Journal of Computer Science 1 (1): 76-82, 2005
ISSN 1549-3636

Y: A New Component-Based Software Life Cycle Model

Luiz Fernando Capretz
Department of Electrical and Computer Engineering, University of Western Ontario
London, Ontario, N6G 1H1, Canada

Abstract: With the need to produce ever larger and more complex software systems, the use of reusable components has become increasingly imperative. Of the many existing and proposed techniques for software development, it seems clear that component-based software development will be at the forefront of new approaches to the production of software systems and holds the promise of substantially enhancing the software production and maintenance process. Attempts to rationalize component-based development have to recognize that the construction of a software system is a complex multifaceted activity that involves domain engineering, frame working, assembling, archiving and design of software components. These activities, among others, are encompassed by a software life cycle, named the Y model, put forward in this study. The Y model provides guidance for the major phases to be followed under its umbrella.

Key words: COTS, Software Life Cycle Model, Software Process, Software Reusability, Component-Based Software Development

INTRODUCTION

It has been claimed that the component-based software development promotes reusability, improves software quality and increases software engineers' productivity. A component is a self-contained piece of software that provides clear functionality, has open interfaces and offers plug-and-play services. Component-based software development is expected to be at the forefront of new approaches to the construction of large and complex software systems. The idea gained its real momentum after COM+ [1] from Microsoft, Enterprise JavaBeans [2] from SUN, IBM Component Broker [3] and CORBA [4] have made their way among mainstream software technologies [5]. Additionally, incremental delivery of software features or platforms that comprise a software product line is expected to be at the forefront of software development in the next few years, therefore component-based software engineering has broad implications for how software engineers acquire, build and maintain software systems [6]. Thus, we should see dramatic changes in designers' primary roles and required skills for software development in the near future.

As software is growing increasingly complex, so too is the effort required to produce it, on this account numerous software life cycle models have been proposed. Their main utility is to identify and arrange the phases and stages involved in software development and evolution, so it is appropriate to generally examine different software life cycle models and point out their strengths and weaknesses before an alternative one is put forward.

The Waterfall model [7] has been long used by software engineers and has become the most prevalent software life cycle model. This model initially attempts to identify phases within software development as a linear series of actions, each of which must be completed before the next is commenced. The Waterfall model is marked by the apparently neat, concise and logical ordering of the series of obvious phases, which must be followed in order to obtain the final software product. Refinements to this model consider that completion is seldom and that iteration back to a previous stage is likely to happen, but it takes no account of bottom-up development and prototyping.

The Spiral model [8] makes software development more flexible and has been proposed mainly to speed up software development through prototyping. Prototyping is the process of building an incomplete piece of software that exhibits some of the most relevant aspects of the final software system. Prototyping provides constructive feedback to designers and potential users so that the system requirements can be clarified and refined early during software development. Evolutionary prototypes provide incremental software development, so that software systems may be gradually developed and tested, allowing major errors to be exposed and corrected early, which means that they are often cheaper to fix, but without effective management to control iterations, this process can degenerate into uncontrollable hacking.

Extreme Programming [9] is a deliberate and disciplined approach to software development, which is aimed to solve customer requirement change problem. The methodology is designed to deliver the software

your customer needs when it is needed. It is based on four essential values: communication, simplicity, feedback and courage. Unlike other traditional software development models, such as Waterfall, which conducts analysis, design, implementation and testing once in its long development cycle, or Spiral model, which has shorter, iterative development cycles, extreme programming is blending all these activities, a little at a time, throughout the entire software development life process.

A growing number of companies in the software industry – including Microsoft – are following a process that iterates among design, building components, testing and getting feedback from customers as the product evolves. Many companies also ship preliminary versions of their products, incrementally adding features of functionality over time in various product releases [10]. Microsoft is different in the sense that it has introduced a structured hacker-like approach in the development of large-scale software products; projects remain under control because teams of designers and testers frequently integrate and stabilize their improvements. It is truly an example of how to take advantage of the exploding demand for PC software and an effective way to deliver products to a "hungry" market. Its competitive strategy revolves around identifying mass markets quickly, introducing a product that is good enough to dominate the market rather than waiting until the product is perfect, then upgrading the product continuously.

The Twin Peaks model [11] also argues for a concurrent, iterative development of requirements and architecture during software development. It presents a partial and simplified version of the Spiral model that illustrates the distinct, yet intertwined activities of requirements engineering and architectural design. This model allows incremental development with the consequent management of risks, compromises architectural choices to accommodate existing commercial of-the-shelf software (COTS) solutions and because the model focuses on finer-grain development, it is more receptive to rapid changes as they occur. However, one of the main shortcomings of all these models is that none of them explicitly encourages reusability along all their phases. Therefore, a software life cycle model that emphasizes the importance of component reuse during software development is still in demand.

The Y Software Life Cycle Model: The Y model, (Fig. 1), has been proposed as a viable alternative to address software reusability during component-based software production. The creation of software is characterized by change and instability, hence the diagrammatic representation of the Y model considers overlapping and iteration where appropriate. Although the main phases may overlap each other and iteration is allowed, the planned phases are: domain engineering, frameworking, assembly, archiving, system analysis, design, implementation, testing, deployment and maintenance.

The main characteristic of this software life cycle model is the emphasis on reusability during software creation and evolution and the production of potentially reusable components that are meant to be useful in future software projects. Reusability implies the use of composition techniques during software development; this is achieved by initially selecting reusable components and assembling them, or by adapting the software to a point where it is possible to pick out components from a reusable library. It is also achieved through inheritance, which is naturally supported by an object-oriented approach [12]. Reusability within this life cycle is smoother and more effective than within the traditional models because it integrates at its core the concern for reuse and the mechanisms to achieve it.

Domain Engineering: Domain engineering is a process of analyzing an application domain in order to identify areas of commonality and ways to describe it using a uniform vocabulary. Thus, domain engineering is an activity that should be carried out at the beginning of software specification if reuse is to be considered. As domain engineering can yield an initial set of vocabulary reflecting the main conceptual entities within an application domain, essential properties of that domain are captured and initial candidates for reusable components emerge. To illustrate, a process control system for a chemical plant is concerned with vessels, pipes and valves of that plant, as well as the flow of liquid and gases, the temperature and pressure at various points in that plant. A payroll system is concerned with employees, the pay they earn, the tax they owe and the holidays they are entitled to. These real-world entities and interrelationships are likely to become part of the vocabulary for these application domains.

User needs, software requirements, provided functionality, objectives and constraints of the system are very much of interest during the system analysis and domain engineering phases. Thus, it is important to understand the real-world application and an abstract model of that application should be depicted. Therefore, the boundary between system analysis and domain engineering may at times seems fuzzy because identifying key abstractions in the application domain may be viewed as part of system analysis or domain engineering. Nevertheless, at this level, domain engineering is also concerned with the identification of potentially reusable components.

Frameworking: A framework could be viewed as a generic structure that provides a skeleton for producing software in a certain application domain. Frameworking attempts to identify components and establish interrelationships perceived important within the

application domain. Such identification of components may arise from the well-known functionality common to that application domain, usually in the form of semantic relationships between components. Consider, for example, the application domain of airline reservation systems; typical entities of these systems are: seats, flights, crews and passengers; and interrelationships can be: reserve a seat, assign a crew to a flight, schedule a flight and so on. So, there are important relationships among these entities, which can be organized into a framework according to their semantic meaning in that application domain.

When performing frameworking, the software engineer might have a sketchy idea about candidate components for reuse. On the other hand, as frameworks comprise sets of components that express a design for a family of related applications, it is sometimes beneficial to change the developing software, so that an available framework fits in, resulting in a tremendous gain in productivity. Building and tailoring software from frameworks is faster and easier than starting from scratch, although frameworks will not be as generally useful outside their application domain because they contain domain-dependent components. Within the proposed life cycle model, the main result of the frameworking phase should be the reuse of software components already developed and the classification of components to form new frameworks. Instead of focusing on the individual application, the goal is to produce workbenches containing software components and generic application frameworks that characterize the software systems in a particular application domain.

Assembly: It focuses on selecting a collection of reusable components or frameworks from specific application domains. There are differences in the mechanisms used to achieve reusability when different kinds of reusable components are involved. The most basic software components are often reused by composition, which can be seen as a process of building a piece of software from elementary self-contained components; although reusability is naturally accomplished by reusing classes through inheritance during object-oriented development, in such case, it takes place by specialization and generalization of commonalities among classes. This phase is usually akin to sifting through a junkyard of books rather than visiting a library.

Archiving: Reusability not only involves reusing existing components in a new software system but also producing components meant for reuse. When a software system has been developed, the software engineer may realize that some components can be generalized for potential reuse in the future. An important consideration in the quest of reusability is how to make a potentially reusable component available to future projects. Archiving should reflect the activities involving cataloging and storage of components. The component must be understandable, well written and well documented. Additionally, extensive cross-referencing is necessary.

Not all components are created equal, they differ in complexity, scope (i.e. GUI, service or domain-oriented components) and levels of functionality. This differentiation among components makes it difficult to create a single database of software assets. Placing such an argument into component technology it produces the following important observation: several interconnected reusable components are more effective than a single universal library of components. Therefore, rather than creating a single library as a centralized repository of components, a better strategy is the development of specific frameworks for certain application domains.

System Analysis: The system analysis phase emphasizes identification of high-level components in a real-world application and decomposition of the software system. The system analysis phase demands the systems analyst to:

* Study the application and its constraints.
* Understand the essential features of the system.
* Understand the requirements expected to be satisfied by the software system.
* Create an abstract model of the application in which these requirements are met.

The main product of the system analysis phase is a graphical or textual description (informal or formal) of an abstract model of the application. At this stage, the services delivered by a software system help figure out its subsystems and major components. However, as compared to functional decomposition, this phase is not concerned with the details of the components. During this phase, the abstract model of the application comprising high-level abstractions of software components is better understood.

Design: Design is an exploratory process. When designers face an application, they should not ask “how do I work out a solution to this problem”? Instead, they should ask, “where are the components that I can directly or indirectly reuse to solve this problem”? At this point, they should be able to examine a reusable library and to select components that closely match the entities necessary to build the software. The designer looks for components trying out a variety of schemes in order to discover the most natural and reasonable way to refine the software solution.

There has been a tendency to present software design in such a manner that it looks easy to do. Nevertheless, in the design of large and complex software, identification of key components is likely to take some time. Repetitions are not unusual, since a good design usually takes several iterations. The number of iterations also depends on the designer's insight, experience and

knowledge about the application domain as it is discussed in the next section. The design process should stop when the key generic abstractions and the software behavior are detailed enough to be translated into a programming language. Hence, the design stage generates the templates for the implementation phase.

Implementation: The implementation phase is characterized by the translation of a design model into a programming language. In this phase the major tasks involve the implementation of components, in order to fulfill the required software functionality. Implementing a component requires defining the data structures and corresponding algorithms to provide the overall software services. The best strategy is to isolate a component and decide whether an available match can be reused, or if it has to be implemented from scratch.
The component must be easily configurable or adaptable for different uses, either in original or in modified form, which means that developing reusable components is considerable more difficult and involves much greater expense then producing ordinary components, although it may still be worth the investment over the longer term, after a sufficiently broad reusable assets are created. Some components picked out during the implementation phase should undergo further refinements, e.g. treatment of exceptional conditions and verification, until they become generic and robust enough to be placed in a reusable library. This surely adds an overhead to software construction, which is more than compensated for by the long term savings when such components are reused in future projects.

Testing: Once software components are implemented, it is time to test them. During the testing phase is not the first time when faults occur, they can be carried through from the system analysis and design phases. But testing is focused in finding faults and there are many ways to make testing efforts more efficient and effective. Testing of component-based software is best viewed as two distinct activities: the testing of the component as a unit and the testing of the assembled system.
In developing a large software system, testing usually involves several stages. First, each component is tested on its own, isolated from the other components in the system. Such testing, known as component test or unit test verifies that the component functions properly with the types of input expected based on the component's design. There are several techniques that can be used throughout this process such as white-box, black-box, code inspection, walkthroughs, formal proof and so on.
Integration testing is the process of verifying that the system components work together as described in the design specification. After a collection of components has been unit-tested, the next step is ensuring that the interfaces among the components are well defined. Once it is assured that the information is passed among components in accordance with the design, system test should be performed to guarantee that the desired functionality is provided. Depending on the profile of the system, further tests should be done, like performance tests, acceptance tests, quality tests, safety-critical tests and a final installation test. The final result of this phase is a functioning system that can be prepared for deployment.

Deployment: This is almost the end of system development, now the system is ready to be presented to the customer. Nevertheless, deployment involves more than putting the system into place, it is the time when users should be helped to understand and feel comfortable with the software. If deployment is not successful, users will not make the most of the system and may be unhappy with its performance. In either case, users will not be as productive or effective as they could be and the care taken to build a high-quality system is put in jeopardy.
The two key issues to successful transfer from the developer to the user are documentation and training, which should be integrated with the software. As the system is developed, software engineers should plan and come up with aids that help users learn about the system, such as on-line help. Accompanying the system is documentation and manuals to which users refer for problem solving, troubleshooting or further information. The quality and type of documentation can be critical, not only to training, but also to the success of the system. Training for users and operators is based primarily on major system functionality; there is no need to be aware of the system's internal operation. Therefore, system deployment should be considered with more care and professionalism than it has been usually dealt with.
In addition, product flexibility is the new anthem of the software marketplace and software family fulfils the promise of tailor-made systems that are delivered quickly, at low costs, built specifically for the needs of particular customers and market segment. This requires constant improvement, upgrading and releases of new versions of a software system that is preferably compatible with old versions.

Maintenance: Many software engineers wrongly assume that once a system is delivered their problems are over. A system life does not end with deployment. Software is normally subject to continuing changes after it is built, when it is operational. Thus the efforts turn now to the challenge of maintaining a continually evolving system. During software maintenance, changes are introduced to a software system. Such changes are not meant only for correcting errors occurred in the operational software; these changes may be also for improving, updating the system to anticipate

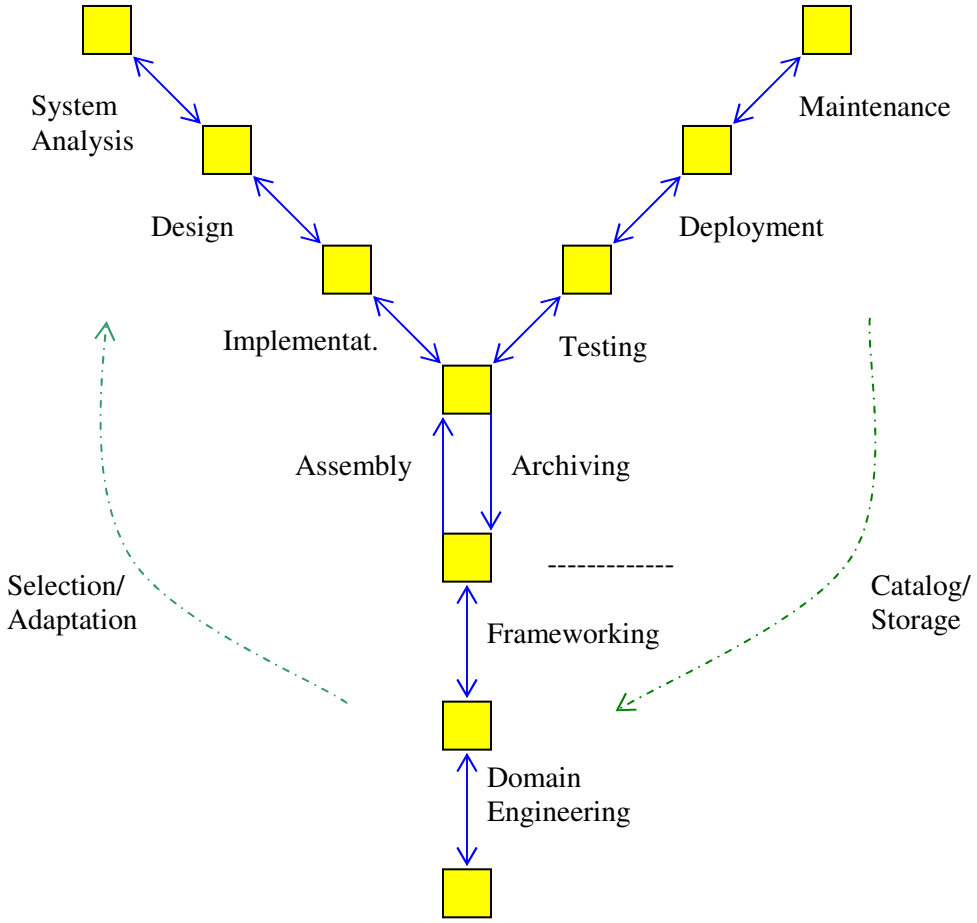

Fig 1: he Y Model for Component-Based Software Development

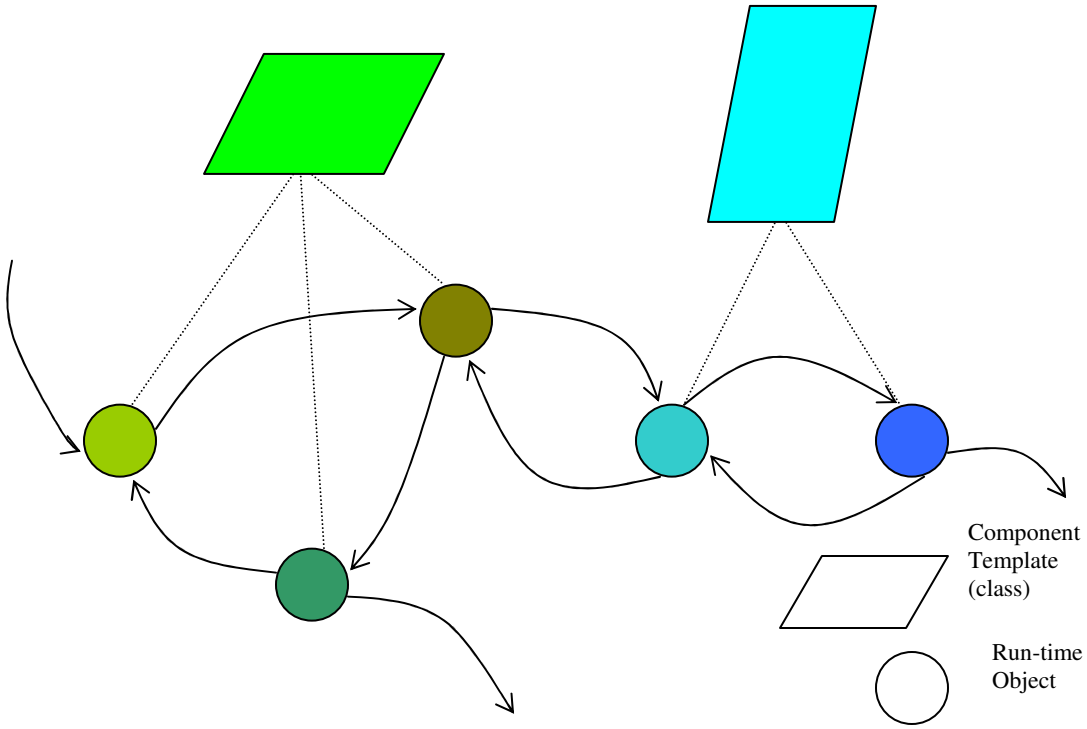

Fig. 2: The Static (Generic) and Dynamic (Specific) Layers of a Software System

Future errors or adapting the system in response to a modification in the environment.

Therefore, during the maintenance phase, software components may be accessed from, as well as new ones may be added to a reusable library of the concerned application domain. For instance, a change to adapt the software to a new environment may specialize already existing components, so that characteristics of the new environment are taken into consideration, hence expanding the spectrum of environments the reusable components are able to operate in. After changes are introduced to the system, an updated release of the software is generated. Maintenance of software system does not only allow the software to evolve but also the reusable library concerning the existing systems expands during the maintenance of a legacy system.

DISCUSSION

People hardly ever solve a new problem from scratch. Instead, they try to figure out similarities among a new application and previously known applications and their solutions, by making suitable assumptions from acquired experience, people attempt to solve the new problem. This process is referred to as *solving by analogy* and is considered to be a natural way by which people learn. The successful use of solving by analogy depends on recognizing similarities between problems and recalling solutions to analogous problems. Therefore, it can be assumed that the knowledge that software engineers have about a certain application domain increases the chance of reusing solutions from that domain. Nevertheless, most of the current software processes do not take this human characteristic into account.

When software engineers are developing software in an unfamiliar application domain they do not apply the same skills as when they are constructing software in a familiar domain. Indeed, there are differences between the ways to produce software, depending on whether or not they can use the knowledge obtained when they developed equivalent software in a well-known domain. Hence, the knowledge software engineers have about an application domain affects the way software development is carried out. Experts tend to rationalize in more abstract and high-level terms following a top-down manner. On the other hand, novices usually start working with low-level abstractions of the software and the development process is thus predominantly bottom-up. The top-down and bottom-up strategies have a significant effect on reusability because in a top-down style reusability is mainly accomplished in terms of generalization and specialization of abstractions, whereas in a bottom-up manner reusability is primarily achieved as aggregation of components.

A strictly top-down or bottom-up strategy to software production is not quite appropriate. The Y model preaches a top-down or bottom-up fashion for software creation, taking into consideration the knowledge that a software engineer has about the application domain. This knowledge naturally determines the prevailing strategy to software development. Thus, the predominant strategy is determined by the software engineer's knowledge about the application domain. In broad terms, it might be concluded that most things are often built top-down, except for the first time when knowledge is limited.

CONCLUSION

The graphical features of a CASE tool for object-oriented design have been developed following the Y model. The experience of using the Y model has firstly shown that it is very difficult to follow either a strict top-down or bottom-up approach and that it is often necessary to switch over between them. This implies that it is helpful to clarify high-level functionality for the software along with the identification of some low-level components and study their interrelationships. As a result, when developing large software, it is important to synthesize ideas from both top-down and bottom-up fashions.

A software system can be seen at two different layers of abstraction as shown in Fig. 2. There is an upper layer that shows the component templates and a lower layer that consists of run-time objects that depict the behavior of a particular software system. The idea of being able to classify parts of a software system as generic and hence potentially reusable, is a powerful feature and indeed for spending more time on the general aspects of the software that might be needed for specific application. The development of a component should therefore be with generality and reuse in mind placing perhaps less emphasis on satisfying the specific needs of an application that is being developed. In contrast, the specific parts of a design are those parts which turn a general set of components into a specific software system for a particular application.

The Y model supports "development with reuse" through component assembly, as well as "development for reuse" through component archiving. Initially, the software engineer identifies potentially reusable components from existing reusable libraries. The components are then selected, adapted and reused through composition, generalization and specialization mechanisms. At the end of software development, there may be many new reusable components that need to be verified, catalogued, classified and then stored into reusable libraries. This facility generally has a thesaurus

of synonyms to help understand the terminology used in the cataloguing scheme. In addition, a repository should address the problem of conceptual closeness to retrieve components that are similar to but not exactly the same as the desired one.

Finally, the Y model appears to cover the likely phases of large software development and enforces software reusability along its phases. Moreover, it takes into account previous knowledge that software engineers may have about the application domain, which has an influence on the prevailing approach (top-down or bottom-up) to be followed during the software development, therefore the Y model addresses the concerns of developing family of software systems, thus it has great applicability in component-based software development.

REFERENCES

1. Microsoft, 2004. COM+, http://www.microsoft.com/com/tech/complus.asp.
2. SUN, 2004. Enterprise JavaBeans, http://www.java.sun.com/products/ejb/index.html.
3. IBM, 2004. Component Broker, http://www.software.ibm.com/ad/cb.
4. Object Management Group, 2004. The Common Object Request Broker Architecture, http://www.omg.org.
5. Wallnau, K. C., S.A. Hissam and R.C. Seacord, 2002. Building Systems from Commercial Components. Addison-Wesley.
6. Clements, P. and L. Northrop, 2002. Software Product Lines. Addison-Wesley.
7. Royce, W.W., 1987. Managing the development of large software systems. Proceedings of 9th IEEE International Conference on Software Engineering, pp: 328-338.
8. Boehm, B.W., 1988. A spiral model of software development and enhancement. IEEE Computer, 21: 61-72.
9. Bent, K., 1999. Extreme Programming Explained. Addison-Wesley.
10. Cusumano, M.A. and R.W. Selby, 1997. How Microsoft builds software. Communications of the ACM, 40: 53-61.
11. Nuseibeh, B., 2001. Weaving together requirements and architectures. IEEE Computer, 34: 115-117.
12. Capretz, L.F. and M.A.M. Capretz, 1996. Object-Oriented Software: Design and Maintenance. World Scientific Press.