1 Basic Concepts of Component-based Software

To do

<table>
<thead>
<tr>
<th>Discuss Binding?</th>
<th>Yes – only concepts, not mechanisms</th>
<th>To do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trading licensing</td>
<td>Seems not a core topic – maybe somewhere later</td>
<td>Elsewhere</td>
</tr>
<tr>
<td>Connectors</td>
<td>Yes – seems key concept</td>
<td>To do</td>
</tr>
<tr>
<td>Wrapper/adaptor</td>
<td>Better later</td>
<td>Elsewhere</td>
</tr>
</tbody>
</table>

1 Basic Concepts of Component-based Software ................................................................. 1
1.1 Component Models and Components ........................................................................ 2
1.2 Component Forms in Components Models................................................................ 4
  1.2.1 Component Execution Platform....................................................................... 12
1.3 Architecture and Component Based Software Engineering................................. 15
  1.3.1 Architecture and Component Based Software Engineering............................. 15
  1.3.2 Product-line Architectures............................................................................. 17
1.4 Rules of the CBSE Game ......................................................................................... 19
1.5 Summary .................................................................................................................. 20
1.6 Review & Assignments............................................................................................. 20
  You should know ....................................................................................................... 20
  You should be able to ............................................................................................... 21
  Assignments ............................................................................................................... 21

Prerequisites

<table>
<thead>
<tr>
<th>Some familiarity with programming languages</th>
</tr>
</thead>
</table>

Date printed: 9-Feb-07
In the field of component-based software engineering specific terminology has emerged to describe the important concept of this field. The goal of this chapter is to introduce this terminology and explaining what concepts they denoted. However, a detailed examination of how these concepts are worked out using specific mechanisms is subject of later chapters. [These concepts will be illustrated with examples from specific technologies?]

Before we explain the concept of a software component, we first need to explain the notion of a component model. We subsequently discuss interfaces, specifications, component implementations and component instances. We explain that different forms of components exist and that these forms are constructed and used in different phases of development. The next key concept is binding. When software is developed in a component-based manner, then almost inevitably also a software architecture is used. We explain their complementary relation in section 1.3. The explanation of the basis concepts will show that different variants of component-based software development are possible. A collection of common choices among the possible variants of CBSE is presented as the rules of the game in section 1.4, After reading this chapter you should know the basic concepts of component based software engineering and some typical rules for their use.

### 1.1 Component Models and Components

Lego is often taken as an example of a component-based approach. Lego provides a set of building blocks in a large variety of shapes and colours. Lego is sold in boxes that contain a number of blocks that can be composed to make up toys such as cars, trains or airplanes. In order to enable their composition, all Lego-blocks provide small cylindrical stubs on the top and complementary holes on the bottom of the shape. The conformance of the blocks to this convention ensures that blocks, possibly from different Lego boxes, can be combined. Moreover, they can be combined into constructions other than the construction suggested by the manufacturer. Hence, Lego-blocks have the characteristic that they are easily composable, and are generic. These are characteristics that we also look for in software components.

![Figure 1 Example of a Lego-block ‘component’](image)

There are however, much more toys that consists of building blocks that can be assembled. Consider: meccano and traditional jigsaw-puzzles. While each of these can be combined with blocks of the same type (e.g. meccano-meccano), they cannot be combined with blocks of another type (e.g. meccano-lego). This illustrates that composability of building blocks is related to the conformance to a set of conventions.
In the world of engineering, the conventions of building blocks are chosen such that the blocks can be used for constructing systems with certain properties. For example, a racing car requires parts that are light-weight and aerodynamic in order to obtain an overall system that is fast. An army tank requires parts that are tough in order to make a robust system. The same holds for software components. Software components encapsulate functionality in a form that conforms to a set of conventions. These conventions ensure composability and determine the properties of the systems that can be built using the particular type of components. Rather than physical properties like weight and shape, software components are required to have properties like computational efficiency, resource efficiency and reliability.

The important issues are that components conform to a standard. This standard ensures composability of components. Ideally, the manner of composition is such that the overall systems satisfy certain properties. These issues are captured in the following definitions:

**Definition** A *Component Model* defines standards for

1. properties that individual components must satisfy
2. methods, and possibly mechanisms, for composing components

Informally, a component model defines the types of building blocks and the recipe for putting these building blocks together. This definition stresses a generic constraint: components are composable.

**Definition** A *Component* is a building block that conforms to a component model.

These definitions are very general – they apply both to software components and to physical components. In the remainder of this book the term component model will by default refer to software components.

For software components we can provide a more specific definition:

**Definition** A *Software Component Technology* is the implementation of a component model by means of:

- standards and guidelines for the implementation and execution of software components
- software tools that supports the implementation, assembly and execution of components.

There are many examples of software component technologies. Some widely known ones are Enterprise JavaBeans (EJB), COM+, .NET [Pla03], OSGi and the CORBA-Component Model (CCM). Each of these technologies embodies some underlying component model. Let’s zoom in on the .Net component technology as an example. Implementation support for
.Net is provided by the Visual Studio .Net-toolset which includes dedicated editors and compilers. Run-time support is provided by a run-time execution platform (called the common language run-time, CLR) that runs on top of the regular operating system.

**Definition** A **Software Component**
- implements some functionality
- has explicit provides and required interfaces
- the only means of communicating with a component is through its interfaces
- the forms of the interfaces and means of communication conform to an encompassing software component model.

The composability requirement of the general definitions has been translated into clauses that require standardization of interfaces and interaction. These conditions may be sufficient to ensure composability at the technical level of the component implementation. However, much more can be said about properties that are generally desired from software components, such as component granularity, encapsulation, cohesion, or testability. These additional properties are related to aspects of design and development of components and will be discussed in Chapter 9.

Although the definition of component model allows a lot of freedom, in practice there is a lot of commonality amongst component models. Comparing component-based software approaches shows that different component technologies standardize conventions for different stages, and sometimes even cover multiple stages. In the next section we show how the standards defined by a component model are related to different stages of the development.

### 1.2 Component Forms in Components Models

When we look at the lifecycle of a component, it passes through the following global stages (depicted in Figure 3): development, packaging, distribution, deployment and execution. In the development stage, the design, specification, implementation and meta-data of components is constructed. In the packaging stage, all information that is needed for trading and deployment of the component implementation grouped into a single package. The distribution stage deals with searching, retrieval and transportation of components. For searching components meta-data is needed that need to be included in the packaging stage. The deployment stage address issues related to the integration of component implementations in an executable system on some target platform. Finally, the execution stage deals with executing and possibly upgrading components.

![Figure 3 Stages of a component lifecycle](image-url)
Across these different stages of their lifecycle, components are represented in different forms. In the development stage components may be represented by means of a design- or specification language (e.g. a set of UML diagrams) and as a set of source-code and configuration files (e.g. a directory with .c and .h files for C, or a set of class files in case of Java) that together can be turned into executable code. In the packaging stage the files that together form a component are bundled into a single piece; for example in a compressed file (e.g. a zip-file). In the distribution stage packaged components are represented in a format that can be transmitted across a network or stored on some physical carrier (such as a disk or memory stick). At the deployment stage, components are unpackaged in a form that can be installed onto a target machine. In the execution stage, components take the form of blocks of code and data in the memory of a processor and cause a set of actions on this processor. All these forms are different views on or manifestations of what is logically a single building block.

Cheesman and Daniels introduced convenient terminology for denoting the different forms of components [CD2000]. Figure 4 shows the terms that they introduced and the relations between them. The relation between these forms and stages of development is shown in Figure 5. Next, we explain each of these forms in more detail.

Figure 4 Characterization of Component Forms
Component Interface

An *interface* defines the actions through which pieces of software may communicate with each other. The actions are performed according to some protocol such as request-response (method-calling), or message passing (signalling). Interfaces are offered by one piece of software in order to be used by other pieces of software. The piece of software that offers an interface is responsible for realizing the actions of the interface. The pieces of software that use the actions of an interface only need to know what action achieves, not how this is achieved. In this sense, users of an interface are shielded from the machinery that is used to implement it. This shielding is a means of abstraction – details about the realization are omitted. As a result of this abstraction, different implementations of an interface may be used to realize some action without users of the interface being aware of it.

A typical use of this is to replace a component by a newer version. This is illustrated in Figure 6. This figure shows that component P can be replaced by component P' if P' realizes the same interface Ix. Component R that uses interface Ix does not need to be changed.

The component that offers an interface is called the *provider* of that interface, the component that uses the interface is called the *requirer* of the interface. The most commonly used protocol according to which components execute actions is request-response. According to this style, the action of the provider promises some response to the requirer. In the context of the request-response style, the provider of an interface is referred to as *server* or *callee* and the requirer of the interface is referred to as *client* or *caller* or *user*. Remember, however, that this terminology is biased towards the request-response style of interaction.
In the CORBA Component Models, requires interfaces are called receptacle and provided interfaces are called facet.

In general, a component can have multiple interfaces. Each interface is a grouping of actions that belong together. The actions are defined by their signature. A signature is defined by the names of the actions together with their parameters and the types of these parameters.

**Example of Interface**

**Example** An example of an interface that is often found in the software of audio/video devices is one that offers play, stop, pause, rewind (rew) and fast-forward (ffwd). These are actions that logically belong together.

```c
interface RenderingControl
{
    Start();
    Stop();
    Pause();
    FastForward();
    Rewind();
}
```

![Figure 7 Interface of Audio-Video device](image)

An important principle in CBSE is that all dependencies of components are explicit. This is done by explicitly listing the dependencies as required interfaces. The use of required interfaces distinguishes components from traditional libraries and objects where the dependencies are hidden in the implementation code.

![Figure 8 Required Dependencies between Objects are implicit](image)

The implicit dependencies of pieces of software are an impediment to their reuse. The explicit declaration of dependencies of components in terms of interfaces makes dependencies independent from any specific implementation. As a result interfaces are an important unit of reuse and are treated as first-class citizens in CBSE. Interfaces are units of structuring in their own right; interfaces have their own name, are defined in separate files, and are subject of version control independent from any implementation.
Some programming languages such as Java and C# provide specific *constructs* for defining interfaces. These are then defined in the same syntax as the host programming language. Some component approaches, for instance CORBA, aim to be independent from any specific implementation language. This is achieved by defining a language for defining interfaces. Such a language is called *interface definition language (IDL)*.

The critical role that interfaces play requires that they be designed very carefully. In particular, interfaces should be stable over long periods of time. The design of stable interfaces is difficult because it requires some insight into possible future uses.

Some component models define a navigation interface. This interface enables other software to find the interfaces and operations that a component provides.

For instance, the COM component model defines that all components must implement the IUnknown interface. The IUnknown interface consists of three operations, one of which is the QueryInterface operation. Through this way COM allows clients to dynamically discover (at run time) whether or not an interface is supported by a component object.
javax.ejb

Interface EJBHome

public interface EJBHome
extends java.rmi.Remote

The EJBHome interface is extended by all enterprise Bean's home interfaces. An enterprise Bean's home interface defines the methods that allow a client to create, find, and remove EJB objects.

Each enterprise Bean has a home interface. The home interface must extend the javax.ejb.EJBHome interface, and define the enterprise Bean type specific create and finder methods (session Beans do not have finders).

The home interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBMetaData</td>
</tr>
<tr>
<td>HomeHandle</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>void</td>
</tr>
</tbody>
</table>

| Table 1 Home interface as described by SUN in its J2EE framework (SDK version 1.2.1) from http://java.sun.com/j2ee |

To Do:
- how about IUnknown & QueryInterface in COM

Component Meta-data and Specification

Component meta-data is any type of data about a component, but not necessarily related to its implementation. Examples are
- information about the company that built the component
- development information, such as the version or known bugs
- trademark and copyright information
- documentation about the component (help file, configuration information)

In .Net a composition of components is called an **assembly**. **Metadata** of an assembly is a comprehensive, standard, mandatory, and complete way of describing what is in an assembly. Metadata describes what types are available in the assembly (classes, interfaces, data types, etc.) and their containing namespaces, the name of each type, its visibility, its base class and so on. The metadata is generated automatically by the high-level compiler directly from the
Component specifications describe properties that are, or should be, realized by the corresponding component implementation. Many different properties may be important in a specification. Some essential properties are:
- functionality (what a component does),
- behaviour (the order of actions a component performs),
- interaction potential (the ways according to which a component can interact with other software),
- quality properties such as performance and reliability.
These properties should be specified in a complete, precise and verifiable manner. We will discuss approaches for component specifications in more detail in Chapter 4.

The purpose and scope of a component specification differ from the purpose and scope of a component interface. The most important difference is that specifications define a realization contract, while interfaces describe usage contracts. During development of a component, the specification is a prescription of what the implementation should realize. As such, the specification may include the definition of interfaces. Once the implementation of a component is finished, a specification describes what architects/developers who wish to use a component may want to know about it. Here, the specification is used for understanding the component. To this end, a specification may include a definition of the information model that a component operates upon. This information model should however be abstract, such that it allows the necessary degree of freedom in its implementation.

An interface represents a usage contract. It defines the actions through which it can interact with other components during execution; i.e. the actions that it can perform for other components and the actions that it needs other components to perform.

Furthermore, a component specification is different in scope: if a component has multiple interfaces, these are all listed as part of the specification. The scope of a specification is the component as a whole. As such it needs to relate the way in which the interfaces of a component relate to each other.

![Figure 9 differences between a component interface and a component specification](image-url)
**Component Implementation**

A *component implementation* can be deployed and executed in a Component Execution Environment. A Component implementation is a realization of a Component Specification.

There have been some discussions on how strict the hiding of internals of components should be. The black-box-requirement on components has been criticized as being too strong a constraint. For the purpose of tailorbility or testability, components should disclose more of their internals. Several variants of black box have been proposed that referred to the colours of components: black-box, grey-box, glass-box and white-box are used to denote the degree to which the internals of a component can be inspected and changed by parties other than its developer.

- **Black-box component**: Only the specification and the contract of the component are available to the user/integrator of a component.
- **Glass-box component**: the internals of a glass-box component may be inspected, but not modified. The implementation can thus add information to (or unfortunately often replaces) the specification of the component.
- **Grey-box component**: Part of the internals of a component may be inspected and disciplined extension or modification is possible; e.g. only certain methods may be defined by the user/integrator.
- **White-box**: A white-box component is open to both inspection and modification by its user/integrator.

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Changeable</th>
<th>Provided Port</th>
<th>Required Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>provided port</td>
<td>required port</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>provided port</td>
<td>required port</td>
</tr>
</tbody>
</table>

**Component Package**

A component package is a unit of distribution. A package contains the implementation of a component, relevant data files, meta-data about the component and meta-data about the package. The meta-data of a package contains a table of content of the files of the package. It
may be compressed to improve download-time and encrypted and certified to improve security. It can be compare to the way that a .zip-file can be used for packaging a collection of files into a single unit.

And last, but not least we come to the executable form of the component.

**Component Instance**
This is the piece of executable code and associated data in the target machine. The relation between a Component Implementation and a Component Instance is like the relation between a Class and an Object.

The component instance will form part of the execution architecture of the running system. The execution architecture is part of the software architecture of a system. The execution architecture determines the performance of a system by defining the mapping of functionality onto run-time entities (such as processes and resources) in a system. The component instance will have to conform to conventions that are dictated for this purpose by the software architecture.

Facilities that are needed for adding and removing components at run-time and for enabling their interaction are the key components of the Component Execution Platform.

### 1.2.1 Component Execution Platform
In order to run a program on a machine certain facilities have to be in place. For a regular program these facilities are provided by the operating system. It can load programs into the memory of a machine and it can then start and stop the execution of that program. A Component Execution Platform (CEP) provides analogous facilities but then at the level of components.

| A component execution platform is to components what an operating system is to applications |

Basic services that a CEP provides are:
- registration and retrieval of component instances,
- life-cycle management: the creating (or launching) and deletion of instances in the run-time system.
- in case of distributed systems:
  - location transparency: hiding the location where a component resides.

More advanced services of a CEP may include:
- quality of service: managing the sharing of resources among multiple component instances,
- managing transactions across component instances,
- providing security facilities,

These facilities are offered to components via a docking interface.

Component models that are designed to fit into systems with little CPU and memory resources try to minimize the overhead of a CEP. These component models avoid the need for a CEP and use the services offered by an operating system.
Component models that take components to be source code at some higher abstraction level require that the CEP provides an interpreter for this language. For example, in EJB, the Java Virtual Machine can be considered part of the CEP.

To Do:
- example of CEP: CORBA

In the .Net component model, the CEP is called the Common Language Runtime (CLR). The CLR is a runtime environment that manages the execution of .NET program code, and provides services such as memory and exception management, debugging and profiling, and security. It also enables interoperability between different programming languages by standardization of the type system (called Common Types System). The .Net CLR builds on functionality provided by the COM+ component model as well as the Windows operating system.

The CORBA Component Model, the emphasis is on connecting existing applications that are running on different computers. Hence, unlike .Net, it does not address issues concerning the execution of code and its memory management – these are left to the native operating system. Instead, CORBA focuses on facilitating smooth interoperation across networks. To this end the CORBA execution platform offers services for calling remote invocation of operations, remote creation and deletion of objects, a naming service for referring to remote objects, a security service, a transaction service.
In every component model, components have a special interface for connecting to the Component Execution Platform (CEP). This interface is called the docking interface. This interface does not offer any application functionality, but is used only for interaction between a component and the CEP. This interaction consists at least of the registering of a component with the CEP so that the CEP knows that the component is available in the system. This registering is a specific type of binding. Depending on the component model, the CEP may provide more functionality such as resource management, transaction management for it may interact with components. Some component models do not use a docking interface; in this case the binding between components and their execution platform is handled via the operating system.

**Figure 11 docking interface and usage interface**

Another

**Concluding Remarks**
This section illustrates that it is generally not sufficient to talk about ‘component’, but that you should use some adjective to make clear what it is exactly that you mean.
1.3 Architecture and Component Based Software Engineering

In order to successfully develop systems in a component-based manner, an architectural plan needs to be in place that organizes how components fit together and how, once assembled, a system meets its quality requirements. In this section we explain the notion of architecture of a software system and then discuss its role in CBSE.

1.3.1 Software Architecture

The complexity of the systems that are being developed has been increasing very much over the last decades. It is not uncommon that systems consist of over millions of lines of program code and are being developed by tens or hundreds of programmers over several years. In order to manage the complexity of such large systems, they are decomposed into subsystems such that each of these subsystems can be understood largely in isolation. Together with this decomposition, policies for the interaction and collaboration between the subsystems are determined to ensure that when they are put together they meet the systems’ requirements. The principles that govern this decomposition and the resulting design are called the software architecture of a system. Some sets of principles reoccur in many systems and are called architectural styles. Examples of architectural styles are client-server and pipe-and-filter. Garlan gives a nice introduction to software architecture in [Ga01].

In addition to components, connectors have been advocated as element of architecture descriptions for capturing the logic of interaction between components. These connectors exist separately from components. A number of arguments have been put forward in favour of having components:
If one considers components to be a mechanism for construction more complex computations out of more basic computations, then an analogous mechanism should exist for building more sophisticated patterns of interaction out of more basic patterns of interaction. Because patterns of interaction address another concern than communication, they deserve a separate, dedicated means of representation.

Separating communication mechanisms from components by means of connectors improves several quality properties of the architecture and its components:
- understandibility and maintainability of the architecture: connectors localize information about interactions of components in a system (this information is no longer spread over all communicating components and therefore it is easier to change).
- reusability of components (the same component can be used in a variety of environments with different communication primitives).
- analyzability of the system: the patterns of communication can be studied in isolation from the computations that a system performs.

One of the earliest advocates of using connectors as first-class citizens in software architectures is Mary Shaw [Sh+95].

**1.3.2 Architecture and Component Based Software Engineering**

In an ideal case, the development may be driven by an architecture that is developed up front. Based on this architecture a component model is selected or a dedicated component model is developed such that the rules of the component model match those of the architecture. Examples of dedicated component models are the Koala model [ref] developed for meeting resource constraints in consumer devices, the SaveCCM [ref] model developed for meeting dependability requirements in the automotive domain. As these examples illustrate dedicated component models are developed in domains that have rather specific extra-functional requirements.

Developing a proprietary component model takes a large amount of effort. Therefore, a practical alternative is to select the architecture and component model in concert. To this end, a set of candidate architectures and a set of component models is selected. Choosing some component model limits the possible architectures that can be realised. If a component model is not an ideally match with the target architecture, then some additional work is needed to provide extra features or eliminate (or hide) superfluous features. Summarizing: an architecture constrains the possible component models, and vice verse: a component model constrains the possible architectures.

[Diagram]

- **Reference Architecture**
  - prescribes
  - conforms to
  - consists of *

- **Component Model**
  - prescribes
  - conforms to
  - implemented by
1.3.3 Product-line Architectures

A trend that is encouraging CBSE and reuse is the increasing orientation towards product-lines or product-families. A software product line is a collection of software systems that share a significant set of features. Typically these systems are built from a common set of software artifacts, such as a reference architecture and a set of (tailorable) components that provide common services. The architecture should facilitate the easy selection and/or tailoring of optional parts. Examples of product-lines can be found in the domains of administrative systems (accounting, logistics/stock-control) as well as in consumer electronics and in technical systems for specialized domains such as in-car systems, avionics and medical scanner systems.

A definition of software product line:

**Definition** A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

Such a product line needs an architecture that facilitates the exploitation of commonalities. This is achieved by a product-line architecture:
**Definition** Product-line architectures (PLAs) are designs for families of related applications; application construction is accomplished by composing reusable components. Evolution occurs by plugging and unplugging components that encapsulate new and enhanced features [Batory 1998; Bosch 1999; Czarnecki and Eisenecker 1999; Software Engineering Institute 2001; Weiss and Lai 1999].

The more investment put into the architecture the more possible reuse of components can be achieved.

![Diagram](image.png)

Figure 13 Example product family – courtesy of Philips (pending)

### 1.3.4 Development Styles of CBSE

Depending on the dominance of the architecture or of availability of components, several different styles of CBSE are possible. More details about the processes of these style are given in Chapter 7.

- **COTS-based CBSE**
  
  An architecture is designed such that existing components are fit in the architecture. These components may be available commercially or from open source. A key characteristic is that the integrator of the component has little or no influence on the design and implementation of the component that it aims to integrate. The degree by which the architecture is adapted to accommodate COTS components depends on strategic objectives. This style of CBSE is common to software development in the domain of information systems. Here systems are typically custom made (one instance of the system for one specific customer) and maintained over a longer period of time.
Typical COTS components are: Database systems, Middleware, transaction monitors, security frameworks.

- **Architecture-based CBSE**
  An organisation defines an architecture for a system that it wants to use. This architecture divides a system up into a number of subsystems. These subsystems are developed and maintained independently.

- **Product-line based CBSE**
  A product-line is aimed at developing a set of systems based on a significant degree of commonality in their implementation, yet with a set of distinguishing features. In this approach, an product-line architecture is developed that supports this goal. The components defined by such an architecture are generally custom-defined, but may also include COTS components. Once a product-line is into use for some time, there is quite limited freedom in evolving the architecture because there are typically many products that depend on it.
  One approach to product-line architectures is to develop a platform that provides common services and a number of places where different features can be incorporated. The set of features can be relatively fixed, or more open-ended.

### 1.4 Rules of the CBSE Game

In this section we summarize a number of principles that are generally believed to hold for CBSE.

- **A component is a ’black box’- you cannot change the internals of components.**
  Components are available for assembly (or composition). In some cases the provider of a component can prevent other parties from modifying their component; e.g. by providing only an executable. The reason he might want to do so, is to protect the intellectual property that is key for the working of a component.

  There are also economic reasons that suggest that the effort that is needed to modify a component (especially the effort needed for understanding how a component works internally) easily exceeds the effort needed for developing a component by yourself. This places a burden on the designer of component to make them offer exactly the right functionality and the right means for configuring components to different contexts.

- **You don’t know what other software your component will be communicating with.**
  It is up to the assembler of a system to determine how to connect up your component to other software.

  A consequence for the design of components is that in order to allow the use of a component in as wide a range of contexts, a component may not assume that it is used in a proper manner, but needs to enforce its proper use itself.

  Consider for example, a component that offers a square root service. It can operate only on non-negative integer values. The implementation of the service in terms of a procedure requires an integer value to be passed. The implementation must check that this integer value is non-negative. If the input parameter is negative, then the

  Similarly, after receiving results from a request to a service, the receiving component should not assume that the results satisfy the post-condition, but check for this itself.

In general
a service provider may not assume that the incoming parameters satisfy its precondition, but must check for this itself.

- a service requestor may not assume that results from a service satisfy the promised post-condition (if it can find this out at all). The requestor must check whether they match its expected post-condition.

A component that receives data should be capable of handling data that does not satisfy the conditions the component expects. Ideally, this service that the component provides should be compromised as little as possible when such illegitimate data arrives.

- **All dependencies of components must be explicit**

  Again this is important because party other than the original developed will have to use the component. The pitfall is that the original developed makes some assumptions that he thinks are so common that they do need to be specified explicitly, yet in other projects it may be necessary to deviate from conventions. Without all dependencies being explicit, it may be impossible for an integrator to obtain a proper working component.

- **The weaker the dependencies between components in a system, the easier it is to adapt and evolve the system.**

  The fact that parties that develop components do not know with what other components it will be assembled, has consequences for the testing: in contrast to complete in-house development, a component can not be tested in the context in which it will have to operate.

**Reflection**

Different project have different requirements. Generally, it seems reasonable to select the weakest set of constraints that enable you to fulfil the requirements. For CBSE this means you should consider whether the aforementioned rules are relevant for your situation. If they are not, don’t bother trying to adhere to them.

### 1.5 Summary

There are many types of software components because different contexts impose different sets of requirement on component-based systems. As a result the individual building blocks are designed to have different properties. The rules and conventions that individual components must meet in order to enable the composition of independently developed components are defined by a component model.

### 1.6 Review & Assignments

**You should know**

- The most important concepts of CBSE: component model, component, component interface, component specification, component implementation, component instance, binding
- why there are different types of component models
- CBSE is not tied to a particular implementation technology such as an programming language, operating system or middleware, although any for any specific implementation technology there are dependencies between these.
- what aspects there are to binding (exogenous, binding time)
You should be able to
- detect whether CBSE terminology is used accurately or not.
- determine the binding-mechanism used in a component model
- …

Assignments

1. The definition of a component model suggests that a component-infrastructure is an indispensable part of it. Are component-infrastructures also present in electrical and civil component-oriented approaches? Are these infrastructures indispensable in those domains?
2. Why are there many types of component models? How can it be that different component models are mutually incompatible?
3. What is the relation between components and architecture?
4. How can a component model determine the properties of the systems that are built using the components?
5. Putting two stone bricks or blocks of Lego together does not again yield a stone brick or a Lego block, but the result of composition is again composable. What is happening here? What is kept invariant by composition? What are the similarities and differences in composing Lego blocks and composing of software components?

References

http://en.wikipedia.org/wiki/Software_component