languages has complemented these advances. Programming languages are in between the programmer and the computer. They must be unambiguous from the computer side and meaningful from the human side. The first program languages were oriented towards computers. Programs looked as sequence of commands and were very difficult to understand for humans. Then programs became bigger and bigger and high-level languages were introduced. The trend has been a move away from languages that tell the computer what to do (imperative languages) toward languages that describe the key abstractions in the problem domain (declarative languages) [11].

**Declarative languages** describe relationships of software system in terms of functions or inference rules, while the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result. The most common examples of declarative languages are logic programming languages and functional languages.

In declarative modeling, we represent a model not as a series of assignment and control statements, but as a set of facts that are true about the model. The order in which we present the facts is irrelevant, unlike a procedural program. The full set of facts defines a model actually constitute a specification the model.

**Imperative languages** describe computation in terms of a program state and statements that change the program state. In much the same way as the imperative mood in natural languages expresses commands to take action, imperative programs are a sequence of commands for the
computer to perform. The hardware implementation of almost all computers is imperative; nearly all computer hardware is designed to execute machine code, which is native to the computer, written in the imperative style. The program state is defined by the contents of memory, and the statements are instructions in the native machine language of the computer. Higher-level imperative languages use variables and more complex statements, but still follow the same paradigm. Since the basic ideas of imperative programming are both conceptually familiar and directly embodied in the hardware, most computer languages are in the imperative style.

Both declarative and imperative views are important during development of software systems. It is not so easy to convert an imperative description into similar declarative and vice versa, that’s why we humans often switch between these two views in our descriptions. Programming languages have to take care about it and give appropriate support from their side. The proper combination of declarative and imperative descriptions can give impressive results, but the challenge is to hold the descriptions in a consistent state with each other.

The lack of declarative mechanisms in current imperative programming languages results in writing a lot of unit testing code. The unit testing code runs together with compiling developing modules in order to ensure their correctness. This is the work that could be possibly done by more advanced compiler. Writing of such tests is the way in which we actually extend our compiler possibility to detect defects of the system.
3. Object-oriented paradigm.

Object-oriented technology is built upon an object model. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence. By themselves, none of these principles are new. What is important about the object model is that these elements are brought together.

Object-oriented programming (OOP) is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

For a language to support inheritance means that it is possible to express "is a" relationships among types, for example, a red rose is a kind of flower, and a flower is a kind of plant. If a language does not provide direct support for inheritance, then it is not object-oriented but object-based.

The chapter contains description of some problems in OOP. I am trying to show that the OOP is not the last word in software development technologies, but just a unique combination of interesting approaches.

3.1 The structure of objects.

The object-oriented paradigm is based on the concept of objects. An object represents a component of a software system. It consists of some private memory and a set of methods. Methods are used for changing the state of an object and usually to restrict access to object's structure or make it easy to use the object. The memory of an object often has some logic structure, so we can say that object consists of other objects, which further consists of other objects and so on. At the end of this hierarchy are some primitive objects without any structure. So we can talk about two kinds of objects: composite objects (which have some decomposition) and primitive objects (without any structure). This division is not absolute. Sometimes we don't need to know the inner structure of an object in order to use it. For example objects like strings and numbers are often considered as primitive, but object, which for example represents a tree or a car, has some structure.

Division between composite and primitive objects is interesting because sometimes the same object can be considered as compound or as primitive. It depends on how we use it. Some methods of an object return values. These values can be considered as other objects. If usage of an object involves call of such methods (with return values), we are interested in the structure of this object and consider it as compound. If we just involve methods without any output parameters, the structure of is uninteresting. We can go deeper and talk about different types of object structure. I can divide between physical structure and logical structure.

The physic structure of an object is its implemented one. We can find many different implementations for the same object. Every implementation defines some structure and has some methods, which work with this structure. There are methods, which change the physical structure, while some other (which have return value) transform it to logic one. Such methods can be called transforming methods. When we change the physic structure we change transforming methods as well in such a way that logic structure remains the same.

The logical structure is the structure which the programmer can see in the interface of an object. The logical structure consists of return values of methods declared in the interface. In
OOP it is important to define the logical structure for an object as early as possible and hold it unchanged as long as possible, because a change of the logical structure changes interface of the object. The logical structure can put some restrictions on the physical one as well as introduce some additional components, which helps using the object. The logical structure is always static because it is described in the interface of an object, (the interface is written during development time and doesn't change during runtime) but the physical structure can be dynamic. Examples of dynamic physic structure can be containers. Containers often have some dynamic inner structure like linked lists, hash tables or a kind of tree.

Sometimes logic structure is restricted in such a way that we do not see what an object consist of, but nevertheless we have some knowledge about it. An example can be set methods without corresponding get methods. The programmer knows that he can change some inner component by a set method, but there is no way to read it back. Another kind of knowledge could be declaration of order of methods call. There is one or more ways to use an object. For every use situation programmer calls methods in some predefined order. Methods and order of its call give meaning to the object. In OOP you are not able to put some restrictions on order of methods call. The only mechanism we have is writing methods, which first will check state of an object and then do what they have to do. Usually we don't do it, but just write how methods must be called and give no guarantee in other situations. So the compiler cannot check the correctness of order in which methods are called. Today we are not able to put all the information that we have about an object in program code. This information is very important for programmers and helps them to use objects properly. All this information forms some meaning structure of an object. Meaning structure has nothing to do with programs but with peoples. It is our knowledge about objects structure and dependencies between its components.

The object-oriented approach says nothing about object structure. We consider that the structure of an object is always uninteresting for us. The only thing we have to know is the interface. The interface describes some structure, but t is only one side of the object structure.

OOP says that we have to declare all data members as private and change object state just by method call. This is said in order to help to keep the interface of an object unchanged. When physical and logical structures are the same and we don't need to add any restrictions, there is no need to protect the physical one and write methods, that returns the logical one. By writing primitive transformations methods, which just return components of the object's physical structure, we reserve place where we can put some logic if it will be necessary in future. With this trick it will be invisible for outside world whether we have changed the physical structure of an object or put some restrictions to protect our object from inconsistence. So, the object-oriented programs often contain many meaningless methods calls, which are trivial set and get methods. It is not a large problem in developing of ordinary applications but restrict use of OOP in developing of real-time applications. In order to reduce this effect some books advice to declare such data member with package level access modifiers. Other objects of the same class can use these members direct, but consequences are the same. This trick can be considered as some kind of optimization and doesn't make development easier.

### 3.2 Objects and methods.

The object-oriented paradigm says that all methods owns by objects. In a pure object-oriented language there are no methods outside of objects. The problem is that sometimes it is difficult to define an object where the method should be. This problem arises if your method doesn't operate with permanent data or, in the other side, has too many data it operates with. Two
techniques are used to solve this problem. The first one is to put such methods in some class and define them as static methods (methods of a class). The other one is to distribute logic of the method between a few classes.

3.2.1 Static methods.

Static methods often operate just with their own, internal variables and input/output parameters. In addition we can define some static data inside of a class, which can also be used by static methods. Static methods can be used without any relations to objects of the class they are in. Classes play the role of namespaces for such methods. The relation between static methods and object-oriented approach is very poor. The only thing we get from it is the possibility to say that all of our programs consist just of objects.

One interesting thing happens when we implement all data and methods of a class as static. Now we don't need to create an instance of such class, if we do they will all use the same data and there will not be any difference between them. We can relate to a static class as an object created during developing time and having a one to one relationship with its class. If we don't need to have more then one instance of a class we can use this techniques. But there are restrictions as well.

This object is always in memory during runtime and there is no way to delete it. (In Java classes load in the memory when you use them for the first time). The way you refer to such objects is different form the one you use to ordinary objects. So, if in the future you want to change this class to an ordinary one, you will need to rewrite the other classes that use it. Because we never know what will happen in the future, it is not popular to write static classes.

3.2.2 Logic distribution.

The second "technique" is to distribute logic between few classes. You divide one big method into many small ones and put them inside different classes. Small methods call each other in some predefined order. It is often called distribution of responsibilities between objects. Good distribution of logic has to be intuitive and easy to understand. Now we can say that objects are involved in cooperation and talks with some "communication protocol". This is often called object-oriented design. But there are some problems as well.

It is evident that there could be many different ways to "distribute responsibilities between objects". Sometimes it is very difficult to find easy and intuitive "distribution of responsibilities" and often there are many possibilities to choose between. Problems are: find, check, choose, implement and test one of decompositions you have found. After that you depend on your choice and it very difficult to change it. If you want to change your "distribution of responsibilities" you can use refactoring [6] to do it.

Distribution of logic is difficult by it self. It is often easier to describe behavior from an external point of view. For example Petja can describe what Sasha, Roma and Kolja have to do. Sasha, Roma and Kolja don't even need to know what they will do. They just can do some operations and delegate to Petja the right to coordinate them. Sasha, Roma and Kolja don't even need to know about each other. But it not the case if they cooperate without Petja. The conclusion is that OOP learns us to make things in more complex way.

Engineers have known for centuries that the less any one part of a system knows about any other part of that same system, the better the overall system. Systems whose components are
highly independent of each other are easier to fix and enhance than systems where there are strong interdependencies among some or all of the components. Highly independent system components are possible when there is minimal coupling among the components, and each component is highly cohesive.

Distribution of logic increases dependencies between objects and lead to more complex systems, but it is the only way to solve the situation in OOP. The problem can be solved by defining objects that can overlap each other. We can define a new object that will contain previously distributed logic together with its necessary data. Having this ability makes it possible to describe logic in one place in stead of distributing it among many objects.

### 3.3 Abstractions of messages and methods.

OOP says that programs consist of objects and the objects send messages to each other. Object-oriented languages do not implement the "message" abstraction at all, but substitute it with the method abstraction. Messages are actually a signature of methods. When object A sends a message to object B it just call one of the methods in object B. So we can set an equality sign between phrases "sending of a message" and "call of a method". It could be very nice if we were possible to do this, but we cannot. The reason is that a message and a method are too different abstractions to substitute one with the other. Method call is not the same as sending of a message. Abstraction of a method is more complicated than abstraction of a message. First of all methods often take some input parameters and sometimes return some output value. Input parameters of a method could be considered as parameters of a message, but what to do with output value?

Absent of an easy relation between methods and messages become reason for further classification of messages. OOP distinguish between synchronic and asynchronic messages. That an object sends a synchronic message means that the object will wait until message work is finished and receive return value (if there is some). This corresponds to ordinary method call.

Asynchronic message means that object will not wait for result of a message and continue with processing right after message is send. Asynchronic messages are not supported by programming languages. Different techniques are used to simulate this behavior. In Java, for example, you can use threads to implement asynchronic messages. Asynchronic messages cannot return anything. If a method, which corresponds to an asynchronic message, returns some value, we have no chance to catch it in our program. Results of asynchronic messages have to be sent as another message.

So we see that synchronic message actually contain two messages (the second one is returning of value, which can be absent sometimes), while asynchronic correspond to one message. All this is need for the ability to say that objects send messages to each other. Communication of objects with messages looks very nice but actually this does not make programs easily. Messages is the way to bring OOP on marked but not to help programmer write better programs.

#### 3.3.1 Representation of messages as objects.

The interface of an object contains public methods, which can be called from the "outside world". All these methods can be considered as different types of messages, the object can accept. It can be interesting to consider a message as an object. Input parameters of the
methods correspond to data members of an object, and the name of a method is the type of the object. Overloaded and overridden methods can look like a hierarchy with one common super class. When an object receives a message it just receives another object, "message object". This view on messages as on objects lets us see restrictions object-oriented languages.

The number of messages an object can receive is constant and described in an interface to this object. If an object receives many messages, the programmer has to choose between two possibilities. He can implement all messages as different methods or combine some similar messages in bigger ones with some input parameter. An example can be setEnabled(boolean b) with some boolean parameter or two methods setEnabled() and setDisabled(), without any input parameters. It looks very primitive and uninteresting, but the choice has to be declared in the interface of the object. This makes it difficult to change in future. Situations can be more complex than the one which was described.

The types of message objects are also constant. If we want to implement an object, which can receive many different messages and number of this messages is not constant and can depend on state of an object, we have explicitly implement message as en object and put some control logic inside of methods. The problem is that OOP defines the abstraction of a message but programming languages doesn't support this abstraction.

3.3.2 Write and Read messages.

Every object has its state. The state of an object can change during runtime. OOP says that an object can change its state only when it receives messages. Some messages change state of the object while other just returns information about the state. Let us call messages, which change state of an object as write messages and messages which return state as read messages.

Dividing messages into write and read can be very important because you can send many read messages to the same object without introduction collisions, but you can not do the same with write messages. The programmer has to implement some lock system, which will resolve conflicts between incoming write messages or say that simultaneous write messages are forbidden.

Programs often have many objects, but only very few of them have lock mechanisms. This is because of the others do not allow you to send many write messages at the same time. Object-oriented paradigm does not distinguish between write and read messages. Restrictions may stay in API documentation and there is no way to check the consistence of it.

Control of write and read messages could theoretically be used to regulate access to shared resources. If many consumers access a resource, but only one of them is changing it, we need no mechanism to regulate access to this resource. We can consider other situations as repeated read or sequential write/read access to resource as well. These can be considered as different levels of requirements which consumer set to resource. The situation is very similar with the one in database systems. This topic is studied very well there and we can use results of this study in compilers. The only difference that is databases implement dynamic lock systems, which are more advanced. Databases don't know anything about future actions of users and must lock data. Implementing the locking system in compilers is an easier task, because we know the "schedule" (or set of possible "schedules") of all messages during compile time. We can control the possibility of simultaneously write messages and inform programmer about this. This can help to keep the code of the system in a more consistent state.
3.4 Interaction between objects.

Objects have interfaces where all possible type of messages they can receive are declared. This point of view considers an object in a role of a server, which give some services to others. The situation when an object just represent a "server object" is not so usual in object-oriented systems. Programs often have many objects, which communicate with each other by sending messages. While it is very natural to receive a message it is not true in case of sending a message. This is because when object receive a message it does not know whom this message has come from. This absence of knowledge has much more positive then negative, because the object is more independent of outside environment.

An object, which sends messages, has to know whom the message is sent to. If an object cooperates with many other objects (sends messages to them) it has to know about all of them. But the phrase "has to know" also means "depends on". Such an object depends on others it sends messages to. I think that this is the most important problem of OOP. It has to do with problems like: distribution of logic, declaring of interface and finding of stable sets of cooperating objects. Because of this problem, design and refactoring (changing design during development without loss of functionality) of object-oriented software have gotten great attention.

Let us look deeper in the problem. If an object knows nothing about outside world it can be used in any environment. Such object is self-sufficient. This is the most positive situation in context of reuse of the object in other applications. If an object is a part of some cooperation of objects, it now depends on other objects in cooperation (at lest on objects it sends messages to).

Dependencies between objects based on interfaces. In OOP interface is set of all messages an object can receive. We can say that interface is a view from an object to the outside world. An object is saying that it can receive the following messages or offer the following services. If we look from outside the situation is different. In most cases one object uses only a part of interface of another object. If an object is used by many other objects these "parts of interface" are usually overlapping. Possibility to group messages can reduce dependability between objects and make them more meaningful. We can imagine this like the possibility to merge many, more meaningful, interfaces into one. The server object will implement all of these interfaces, while client objects will depend on more specific ones.

Object-oriented methodology doesn't talk about this. The similar concepts are multiple inheritance in C++ and interfaces in Java, but they solve other problems. Multiple inheritance in C++ has much to do with implementing, while I talk just about interfaces of objects. Interfaces in Java have something common with abstract classes and polymorphism. They give functionality of dynamic linking (which is not used so often) and it is not possible to merge interfaces together.

3.4.1 Semantic differences between messages and events.

In OOP the problem of dependences on outside environment is solved with the notification events mechanism. The following scenario take place: an object, which wants to communicate with other objects, implements function of registering event listeners. The event listener is an interface that client object (event listener) must implement. Communication often goes with help of some event object, which are parameter of all methods of event listener.

The event model is quite powerful and flexible. Any number of event listener objects can listen for all kinds of events from any number of event source objects. For example, a
program might create one listener per event source. Or a program might have a single listener for all events from all sources. A program can even have more than one listener for a single kind of event from a single event source.

But this approach has some bad sides as well. First of all this is a very dynamic solution, which tries to think about all possible situations which can happen. This is usually not required, but is done in order to implement the event subsystem only once. Too much dynamism is a very common way to solve reuse problems in object-oriented systems. The other problem we get is when one event listener object tries to implement interfaces containing equally methods definitions. In this situation the programmer have no choice other than having one object as implementation of two such event listener interfaces. The event model solves the problem of the outside world, but in object-oriented way, with as much dynamic as possible.

3.5 **Relation between class and object.**

Classes and objects are separate but very closely connected abstractions. In OOP every object is an instance of some class. Classes are created during development of an application and are static. Objects can be created and deleted during runtime and have a certain state every moment they are “alive”. The notion of class is related to the minimum three following things. Class describes realization/implementation of objects, which can be created with the help of it. It means that class contains all data members and implementation of all methods. Class is used during creation of an object and it contain all necessary information for object creation. Class defines the border an of object’s state. Objects cannot go over these borders during its life.

A class defines the type of an object. OOP theory divides notions of class and type, but realization in programming languages doesn’t do this. This creates some problems for programmers. During design of applications we look on class as the type of an object, but during implementation we have to change our view on class to realization. Here is one of the most trouble areas of OOP.

Type is a very powerful notion. Its purpose is to fill programs with meaning, to make them easier to write and understand. Type is a classification label and has nothing to do with implementation or the inner structure of objects. In the general situation, relation between an object and its type is multiply and dynamic. Type is a combination of properties with some logical value or by other words with some meaning. By reducing the number of properties we can get a type, which contain just one property. So we can say that property is a special case of a type.

An object can have many types (many things it uses for) and it can change its set of types dynamically. For example caterpillar became a butterfly, but it still has the same identity. There is no possibility of dynamic typing in OOP, nor is there a possibility of overlapping types.

In OOP we can not operate with the type of objects in the sense I have describes it, but It gives us possibility to write classes. Class is a very complex abstraction and contains many more properties besides classifying of objects. Problem is that classes in OOP considerably reduce our ability to classify objects and following the ability to fill programs with meaning. Usually an object relates to only one class and it cannot change its class during its life. This is because a class plays the role of an implementing description but not as a classifier.
Multiple inheritance gives the ability to relate to two or more classes with one object, but then our object will contain implementation of all classes. Sometimes it creates problems and there is not any meaningful answer to how to solve them. Therefore multiple inheritance is not recommended to use. In OOP we can create abstract classes or interfaces, without any implementation, and use them as classifiers, but this technique is close related with polymorphism and has some restrictions as well.

Type can be considered as a contract between objects. Type specifies a set of properties and at the same time guarantees the presence of them. On the other side a type can play the role of requirements that one object set for another. Reduction of requirements makes it easier to combine objects and see dependencies between them.

### 3.6 Inheritance and Polymorphism.

Polymorphism is one of the most interesting techniques in programming. Actually it is not native object-oriented mechanism, but OOP puts some unique restrictions on it. Polymorphism can be very primitive considered as choosing of methods to be called. This decision can be made during compiling time as well as during runtime. Hence we can distinguish between static and dynamic polymorphism. In OOP the term polymorphism means dynamic choice of a method to call.

Polymorphism is an interesting mechanism because linking occurs during runtime. In OOP the choice is based on the type of an object. So, we can see that object-oriented polymorphism is based on the typing system. You can define one or more subclasses for one super class and override some methods of the super class in the subclasses. These methods must have the same name and same set of parameters. Now, the compiler cannot define which method to call and the decision is made during runtime, based on the type of an object.

The most significant property of object-oriented polymorphism is that you do not have to recompile existing code. You only need to compile a new subclass that you have written. The problem is that situation when we need this property not happens so often. In most cases we don’t depend on this “most significant” property, because we must have source the code of the super class during compilation anyway. Very often the number of subclasses is finite and known during development time, so we know the number of existing overridden methods to choose between. We could theoretically use this knowledge and the compiler could generate more reliable and optimal code. But compilers for object-oriented languages implement most general situation when set of overloaded methods is open.

Another restriction is that methods must belong to classes, which must be organized in some hierarchy, with the signatures of overridden methods in the super class. If we have some “compatible” methods, which we want to use dynamic polymorphism with, but they have different names or are in classes without a common super class, we miss this possibility. We need to change inheritance relation between classes and, possible, name of the methods, in order to get polymorphism. The opposite situation can be found in C++, where programmer can create pointers to functions. With such pointers you can dynamically choose functions during runtime, without any other requirements except equal parameter sets. But this has nothing to do with OOP.

The object-oriented view on polymorphism is based on inheritance, and hinders its use in other situations. The right way to use polymorphism in OOP is to begin with the definition of a class hierarchy and then override some methods of the super class. If you already have methods you want to call dynamically, so OOP will stay in your way until you structure your
code properly. Such object-oriented requirements to code structure make ad-hoc programming more difficult.

Organizing classes in hierarchies has the goal to put some semantic property on overridden methods. Actually the programmer has to override methods in such a way that they could be substituted in respect to some logical property, but problem is that relation between such properties, in common situation, could not be represented as hierarchy but is “many to many” relation. This problem could be solved with some type of multiply inheritance, but not the one, which OOP give us. Thoughts about such multiple inheritance mechanism, capable of solving described problem, rapidly became very complex, and the possibility of implementing this feature looks problematical.

3.7 Reuse in OOP.

Object-oriented technology has two properties, which make it especially suitable for creating reusable components: inheritance and encapsulation [11].

Reuse based on inheritance. Inheritance and polymorphism permit objects to be defined as being similar to other objects with specified points for modifications and addition. Such reuse solutions are often described as patterns. A pattern describes a general problem and gives directions for its solution. Patterns can be used in many disciplines and for many purposes. We use them to describe and reuse base structures and activities in the areas of system user, system requirements, and system design modelling. Patterns are excellent reusable competence.

Reuse based on encapsulation. Object encapsulation separates the object's externally visible properties from their internal realizations. It enables us to replace one object with another as long as the latter behaves properly, and to bind different objects into a variety of structures. Reuse based on encapsulation is particularly interesting. Programmers focus on the creation of new programs, while the cheapest and safest way to produce new object structures is to create new configurations of objects from existing classes or to copy a validated master structure.

These techniques have some negative properties as well. In order to use inheritance in reuse, we have to construct the appropriate hierarchy of classes. In other words we have to know what kind of reuse we will need in the future. It is always difficult to think ahead and therefore such reuse is very difficult to use in ad hoc programming.

In situations, when one object is used instead of another, the problem is that we cannot use a part of the reused object but have to use it whole. In general, the reused object has many unnecessary properties and much more functionality than we need. The result is a system with a lot of unused code and high requirements of resources.

3.8 Object-oriented design.

Software design is an important activity within the software lifecycle and its benefits are well documented. The reality is that many developers either do not create designs at all, create very minimal, informal design “sketches” that are discarded once system development is underway, or fail to keep their designs up-to-date as requirements and code evolve. At best, this means that developers cannot obtain the benefits of design through the maintenance and evolution phases, which constitute the majority of software system's lifetime. The popularity
of UML might lead to more, and more widely understood, designs being created during the design phase, but creating designs during the initial design phase does not address the issue of keeping designs up-to-date later in the software lifecycle. The main reason for this is that design information is not included in source code.

There are three primary problems underling the inability or disinclination of developers to use object-oriented designs throughout the software lifecycle. First, design models are often large and monolithic. This reduces comprehension, maintainability, and reusability. Further, monolithic designs can inhibit many useful forms of concurrency during design processes. The abstraction units in object-oriented designs—interfaces, classes, and packages—are centralized notions; only one designer at a time can work on a given unit. Centralization means that designers are forced to commit early to the structure and contents of shared design units and concepts, which may overly constrain the set of possible designs too early and may consequently lead to significant impact of change.

Second, we believe that designs are too difficult to reuse. Designs, like code, tend to bundle too many pieces together. Complete classes designed for a particular system are typically too specialized to be of general use. If they really are more generally useful, they often include much more functionality than any given client would use, which decreases comprehensibility and, potentially, usability. Further, effective reuse requires powerful mechanisms for customization and adaptation. The standard object-oriented mechanisms—subclassing, polymorphism, delegation and design patterns—are useful in this context, but not sufficient, particularly because they require a considerable amount of preplanning. Developers may therefore be forced to make invasive, rather than additive, changes to adapt design units, which compromises reuse and future evolution.

Finally, and perhaps most importantly, there is significant structural misalignment [7] between requirements and code, with design caught in the middle. The units of abstraction and decomposition in requirements tend to relate to features and capabilities and other major concepts in the end user domain. New or changed requirements, which cause system evolution, also tend to be structured this way. Object-oriented code, however, focuses on interfaces, classes, and methods. These dramatically different structures mean that traceability between requirements and code is poor. This leads to a host of problems, including impaired comprehension, inability to determine how a change in one artifact affects others, increased complexity of addition, removal, or modification of requirements, and potentially high impact of change—even a relatively small, well-contained change to requirements can affect a large part of the design.

The use of modern object-oriented design languages, including UML, produces designs that align well with object-oriented code, and for good reason. As a result, however, these designs align poorly with requirements, introducing traceability and tangling problems. When a requirement is added or changed, it typically leads to widespread changes across both design models and code. Developers can be forgiven for not incurring the cost of dealing with such changes twice—once in design and once in code—and changing only the code. As long as requirements, design, and code are misaligned, these problems are fundamental.
4. Multi-dimensional separation of concerns.

This chapter introduces “Multi-dimensional separation of concerns” [9]. It is a new and modern programming approach, which can bring a lot of advantages in development of software.

4.1 Concerns.

We can view a complex software system as a combined implementation of multiple concerns. A typical system may consist of several kinds of concerns, including business logic, performance, data persistence, logging and debugging, authentication, security, multithread safety, error checking, and so on.

Concern is at the core of software engineering. In its most general form, it refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. Many different kinds of concerns may be relevant to different developers in different roles, or at different stages of the software lifecycle. For example, the prevalent kind of concern in object-oriented programming is data or class. Each concern in this dimension is a data type defined and encapsulated by a class. Features, like printing, persistence, and display capabilities, are also common concerns, as are aspects, like concurrency control and distribution, roles, viewpoints, variants, and configurations. The set of relevant concerns varies over time and is context-sensitive — different development activities, stages of the software lifecycle, developers, and roles often involve concerns of dramatically different kinds [4].

4.2 Dimensions of concerns.

We can refer to a kind of concern, like class or feature, as a dimension of concern. Separation of concerns involves decomposition of software according to one or more dimensions of concern. Achieving a "clean" separation of concerns has been hypothesized to reduce software complexity and improve comprehensibility; promote traceability within and across artifacts and throughout the software lifecycle; limit the impact of change, facilitating evolution and non-invasive adaptation and customization; facilitate reuse; and simplify component integration.

Separation along one dimension of concern may promote some goals and activities, while impeding others; thus, any criterion for decomposition and integration will be appropriate for some contexts and requirements, but not for all. For example, the data decomposition in object-oriented systems greatly facilitates evolution of data structure details, because they are encapsulated within single (or a few closely related) classes, but it impedes addition or evolution of features, because they typically include methods and instance variables in
multiple classes. Further, multiple dimensions of concern may be relevant simultaneously, and they may overlap and interact, as features and classes do. Thus, modularization according to different dimensions of concern is needed for different purposes: sometimes by class, sometimes by feature, sometimes by viewpoint, aspect, role, or other criterion.

Good system architecture considers present and future requirements to avoid a patchy-looking implementation. However, predicting the future is a difficult task. If you miss future crosscutting requirements, you'll need to change, or possibly reimplement, many parts of the system. On the other hand, focusing too much on low-probability requirements can lead to an overdesigned, confusing, bloated system. Thus a dilemma for system architects: How much design is too much? Should we lean towards underdesign or overdesign?

For example, should an architect include a logging mechanism in a system that does not initially need it? If so, where should the logging points be, and what information should be logged? A similar dilemma occurs for optimization-related requirements – with performance, we seldom know the bottlenecks in advance. The usual approach is to build the system, profile it, and retrofit it with optimization to improve the performance. This approach requires potentially changing many system parts indicated by profiling. Further, over time, new bottlenecks may appear due to changed usage patterns. The reusable library architect's task is even more difficult because he finds it harder to imagine all the usage scenarios for the library.

In summary, the architect seldom knows every possible concern the system may need to address. Even for requirements known beforehand, the specifics needed to create an implementation may not be fully available. Architecture thus faces the under/overdesign dilemma.

### 4.3 Crosscutting concern problems.

Although crosscutting concerns span over many modules, current implementation techniques tend to implement these requirements using one-dimensional methodologies, forcing implementation mapping for the requirements along a single dimension. That single dimension tends to be the core module-level implementation. The remaining requirements are tagged along this dominant dimension. In other words, the requirement space is an n-dimensional space, whereas the implementation space is one-dimensional.

A few symptoms can indicate a problematic implementation of crosscutting concerns using current methodologies. We can broadly classify those symptoms into two categories:

**Code tangling:** Modules in a software system may simultaneously interact with several requirements. For example, oftentimes developers simultaneously think about business logic, performance, synchronization, logging, and security. Such a multitude of requirements results in the simultaneous presence of elements from each concern's implementation, resulting in code tangling.

**Code scattering:** Since crosscutting concerns, by definition, spread over many modules, related implementations also spread over all those modules. For example, in a system using a database, performance concerns may affect all the modules accessing the database.

Combined, code tangling and code scattering affect software design and developments in many ways:
Poor traceability: Simultaneously implementing several concerns obscures the correspondence between a concern and its implementation, resulting in a poor mapping between the two.

Lower productivity: Simultaneous implementation of multiple concerns shifts the developer's focus from the main concern to the peripheral concerns, leading to lower productivity.

Less code reuse: Since, under these circumstances, a module implements multiple concerns, other systems requiring similar functionality may not be able to readily use the module, further lowering productivity.

Poor code quality: Code tangling produces code with hidden problems. Moreover, by targeting too many concerns at once, one or more of those concerns will not receive enough attention.

More difficult evolution: A limited view and constrained resources often produce a design that addresses only current concerns. Addressing future requirements often requires reworking the implementation. Since the implementation is not modularized, that means touching many modules. Modifying each subsystem for such changes can lead to inconsistencies. It also requires considerable testing effort to ensure that such implementation changes have not caused bugs.

4.4 Concerns in the modern programming languages.

Modern languages and methodologies, however, suffer from a problem that has been termed the "tyranny of the dominant decomposition" [9]. They permit the separation and encapsulation of only one kind of concern at a time. Examples of tyrant decompositions are classes (in object-oriented languages), functions (in functional languages), and rules (in rule-based systems). It is, therefore, impossible to encapsulate and manipulate, for example, features in the object-oriented paradigm, or objects in rule-based systems. Thus, it is impossible to obtain the benefits of different decomposition dimensions throughout the software lifecycle. Developers of an artifact are forced to commit to one, dominant dimension early in the development of that artifact, and changing this decision can have catastrophic consequences for the existing artifact. What is more, artifact languages often constrain the choice of dominant dimension (e.g., it must be class in object-oriented software), and different artifacts, such as requirements and design documents, might therefore be forced to use different decompositions, obscuring the relationships between them. The tyranny of the dominant decomposition is the single most significant cause of the failure, to date, to achieve many of the expected benefits of separation of concerns.

Different artifacts for the same software may be written in languages with different "dominant" dimensions, leading to conceptual mismatches between artifacts and to poor traceability, which further complicates evolution. For example, requirements are often specified by function or by feature — this is how the customers who specify the requirements think of the software — while object-oriented designs and code are decomposed using classes. Thus, developers must continuously translate between different expressions of the same concepts across different views. Unless an artifact language specifically supports a given dimension, it is not possible for developers to identify, separate, and encapsulate concerns along that dimension in that particular artifact. And, as we have seen, if some kinds of concerns cannot be identified, encapsulated, and manipulated as first-class entities, the software engineering benefits that they might provide
cannot be achieved. The "tyranny of the dominant decomposition" becomes oppressive whenever the concerns a developer has, at some point during the lifecycle, do not match any of the ones that have been, or can be, encapsulated. Its symptoms are the kinds of scattering, tangling, and cascaded, high-impact changes that this scenario demonstrates.

First, the set of dimensions of concern, and the set of concerns within those dimensions, vary over time. For example, that the design patterns, configurations, and logging dimensions were not relevant to the initial software—they only became relevant with the introduction of new requirements. Second, the fact that each dimension of concern provides only a subset of desirable software engineering benefits means that developers will find different dimensions to be more or less useful, depending on the developer, his role, the stage of development, and the particular goals he wishes to accomplish. Thus, for example, adding the new style-check feature would be simple and additive if the software were decomposed by feature, but because features could not be encapsulated, the feature had to be retrofitted into the object dimension. These concerns do not match—they cut across each other—so the modification is conceptually difficult, invasive, high-impact, and costly. Each of the dimensions of concern has both positive and negative software engineering characteristics; there is no "best" dimension for all purposes, which is one reason why different artifact languages are used for different purposes.

4.5 Challenges of multi-dimensional separation of concerns.

Full support for multi-dimensional separation of concerns opens the door to on-demand remodularization, allowing a developer to choose at any time the best modularization, based on any or all of the concerns, for the development task at hand.

The observations in the previous section lead to some important requirements on separation of concerns mechanisms. We use the term multi-dimensional separation of concerns to denote separation of concerns mechanisms that satisfy these requirements:

It is necessary for developers to be able to identify and encapsulate any kinds, or dimensions, of concern, simultaneously. Further, all dimensions must be created equal—there must not be "tyrant" dimensions that preclude decomposition along other dimensions.

Developers must be able to identify new concerns, and new dimensions of concern, incrementally, at any time during the course of the software life-cycle. For example, it must be possible to identify only some dimensions, or some of the concerns in a given dimension when the dimension is first introduced, and then identify or introduce others as they are needed, without having to rearchitect the software or make invasive modifications.

Developers must not be required to know about, or pay attention to, any concerns, or dimensions of concern, that do not affect their particular activities. One key purpose of separation of concerns is to reduce the amount of complexity a developer must deal with. Forcing all developers to know about all concerns would, instead, increase this complexity.

It must be possible to represent and manage overlapping and interacting concerns. As noted earlier, while independent or "orthogonal" concerns have particularly pleasing properties, overlapping and interacting concerns are at least as common in the real world. In representing such concerns, it must also be possible to identify the points of interaction and maintain the appropriate relationships across these concerns as they evolve.
Separation of concerns is clearly of limited use if the concerns that have been separated cannot be integrated together. Thus, any separation of concerns mechanism must also include powerful integration mechanisms, to permit the integration of separate concerns.

An important additional goal, though not, in our opinion, a defining characteristic of multi-dimensional separation of concerns, is the ability to impose new decompositions on existing software (i.e., decompose it into concerns along a new dimension), without explicit refactoring, reengineering, or other invasive change. We call this capability on-demand remodularization. It allows a developer to choose, at any time, the best modularization for the development task at hand, without perturbing existing ones. In addition to reducing impact of change substantially, this feature opens the door to non-invasive system refactoring and reengineering.

There are potentially many ways to achieve multi-dimensional separation of concerns. There are a variety of modern mechanisms that break the tyranny to at least some extent. The goals listed above are extremely challenging, however, and much research remains, for us and for others, before they are fully achieved.

Multi-dimensional separation of concerns represents a set of very ambitious goals. They apply irrespective of software development language or paradigm. No existing mechanism fully satisfies them, and much research remains to be done in pursuit of these goals. We believe that it is necessary to achieve them in order to overcome the problems associated with the tyranny of the dominant decomposition.
5. Overview of the new software technologies.
The overview is based on the information from the internet. The deeper discussion can be found in [4] and [8]. I tried to take the most important properties of the technologies and the consequences of them.

5.1 Aspect-oriented programming.
AOP strives to cleanly separate concerns to overcome the problems discussed above. AOP lets you implement individual concerns in a loosely coupled fashion, and combine these implementations to form the final system. Indeed, AOP creates systems using loosely coupled, modularized implementations of crosscutting concerns. OOP, in contrast, creates systems using loosely coupled, modularized implementations of common concerns. The modularization unit in AOP is called an aspect, just as a common concern's implementation in OOP is called a class.

AOP involves three distinct development steps. First we decompose the requirements to identify crosscutting and common concerns. You separate module-level concerns from crosscutting system-level concerns. Then we implement each concern separately. At the last step an aspect integrator specifies recomposition rules by creating modularization units – aspects. The recomposition process, also known as weaving or integrating, uses this information to compose the final system.

An AOP implementation can implement the weaver in various ways, including source-to-source translation. Here, you preprocess source code for individual aspects to produce weaved source code. The AOP compiler then feeds this converted code to the base language compiler to produce final executable code. For instance, using this approach, a Java-based AOP implementation would convert individual aspects first into Java source code, and then let the Java compiler convert it into byte code. The same approach can perform weaving at the byte code level; after all, byte code is still a kind of source code. Moreover, the underlying execution system could be aspect aware. Using this approach for Java-based AOP implementation, for example, the virtual machine would load weaving rules first, then apply those rules to subsequently loaded classes. In other words, it could perform just-in-time aspect weaving.

An AOP implementation consists of two parts: a language specification and an implementation. The language specification describes language constructs and syntax. The language implementation verifies the code's correctness according to the language specification and converts it into a form that the target machine can execute.

An AOP implementation can employ another programming methodology as its base methodology, thus keeping the base system's benefits intact. For example, an AOP implementation could choose OOP as the base system to pass on benefits of better implementation of common concerns with OOP. With such an implementation, individual concerns could employ OOP techniques for each identified concern. That is analogous to a procedural language acting as the base language for many OOP languages.

AOP differs most from OOP in the way it addresses crosscutting concerns. With AOP, each concern's implementation remains unaware that other concerns are "aspecting" it. That represents a powerful paradigm shift from OOP and is the most significant property of AOP.
5.1.1 AOP benefits

AOP helps overcome the problems caused by code tangling and code scattering. Here are other specific benefits AOP offers:

Modularized implementation of crosscutting concerns: AOP addresses each concern separately with minimal coupling, resulting in modularized implementations even in the presence of crosscutting concerns. Such an implementation produces a system with less duplicated code. Since each concern's implementation is separate, it also helps reduce code clutter. Further, modularized implementation also results in a system that is easier to understand and maintain.

Easier-to-evolve systems: Since the aspected modules can be unaware of crosscutting concerns, it's easy to add newer functionality by creating new aspects. Further, when you add new modules to a system, the existing aspects crosscut them, helping create a coherent evolution.

Late binding of design decisions: With AOP, an architect can delay making design decisions for future requirements, since he can implement those as separate aspects.

More code reuse: Because AOP implements each aspect as a separate module, each individual module is more loosely coupled. For example, you can use a module interacting with a database in a separate logger aspect with a different logging requirement.

In general, a loosely coupled implementation represents the key to higher code reuse. AOP enables more loosely coupled implementations than OOP.

5.2 Subject-oriented programming.

Subject-oriented programming is a program-composition technology that supports building object-oriented systems as compositions of subjects. It gets its name from the fact that each subject defines a subjective view of objects. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. It is written in a standard object-oriented language. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with "glue" or "adapter" subjects).

When different subjects are composed, the correspondence between the classes and operations in the various subjective views must be specified. Corresponding class definitions are then combined by the system.

The flexibility of subject composition introduces new opportunities for developing and modularizing object-oriented programs. Subject-oriented programming involves determining how to subdivide a system into subjects, and writing the composition rules needed to compose them correctly. It complements object-oriented programming, solving a number of problems that arise when object-oriented technology is used to develop large systems or suites of interoperating or integrated applications.

Subject-oriented programming is an enhancement of object-oriented programming that allows decentralized class definition. An application developer who needs new operations
associated with classes can implement them, not by editing existing code for the classes, but as a separate collection of class definitions called a subject. Multiple subjects can be composed to yield a complete suite of applications; class definitions within the subjects will be combined so as to satisfy the needs of all the applications in the suite. Neither source code access nor recompilation are required to perform this composition, allowing extension and composition of object-code-only applications.

This ability to reconcile different views of objects at composition time means that it is not essential to have advance agreement about the meaning of classes and operations, addressing the first aspect of negotiation mentioned above. The degree of disparity that can be handled effectively is still a matter of investigation. Advance negotiation about shared concepts remains valuable, but is considerably reduced in both quantity and importance. An application developer can write all the code needed for the application, irrespective of which classes are involved, without levying development requirements on others. The cost of this flexibility is a small run-time overhead on operation calls.

Subject-oriented programming supports decentralization in time as well as in space. The developers of an application can program extensions to it as separate subjects to be composed with the base application, perhaps in multiple configurations. This leads to requirement-based development: the code that implements a new requirement is built as a coherent subject rather than being interleaved amongst other application code in a manner that makes it difficult to identify and maintain. Customers would benefit directly from subject-oriented programming, in addition to benefiting indirectly from improved application development and maintenance. A software product bought as a composition of subjects could be reconfigured according to local requirements or preferences, could be extended by composition with other subjects (from the same supplier, different suppliers or written in-house), and could serve as a library of reusable parts. All configuration and composition could be done without access to source code. This technology has the potential to allow customers to compose independently-written applications into integrated suites of applications in ways not preplanned by the authors of the applications, and with much less effort or programming skill than integration endeavors currently require. Further research is required to realize the potential.

### 5.2.1 Subject-Oriented Composition.

Composition is performed on binary code. The source-code for combined subject is not produced as such. Instead, each subject is compiled separately to produce a binary subject. The binary subject consists of a label providing information about it, and binary code produced by the compiler. The subject-oriented compositor uses information in the labels to tie the subjects together. It does not examine or modify the individual subjects' binary code.

This ability to perform composition of binary subjects means that subjects can be sold object-code-only and yet still be composable. The cost is a small run-time overhead on operation calls. Implementers of run-time support for subject-oriented programming can reduce this overhead in various ways, especially for operation calls which, for a particular composition, involve only one subject.

The developer performing a composition of subjects can provide a composition rule specifying details of the composition desired. One option is distribution. The rule can specify whether all subjects are to be combined into a single executable, that executes as a single process, or whether some subjects are to run in separate processes, perhaps on
different machines. The same binary subjects can be used without change in local or distributed compositions. This characteristic of moving the packaging decisions out of the method code is crucial to the reuse of class definitions in different compositions.

Unlike the simple example earlier, the subjects being composed might define a number of different classes, operations and instance variables, some of which might be shared by the subjects and some of which might not. The composition rule must specify how classes, operations and instance variables in different subjects correspond.

Composition rules can specify correspondence in simple, generic terms (such as the default "name-based matching"), or in detail. Detailed specifications of operation correspondence can include transformation of parameters. Detailed specifications of instance variable correspondence identify explicitly which instance variables are shared by multiple subjects. An interesting area of ongoing research is exploring high-level rules that are not fully explicit, with the compositor deducing the details (probably with user assistance obtained interactively).

Support for composition performed just before program execution begins has already been designed, and most key aspects of it have been prototyped. We are also exploring dynamic composition, in which actually running subjects can be composed without interrupting their execution. This would allow running applications to be extended or to begin cooperating with other applications as the user requires. We do not see major obstacles to extending the support to handle dynamic composition.
6. Conclusion.

The conclusion of this report is that object-oriented technology is not the last one in software development. Object-oriented approach reduces complexity and increase maintainability of software systems, but it has some restrictions as well. It has changed the way we develop software systems, but brings both positive and negative side in process of development.

Object-oriented analysis and design are necessary attributes of most development processes. Design of object-oriented programs is not an easy task and therefore usage of collections of good solutions, known as patterns, became very popular. Refactoring is another methodic that helps to struggle with difficulties of object-oriented design. Design is most the important task during object-oriented development. The importance of the design is the consequence of OOP but not a natural process.

The main problem of OOP and previous programming techniques is inability to support overlapping abstractions. Clear separation of concerns has been the main idea of all artificial languages, but now we see that multi-dimensional separation of concerns is more natural way to describe systems. Multi-dimensional separation of concerns can make descriptions more meaningful and easy to maintain, as well as reduce the necessity to pre-design software systems. The new approaches, such as aspect-oriented and subject-oriented programming, have been developed in order to support multi-dimensional separation of concerns. The approaches are compatible with OOP and can be considered as extension of it, but actually, the compatibility is artificial and caused by great amount of existing code and the popularity of OOP.

Multi-dimensional separation of concerns allows the existence of many models of the same system simultaneously. It also changes the process of software development. Now, the programmer first implements or finds implementation of some concern and then integrates it with the rest of the system. Proper integration must not introduce any new dependencies, but just add functionality to the system. This is very well corresponds with development based on use cases and gives more freedom for testing of solutions.

I believe that new programming languages that will support multidimensional separation of concerns will be introduced in future. This methodic need much research and opens many possibilities.
Bibliography.


