CHAPTER FOUR

What a component is and is not

The terms "component" and "object" are often used interchangeably. In addition, constructions such as "component object" are used. Objects are said to be instances of classes or clones of prototype objects. Objects and components are both making their services available via interfaces, and interfaces are of certain types or categories. As if that was not enough, object and component interactions are described using object and component patterns and prescribed using object and component frameworks. Both components and frameworks are said to be whitebox or blackbox, and some have even identified shades of gray and glassboxes. Language designers add further irritation by also talking about namespaces, modules, packages, and so on.

This plethora of terms and concepts needs to be reduced by eliminating redundancies or it needs to be unfolded, explained, and justified. The next section considers this universe of terms and concepts and provides brief explanations, relating the concepts to each other. The goal is to establish some degree of order and intuition as a basis for further discussions. Then, a refined definition of the term "component" is presented and discussed. Finally, the linkages to standards for horizontal and vertical markets are summarized.

4.1 Terms and concepts

Some degree of familiarity with most of the terms covered in this section is assumed – and so is some degree of confusion about where one term ends and another starts. One way to capture the intuitive meaning of a term is to enumerate characteristic properties. The idea is as follows: something is an A if it has properties a1, a2, and a3. For example, according to Wegner's (1987) famous definition, a language is called object-oriented if it supports objects, classes, and inheritance.

Unfortunately, the concepts relevant to component technology encompass many aspects. The massive overloading of the term object is the best example. Over time, the notions of module, class, and component have all become embraced by the term "object." (More recently, the same is done to the term

"software component" – even to a point where plain old objects are now called components!) Combining several terms into one can simplify things superficially, but not to good advantage beyond the simplest thoughts. As precision and richness of the vocabulary decrease, so does the richness of expressible and distinguishable, yet concise, thoughts. It is essential to strive for a balance, preserving conciseness and intuition. The following subsections thus present definitions of some key terms and relate them to each other.

4.1.1 Components

The characteristic properties of a component are that it:

- is a unit of independent deployment;
- is a unit of third-party composition;
- has no (externally) observable state.

These properties have several implications. For a component to be independently deployable, it needs to be well separated from its environment and other components. A component, therefore, encapsulates its constituent features. Also, as it is a unit of deployment, a component will never be deployed partially. In this context, a third party is one that cannot be expected to have access to the construction details of all the components involved.

For a component to be composable with other components by such a third party, it needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces.

Finally, a component should not have any (externally) observable state – it is required that the component cannot be distinguished from copies of its own. Possible exceptions to this rule are attributes not contributing to the component's functionality, such as serial numbers used for accounting. The specific exclusion of observable state allows for permissible technical uses of state that can be crucial for performance without affecting the observable behavior of a component. In particular, a component can use state for caching purposes (a cache is a store that can be eliminated without any consequence, except possibly reduced performance).

A component can be loaded into and activated in a particular system. However, due to the stateless nature of components, it makes little sense to have multiple copies in the same operating system process as these would be mutually indistinguishable anyway. In other words, in any given process (or other loading context), there will be at most one copy of a particular component. Hence, it is not meaningful to talk about the number of available copies of a component.

In many current approaches, components are heavyweight units with exactly one instance in a system. For example, a database server could be a compo-

nent. If there is only one database maintained by this class of server, then it is easy to confuse the instance with the concept. An example would be the payroll server of a company. For example, the database server, together with the database, might be seen as a module with an observable state. According to the above definition, this "instance" of the database concept is not a component. Instead, the static database server program is, and it supports a single instance – the database "object." In the example, the payroll database server program may be a component, while the payroll data is an instance (an object). This separation of the immutable "plan" from the mutable "instances" is essential to avoid massive maintenance problems. If components were allowed to have observable state, then no two installations of the "same" component would have the same properties.

It is important to avoid a common confusion at this point. The component concept argued for here does not in any way promote or demote the use of state, observable or not, at the level of objects. Also, it is unrelated to the lifetime of such object state (per call, per session, or persistent). These are all object-level concerns that are not tied to the component concept, although components can be used to provide objects of any of these natures.

4.1.2 Objects

The notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, the characteristic properties of an object are that it:

- is a unit of instantiation, it has a unique identity;
- may have state and this can be externally observable;
- encapsulates its state and behavior.

Again, a number of object properties directly follow. Because an object is a unit of instantiation, it cannot be partially instantiated. Since an object has individual state, it also has a unique identity that suffices to identify the object despite state changes for its entire lifetime. Consider the apocryphal story about George Washington's axe. It had five new handles and four new axe heads, but was still George Washington's axe. This is a good example of a real-life object of which nothing but its abstract identity remained stable over time.

As objects are instantiated, there needs to be a construction plan that describes the state space, initial state, and behavior of a new object. Also, that plan needs to exist before the object can come into existence. Such a plan may be explicitly available and is then called a class. Alternatively, it may be implicitly available in the form of an object that already exists – that is, sufficiently close to the object to be created, and can be cloned. Such a pre-existing object is called a prototype object (Lieberman, 1986; Ungar and Smith, 1987; Blaschek, 1994).

Whether using classes or prototype objects, the newly instantiated object needs to be set to an initial state. The initial state needs to be a valid state of

the constructed object, but it may also depend on parameters specified by the client asking for the new object. The code required to control object creation and initialization can be a static procedure – usually called a constructor if it is part of the object's class. Alternatively, it can be an object of its own – usually called a factory object, or factory for short if it is dedicated to this purpose. Methods on objects that return freshly created other objects are another variation – usually called factory methods.

4.1.3 Components and objects

Obviously, a component is likely to act through objects and therefore would normally consist of one or more classes or immutable prototype objects. In addition, it might contain a set of immutable objects that capture default initial state and other component resources.

However, there is no need for a component to contain classes only, or even to contain classes at all. Instead, a component could contain traditional procedures and even have global (static) variables (as long as the resulting state remains unobservable), or it may be realized in its entirety using a functional programming approach, or using assembly language, or any other approach. Objects created in a component – more precisely, references to such objects – can leave the component and become visible to the component's clients, usually other components. If only objects become visible to clients, there is no way to tell whether or not a component is "all object-oriented" inside.

What, then, is the difference between state maintained by objects created by a component and state maintained by a component? This is a subtle but critically important point. State maintained by an object is abstracted by that object's reference. A component that does not maintain observable state cannot (observably) maintain references even to the objects it created. A reference to the component itself (the component's fully qualified name) cannot be used to retrieve any objects. Interestingly, this property can be achieved in a non-object-oriented setting. A functional component can create closures and a procedural component can maintain tables of stateful records that are only manipulated in table indices, which themselves are not kept by the component. Whether or not any such state (in objects, closures, or tables) is persistent across component activations is a separate question, the correct answer to which depends on the intended use of a particular component.

A component may contain multiple classes, but a class is necessarily confined to being part of a single component. Partial deployment of a class would not normally make sense. Of course, just as classes can depend on other classes using inheritance, components can depend on other components – this is an import relation.

The superclasses of a class do not necessarily need to reside in the same component as the class itself. Where a class has a superclass in another component, the inheritance relation between these two classes crosses component

boundaries, forcing a corresponding import relationship between the two underlying components. Inheritance of specifications is an essential technique for establishing correctness, as, by referring to the same specification, two components establish a common basis. Whether or not inheritance of implementations across components is a good thing is the focus of a heated debate between different schools of thought. The deeper theoretical reasoning behind this clash is interesting and close to the essence of component orientation. Further detail and arguments follow in Chapter 7.

4.1.4 Modules

From the discussions so far, it should be clear that components are rather close to modules, as introduced by modular languages in the late 1970s (Wirth, 1977; Mitchell *et al.*, 1979). The most popular modular languages are Modula-2 (Wirth, 1982) and Ada. In Ada, modules are called packages, but the concepts are almost identical. An important hallmark of truly modular approaches is the support of separate compilation, including the ability to type-check across module boundaries properly.

With the introduction of the language Eiffel, it was claimed that a class is a better module (Meyer, 1988). This seemed to be justified, based on the early ideas that modules would each implement one abstract data type (ADT). After all, a class can be seen as implementing an ADT, with the additional properties of inheritance and polymorphism. However, modules can be used, and always have been used, to package multiple entities, such as ADTs or, indeed, classes, into one unit. Also, modules do not have a concept of instantiation, whereas classes do. (In module-less languages, this frequently leads to the introduction of "static" classes that essentially serve as simple modules.)

In more recent language designs – such as Oberon, Modula-3, Component Pascal, and C# – the notions of modules (or assemblies in C#) and classes are kept separate. In all cases, a module can contain multiple classes. (In languages such as Java that do not have a separate module concept, modules can be emulated to a degree by using nested classes.) Where classes inherit from each other, they can do so across module boundaries. As an aside, it should be mentioned that in Smalltalk systems, it was traditionally acceptable to modify existing classes to build an application. Attempts have been made to define "module" systems for Smalltalk capturing components that cut through classes – for example Fresco (Wills, 1991). Composition of such modules from independent sources is not normally possible, though, and this approach is therefore not further followed in this book.

Unlike classes, modules can indeed be used to form minimal components. Even modules that do not contain any classes can function as components. A good example is traditional math libraries that can be packaged into modules and are of a functional rather than object-oriented nature. Nevertheless, one aspect of fully fledged components is not normally supported by module concepts.

There are no persistent immutable resources that come with a module, beyond what has been hardwired as constants in the code. Resources parameterize a component. Replacing these resources allows the component to be configured without the need to rebuild its code. For example, resource configuration can be used for localization. The configuration of resources seems to assign mutable state to a component. However, as components are not supposed to modify their own resources, resources fall into the same category as the compiled code that also forms part of a component. Indeed, it is useful to regard a localized version of a component as a different (but related) component. Tracking the relationship between a component and its derived localized versions is similar to tracking the relationship between different release versions of a component.

It is instructive to explore cases where modules do not qualify as components. Under the definition used here, components do not permit observable state, while modules can clearly be built to use global (static) variables to expose observable state. Furthermore, modules tend to depend statically on implementations in other modules by importing direct interfaces from other modules. For components, such static dependencies on component-external implementations are allowed but not recommended. Static dependencies should be limited to contractual elements, including types and constants. Dependencies on implementations should be relegated to the object level by preferring indirect over direct interfaces in module dependencies to enable flexible compositions using multiple implementations of the same interface.

To summarize, modularity is a prerequisite for component technology, but rules beyond the traditional modularity criteria are needed to form components rather than just modules. Many modularity criteria go back to Parnas (1972) and include the principle of maximizing cohesion of modules while minimizing dependencies between modules. Modularity is thus certainly not a new concept. Unfortunately, the vast majority of software solutions today are not even modular. For example, it is common practice for huge enterprise solutions to operate on a single database, allowing any part of the system to depend on any part of the data model. Adopting component technology requires adoption of principles of independence and controlled explicit dependencies. Component technology unavoidably leads to modular solutions. The software engineering benefits can be sufficient to justify initial investment into component technology, even when component markets are not foreseen in the mid-term.

4.1.5 Whitebox versus blackbox abstractions and reuse

Blackbox and whitebox abstraction refer to the visibility of an implementation "behind" its interface. In an ideal blackbox abstraction, clients know no details beyond the interface and its specification. In a whitebox abstraction, the interface may still enforce encapsulation and limit what clients can do, although

implementation inheritance allows for substantial interference. However, the implementation of a whitebox is fully available and can thus be studied to enhance the understanding of what the abstraction does. (Some authors further distinguish between whiteboxes and glassboxes, with a whitebox allowing for manipulation of the implementation and a glassbox merely allowing study of the implementation.)

Grayboxes are those that reveal a controlled part of their implementation. This is a dubious notion, as a partially revealed implementation could be seen as part of the specification. A complete implementation would merely have to ensure that, as far as was observable by clients, the complete implementation performs as the abstract partial one. This is the standard notion of refinement of a specification into an implementation. Indeed, specification statements can be seen as graybox specifications (Büchi and Weck, 1997).

Blackbox reuse refers to the concept of reusing implementations without relying on anything but their interfaces and specifications. For example, in most systems, application programming interfaces (APIs) reveal nothing about the underlying implementation. Building on such an API is equivalent to blackbox reuse of the implementation of that API.

In contrast, whitebox reuse refers to using a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation. Most class libraries and frameworks are delivered in source form, and application developers study the classes' implementation to understand what a subclass can or has to do.

The serious problems of whitebox reuse are analyzed in detail in Chapter 7. For now it suffices to say that whitebox reuse renders it unlikely that the reused software can be replaced by a new release. Such a replacement will probably break some of the reusing clients, as these depend on implementation details that may have changed in the new release.

A definition: software component

From the above characterization, the following definition can be formed:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This definition was first formulated at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one outcome of the Workshop on Component-Oriented Programming (Szyperski and Pfister, 1997). The definition covers the characteristic properties of components discussed before. It has a technical part, with aspects such as independence, contractual interfaces, and composition. It also has a market-related part, with aspects such as third parties and deployment. It is a property unique to components, not only in the software world, to combine technical and market aspects. (An

interpretation of this definition from a purely technical point of view is presented in Chapter 20.)

From a more modern point of view, this definition requires some clarification. The contract of a deployable component specifies more than dependencies and interfaces, it specifies how the component can be deployed, how, once deployed (and installed), it can be instantiated, and how the instances behave through the advertised interfaces. This latter aspect goes beyond a mere sum of per-interface specifications – an instance maintains an invariant that couples the per-interface specifications. In fact, the per-interface specifications need to be seen in isolation from any particular component that provides or requires an implementation of such an interface.

For example, consider a queuing component that requires stable storage via one interface and provides enqueue and dequeue operations via two further interfaces. The component contract states that what is enqueued via one interface can be dequeued via the other – a correlation that the individual interface's specifications cannot provide. The component contract also states that the component, once instantiated, can only be used by connecting it to a provider implementing the stable storage interface. This latter notion of connecting components that have matching provided and required interfaces needs to be effected by the composition rules of a supporting component model. The details of deployment and installation need to be supported by a specific component platform.

Interfaces are discussed in more detail in the following subsection. A discussion of context dependencies follows in the subsequent subsection.

4.1.6 Interfaces

Part One has already introduced the basic market aspects of component technology. Chapters 5–7 cover the aspects of interfaces, contracts, semantics, and composition in detail. For the following, more market-oriented, discussion, it suffices to consider the interface of a component to define the component's access points. These points allow clients of a component, usually components themselves, to access the services provided by the component. Normally, a component will have multiple interfaces corresponding to different access points. Each access point may provide a different service, catering for different client needs. Emphasizing the contractual nature of the interface specifications is important because the component and its clients are developed in mutual ignorance, so it is the contract that forms a common middle ground for successful interaction.

What are the non-technical aspects that contractual interfaces have to obey to be successful?

First, as mentioned in Part One, the economy of scale has to be kept in mind. A component can have multiple interfaces, each representing a service that the component offers. Some of the offered services may be less popular

than others, but if none is popular and the particular combination of offered services is not popular either, the component has no market. In such a case, the overheads involved in casting the particular solutions into a component form may not be justified.

Notice, however, that individual adaptations of component systems may well lead to the development of components that themselves have no market. In this situation, extensions to the component system should build on what the system provides, and the easiest way of achieving this may well be the development of the extension in component form. In this case, the economic argument applies indirectly in that the extending component itself is not viable, but the resulting combination with the extended component system is.

Second, undue fragmentation of the market has to be avoided as it threatens the viability of components. Redundant introductions of similar interfaces have thus to be minimized. In a market economy, such a minimization is usually the result of either early standardization efforts among the main players in a market segment or fierce eliminating competition. In the former case, the danger is suboptimality due to "committee design" and, in the latter case, it is suboptimality due to the non-technical nature of market forces.

Third, to maximize the reach of an interface specification and components implementing this interface, there need to be common media to publicize and advertise interfaces and components. If nothing else, this requires a small number of widely accepted unique naming schemes. Just as ISBN (International Standard Book Number) is a worldwide and unique naming scheme to identify any published book, a similar scheme is needed to refer abstractly to interfaces "by name." Just as with an ISBN, an interface identifier is not required to carry any meaning. An ISBN consists of a country code, a publisher code, a publisher-assigned serial number, and a checking digit. Although it reveals the book's publisher, it does not code the book's contents. Meaning may be hinted at by the book's title, but titles are not guaranteed to be unique.

An interesting variation on the theme of interface standardization is the standardization of message formats, schemas and protocols. Instead of formalizing interfaces as collections of parametric operations, the focus is on what is passed back and forth. This viewpoint is sometimes described as the "viewpoint of the wire" or as "wire formats," alluding to the importance of standardizing message schemas, formats, and protocols when interconnecting machines in a network. Standardization of message formats, schemas and protocols is indeed the main approach of internet (IP, UDP, TCP, SNMP, and so on) and web (HTTP, HTML, and so on) standards. To achieve broader semantic coverage, it is useful to standardize message schemas in the context of a single generic message format. This is the rationale behind XML, a single generic format, the large number of related standards (including SOAP and several XML web services standards), and the growing number of XML schema standardization efforts (see Chapter 18).

4.1.7 Explicit context dependencies

Besides the specification of provides interfaces (more commonly called required interfaces), the above definition of components also requires components to specify their needs. In other words, the definition requires specification of what the deployment environment will need to provide so that the components can function. These needs are called context dependencies, referring to the context of composition and deployment. They include the component model that defines the rules of composition and the component platform that defines the rules of deployment, installation, and activation of components. If there were only one software component world, it would suffice to enumerate requires interfaces (more commonly called required interfaces) of other components to specify all context dependencies (Magee *et al.*, 1995; Olafsson and Bryan, 1997). For example, a mail merge component would specify that it needs a file system interface. Note that, with today's components, even this list of required interfaces is not normally available. The emphasis is usually just on provides interfaces. (Note that the more common terms of provided and required interface aren't quite accurate. Interfaces sit between components and are, as such, neither required nor provided.)

In reality, there are several component worlds that partially coexist, partially compete, and partially conflict with each other. For example, today there are three or four major component worlds, based on OMG's CORBA, Sun's Java, and Microsoft's COM and CLR. In addition, component worlds are themselves fragmented by the various computing and networking platforms that they support. This is not likely to change soon. However, from another perspective, these worlds collapsed to only two – the CORBA+Java world and the Microsoft world (including COM+ and .NET/CLR). Yet, despite this apparent culmination in just two poles, there is a surprising diversity at the level of actual offerings, even including an open source effort to independently implement the CLI specification underlying CLR (www.ximian.com).

Just as the markets have so far tolerated a surprising multitude of operating systems, there will be room for multiple component worlds. In a situation in which multiple component worlds share markets, a component's specification of context dependencies must include its required interfaces as well as the component world (or worlds) that it has been prepared for. There will, of course, also be secondary markets for cross-component world integration. By analogy, consider the thriving market for power plug adapters for portable electrical devices. Thus, chasms are mitigated by efforts to provide bridging solutions. Examples are OMG's Interworking standard (part of CORBA since version 2.0, July 1996 revision), which forms a bridge between Microsoft's COM and CORBA, and OMG's EJB interoperation specification that has formed part of the CORBA Component Model since CORBA version 3.0. Nevertheless, such bridging will always compromise where the bridged worlds are too different for the gap to be closed fully. To a degree, it is even demonstrably impossible to bridge separate component worlds completely (Smith *et*

al., 1998). When having a single universal standard offers overwhelming benefits, a "shake-out" effect is likely to eliminate most competing standards, as happened with VCR standards (for a detailed discussion see Messerschmitt and Szyperski, 2002). Following the same example, the ongoing emergence of new media for video storage and recording, along with new standards (CD, CD-R, CD-RW, DVD, DVD-RAM, memory stick, compact flash, and so on), also suggests that such convergence is not necessarily durable in the presence of technological evolution.

4.1.8 Component "weight"

Obviously, a component is most useful if it offers the "right" set of interfaces and has no restricting context dependencies at all – in other words, if it can perform in all component worlds and requires no interface beyond those the availability of which is guaranteed by the different component worlds. However, only very few components, if any, would be able to perform under such weak environmental guarantees. Technically, a component could come with all required software bundled in, but that would clearly defeat the purpose of using components in the first place. Note that part of the environmental requirements lie with the machine that the component can execute on. In the case of a virtual machine, such as the Java VM, this is a straightforward part of the component world specification. On native code platforms, a mechanism such as Apple's "Fat Binaries," which packs multiple binaries into one file, would still allow a component to run "everywhere."

Instead of constructing a self-sufficient component with everything built in, a component designer may have opted for "maximum reuse." To avoid redundant implementations of secondary services within the component, the designer decided to "outsource" everything but the prime functionality that the component offers itself. Object-oriented design has a tendency toward this end of the spectrum, and many object-oriented methodists advocate this maximization of reuse.

Although maximizing reuse has many oft-cited advantages, it has one substantial disadvantage – the explosion of context dependencies. If designs of components were, after release, frozen for all time, and if all deployment environments were the same, then this would not pose a problem. However, as components evolve, and different environments provide different configurations and version mixes, it becomes a showstopper to have a large number of context dependencies. With each added context dependency, it becomes less likely that a component will find clients that can satisfy the environmental requirements. To summarize:

Maximizing reuse minimizes use.

In practice, component designers have to strive for a balance. When faced with requirements that specify the interfaces that a component should at least provide,

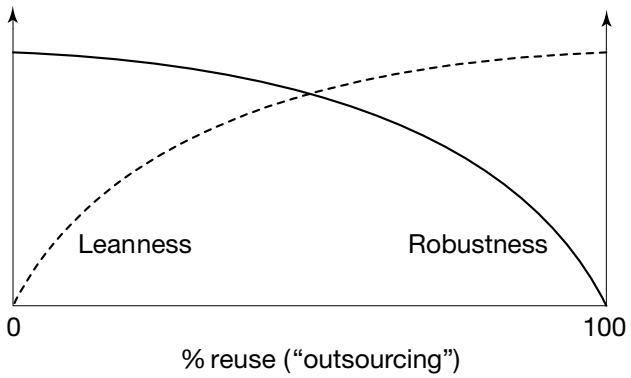

Figure 4.1 Opposing forcefields of robustness (limited context dependence) and leanness (limited "fat"), as controlled by the degree of reuse within a component.

a component designer has a choice. Increasing the context dependencies usually leads to leaner components by means of reuse, but also to smaller markets. Additionally, higher vulnerability in the case of environmental evolution must be expected, such as changes introduced by new versions. Increasing the degree of self-containedness reduces context dependencies, increases the market, and makes the component more robust over time, but also leads to "fatter" components. Figure 4.1 illustrates the optimization problem resulting from trading leanness against robustness.

The effective costs of making a component leaner, compared with making it more robust, need to be estimated to turn the qualitative diagram of Figure 4.1 into a quantitative optimization problem. There is no universal rule here. The actual costs depend on factors of the component-producing organization and of the target markets for the component. The markets determine the typical deployment environment and client expectations, including component "weight" and expected lifetime.

Note that it is not just coincidence that Figure 4.1 and Figure 1.1 (p. 6) are so similar. The discussion in this section focused on the "outsourcing" of parts of a component. In contrast, the discussion in Chapter 1 concentrated on the outsourcing of parts of a system – that is, the outsourcing of components. The former is about reuse across components, whereas the latter is about reuse of components.

4.2 Standardization and normalization

The "sweet spot" of the optimization problem introduced above can be shifted toward leaner components by improving the degree of normalization and standardization of interface and component worlds. The more stable and widely supported a particular aspect is, the less risky it becomes to make it a specified requirement for a component. Context dependencies are harmless where their support is ubiquitous. For example, only 50 years ago it would

have been a bad idea in many cases to form a business that depends on its customers having access to a telephone. Nowadays, in many areas of the world, this is clearly safe. (In some areas and cases it is now even safe to assume that customers have internet and web access. Assuming broadband connectivity will be next.)

4.2.1 Horizontal versus vertical markets

When aiming for the formation of standards that cover all areas representing sufficiently large markets, it is useful to distinguish standards for horizontal and vertical markets. A horizontal market sector cuts through all or many different market domains; it affects all or most clients and providers. A vertical market sector is specific to a particular domain and thus addresses a much smaller number of clients and providers. For example, the internet and the world wide web standards are both addressing horizontal market sectors. In contrast, standards for the medical radiology sector address a narrow vertical market sector, which, as in this case, can have a substantial market volume all the same.

Standardization is hard in horizontal market sectors. If a service is relevant to almost everyone, the length of the wish list tends to be excessive. Consider, as an example, the situation that standards committees for general-purpose programming languages have to face. At the same time, it is the horizontal market sectors in which a successful standard has the highest impact. The web is one of the best examples of this.

Surprisingly, standardization in vertical sectors is just as difficult as it is in horizontal ones, but for different reasons. The number of players is smaller, so the likelihood of finding a compromise should be higher. However, the vertical sector considered for a standard has to be wide enough for a viable market. Also, with a smaller number of players, the mechanisms of market economies work less well and it is less likely that good, cost-effective solutions are found within a short time.

4.2.2 Standard component worlds and normalization

Component approaches are most successful where the basic component world and the most important interface contracts are standardized and these standards are sufficiently supported by the relevant industry. However, for standardization to help, it is important to keep the number of competing standards low. With a single strong international standardization body, a single strong company, a strong coalition of companies, or other organization behind a standard, this can work. However, more likely than not, standards will compete. A particularly dramatic explosion of "mutually unaware" competitors can arise if vertical fragmentation leads to the reinvention of standards in allegedly different sectors when the same standard would suit multiple sectors. For

example, it is conceivable that several image-processing standards in, say, medical radiology and radio astronomy could be shared.

The risk of having large numbers of competing standards – and thus small markets for many of them – can be reduced by means of normalization. By publishing and cataloging "patterns" of common design, it is likely that otherwise mutually unaware standardization bodies will discover mutual similarities in their target domains. It is, of course, a matter of scale whether discovery and exploitation of such similarities are worthwhile – that is, cost-effective – or not.