Exploratory Software Development with Class Libraries

Johannes Sametinger, Alois Stritzinger

Christian Doppler Labor für Software Engineering
Institut für Wirtschaftsinformatik
Johannes Kepler Universität Linz
A-4040 Linz, Austria

Abstract. Software development based on the classical software life-cycle proves inadequate for many ambitious projects. Exploratory software development is an alternative way of building software systems by eliminating deficiencies of the conventional software life cycle. Instead of exactly defining the various phases of the life cycle, exploratory software development takes small development steps, whereby a single step results in an extension or an improvement of the existing system.

The object-oriented programming paradigm has resulted in increased reuse of existing software components. Therefore, class libraries will become very important in the near future. Exploratory software development is very well suited to this situation and thus provides a major step forward in economically developing software systems.

In this paper we depict deficiencies of the classical software life cycle, present the exploratory software development strategy, and especially illustrate exploratory software development in combination with the reuse of class libraries.

1 Classical Software Life Cycle

Software is usually developed according to the classical software life-cycle. Various models for this life cycle do exist, but basically they are very similar (see [Boehm79, Pomberger 86]). According to the software life cycle the software development process is divided in well-defined phases. In general, each phase has to be finished before the next one can be started (see Fig. 1). This enforces a linear process, which implies that executable programs are available very late. Therefore, any misunderstandings between customers and developers remain hidden for a long time. Besides, any technical problems (e.g., an inefficient file system) cannot be perceived before the test phase. Usually modifications becoming necessary are very costly because they are so late.

The classical software life cycle presupposes static requirements and does not deal with incomplete and inconsistent specifications. For given and static specifications, software developers have to deliver a tailor-made design and a corresponding implementation. The better the implemented program fulfills the given requirements, the better was the work of the

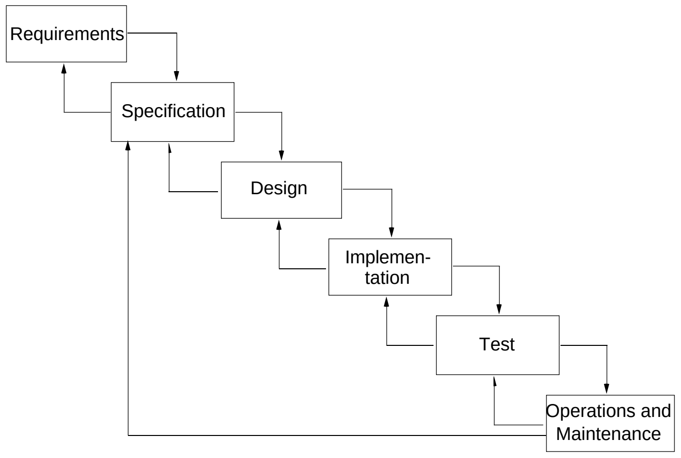

Fig. 1: Classical Software Life cycle

software developers. This approach is in contradiction to reality, because past experience has shown that programs need to be continuously modified and extended. This results in thousands of programmers being engaged with adapting existing software systems to new or changed requirements. Statistics even say that nowadays more time is spent on software maintenance than on software development (see e.g., [Gibson89]). This unsatisfactory situation is partly propagated by the classical software life cycle.

2 Exploratory Software Development

Recently the term *prototyping* has become a buzzword (see [Bischofberger91, Budde84]). The emphasis of prototyping is on the evaluation rather than on long-term use. Software prototypes very often implement the user interface of an application program in order to give potential users an early possibility to evaluate the usefulness and the proper design (of the user interface) of the product. This communication vehicle between developers and customers helps to avoid misunderstandings and usually improves the user interface considerably. However, software prototypes are not restricted to user interface aspects; they can be extended to the finished product step by step.

The term prototyping stems from industry, where prototypes are first models of a certain product. Such prototypes (e.g., cars) are used to investigate certain aspects of a product before it goes into production. As software is simply copied rather than produced in quantity, the term software prototype is somewhat misleading. Besides, this approach can be used not only at the beginning of software development but throughout the whole life cycle. For that reason we prefer the term *exploratory software development*. To begin with, exploratory software development means the production of software to meet the known requirements. Testing the product leads to more requirements and results in modifications and tests to fulfill them. This process is repeated until the developed software system performs satisfactory (see [Sandberg87]). Exploratory software development is a strategy that is best suited when an inherent goal of the project is to identify elusive requirements (specification), to

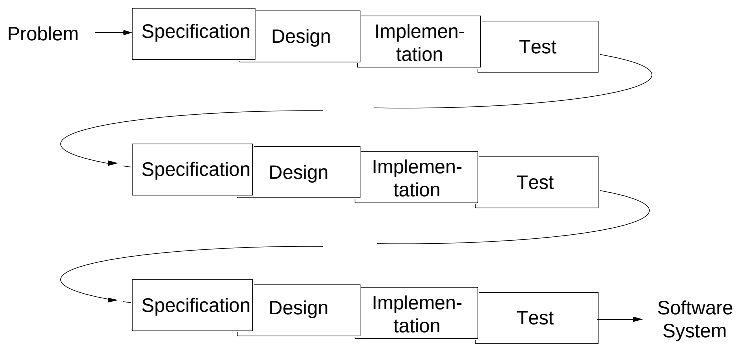

Fig. 2: Exploratory Software Development

establish a suitable system architecture (design), or to explore possible implementation techniques.

Exploratory software development involves repeatedly applying small steps. Each step results (ideally) in an improvement of the current program version until both the developer and the customer are satisfied with the result. Typically one step lasts several hours or even less (see Fig. 2).

When using exploratory software development, programmers have to work with utmost discipline. For example, extending the functionality of a system before its existing parts have reached a (preliminary) satisfactory condition is inexpedient. Additionally, programmers should be aware of writing all the code in a “quick and dirty” fashion, though sometimes it might be useful to temporarily use “quick and dirty” solutions.

The usefulness of exploratory software development emerges from the lack of alternatives in many situations. Both customers and developers not yet knowing exactly what they really want is a typical development situation. Programmers also might not know how to (best) solve certain (implementation) problems. In these cases it is appropriate to work with experimental versions of the software system. By experimenting both customers and developers can gain new insights into their problem domains and thus come closer to better solutions.

Another important justification for using exploratory software development is the increase in complexity of today’s software systems. High complexity makes it impossible for human reasoning to deal with all the problems in a linear way, as the classical software life cycle proposes.

Software can best be developed in an exploratory way whenever one or more of the following conditions hold:

- The specification is very vague. Customers are unable to clearly specify their wishes and needs.
- Critical design decisions cannot be made based on theoretical considerations.

- Software developers do not have enough experience with the implementation of similar systems (and the system to be developed is sufficiently complex).
- Programmers do not have (enough) experience in using the programming language or library. (It is impossible to gain familiarity with a class library without experimenting.)
- The system to be developed is too complex and too ambitious to be built linearly.

In our opinion, about half of all projects satisfy one or more of the conditions mentioned above and thus are candidates for exploratory development. The main advantages of exploratory software development are:

- Experimental program versions are excellent vehicles for communication among developers and customers.
- The exploratory approach reduces risks because typically problems are perceived earlier than in the classical software life cycle.
- Stepwise developed programs are better structured and better suited for modifications and extensions because programmers are forced to permanently modify and extend the current version of the software system to be developed. This encourages and trains programmers to write better modifiable code.
- As modifying the system is part of the work being permanently done, it is easier to take new ideas into consideration. The statement: “The next time I would try a wholly different approach!” is more seldom among exploratory programmers.
- Programmers are strongly motivated by working on an executable program rather than writing specifications and design papers for a long time without having an executable program.

Unfortunately, there are also some disadvantages:

- Exploratory development in large teams is possible only when the software system can be clearly separated into various parts.
- It is difficult to estimate the duration and the costs of a certain project. New estimation methods have to be found for this purpose.
- Programmers have to be well trained and to work with discipline. This is extremely necessary in exploratory software development because otherwise the resulting programs are not easily modified ore extended.
- Documentation gets lost in the shuffle.
- Version control and backtracking need to be supported (by tools).

In commercial software projects these disadvantages may be too hard. In order to get estimates of the cost and the duration of a project, we suggest making a rudimentary specification and an initial design according to the classical software life cycle and applying the exploratory approach in the next steps only. This makes it possible to divide a project into small and easily surveyed parts that can be processed by small programming teams.

3 Reusable Class Libraries and Application Frameworks

Conventional libraries, toolboxes, drawing routines, etc. offer fixed functionality at a higher abstraction level than bare programming languages. In the design of the software system the designers have to consider the interfaces of the given components carefully and have to use the provided functions in an appropriate manner. Usually it is not a major problem to build a system upon such libraries when their functions and components are not strongly interrelated. This holds for simple user interface components, data containers, and mathematical and graphical operations.

When working with application frameworks, which define the core structure of the overall application, the designers cannot develop an architecture top-down. In this case the architecture is already predefined to a certain degree by the set of related framework classes which anticipate very early design decisions. The job of the designers is to append the application-specific functionality at appropriate places in the framework. The more powerful and extensive the framework is, the more design decisions are already anticipated in the provided classes.

Commercial applications usually do not use domain-specific interaction techniques or sophisticated algorithms. For such applications classical design methods become superfluous. Although complex software systems could never be designed by means of applying classical techniques and methods such as stepwise refinement or the Jackson System Development Method (see [Cameron89]) alone, application frameworks make these aids less useful. This does not imply that classical techniques will become obsolete as a consequence of frameworks, but their use will be restricted to certain domain-specific components.

Another drawback of classical design methods stems from the fact that applications made from frameworks are implemented in an object-oriented way. Object-oriented systems cannot be designed adequately by means of classical methods. A considerable number of software engineering scientists see the need for a new or modified design method to overcome the current dilemma. A rapidly increasing flood of articles and books about object-oriented design methods, e.g., [Booch86, Coad90, Rumbaugh91], mirrors the expectations of the unhappy software industry.

4 Exploratory Development Approach with Class Libraries

Powerful and well-structured class libraries are a crucial advantage for exploratory software development. The quality and extent of the library used are often more important than the power of the programming language or the development tools.

The exploratory approach has proven its excellence particularly in the development of highly interactive applications with graphical user interfaces. Below we will describe the various tasks that are typical in exploratory software development with class libraries. In general these tasks are seldom completed at once. Usually one does just a portion of a certain task; the next step is taken at the next iteration of the cycle. Furthermore, one should keep in mind that not everything can be done right the first time. But even when information is missing to make a sound design decision, one should not hesitate too much. Experimentation and exploration often lead to better solutions than intense analytical studies. The steps of the exploratory development approach are as follows (see also [Stritzinger92]):

Step 1:

Start with the design of the user interface in a prototyping-oriented way. Concentrate on the essentials first. Whenever some parts of the interface are unclear, try a rudimentary design.

Step 2:

Try to identify classes for the implementation of the user interface components. An extensive class library should offer a lot of support in this respect. Typical classes include: Window, Menu, View, TextView, ListView, and control elements like Button and Scrollbar. If you cannot find exactly what you are looking for, search for classes that already implement part of the desired functionality. Inheriting is most often cheaper than implementing.

Step 3:

Try to identify classes that describe important objects in your problem domain. These classes often correspond to object categories of the real world (employee, car, etc.). Although it is not as likely as with user interface classes, there is still a chance to find classes in the library from which you can inherit. If objects in your program have a close correspondence to real-world objects, slight changes in the real world will just cause slight changes in the program. All objects that describe application-specific data should be connected somehow. This complex object web is usually called the *model*. Relationships among model objects can either be established by application-specific compound objects (faculty, assemblyLine, etc.) or by general-purpose collection objects (queue, tree, etc.). The whole model should be accessible by a single (or a small number of) reference(s). If there are objects that share a lot of commonalties but differ in some respects, the commonalties should be described collectively (factored into a common superclass). In many cases abstract classes are rather useful. Abstract classes (e.g., GraphicShape) are classes that do not have instances; they just serve for factoring commonalties out of their subclasses. The more complex the problem is, the more imaginary classes have to be invented. Finding appropriate imaginary classes is a very difficult job that requires some experience. Fortunately, you can find such classes incrementally.

Step 4:

Identify relevant object states for all classes. Object attributes that carry state information are (usually) modeled as instance variables of the corresponding class. Redundancy among instance variables should be avoided.

Step 5:

Think about the messages (operations) your objects should respond to. Each instance variable has to be addressed; i.e., each variable must get a value and must be accessible somehow. The semantics of each message should be clearly describable. Messages should be as powerful as possible, but as flexible as necessary.

Step 6:

Implement a method for each message. Do not duplicate code from superclasses; send messages to invoke the overridden method instead. Extensive methods should be split into several, possibly private methods.

The above steps are often performed in a non-sequential way. For instance, it may happen that while implementing a method the need for an additional instance variable arises. Simultaneous development of various small life cycles is typical for the reuse of class libraries and is also called a *cluster model* (see [Meyer88], [Pree91]).

It is always advisable to define classes somewhat more generally than actually necessary. Modifying and extending existing code is typical in the exploratory approach. The more general classes are, the less widespread is the impact of changes and extensions.

5 Conclusion and Outlook

In summary, we claim that an exploratory, object-oriented development approach together with application frameworks is the most productive way to develop highly interactive applications with high quality standards. The problems in designing complex systems are rather a symptom of an insufficient strategy than a lack of methods. Innovative and sophisticated software systems can never be developed in a linear process of applying recipes. Similar to other high-tech products, knowledge, skills, experience and motivation play a crucial role in the successful realization of ideas.

In our opinion, one of the strongest drawbacks of object-oriented software development is the huge complexity of many widespread class libraries and application frameworks. This complexity, together with the manifold structuring options of object-oriented programming, make extremely high demands on programmers – even with an exploratory approach. Many programmers in the field are unable to take advantage of these powerful techniques. Therefore software engineering experts are called upon to develop tools that permit less experienced programmers to utilize the advantages of object-oriented programming with class libraries by helping to master the complexity and by supporting the comprehension process (see [Sametinger90] for an example).

A first step in the right direction is so-called interface builders. By means of interface builders construction of complex user interfaces can be done in a simple, interactive way by directly manipulating interface components. 4th generation systems form another possibility for a quick development of applications at a high level of abstraction. The drawback of 4th generation systems is often the connection between user interface and database, which usually have to be programmed with a rather conventional programming language. The developers are confronted with a huge gap in the abstraction level whenever the built-in functionality is not sufficient. Furthermore, only a minority of contemporary 4th generation systems are based on the object-oriented paradigm.

The goal of a thoroughly seamless development process at a very high abstraction level could be reached by a kind of tool (or tool set) which could be called *application builder* or

5th generation system. Such a system should support interactive, graphical construction of user interfaces and (external and internal) data models. In addition, a 5th generation system should offer the opportunity to combine predefined, reusable and user-defined building blocks in a comfortable, yet flexible and preferably visual way.

Unfortunately, such 5th generation systems are not available yet. But there is a good chance that mechanisms and tools will be developed which can fulfill the goal of a thoroughly seamless development process at a high abstraction level. Then object-oriented programming with extensive libraries will become a widespread technology available to almost everybody.

6 References

1. Bischofberger W., Kolb D., Pomberger G., Pree W., Schlemm H.: Prototyping-Oriented Software Development - Concepts and Tools, Structured Programming, Vol. 12, No. 1, New York, 1991
2. Boehm B., W.: Software Engineering, in Classics in Software Engineering, Yourdon N.E. Editor, pp. 325-361, Yourdon Press, 1979.
3. Booch G.: Object-Oriented Development, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986.
4. Budde R., et al (Editors): Approaches to Prototyping, Springer-Verlag, 1984.
5. Cameron J.: JSP & JSD: The Jackson Approach to Software Development, IEEE Computer Society Press, 1989.
6. Coad P., Yourdon E.; Object-Oriented Analysis, Yourdon Press Computing Series, Prentice Hall, 1990.
7. Gibson V.R., Senn J.A.: System Structure and Software Maintenance Performance, Communications of the ACM, Vol. 32, No. 3, pp. 347-358, 1989.
8. Meyer B.: Object-Oriented Software Construction, Prentice Hall, 1988.
9. Pomberger G.: Software Engineering and Modula-2, Prentice Hall, 1986.
10. Pree W.: Object-Oriented Software Development Based on Clusters: Concepts, Consequences and Examples, TOOLs Pacific (Technology of Object-Oriented Languages and Systems), pp. 111-117, 1991.
11. Rumbaugh J., et al: Object-Oriented Modeling and Design, Prentice Hall, 1991.
12. Sametinger J.: A Tool for the Maintenance of C++ Programs, Proceedings of the Conference on Software Maintenance, San Diego, CA, pp. 54-59, 1990.
13. Sandberg D.W.: Smalltalk and Exploratory Programming, ACM Sigplan Notices, Vol. 22, No. 10, 1987.
14. Stritzinger A.: Reusable Software Components and Application Frameworks—Concepts, Design Principles and Implications, to be published in VWGÖ, Vienna, 1992.