1. From structured programming to object-oriented programming

We will assume that the reader of this material has some knowledge of imperative programming, and that the reader already has been exposed to the ideas of structured programming. More specifically, we will assume that the reader has some background in C programming. In Chapter 6 (corresponding to the second lecture of the course) we summarize the relationships between C and C#.

1.1. Structured Programming

Lecture 1 - slide 2

We approach object-oriented programming by reviewing the dominating programming approach prior to object-oriented programming. It is called *structured programming*. A brief background on structured programming, imperative programming, and - more generally - different schools of programming is provided in Focus box 1.1. I will recommend that you read the Wikipedia article about structured programming [wiki-str-pro]. It captures, very nicely, the essence of the ideas.

> *Structured programming* relies on use of high-level control structures instead of low-level jumping
>
> Structured programming is also loosely coupled with *top-down programming* and *program development by stepwise refinement*

Structured programming covers several, loosely coupled ideas. As summarized above, one of these is the use of control structures (such as if, switch/case, while and for) instead of gotos.

Use of relatively small procedures is another idea. A well-structured program should devote a single procedure to the solution of a single problem. The splitting of problems in subproblems should be reflected by breaking down a single procedure into a number of procedures. The idea of *program development by stepwise refinement* [Wirth71] advocates that this is done in a top-down fashion. The items below summarize the way it is done.

- Start by writing the main program
    - Use selective and iterative control structures
    - Postulate and call procedures P1, ...,Pn
- Implement P1, ... Pn, and in turn the procedures they make use of
- Eventually, the procedures become so simple that they can be implemented without introducing additional procedures

Only few programmers are radical with respect to top-down structured programming. In the practical world it is probably much more typical to start somewhere in the middle, and then both work towards the top and towards the bottom.

Imperative programming, Structured programming, and Programming paradigms.

FOCUS BOX 1.1

Imperative programming is one of the four main *programming paradigms*. The others are functional programming, object-oriented programming, and logic programming.

Imperative programming is closely related to the way low-level machine languages work: Commands are used to change the values of locations in the memory of the computer. In high-level languages, this is achieved by use of *assignment statements*, which is used to change the values of variables. The assignment statement is therefore the archetypical command in imperative programming. Control structures (sequence, selection, and iteration) come on top of that together with procedural abstractions.

Programming done in the early years of the computing era (before the introduction of Algol) is often thought of as "unstructured programming". Unstructured programming is largely characterized by use of "jumping around" by means of **goto** commands. The introduction of **if** and **while** control structures together with procedures eliminated the need for gotos. This can be shown theoretically, but - more important - it also holds true in the practical world of imperative programming. Armed with the common control structures (**if** and **while**, for instance) and procedural abstraction, very few programmers are tempted to use a *goto* statement in the programs they write. Such programming, without use of goto statements, is often called *structured programming*.

1.2. A structured program: Hangman

Lecture 1 - slide 3

In order to be concrete we will look at parts of a C program. The program implements a simple and rudimentary version of the well-known Hangman game. We will pretend that the program has been developed according to the structured programming ideas described in Section 1.1.

The main Hangman program, `main`, is shown in Program 1.1. The fragments shown in **purple** are postulated (in the sense discussed in Section 1.1). I.e., they are called, but not yet defined at the calling time. The postulated procedures are meant to be defined later in the program development process. Some of them are shown below.

```
1  int main(void){
2    char *playerName;
3    answer again;
4
5    playerName = getPlayerName();
6    initHangman();
7    do{
8      playHangman(playerName);
9      again = askUser("Do you want to play again");
10   } while (again == yes);
11 }
```

Program 1.1 *The main function of the Hangman program.*

The function `getPlayerName` is intended to prompt the Hangman player for his or her name. As it appears in Program 1.2 this function only uses functions from the C standard library. Therefore there are no emphasized parts in `getPlayerName`.

```
1 char *getPlayerName(){
2   char *playerName = (char*)malloc(NAME_MAX);
3
4   printf("What is your name? ");
5   fgets(playerName, NAME_MAX, stdin);
6   playerName[strlen(playerName)-1] = '\0';
7   return playerName;
8 }
```

Program 1.2 *The function getPlayerName of main.*

The function `initHangman` calls an additional initialization function called `initPuzzles`, which reads a puzzle from a text file. We will here assume that this function does not give rise to additional refinement. We do not show the implementation of `initPuzzles`.

```
1 void initHangman (void){
2   srand(time(NULL));
3   initPuzzles("puzzles.txt");
4 }
```

Program 1.3 *The function initHangman of main.*

`askUser` is a general purpose function, which was called in `main` in Program 1.1. We show it in Program 1.4 (only on web) and we see that it does not rely on additional functions.

The function `playHangman`, seen in Program 1.5, is called by `main` in the outer loop in Program 1.1. `playHangman` contains an inner loop which is related to a single round of playing. As it appears from Program 1.5 `playHangman` calls a lot of additional functions (all emphasized, but not all of them included here).

```
1  void playHangman (char playerName[]){
2    int aPuzzleNumber, wonGame;
3    puzzle secretPuzzle;
4    hangmanGameState gameState;
5    char playersGuess;
6
7    initGame(playerName, &gameState);
8    aPuzzleNumber = rand() % numberOfPuzzles();
9    secretPuzzle = getPuzzle(aPuzzleNumber);
10
11   while ((gameState.numberOfWrongGuesses < N) &&
12          (gameState.numberOfCorrectGuesses < secretPuzzle.numberOfCharsToGuess)){
13     gameStatistics(gameState, secretPuzzle);
14     presentPuzzleOutline(secretPuzzle,gameState);  printf("\n");
15     presentRemainingAlphabet(gameState);  printf("\n");
16     if (CHEATING) presentSecretPuzzle(secretPuzzle);
17     printf("\n");
18     playersGuess = getUsersGuess();
19     clrconsole();
20     updateGameState(&gameState, secretPuzzle, playersGuess);
21   }
22   gameStatistics(gameState, secretPuzzle);
23   wonGame = wonOrLost(gameState,secretPuzzle);
24   handleHighscore(gameState, secretPuzzle, wonGame);
25 }
```

Program 1.5 *The function playHangman of main.*

In Program 1.6 (only on web) and Program 1.7 (only on web), we show two additional functions, `initGame` and `getPuzzle`, both of which are called in `playHangman` in Program 1.5.

As already brought up in Section 1.1 many programmers do not strictly adhere to structured programming and top-down refinement when coding the hangman program. If *you* have programmed Hangman, or a similar game, it is an interesting exercise to reflect a little on the actual approach that was taken during your own development. In Section 4.1 we return to the Hangman example, restructured as an object-oriented program.

Exercise 1.1. *How did you program the Hangman game?*

This is an exercise for students who have a practical experience with the development of the Hangman program, or a similar game.

Recall how you carried out the development of the program.

To which degree did you adhere to *top-down development by stepwise refinement*?

If you did not use this development approach, then please try to characterize how you actually did it.

1.3. Observations about Structured Programming

Lecture 1 - slide 4

We will now attempt to summarize some of the weaknesses of structured programming. This will lead us towards object-oriented programming.

Structured programming is not *the wrong way* to write programs. Similarly, object-oriented programming is not necessarily *the right way*. Object-oriented programming (OOP) is an alternative program development technique that often tends to be better if we deal with large programs and if we care about program reusability.

We make the following observations about structured programming:

- Structured programming is narrowly oriented towards solving one particular problem
  - It would be nice if our programming efforts could be oriented more broadly
- Structured programming is carried out by gradual decomposition of the functionality
  - The structures formed by functionality/actions/control are *not* the most *stable* parts of a program
  - Focusing on data structures instead of control structure is an alternative approach
- Real systems have no single top - Real systems may have multiple tops [Bertrand Meyer]
  - It may therefore be natural to consider alternatives to the top-down approach

Let us briefly comment on each of the observations.

When we write a 'traditional' structured program it is most often the case that we have a single application in mind. This may also be the case when we write an object-oriented program. But with object-oriented programming it is more common - side by side with the development of the application - also to focus on development of program pieces that can be used and reused in different contexts.

The next observation deals with 'stable structures'. What is most stable: the overall control structure of the program, or the overall data structure of the program? The former relates to use of various control structures and to the flow procedure calls. The latter relates to data types and classes (in the sense to be discussed in Chapter 11). It is often argued that the overall program data structure changes less frequently than the overall program control structure. Therefore, it is probably better to base the program structure on decomposition of data types than on procedural decomposition.

The last observation is due to Bertrand Meyer [Meyer88]. He claims that "Real systems have no top". Let us take the Hangman program as an example. Even if it is likely that we can identify a single top of most hangman programs (in our program, `main` of Program 1.1) the major parts of the program should be able to survive in similar games, for instance in "Wheel of Fortune". In addition, a high score facility of Hangman should be applicable in a broad range of games. The high score part of the Hangman program may easily account for half of the total number of source lines in Hangman, and therefore it is attractive to reuse it in other similar games. The simple textual, line-oriented user interface could be replaceable by a more flexible user graphical user interface. In that way, even the simple Hangman program can easily be seen as a program with no top, or a program with multiple tops.

Readers interested in a good and extended discussion of 'the road to object-orientation' should read selected parts of Bertrand Meyers book 'Object-oriented Software Construction' [Meyer88]. The book illustrates object-oriented programming using the programming language Eiffel, and as such it is not directly applicable to the project of this course. The book is available in two versions. Either of them can be used. In my opinion 'Object-oriented Software Construction' is one of the best books about object-oriented programming.

1.4. Towards Object-oriented Programming

Lecture 1 - slide 5

We are now turning our interests towards 'the object-oriented way'. Below we list some of the most important ideas that we must care about when we make the transition from structured programming to object-oriented programming. This discussion is, in several ways, continued in Chapter 2.

- The gap between the problem and the level of the machine:
  - Fill the gap bottom up
- Use the data as the basic building blocks
  - Data, and relations between data, are *more stable* than the actions on data
- Bundle data with their natural operations
  - Build on the ideas of *abstract datatypes*
  - Consolidate the programming constructs that encapsulate data (structs/records)
- Concentrate on the *concepts and phenomena* which should be handled by the program
  - Make use of existing theories of phenomena and concepts
  - Form new concepts from existing concepts
- Make use of a programming style that allows us to collapse the programming of objects

Our approach to object-oriented programming is continued in Chapter 2. Before that, we will clarify the concept of abstract data types.

1.5. Abstract Datatypes

Lecture 1 - slide 10

A *data type* (or, for short, a *type*) is a set of values. All the values in a type share a number of properties. An *abstract data type* is a data type where we focus on the possible operations on the values in the type, in contrast to the representation of these values. This leads to the following definitions.

> A *datatype* is a set of values with common properties. A datatype is a classification of data that reflects the intended use of the data in a program.
>
> An *abstract datatype* is a data type together with a set of operations on the values of the type. The operations hide and protect the actual representation of the data.

In this material, boxes on a dark blue background with white letters are intended to give precise definitions of concepts.

To strengthen our understanding of abstract data types (ADTs) we will show a few *specifications* of well-known data types: Stacks, natural numbers, and booleans. A specification answers "what questions", not "how questions". The details are only shown in the web version of the material.

1.6. References

[Meyer88]	Bertrand Meyer, *Object-oriented software construction*. Prentice Hall, 1988.
[Wirth71]	Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227.
[Wiki-str-pro]	Wikipedia: Structured_programming http://en.wikipedia.org/wiki/Structured_programming

2. Towards Object-oriented Programming

In this and the following chapter we will gradually unveil important theoretical and conceptual aspects of object-oriented programming. After this, in Chapter 4 we will be more concrete and practical, again in terms of the Hangman example.

In this chapter we will deal with a number of different aspects that lead in the direction of object-oriented programming. We do not attempt to relate these aspects to each other. Thus, in this chapter you will encounter a number of fragmented observations that - both individually and together - bring us towards object-oriented programming.

2.1. Client, Servers, and Messages

Lecture 1 - slide 7

We will start with message passing in between objects. One object (often called the *client*) sends a message to another object (often called the *server*). The client asks for a service. The server will do the job, and eventually return an answer to the client.

"Client" and "server" are general role names of objects. When the server receives a message, it may decide to forward the message to some subserver (because it cannot handle the request - solve the problem - itself). In this way, the server becomes a client of another server.

We will primarily be concerned with message passing where the client waits for an answer from the server. Thus, nothing happens in the client before the server has completed its work. This is referred to as *synchronous message passing*. *Asynchronous message passing* is also possible. This involves parallel activities. This is a slightly more advanced topic.

Peter orders a Pizza at AAU Pizza by email. Via interaction between a number of service providers, a pizza is delivered to Peters group room

Below we study an everyday example of message passing between an object (person) who orders a pizza, and a "Pizza server". The Pizza server relies on other subservers (subcontractors), in our example the butcher, the greengrocer, and a transport service. Thus, the Pizza crew are customers in other shops, and they make use of other services.

Notice that Peter - the hungry guy - is not aware of the subcontractors. Peter only cares about the interface of the Pizza server.

In some versions of this material you may interactively play the Pizza scenario in order to find out how the objects cooperate when Peter orders a Pizza. The scenario emphasizes that there is always a single current object (at least as long as we deal with synchronous message passing).

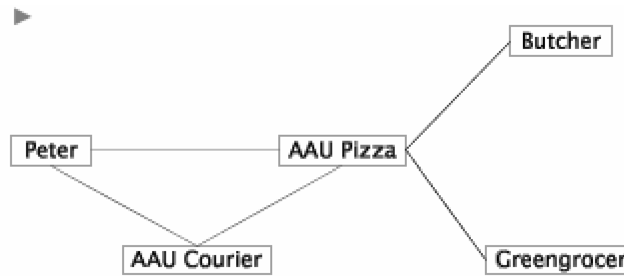

Figure 2.1 *The scenario of pizza ordering. The scenario focuses on a number of objects (persons) who communicate by message passing.*

Is it reasonable that Peter is idle in the period of time in between pizza ordering and pizza delivery? It depends on the circumstances. If you wait in the restaurant you may actually be left with feeling of 'just waiting'. If Peter orders the Pizza from his group room, Peter probably prefers to send an asynchronous message. In that way he can do some work before the pizza arrives. In this mode, we should, however, be able to handle the *interrupt* in terms of the actual pizza delivery. Again, this is a more advanced topic.

A ***client*** asks for a service at some given service provider (***server***).

This may lead the service provider (which now plays a client role) to ask for subservices

Clients and servers communicate by ***passing messages*** that return results

Try the accompanying SVG animation

In our model of message passing, it is inherent that messages return a result. Alternatively, we could use a model in which the 'the message result' is handled by a message in the other direction. We have chosen a model, which can be used directly in mainstream object-oriented programming languages (such as C#).

We will come back to clients and servers in the context of the lecture about classes, see Section 10.2. Message passing is taken up in that lecture, see Section 10.3.

2.2. Responsibilities

Lecture 1 - slide 8

Responsibility - and distribution of responsibility - is important in a network of cooperating objects. In Section 2.1 we studied a network of people and pizza makers. The Pizza maker has certain assumptions about orders from customers. We cannot expect the pizza maker to respond to an order where the customers want to buy a car, or a pet. On the other hand, the customer will be unhappy if he or she receives spaghetti (or a chocolate bar) after having ordered a pizza calzone.

Objects that act as servers manage a certain amount of responsibility

We will talk about the *responsibility of an object* as such. The object is responsible to keep the data, which it encapsulates, in good shape. It should not be possible to bring the object in an inconsistent state.

The *responsibility of an operation* of a class/object does also make good sense. If the sender of the message, which activates an operation fulfills certain (pre)conditions, it is the obligation of the operation to deliver a result which comply with a certain (post)condition.

The responsibilities of an object, together with the responsibilities of the operations of the object, sharpen the profile of the object, and they provide for a higher degree of cohesion of the object.

- Responsibility
  - Of an object, as reflected by the interface it provides to other objects
  - Of an operation
    - Precondition for activation - proposition about prerequisites for calling
    - Postcondition - proposition about result or effects
  - Well-defined responsibilities provide for coherent objects

In Chapter 49 through Chapter 53 we will devote an entire lecture to discussion of responsibilities, and how to specify the distribution of responsibilities among objects. This will involve *contracts*, which (again) is a real-world concept - a metaphor - from which can we can gain useful inspiration when we develop software.

You should care about the responsibilities of both objects and operations

The *distribution of responsibilities* will become a major theme later in the course

2.3. Data-centered modularity

Lecture 1 - slide 9

Message passing is mainly a dynamic (run-time) aspect of object-oriented programs. Let us now focus on a static aspect: modularity.

Modularity is the property of a computer program that measures the extent to which it has been composed out of separate parts called modules [Wikipedia]

Non-modular programs (programs written without decomposition) are unwieldy. The question we care about here is the kind of modularity to use together with abstract data types. We will identify the following kinds of modularity:

- **Procedural modularity**
  - Made up of individual procedures or functions
  - Relatively fine grained
  - Not sufficient for programming in the large
- **Boxing modularity**
  - A wall around arbitrary definitions
  - As coarse grained as needed
  - Visibility may be controlled - import and export
- **Data-centered modularity**
  - A module built around data that represents a single concept

- High degree of cohesion
- Visibility may be controlled
- The module may act as a datatype

Procedural modularity is used in structured programming, e.g. in C programs. It covers both functions and procedures. Procedural modularity remains to be very important, independent of programming paradigm!

Boxing modularity (our name of the concept) captures the module concept known from, e.g. Ada [Ada80] and Modula-2 [Wirth83]. In C, there are only few means to deal with boxing modularity. Most C programmers use the source files for boxing.

Boxing modularity allows us to box a data type and the operations that belong to the type in a module. When using data centered modularity *the module becomes a type itself*. This is an important observation. Object-oriented programming is based on data centered modularity.

Object-oriented programming is based on data-centered modularity

2.4. Reusability

Lecture 1 - slide 11

Let us now, for a moment, discuss reusability. The idea is that we wish to promote a programming style that allows us to use pieces of programs that we, or others, have already written, tested, and documented. Procedure libraries are well-known. Object-oriented programming brings us one step further, in the direction of class libraries. Class libraries can - to some degree - be thought of as reusable abstract data types.

More reuse - Less software to manage

We identity the following reusability challenges:

- Find
  - Where is the component, and how do I get it?
- Understand
  - What does the component offer, and how does it fit with my own program?
- Modify
  - Do I need to adapt the component in order to (re)use it?
- Integrate
  - How do I actually organize and use the component together with the existing components?

Finding has been eased a lot the last decade, due to the emergence of powerful search machines (servers!). Understanding is still a solid challenge. Documentation of reusable parts is important. Tools like JavaDoc (developed as on-line resources by Sun as part of the Java effort) are crucial. We will study interface documentation of class libraries later in this material. Modification should be used with great care. It is not a good idea to find a procedure or a class on the Internet, and rewrite it to fit your own needs. When the next

version of the program is released you will be in great trouble. A modular modification approach, which separates your contributions from the original contributions, is needed. In object-oriented programming, inheritance alleviates this problem. The actual integration is relatively well-supported in modern object-oriented programming languages, because in these languages we have powerful means to deal with conflicts (such as name clashes) in between reused components and our own parts of the program.

2.5. Action on objects

Lecture 1 - slide 12

The final aspect that we want to bring up in our road towards object-oriented programming is the idea of action on objects. Actions should always be targeted at some object. Actions should not appear 'just up in the air'. Bertrand Meyer [Meyer88] has most likely been inspired by a famous John F. Kennedy quote when he formulated the idea in the following way:

> *Ask not what the system does: Ask what it does it to!*
> [Bertrand Meyer]

- Actions in general
  - Implemented by procedure calls
  - Often, but not always, with parameters
- Actions on objects
  - Activated via messages
    - A message always has a receiving object
    - A message is similar to a procedure calls with at least one actual parameter
  - A message activates an operation (a method)
    - The receiving object locates the best suited operation as responder (method lookup)

The activation of a concrete procedure or function is typically more complex than in ordinary imperative programming. The message is sent to an object. The reception of the message may cause the object to search for the best suited operation (method) to handle the request by the message. This process is sometimes called *method lookup*. In some object-oriented language the method lookup process is rather complicated.

In the next section we continue our road towards object-oriented programming, by discussing concepts and phenomena.

2.6. References

[Meyer88] Bertrand Meyer, *Object-oriented software construction*. Prentice Hall, 1988.

[Wirth83] Wirth, N., *Programming in Modula-2*, third. Springer-Verlag, 1985.

[Ada80] *Ada Reference Manual*. United States Department of Defence, July 1980.

3. Phenomena and Concepts

Metaphors from the real life are important inspiration when we program the computer. It is limiting - and in fact counterproductive - to focus only on the technical computer concepts (bits, bytes, CPUs, memory words, USB ports, etc). According to my favorite dictionary (the American Heritage Dictionary of the English Language) a metaphor is

> "a figure of speech in which a word or phrase that ordinarily designates one thing is used to designate another, thus making an implicit comparison."

Many familiar programming concepts are not from the technical world of computers. Quite a few, such as int, float, and double come directly from mathematical counterparts. Messages and message passing, which we discussed in Section 2.1, are widely known from our everyday life. Even before email was invented, people communicated by means of messages (Morse codes, telegrams, postal mail letters).

It turns out that the ideas of classes and objects are - in part - inspired from the theory of concept and phenomena. We will unveil this in the following sections. Be warned that our coverage is brief and dense. It may very well be the case that it takes time for you to digest some of the ideas and concept that we are going to present.

3.1. Phenomena and Concepts

Lecture 1 - slide 14

A *phenomenon* is a thing that has definite, individual existence in reality or in the mind. Anything real in itself.

A *concept* is a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection

The definitions of phenomenon and concept are taken from the PhD thesis of Jørgen Lindskov Knudsen and Kristine Stougaard Thomsen, Aarhus University [jlk-kst]. This thesis is also the source behind Section 3.2 - Section 3.4.

The characteristic aspects of a concept are the following:

- **The concept name**
- **The intension:** The collection of properties that characterize the phenomena in the extension of the concept
- **The extension:** The collection of phenomena that is covered by the concept

The name of the concept is also called the *designation*. The designation may cover a number of different names, under which the concept is known.

The word *intension* is used in the less well-known meaning (from logic) "the sum of the attributes contained in a term" (see for instance the American Heritage Dictionary of the English Language).

The word *extension* is used in the meaning (again from logic): "The class of objects designated by a specific term or concept" (according the same dictionary). Be careful not to confuse this meaning of *extension* with the more common meaning of the word, used for instance in Chapter 26 for extension of classes.

Concepts can be viewed in two different ways: The Aristotelian and the fuzzy way.

Using the *Aristotelian view*, the properties in the intension are divided into *defining properties* and *characteristic properties*. Each phenomenon of a concept must possess the defining properties. It is assumed that it is objectively determinable if a given phenomenon belongs to an Aristotelian concept.

Using the *fuzzy view*, the properties in the intension are only examples of possible properties. In addition to the example properties, the intension is also characterized by a set of prototypical phenomena. It is not objectively determinable if a given phenomenon belongs to a fuzzy concept.

We will primarily make use of the Aristotelian view on concepts. The relative sharp borderline between different concepts is attractive when we use concepts as the basis for the classes that we program in an object-oriented programming language. Many such classes represent *real-life concepts*, simply because many of our programs administrate things from the real world. It is, however, also common to make use of *imaginary concepts*, which have no real-life counterparts (such, as for instance, a hashtable).

3.2. Classification and exemplification

Lecture 1 - slide 15

To *classify* is to form a concept that covers a collection of similar phenomena.

To *exemplify* is to focus on a phenomenon in the extension of the concept

Classification and exemplification describe a relation between concepts and phenomena.

Classification forms a concept from a set of phenomena. The *intension* of the concept is the (defining) properties that are shared by the set of phenomena (according to the Aristotelian view).

Exemplification is the inverse of classification. Thus, the exemplification of a concept is a subset of the extension of the concept.

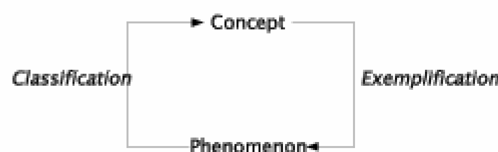

Figure 3.1 *The relationships between concepts and phenomena. Given a concept we can identify the examples of phenomena in the extension of the concept. Given such an example, we can (the other way around) find the concept that classifies the sample phenomenon.*

3.3. Aggregation and Decomposition

Lecture 1 - slide 16

In this and the following section we will see ways to form new concepts from existing concepts. First, we look at concepts related to 'parts' and 'wholes'.

> To *aggregate* is to form a concept that covers a number of parts
>
> To *decompose* is to split a concept into a number of parts

The concept of a house is an aggregation of (for instance) of the concepts wall, window, door, and roof. The latter are the decomposition of the house concept.

The intension of the aggregated concept corresponds to the intensions of the part concepts. But, in some cases, *the whole is more than the sum of its parts*. Thus, the aggregated concept may have additional properties as well.

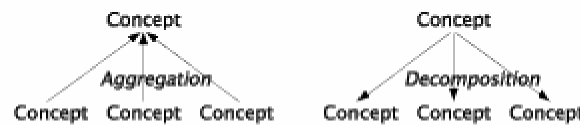

Figure 3.2 *An illustration of aggregation and decomposition. Notice that the relations between wholes and parts are in between concepts. Thus, aggregation and decomposition show how to form new concepts from existing concepts.*

In Figure 3.3 we show an example, namely the aggregation of a bike. Notice that we do not address the number of parts of the aggregated concept (no cardinalities). Following the tradition of UML notation, we use a diamond shape next to the aggregated concept. Notice, however, that it is not our intention to use exact UML notation in this material. We will primarily be concerned with programming notation, as defined (precisely) by a programming language.

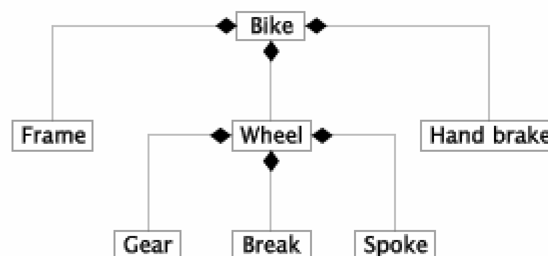

Figure 3.3 *An aggregation of a Bike in terms of Frame, Wheel, Brake, etc. This illustration does not capture the number of involved parts. Thus, the diagram does not capture the number of spokes per wheel, and the number of wheels per bike. The diamond shape is UML notation for aggregation.*

Exercise 1.2. *Aggregated Concepts*

Take a look at the concepts which are represented by the phenomena in the room where you are located. Identify at least four aggregated concepts. Enumerate the concepts of the decomposition.

3.4. Generalization and Specialization

Lecture 1 - slide 18

Generalization forms a broader concept from a narrow concept

Specialization forms a narrow concept from a broader concept

Generalization and specialization are seen as ways to form a new concept from an existing concept. The extension of a specialization S is a subset of the extension of the generalization G.

It is more difficult to capture specialization and generalization in terms of the intensions.

The concepts of Encyclopedia, Bible, and Dictionary are all specializations of the Book concept. Encyclopedia, Bibles and Dictionaries are all subsets of Books. It may be the case that the set of encyclopedia and the set of dictionaries are overlapping.

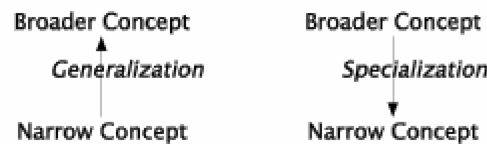

Figure 3.4 *An illustration of generalization and specialization.*

Below, in Figure 3.5 we show a generalization/specialization hierarchy of transportation concepts. Each parent in the tree is a generalization of its sons.

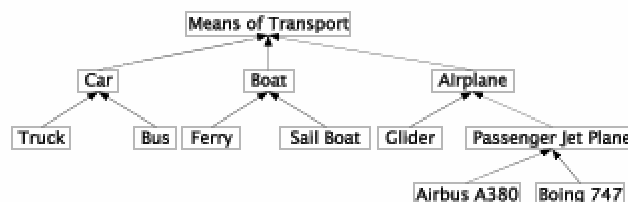

Figure 3.5 *A generalization/specialization hierarchy of 'Means of Transport'. All the concepts in this diagram are specialized means of transport. Notice that all the nodes in the specialization trees are concepts - not individual phenomena.*

The ideas of generalization and specialization among concepts are directly reflected in generalization and specialization among classes (see Chapter 25) as supported by inheritance in object-oriented programming languages.

Exercise 1.3. *Concepts and Phenomena*

The purpose of this exercise is to train your abilities to distinguish between concepts and phenomena.

Decide in each of the following cases if the mentioned item is a concept or a phenomena:

1. The door used to enter this room.
2. Todays issue of your favorite newspaper.
3. Your copy of today's issue of your favorite newspaper.
4. The collection of all copies of today's newpapers
5. Denmark.
6. European country.
7. The integer 7.
8. The set of integers between 1 and 10.
9. The set of all students who attend this course.
10. The oldest student who attend this course.

For an item considered as a phenomenon, identify the underlying concept.

Exercise 1.4. *University Concepts*

In a university study, the study activities are usually structured in a number of semesters. There are two kinds of study activitities: projects and courses. At Aalborg University, there are currently two kinds of courses: Study courses (dk: studieenhedskurser) and project courses (dk: projektenhedskurser).

Characterize the concepts of *university study*, *study activity*, *semester*, *project*, *course*, *study course*, and *project course* relative to Aggregation/Decomposition and Generalization/Specialization.

3.5. References

[Jlk-kst] A Conceptual Framework for Programming Languages: Jørgen Lindskov Knudsen and Kristine Stougaard Thomsen, Department of Computer Science, Aarhus Universitet, PB-192, April 1985.

4. Towards Object-oriented Programs

Below we will return to the example of the Hangman game, which we studied as a structured program in Section 1.2.

4.1. An object-oriented program: Hangman

Lecture 1 - slide 21

In Figure 4.1 we show a class diagram of our object-oriented version of the Hangman game.

The class `Puzzle` encapsulates the data of a single puzzle (the category and the word phrase). The class also offers an interface through which these informations can be accessed.

The class `PuzzleCollection` represents a number of puzzles. It is connected to the file system (or a database) where we keep the collection of puzzles while not playing a game. How this 'persistency' is actually handled is ignored for now.

Similar observations can be done for `HighscoreEntry` and `HighscoreList`.

The class `HangmanGame` encapsulates the state of a single (round of the) hangman game. It has associations to a player and a secret puzzle, and to the collections of puzzles and highscore entries. We do not in `HangmanGame` want to commit ourselves to any particular user interface. Thus, the actual user interface of the game is not part of the `HangmanGame` class.

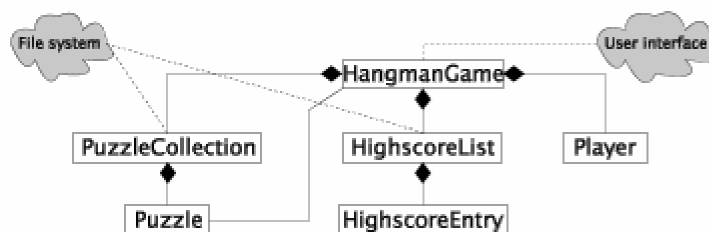

Figure 4.1 *The classes of a Hangman program. At the left hand side we see that PuzzleCollection is formed by Puzzle parts. Similarly, the HighscoreList is formed by HighScoreEntry parts. The HangManGame class if formed by three parts: PuzzleCollection, HighScoreList, and Player. Both file system and user interface aspects are "cloudy" in this diagram.*

Below we show sketches of the individual classes in the game. The classes and all the operations are marked as abstract, because the operations of the classes are not implemented, and because the current OOP version of the Hangman game is written at a very high level of abstraction. The concatenation of all classes in Program 4.1 - Program 4.5 can actually be compiled with a C# compiler. Abstract classes are discussed in Chapter 30.

The operation interfaces of the classes are most probably not yet complete, but they are complete enough to let you have an impression of the object-oriented programming approach.

```
1  abstract class Puzzle {
2
```

```
3    public abstract string Category{
4      get;
5    }
6
7    public abstract string PuzzlePhrase{
8      get;
9    }
10
11   public abstract int NumberOfCharsToGuess();
12 }
```

Program 4.1 *The class Puzzle.*

Given that Program 4.1 - Program 4.6 contain so many abstract operations we will touch a little bit on what this means. It is not intended that you should learn the details of abstract classes here, however. This is the topic of Section 30.1. As noticed above, the abstract class shown in Program 4.1 can actually be compiled with a C# compiler. But it is clear that the class cannot be instantiated (no objects can be made). It is necessary to create a subclass of `Puzzle` in which we give the details of the abstract operations (methods and properties). Subclassing and inheritance will be discussed in Chapter 25 and subsequent chapters. In the subclass of `Puzzle` we need to supply puzzle data representation details.

```
1  abstract class HighScoreEntry {
2
3    public abstract Player Player {
4      get;
5    }
6
7    public abstract int Score{
8      get;
9    }
10 }
```

Program 4.2 *The class HighScoreEntry.*

Let us make a technical remark related to programming of abstract classes in C#. It is necessary to mark all operations (methods and properties) without bodies as 'abstract'. When a class contains at least one abstract operation, the class itself must also be marked as abstract. It is not sufficient to use the abstract modifier on the class.

```
1 abstract class Player {
2
3   public abstract string Name{
4     get;
5   }
6
7 }
```

Program 4.3 *The class Player.*

```
1  abstract class PuzzleCollection {
2
3    public abstract string Count{
4      get;
5    }
6
7    public abstract Puzzle this[int i]{
8      get;
9    }
10
11   public abstract void Add(Puzzle p);
12
```

```
13 }
```

Program 4.4 *The class PuzzleCollection.*

```
1  abstract class HighScoreList {
2  
3    /* Invariant: Entries always sorted */
4  
5    public abstract void Open(string FileName);
6  
7    public abstract string Count{
8      get;
9    }
10 
11   public abstract HighScoreEntry this[int i]{
12     get;
13   }
14 
15   public abstract void Add(HighScoreEntry e);
16 
17   public abstract void Close();
18 
19 }
```

Program 4.5 *The class HighScoreList.*

The class `HangmanGame` in Program 4.6 (only on web) shows an outline of top-level class, cf. Figure 4.1. The operations in this class are intended to be called directly or indirectly by the `Main` method (not shown).