Independently Extensible Component Frameworks

Wolfgang Weck

Institute of Scientific Computing, ETH Zürich, Weck@inf.ethz.ch

1 Introduction

The terms *component* and *component framework* are currently used differently, depending on context and products. Different understandings can be found for instance with Delphi, OpenDoc [5], and Oberon/F [6].

We believe, that the purpose of software components is to create a *component market*. The basic technology for this are *independently extensible software systems*. Independently extensible software in turn requires standards and guidelines for the extension creators. These guidelines depend on the application domain. We suggest to see component frameworks as software that enables and enforces obedience to such guidelines.

In this paper we review the requirements of a component market and show the role of component frameworks within it. We suggest definitions of the terms *component framework*, *dimensions of extension*, *parallel extensions*, and *orthogonal extensions*.

2 A Component Market Needs Independently Extensible Systems

An *extensible system* allows to add functionality at run time. Only the functionality currently used is loaded into the computer and may consume resources. Further functionality can be added when needed. The software to be added can even be retrieved via the network.

A system is called *independently extensible* if extensions can be developed by different people in complete ignorance of each other. Still, extensions are expected to cooperate where appropriate; at least, they must not to interfere with each other [7].

Independently extensible systems can change the way software is marketed. Instead of monoliths, designed to fit all, small *software components* can be offered. The customers select the functionality they want, buy the appropriate components, and compose their system from them. We call this market a it component market. In the future, such software components may even be billed per usage [2].

A component market would be much more lively than the current software market. The possibility to sell specialized software gives smaller manufacturers a chance. It is no more necessary to deliver a complete application package, which requires huge development resources. Instead, small ventures can specialize on high quality add-ons to existing software [1]. This will increase competition between vendors and finally generate the pressure for quality, which our current software market lacks.

Independent extensibility of software systems is a requirement of a component market. Only if components from different suppliers can be composed, the customer has a true choice.

3 Component Individualism

To assert correctness of an independently extensible system is a considerable problem. The main reason is that no static checks (like assertion of invariants or type assertions) can be performed on the complete system, since the system will never be complete. Static checks can only be done within individual components or based on component specifications and interfaces. All other checking must be deferred until run-time, leading to late detection of errors [7].

Further, independently extensible systems use a kind of composition that is different to reuse and specialization of objects and libraries. Extensions of independently extensible systems are developed individually and will be combined later with other components developed in parallel. The different developers do not know of each other. This is typical for independently extensible systems.

As a consequence, designers of components need to follow strict rules. Obedience to these rules is essential to guarantee the composed system's correctness. This shall be illustrated by an analogy: we will look at humans as independent individuals within a large population.

A human population consists of many individuals. None of the individuals knows all the others but still may be required to interact with any of them. In such cases, humans will judge the individual situation, negotiate, and compromise. This is not always possible, however: either because communication is restricted, or because there is not sufficient time for negotiation, or because the individuals are not willing to agree. In these cases some arbitration is needed.

Such arbitration is made effective and just by using predefined rules or laws. Since these laws are known in advance, the result of an arbitration process can often be foreseen and is reproducable. Therefore, most of the time individuals will compromise directly, saving the time for arbitration.

Maliciously behaving humans are locked up in a closed cell. Within that space they have all freedom, but it is hard for them to communicate with the outside or with the individual in the neighbouring cell. Isolating every individual is very safe, but it hinders cooperative work. For productive and effective cooperation, individuals need some freedom and possibilities for interaction. However, the freedom of one individual must end where another individual is impaired.

Of course, when going back to components of independently extensible systems, we discover that the analogy does not work in all aspects. Most importantly, software components are not able to negotiate their behaviour within the composed system. ([3] discusses how components could be adapted late. Such adaptations help to reuse components a programmer is aware of, which is not the case with independent extensibility.)

Software must be designed more rigorously. Programmers cannot change their product's behaviour after delivery. They have to get it right from the very beginning. When the user detects that a certain combination of components does not work correctly, it is too late.

On the other hand, when components cannot be trusted to behave well and to not affect other's correctness, they must be kept isolated. This is done by using separate address spaces for each component and strictly controlling resources allocated to components. However, this restricts interaction severely. Data to be exchanged, for instance, must be linearized for transmission and replicated in the receiver's memory.

We conclude that it is preferable if components behave well. Behaving well means to obey certain existing rules. These rules need to rule out every behaviour, that could lead to a conflict, and they must define protocols (or contracts) for interaction between components.

It is important to note that the rules are not a matter of distrust into the programmer's cooperation. In the contrary: the rules form the only guidelines a cooperative component programmer will have. Only clearly specified rules give the programmer freedom beyond entirely conservative programming. This freedom is founded on the limits imposed to other components, which will coexist at run time. The component design rules are the allowances and the limits at the same time.

4 Component Frameworks

The set of rules to be obeyed by components in a certain environment is what we call a *component framework*.

Component frameworks differ from object-oriented frameworks. Object-oriented programming uses the term *framework* for the implementation of a design pattern. The latter is a solution recipe, applicable to a family of problems. The premanufactured solution is adopted to a particular problem by specializing the framework. Emphasis is laid on the functionality offered for reuse.

A *component framework*, in contrast, is an incarnation of abstractions which define rules for extending components. These rules form a base on which multiple components can coexist in a single environment. Whereas object-oriented frameworks materialize a functional design pattern, component frameworks materialize design rules for components. Actual software solutions are contributed by the components. The framework itself does not necessarily need to implement any functionality. Still, it may manage shared resources and provide for communication between extension components.

In theory, a comprehensive documentation of the rules to be obeyed by component designers would be sufficient to form a component framework. Such a framework does not provide any safety. It is of no help in asserting system wide invariants.

A component framework's designer should strive for abstractions that enforce the necessary rules. The only tool available to implement these abstractions and to protect them against violation is information hiding behind the framework's interface.

By specifying the abstractions to be used by future components, much of the component's design must be anticipated. Component framework design is meta design. It presents a major challenge for today's software engineering.

5 Dimensions Of Extension

The above implies that a component framework can only be extended in those ways that have been planned for ahead, i.e. in certain *dimensions*. Also, the decision to use a certain component framework requires to accept the design rules and the abstractions defined by it. If such a framework contains the abstraction of some data type, this abstraction has to be taken as it is. It cannot be changed. In general, twisting existing software to become extensible towards a particular, not foreseen task is comparable to code reuse as being banned by [4].

It does not make sense to speak of *extensibility* as such. This situation is similar to a program's correctness. I makes no sense to state that a program is *correct* as such. Correctness can only be stated with respect to a given specification, and extensibility can only be stated with respect to certain extension dimensions.

Obviously, it cannot be the goal that every extension must be anticipated in detail, before creating a component framework. The challenge is to classify possible extensions. These classes we call *dimensions of extension*.

Consider compound documents as an example. A compound document framework will allow for extensions in two dimensions: containers and document parts. For each dimension the common aspects of all extensions are defined but not the details.

Note, that a single component may extend a component framework in more than one dimension simultaneously. Extensions of a compound document framework can implement parts which can be containers themselves. Dimensions of extensions were introduced to classify the extensions not to separate them completely. A reason for still using the term *dimension* can be found below.

6 Parallel And Orthogonal Extension

Two different types of independent extensibility can be distinguished. Extensions of the first type are mutually equal components expected to exist in parallel. We suggest to call this type of extension *parallel extension*, and the respective components *parallel components* (or *parallel extensions*).

From the component framework's point of view, parallel components are treated equally. They have to follow a common set of rules. Separating the individual components is the most important issue. Parallel extensions will access the same resources and perform similar operations.

A typical example for parallel extension can be found in frameworks for compound documents: document parts. Any number of document part components can be loaded. They are parallel extensions to the system. One important task of the compound document framework is to manage access to devices, like screen and keyboard.

The second type of extensibility is used to separate concerns. Different extensions will have different purposes. We suggest to address this kind of extensions as *orthogonal extensions*.

One purpose of such separation of concerns is to allow for unrestricted (pairwise) combination of components from different dimensions, i.e., to establish orthogonality. Often, of two orthogonal dimensions one will contain implementations of certain functionality, and the other will contain users of this functionality.

The main task of an orthogonal extensible component framework is to provide an interface for interaction of components from different dimensions. This interface must allow for efficient and effective interaction of the different components, but at the same time must also be general enough to not couple the components too tightly.

Document containers and document parts are an example for orthogonal extensions: any type of part can be put into any type of container.

The two types of extension can be combined. One framework can allow for extensions in several orthogonal dimensions, while supporting parallel extensions in some or all of these dimensions. On the other hand, a single component can extend a framework in several dimensions at the same time.

Component frameworks will be designed differently, depending on whether they support parallel or orthogonal extensions. For parallel extensions separation of the individual components is in the foreground. It has to be assured that independent extensions will not interfere. For orthogonal extensions, on the other hand, the definition of the interface used for communication between extensions of the different dimensions is most important.

7 Component Frameworks Are Domain Dependent And Nested

We have defined component frameworks as software, which implements the design rules for component designers. These rules depend on the domain of problems, for which the respective components shall implement solutions. Consequently, there will not be just one component framework in a particular system, but several.

Some component frameworks will rely on lower level abstractions, i.e. other component frameworks. Component frameworks will be nested and thus form hierarchies.

Usually, components that implement frameworks can be used by end users only together with extension components. On the other hand, top level components will not work without the underlying framework components they depend on. Therefore, the end user may need to buy several components to extend a system. It may be reasonable, to market such components as bundles. In Oberon/F [6], for instance, such bundles are called *subsystems* .

A component software environment forms the lowest level of a nested framework's hierarchy. It implements the rules for sharing hardware like memory and devices. Programming languages are important parts of this lowest level component frameworks. Through type systems, for instance, they implement essential rules, which protect the memory.

8 Conclusions

A system is called independently extensible if the user can compose it at run time from independently developed software components. Components are the units of extension. They are also units of encapsulation, thereby allowing for static assertion of certain properties.

Extension components will be developed by different people, in complete ignorance of each other. When the user composes these components, they must work together, or at least they must not interfere with each other.Individual components must be designed to allow for composition with other, unknown components. The only way to achieve this is to set up design rules for component developers in advance.

These design rules are specifications for future components. They will lead to certain abstractions. It is the purpose of component frameworks to implement these abstractions and to enforce obedience to the specification, as far as possible. This is particular important as far as global security is concerned. Like object-oriented frameworks are implementations of design patterns, component frameworks are implementations of the abstractions to be used by extending components.

Component frameworks can only be extended in the directions that have been anticipated and for which they have been prepared for. The design principles of the framework are inherited by any extension programmer.

The set of rules to be found and imposed depends on the domain of problems the components shall solve. Therefore, more than one component framework will be present in a running system. The frameworks will partly be nested within each other.

References

[1] R.Beech, *The Business Case for Component Software,* Apple Directions, pp. 19-23, Febr. 1996.

[2] B.Cox, *No Silver Bullet Reconsidered*, American Programmer Magazine, Nov 1995.

[3] Urs Hölzle, *Integrating Independently-Developed Components in Object-Oriented Languages*, Proceedings ECOOP'93, LNCS 707, Springer-Verlag, Germany, July 1993.

[4] B. Magnusson, *Code Reuse Considered Harmful*, Guest Editorial, Journal of Object-Oriented Programming, Vol. 4, No. 3, November 1991, p. 8, 1991.

[5] *The OpenDoc White Paper*, http://www.cilab.org/aboutod.html.

[6] *The Oberon/F User's Guide*, http://www.oberon.ch/customers/omi, Oberon microsystems, Inc., Basel, Switzerland, 1994.

[7] C.Szyperski, *Independently Extensible Systems - Software Engineering Potential and Challenges*, Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31 - February 2, 1996.