Applying CBSE theory on corporate resources

Niklas Eriksson
ISBIT AB
Kopparbergsv. 14
722 13 Västerås Sweden
+46 (0)21 171920
niklas.eriksson@isbit.com

Ted Vaalundi
ISBIT AB
Kopparbergsv. 14
722 13 Västerås Sweden
+46 (0)21 171920
ted.vaalundi@isbit.com

ABSTRACT
Currently, there are a lot of articles written on the theoretical side of CBSE, but these theories in general did not reach the state of practice. There are still too many problems to which this paper presents a practical case where CBSE technologies and methods are applied to different assets in a corporation. We discuss how we, as a small consultant company, have chosen to deal with some of the CBSE problems. We feel that the term software-component must be expanded so that we can use it to encapsulate those assets, not just binary software. We will also introduce a “component environment”, a prototype for handling different types of components

Keywords
CBSE, Component, CM, Software Reuse

1 INTRODUCTION

Component Based Software Engineering (CBSE) or Component Based Development (CBD) is thought to be the future of software development. At least if we should judge by all articles that are published by the software development community. We constantly learn about different component technologies, component concepts and component models. We read about how different components can be glued together into different types of applications. This sounds easy, but still we do not find too many implementations and support where these concepts can be used in everyday development.

We have been working with the COM[1] technology for about four years in different software projects. We have also been involved in a lot of Configuration Management (CM) issues around this topic. Our experience of CBSE is that the concept is very immature regarding CM- and Quality Assurance (QA) -issues. Software companies must understand what CBSE is and how to take advantage of it. Several companies that we have worked for do not want to use any component technology simply because they do not understand CBSE and the consequences of how CBSE will affect the development cycle of a project. Many companies are still implementing software as monolithic applications and thereby neglecting the power of CBSE.

ISBIT AB is a small software company that mainly works with software consulting. Our area of interest is modern software engineering focused on industrial applications. To strengthen our competitiveness we, as software consultants, need an effective way to reuse knowledge and distribute information in our organization. However being a consultant company we are in a different situation compared to ordinary software companies. Working with different technologies and different concepts, reusing knowledge in a practical way is in fact quite difficult. Is there a way to solve this?

This article will discuss and summarize some of the problems raised during our work with different CBSE technologies. We will also discuss the work of a CBSE related project called ISBIT Component Environment (ICE). Can corporate assets be modeled as components and why is there a difference between developing and using components? As a proposed solution, we describe the requirements behind ICE and the underlying architecture.

1.1 ICE – ISBIT Component Environment

In order to explain our point of view and some of the assumptions mentioned in this article we intend to use the ICE-project as a practical example of how CBSE can be used in a powerful, yet corporate-friendly way.

The ICE-project started as an idea about software reuse in the end of 1998 at ISBIT AB, Sweden. The goal with ICE was at first to gather, maintain and develop a C++ class library that would contain various functionality needed by a software developer developing COM-components. During our work with implementing COM-components we often found ourselves reinventing different functionality over and over again. Reuse of binary components was not enough. We needed a way of reuse corporate specific source code with functionality that was not part of none of the libraries

STL, ATL or MFC[1]. We needed a corporate framework for implementing components.

As the work with ICE grew, we soon realized that we needed to extend ICE to incorporate other programming languages than just C++. We also needed to incorporate binaries and even knowledge. Along with the source code and binaries, we wanted to add test-programs and examples implemented in different programming languages. But how could this be achieved since ICE until now had been considered as a framework for C++ programmers only? The component concept was soon identified as a solution. But could we consider a C++ class as a component? Or event more abstract, could we consider knowledge as a component? The entire idea with ICE was about to change. Today ICE contains both source code and binaries such as:

- C++ classes
- Visual Basic modules
- Delphi VCL[2] components
- ActiveX controls
- HTML/XML-support such as behaviors, cascading style-sheets and etc.
- ICE CM-tools such as B.A.D (Build And Deployment), Documentation tools, Scripts etc.

2 COMPONENT GRANULARITY

Consider the following statement for one moment:

"A component is an asset that can be reused"

By defining a component like this we automatically neglect the fact that one might use components to achieve scalability and modularization instead of reuse. The primary goal of implementing a component may not be the reuse in itself. The choice of implementing an application based on components may very well be influenced by scalability and modularization requirements.

We think that software developers must consider components differently regarding the level of abstraction that they are working with. Take the hardware industry as an example. In the hardware industry components such as semiconductors, capacitors, inductors and resistors are used at the lowest level of abstraction. From these basic components integrated circuits are built which must be considered as the next level of abstraction. Further more advanced circuits are built from the integrated circuits as complexity grows for each level of abstraction.

Considering the software industry, we can apply the levels of abstraction here too. For example: Compare the work of a software developer that implements device drivers with a software developer that implements user interfaces. The software developer that implements device drivers are probably programming in ANSI-C with its standard library along with some kind of Software Development Kit (SDK) for device driver development. Can we consider the ANSI-C standard library as a component? Can we consider the device driver SDK as a component? Can we consider the result (the device driver itself) as a component? The software developer that implements user interfaces works on a different level of abstraction. For this kind of work Visual Basic and different kinds of ActiveX controls can be used to implement the user interfaces wanted. Can we consider a Visual Basic module as a component? Can we consider different ActiveX controls as components?

From our point of view the definition of a component is different depending on the level of abstraction. We call this *component granularity* (compare this to Szyperski [7]). If you are working with C++, a class may be considered as a component. If you are working with Visual Basic an ActiveX control may be considered as a component. But what about interfaces and contracts in our discussion? Is this not an important part of the component concept? Of course it is. We think that interfaces and contracts are merely a secondary aspect on witch rules and constraints for updating and changing interfaces must be applied in the CM-process for CBSE. For example: during an iterative development cycle, changes of requirements often result in changed interfaces. Doing this must be allowed as long as the component is not released as a product. Once the component has been released as a product the author(s) of the component has no control over the usage of the component. This means that bugs and errors may be corrected as long as the interfaces aren't changed. If an interface needs to be changed or updated, the work of this must be carried out so it complies with the rules of the component model used.

3 TWO "SIDES" OF A COMPONENT

From a corporate point of view it is often not enough to gather components in a repository and then use them. Especially since a company usually develops and maintains components of its own. There are many issues to consider whether you are using a component or if you are implementing one.

In reality several components may be incorporated in the same package in order to simplify the deployment of components. If we consider the Windows operating system we soon realize that we are using different parts and components of the operating system depending on the

[1] STL, ATL and MFC are different C++ class libraries. STL is short for Standard Template Library, ATL is short for Active Template Library and MFC is short for Microsoft Foundation Classes.

[2] VCL is short for Visual Component Library and is a class library used in Borland Delphi and Borland C++ Builder. VCL is implemented in Object Pascal.

application that we are developing. The operating system is a package that contains a magnitude of different components, some of which has dependencies to each other.

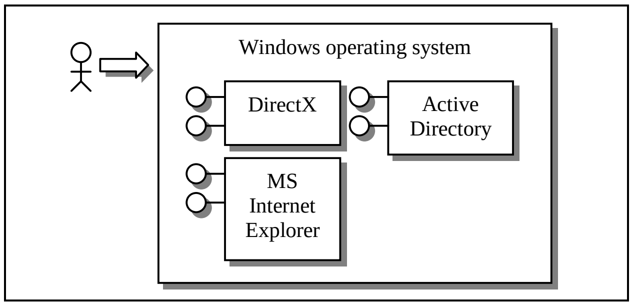


Figure 1. The Windows operating system contains several components yet in one package. A user only uses the components that he/she needs.

If we look at the COM-technology a COM-server[3] can contain several COM-components. This scenario is illustrated in Figure 2 where a component named *AxTypeLib* is used. The *AxTypeLib* component has dependencies to three other components: *AxInterface*, *AxDispInterface* and *AxCoClass*. These three components may be accessed directly, or through the *AxTypeLib* component. For easy deployment and distribution of these particular components all the components are packed in one module (in this case a DLL[4]) called *AxDolphin.DLL*.

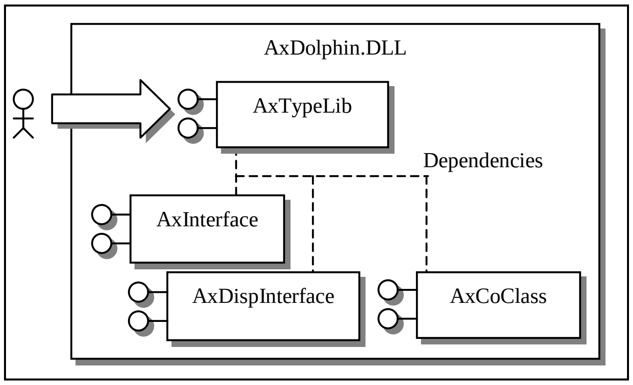


Figure 2. From a deployment point of view it may be practical to deploy components that have tight dependencies in one package. In this case a DLL hosts four COM-components.

[3] The COM specification specifies three different kinds of servers: In-process servers, local servers and remote servers. A server is a module that hosts components. In the Windows operating system in-process servers are dynamic link libraries (DLL) while local- and remote servers are executable files (EXE).

[4] DLL – Dynamic Link Library.

Can we apply the above component-concept when using C++? Consider the illustration in 0. Here we have an example of several C++ classes that are hosted in a class library. Some of the C++ classes have dependencies to each other, and that makes them suitable to distribute together.

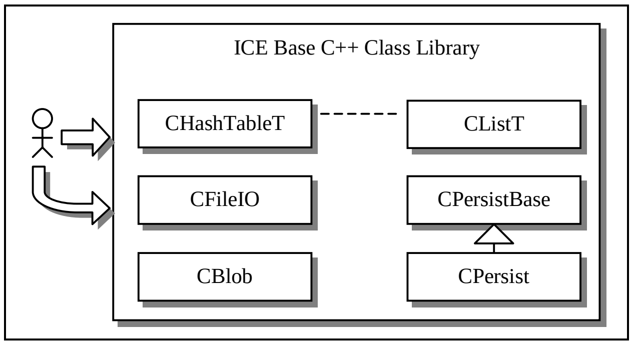


Figure 3. A C++ class library with dependencies and inheritance relations. In this case it is practical to deploy and distribute the entire class library instead of separate C++ classes.

3.1 Consumer/Producer side of component reuse

What we wanted to illustrate in Figure 1, Figure 2 and 0 is that we can apply our discussion of the granularity, and the two sides of a software-development, on components. We also wanted to illustrate that components are handled differently regarding usage and deployment. If we consider components in this way all components has two "sides", one regarding usage and one regarding storage and deployment as shown in Figure 4. We like to refer to the left side as the *consumer side* and the right side as the *producer side*. The consumer side illustrates the use and reuse of components that resides in a repository. The producer side illustrates the version handling system where the source code of each component resides. The B.A.D builds and deploys each component and installs the component in the repository. In the rest of this article we will focus our discussions around the producer side, which also is what ICE is all about.

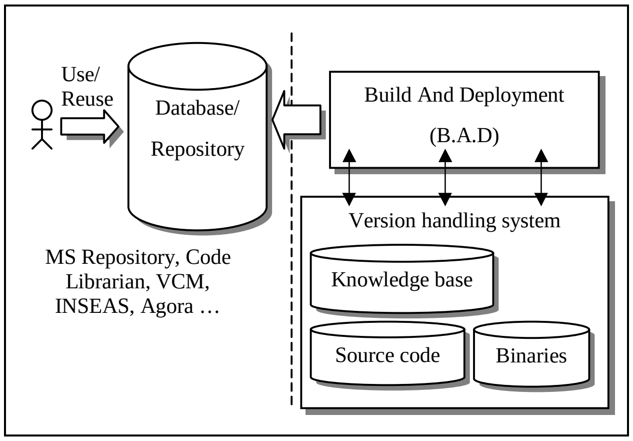


Figure 4. The consumer/producer side of component reuse as depicted in ICE.

As far as the consumer side in Figure 4 is concerned we have looked at some existing databases/repositories for components such as Microsoft Repository, Code Librarian in Microsoft Office, Microsoft Visual Component Manager[1]. INSEAS[3] and Agora[4]. We have tested the Microsoft Visual Component Manager in close relationship with ICE and are quite satisfied with its functionality.

By dividing the handling of components like this we separate the usage of components from the development, maintenance and B.A.D processes. This provides us with the choice of using different applications, utilities and tools when using components as *consumers*. A very important goal with B.A.D is to use feedback from the consumer side. This might for example be bug-reports and registered usage of components. We will go into more detail about this in 4.2.2.

4 THE ICE-PROJECT OBJECTIVE

The functionality of a component should reflect the requirements of it. This is quite obvious since this should be the case of all software that is developed for commercial purposes. In our previous discussion about component granularity we haven't, until now, said anything about requirements although it seems like a very important issue here. Like our perceptions about component granularity we also think that requirements are different regarding the level of abstraction. We call this *requirement granularity*.

We think that it is just as important to trace requirements in component-based software as it is in traditional software. This may in fact be considered as an important requirement on software components and software applications generally. The reason why we mention this in this article is that the requirements play a very important role in ICE regarding the architecture (see chapter 5).

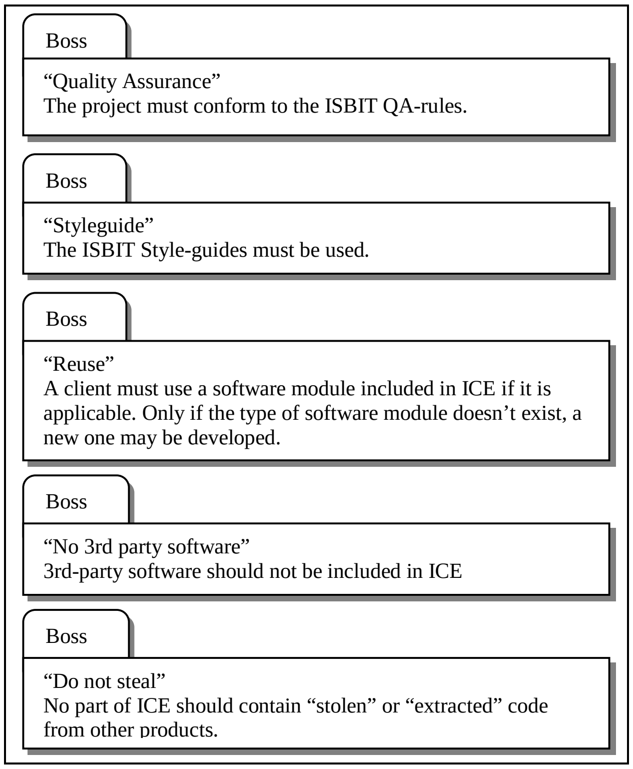
4.1 Requirement model used in ICE

In order to get a comprehensive set of requirements, we invented three different user-roles: *boss*, *guru* and *client*. These user-roles define the top requirements that are brought upon ICE from different viewpoints. The goal with this approach was to produce a set of general requirements. Each general requirement was then broken down into several concrete requirements. If the requirements still were too abstract to reflect some given functionality, the requirements could be further refined. This finally produced a hierarchy of requirements for the different levels of abstraction. The approach gave us an easy way to ensure that each function realize its requirements since we were able to trace the function all the way through the requirement- hierarchy.

One thing we learned during the definition and breakdown of the requirements was that a requirement that seemed obvious perhaps was not. The thing we discovered was that many of the requirements that we first defined, was in fact consequences of the top-requirements. So, one future research issue here may be to look at how requirements are defined.

4.1.1 Boss

The *boss* user-role impersonates the requirements that a company or organization has on ICE.



Boss

"Quality Assurance"
The project must conform to the ISBIT QA-rules.

Boss

"Styleguide"
The ISBIT Style-guides must be used.

Boss

"Reuse"
A client must use a software module included in ICE if it is applicable. Only if the type of software module doesn't exist, a new one may be developed.

Boss

"No 3rd party software"
3rd-party software should not be included in ICE

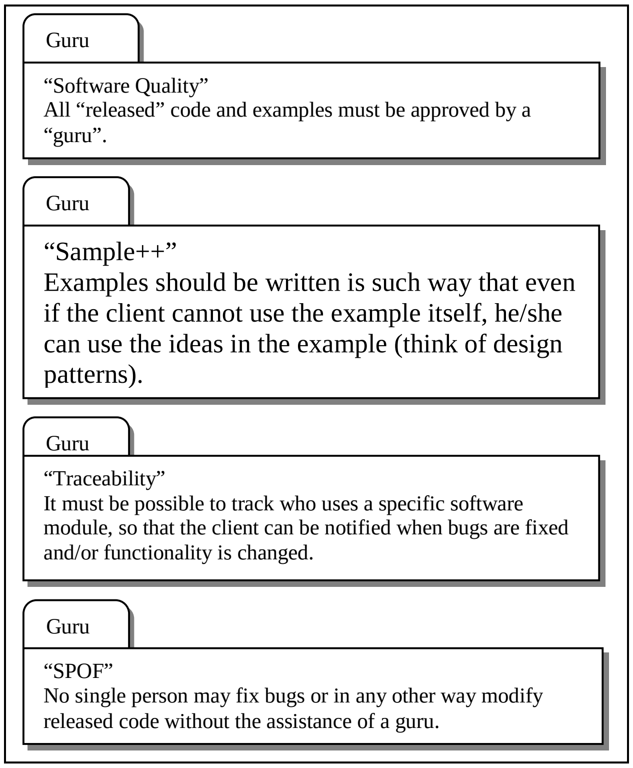
Boss

"Do not steal"
No part of ICE should contain "stolen" or "extracted" code from other products.

Figure 5. Requirements of the "boss" user-role.

4.1.2 Guru

The *guru* user-role impersonates the requirements that a software developer implementing components has on ICE. If we refer to our previous discussion in chapter 3 about the two sides of a component, these requirements mostly reflects the requirements of the producer side.


Guru

"Software Quality"
All "released" code and examples must be approved by a "guru".

Guru

"Sample++"
Examples should be written is such way that even if the client cannot use the example itself, he/she can use the ideas in the example (think of design patterns).

Guru

"Traceability"
It must be possible to track who uses a specific software module, so that the client can be notified when bugs are fixed and/or functionality is changed.

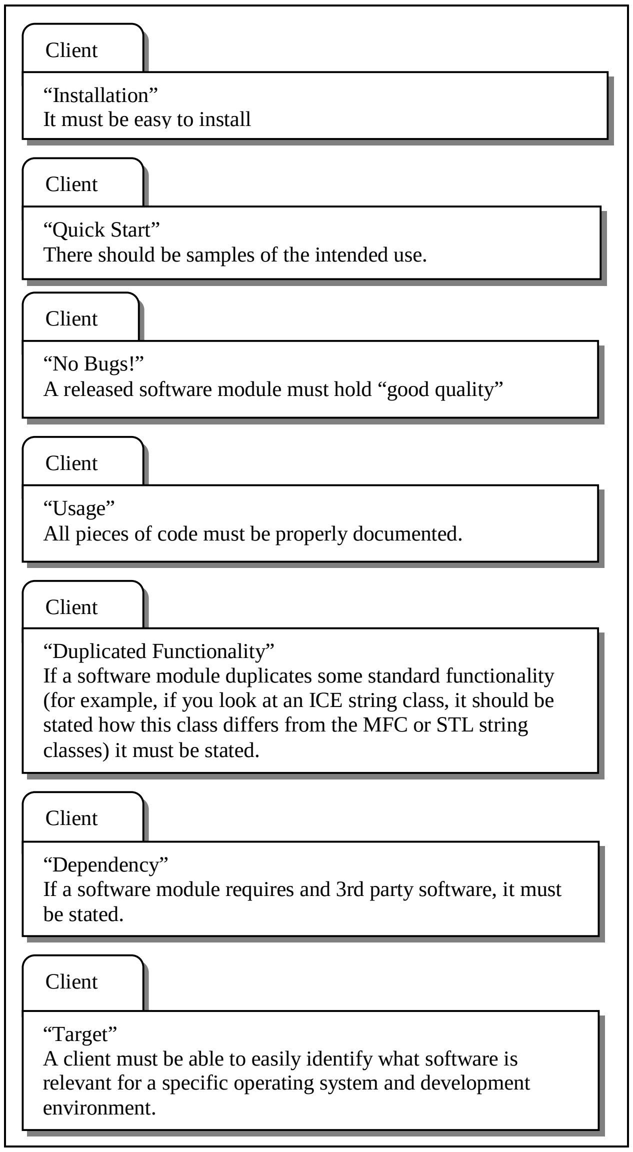
Guru

"SPOF"
No single person may fix bugs or in any other way modify released code without the assistance of a guru.

Figure 6. Requirements of the "guru" user-role.

4.1.3 Client

The *client* user-role impersonates the requirements that a software developer using components has on ICE. Referring to our previous discussion in chapter 3 about the two sides of a component, these requirements mostly reflects the requirements of the consumer side.


Client

"Installation"
It must be easy to install

Client

"Quick Start"
There should be samples of the intended use.

Client

"No Bugs!"
A released software module must hold "good quality"

Client

"Usage"
All pieces of code must be properly documented.

Client

"Duplicated Functionality"
If a software module duplicates some standard functionality (for example, if you look at an ICE string class, it should be stated how this class differs from the MFC or STL string classes) it must be stated.

Client

"Dependency"
If a software module requires and 3rd party software, it must be stated.

Client

"Target"
A client must be able to easily identify what software is relevant for a specific operating system and development environment.

Figure 7. Requirements of the "client" user-role.

4.2 Organizational problems related to CBSE

One thing that we have learned working with CBSE is that it is very important that all people within a corporation using CBSE, understands the consequences of it. We have already mentioned this in the introduction of this article. The introduction of CBSE in a company will affect the entire organization [5]. However it seems very difficult to motivate changes to an already well-trimmed organization simply because of programmers want to reuse components.

4.2.1 ICE project group

In order to administrate the ICE project, the project itself needed an organization. Several of the requirements in ICE simply could not be implemented with some fancy tool. Therefor the ICE project group was formed to handle several CM related issues such as:

- Version handling
- Change request handling
- Build and deployment
- Verify requirements against functionality

In a large organization you might want to consider dividing this group into several groups/boards (compare this with SCM and SCCB in CMM [6]).

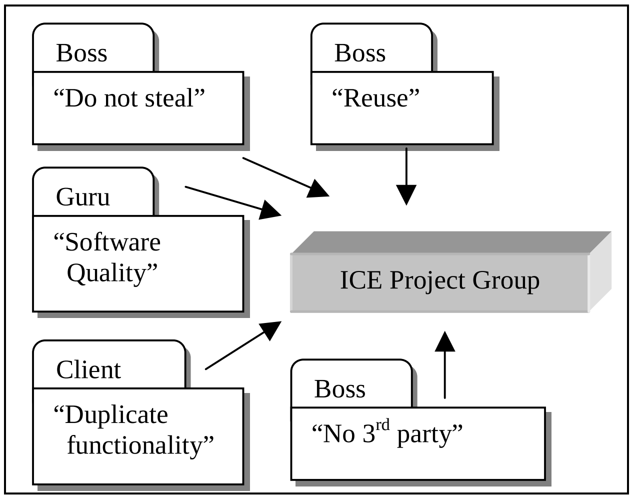


Figure 8. ICE project group requirements.

4.2.2 B.A.D

The build and deployment function is a vital part of ICE. Consider Figure 8 below and you will see the requirements for the build and deployment function. Both *Independency* and *Plug-in Architecture* are derived requirements not fully explained in this article. However here is a short explanation:

- Independency. Since we are working with several different platforms and systems we must be able to change the plumbing.
- Plug-in Architecture. Since the information associated with components may differ between abstraction levels and/or component models we must be able to extend B.A.D. For example in the future someone may want to associate a ”ToDo-List” with some components.

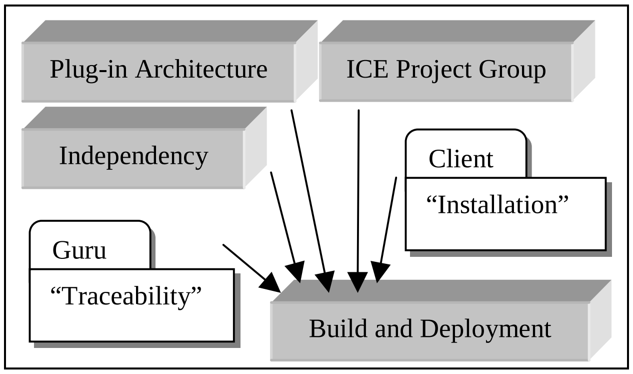


Figure 9. Build and deployment requirements.

As we discussed in 3.1 we need a way to interchange information between the two sides. Because both sides can consist of several different tools and information systems we can use the plug-in functionality in B.A.D to achieve this. To exemplify this, there are many products on the market that handle a specific CBSE issue. *Rational ClearCase* is an excellent version-handling system. *Visual Intercept* is a great system for handling change-requests. These kinds of products are a necessity to any type of modern software development, component-based or not.

5 B.A.D ARCHITECTURE

In this chapter, we will try to describe the overall architecture of the "build and deployment"-part (B.A.D for short) of ICE. The goal of the architecture is to make the components easily maintainable while allowing maximum flexibility when defining the components and packages (see chapter 3).

5.1 A Small Example

To better understand the problems that the architecture is dealing with, we start off with a small example. First, take a look at the description of the component and package hierarchy (to your right). As you know (see chapter 3), a package can consist of several components. To make things easier to handle, we probably need to group the packages in some way, which introduces the *subsystem*. A subsystem is simply a slot where you, as CM responsible, place packages that are related. This makes it easier to get an overview of your components (especially in large systems). The Capability Maturity Model [6] defines Configuration item/Configuration component/unit as the corresponding structure.

```
System ->
        Subsystem

Subsystem ->
        Subsystem* Package*

Package ->
        Component*

(Component ->
        Item*)
```

Below you see snapshot of a small part of ICE. This does look a lot like an ordinary directory structure, with one directory representing each subsystem/package/component.

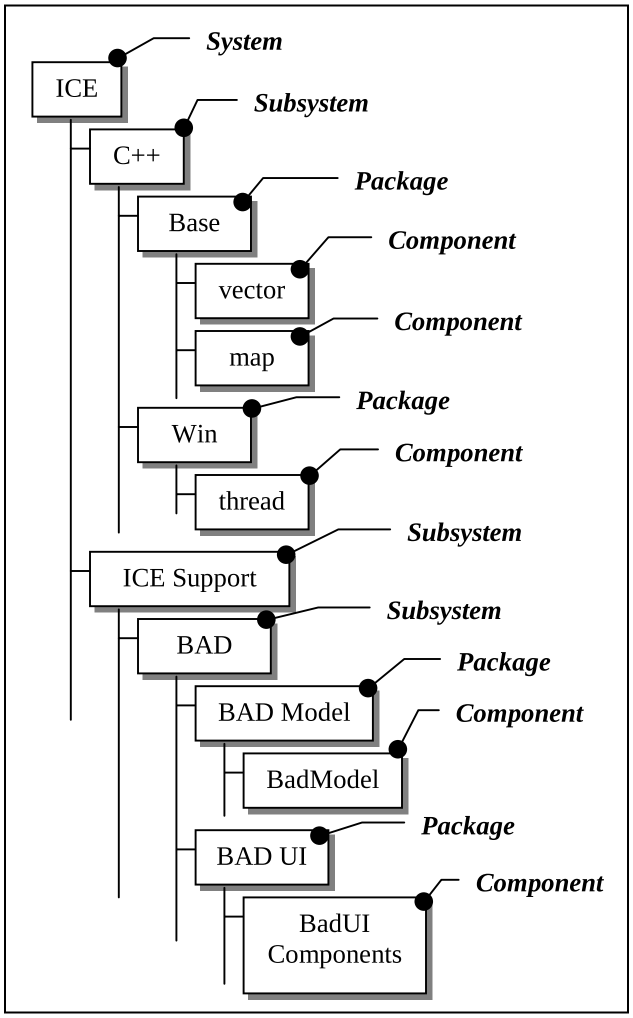


Figure 10. Snapshot of the ICE structure

5.2 The Model

Now that you know how the component hierarchy is used, we can go on with the architectural discussion. Think about the hierarchy of the packages and components again. When dealing with software, a component would normally consist of a number of files. One obvious solution would be to let each component, package and subsystem be represented by a directory on the file system. You could then put a small file in each directory stating which type that particular directory represents. The drawback with that simple solution is that some of the information might not be consisting of files. For example, you might want to use your expensive change-request system to keep track of changes to make for each component.

The first step from the file system would be to create an automation model[5] of the component structure. A client would then access the components via the model instead of looking at the file system. An UML-type description of the model can be seen in (Figure 11).

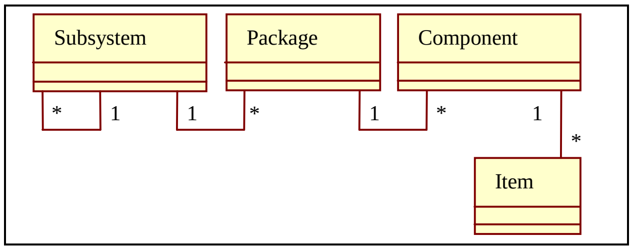


Figure 11. UML-model of the component hierarchy

The standard approach is then to create an application that the user interacts with. The application would interact with the model and the model could use files or whatever other source of information necessary to perform its task. You might recognize this as the standard three-tier solution (see Figure 12). Note that this opens for web-deployment of the user-interface and all the other advantages that you would normally expect from this kind of design.

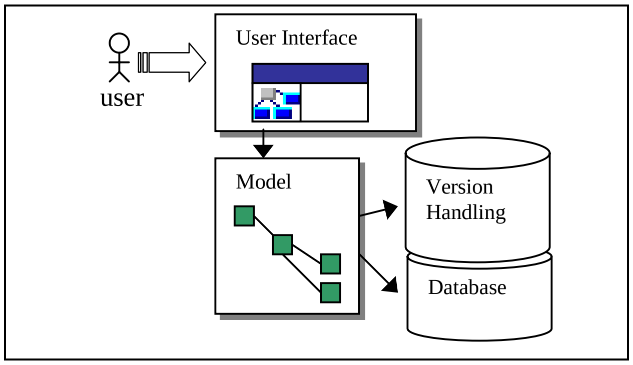


Figure 12. The three tiers of B.A.D

5.3 Inside the model

The real challenge, as you might have expected, is inside the model. We want the automation-model to do all the work for the client, and that is probably quite much. The components will usually be version-handled and consist of information from different sources. To make things even worse the components might be version-handled in different version systems. If we want the automation-model to be flexible and possible to reuse in several projects, we need a more general approach[6]. To make this solution more concrete we propose a framework.

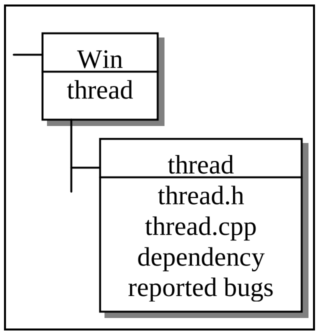
Again, we set the stage for a small example. This time we

---

[5] The term "automation model" refers to a set of COM objects that support OLE Automation (or just Automation).

[6] A smart person once said that you could generalize any problem by introducing one extra step of indirection. Some skeptic then mentioned that you could optimize the program by removing that level of indirection.

put focus on a part of one of the packages in ICE, namely the "win-package". For simplicity, win consists of only one component called "thread", and thread consists of four items. You find a birds-eye view of that scenario to your right. This all looks simple, but there are more to this that meets the eye. The first two items in the thread-component; "thread.h" and "thread.cpp", are both files in a version handling system. The dependency item in this case, can be figured out by analyzing the source-files, and the reported bugs are situated in a separate database. This gives us a bit more complicated picture, 0.



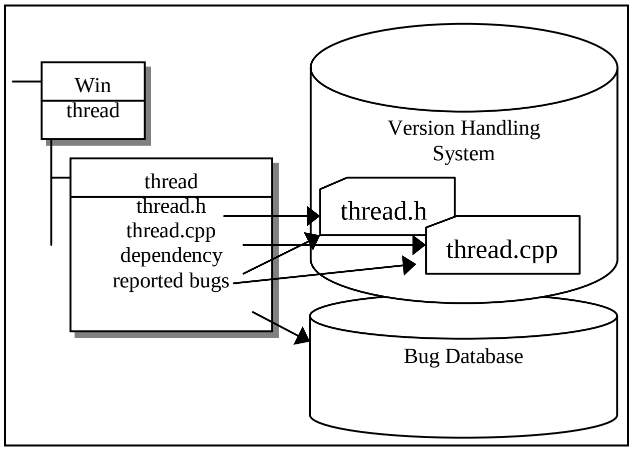


Figure 13. Thread component example – behind the scenes

To make it possible for each item of a component to be fetched from different sources, we need a uniform way to do that. Introducing the *supplier*. A supplier is a COM-object that fetches information from a source that is only known to the supplier it self, and returns it to the caller. In the example, we would have three different suppliers:

- File-version supplier. This supplier would know how to communicate with the version-handling system, and get the proper version of the thread-files.
- Dependency supplier. This supplier would scan the thread-files for dependencies (being C++ files, the dependency-supplier would be C++ specific).
- Bug supplier. A supplier that can fetch the bug reports for a component from the bug-database.

So the component object in the automation-model would use the different suppliers to know what items belongs to the component and use them to handle the users requests. But one question remains; how does the component know which suppliers to use? The beginner's answer to the question might be "from the configuration properties associated with the component object". But since we are searching for a neat design the answer is "behold the meta-supplier". Associated with each object in the model, there is a meta-supplier. The only job of the meta-supplier is to keep a list of the other suppliers associated with the specific object (see Figure 14).

5.4 Supplier types

There are two basic types of suppliers; meta-suppliers and suppliers. As you might have noticed in the example above (especially 0) the non-meta-suppliers are denoted "InfoSupplier". There are currently three non-meta-

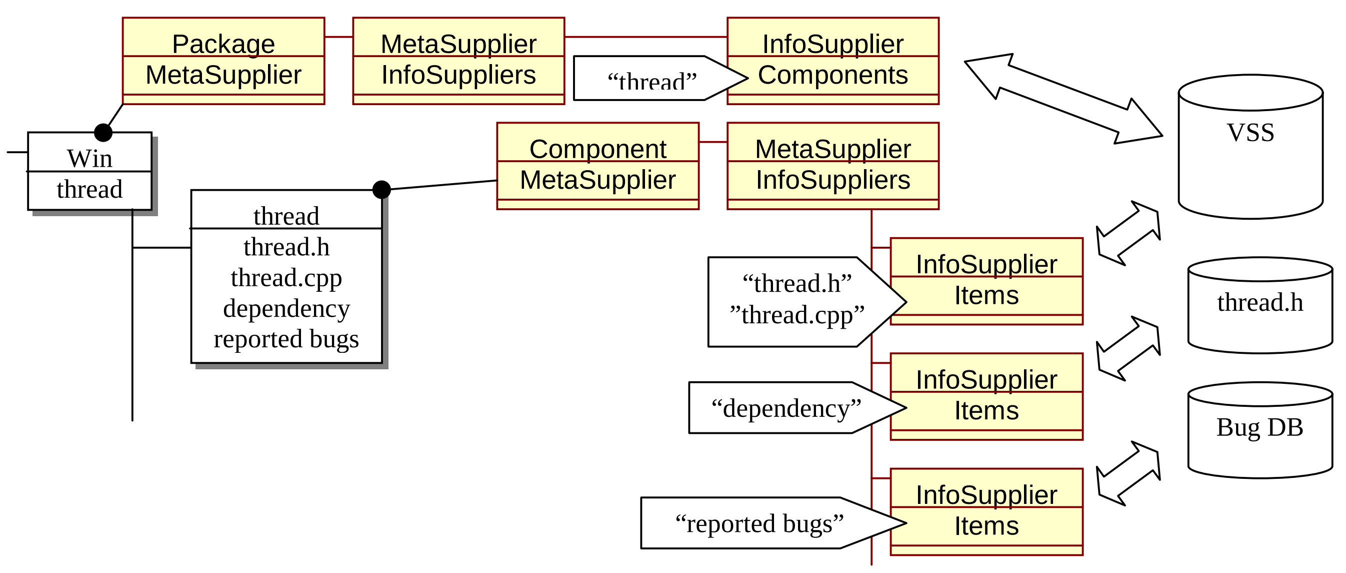


Figure 14. Thread component example – detailed view

suppliers:

1. Info-Supplier. This type of supplier provides items to the other suppliers and the user. On a subsystem, the info-supplier would state the packages belonging to the subsystem and so on.

2. Build-Supplier. The build-supplier provides information and functionality usable when building the components and packages. For example, a C++ component might have to be built using a specific compiler.

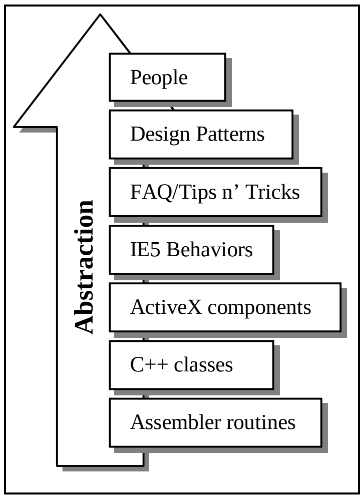
3. Deploy-Supplier. This supplier knows which files and items that are important when deploying the components and packages. For example, a VB project representing an ActiveX control contains a lot of files interesting when building the ActiveX, but the DLL created is the only file needed when deployed (and that one was not even there before we performed the build).

6 CORPORATE ISSUES ON CBSE

In a software corporation, the most obvious area of reuse is the software development itself. Treating software as components should increase reusability and hopefully boost productivity. Developing general components or full-fledged functionality is however very expensive [7], and common practice tells us that one individual project cannot afford to do this. Therefore we have found that it is important to "announce" intended functionality as soon as possible, so that several projects can work together and thereby dividing costs, making them affordable. To make this effective, it requires support from the entire organization.

1.1 Resources are components

Our intention is to treat all kinds of resources as components. For example there are several resources in a corporation that could benefit from reuse. Some of these resources can be directly associated with the software components themselves, tips and tricks and so on. Other might not be associated with individual components such as design-patterns.



We can also see that some resources can be modeled as components although not from the reuse point of view. One example of this kind of resource can be the people working in the company. As illustrated in the figure we can see that even if the level of abstraction increases, we can still apply the concept of components. In order to make things a little more concrete, we take a look at the coworker modeled as a component.

6.1 The two-faced consultant

Something very common in consultant companies are "consultant profiles". This is a document much like your normal "CV"; a summary of things you have done and things you are good at. This document is handed out to the potential customer as marketing material for the individual consultant. A big problem here is that there is a lot of information that the company would like to keep track of, but the customer should not know about. For example what a consultant thinks a different customers.

If you take a step back and think of a person as a component, you might see that the problems when choosing a component for your project is quite similar to selecting a consultant for your project. You need a description of its abilities, cost, timeframe and so on. If people were modeled as components, then we could use the same techniques on them as on software components.

There are several aspects of human resources that can be compared to software components, however some can't. Furthermore there are restrictions on the way we can use the human components. For example: there is only one physical instance of a person. There can also be different problems regarding human components. For example: It can be very difficult to grade the level of knowledge if we try to model abilities (C++ programming, UML modeling… etc) as interfaces.

7 CONCLUSION

We all praise the myth of reuse, and in particular software reuse. A software component is a piece of reusable software, but our experience tells us that the reuse-issue is much bigger then that. Regardless of whether we are developing an application that uses components, or the component itself, we reuse non-component software. There are even tools that are intended help with this particular problem (e.g. Code Librarian, shipped with Microsoft Office 2000). Our experience is that it is easier to reuse small components than large. So it would be a mistake to base the reuse on components only, when there is so much more information that could be shared. Of course, creating a "tips and tricks" database probably won't make a good marketing thing (you write a book instead and get filthy rich, but that is a different story). However a smaller company could benefit from reusing information on all kinds of abstraction levels.

As we have already stated, when striving for maximum productivity, we often want to reuse both small code snippets as well as larger solutions. To be able to handle different kinds of resources in a uniform way is a big advantage, though it is not an easy task. As the first step, the ICE project tries to expose the reusable assets in our company by applying the CBSE theory. Here are the main drawbacks derived from the ICE project:

- Convincing the management that the initial costs for a framework like ICE will pay off in the long run is very hard.
- The B.A.D framework is based on COM and Windows NT. This means that we must port the framework if we want to use other development platforms (e.g. Linux).
- Many types of reusable assets in a company are not easily mapped to components. For example, applications and utilities. A big problem with this is that it can take a lot of effort to make the actual mapping, thus wasting a lot of time and money.
- Making the B.A.D framework general enough to accept all types of "components" will make it very complex and difficult to extend. The trick is to find a good balance between what should be mapped to components and the framework adaptation.

8 REFERENCES

[1] The Component Object Model Specification. URL: http://msdn.microsoft.com/library

[2] Microsoft Visual Component Manager. URL: http://msdn.microsoft.com

[3] INSEAS – Intelligent Search Agent for Software Components. Seoyung Park, Chisu Wu.

[4] Agora: A Search Engine for Software Components. Robert C. Seacord, Scott A. Hissam, Kurt. C. Wallnau.

[5] Human, Social and Organizational influences on Component-Based Software Engineering. Douglas Kunda, Laurence Brooks.

[6] The Capability Maturity Model. Mark C. Paulk, Charles V. Weber, Bill Curtis, Addison Wesley Pub Co, ISBN: 020 154 66 47

[7] Component Software, Beyond Object Oriented Programming. Clemens Szyperski, Addison Wesley Pub Co, ISBN: 0-201-17885-5