Computación y Sistemas Vol. 9 Núm. 1, pp. 041-054

The Practical Design Method: A Software Design Method for a First Object-Oriented Project

El Método Práctico de Diseño: Un Método de Diseño de Software para un Primer Proyecto Orientado a Objetos

Jorge L. Ortega Arjona
Departamento de Matemáticas,
Facultad de Ciencias, UNAM
Tel. +52 56 22 4858
E-mail jloa@fciencias.unam.mx

Article received on February 24, 2002; accepted on March 18, 2005

Abstract

Commercial object-oriented design methods are often complicated and hard to learn and use. This paper presents a software design method that is a pragmatic and simple approach to designing object-oriented applications. It is based on the fundamental principles of object-oriented design: objects and their cooperation. Design process descriptions are described, considering modeling notations, as well as steps from collecting customer requirements to implementing code. The design method includes only three notations and five clear steps. Still, it covers the software design process from requirements capture to testing. The approach presented here is simple and easy enough to apply and further develop. It can be used in the very first object-oriented projects of a company, and in domains and environments that are clear and simple enough, such as a design course in a computing school. For more complex domains or environments, the method scales up.
Keywords: Object-Oriented, Software Design, Design Method, Notation, Process, Artifactcs.

Resumen

Los métodos de diseño orientados a objetos comerciales son frecuentemente complicados y difíciles de aprender y usar. Este artículo presenta un método de diseño de software como una aproximación pragmática y sencilla para el diseño de aplicaciones orientadas a objetos. Se basa en los principios fundamentales del diseño orientado a objetos: los objetos y su cooperación. Las descripciones de los procesos de diseño se describen, considerando notaciones para el modelado, así como para los pasos desde la recolección de requerimientos del cliente hasta la implementación del código. El método de diseño incluye tan solo tres notaciones y cinco pasos. Aun así, cubre el proceso de diseño desde la captura de requerimientos hasta las pruebas. La aproximación que se presenta aquí es sencilla y suficientemente fácil para aplicarse y continuar desarrollándose. Puede usarse en los primeros proyectos orientados a objetos de una compañía, y en dominios y ambientes suficientemente claros y sencillos, como un curso de diseño en las escuelas de computación. Para dominios y ambientes más complejos, el método escala.
Palabras Clave: Orientación a Objetos, Diseño de Software, Método de Diseño, Notación, Proceso, Artefactos.

1 Introduction

Object-Oriented literature is full of different methods and notations for software design. There are plenty of books and publications explaining how to design object-oriented software in the "*right*" way. Various companies are selling consulting

and courses in object-oriented design. Moreover, information about how to design software in an object-oriented way is available for those who are willing to study.

In practice, however, many companies cannot afford to study, take courses, and pay for consulting during long periods of time. Especially small and medium size companies need an easy start, since they cannot hire expensive consultants to take care of the particularities of object-oriented design. Therefore, companies need a simple but effective way to design their first object-oriented software system.

A design method is more likely to be used when it is simple, clearly effective, and small [3]. Still, it is noticeable that most object-oriented design methods are too big and complex to be used in small software projects. This problem has been widely discussed by Lilly [11], and Henderson-Sellers and Edwards [6]. Their final conclusion has been that instead of detailed and complex modeling notations, practical methods are needed with clear notations and a simple design process description. These methods should be simple enough for an average software designer to apply; and they should be easy to learn. The main disadvantage of such design methods is that they may be too simple for computer scientists, who can easily find their weaknesses and criticize them. Nevertheless, in practical terms, it does not really matter if a design method does not cover full details. Instead, the design method should first concentrate on covering the most important aspects of software design. Only after that, the details can be handled.

Current object-oriented design methods cannot be described as simple and easy to understand, because they try to cover every single aspect of software design. For example, the very introduction of some traditional design methods, such as those in OMT [14] and OOSE [9], requires reading about 500 pages of text. The complexity of these design methods have made hard to notice what the essentials in them are. Furthermore, the Unified Software Development Process (or simply UP) [8] seems to even increase the complexity of its predecessors, OMT and OOSE, by introducing further notation details and concepts. UP and UML [1] may one day provide all the tools for handling all the details of object-oriented design. Nevertheless, a simplified version is still needed, especifically for begginers and for people working in small projects.

A systematic approach is needed to guide software design. A design method, even a simple one, which can assist an organization in managing their own way of creating software. Process improvement activities, such as the ones presented in the Capability Maturity Model [12], for example, can be performed by developing and modifying such a design method. This can only be done if this design method is followed in practice. Without a repeatable design method in use, the organization cannot learn from its mistakes and cannot improve its software design process.

This paper presents an example of a compact and pragmatic approach to designing and constructing object-oriented applications. The approach is called “Practical Design Method”, since it is based on the Practical Design Methodology proposed by Salt and Rothery [15], but aiming especifically for Software Design.

2 Design Method Requirements

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

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

Moreover, a design method must support three aspects of the system to be designed. First of all, the design method must model the functionality of the system, this is, what the system provides or does for the end-user. Secondly, the design method must model the objects that constitute the system; what the objects are and how they are related to each other. The design method must help to discover the objects based on the analysis of requirements. The method must also refine and

transform the objects into a form that can be implemented in a programming language. Thirdly, the design method must model how objects collaborate to provide the desired functionality. In addition to these three aspects, the design method should make clear to every designer the reason why each diagram or text is produced, and how they support software design.

3 Overview of the Practical Design Method

3.1 Notation

The notation of the Practical Design Method includes four main elements: *natural language*, *object diagrams*, *class diagrams*, and *interaction diagrams*. In fact, the method is proposed to use the UML object modeling notation [1]. However, not all the details of the UML notation are, by now, necessary [4]. Natural language is the main tool to capture requirements, and it is typically used whenever there is a need to communicate with end-users. Natural language is also used if there is a need to emphazise something related to diagrams. Object and class diagrams provide a static view of the objects related within the system during various phases of design. Interaction diagrams provide a functional view of the objects, by illustrating the cooperation among them.

All diagrams should be as clear and readable as possible. Objects diagrams, class diagrams, and interaction diagrams should only describe what is essential. If at a certain moment it is necessary to choose between under-modeling and over-modeling, under-modeling should always be chosen, adding some textual commentaries. This principle is also followed when using UML [1].

Typically, it is enough to depict relations such as associations, aggregations, and compositions in object diagrams, and associations, aggregations, compositions, and inheritance in class diagrams. In both these diagrams, it is a good habit to particularly give names to associations. In the class diagrams, a class name and lists of possible variables and operations are considered for each class. Figure 1 depicts the main elements of the notation considered for object diagrams, while Figure 2 shows the basic elements considered for class diagrams [4].

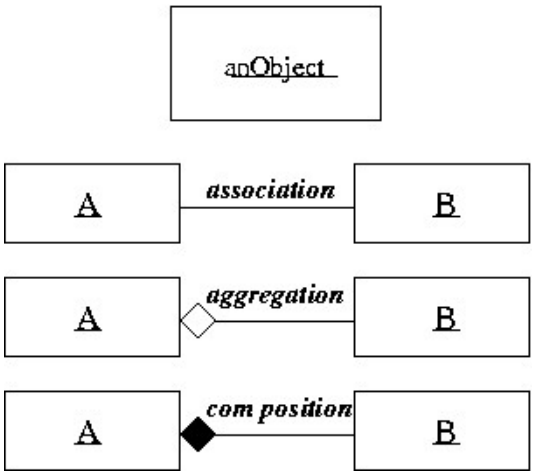

Fig. 1. Basic elements of an Object Diagram

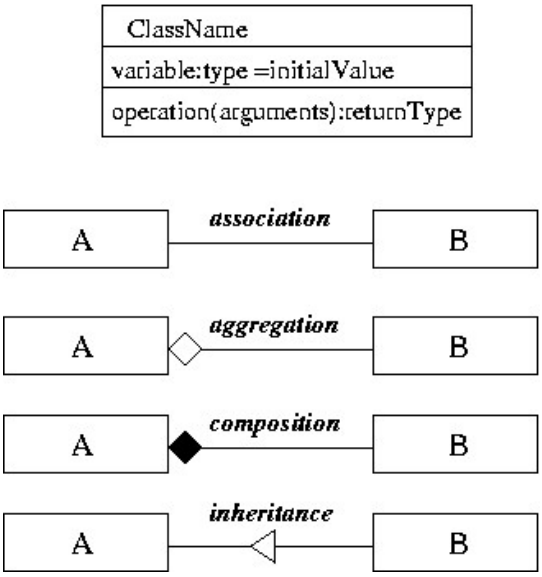

Fig. 2. Basic elements of a Class Diagram

Interaction diagrams illustrate how the instances of various classes communicate with each other. Each interaction represents a sequential flow of events though time. The flow may be set in motion by the end-user's actions, such as pressing a button or a slider, or by internal incidents, such as calls from another object. Thus, interactions depict how a set of objects communicate in order to provide a desired functionality. Figure 3 shows the basic elements of interaction diagrams [4]. Time runs from top to bottom. The names of objects are placed at the top of the diagram. Arrows represent messages from one 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. Also, some pre- and post-conditions can be added above the interaction diagram, considering the description, assumption(s) and result(s) of the interaction. Finally, exceptions to the normal sequence of events are also added below the interaction diagram [7].

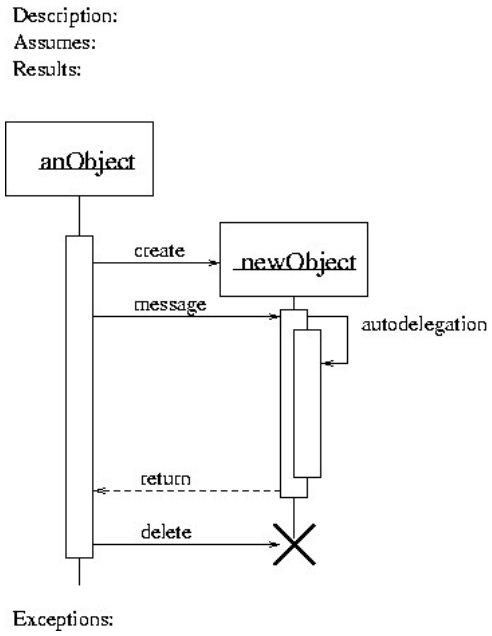

Fig. 3. Notation for an Interaction Diagram

3.2 Process and Artifacts

Figure 4 shows the five phases of the Practical Design Method, namely *Requirements Analysis*, *System Design*, *Detailed Design*, *Programming*, and *Testing*. Although the phases are listed sequentially, an iterative approach can be adopted (as it is discussed later in section 5). Requirements Analysis refers to collect all the requirements that there are for the system to be designed. The System Design aims at modeling the main components or objects of the system, as well as developing the operations of the system as a whole. In the Detailed Design phase, the products of the System Design phase are transformed into a form that can be programmed. Both System Design and Detailed Design phases illustrate 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 Testing phase tests the system against the requirements.

For every phase in this design method, there are two parallel descriptions of the system: the *form*, which uses object and class diagrams to show the static properties of the software system, and the *function*, which uses operation descriptions and interaction diagrams to show the functional properties of the software system. These two paths are related to each other, and they both aim at a code that has been tested, as shown in Figure 4.

4 Phases of the Practical Design Method

4.1 Requirements Analysis

The process of developing a software system starts with the requirements analysis. The purpose of this phase is to communicate with end-users, documenting and analyzing their requirements. The requirements are divided into *run-time* and *build-time* requirements. Run-time requirements are short stories describing the use of the software system, and they can be modeled through use cases [9] [13]. Build-time requirements are characteristics that the software system ought to meet, which are documented as a numbered list. All requirements should be described as exact and measurable as possible.

This paper uses a simple example to illustrate the phases of the Practical Design Method. The example describes the design of an application that allows elementary school students to compose simple musical tunes. This toy application is called “Elementary Composer” [7]. Figure 5 shows the use cases and Figure 6 describes some built-time requirements of this small application.

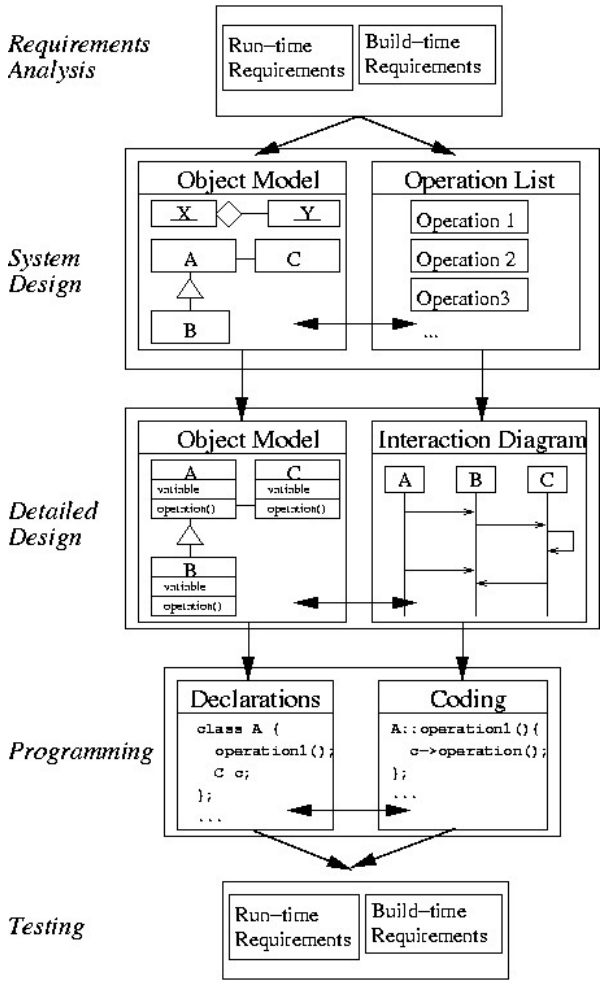

Fig. 4. Phases of the Practical Design Method

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 quater 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.
Tha 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. 5. Use cases as run-time requirements for the "Elementary Composer" example application

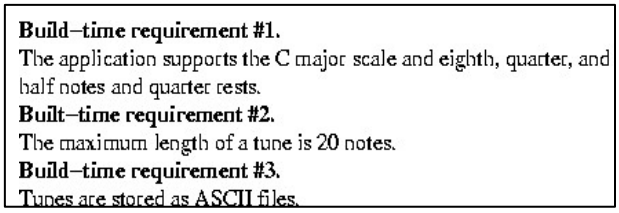

Fig. 6. Build-time requirements for the "Elementary Composer" example application

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 run-time and build-time requirements have been documented and agreed on with the customer, they form the *Requirements Specification* document, which is the basis for the following phase of System Design. In each step, phase products must be checked against the requirements specification. At the end, the requirements form the basic test case set for the software system testing.

4.2 System Design

The purpose of System Design is to understand and link the problem domain with elements of the software system to be implemented. This "linking" activity is the essence of design. As stated before, the System Design phase is based on the collected requirements, and the phase includes descriptions of *form* and *function*. The form aims at specifying all key subsystems and object components related within the system that is being designed. It produces an *object model* that documents the subsystems and object components of the system. The function defines the operations that the user performs with the system, representing the system as a black box. It models only the external functionality of the system, produces an *operation list*, and considers how subsystems and object components should act in order to carry out such an operation list. The final system must support the performance of all operations in the list. Both object model and operation list define the *System Specification* document.

Although the object model and the operation list are separate models, operations include and use the concepts defined by and included in the object model. Still, the idea is not to push back into the object model by guessing operations for objects and clases. Thus, the object model includes few operations; typically, only variables are considered.

There is a lot of information about system or architectural design, in which a basic organization of subsystems or objects can be found and refined. Still, it seems that System Design is a difficult phase to master. Therefore, it is suggested that system design be done in teams that consist of the object modeling developers and the users of the problem domain. While the users, such as the customers themselves, can contribute and participate in system design, object modeling developers should focus on refining knowledge and drawing useful object models.

In this phase, a *system architecture* must be selected or developed [2]. A system architecture is a form and function description at the system level of a known design solution. It forms the basis of future design decisions. For example, the Model-View-Controller architectural pattern [2][10] could be selected. Most modern programming environments provide help for developing system architectures upon which the design decisions can be built. Successful design requires good knowledge of the selected environment programming language, operating system, hardware, and other issues which affect the final coding.

The System Design phase results on the System Specification document, which encompasses the object model and the operation list by describing a *first version* of the system. Figure 7 and Figure 8 show respectively the object diagram and the

class diagram of the very first version for the Elementary Composer example [7]. This first version includes the objects and classes representing the windows of the user interface. The Practical Design Method presented here does not include a separate phase for the user interface specification. Typically, simple user interfaces can be designed by drawing them with one of the various user interface builders or application development environments. User interface prototypes can also be built with the help of real end-users.

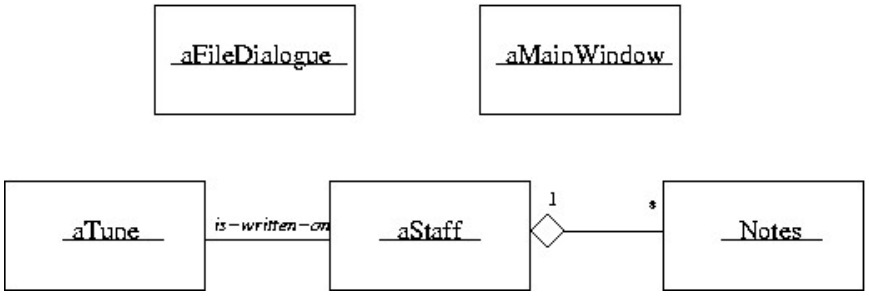

Fig. 7. Object diagram for the first version of the Elementary Composer example

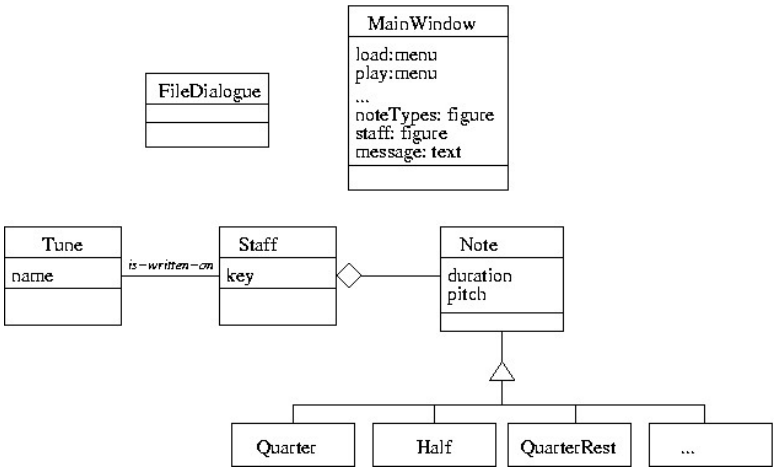

Fig. 8. Class diagram for the first version of the Elementary Composer example

The operation list is written based on the use cases. Figure 9 lists all the operations performed by the Elementary Composer [7]. 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. 9. Operations performed by the Elementary Composer application

4.3 Detailed Design

The purpose of the Detailed Design phase is to transform the object model and the operation list of the System Specification into a form that can be implemented in a programming language. While System Design concentrates on objects and functionality for both end-user and programmer, Detailed Design only focusses on objects and functions from the programmer's point of view. In fact, the first version described in the System Specification document is obviously incomplete, and it needs a lot of refining and tuning. Refining is done by applying interaction diagrams systematically: every operation in the operation list is analized, and an interaction diagram is drawn for each operation using the proposed object model. By doing so, connections between the objects and classes are refined by adding operations, variables, and new classes.

As the System Design phase, the Detailed Design includes form and function descriptions, as shown in Figure 4. In the form description, each class is considered using the selected programming language. In the function description, each operation of the operation list is modeled as an interaction diagram, and considered to potentialy belong to a particular class. In the System Design phase, form and function are considered related but independent from each other. Here, form and function must converge. The interaction diagrams modify the object and class diagrams. Here, "modify" means adding implementation specific classes, changing class structures, and identifying operations and variables.

Basic design rules, such as those on how to manage user interfaces and how to handle databases, are typically dictated by the selected tools. The Elementary Composer example is implemented in Object Pascal, using Borland's Delphi programming environment [7]. In the Delphi environment, the user interface is 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 interface objects, thus allowing the communication with the end-user by providing the functionality of the application.

When drawing interaction diagrams, the responsabilities that objects have and how they function in practice are specified. An arrow starts from an object that calls the memeber 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 object and class diagrams, an interaction diagram is drawn to show 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 [7]. Before the operation can start, the user has to select a note type, and the type information is stored within the `MainWindow` object. The interaction diagram for the selection operation is not shown in this paper.

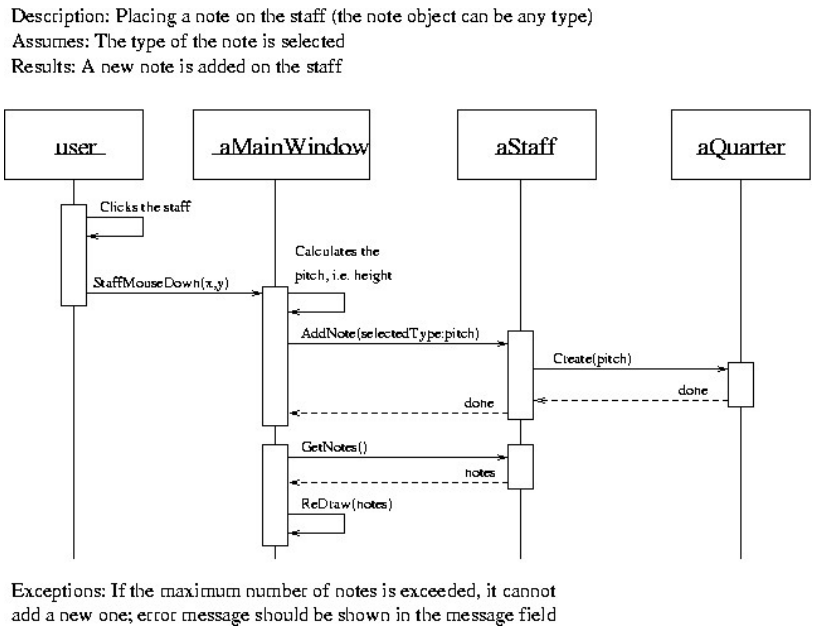

Fig. 10. Interaction diagram for "placing a note on the staff" operation

In Figure 10, as a first action, the user clicks the staff on the screen. This launches the `StaffMouseDown(x,y)` operation of the `MainWindow`. At first, this operation determines the pitch (the height of the note to be added) by calculating the position of the mouse relative to the staff. After this calculation, the `MainWindow` object calls the `AddNote(selectedType,pitch)` operation of the `Staff` object, which creates the note object. In this case, the type of note could be a Quarter note. The `AddNote(selectedType,pitch)`operation returns an acknowledge 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` object calls its own `ReDraw(notes)` operation.

Here, each interaction diagram specifies a set of operations, variables, and associations to be added into the object and class diagrams. For example, an arrow pointing to an object adds a member function to the class of the object in question. Similarly, a connection between two objects of the object diagram implies adding a connection between the corresponding classes into the class diagram. Nevertheless, the interaction diagrams do not show object communication in every detail. Instead, the interaction diagrams should be readable and simple. They should only depict the threads of execution as they are normally performed.

Figure 11 and Figure 12 show respectively the object diagram and the class diagram of a *second version*. The operations and connections suggested by the interaction diagram in Figure 10 are added. Based on the interaction diagram, operations have been added to various classes, and a new connection between `MainWindow` and `Staff` has been established.

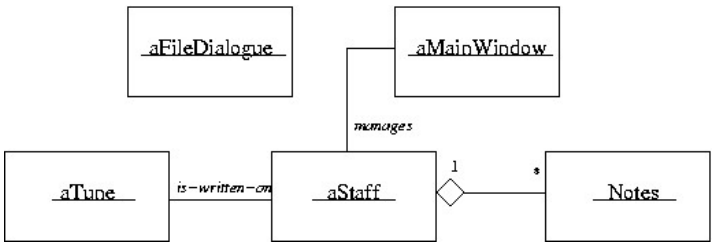

Fig. 11. Object diagram for a second version of the Elementary Composer

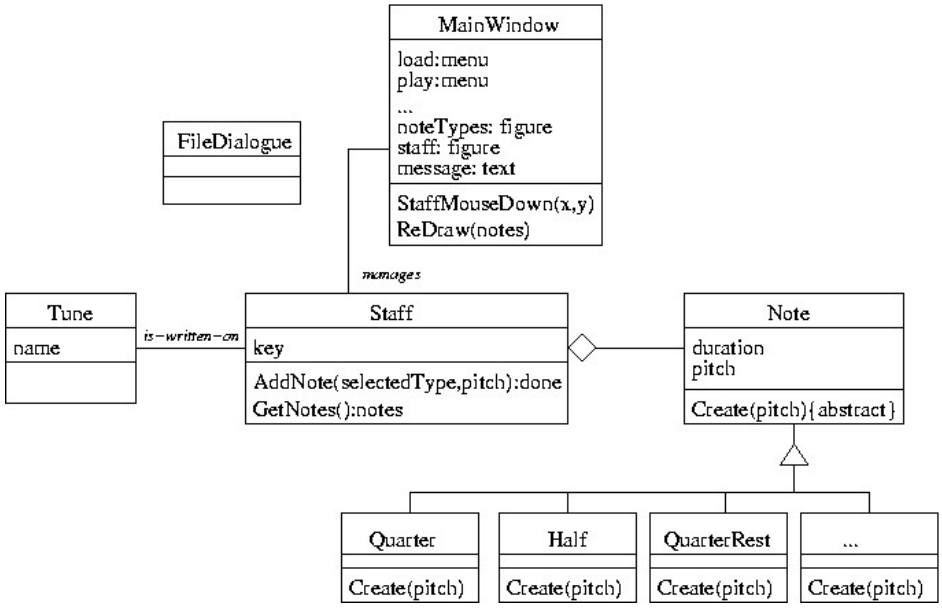

Fig. 12. Class diagram for a second version of the Elementary Composer

Now, Figure 13 shows how the objects cooperate with each other when the user wishes to hear a tune. Thus, this figure depicts the “playing a tune” operation [7].

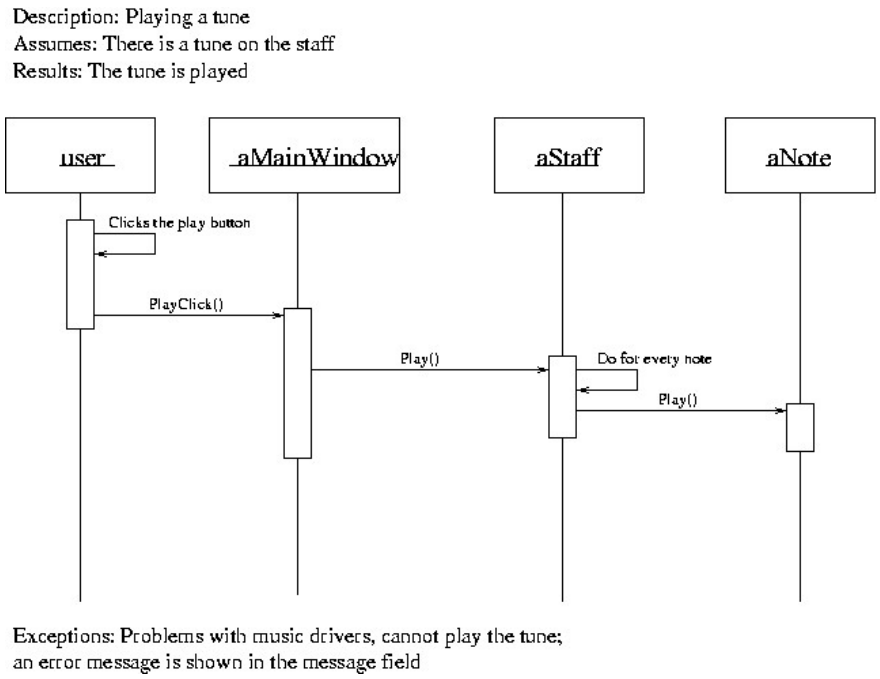

Fig. 13. Interaction diagram for “playing a tune” operation

Figure 14 presents a *third version* for the class diagram of the Elementary Composer, with the modifications suggested by the second interaction diagram. Notice that the object diagram for these modifications remains the same as shown in Figure 11.

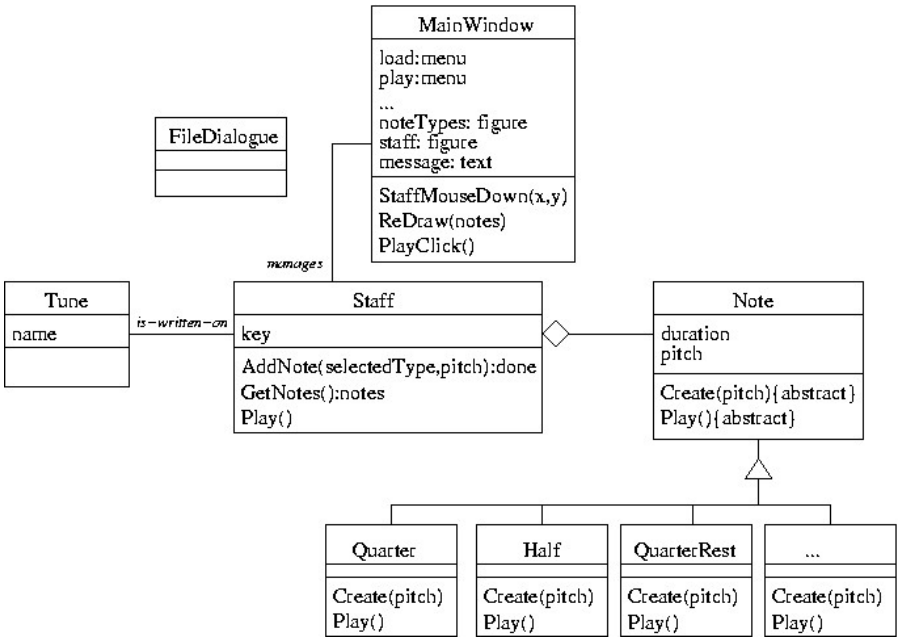

Fig. 14. Class diagram for a third version of the Elementary Composer

It is necessary to specify all the operations in the operation list as interaction diagrams, and refine the object and class diagrams accordingly. When drawing the interaction diagrams, it should be thinking about the final implementation, this is, how the software system will function in practice. When drawing interaction diagrams, it is often needed to add new design-specific objects and classes. It may also be the case of altering the object structure specified previously, or remove classes that are not involved in any interaction diagram. In any case, all changes should be justified and minimized, but all unnecessary changes to the System Design specification should be avoided. The goal is to design classes that are suitable for implementation.

4.4 Programming

The purpose of programming is to transform the class diagram and the interaction diagrams into code constructs in a programming language. The design phases have already modeled all classes of the system, as well as the communications between the instances of these classes. However, the inner functionality of individual objects has not being modeled during design. Instead, the programming of classes is based on the object, class, and interaction diagrams. Thus, while design concentrates on object structures, object interfaces, and cooperations between objects, programming deals with the inner functionality of individual objects.

The class declarations are based on the class diagram, and the coding of individual operations is based on the interaction diagrams. Figure 15 shows the declaration of the Staff class in Object Pascal [7]. This declaration is based on the class diagram in Figure 14.

```
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. 15. Declaration of the Staff class

Figure 16 shows the code of the StaffMouseDown operation of the MainWindow class. It is also written in Object Pascal, and based on the interaction diagram in Figure 10 [7].

```
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. 16. Declaration of the StaffMouseDown method of the MainWindow class

It is important that all classes, their interfaces, and the cooperation between objects have been modeled during the design phases. Based on such modeling, a programmer can concentrate on one class and one operation at a time. This is the reason why the Practical Design 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.

4.5 Testing

The purpose of testing is to find errors and ensure that the software behaves as planned. Therefore, testing is performed against the requirements. According to the Practical Design Method presented here, each run-time requirement is tested on the implemented system, and every build-time 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 run-time requirements. Such black-box testing can highlight, at least, the most severe errors.

5 Applying the Practical Design Method

The Practical Design Method attemps to introduce what is essential for software systems design. Thus, it is not a thorough description of every step or consideration which has to be taken during software design. Nevertheless, once the design method has been known and used within an organization, other practicalities about its application can be considered, such as using it within an iterative design process, the documentation and tools it uses, and its extension to be used in larger and more complex projects.

5.1 Iterative Software Design

The Practical Design Method can be applied as an iterative process. Systems are designed in cycles. Each cycle is composed of the phases of the design method as previously described in section 4. So, each cycle develops only a slice of the required functionality. Each new cycle builds on previous ones, so each cycle can learn from experiences gained during the previous cycles. After each cycle, it is possible to validate requirements by testing the implemented part of the system, and even by delivering it to the customers.

Figure 17 shows the phases of an iterative software design. The length of each slot represents the amount of work and time needed. The capture and analysis of requirements should be performed thoroughly and carefully already during the first cycle. This is how it is ensured that the right goals are considered for the following cycles. All available requirements should be collected before the design phases. So, all use cases that can be specified have to be written down.

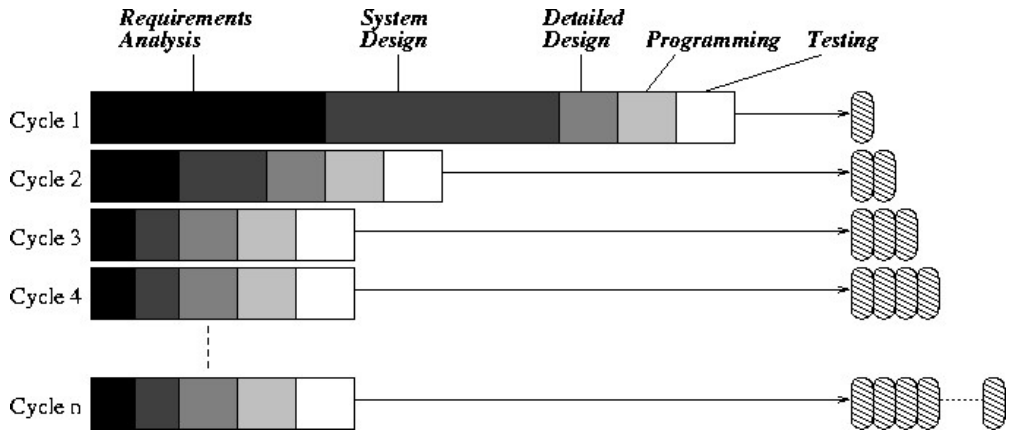

Fig. 17. Phases of iterative software design

Notice that during the first cycle System Design should be as complete as possible. Without a thorough System Design, it is easy to end up with design solutions that cannot be extended beyond the first iteration cycle. In addition to the System Design phase, it should be identified a 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 function from the end-user's point of view. Only the selected operations are transformed into the form of interaction diagrams during the design phases,

and programmed. Some objects and operations that may have been discovered already in the System Design phase are not implemented at all. They will be taken care of during the next cycles, with the rest of the operations.

It is essential and of main importance that the System Design phase is based on a solid system architecture. An adequate architecture allows to "plug-in" new classes, and implement new methods without the need to re-implement the results of the various iteration cycles. For example, the MVC architectural pattern [2] [10] forms an structure where new dialogues, objects, operations, and other elements can be added to the application skeleton built during the first cycle. Thus, each new iteration cycle builds on previous ones.

5.2 Documentation and Tools

The software design process must be visible. All the phases of the process must produce visual phase products that can be discussed. There are plenty of tools that support the 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 could be the right 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 diagrams and other models can be attached.

The software design process should produce concrete phase products, which can be organized as documents. The *Requirements Especification* document should include run-time and build-time requirements. The *System Especification* document should contain the object model and the operations list. Also, it can include windows and dialogues of the user interface. The *Detailed Design* documents should include the object and class diagrams, and all the interaction diagrams. It also should include all other design decisions, such as database selection, for example. The program itself is also a document. It is read both by the compiler and the maintainer of the application.

5.3 Extending the Design Method

The design 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 design method presented here, then missing phases and notations, problems in controlling the design process, and other similar issues should be documented and analyzed. Based on these experiences, the design method can be improved for further projects.

In terms of thoroughness, commercial design methods count with advantages and a number of extensions compared to the design method presented in this paper: requirements analysis uses more complete use case analysis; subsystem division and architectural design is considered as an integral part of the process; controlled user interface specification phases are included; testing extends to include three separate phases: module testing (which is a white-box testing phase for the implemented classes, processes and libraries), integration testing (which tests the cooperation between various subsystems), and system testing (which tests the system from the end-user's point of view). In addition, and among other things, the extension may introduce the use of Software Patterns [2][5] and usability evaluations to the design process.

6. Conclusions

When a company starts to design 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 of the pilot project, the company needs a simple but powerful design method for developing the software. Without a systematic approach, the company cannot maximize learning. *Ad hoc* hacking during the pilot project does not provide real knowledge for future

projects, although some details may be tackled. Thus, a simple and practical design method is needed to form the basis for improvement of the object-oriented practices of the company, and this design method should be used from the very first pilot project.

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

The Practical Design Method presented here is a trimmed down version of another design method, as it is suggested by Lilly [11]. The Practical Design Method does not introduce any new notation or phases. Instead, it uses tools already proposed by other design methods, such as OMT [14] and OOSE [9]. The design method introduced in this paper is based on concentrating on what seems to be common and essential in each phase. The Programming phase is illustrated with Object Pascal, which is an easy object-oriented language than other more complex languages such as C++. This is an important consideration suggested by Henderson-Sellers and Edwards [6].

The Practical Design Method does not address reuse as its main goal. Before any reuse can take place, the company needs to follow a systematic and repeatable design method. Activities to support reuse, such as identification of reusable components and modifications in class inheritance hierarchies, should be added to the design method once it is in operational use. Also, the design method here assumes 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.

There are risks in adopting a simple approach for software design. A design method that is too simple may fail in specifying the key properties of object systems. Still, the real problem with the methods available is the complexity on them. A design method needs to be simple and intuitive enough so that it is easy to notice the added value obtained by following it. If it is too complicated, the designers and programmers do not commit themselves to working with it, and a repeatable process cannot be achieved. Therefore, instead of tryng to cover full details of software design, the design method should first support only the most important phases. Only after application in practice, the method can be extended.

As the design method is adopted by an organization, it seems a good idea to have a design process improvement team within the organization. Such a team, or maybe an individual, is able to support the use of the selected design method, collecting the best practices, and improving the design method together with the rest of the members of the organization. The Practical Design Method presented in this paper forms a good basis for such improvements. It can be used as the backbone of the design projects, and it can be extended, for example, when moving up through the CMM levels.

References

1. **Booch, G., Rumbaugh, J., and Jacobson I.:** The Unified Modeling Language User Guide. Addison-Wesley, 1998.
2. **Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., and Stal, M.:** Pattern-Oriented Software Architecture. A System of Patterns. John Wiley & Sons, Ltd., 1996.
3. **Cockburn, A.R.:** “In Search of a Methodology”. Object Magazine, July-August 1994.
4. **Fowler, M**.: UML Distilled. Addison-Wesley Longman Inc., Reading MA., 1997.
5. **Gamma E., Helm, R., Johnson, R. and Vlissides, J.:** Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading MA., 1995.
6. **Henderson-Sellers, B. and Edwards, J.M.:** “Identifying Three Levels of O-O Methodologies”. Report of Object Analysis and Design, July-August 1994.
7. **Jaaksi, A.:** “A Method for Your First Object-Oriented Project. Fundamentals principles of O-O software”. In Object Expert, Vol. 2, No. 5, July/August 1997.

8. **Jacobson, I., Booch, G., and Rumbaugh, J.:** The Unified Software Development Process. Addison-Wesley Longman Inc., 1999.
9. **Jacobson, I., Christenson, M., Jonson, P., and Övergaard, G.:** Object-Oriented Software Engineering. A Case driven Approach. Addison-Wesley, Reading MA., 1992.
10. **Krasner, G.E. and Pope, S.T.:** "A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80". Journal of Object-Oriented Programming, August-September 1988.
11. **Lilly, S.:** "Planned Obsolecence". Object Magazine, September 1994.
12. **Paulk, M., Curtis, B., Chrissis, M., and Weber, C.:** "Capability Maturity Model, version 1.1" In IEEE Software, 10-4, 1993.
13. **Rumbaugh, J.:** "Getting Started, Using Use Case to Capture Requirements". In Journal of Object-Oriented Programming, September 1994.
14. **Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W.:** Object-Oriented Modeling and Design. Prentice-Hall, Englewood Cliffs, NJ., 1991.
15. **Salt J.E. and Rothery, R.:** Design for Electrical and Computer Engineers. John Wiley & Sons, Inc., 2002.

Jorge Luis Ortega Arjona. *He is a full-time lecturer at the Department of Mathematics, Faculty of Sciences, National Autonomous University of Mexico (UNAM). He obtained a BSc. in Electronic Engineering, as well as a MSc in Computer Science, at UNAM, and pursued a PhD from the University College London (UCL). His research interests include Software Architecture and Design, Software Patterns, Object-Oriented Programming and Parallel Processing.*