From Objects to Components: a Quantitative Experiment

Miguel Afonso Goulão and Fernando Brito e Abreu

Information Systems Group (INESC-ID)
Departamento de Informática (FCT/UNL)
2825-114 Monte da Caparica, Portugal
+351 212948300 (ext. 10731)
{miguel.goulao, fba}@di.fct.unl.pt
http://ctp.di.fct.unl.pt/~mgoul

Abstract. Component based software development (CBD) is increasingly becoming a de facto approach to software development. Most software professionals were originally trained to build software using another paradigm, such as the object orientation (OO) paradigm, or the structured programming one. To face the trend to CBD, software professionals are required to make a paradigm shift. Such a shift incurs in considerable costs. This paper describes an experiment where part of a legacy software application built with the OO paradigm was transformed into a software component, using two different technologies (Object Pascal and C++). In this experiment, we were concerned not only with the qualitative aspects of the problems dealt with by a software professional in this transformation, but mostly with some quantitative ones. In particular, we compared the effort required to make such transformations with each of the technologies. The subject performing the experiment was at ease with all the involved OO languages, but not with the component models supported by the used platforms.

Keywords

Migration from OOP to CBD, Paradigm Shift, Component Based Software Engineering

1 Motivation

Component based software development (CBD) is becoming increasingly important for the software industry. Its' market has been growing steadily (Bass, 2000; Williams, 2000) and this is perceived as a business opportunity to several organizations.

Components provide a convenient way for shipping functionalities in a black-box to software developers.

Throughout software development history, one of the goals of software developers has been the production of extensible systems. Object-oriented development represented an important step towards extensibility. The combination of fundamental OO mechanisms such as polymorphism, late binding, information hiding and inheritance provides a good foundation technology for software extensibility, but is not enough. Higher levels of encapsulation and safety are required (Szyperski, 1995).

Full use of inheritance breaks encapsulation (Synder, 1986), to a certain extent. A disadvantage of class inheritance as a form of reuse is that the subclass becomes dependent on the parent class implementation. The combination of insufficient documentation and availability of the libraries' source code often makes the clients use that source code as the actual library documentation. The goal of decoupling between clients and providers is somewhat missed, as the interaction between clients software and the libraries should not depend on the libraries' implementation details. This may also have an effect on market. Several potential vendors are not willing to provide their source code.

According to Szyperski (Szyperski, 1995), the support provided by many OO languages both to late binding and to information hiding is not enough. In a component oriented system, the code required to handle a particular component may or may not be loaded at the time the component is referenced. The inclusion of late linking is required so that code can be integrated by the client on demand. This includes locating, loading and linking the code, as it happens with dynamic link libraries.

Many of the object-oriented languages support information hiding at the class level, but not beyond that, as one would have with a module. Safety is a key issue. Components need to be validated independently, as it is not feasible to rely on integration testing. After all, components are developed by independent organizations without prior knowledge of the applications they will be used in. Safety support properties such as garbage collection and static type checking wherever possible, complemented by dynamic type checking need to be established. Szypersky claims that the establishment and maintenance of properties that hold on entire components regardless of what other interacting components are doing requires a stronger level of encapsulation, than the one provided by individual classes.

The introduction of a new technology or paradigm in the software process may bring many benefits, but it also has its costs. If we go back to the paradigm shift from structured programming to the object-oriented one, we have to recognize that the introduction of new mechanisms such as inheritance or polymorphism brought not only their expected benefits, but also some new challenges, such as:

- the need to learn how to use and take advantage of these mechanisms;
- the requirement for developing different life cycles, with the corresponding need to train professionals to work under these new circumstances (Sellers, 1993);
- new development errors that were not present in structured languages, requiring the creation of different testing techniques to cope with them (Binder, 1995);

- an evolution on empirical software engineering practices: we need new criteria and methodologies both for quality evaluation and cost estimation.

A similar situation occurs with the paradigm shift from Object Oriented Programming (OOP) to Component Oriented Programming (COP). Practitioners need to adjust their skills in order to properly take advantage of COP. This requires some learning effort that should be accounted for.

Although COP is an active research area, not much attention has been devoted to the quantitative evaluation of the costs and benefits involved in the adoption of COP. There are few research contributions in the realm of empirical software engineering devoted to CBD in general (Heineman, 2001), and particularly to COP. Among other aspects, components' quality and complexity evaluation is an open topic for research. From a producer's point of view, the capacity to evaluate the quality of a component and its complexity may be determinant factors for forecasting human resources distribution and, therefore, for establishing a pricing strategy. From a consumer's point of view, the ability to make an informed buy is limited if no measure of the product's quality and complexity is available.

Many other questions are still to be answered. What sorts of learning difficulties are typical from the development with components? What gains of productivity are achieved through the usage of component-based development, when compared to its object-oriented counterpart? Which are the benefits in terms of quality in the final systems due to this paradigm shift?

In this paper, we are concerned with the learning difficulties faced by software developers who are experienced with OOP, but are taking their first steps with COP. We have devised an explorative experiment where the same developer converts a functional subset of an application developed with OOP to a software component. This experiment is part of an ongoing research effort aiming at developing a CBD quality model. Before pursuing more ambitious experiments, we wanted to get a grasp of the earlier mentioned difficulties, so that appropriate research hypotheses can be formulated and then tested.

This paper is organised as follows: we will describe the methodological approach followed in this experiment in section 2. The collected data will be presented and analysed in section 3. Section 4 will present a brief overview on related work. Conclusions and further work will be outlined in section 5.

2 Methodological Approach

2.1 Establishing goals

The purpose of the experiment was to evaluate the difficulties faced by an experienced OO programmer without experience with COP to transform a part of an OO application into a software component.

To achieve this goal, the following questions were asked:

- How much effort does it take to browse through the online help provided by the adopted development environments before the practitioner feels comfortable enough to create the component based on the existing application?
- How much effort does it take to make the transformation for the first time?
- Which are the tasks involved in that transformation?

2.2 Experiment setup

The first task was to choose the platforms on which this experiment would be set. The following criteria were established for the selection of the programming platforms:

- Subject's previous experience with the programming languages. It was important to guarantee, as much as possible, that the OO programmer had a similar level of experience with all the programming languages, to avoid biasing the experiment with a different level of expertise in each language.
- Availability of platforms on the experiment site.
- Language support for OOP as well as COP, at least to some extent.

From the plethora of available technologies two were chosen: Object Pascal and C++. Both technologies fulfilled the requirements, as they are two of the OO languages taught by the subject to his students in different undergraduate courses (thus fulfilling the first two requirements) and can be used as implementation languages in COP as well.

The legacy application had been developed in Object Pascal a few years earlier, with an old version of Borland Delphi. It used a Graphics Package (Conner, 1995) that hides many details on the usage of Delphi's graphical components. The application was updated to directly handle Delphi's visual components. It was then migrated to the Microsoft Visual C++ .Net platform, so that we would have a similar starting point for each of the technologies.

From then on, the time required by all the tasks involved in the development of the components was recorded, for the purpose of this evaluation.

2.3 The OO Legacy Application

The OO legacy application chosen for the purpose of this experiment was a 3D function viewer, originally developed with Borland Delphi as an exercise for a programming course. The application allows the customization of 3D functions and then draws them on a window. The drawing style (wireframe or filled), its perspective (oblique, isometric or dimetric) and colors can be chosen by the user. The application

interface consists of a dialog box where the user can configure the function and its graphical presentation and then launch a viewer. The viewer is a second window where the function is shown. For the sake of equity, this application was then migrated to the other base technologies used in this experiment keeping the same fundamental interface and architectural characteristics. Figure 1 presents a partial view of the application's class diagram.

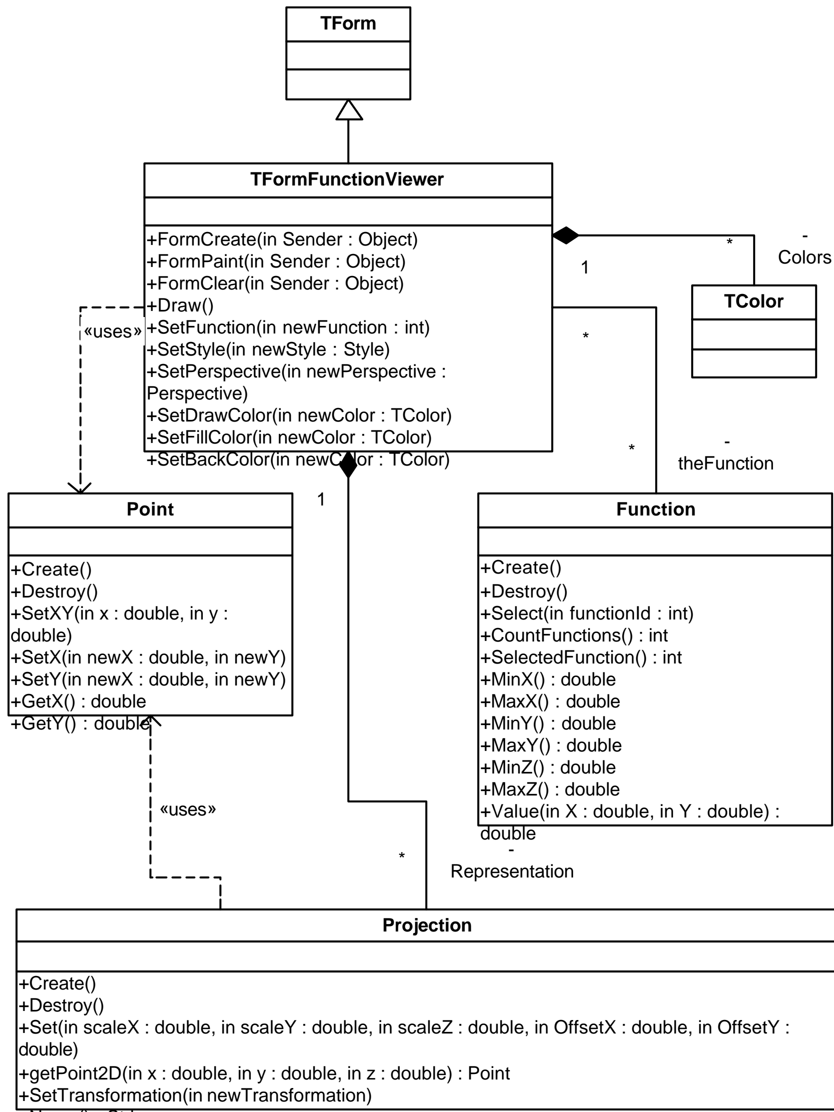

Fig. 1. Partial view of the application class diagram.

The legacy system clearly separates classes that deal with the drawing of the functions and related computations and those that are responsible for the GUI (Graphical User Interface) configuration of the drawn function.

2.4 Transformation Process

The simulation of the learning effort was performed with the following platform sequence: Object Pascal (Borland Delphi) and C++ (Microsoft Visual Studio .Net). Furthermore, it was assumed that this learning effort would be carried out in an experimental fashion, to simulate the situation of a software professional who has to quickly learn a new technology by himself, using online manuals and the tools themselves as learning materials. The following sections outline the main transformations required by each of the simulations.

2. 4. 1 Transformations with Delphi

With Borland Delphi, the creation of a component is achieved through the creation of a unit, followed by the derivation of a component class, its registration, compilation and installation on the component palette, making it available for integration with other applications. Delphi's component creation wizard guides the software professional through project configuration.

The adaptations from the original system were relatively simple. In Object Pascal, a component's interface is defined through a unit's interface. This is a common feature, whether we are following the OOP paradigm or the COP one. For the purpose of our component, we were interested in encapsulating the 3D function computation and drawing classes in the `Function3D` component, rather than in the ones related to the GUI. Although this component is implemented with a combination of several classes, its interface can be adapted from the `TFormFunctionViewer` class. The following changes have been made:

- There are a few adaptations concerning the location of type definitions and so on, so that all the necessary information for the component's interface can be made available in the component class file. The enumerated data types that were used in the component's interface had to be moved to the unit where the component's class is identified, so that they were available on the component's interface.
- Instead of inheriting from `TForm` (a class that represents a form in Object Pascal), we now had to use a visual component (`TFigure`) as a base component class. The base component allows the inclusion of images in forms. The derived one adds to it the ability to make the 3d function plots.
- Some methods were renamed, to better reflect their effect in the component. For instance, it no longer made sense to call `FormCreate` to the constructor of the component, so this was changed to `Create`.
- A Register procedure was added, to register the component in the component's palette (this task was automated by the IDE).

For illustration purposes, we present an excerpt of the components' interface, as specified with Object Pascal. This code is quite similar with the one we would expect to find in an ordinary class.

```
interface
{other types here…}

TFunction3D = class(TImage)
    { other declarations … }

  published
    { Published declarations }

    procedure Create(Sender: TObject);
    procedure Paint(Sender: TObject);
    procedure Clear;
    procedure Draw;
    procedure SetFunction (newFunction: integer);
    procedure SetStyle (newStyle: TStyle);
    procedure SetPerspective (newTransformation: TPerspective);
    procedure SetDrawColor(newColor: TColor);
    procedure SetBackColor(newColor: TColor);
    procedure SetFillColor(newColor: TColor);
end;
{…}
procedure Register;
implementation
procedure Register
begin
  RegisterComponents('Samples', [TFunction3D]);
end;
{…}
```

2.4.2 Transformations with C++

Microsoft Visual Studio .Net offers 15 different wizards for C++ projects creation, each of them with a wide range of options. This variety is, at first, a problem for the unexperienced component designer, as the adequate choice is far from obvious. In our experiment, we used the *ATL (Active Template Library) project wizard*, which allows the creation of a DLL (Dynamic Link Library) where the COM (Component Object Model) object can be deployed. The project's creation results in the generation of several source files that will be used as a starting point for the creation of our DLL. This is somewhat similar to what happens with the creation of visual applications with this programming environment. The next problem, then, is to find out where to insert our code in the solution skeleton provided by the IDE (Integrated Development Environment).

The component is created through another wizard, as an *ATL simple object*, which generates some more source files with source code skeletons for the programmer to fill later. Until this point, no source code has been directly edited, although there are

already about 15 source files involved in the project. Among them, we find IDL type definitions of the type library, component interfaces, component implementation and so on.

Adding methods and properties to the component is done through wizards, to make sure all the involved source files are coherently changed when we add some new feature to the component. As usual with this IDE, the right spot for introducing our own code in the source code skeleton is marked with appropriate comments and accessible through browsing facilities.

The following example illustrates a small portion of one of those files. In this case, we chose part of the IDL interface for Function3d.

```
interface ITFunction3d : IUnknown{
   [helpstring("method Create")] HRESULT Create([in] TObject
Sender);
   [helpstring("method Paint")] HRESULT Paint([in] TObject
Sender);
   [helpstring("method Clear")] HRESULT Clear(void);
   [helpstring("method Draw")] HRESULT Draw(void);
   [helpstring("method SetFunction")] HRESULT SetFunction([in] int
newFunction);
   [helpstring("method SetStyle")] HRESULT SetStyle([in] T_Style
newStyle);
   [helpstring("method SetPerspective")] HRESULT
SetPerspective([in] T_Perspective new_Transformation);
   [helpstring("method SetDrawColor")] HRESULT SetDrawColor([in]
TColor newColor);
   [helpstring("method SetFillColor")] HRESULT SetFillColor([in]
TColor newColor);
   [helpstring("method SetBackColor")] HRESULT SetBackColor([in]
TColor newColor);
};
```

Building the project automatically creates the DLL and registers the new component, making it available for other applications.

The component can be tested in applications developed in C++ or other language (for instance, Visual Basic). With C++, this involves using functions such as `CoInitialize()`, `CoCreateInstance()` and `CoUninitialize()`, to use the component in an application. Programmers without COP experience are not likely to be familiar with many of the constructs (functions, macros and so on) required for using the components in the application code.

2.5 Component Specification

The Function3DViewer component partially implements the functionality described in the previous subsection. The component's interface allows for the specification of the function, perspective, drawing style and colors that are to be used in the graphical representation, along with a reference to the viewer where the graphic is to be drawn, if

necessary (this varies with the chosen platform). Figure 2 shows a component diagram representing the Figure3D component, as defined with Borland Delphi.

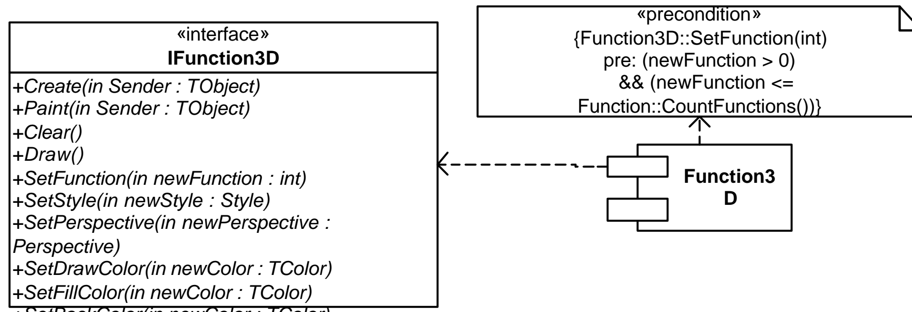

Fig. 2. Function3DViewer component diagram.

3 Data Analisys

In order to get a grasp on the amount of effort required by the transformation of our application into components using the two different technologies, we recorded 3 different tasks. These tasks were performed concurrently, as development was often made in paralell with documentation review:

- Documentation – here we consider the time spent searching and reading online documentation either provided by the tools themselves or found on the internet.
- Transformation process – this measures the time spent in the actual development of the component. Most of the source code editing was performed with simple cut & paste from the legacy applications.
- Creation of a testing application – time spent on creating a testing application that uses the component. The testing application was deliberately simple, just to test whether or not the component was available (i.e., suitably registered) and working.

Table 1 summarises some quantitative aspects regarding this experiment.

	Object Pascal	C++
Documentation	45	300
Transformation process	20	120
Creation of a testing application	5	30
Total	70	450

Table 1 – Effort spent with each task (in minutes)

For the sake of discussion, it should be pointed out that these values are only indications of a tendency and should be regarded as such. Although, as much as possible, the experiment has been conducted with minimal interruptions, the subject was not on a completely isolated environment that would allow an even more accurate effort time recording. Occasional interruptions occurred, but their effect has been taken into account in these time measurements.

As one may conclude from the transformation descriptions in section 2, the transition turned out to be much simpler with Object Pascal. A relatively quick browse through Delphi's documentations shows that the amount of new concepts that the practitioner has to master before he can try to create a new component is relatively small. The component specification is almost identical to a normal class specification. For a programmer used to create form-based interfaces with Delphi, the new component is just another graphical component, so including it on the form offers no new challenge.

The development of a component with Visual C++ involves several "new" concepts for the practitioner. He has to build interfaces, use IDL, learn how to access components through their unique identification code, learn how to initialize a new component and how to use it. Although the IDE offers a valuable help in filtering this extra complexity, it does not hide it completely. All these new challenges require time, both dedicated to reading documentation and to experimenting with the new concepts. The simple existence of several sophisticated new tools and language constructs to master presents an extra difficulty.

In summary, we believe that Object Pascal (which is generally regarded as a simpler language than C++) offers a less abrupt paradigm shift from OOP to COP than Visual C++. However, it should be noted that the tested C++ environment is part of a family of development environments that share a common basis. The investment in understanding the usage of components in one of those environments is likely to pay for itself if after learning how to develop components with C++ we try to do the same for C#, for instance.

4 Related Work

Taylor defined paradigm as "an acquired way of thinking about something that shapes thought and action in ways that are both conscious and unconscious. Paradigms are essential (…), but they can present major obstacles to adopting newer, better approaches". He then defined paradigm shift as "a transition from one paradigm to another. Paradigm shifts typically meet with considerable resistance followed by gradual acceptance as the superiority of the new paradigm becomes apparent. Object-oriented technology is regarded by many of its advocates as a paradigm shift in software development" (Talylor, 1992). (Lindsey, 1997) supported this and argued that a real and nontrivial paradigm shift occurred in many organizations when moving from the procedural world to the object world. The lack of skills with object orientation was viewed as one of the major inhibitors for its adoption.

The paradigm shift related difficulties (from procedural to OOP) are also recognized by educators. The pedagogical patterns project is a good example on how educators themselves try to make a paradigm shift in their approaches, namely when teaching OOP to students used to structured programming (Eckstein, 1997).

On a different context (the adoption of Agile Modeling), Ambler points at misinformation and the fear of the unknown as factors that are responsible for skepticism in the adoption of new approaches to software development (Ambler, 2001). A cultural shock is always something to expect when the introduction of changes can be viewed as a threat to the *status quo*. Although there are success stories, Ambler expects it will take a few years before significant studies on agile modeling are performed. This is somewhat similar to what we have observed with component based software engineering so far: there is a lack of empirical studies showing the benefits of CBD adoption (Goulão, 2002).

5 Conclusions and Future Work

With the growing importance of COP, there is a need for empirical studies on it that can help us to better understand its impact on the software development process and on software quality. This is an open topic for research.

In this paper, we described an experiment that is part of a research effort aimed at developing a software component quality model. The experiment was concerned with the learning difficulties presented by the adoption of COP. It should be emphasized that the nature of this experiment does not allow us to draw definitive conclusions on the learning difficulties of the analyzed paradigm shift. This experiment had an explorative nature. One of our goals was to establish the grounds for more ambitious experiments rather than making generalizations from the results in the experiment.

We expect to perform controlled experiments with homogeneous groups of students, developing the same program with a detailed specification as a basis. Half of the teams will develop the software using CBD, while the other half will use the OO paradigm. In this controlled experiment, we will use the pedagogical pattern PRCM (Peer Review and Corrective Maintenance) (Abreu, 1996) to limit the effects derived from the personal variability of the subjects used in the experiment.

The costs for shifting from an old paradigm to a new one need to be tracked and rolled into cost estimates. The effect of shifting from the procedural programming paradigm to the object oriented one was observed in a software's physicist user community, that, among other tasks, is responsible for the development of software subsystems within a large international project. A large stretch-out in time when migrating subsystems originally developed with Fortran to C++ has been observed and explained by the difficulties resulting from the lack of experience with OO development. A similar phenomena is expected to happen while shifting to the CBD paradigm.

References

(Abreu, 1996) Fernando Brito e Abreu: Peer Review and Corrective Maintenance (PRCM) and Preparation, Industrial Presentation and Roundtable (PIPR) Patterns. Workshop on Pedagogical Patterns (OOPSLA'96). San José, California, USA, 1996.

(Ambler, 2001) Scott W. Ambler, The Threat of the New, Software Development Magazine, December, 2001. Available on the web at http://www.sdmagazine.com/

(Atlas, 2001) Comments to Nov 2000 DOE/NSF Review, 2001. Available on the web at http://www.usatlas.bnl.gov/computing/0105Reviews/agencyComments.txt

(Bass, 2000) L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord and K. Wallnau: Volume I: Market Assessment of Component-Based Software Engineering, Software Engineering Institute, Technical Note CMU/SEI-2001-TN-007, May 2000.

(Binder, 1995) Robert V. Binder: Object-Oriented Testing: Myth and Reality. Object, May 1995. Available on the web at http://www.rbsc.com/pages/myths.html

(Conner, 1995) B. Conner, D. Niguidula and A. van Dam: Object-Oriented Programming in Pascal - a Graphical Approach. Addison-Wesley, 1995.

(Eckstein, 1997) Jutta Eckstein: A Paradigm Shift in Teaching OOT. Proceedings of the Educators' Symposium at OOPSLA '97. Atlanta, Georgia. 1997.

(Goulão, 2002) Miguel Goulão and Fernando Brito e Abreu: The Quest for Software Components Quality. To be presented at COMPSAC'2002. Oxford, UK, August, 2002.

(Heineman, 2001) G. T. Heineman and W. T. Councill, Component-Based Software Engineering - Putting the Pieces Together. Boston, MA. Addison-Wesley, 2001.

(Lindsey, 1997) A. H. Lindsey and P. R. Hoffman: Bridging Traditional and Object Technologies: Creating Transitional Applications. Application Development, Vol. 36, No. 1, 1997. Available on the web at http://www.research.ibm.com/journal/sj/361/lindsey.html

(Sellers, 1993) Brian Henderson-Sellers and J. M. Edwards: The Fountain Model for Object-Oriented Systems Development. Object Magazine, Vol 3, Issue 2, pages 71-79, 1993.

(Synder, 1986) Alan Synder: Encapsulation and Inheritance in Object-Oriented Languages. Proceedings of Object-Oriented Programming Systems, Languages, and Applications, pages 38-45, 1986.

(Szyperski, 1995) Clemens Szyperski: Component-Oriented Programming: A Refined Variation on Object-Oriented Programming, The Oberon Tribune, Vol 1, No 2, December 1995.

(Taylor, 1992) D. A. Taylor: Object-Oriented Technology: A Manager's Guide, Addison-Wesley Publishing Company, Reading, MA, 1992.

(Williams2000) J. D. Williams: Raising Components, Application Development Trends, vol. 7, pp.27-32, 2000.