

Improving software development practices through components

Andrea Valerio, Guido Cardino, Vincenzo Di Leo
COCLEA, via Magazol 32 – 38068 Rovereto (TN) - Italy
Andrea.Valerio@coclea.it, Guido.Cardino@coclea.it, Vincenzo.Dileo@coclea.it

Abstract

Today software organizations face the challenge to develop quality products and services to respond to the rising customer requests and to sustain the growing of modern society. Software crisis, a concept coined many years ago but still actual today, synthesizes the chronic inadequacy of the software firm to satisfy customer needs. Software development is a very complex process that has not achieved the maturity level that the market requires; software engineering and specific technologies like object orientation and components promise to raise the process capability, reducing development time and costs and increasing software quality

This paper presents a case study concerning the improvement of the software development process of a small Italian firm through the adoption of a component-based approach. It describes the methodology adopted from a technical point of view and it details the impact that the improvement action had on the organization.

1. Introduction

This paper describes the introduction inside a small Italian software firm of a component-based development approach. This experience had the goal to improve the software development process of the firm in order to advance the technological basis and to achieve actual business benefits.

Our everyday life relies today on electronic devices and software applications are in most of the cases the core of these devices. Electronics and hardware industry has achieved the capability to deliver to the market new apparatus very sophisticated but at low costs; this is a major result of the standardization and engineering of the development process, in particular toward the definition of modular architecture and standard components. A computer is now the assembly of several cards and components of possible different producers over a general architecture (the main board).

The software industry has not yet achieved this maturity; the process is still rather creative and it relies on the skills and capabilities of the people involved in the process. Software engineering and related fields have not had the impact that software practitioners expected at the

beginning: the basic idea to transfer the main successful characteristics of the hardware production into the software process encountered several problems linked to the intrinsic characteristics of the software products.

Nevertheless, many of the technologies proposed demonstrated an effective potential to improve the software development process, laying concrete steps in bridging the gap between the firm's capability and the market requests.

In this context, component-based software engineering (CBSE) proposes to shift the attention of the development process toward the identification and re-use of components, promising in the medium term an increased product quality and decreased development time and costs.

But, CBSE requires an investment, not only of money, and it does not guarantee a short term return to the enterprise; this is a very critical point for small and medium enterprises that need clear benefits as a necessary condition to survive with the limited capitals they have.

This paper describes an experience conducted in a small software firm with the introduction of component-based development approach, emphasizing how this technology is integrated in the software development process. One of the peculiar aspects we focused is the trade-off decided among the initial requirements and the need to focus the action in order to reach the maximum benefits with the minimum effort and to maximize the integration with the existing context.

In the following chapter we describe the context of this case study; chapter 3 presents the component-based approach and its introduction in the development process. Chapter 4 discusses the impact that the introduction of a component-based development approach can have in a small software organization and gives some concluding considerations.

2. Background

COCLEA is a small Italian company offering software products, services and advice in the area of e-work, e-business and Internet exploitation. Object-orientation is conceived by the company as a leading approach to reach strategic business and technical results, such as better product quality, reduced time-to-market and costs. COCLEA decided two years ago to invest in the object-

oriented technology and to adopt it inside its software development environment. The personnel involved in the software development process had a background in object-orientation, in particular considering programming languages, such as C++, Java, Visual Basic and other ones.

The objective of the COPPER project is to experiment in COCLEA a software development approach based on component integration in specified product architecture. The technical objectives expected are: to better software application design, to improve modularity and understandability of applications, to increase component cohesion and decrease inter-component coupling, to shift the focus from objects and classes in the code to components and architectures in the analysis and design. For the business operation of the firm, it is expected that COPPER contribute to achieve a greater efficiency of the development process. Besides, it should help in reducing customer assistance and maintenance costs, leading to an increased quality of products and services perceived by customers. The COPPER take-up action will contribute to move toward the definition of a corporate software reuse strategy in the continuous improvement program of the firm.

The main lines underlining the COPPER project are:

- object orientation: the component-based development approach is considered a step forward from object-oriented development, but it relies on these core technologies already introduced in the organization;
- Uniform Modelling Language (UML) [1]: UML represent a stable language supporting the documentation of several development activities; it has achieved a broad consensus from the scientific community and it is an efficient tool for the documentation of object-oriented and component-based development;
- Integration with the existing development practices, reducing the impact of the new approach on the organization in order to better control it and estimate its side effect on the business operations of the firm;
- Motivation of the personnel and adequate training in order to prepare the people to work with the new methodology, strengthening the cooperation and creating the basis for the efficient team work required to transform components into a competitive advantage.

These premises led to the definition of a component-based approach that follows the indications given in the book 'Software Reuse' of Ivar Jacobson, Martin Griss and Patrik Jonsson [5]. In the next chapter we describe the integration of a component-based development into the software process.

3. Component-based software development

The component-based methodology employed in COPPER is based on an interactive software life cycle. It focuses on components and it identifies two main views: the production of components and the (re)use of components inside software applications.

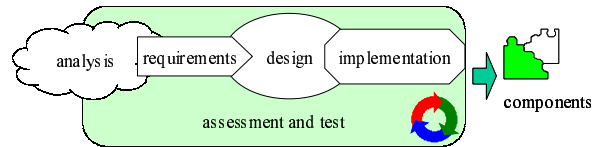


Figure 1. The high-level process model representing the production of components.

A component can be defined as a package of a homogeneous set of objects that collaborate to perform a feature or functionality and exposing a component interface that allows to integrate it in a system and make available to the external environment a set of services.

Producing a component is a difficult task because it requires analysing a functionality in a specific context deriving a set of requirements and then extracting the common aspects and the possible variants. It resembles a traditional software development process but it requires an horizontal analysis of the relationships and constraints that the component has (and can have in general) with the external environment. Very often this process requires to find an adequate trade-off between different factors in order to produce a component that is effectively reusable. This activity is contained in a more general process that is usually referred as domain analysis. The output is a component (model and implementation) that supports a set of variability options.

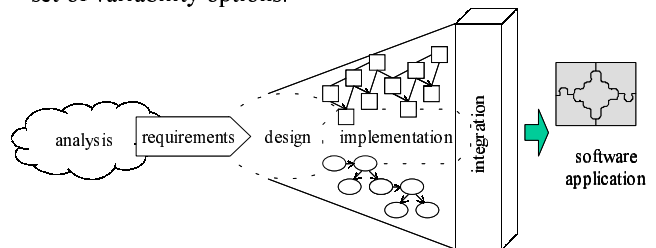


Figure 2. Variability and component variants.

The second view concerns the (re)use of components inside a software system. It focuses on the aspects that regards the integration of existing components inside a general model of a software application.

The initial analysis step focus on discovering customer needs and formalising them into a requirement document, adopting structured textual descriptions and use case models. The requirement document is the input to the design step. Usually, in traditional software development focused on a specific application for a customer, the

design aims to define an optimal solution to the initial problem. Using UML, several techniques can be used, depending on which are the most important aspect we deal with. Class and object diagrams focus on the static aspects of the system, while collaboration and sequence diagram focus on the dynamics and communication in the system. The design model is refined step by step until it is reached an adequate detail level and the model elements can be mapped into code.

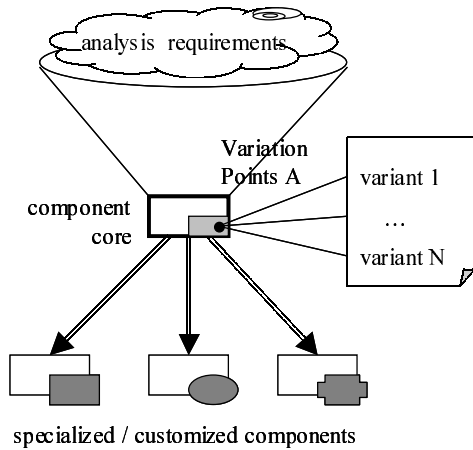


Figure 3. Traditional software development process.

Considering a component-based approach, the general process described has to be changed in order to explicitly work at the component level. Designing and developing with components requires a change in the way we move from the analysis to the definition of the design models. In a traditional product-centred process, the design is a direct consequence of the requirements; usually it does not focus on identifying components and sub-system; the architecture of the system is made up by collaborating classes and objects. Adopting a component-based approach and moving in the perspective of introducing (component) reuse practices, in the design phase we focus our attention on identifying components and their relations in the framework of a general architecture. This approach allows to re-use components previously developed (the more we reuse a component the higher is the benefit we obtain) and to integrate them into the application architecture.

The sections that follow will deepen the single aspects of the software process we depicted

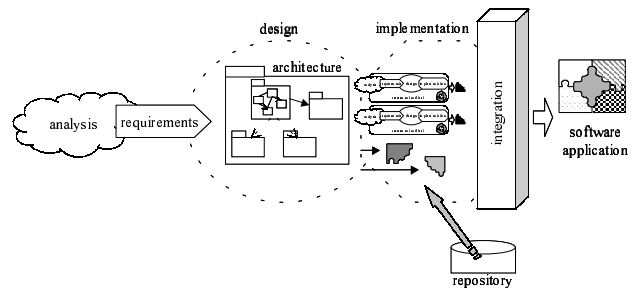


Figure 4. Component-based software development process.

3.1. Components analysis and requirements definition

Very often when we start a software project all the system requirements are not fully known. The reasons could be many: the customer does not express explicitly all what he/she wants or he/she gives only an incomplete description of the problem we have to cope with, the analysts are not able to understand and to identify all the requirements or they do not succeed in formalizing them at the first step, etc. Moreover, it is well known that reality is continually changing and the solution that the software gives must adhere to this reality; in other words it is in the nature of things that requirements change during the software life cycle.

Iterative development processes aim at coping with this uncertainty, reducing the distance between developers and customers: at each iteration the process focuses on a limited set of requirements that are stable and clear, building step by step the system, or it focuses on deepening the understanding (part of) the requirements to clarify them and to start their implementation.

There are two major categories of requirements we have to deal with when considering a component or a software system:

- requirements that specify the general nature of the component, i.e. the core functionalities that the component must perform independently from the context where it is introduced;
- requirements that characterise the component in a specific context or peculiar characteristics that differentiates or customise the components in respect to the external environment.

In this second case, we focus on specific aspects that correspond to variations of the component; these will be translated into variation points and the specific customisation will be variants of the component. Abstract components, i.e. components that include one or more variation points, are specialized by attaching one or several variants to its variation point(s).

The techniques used to capture requirements are mainly based on textual structure descriptions and Use Cases. Use Cases are defined by UML and they define

what the system should do through a graphical and intuitive notation (a main way to communicate and discuss with the customers).

In the requirement documentation the variation points and the variants are explicitly identified; each variation point and the possible variations it can assume are described and the possible constraints and context element linked to them are exposed. In the component design and implementation this information is crucial and it leads to specific mechanisms for variability management.

The output of the analysis and requirement definition is a document that exposes all the information just described.

3.2. Components description

A component is a homogeneous set of interacting objects that performs, usually, one specific functionality. Object-orientation teaches the principle of ‘design for change’: if we structure the code into modules we can minimize the impact of future changes, i.e. side effects on the structure as a whole. The implementation of this principle relies on information hiding and inheritance.

The same principles can be employed in component development, and they simplify the reuse of a component in a black-box way, in other words integrating it in the system without modifying its implementation. But it happens that we have to modify a component, adapting it or its characteristics to the specific system we are building. And to modify implies that we must understand – at least in part – the implementation of the component, a possibly very complex class hierarchy. This requires an in-depth knowledge of the component being modified and of its relationships with the external context, and if we have to re-build this knowledge, the potential benefits of reusing a components are lost.

In order to simplify the reuse of components and to document them in an appropriate way, we adopted a three layers model. This model allows to document a component at different level of detail: a first level, the context view, focuses on the description of the relationships of the component with the context, i.e. with the other components of the system or within the specific framework that operates it. A second level, the design view, focuses on the description of the component interface and presents the information required to reuse it in a black-box fashion or with a small amount of changes. The third level, the implementation view, gives the full detail of its implementation and of the class hierarchy it encapsulate. This inner view presents all the information needed to operate a white-box reuse of the component, in case of great changes required to its structure.

Similar approaches based on a layered description of components are proposed by Yacoub, Ammar and Mili [2][3] and by Kruchten [4]. In the first case, the authors propose to characterize a software component through

three categories very similar to the three levels we employ: informal descriptions, internals, and externals. Kruchten proposed a 4+1 view model of architecture, considering the logical view, the process view, the physical view, the development view plus the use of scenarios for illustrating architectural decisions with a few selected use cases.

Context view: it focuses on the description of the component in the context of the surrounding world, i.e. the systems and the components with which it interacts. The description is based on a set of initial diagrams representing the conceptual characteristics of the component and its boundaries in respect to the rest of the software system. Every architectural assumption done by the component itself on other systems or components should be made explicit, with the objective to avoid as much as possible the rise of architectural mismatches. Every fundamental part of the diagrams and every important decision taken in their definition should be traced back to the corresponding requirement by means of proper UML associations (e.g. «trace») or references.

The diagrams and notation employed for the context view describes the major structural aspect with Class diagrams and Implementation diagrams, but other UML diagrams can be used. Use Case diagrams can be used to show the relationships among actors (such as other components) and use cases within the component, i.e. functionalities that the component manifests to external interactors within the system; Behavioural diagrams can show important dynamic aspects. Specific UML stereotypes, like the “Nodes” of Deployment diagrams and the “Components” of the Component diagrams, can be especially useful in describing context constraints or components relationships.

The context view should clearly identify and describe variation points and variants and their relations with the overall context in which the component operates.

Design view: it focuses on the description of the programming interface of the component, usually defined as a set of collaborating classes with their methods. Considering the heterogeneous sets of software assets developed in a firm, ranging, as an example, from executables and libraries written in C/C++ to PHP scripts and class libraries coded in Java, the design view is not limited to Class diagrams, but it can adopt Deployment, Implementation diagrams or dynamic models like Collaboration diagrams.

The design view goes a step further in the description of variation points and variants. In particular, components that include variation points can be customised, i.e. specialised by substituting in the variation point a specific variation, using a wide range of technique or variability mechanisms. These techniques depend from several factors, including the language employed, the complexity

of the variations, the possible variants, etc. Among the different techniques we can list the following:

- inheritance (generalization and specialization), both at the level of Use Cases or design and implementation;
- extensions and extension points;
- parameterization;
- configuration (also through configuration languages);
- generation (from languages or templates).

The design view has to detail the variability management, describing and documenting the specific technology employed in each case.

In general, complex components should be accompanied by a collection of usage rules describing how best to reuse the component.

Following the discipline used in moving from the requirements to the context view, also in the step from the context view to the design view the traceability must be maintained. The result is a trace mapping from (group of) elements in the context models to (group of) elements in the design models, describing how the general idea in the context diagrams are mapped into programming and variation interfaces in the component structure.

Implementation view: it shows the internal structure of the component. Ideally, the implementation view should not be strictly necessary in building new systems by component (black-box) integration. A formal definition of the component internal structure is motivated by:

- it is necessary to store the corporate knowledge regarding component implementation, so that to make possible the execution of maintenance activities (e.g. bug fixing, small changes) in a later stage (possibly involving different employees than the original developers);
- components are a great and important investment, therefore any relevant and useful information about them is a richness for the company;
- the internal structure of a component and its traceability to design, context and requirements can be used in training new employees to the new component-based approach;
- components are designed and coded to last in time, therefore they probably will evolve if the technology, market or customers' needs change; in principle, they could be the subject of great modifications involving their implementation (if this change will be accepted as economically affordable).

The implementation view is made up of a set of UML diagrams. Since the focus is on the internal organisation of collaborating software modules, it is expected a wide use of Class and Interaction Diagrams, together with behavioural diagrams (e.g. State Charts or State Transition

Models), whereas Use Cases will probably have a limited application here.

The implementation view corresponds to the typical activity that software developers carry out in traditional projects, but it is focused on components (the implementation of functionality blocks with specific interfaces) and in particular on variability management.

3.3. The architectural model

With software architecture we define a structure or structures of the system, which consists of software components, the externally visible properties of those components, and the relationships among them (Len Bass, 1998). Large systems are characterised by a large number of objects with complex collaboration patterns. Architecture allows to effectively cope with these systems through abstractions aimed at simplifying the model using higher level components that interact. Simply speaking, we move from the class and object level to the subsystem and component level.

The architectural model moves from the analysis of the system to the design of a solution. It is in charge of defining the high-level design elements, their interactions and the structure of the system. It describes the main components participating to the system and the relationships among them; it presents the distribution constraints and the software organization.

We base our approach on the proposal given in [5]: starting from the requirement document we have to identify the main entities that participate to the system. In the abstraction activity we can draw one or more sketch diagrams (we can employ static structure diagrams or dynamic representations). It is important that we do not lose the links between these diagrams and the requirements in order to maintain a strict connection with the documentation. The resulting model represents an initial abstraction of the main elements participating in the system. In the traditional project oriented process we proceed by successive refinement in order to design the single objects in the system and their exact collaboration patterns and then we move to the coding phase. Considering a component-based approach, we refine the initial description of the system into a higher level architectures: we group objects that are strictly connected (high cohesion elements) into components and subsystems and we highlight the interactions among them. Package and subsystem UML elements with the standard «import» and «access» stereotypes are used to describe the high level organization of the system. Abstraction dependencies can be useful to document the architecture and its origin. For example, an analysis-level object might be split into several design-level objects linked through abstraction dependencies. The UML standard stereotyped classes of abstraction are derivation, realization,

refinement, and trace (corresponding to the stereotypes «derive», «realize», «refine» and «trace», respectively).

The architecture focuses on structuring the collaboration among the components in order to give the capability that the customer expects from the system. The components that participate to the architecture can be reused if we have already developed a component that matches the profile requested or we can start a component development project to build them.

Variation points and variability is managed through the identification of the variation points in the architecture (encapsulating them in the components when feasible) and then mapping them in the detailed design of the elements they are referred to. Each variation points must be traced to the related documentation in order to be able during the detailed design and implementation phase to correctly embed it.

Architecture documentation is crucial: recording and explaining both the decision taken and the information related to the architecture is essential to move to the next steps of the process and for maintaining and modifying the system during its life cycle. The documentation produced with the architectural model usually includes the following information:

- a software architecture document presenting the high level structure of the system, the organization and the relationships among the components;
- the design decisions (reasons for the decision, constraints and alternatives, too) that have been taken and the rationale that is behind the architecture document;
- the possible variation points and the variants that have been considered.

Textual information are recorded in a document accompanying the architectural diagrams; textual comments are attached to the diagram(s) in the form of annotations.

UML proposes some useful stereotypes:

- «requirement» comment: specifies a desired feature, property, or behaviour of an element as part of a system;
- «responsibility» comment: specifies a contract or an obligation of an element in its relationship to other elements.

As a general rule, it is possible to link a document to architectural elements. UML provides the concept of comment: a comment is an annotation attached to a model element or a set of model elements. It has no semantic force but it contains information useful to the modeller. Scenarios (or collaboration and sequence diagrams at the component level) can be also used to document design decisions or to clarify behavioural and synchronization aspects.

4. The impact of CBSE

A formal final validation of the new approach is scheduled after the summer of 2001, when quantitative data will also be available. However, we can draw some considerations regarding the introduction of the component-based approach in the corporate development practices.

First of all, even if a set of UML supporting applications appears to be available in the market, it is really difficult to find a suitable tool for a small-sized enterprise. In most cases, prices and hardware requirements for workstations are very high. Moreover, several UML tools lack the flexibility we need for the definition of diagrams, and the stress towards formal correctness of UML diagrams found in many tools reveals often in a drawback. The important point to consider here is that we want mainly use these tools for documentation (targeted to human beings) and not for automatic analysis (performed by machines on a formally correct UML diagram).

The change from object orientation to component-based engineering seems to be a less radical shift than from functional to object-oriented programming. Programmers and designers have only to 'update' their perspective, applying their expertise to a new level of complexity (e.g. encapsulation is applied to a component - a set of classes - instead of to a single class).

Many difficulties arise at the organisational level, in particular regarding the new organization of role and activities and the management of the documentation produced. One critical point, for example, is the classification and storing of reusable components for successive reuse.

The massive introduction of UML, in particular as a major communication language, poses requirements to the skills and professional preparation of the employees; this confirm the conviction that a key factor in the software process is the people, and strength the need for continuous training and education initiatives.

5. Acknowledgments

The COPPER project is co-financed by the European Commission under the IST program (reference: IST-1999-20797).

6. References

- [1] OMG, *Unified Modeling Language Specification*, Object Management Group, Version 1.3, June 1999, <http://www.omg.org>.
- [2] Sherif Yacoub, Hany Ammar, and Ali Mili, *Characterizing a Software Component*, ICSE99 - International Workshop on Component-Based Software Engineering, May 1999

[3] Sherif Yacoub, Hany Ammar, and Ali Mili, *A Model for Classifying Component Interfaces*, ICSE99 - International Workshop on Component-Based Software Engineering, May 1999

[5] Philippe Kruchten, *The 4+1 View Model of Architecture*, IEEE Software, November 1995, 12 (6), pp.42-50

[5] Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success*, ACM Press, Addison Wesley, New York, 1997