A Method for Your First Object-Oriented Project

Ari Jaaksi
Nokia Telecommunications
Network Management Systems
Hatanpäänvaltatie 36 B, FIN-33100 Tampere, Finland
ari.jaaksi@ntc.nokia.com

Jaaksi A., 'A Method for Your Object-Oriented Project' in JOOP, (Journal of Object-Oriented Programming), Vol 10, No 9, January 1998, pp 17 - 25.

Keywords: object-orientation, methodology, software process, notation

Abstract

This paper presents a pragmatic and simple approach to developing object-oriented applications. The approach is based on two fundamental principles of object-oriented software: objects and their co-operation. Process descriptions from collecting customer requirements to implementing code, as well as modelling notations are presented.

Commercial object-oriented methods are complicated and often hard to learn and use. The approach presented here is simple and easy to apply and develop further. The method includes only two notations and five clear steps. Still, it covers the software development process from requirements capture to testing. The approach can be used in the very first object-oriented projects of a company, and in domains and environments that are clear and simple enough. As regards more complex domains or environments the method scales up.

1. Introduction

Object-oriented literature is full of different methods and notations. There are plenty of books and publications explaining how to develop object-oriented software 'in the right way'. Various companies are selling consultation and courses in object-orientation. Information about how to develop software in an object-oriented way is available for those who are willing to study.

In practice, companies cannot afford to study, take courses, and buy consultation for long periods of time. Especially small and middle sized companies would need an easy start, since they cannot hire expensive consultants to take care of object religion issues. Therefore, companies need a simple but effective way to develop their first object-oriented systems.

A methodology is more likely to be used when it is simple, clearly effective and small[1]. Still, I have noticed that most object-oriented software development methods are too big and complex to be used in real software projects. The problem is also discussed by Lilly[2], and Henderson-Sellers and Edwards[3], for example. Instead of detailed and complex modelling notations we need practical methods with clear notations and simple process description. These methods should be simple enough for an average software company to apply, and they should be easy to learn. The problem is that such methods may be too simple for us software scientists, because we all can find weaknesses in them. I argue that it does not really matter if the method does not cover full details. Instead, the method should first concentrate

on covering the most important aspects of software development. Only after that can the details be handled.

Current object-oriented methods cannot be described as simple and easy to understand because they try to cover each and every aspect of software development. For example, the mere introduction of OMT[4] and OOSE[5] requires about 500 pages of text. The complexity of these methods makes it sometimes hard to see what is really essential in them. The new Unified Modeling Language[6] seems to even increase the complexity by introducing new notation details and concepts. Thus, a simplified version is needed, especially for beginners and for people working in small projects.

A systematic approach is needed to guide software development. A method, even a simple one, can assist an organisation in managing their own way of creating software. Process improvement activities, such as the ones presented in CMM[7], for example, can be performed by modifying and developing the method. This can only be done if the method is followed in practice. Without a repeatable method in use the organisation cannot learn from mistakes and cannot improve its software development processes.

This paper presents an example of a compact and pragmatic approach to constructing object oriented applications. The approach is called 'The Simplified Method', since it is a simplification of the OMT++ method[8], which is widelly used, e.g., within Nokia Telecommunications. OMT++ is based on commercial methods, such as OMT[4], Fusion[9], and OOSE[5], and on the work done within Nokia when adopting the methods in real software projects. OMT++ is used, for example, in the development of one of the leading network management systems, the Nokia NMS/2000, for the digital GSM/DCS cellular networks. The size of the system is nearly two million lines of C++ code and nearly a hundred executable applications. Software runs in UNIX workstations using the X-Window System, relational databases and various third-party components. The product itself won the software product of the year '95 award in Finland[10]. Some other smaller projects within Nokia Telecommunications have left elements out from the OMT++. These projects use OMT++ very much in a form of the Simplified Method presented in this paper.

2. Requirements for a Methodology

A method, even a simple one, should meet the following requirements. The method should

- guide the development of a system all the way from customer requirements to testing,
- include both notations and process descriptions,
- specify phase products, such as documents and figures,
- allow extension, and
- be easy to learn and use.

The method must support three aspects of the system to be developed. First of all, the method must model the functionality of the system, i.e., what the system provides to the end user. Secondly, the method must model the objects that constitute the system; what the objects are and how they relate to each other. The method must help to discover the objects based on the analysis of the problem domain. The method must also refine and transform the domain objects into a form that can be implemented in a programming language. Thirdly, the method must model how objects collaborate to provide desired functionality. In addition to these three, it should be clear to every designer why each figure or text is produced, and how they support software development.

3. Overview of the Simplified Method

3.1. Notation

The notation of the Simplified Method includes three main elements: *natural language, class diagrams,* and *sequence diagrams*. Natural language is the main tool to capture requirements, and it is typically used whenever there is a need to communicate with the end users. Natural language is also used if there is a need to emphasise something related to diagrams. Class diagrams provide a static view to the objects related to the system in various phases of system development. Sequence diagrams provide a functional view to the objects by illustrating the co-operation between the objects.

All figures should be clear and readable. Class diagrams and sequence diagrams should only illustrate what is essential. If we need to choose between under-modelling and over-modelling, we should always choose under-modelling and add textual commentary.

The Simplified Method uses the class diagram notation of UML[6], although not all the details of the notation are necessary. Typically, it is enough to depict one-to-one associations, one-to-many associations, many-to-many associations, aggregation and inheritance. Also, it is a good habit to give

names to associations, that are not inheritance or aggregation. A name and possibly attributes and operations are illustrated for each class. Figure 1 depicts all the main elements of the class diagram notation. A car, which is either a sedan or a station wagon, includes one engine, and an owner may own many such cars.

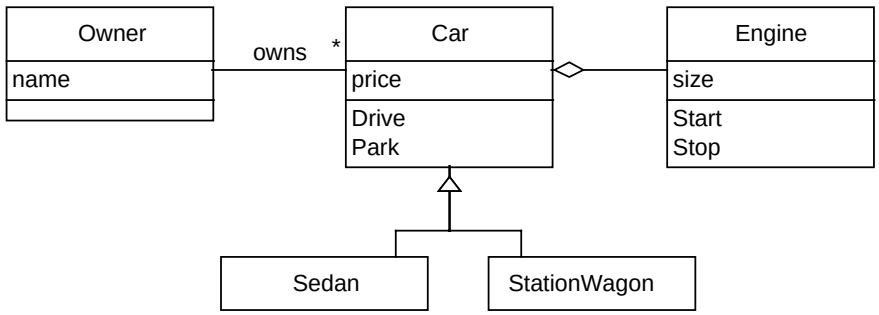
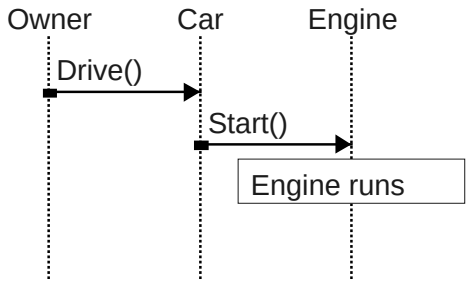


Fig. 1. Basic elements of the UML class diagram notation

Sequence diagrams illustrate how instances of various classes communicate with each other. Each sequence diagram illustrates a sequential flow of events. The flow may be set in motion by the end user's actions, such as pressing a button or moving a slider, or by internal incidents, such as calls from a timer. Thus, sequence diagrams depict how a set of objects communicate in order to provide desired functionality. Figure 2 illustrates the basic elements of the sequence diagram notation. Time runs from top to bottom. The names of objects are written at the top of the sequence diagram figure. Arrows are messages from an object to another. An arrow starts from a caller object and points to an object that is being called. The name of each message is written above the arrow, and the message can include parameters. Comments and actions that refer to a single object can also be added.

```
Description:    The owner starts to drive his car
Preconditions:  The car is parked
Postconditions: The owner is driving his car
```




```
Exceptions:     Cannot start the engine
```

Fig. 2. The notation of sequence diagrams

3.2. Process and artefacts

Figure 3 illustrates the five phases of the Simplified Method, namely *Requirements Capture, Analysis, Design, Programming,* and *Testing*. Although the phases are listed sequentially, iterative approach can be adopted, as will be presented later in this paper. Requirements capture collects all requirements that there are for the system to be developed. The analysis phase aims at modelling the concepts, i.e., the objects of the problem domain, as well as analysing the operations of the system. In the design phase the products of the analysis phase are transformed into a form that can be programmed. Design illustrates how the objects form structures, what their interfaces are, and how they collaborate. The programming phase produces the code and typically concentrates on one class at a time. Finally, the test phase tests the system against the requirements.

There are two parallel paths in the process of system development: The *static path* uses class diagrams to illustrate the static properties of the system, and the *functional path* uses operation descriptions and sequence diagrams to illustrate the functional properties of the system. These two paths are related to each other, and they both aim at a code that has been tested, as illustrated in Figure 3.

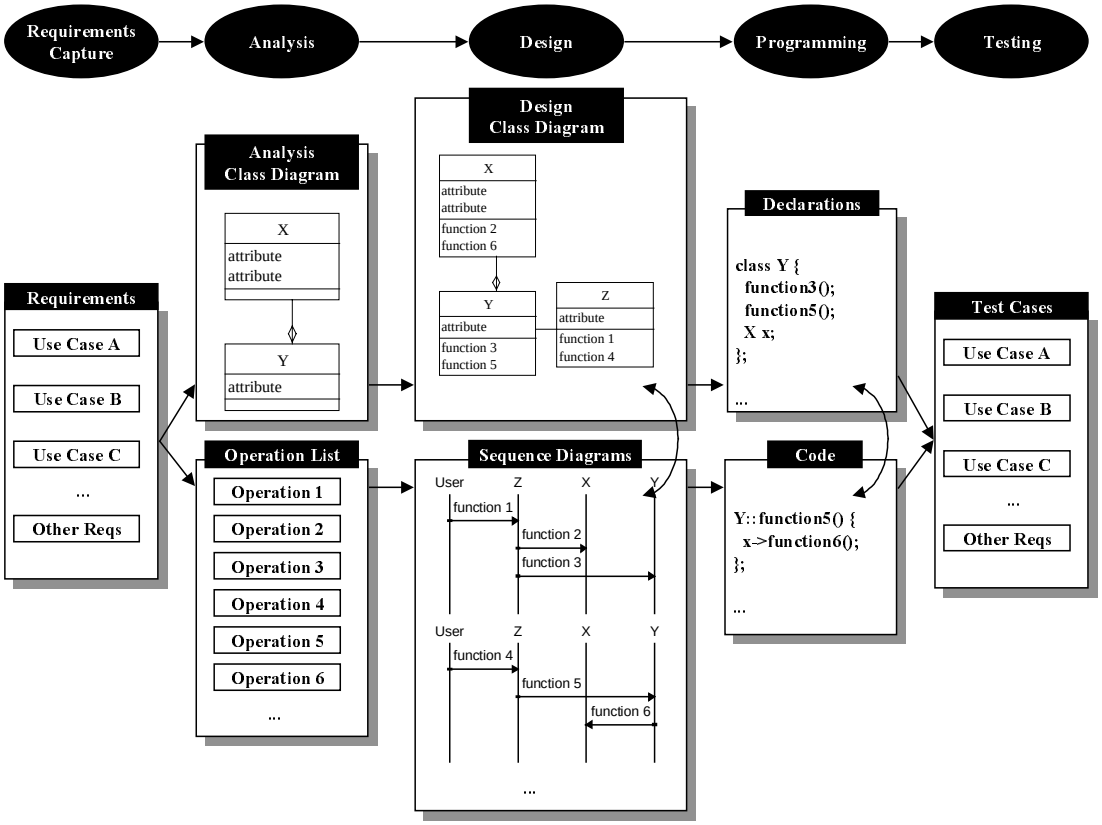


Fig. 3. Static and functional paths of the method

4.Phases of the Method

4.1. Requirements Capture

The process of developing a system starts with requirements capture. The purpose of the phase is to communicate with end users and document the requirements. All requirements should be exact and measurable. The requirements are divided into *functional* and *non-functional requirements*. Functional requirements can be modelled in the form of use cases [5,11,12], which are textual stories describing the use of the future system. For simplicity, the Simplified Method does not use any the notation for the use cases.

This paper uses a simple example to illustrate the phases of the Simplified Method. The example illustrates the development of an application that allow elementary school pupils to compose simple musical tunes. The application is called 'Elementary Composer'. Figure 4 describes the use cases of the application, and Figure 5 illustrates all the non-functional requirements.

```
Use Case #1: Composing a tune
      A pupil starts the application showing an empty staff and a selection of
      possible note types. The pupil selects a note type, i.e. a quarter note,
      a half note, a quarter rest, etc. with a mouse. Then he points to a place
      on the staff where he wants the note of the selected type to appear. By
      selecting note types and pointing to locations on the staff he constructs
      a tune. The pupil plays his tune and saves it on a disk. Finally, he
      closes the application.

Use Case #2: Listening to a previously composed tune
      The pupil starts the application. He loads a tune that he wants to hear
      from a disk. All notes of the selected tune appear on the staff. After
      that the pupil plays the tune. Finally, he closes the application.
```

Fig. 4. Use Cases, i.e., functional requirements of the example application

```
Non-functional requirement #1:
      The application supports the C major scale and eight, quarter, and half
      notes and quarter rests.
Non-functional requirement #2:
      The maximum length of a tune is 20 notes.
Non-functional requirement  #3:
      Tunes are stored as ASCII files.
```

Fig. 5. Non-functional requirements of the example application

The requirements are discussed with the customer. If possible, the customer should participate in the writing of the use cases. In any event, the use cases are written so that the customer can understand them and make comments. Sketches of the user interface can make the use cases more concrete.

After the use cases and other requirements have been documented and agreed on with the customer, they form the basis for the following phases of the system development. In each step, phase products

must be checked against the use cases and non-functional requirements. Finally, the use cases form the basic test case set for system testing.

4.2. Analysis

The purpose of analysis is to understand the problem domain and the system to be implemented. The analysis phase is based on the collected requirements and use cases, and the phase includes two tasks, namely *object analysis* and *behaviour analysis*. Object analysis aims at specifying all key concepts related to the system to be developed. It produces an *analysis class diagram* that documents the concepts of the problem domain. Behaviour analysis defines the operations that the user performs with the system. Behaviour analysis models the system as a black box. It models only the external functionality of the system and produces an *operation list*. The final system must support the performance of all the operations in the list.

Although the operation list and the analysis class diagram are separate models, operations include and use the concepts defined by the class diagram. Still, we do not try to push operations back into the analysis class diagram by 'guessing' member functions for the classes. Thus, the analysis class diagram includes few operations; typically, only classes and their attributes are presented.

There is a lot of information available about object analysis. For example, OMT[4] and OOSE[5] methods explain how analysis objects can be found and refined. These practices are based on identifying concrete real world objects, which can be found from the requirements by searching for nouns and analysing terminology used by the end users, for example. Still, it seems that analysis is a difficult phase to master. Therefore, it is suggested that analysis is done in teams that consists of experts on object analysis and experts on the problem domain. While experts on the problem domain, such as the customers themselves, can contribute and participate in analysis, experts on object modelling can refine knowledge and draw correct and useful class diagrams.

Figure 6 illustrates the analysis class diagram of the Elementary Composer application. The composed tune is written on staves. Each staff consists of notes, and the duration and pitch of each note is specified. The notes can be of various types, such as quarter notes, half notes, or quarter rests.

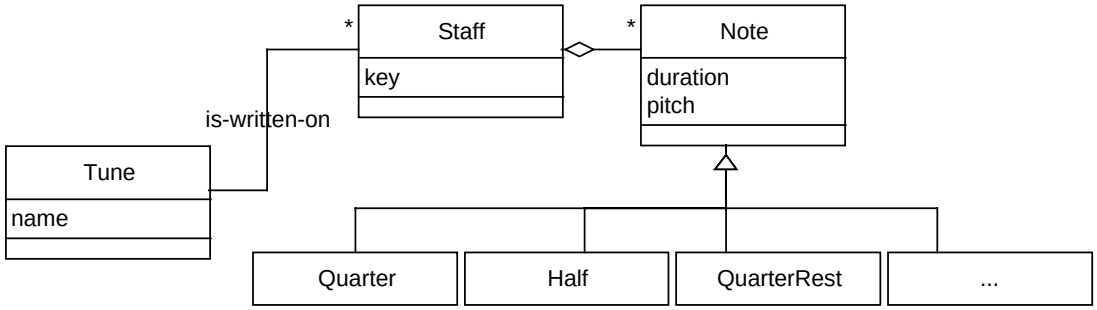


Fig. 6. The analysis class diagram of the Elementary Composer application

Behaviour analysis produces the operation list, which is constructed on the basis of use cases. Figure 7 lists all the operations performed with the Elementary Composer application. Operations 1 - 5 and 7 are found from the first use case and operations 4, 6, and 7 are found from the second use case.

```
1. Starting the application.
2. Selecting a note type.
3. Placing a note on the staff.
4. Playing a tune.
5. Saving a tune.
6. Loading a tune.
7. Closing the application.
```

Fig. 7. Operations performed with the Elementary Composer application

The Simplified Method presented in this paper does not include a separate phase for the specification of user interfaces. Typically, simple user interfaces, such as presented in figure 8, can be designed by drawing them with one of the various graphical user interface builders or application development environments. User interface prototypes can also be built with the help of real end users. Still, large and complicated user interfaces may require more complete analysis methods, such as presented in OMT++[13], for example.

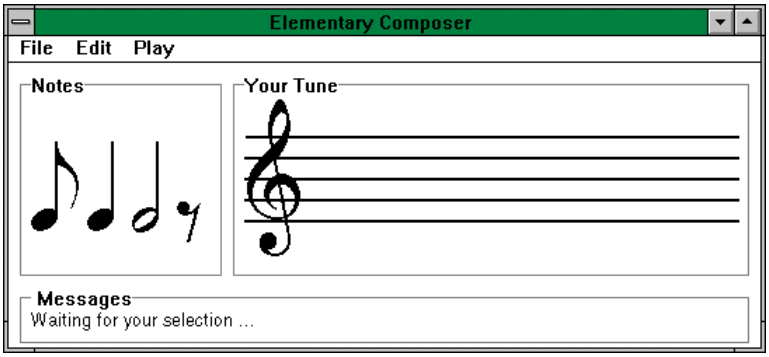


Fig. 8. The main window of the example application

4.3. Design

The purpose of design is to transform the products of the analysis phase into a form that can be implemented in a programming language. While analysis concentrates on objects and functionality which are relevant to the end user, design deals with objects and functions that will be programmed.

The design phase includes two paths, as illustrated in Figure 3. The analysis class diagram is transformed into the design class diagram, and the operations in the operation list are modelled as sequence diagrams. Domain objects discovered in analysis are modified so that they can be implemented with a selected programming language. Modification means, for example, adding implementation specific classes, modifying class structures, and identifying operations and attributes. In the analysis phase the static and functional paths were separate from each other, but in the design phase these two paths meet. The sequence diagrams modify the design class diagram, and the design class diagram provides objects and attributes for the sequence diagrams.

The creation of the design class diagram is a controlled process. We must select or develop an appropriate *application architecture*, an application framework if you will, to form the basis of design decisions. For example, Model-View-Controller approach[14] could be selected for Smalltalk, and the MVC++ approach[15] could be selected for C++ applications. Most of the modern programming environments, such as Borland's Delphi and Microsoft's Visual C++, for example, provide simple frameworks upon which the design decisions can be built on. Thus, successful design requires good knowledge of the selected programming language, operating system, hardware, and other issues which will affect the final coding.

Basic design rules, such as rules on how to manage user interfaces and how to handle databases, are typically dictated by the selected tools. Let us assume that we are going to implement the Elementary Composer application in Object Pascal by using Borland's Delphi programming environment. In the Delphi environment the user interface will be implemented within the user interface classes. Typically, each window or dialogue box is an object. Push buttons, menus, and other controls are objects, too, and they are object members of windows and dialogues. Other objects of the application work together with the user interface objects, thus allowing the communication with the end user and providing the functionality of the application.

We start the design by constructing the *first version* of the design class diagram, which is shown in Figure 9. In the very first version, the analysis class diagram forms the basis of the design class

diagram, and the classes representing the windows of the user interface are added at the top of the model. The user interface classes and their attributes can be found from the user interface figures.

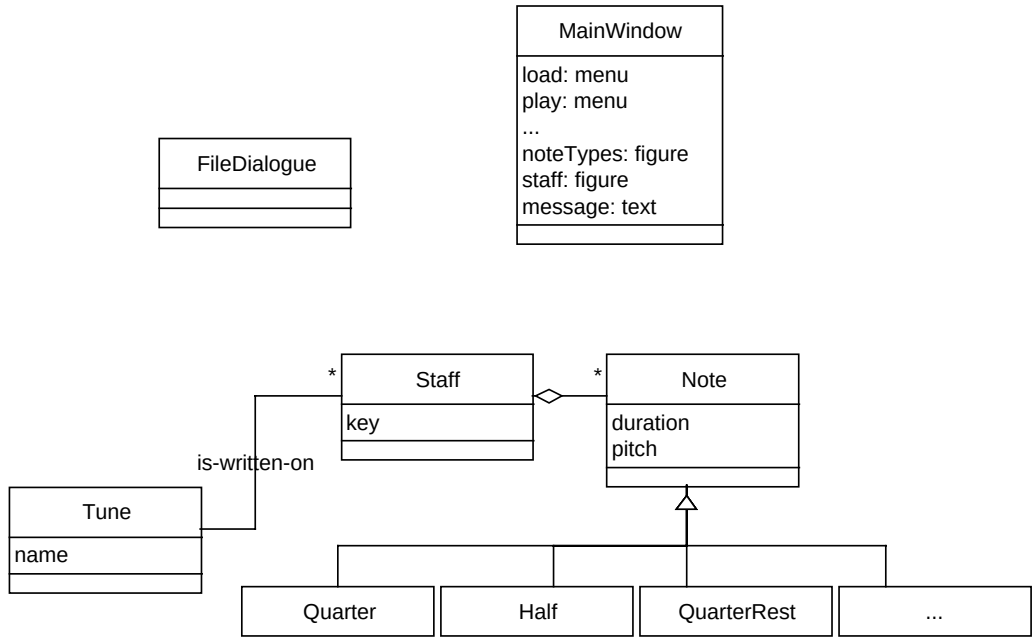


Fig. 9. The first version of the design class diagram

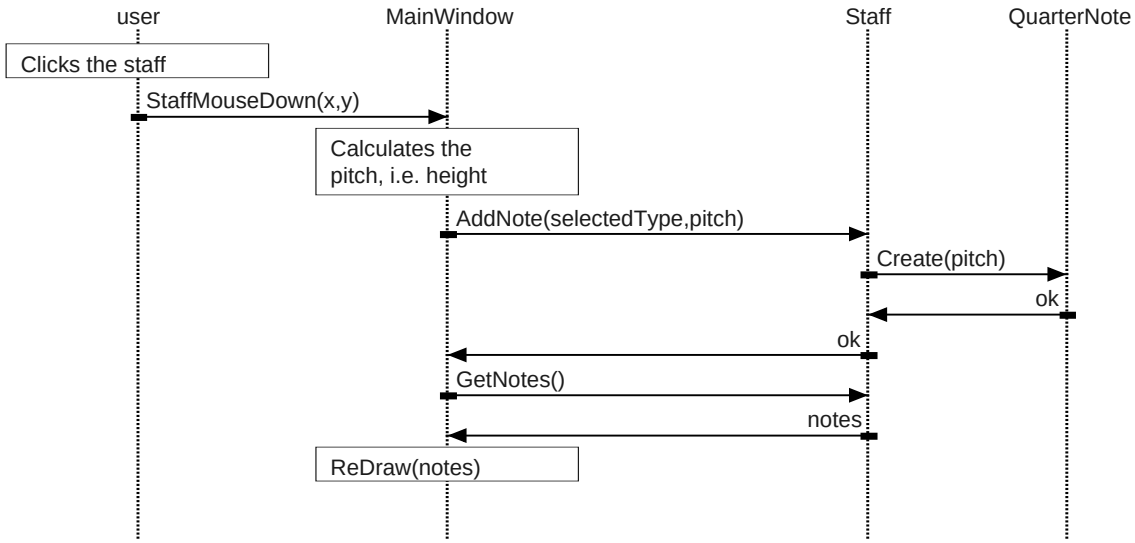
The first version of the design class diagram is immature, and it needs a lot of refining and tuning. Refining is done by using sequence diagrams systematically. We analyse every operation in the operation list and draw sequence diagrams for each operation by using the design class diagram. By doing so we refine connections between the classes and add operations, attributes, and new classes.

When drawing sequence diagrams we specify what responsibilities the design objects have and how they function in practice. An arrow starts from an object that calls the member function of another object, and points to this other object. The name of the function is written above the arrow. Function calls are written with the needed parameters in round brackets, and the return values of the function calls are written without round brackets.

As a first step to refine the design class diagram, we draw an sequence diagram to illustrate how objects communicate with each other and thus allow the end user to place a note on the staff. Figure 10 depicts the ‘Placing a note on the staff’ operation as it will be programmed. Before the operation can start, the user has to select a note type, and the type information is stored within the MainWindow object. The sequence diagram for depicting the selection operation is not illustrated in this paper. As his first action the user clicks the staff on the screen. The clicking launches the ‘StaffMouseDown(x,y)’

member function of the MainWindow. At first the function determines the pitch, i.e., the height of the note to be added, by calculating the position of the mouse relative to the staff. Then the MainWindow object calls the 'AddNote(selectedType,pitch)' member function of the Staff object, which creates the note object. In this case the type of the note is QuarterNote. The 'AddNote(selectedType,pitch)' member function returns the 'ok' message if the creation of the new note has succeeded. Finally, the MainWindow asks for the entire set of notes in order to redraw the tune on the staff. For this the MainWindow calls its own 'ReDraw(notes)' method.

```
Description:    Placing a note on the staff (The note object can be of any
                type, and the sequence diagram illustrates the quarter note
                case.)
Preconditions:  The type of the note is selected
Postconditions: New note is added on the staff
```




```
Exceptions:          Maximum number of notes is exceeded, cannot add a new one:
                     error message is shown in the message field
```

Fig. 10. An sequence diagram illustrating the co-operation between objects when the user places a note on the staff

Each sequence diagram specifies a set of member functions, attributes, and associations and adds them into the design class diagram. For example, an arrow pointing to an object adds a member function to the class in question. Correspondingly, an arrow between two objects adds a connection between the classes in question into the design class diagram. Still, the sequence diagrams do not illustrate object communication in every detail. Instead, the sequence diagrams should be readable and simple. They should only depict the threads of execution as they are normally performed. Therefore, pre- and post-

conditions are written above the sequence diagram and exceptions to the normal sequence of events are written below the sequence diagram.

Figure 11 illustrates the *second version* of the design class diagram. The operations and connections suggested by the sequence diagram illustrated in Figure 10 are added into the model. Based on the sequence diagram, member functions have been added to various classes and a new connection between the MainWindow and the Staff has been established.

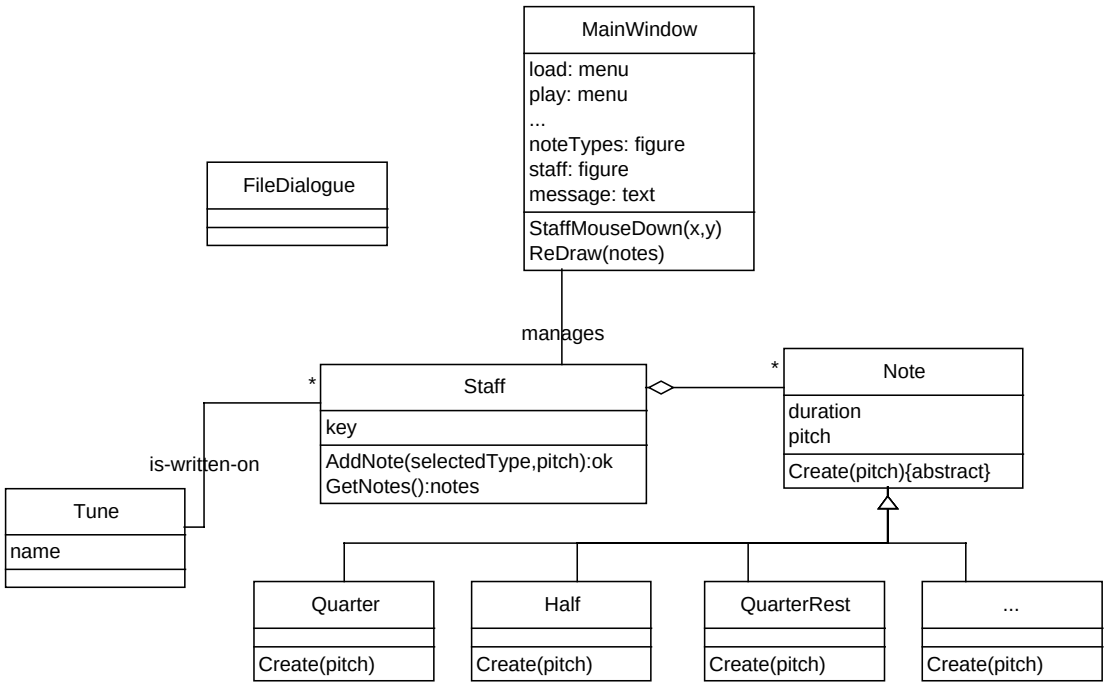
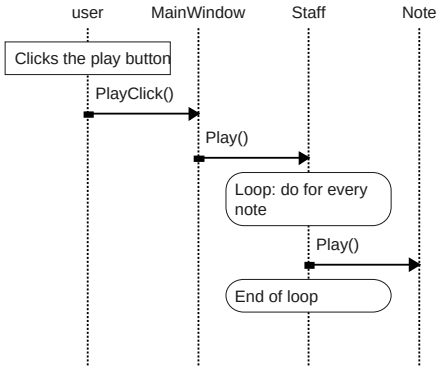


Fig. 11. The second version of the design class diagram

Figure 12 shows how the objects of the design class diagram communicate with each other when the user wants to hear his tune. Thus, the figure specifies the 'Playing a tune' operation. Figure 13 presents the *third version* of the design class diagram with the modifications suggested by the second sequence diagram.

```
Description:    Playing a tune.
Preconditions:  There is a tune on the staff.
Postconditions: The tune is played.
```




```
Exception:      Problems with music drivers, cannot play the
                tune: error message is shown in the message field
```

Fig. 12. Co-operation between objects when playing the tune on the staff

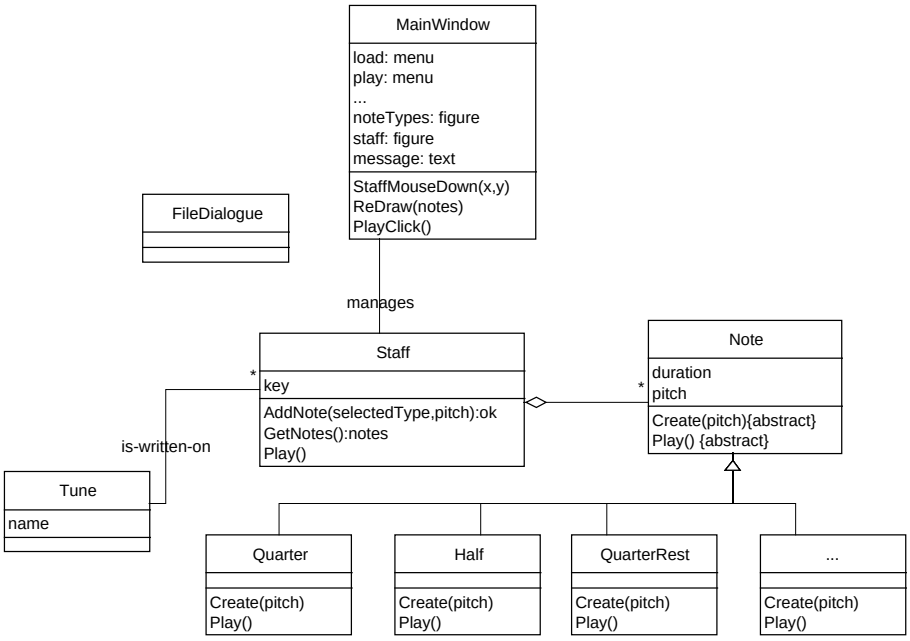


Fig. 13. The third version of the design class diagram

We must specify all the operations in the operation list as sequence diagrams, and refine the design class diagram accordingly. When we draw the sequence diagrams we think about the final implementation, i.e., how the software will function in practice. When drawing sequence diagrams we often need to add new design-specific classes into the design class diagram. We may also decide to alter the object structure specified in the analysis phase, and remove classes that are not involved in any sequence diagram. In any event all changes should be justified and minimised, and unnecessary changes

to the analysis class diagram should be avoided. The goal is to design classes that resemble objects of the real world and are at the same time suitable for implementation.

4.4. Programming

The purpose of programming is to transform the design class diagram and the sequence diagrams into programming language. Design has already modelled all classes of the system and communication between the instances of these classes. According to the Simplified Method, we do not model the inner functionality of individual objects during design. Instead, the programming of classes is based on the design class diagrams and sequence diagrams. Thus, while design concentrates on object structures, object interfaces, and co-operation between objects, programming deals with the inner functionality of individual objects.

The class declarations are based the design class diagrams, and the code of individual member functions is based on the sequence diagrams. Figure 14 illustrates the declaration of the Staff class. The declaration is based on the class diagram presented in Figure 13. Figure 15 illustrates the code of the StaffMouseDown member function of the MainWindow class. It is written in Object Pascal on the basis of the sequence diagram illustrated in Figure 10.

```
NoteStructure = array[0..20] of ^Note;

Staff = class
private
    key: integer;
    notes: NoteStructure;
public
   function AddNote(selectedType: Integer; pitch: Integer):Boolean;
   function GetNotes: PChar;
   procedure Play;
end;
```

Fig. 14. The declaration of the Staff class

```
procedure TMainWindow.StaffMouseDown(...X, Y: Integer);
  var
     pitch : Integer;
     ok : Boolean;
     notes : PChar;
  begin
     {Calculate the pitch based on the Y parameter}
     ...
     ok := MyStaff.AddNote(selectedType, pitch);
     if (ok = false) then
          MessageText.Caption:='Cannot add more notes, sorry.';
     notes := MyStaff.GetNotes;
     ReDraw(notes);
end;
```

Fig. 15. The declaration of the StaffMouseDown procedure of the MainWindow

It is important that all classes, their interfaces, and the co-operation between objects have been modelled during design. Based on such a design a programmer can concentrate on one class and one member function at a time. This is why the Simplified Method does not use any notation for the internal functionality of a single object. In most cases there is no need to assist the implementation of a single class with any additional graphical notations.

4.1. Testing

The purpose of testing is to find errors and ensure that the software functions as planned. Therefore, testing is performed against the requirements. According to the Simplified Method, each use case is run with the implemented system, and every non-functional requirement is checked. Various testing tools can improve the quality of testing by providing views into the implemented code. Still, the most important job in testing is to run each use case and compare the results against the initial use cases. Such black-box testing can spot at least the most severe errors.

5. Iterative software development

Iterative software development constructs applications piece by piece. Systems are implemented in sequential cycles, and each cycle implements only a slice of the required functionality. Each new phase builds on the previous phases and each phase can learn from experiences gained during the previous cycles. After each cycle we can also validate requirements by testing the implemented part of the system and even by delivering it to the customers.

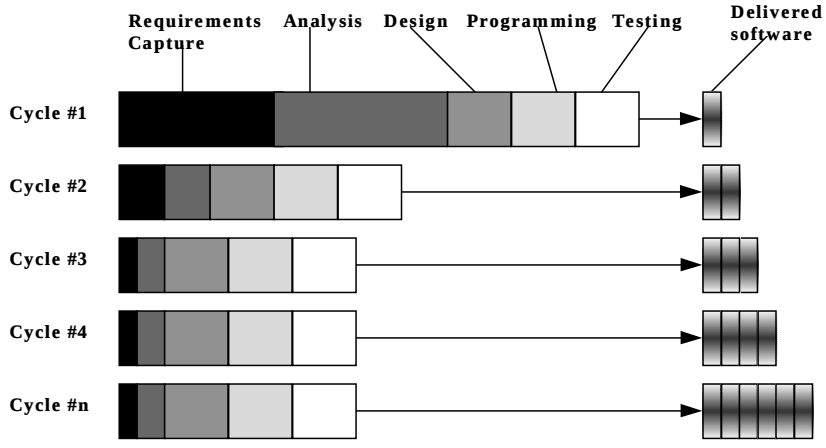


Fig. 16. Phases of iterative software development

Figure 16 illustrates the phases of iterative software development. The length of each slot illustrates the amount of work and time needed. We should perform requirements capture and analysis thoroughly

and carefully already during the first cycle. This is how we ensure that we have right goals for the following cycles. We should collect all requirements that are available before we start the analysis phase. We should therefore write all the use cases that can be specified.

Analysis should be as complete as possible during the first cycle. Without thorough analysis we could easily end up with design solutions that cannot be extended beyond the first iteration cycle. In addition to the analysis phase already presented, we should identify the subset of operations that needs to be implemented during the first cycle. The implementation of these operations forms the first version of the system. This version, capable of performing only a few operations, should then be given to the end users for comments. Typically, the operations of one single use case form a good set to start with. The use case selected for the first iteration should be the most important use case from the end user's point of view.

Only the selected operations are transformed into the form of sequence diagrams during the design, and programmed. Some objects and operations that may have been discovered already in the analysis phase are not implemented at all. The next cycles will take care of the rest of the operations.

It is essential that design is based on a solid architecture. A good architecture allows us to 'plug in' new classes and implement new methods easily without a need to re-implement the results of the previous iteration cycles. For example, the MVC++ approach[15] forms an architecture where new dialogues, objects, operations and other elements can be added to the application skeleton built during the first cycle. Then, each new iteration cycle will build on the previous ones.

6. Documentation and Tools

The software process must be visible. All the phases of the process must produce visual phase products that can be discussed. There are plenty of CASE tools available that support the modelling notations used in this paper. Still, the process should not be dictated by the tools. Especially during the first object-oriented projects it is better to keep tools simple and use, for example, whiteboards and stick-on labels.

Regular office applications can be used for drawing diagrams and writing textual descriptions. For example, a word processor with pre-defined document templates is a good tool. A document template makes all documents look similar and therefore makes them easier to read. The template should define the table of contents and the places where various figures and other models can be attached to.

The software process should produce concrete phase products, which can be organised as documents. The Requirements Document should include use cases and all non-functional requirements. The Analysis Document should contain the analysis class diagram and the operation list. The Analysis Document can also include windows and dialogues of the user interface, although a prototype alone may be sufficient. The Design Document should include the design class diagram and all sequence diagrams. The Design Document should also include all other design decisions, such as database solutions, for example. The code itself is also a document. It is read both by the compiler and the maintainer of the application.

7. Extending the Method

The method presented in this paper may not be adequate for supporting large software projects. Therefore, it must be extended. If the first and rather simple projects are performed according to the method presented in this paper, missing phases and notations, problems in controlling the process, and other similar issues should be documented and analysed. Based on these experiences the method can be improved for further projects.

The Simplified Method is a subset of the OMT++ method[8]. OMT++ is tuned to handle the construction of large systems[16]. In addition to the method presented in this paper, OMT++ has a number of extensions. Requirements capture uses more complete use case analysis[5,11,12]. Sub system division and architectural design is a part of the process[8,16.]. A controlled user interface specification phase has been included[13]. The design phase is supported by the MVC++ application framework[15] which is based on the MVC model[14]. The framework has been extended with object libraries consisting of user interface components[17], for example. Testing includes three separate phases: module testing is a white-box testing phase for the implemented classes, processes and libraries, integration testing tests the co-operation between various sub-systems, and system testing tests the system from the end user's point of view. In addition to these extensions, we are currently introducing, among other things, the use of design patterns[18] and usability evaluations in our process.

It seems to be a good idea to have a process improvement team within the organisation. Such a team, or an individual, is able to support the use of the selected method, collecting the best practices, and improving the method together with the rest of the members of the organisation. The Simplified Method presented in this paper forms a good basis for such improvements. It can be used as a

backbone of the projects, and it can be extended when moving upwards the CMM ladders, for example [3,7,16].

Conclusions

When a company starts to develop software in an object-oriented way, a pilot project is needed. During the pilot project the company can learn the new technology and adopt new ways of working. In order to get the most out of the pilot project, the company needs a simple but powerful method for developing the software. Without a systematic approach the company cannot maximise learning. Ad hoc hacking during the pilot project does not provide real knowledge for the future projects, although some details may be tackled. Thus, a simple and practical method is needed to form the basis for the development of object-oriented practices of the company, and this method should be used from the very first pilot project.

This paper presents such a method. The method consists of five clear steps and a simple notation. Although the method is simple, it covers all phases from collecting customer requirements to testing the code. It includes notations and process descriptions and specifies phase products. It is a complete methodology that can be extended and modified. Although the method itself is described in a waterfall form, iterative software development is also discussed.

The Simplified Method presented in this paper is some kind of a OMT++-Lite; a trimmed down version of a method, as suggested by Lilly[2]. The Simplified Method does not introduce any new notations or phases. Instead, it uses the tools of UML[6], OMT[4], OOSE[5], and Fusion[9]. This paper is based on the experiences gathered when applying these methods in real software projects, and it concentrates on what seems to be essential in each phase. The programming phase is illustrated with Object Pascal, which is an easy object-oriented language. Instead of a more complex object-oriented language, such as C++, this paper uses a simpler language, Object Pascal, as suggested by Henderson-Sellers and Edwards[3], for example.

The approach presented in this paper is a subset of the OMT++ method[8] which is in operational use in Nokia Telecommunications and some other European companies. In addition to the method presented in this paper, OMT++ includes means to manage large systems and frameworks, specify user interfaces, design process division, etc. OMT++ aims at developing large and complex systems, whereas the method presented in this paper aims at developing small or middle sized systems only. Also, the method

presented in this paper is tailored for the first object-oriented projects of a company, and therefore, it has been simplified. The method can be extended, though, as have been presented.

The Simplified Method does not address reuse as its main goal. Before any reuse can take place, the company needs to follow a systematic and repeatable method. Activities to support reuse, such as identification of reusable components and modifications in class inheritance hierarchies, should be added to the method once it is in operational use. Also, the method presumes that the system is built from scratch. Reusing the components of the system in other projects or building on the current software is not discussed in this paper.

There are risks in adopting a simple approach for software development. A method that is too simple may fail in specifying the key properties of object systems. Still, I feel that the real problem with the methods available is the complexity of them. A method needs to be simple and intuitive enough so that it is easy to identify the added-value obtained by following the method. If the method is too complicated, the engineers will not commit themselves in working with it, and repeatable process can not be achieved. Therefore, instead of trying to cover full details of software development, the method should first support only the most important phases. Only after they are applied in practise can the method be extended.

Acknowledgements

I would especially like to thank my colleague Juha-Markus Aalto. He has given me valuable feedback on this article. I would also like to thank Ilkka Haikala and Reino Kurki-Suonio from the Tampere University of Technology, and Leena Rasinaho from our own organisation for their comments on various versions of this paper.

References

[1]. A.R. Cockburn. 'In Search of a Methodology'. *Object Magazine*, Vol. 4, Num. 4, July-August, 1994, pp. 52-76.

[2]. S. Lilly. 'Planned Obsolescence'. *Object Magazine*, Vol. 3, Num. 5, 1994, pp. 79-80.

[3]. B. Henderson-Sellers and J.M.Edwards. 'Identifying Three Levels of O-O Methodologies', *Report of Object Analysis and Design*, Vol. 1. Num 2., July-August, 1994, pp 25 - 28.

[4]. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, 1991.

[5]. I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard. *Object-Oriented Software Engineering. A Use Case Driven Approach.* Addison-Wesley Publishing Company, 1992.

[6]. UML Notation Guide. Version 1.0. Rational Software Corporation, 1997 (www.rational.com)

[7]. M. Paulk, C. Weber, S. Garcia, M. Chrissis and M.Bush. *Key Practices of the Capability Maturity Model, Version 1.1*. Software Engineering Institute, Carnegie Mellon University, 1993.

[8]. J-M. Aalto and A. Jaaksi. 'Object-Oriented Development of Interactive Systems with OMT++'. In R.Ege, M.Singh, B.Meyer, editors, *TOOLS 14, Technology of Object-Oriented Languages & Systems*, Prentice Hall, 1994, pp 205-218.

[9]. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P.Jeremaes, *Object-Oriented Development. The Fusion Method.* Prentice Hall, 1994.

[10]. Tietoviikko, Vol. 4, Num 37, 1995. (In Finnish)

[11]. I. Jacobson. 'Basic Use Case Modeling', in Report of Object Analysis and Design, Vol. 1, Num. 2, July-August, 1994, pp. 15-19.

[12]. J. Rumbaugh. 'Getting Started, Using Use Cases to Capture Requirements', in *Journal of Object-Oriented Programming*, September 1994, pp. 8-23.

[13]. A. Jaaksi. 'Object-Oriented Specification of User Interfaces', in *Software Practice & Experience*; Volume 25, No. 11, November 1995, pp 1203-1221.

[14]. G.E. Krasner, S.T. Pope. 'A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80'. *Journal of Object-Oriented Programming*, August/September, 1988, pp 26 - 49.

[15]. A. Jaaksi. 'Implementing Interactive Applications in C++', in *Software Practice & Experience*. Volume 25, No. 3, March, 1995, pp 271-289.

[16]. J-M. Aalto. Challenges in Applying Objects to Large Systems. In J. Iivari, K. Lyytinen, M. Rossi, editors, '*Advanced Information Systems Engineering, 7th International Conference, CAiSE '95'*, Springler, pp.154 - 167.

[17]. V. Punkka. 'Intelligent Reusable User Interface Components', in *The X Resource, Issue 13*. O'Reilly & Assoc. Inc. 1995, pp 201-215.

[18]. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.