Independently Extensible Systems —Software Engineering Potential and Challenges—

Clemens Szyperski

School of Computing Science Queensland University of Technology Brisbane, Australia

c.szyperski@qut.edu.au

Abstract

Component-based software, open systems, and document-based user interfaces are about to revolutionise most areas traditionally addressed by the software engineer. We claim that many traditional software engineering methods, from life-cycle models to programming languages to system architectures are at least insufficient when facing the new trends. In this paper we present the main points of criticism and state a few unavoidable facts of life: extensible systems are in principle modular, have no final form or final integration phase, cannot be subjected to final total analysis, cannot be exhaustively tested, and have to allow for mutual independence of extension providers. We also hint at possible solutions for part of the problem set. In particular, we investigate the problem of dependence on global analysis, the effects of Cartesian Products in the design space, and the resulting design constraints on programming languages as the exemplary and most important tool of the software engineer. The relevance of the observations will be underlined by several examples drawn from contemporary programming languages and methods that got it wrong

Keywords Independent Extensibility, Cartesian Products, Global Analysis, Component-Orientation.

1 Introduction

A wave of new software technology is about to take over: component-based software executing in open systems. The promise is exciting: on the grounds of the very general document-based paradigm, users see the services provided as being directly represented as *document parts*. Document parts in turn are supported by software components, and there is no need for a user to even own components that support document parts that are never needed. On the other hand, since documents form the basis of information interchange, a document-based system has to expect that another user's document does contain parts that require support from additional components. These are components that either need to be retrieved and integrated into the running system, or that need to be compensated for in case of their unavailability.

Traditional systems catered for needs of extensibility by admitting the addition of new applications. Currently, we can observe a strong trend in the industry to move towards extensibility at a much finer level of granularity. Instead of adding applications, there is a desire to add small components that work together to replace the notion of isolated applications. The leading examples are Microsoft's OLE 2.0 [1] (Object Linking and Embedding) and Component Integration Laboratory's OpenDoc. The underlying component models are Microsoft's COM (Compound Object Model) and IBM's SOM (System Object Model).

It can be expected that this trend will continue to grow in strength and over time replace most of the traditional application technology. This will happen fastest on client computers running off-theshelf code with intensive use of graphical user interfaces. It will be followed by customised client software, and server-based software will be last. This is a natural evolution and correlates well with the policy found in many industrial organisations to consider client software to be far less long-lived or stable than server software (server data bases in particular).

The trend towards a software component industry will have many advantages. Component manufacturers can concentrate their efforts on their local strengths and users can expect a much richer software space. Component integration and configuration will spawn an entire industry of its own, replacing today's off-the-shelf monster packages by off-the-shelf standard configurations that will still be open to fine tuning by the more demanding user. In any case, it can be expected that times are over for "featurism"; there will no longer be a point in constructing a single component that

Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31-February 2, 1996.

can do it all. Reduction of the feature space to what is really needed by the individual user will reduce training costs and improve productivity. However, a good handle on the requirements for making systems truly extensible is required to fulfill the promise of component software.

The remainder of the paper is organised as follows. In the next section the essential ingredients of extensible systems are investigated. As it turns out, object-oriented programming is almost ideal to support extensible programming, but – as outlined in Section 3 – it is not sufficient, leading to the definition of the more specific notion of *extensibly object-oriented* in Section 4. The particularly important generic problems of global analysis and Cartesian products are introduced in Sections 4 and 5, respectively. Section 6 covers a wide range of established approaches that have problems in conjunction with extensibility. Section 7 presents two particular solutions and hints at open problems.

2 The Essential Ingredients

A system that allows for components to be plugged into the running system when needed is called *ex*tensible. This is not a technical definition and it is not likely that on this general ground such a definition could be provided. Nevertheless, let us try and challenge the definition: what does extensibility really mean? After all, even assembly programs are extensible; in a sense even better so than anything else! Also, extending an operating system by loading an application is quite an old achievement. Hence, on the one hand, if arbitrary code manipulations are allowed, arbitrary extensions are easy. On the other hand, if extensions are limited to a single level using a fixed interface (OS extended by applications) and a restricted model of extension interaction (applications operate on OSprovided files that are managed by the user), things are also easy, or at least well understood.

A more precise definition of extensibility has to take the interaction of mutually independent extensions into account. We call a system *independently* extensible, if it can cope with the late addition of extensions without requiring a global integrity check. (From now on, we only consider independent extensibility.) It is easy to see that this is a useful definition. Let us consider the case where a base system gets extended by two different vendors. Of course, a client expects that in most cases the two individually purchased extensions would go together. Naturally, some independent extensions may exclude each other, for example by providing alternative solutions to the same problem. Such a conflict and therefore the need to make a choice must be obvious to the client however.

The oldest extensible systems are operating systems. Loading a new application extends the func-

tionality of the overall system, and yet there is no need to check the combination of several concurrently loaded applications and the operating system itself. Of course, in the real world there are still many problems, mostly in terms of complex installation and configuration procedures. However, extending systems by adding new application to an operating system works well.

There are a few points that can be noted when looking at how an operating system achieves extensibility (of the overall system, not of itself):

- There are units of extension.
- The presence of one extension does not preclude or affect the availability of another extension.
- There is a polymorphic base.
- There is a late linking mechanism.
- There is a centralised and automatic management of resources.
- There is an abstract interface to operating system services.

In the case of (modern) operating systems extended by applications, all these points hold. Applications are the units of extension and there can be many applications in use at once. Operating systems provide at least a simple *polymorphic base* by means of untyped file systems and untyped process identifiers—all applications share this common base despite their varying nature. Late linking of extensions is performed by the loader, and the operating system uses mechanisms like cleanup on process termination to provide a centralised and automatic management of resources.

These points are quite general and we claim they form the minimal basis, ie the "essential ingredients", of *any* extensible system.

3 OOP gets close ...

The dream of a software component industry is old and so far has largely remained a dream. Object-oriented programming promises to be a foundation technology for a component industry. Nevertheless, pure object-oriented programming is not enough. (Recall a recent BYTE cover title ComponentWare—Object-oriented computing has failed. But component software [...] is succeeding. [17])

The claim that object-oriented computing has failed is overdoing the point. Nevertheless, a fundamental problem is the often mistaken emphasis on software reuse. The effective reuse of source code in a class hierarchy¹ can increase productivity when controlled carefully. However, source code reuse across small project groups is less productive and reuse across organisations can even be fatal. Current technology allows to describe interfaces, eg using an Interface Definition Language (IDL), but not the intricate self-recursion patterns² injected into code inheriting from another class. As a result, the inherited code itself is the only complete documentation.

With current technology, it is impossible to fully document classes that are used to inherit code from. This is admitted by class library vendors by routinely passing on to their clients the source of their libraries. However, if the class implementation is its only complete documentation, then the decoupling of clients and providers via a clearly defined interface has failed. In particular, evolution of class libraries easily leads to disaster. "Programming by contract", designed to lift interfaces beyond signatures by adding pre- and post-conditions, would help if it could fully cover the self-recursion patterns. However, with current technology this is not the case.

At this point it is useful to step back and have a second look at the idea of component software and compare it to well established component industries in other engineering disciplines. Code inheritance from class libraries is similar to copy and paste applied to blueprints. This is not the way how component industries work! While component providers do rely on reuse of blueprints internally, they rarely sell their designs. Instead they sell components. In other words, it is objects not classes that get sold. However, for components from different vendors to interact nicely, there needs to be standards. These standards state the general interfaces for the specific components to comply to. A standard is a type not a class.

Besides code inheritance, object-oriented programming introduces dynamic polymorphism (inclusion polymorphism, subtyping), i.e. the capability of a typed variable to reference objects of that type or a subtype thereof. This is one of the most important concepts—perhaps the most important one—found in object-oriented languages. (Note that languages like Smalltalk do not have explicit types. Nevertheless, Smalltalk is fully polymorphic: any variable can hold references to all possible objects.)

Traditional coding practice copes with variants by using explicit case analysis. Obviously, this is not extensible. Adding new variants later requires upgrading all points in the system that analyse cases. Dynamic polymorphism is simply required to solve this problem in a reasonable way. In a polymorphic program, the more specific can be substituted for the more general. In turn, extensions can be plugged in where only some general behaviour is expected.

Clearly, the notion of subtyping as manifest in todays object-oriented languages is too weak. The concept of *substitutability* is not supported explicitly—programmer's can be encouraged to limit subtyping to those cases that truly allow for substitution of the special for the general, but the required behavioural compatibility is usually not enforceable. Recent research tries to close this gap, eg Liskov and Wing [7].

Another aspect is the granularity of extensions. An extension almost always comprises more than a single object. Usually, an extension adds a subsystem. Proper language support for modules and subsystems can be very helpful to resolve configuration problems. One of the most important aspects is isolation or encapsulation of an entire subunit so that interference with other units from other vendors can be controlled statically, i.e. without inspecting the actual merger.

While most traditional languages fail to support dynamic polymorphism, many of the objectoriented languages fail to support static encapsulation of units that comprise of several classes and objects. Since classes are not the units of extension, the latter languages fail to provide any semantical guarantee for coherence of the actual units of extension that need to be gathered with extra-lingual means.

Finally, and most subtle, extensibility on the base of fine-grained components leads to the interaction between components on the level of references to individual objects. In an extensible system there is no way for an individual component to know when an object can be released again. There is no way around it: to be extensible, a system needs to support garbage collection and a language should not support explicit deallocation of objects belonging to a foreign component (or, even better, fully rely on garbage collection). (COM and to some extend SOM use reference counting under control of the programmer to get around thisthis approach is not safe; simple mistakes can lead to the known problems with dangling references or memory leaks.³)

¹Object-oriented programming either relies on code inheritance or on delegation to build new objects out of old ones that are "close" to what is required of the new ones. For this discussion it is irrelevant whether code inheritance or delegation is used.

 $^{^{2}}$ See Section 8.1 for a brief explanation of self-recursion.

³An additional problem of reference counting is that it cannot cope with cyclic references: Two components mutually referring to each other mutually keep their reference counts above zero, although no other component might still be referring to any of the two. Resolving cyclic reference conflicts is fully left to the programmer.

4 Extensible Object-Orientation & The Global Analysis Problem

The definition of "object-oriented" given by Wegner [18] does neither prevent nor enforce the construction of extensible systems⁴. Extensibility is a separate design dimension that needs to be considered carefully, leading to the term of *Extensible Object-Orientation* or EOO for short, Szyperski [15, 13].

The one added requirement for a system to be EOO should be obvious by now: the system needs to be extensible in multiple dimensions without extensions interfering with each other. However, this requirement is difficult to reduce to a hard technical definition.

Surprisingly, there is a simple necessary (but not sufficient) condition: to enable EOO, the design and implementation languages must be *sepa*rately compilable in principle. This does not preclude the use of global analysis or run-time compilation to improve performance. However, it does require a language⁵ to provide for units of separate compilation that can be used to check the system in increments. The key requirement is that a checked and unmodified unit shall under no circumstance be invalidated merely by adding another unit to the system.

An extensible system is never complete. Methods that require inspection of all parts⁶ of a system, ie global analysis, cannot be applied to independently extensible systems. Section 6 below presents a series of examples of such methods requiring global analysis.

In an extensible system, interfaces gain a dominant rôle. In traditional systems an interface hides the implementation and thus separates clients from the implementation of the provider. However, the interface and its implementation usually correspond one-to-one for a given system. In an extensible system, the interface may actually be implemented by a number of providers, and some of these implementations may only become available while the system is already running.

Thus, interfaces between units of extension in an extensible system are fully distinct from both, the interface provider and the interface client. Interfaces exist in their own right and cannot be fully reconstructed by inspecting any one particular provider.

In consequence, interfaces are a strong barrier for analysis aiming at system correctness. In princi-

ple, a system could perform certain global analyses at run-time once a binding of client and provider has been established. Relying on such *late global* analysis to verify correctness properties is equivalent to dynamic type checking—the detection of errors is likely to occur too late. Since the final integration of components in an extensible system is left to the customer, errors caused by assumptions based on global properties that cannot be statically enforced are likely to occur after product (ie component) delivery.

As noted above, a particular implementation strategy may still perform late global analysis to improve overall performance or resource utilisation.

All extensible systems have to face the design and implementation problem of limited scopes of analysis. A possible solution will be presented in Section 7.2. Before that, let us have a look at a second major problem.

5 The Cartesian Product Problem

Despite the general setting, there is a key problem that a designer of an extensible system must face: the avoidance of Cartesian products of individually extensible aspects of the system⁷. It is surprising that hunting for this single phenomenon turns out to catch many (not all) technical obstacles preventing independent extensibility.

For example, consider a text editor that can be extended by installing different text models, ie different implementations of the text abstraction. Let us assume that the text abstraction merely consists of the following operations:

- $new: \rightarrow \text{Text}$
- $append: Text \times Text \rightarrow Text$
- write: Text \times Character \rightarrow Text
- read: Text \times Position \rightarrow Character \cup EOT

The nullary operation *new* creates a new empty text. The *append* operation takes two texts and returns the first modified to have the second appended. The *write* operation writes a character to the end of a text. Finally, *read* returns the character in a text at a given position⁸, or a special *End-Of-Text* symbol.

Obviously, there is no problem when using only one implementation of texts. However, as soon as the system enables the installation of several text implementations to coexist, some problems occur.

⁴Wegner's definition essentially requires the notions of objects, classes of objects sharing a common implementation, and inheritance relations between classes (traditional code inheritance) or objects (delegation).

⁵In a multi-lingual environment, all languages need to be considered that cross individual components.

 $^{^{6}\}mbox{More}$ precisely: all parts falling into a certain category, such as all types.

⁷A Cartesian product of n sets is the set of n-tuples representing all possible combinations of the elements of the given sets. A Cartesian product can be depicted as a discrete n-dimensional space of points.

 $^{^{8}\,\}mathrm{We}$ assume some indexing scheme, the details are of no relevance.

The *new* operation needs to know which implementation to use when creating a new text. Also, *append* poses a tricky problem. How can a text fragment be extracted from one text and appended to another if the implementation of the two may vary, and in fact, if the list of possible text implementations is open?

The *new* problem can be solved by providing a configuration mechanism that allows clients to indirectly call one of the implementation-specific *new* operations. A typical approach is the use of *factory* or *directory objects*, Szyperski [15].

The *append* problem remains. Indeed, the two separately extensible argument types of append span a Cartesian product. As depicted in Figure 1, a complete implementation of append would need to consider the entire matrix of pairs of two text objects, each from a possibly separate implementation. It is reasonable to assume that all extensions know their base type and that all extensions support a homogeneous operation—these are the cases marked "x" in the figure. However, it is not reasonable that an extension knows about all other extensions-the corresponding heterogeneous non-base cases are marked "?" in the figure. Generally, all pairs $(T_i, T_j), i > 0, j > 0, i \neq j$ fall into the latter category.



Figure 1: Cartesian product created by independent extensions. The figure shows the combinations of types that need to be considered when implementing a binary operation over objects of extensible type. The $T_i(i > 0)$ are independent extensions of a common base type T_0 . Note that any number of such extensions may exist. Then, the Cartesian Product Problem is that no one can be expected to provide an implementation for a binary operation that specifically caters for all heterogeneous nonbase cases, marked "?" in the figure.

For a closed system⁹, there are a number of ways to approach Cartesian products. The simplest is an explicitly programmed nested case analysis. To avoid some of the maintenance problems, one could use more advanced approaches such as operator overloading as in Ada [10] or multiple dispatch methods ("generic functions") as in CLOS, DeMichiel and Gabriel [6]. However, all these approaches rely on someone inspecting the entire space of combinations, ie the Cartesian product, and deciding how to deal with each individual case. In the case of an extensible system with an unbounded number of potential alternatives, this is not feasible. Along similar lines, many other traditional approaches break down when considering extensible systems (cf Section 6 below).

In an extensible system it is unavoidable to face the problem: bridging the gap of incompatibility between two independent and mutually unaware extensions. In the next section we will examine a series of approaches that all are in danger of colliding with either the Global Analysis Problem or the Cartesian Product Problem. (Note that the Cartesian Product Problem can be seen as a special case of the Global Analysis Problem. However, since Cartesian products are such a common phenomenon, a separate treatment seems appropriate.)

6 Endangered Species

With the goal of extensibility and the pitfall of Cartesian products in mind, let us now examine a wide range of established approaches from different areas of software engineering.

Simple Life-Cycle Models. The idea of starting with a specification and ending with a product has to be rethought. An extensible system can never be fully specified—the challenge is quite the opposite: to leave open as much as possible. The known problems with the design of *frameworks* indicate that traditional engineering models fail: many tedious revisions are required to extract a useful common ground, a framework, out of a more or less unrelated (and necessarily incomplete!) set of cases. The currently emerging design a little, code a little, test a little approach follows the same direction. However, it has the major drawback of not emphasising design in the large. Also, as mentioned before, the life-cycle phases of integration and testing need to be re-thought, as final integration now takes place after delivery-too late to allow for testing and therefore excluding traditional integration testing beyond individual components.

Type Systems. In general, one of the most valuable tools of the software engineer are well-crafted type systems and as such they are *not* "endangered species". Type Systems can be used to capture important parts of the specification in a way that enables automatic checking. However, several of the more *advanced* approaches in type systems fail

⁹Some authors, such as Cardelli [2], distinguish closed and sealed systems, where a closed system can still be extended but only by explicit declaration, while a sealed system cannot be extended anymore. Within this paper this distinction is not made and closed systems are already considered non-extensible.

to support extensible systems. Some of the problematic approaches are detailed below. Work on existential quantification to capture abstract types is promising, though (cf Section 7.1).

Non-local Type Inferencing. For a closed (nonextensible) program it is possible and under certain conditions even practically useful to perform a global analysis to reconstruct type information, or to reject a program if such a reconstruction fails. In an extensible system type inferencing has to stop at extension interfaces: these necessarily exist before their implementations and the full set of different implementations of the same interface will never be available. Hence, relying on type inferencing to establish aspects of global correctness for a particular composition of components does not work well for extensible systems. In particular, it needs to be applied at system integration time, ie at run-time, and therefore would report errors to the end-user rather than the developer.

Covariant Subtyping and System-level Type Checking. A particular approach called "systemlevel type checking", following along the lines of system-wide type inferencing, has been used to "fix" the language Eiffel, Meyer [8]. In the original specification, the language could express type-unsafe programs¹⁰, Cook [5]. The problem is quite simple to explain: assume an operation like the above defined *append*. Eiffel allows such an operation to be expressed as a method of one text, ie the first Text-typed parameter becomes the receiver of the method. Additionally, the second Text-typed parameter can be an argument of the method and be declared to have type LIKE CURRENT. This is called covariant subtyping: the receiver and one of its method arguments are co-subtyped.

To further investigate the problem of covariant subtyping, consider a class B that forms a subtype of a class A. If an argument of a method in Bthat is also present in A can be forced to be of a subtype of that defined in A, then this argument is covariantly subtyped. This is type-unsafe. To see why, consider a client that knows nothing about B, but relies on the definition of A. Such a client could call an object of class B via the interface defined for A, since B is a subtype of A. If the client passes an argument of the type specified for A's method argument, rather than of the subtype required in B, then the code in B's method is likely to break. There are two known ways around this problem: (a) forbid covariant (or any) subtyping of arguments—this is the case in most OO languages;

(b) apply system-level type checking. The idea of system-level type checking is to use a conservative check at the final system integration time to make sure that, although type-unsafe, the actual program can never reach a state in which the problem pops up. As should be obvious by now, system-level type checking is *incompatible with extensibility*: it is based on a global analysis of the final system.

Incremental Program Validation. It has been proposed to use incremental program validation to deal with extensions, Wills [19]. This is obviously the right way to go. Unfortunately, the same proposal presents a way of "vertical extensions" (called *capsules*) where an extension is allowed to modify every part of the extended system. This neatly captures the standard approach of Smalltalk development, but ignores the problem of recombining two separately extended systems.

Multi-Methods. The Cartesian product problem is actually quite common. Probably the oldest example are arithmetic expressions over a wide range of algebraic structures with various defined homomorphisms. Since the times of Algol this has been dealt with by using various forms of operator overloading: multiplication of two integers is expressed using the same operator symbol as is used for multiplication of two reals, an integer and a real, etc.

In languages that treat all abstractions as classes and therefore all entities as objects, this becomes rather more involved. DeMichiel and Gabriel [6] introduced multi-methods to address this problem. In Cecil, Chambers [3] applies multi-methods to object-oriented programming. A multi-method gets dispatched on multiple arguments: the actual type of more than one argument is inspected to select the method implementation that is supposed to handle the specific argument combination at hand. Of course, this is just elegant sugar for an explicit coding of a Cartesian product. Hence, parts of systems that rely on multi-methods are *not extensible*.

7 Some Solutions

7.1 Existentially Quantified Types

The notion of extensibility is strongly related to the ideas of abstract types and abstract subtypes. Whenever an interface requires extensions to handle two abstract types (with possible abstract subtypes) at the same time, the Cartesian Product problem occurs.

As described by Mitchell and Plotkin [9], abstract types can be modelled accurately using existential quantification. An interesting consequence of this approach is that interfaces requiring operations based on two or more abstract types form an abstract type that itself *cannot be implemented*.

¹⁰A construction is type-unsafe when type checking establishes an assertion that a certain variable is of certain type and this assertion turns out to be wrong for certain execution runs. When compiler and run-time essentially rely on such a type assertion, its failure can lead to severe and arbitrary program malfunctioning.

For a good explanation in relation to programming languages, see Cardelli [2].

While not solving the problems of Cartesian Products in interfaces, the use of existentially quantified types helps to prevent in the first place the design of interfaces that would be subject to the problem.

7.2 Units of Analysis

Modules. The simplest approach to attack the global analysis problem is the use of a static subdivision of programs into *atomic units of analysis and extension*, ie units providing a *confined context* of analysis for the enclosed program parts.

A particularly useful language feature to do so—although by no means new—are modules. Unfortunately, modules got mixed up with classes and as a result are missing in many current objectoriented languages, cf Szyperski [14]. A module is a natural unit of separate compilation (in the sense of static checking), as it can—and should—contain all parts of a system that are interrelated in a way that prevents separate checking of these parts. Modules can and should form the *minimal units of extension* in a system. Of course, it is quite likely that modules again are clustered into higher-order building blocks, such as subsystems. Still, they remain the atoms of replacement and extension. (As an aside, modules and possibly subsystems also help to tidy up name spaces, an important aspect when considering extensibility.)

Modula-2's modules (Wirth [20]) or Ada's packages can well serve this purpose. Oberon (Reiser and Wirth [12]) restricts the module system to a simple flat module space. This is clearly along the lines of providing atoms instead of nested structures with partially static interconnections, as was the case with nested modules in Modula-2 or nested packages in Ada. Ada95 adds module hierarchies in the form of parent and child packages, where a child package can be public or private to its parent package, without being textually nested.

A simple design guide line is to put all those things together into one module that need coupling beyond mutually known interfaces. Such strong coupling usually takes the form of partial implementations encapsulated by a module, embodying and enforcing semantics beyond what could possibly be stated by a typed interface. This approach has been followed successfully to construct the extensible object-oriented operating system Ethos, Szyperski [15], and the commercial framework Oberon/F [11]. Clearly, more work is needed to provide more detailed guide lines and to fully explore the potential of module constructs for building extensible systems.

Resolving Modules. The initial design of Cecil made the design of extensible systems quite impossible unless multi-methods were, by convention, not used or restricted to single dispatch, ie traditional OO methods. This problem has been acknowledged in the meantime and a recent enhancement to Cecil introduced the notion of resolving modules, Chambers and Leavens [4]. A resolving module is essentially the top of a group of code units that together provide the definition of a multi-method. No unit outside the *static scope* of the resolving module can contribute further to the multi-methods combinatorial space—the Cartesian product is closed off and the number of combinations gets bounded. Therefore, a Cecil system can statically verify that for each multi-method (a) a resolving module exists and b) all combinations of the Cartesian product are handled by the combined definition of the respective multi-method.

Closed Subsystems. Adding a resolving module to a subsystem closes off that subsystem with respect to a multi-method. In other words, that subsystem is not extensible with respect to the closed multi-method, while other parts of the system that do not rely on multi-methods can remain open and extensible. In particular, the later addition of a new subsystem with its own new and fully enclosed multi-methods remains possible.

7.3 Bottleneck Interfaces

The Cartesian Product Problem essentially states that there cannot be a reasonable provider for specific operations over combinations of independent extensions. Sometimes such "adaptors" are unavoidable: typical examples are separately emerging coexisting "industry standards" that force many clients to use special cross-standard adaptors. For a strictly limited and small number of coexisting standards this works and creates a new industry selling adaptors. If the number of alternatives is too large or alternatives are too far apart, the approach breaks down and the common result are technology islands.

For extensible systems, the Cartesian Product Problem is omnipresent and needs to be dealt with explicitly. A standard approach to do so is the construction of a *bottleneck interface*, Szyperski [15]. Instead of trying to handle each and every combination individually, all cases are reduced to two separate mappings provided by the extensions themselves.

Following the Text example above (Section 5), the two mappings, called down and up, would take the form—

- down: $Text \rightarrow T_0$
- $up: T_0 \to Text$

Text stands for any text type that is identical to or a subtype of T_0 , and T_0 therefore is the base type of all texts.

Figure 2 illustrates the use of the common bottleneck interface mapping to operate across otherwise fully independent extensions, in the example by appending texts of independently extended type. Operation *append* takes two arbitrary texts conforming to the minimal interface for texts (type T_0) and returns the first text modified by appending the second one. In the figure, texts t_1 and t_2 are assumed to be of mutually incompatible types T_1 and T_2 , respectively¹¹. Both, T_1 and T_2 are subtypes of T_0 , though. Hence the bottleneck mappings down: $T_2 \rightarrow T_0$ and up: $T_0 \rightarrow T_1$ can be applied to reduce the problem to a homogeneous append: $T_1 \times T_1 \to T_1$. (As an aside: the required covariant typing of the return value of append is safe.) In the figure, the intermediate text resulting from mapping t_2 to type T_0 is $t_2 \downarrow T_0$; likewise that of mapping up to T_1 is denoted $(t_2 \downarrow T_0) \uparrow T_1$.

$$\begin{array}{c|c} \begin{array}{c} \begin{array}{c} \text{append: } T_1 \times T_2 \twoheadrightarrow T_1 ? \\ \hline t_1 : T_1 \times t_2 : T_2 & & \\ \end{array} \\ \hline \end{array} \\ \begin{array}{c} \begin{array}{c} \text{down: } T_2 \twoheadrightarrow T_0 \\ \downarrow \\ t_2 \checkmark T_0 \\ \downarrow \\ t_2 \checkmark T_0 \\ \downarrow \\ \end{array} \\ \begin{array}{c} \text{up: } T_0 \twoheadrightarrow T_1 \\ \downarrow \\ \end{array} \\ \hline \end{array} \\ \begin{array}{c} \text{append: } T_1 \times T_1 \twoheadrightarrow T_1 \end{array} \end{array}$$

Figure 2: Using commonly accepted down and up mappings to interconnect arbitrary independent extensions. The figure uses the example of operation append on texts of independently extended type by mapping one of the texts first to a common base type (down) and then to the type of the other text (up). As a result, a homogeneous append operation dealing with only one type of text can be used.

Recall the definitions for an abstract text presented in Section 2 above: down is simply a repeated application of *read* over increasing indices and up is simply a repeated application of *write*. Hence, in this example, there is no need to add any new operations.

With down and up available, it is possible to implement append fully generically, that is without knowing any of the possible implementations of texts. Of course, a provider of a new text implementation may take advantage of known other implementations by specialising append in these cases. Note that there is no need to use multi-methods for this: the approach is necessarily intertwined with a single fully accessible implementation, that is the new extension. Hence a traditional method interface for *append* would do and the implementation of *append* can explicitly check for the (hopefully few) special cases.

A generic use of the bottleneck interface cannot cope with refined attributes added by extensions that have not been thought of when designing the bottleneck interface. In the text example this might be font attributes attached to runs of characters. The generic *append* would drop such attributes and if the target text had such attributes it would have to provide some default setting. In general, this problem is unavoidable.

In the special case of cooperating with a known other extension, specialisation can again be used to improve the situation. As outlined above, a specialised *append* would at least preserve font attributes when source and target text are of the same type, but possibly also when the other text is known to support font attributes *and* it is known how these attributes can be converted into the local representation.

Finally, it should be noted that bottleneck interfaces can also be used when extensions are formed recursively, that is when extensions themselves form the basis for further extensions. Care must be taken not to confuse the bottleneck interfaces corresponding to the different levels of extension—chosing the wrong *down* operation will lead to a too primitive base type and thus too much information may be destroyed.

8 Open Problems and Future Work

8.1 Unrestricted Code Inheritance

A problem that is wide open with current OO technology is the proper handling of self-recursions defined in classes that other code inherits from. Self-recursion is fundamental to object-oriented programming and refers to the invocation of further methods of the current object ("self") as part of the implementation of one of its methods. When inheriting code from another class such calls to own methods are inherited as well. The code resulting from the combination of inherited code, overridden methods, and super calls tends to rely on the self-recursion patterns, ie the order in which self-recursive calls to methods of an object occur. Self-recursion can be caused by self-references in a class, in an object, or both. As such it is an essential part of the object-oriented programming paradigm. However, it becomes problematic when self-recursion crosses boundaries of classes or, in the case of delegation, boundaries of objects in a parent/child relation.

If the inheriting code knows no more than the interface definition of the base class, eg by relying on the information specified using an Interface Def-

¹¹ As usual with updating operations, the objects passed to the operation are primed (eg t'_1) to distinguish them from the modified objects returned by the operation.

inition Language (IDL), then the inheriting code must not rely on any self-recursion injected by the base class—otherwise the base class could no longer be evolved or replaced by an alternative implementation without risking to break client code. However, since object-oriented programming is statebased, side effects in an object due to different orders of self-recursive calls are observable. Therefore, the programmer of a class can actually observe the implementation of a class beyond its interface and in fact is likely—after some extensive debugging to get things "right"—to actually depend on it.

However, current technology (and current IDLs) do not allow for mechanical enforcement of a client not relying on its base class implementation. Strictly avoiding code inheritance is one possible approach, followed for example in COM. Another approach is admitting the strong coupling caused by code inheritance and therefore carefully separating the forming of subtypes and the use of code inheritance. In Sather (Szyperski, Omohundro and Murer [16]) this has been taken to the point where subtyping can only be based on fully abstract classes (no implementation at all) and where inheritance is indeed defined to be equivalent to textual inclusion of source code. This admits that code inheritance is a low-level concept and essentially reduces it to compiler-performed editing operations.

To summarise, if code inheritance is used to implement extensions for a given base system, the resulting coupling of extension and base goes far beyond using the base system's interface. The unfortunate state of the art thus is to deliver class libraries in source form, basically admitting that the documentation and abstract interfaces are not sufficient.

8.2 Granularity of Extensions

It should be noted that while the requirements of independently extensible systems leave many proven methods behind, these methods still have their place, even within the area of extensible systems. An extensible system will always be composed of smallest components—atoms—that themselves are not extensible and hence can be developed using traditional approaches.

These atoms are *closed subsystems* of an extensible system. The usefulness of traditional methods for development of such atoms depends directly on the size and complexity of the atoms. The size of the atoms in turn affects the degree of extensibility of the overall system. In the absense of a guiding theory the choice of the right atom size remains a difficult engineering problem.

8.3 Software Engineering Process

The software engineering process of extensible systems must be considered an unsolved problem. It would seem that methods and processes for development of object-oriented software would do better. However, they suffer from the same fundamental problem: it is assumed that analysis and design can succeed in enumerating the requirements and working out the details. With extensible systems this is questionable.

It has been acknowledged that frameworks cannot be developed using a standard process and that instead a spiral model is required that takes feedback from actual use of the framework into account. Object-oriented frameworks are an important technology for building extensible systems, but the requirement of independent extensibility adds further problems and increases the burden on the engineering process.

In a market of software components it would seem that traditional processes are at least able to support the development of individual components since those are integrated before delivery. It is unknown to what extend this is true; for this approach of "understood engineering in the small" to work, the requirement that a component has to extend its base system in a proper way needs to be formalised.

9 Conclusions

The emergence of a true software component industry is promising indeed. However, the unavoidable mine field of independent extensibility has to be tackled. An informal treatment of the principles and challenges of extensible systems is presented in this paper. Based on the inspection of many established software engineering methods and techniques, down to actual programming languages, it can be concluded that major revisions of our approaches and tools of our trade are required. However, at the same time it should be noted that the established methods have their validity when constructing individual extensions that themselves are not supposed to be extensible.

The main contribution of this paper is to actually point at an emerging and important problem, showing that there is no easy way out, and that many traditionally approaches need to be rethought. The proposed solutions are a mere beginning, much remains to be done. A formal theory of independently extensible systems, their requirements, and their precise potential is still missing.

Acknowledgments

The anonymous reviewers provided many helpful hints that helped to improve this paper. In particular, one of the reviewers pointed at the relevance of existentially quantified types for interface typing in exstensible systems. Another reviewer hinted at Ada95's child package construct.

Wolfgang Weck carefully read an earlier version of the paper and suggested many improvements. Following his suggestion, the earlier title "Extensible Systems ..." has been changed to "*Independently* Extensible Systems ...", clarifying the focus of this paper.

References

- K. Brockschmidt. Inside OLE 2.0. Microsoft Press, 1994.
- [2] Luca Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center, Palo Alto, CA, May 1989.
- [3] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands, Volume LNCS 615 of Lecture Notes in Computing Science, pages 33-56. Springer Verlag, June 1992.
- [4] Craig Chambers and Gerry T. Leavens. Typechecking and Modules for Multi-Methods. In Proceedings of the Ninth Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94), pages 1-15, October 1994.
- [5] William R. Cook. A proposal for making eiffel type safe. In Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP'89), Nottingham, England, pages 57-70. Cambridge University Press, July 1989.
- [6] L. G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In Proceedings of the First European Conference on Object-Oriented Programming (ECOOP'87), Volume LNCS 276 of Lecture Notes in Computing Science. Springer Verlag, June 1987.
- [7] Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, Volume 16, Number 6, November 1994.
- [8] Bertrand Meyer. Eiffel The Language. Prentice Hall, 2 edition, 1992.
- [9] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In Proceedings, 12th Symposium on Principles of Programming Languages (POPL'85), 1985.

- [10] Department of Defence. Reference Manual for the Ada Programming Language. United States DOD, Washington D.C., November 1980.
- [11] Cuno Pfister and Clemens Szyperski. The Oberon/F Framework – Tutorial and Reference. Oberon microsystems Inc, Basel, Switzerland, 1994.
- [12] Martin Reiser and Niklaus Wirth. Programming in Oberon - Steps beyond Pascal and Modula. Addison-Wesley, 1992.
- [13] Clemens Szyperski. Extensible objectorientation. In Proceedings, 1st Workshop on Object-Oriented Programming Languages – The Next Generation (at OOPSLA'92), Vancouver, Canada, October 1992.
- [14] Clemens Szyperski. Import is not Inheritance. why we need both: Modules and Classes. In Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP'92), Utrecht, The Netherlands, Volume LNCS 615 of Lecture Notes in Computing Science, pages 19-32. Springer Verlag, June 1992.
- [15] Clemens Szyperski. Insight ETHOS: On Object-Orientation in Operating Systems, Volume 40 of Informatik-Dissertationen ETH Zürich. Verlag der Fachvereine, Zurich, Switzerland, 1992.
- [16] Clemens Szyperski, Stephen Omohundro and Stephan Murer. Engineering a Programming Language – the Type and Class System of Sather. In Proceedings, First Intl Conf on Programming Languages and System Architectures, number 782 in Springer LNCS, Zurich, Switzerland, March 1994.
- [17] Jon Udell. Componentware. BYTE, Volume 19, Number 5, pages 46-56, May 1994.
- [18] Peter Wegner. Dimensions of object-based language design. In Proceedings of the Second Conference on Object-Oriented Programming Systems, Languages, and Applications (OOP-SLA'87), pages 168-182, October 1987.
- [19] Alan Wills. Capsules and types in fresco: Program verification in smalltalk. In Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP'91), Geneva, Switzerland, Volume LNCS 512 of Lecture Notes in Computing Science, pages 59-76. Springer Verlag, June 1991.
- [20] Niklaus Wirth. Programming in Modula-2. Texts and Monographs in Computer Science. Springer Verlag, 4 edition, 1988.