Evolutionary software engineering: a component-based approach

P. Klint
Centrum voor Wiskunde en Informatica, and
University of Amsterdam
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands,
e-mail: Paul.Klint@cwi.nl

C. Verhoef
University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands,
e-mail: x@wins.uva.nl

Abstract
A software industry producing high-quality components that can be reused in many ways is an—unfulfilled—dream as old as the field of "software engineering" itself. In this paper we present an evolutionary approach to software development based on the following premises:

- software systems are unavoidably heterogeneous and distributed;
- development and implementation techniques may be different for various parts of a system;
- the parts may be in different phases of their life-cycle;
- the parts—implemented as components in different languages—should be coordinated and exchange information in a standardized fashion;
- reengineering and system renovation form an integral part of software development.

We illustrate this approach in various case studies and indicate some lines for further research.

Keywords
Evolutionary development, software components, coordination architectures, heterogeneous systems, integrating forward engineering and system renovation, case studies

1 SETTING THE STAGE

1.1 History

In the late 1960s, the NATO Science Committee organized a conference that is generally considered as the beginning of the field of software engineering. Quoting from the introduction of the proceedings (Naur & Randell 1969) we read:

> The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

In the same proceedings, McIlroy (1969) states that "the software industry is not industrialized" and gives his vision on mass produced software components. In the following decades, a large variety of approaches and techniques have been proposed to build software: the life-cycle and waterfall models in many flavors, formal specification, rapid prototyping, fourth generation languages, code generators, computer-aided software engineering, object-orientation, and much, much more.

Although all these approaches can claim success in certain areas, it is astonishing to see that they only have modest success in the day-to-day reality of, for instance, shrink-wrap software production (McConnell 1993) where "good enough" software is the standard (Yourdon 1996). For most software products time-to-market is the prevailing concern and manufacturers maintain defect levels that are both comparable to those of their competitors and that are acceptable for their customers. McIlroy's vision of a software industry producing high-quality components in many variations is still far from being a reality.

All these approaches are based on the assumption that the software production process can be controlled in a global, top-down, fashion. This is not only the theory taught to students and the fiction maintained by managers, but also the myth exposed by real programmers. This situation is not unique for software development: Kelly (1994) gives ample evidence that we are "out of control" in many areas of complex technological development and that evolutionary principles are becoming more important.

1.2 Techniques used by Real Programmers

(a) Information hiding

Information hiding is probably the single most important principle guiding software design and construction. Goal is to achieve two forms of flexibility.

Flexibility of *implementation* is achieved by hiding the low-level implementation aspects of a program part—the specific data representations or algorithms used—thus concentrating and hiding implementation knowledge; new implementations can be introduced without affecting the use of the program part. Flexibility of *use* is achieved by encapsulating the program part in such a way that only its provided functionality is visible from the outside. Information hiding is the fundament of, for instance, abstract data types, object-oriented design, and object-oriented languages. The major challenge for information hiding is how to accommodate changes in the visible interface of program parts over their life time.

(b) Reuse

Reuse is a related, important notion for programmers (Krueger 1992): by first investing in encapsulated program parts that are of sufficient quality, one can later on reuse these parts to build new systems. The result is higher productivity of programmers and higher quality of the resulting code (Lim 1994). As already mentioned, McIlroy (1969) was one of the first to create the vision of reuse and component-based manufacturing of software. Four reuse techniques are in wide use: subroutine libraries, application generators, Unix pipelines and code scavenging.

Subroutine libraries are the oldest and the most successful reuse technique and have been applied to package reusable code in areas as disparate as operating system services, input/output routines, mathematical functions, graphics, databases, multi-media, and more. Each library provides a language-specific (e.g., Fortran, C, COBOL) interface, but the same library may be accessible from different languages. A typical example is a library of mathematical functions that can be called from both Fortran and C. The data exchanged between the subroutines in the library and the program using the library are restricted to data types of the host language. However, when subroutine libraries mature, their size tends to grow geometrically. This is due to all the feature variations that have to be provided regarding, for instance, precision, robustness, algorithm used, efficiency, and memory usage. This calls for sophisticated search and retrieval methods such as, for instance, described by Prieto-Diaz & Freeman (1987).

Application generators provide an abstraction mechanism to give access to an underlying subroutine library. Programming language compilers and application generators have much in common. The former compiles a program into machine code that may call routines from a run-time system to perform common tasks such as, for instance, stack and memory management, run-time checking, and input/output. The latter takes a concise application description as input and generates executable code, containing calls to the underlying subroutine library. Application descriptions are typically very high-level and designed for specialized, narrow, application domains. Application generators extend ordinary subroutine libraries by automating common usage patterns of the library and hiding implicit dependencies between routines. Application generators have been applied successfully in areas like compiler construction, user-interfaces, and databases (Horowitz, Kemper & Narasimhan 1985, Cleaveland 1988) and they have a close relationship with Domain Specific Languages (see Section 3.2) and *application frameworks* (Fayad & Schmidt 1997).

Unix pipelines (Ritchie & Thompson 1974) provide a global reuse mechanism: the input and output of individual Unix programs can be connected together to form a pipeline of programs that carry out a certain task. Typical pipelines perform sequences of operations for searching, replacing, transforming, and formatting text streams. Each program in the pipeline may be written in a different language, the data exchanged between programs has the form of a list of strings, and programs can only be combined in a strictly sequential fashion. As a prelude to discussions later (in Section 2.3), we observe that the pipe mechanism has two fundamental properties: (*i*) a standardized format to exchange data between programs; and (*ii*) a sequential composition operator to connect programs.

Code scavenging is a frequently used, but largely under-documented technique for reuse: when a programmer needs to implement a certain functionality she searches through the sources of existing programs and looks for code that provides functionality that is comparable to the one that is desired. When such code is found, she reuses it after appropriate editing and modification rather than writing the code from scratch. At least three aspects of code scavenging are remarkable: (*i*) it is the inverse of the instantiation of parameterized data types; (*ii*) the common origin of the original and the modified code are immediately lost, but might be recovered using reverse engineering techniques; (*iii*) although, the technique is frequently used in practice, there is hardly any support for it let alone any supporting theory. We will discuss this topic further in Section 4.3.

(c) Tools

In common use for program construction are tools for compilation, editing, debugging, configuration and version management, execution profiling, coverage measurement and other testing tools. In some specialized areas (compiler construction, databases, user-interfaces) program generators are being applied successfully. For researchers, it is rather disappointing to see that the high-tech tools and techniques that result from research projects only have a very limited impact in practice (Hoare 1996). A single version management tool such as, for instance, CVS (Cederqvist 1993) has probably done more for software quality than all existing verification techniques combined.

1.3 Plan of this paper

The contribution of this paper are speculations on emerging software engineering approaches that promote evolution and reuse. These speculations are, however, based on extensive experience in the area of building software tools, i.e., tools that analyze, transform, or generate software.

The plan for the paper is as follows. In Section 2, we sketch a new approach to building software that we are currently applying in various projects. In Section 3, we describe case studies that illustrate our approach. In Section 4, we draw some general conclusions for the field of software engineering as a whole and point at several research directions.

Managerial and economic aspects of reuse—and of software engineering in general—are of utmost importance (Lim 1994) but we will largely ignore them. The only exception is Section 4.4 where we discuss usage-based pricing methods for software in relation to the evolutionary software engineering that we propose in this paper. Basili, Caldiera & Cantone (1992) give an organizational view on component factories that can be used as complement to the technical view presented by us.

2 AN EVOLUTIONARY APPROACH

2.1 Preliminary observations

In the theory and practice of software engineering, most attention is given to *forward* engineering: building new systems. In reality, only 30% of the total costs of a system are devoted to its initial construction: the remaining 70% are spent on maintenance and adjustments to new requirements and new operating environments. These costs are confirmed in many studies (Lientz & Swanson 1980, Reutter 1981); McConnell (1996) gives a recent summary of these findings. For instance, based on an extensive case study Reutter (1981) found that 70% of the programming costs occur after installation of the initial product and that only 8% can be attributed to emergency repair and corrective coding. Then 25% of the costs are due to environment changes and 67% can be attributed to upgrades and enhancements of the system. The latter costs include upgrades, changes due to new business conditions, and modifications and/or addition of new subsystems. The remaining costs are directly related to a system's evolution: adjusting to new infrastructure and building new versions and adding new functionality. The conventional models like the waterfall model do not cover these evolutionary aspects very well.

Based on the observations made so-far, we can draw some preliminary conclusions:

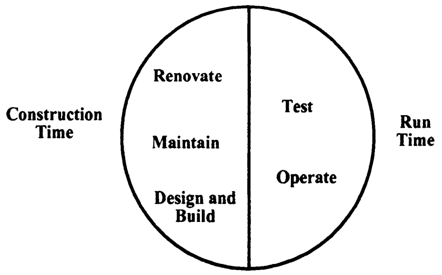

Figure 1 Construction-time versus run-time.

- We need software engineering practices that smoothly integrate forward engineering, maintenance, reverse engineering and system renovation.
- Software development as a whole seems to be too complex to be controlled in a top-down fashion. Evolutionary mechanisms seem more appropriate.
- It is not easy to indicate in which evolutionary phase a complete system is: typically its parts are in different phases of their evolution.
- Reuse is a key to more effective, higher quality, software development. In particular, reuse across different implementation languages should be supported.

The above insights have been stated, maybe in a different wording or a different emphasis, by other authors like, for instance, Yourdon (1993, p. 262). They remain, however, of interest and form the point of departure for the remainder of this paper.

2.2 Construction-time versus run-time

It is worthwhile to make a distinction between the moments that one is building software and the moments one is executing the resulting programs, see Figure 1. We will call the former *construction-time* which includes the phases: (*i*) Design and Build: Requirements Analysis, Design, Implementation (coding); (*ii*) Maintain; (*iii*) Renovate: Reengineering and System Renovation (Chikofsky & Cross 1990, van den Brand, Klint & Verhoef 1997). The moments one is executing programs will be called *run-time* and includes the phases: (*i*) Test: Unit Testing, System Testing; (*ii*) Operate. One could make a further distinction between test-time and exploitation-time, but for the purposes of this paper we will call them both run-time.

Development of a system can be seen as the first time a system is in the construction-time environment; after delivery it enters the run-time environment. During maintenance or renovation the system enters the construction-time environment again. During its life time, a software system will thus alternate between construction-time and run-time. Two aspects of this approach are worth emphasizing. First, the phase Renovate is usually not included in the life-cycle. We stress its importance by including it as a phase in our evolutionary life-cycle model. Second, only the life-cycle of a complete system is usually taken into consideration. In Section 2.3 we will present an approach that supports the *component life-cycle*: each part of a system may be in a different phase of its evolutionary life-cycle.

We will now first discuss run-time (Section 2.3) and then construction-time (Section 2.4).

2.3 Run-time

As already discussed earlier (Section 1.2), Unix pipelines provide a successful mechanism for the connection of the input and output of individual programs that provides (*i*) a standardized format to exchange data between programs (a list of strings); and (*ii*) a sequential composition operator to connect programs. However, pipes have several shortcomings. First of all, they lack the basic functionality to exchange data that have more structure than a list of strings. A second shortcoming is that only sequential composition of programs is provided: the output of program A can be connected to the input of program B, but it is impossible to feed partial outputs of B back to A. As a result, it is impossible to describe interaction and cooperation between programs very well. Both shortcomings are addressed below.

In previous work Bergstra & Klint (1996*a*, 1996*b*) and van den Brand, Klint & Verhoef have already argued that the key for obtaining a flexible run-time architecture is the separation between *computation*—the functions to calculate application dependent values and the procedures to perform application-dependent actions—and *coordination*—the flow of control between the computational parts of a system. A system using this distinction is naturally subdivided into components that carry out specific services such as, for instance, storage, retrieval, transformation of data, and user-interfacing. Each component carries out specific tasks and does not rely on direct connections with other components but relies on a coordination mechanism to make such connections. In this way, the same component can be reused in many ways by using it in combination with different components. We discuss now three key aspects: *coordination*, *representation of intermediate data*, and *testing and debugging*.

Coordination Bergstra & Klint (1996*a*, 1996*b*) have introduced the ToolBus, a component interconnection architecture resembling a hardware communication bus. To control the possible interactions between software components connected to the bus direct inter-component communication is not supported by the architecture.

The ToolBus serves the purpose of defining the cooperation of a number of *components* C_i ($i = 1, \dots, m$) that are to be combined into a complete sys-

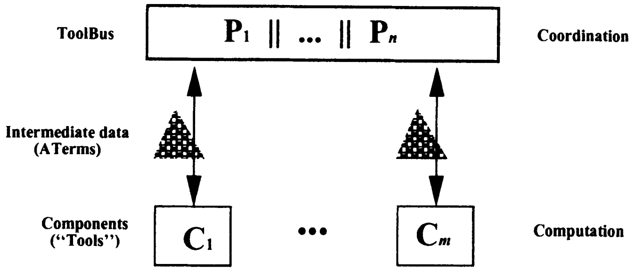

Figure 2 Global organization of the TOOLBUS.

tem as is shown in Figure 2. The internal behavior or implementation of each component is irrelevant: they may be implemented in different programming languages or be generated from specifications. Components may, or may not, maintain their own internal state. The *parallel process* $P_1 \parallel \cdots \parallel P_n$ in Figure 2 describes the initial behavior of the interaction between the components $C_1, \ldots, C_m$ and the interactions between the *sequential processes* $P_1, \ldots, P_n$. A sequential process P_i can, for example, describe communication between components, communication between processes, creation of new components, or creation of new processes. TOOLBUS processes also support relative and absolute time aspects. In the communication between the TOOLBUS and a component both can take the initiative for the communication. In this way, the TOOLBUS can send computation requests to components (ultimately resulting in some answer from the component), or the component can send a request to the TOOLBUS (typically, to notify the TOOLBUS of user-interaction or an error condition). The operators like the parallel composition operator $\parallel$ that we use in TOOLBUS processes stem from concurrency theory, more precisely, they are based on the algebra of communicating processes (ACP) described by Baeten & Weijland (1990) and Baeten & Verhoef (1995).

Coming back to the discussion on Unix pipelines (Section 1.2), it will be clear that we have extended the set of composition operations from only sequential composition to a much richer set of operators. Bergstra & Klint (1996*b*) give a comparison of the TOOLBUS with other approaches to software integration. Here it is sufficient to summarize the most innovative aspects of the TOOLBUS: (*i*) the process-oriented description of cooperation; and (*ii*) the standardization of the intermediate data exchanged between components as described in the next section.

The overall effect of the TOOLBUS approach is that components become black boxes that can transform and generate information in a common intermediate data format. We have explicitly chosen to ignore the *meaning* of the operations performed by the components; it remains the responsibility of a system's designer to compose components in meaningful ways.

Intermediate representation van den Brand, Klint & Verhoef (1996) have proposed a data structure, called the *Annotated Term Format (ATF)* specifically designed for the data exchange between (possibly) heterogeneous or distributed components in a software system. Instances of this format—such as communicated via the ToolBus—are called *ATerms*. This data format describes terms that can be dynamically extended with *annotations*. Annotations may be arbitrarily complex—even annotated—ATerms. ATF is able to accommodate the representation of *all* possible data that might be exchanged between components. The representation is such that individual components may access and modify annotations that have been added by other components but they may also completely ignore them. Components are thus immune for annotations they are not interested in.

ATF can also be used as internal data structure of components themselves. It is a powerful format in which it is possible to represent, for instance, parse trees that can be annotated with diverse information like textual coordinates, access paths, or the result of program analysis.

ATF has been implemented as a subroutine library (C, Java). An important characteristic of the implementation is that it maximizes the *sharing* of subterms. In other words, ATerms are treated as directed acyclic graphs with maximal sharing of subgraphs. In this way, we can maintain a simple, term-oriented, view of ATerms at the conceptual level but still provide an efficient and concise representation at the implementation level.

Testing and debugging In the ToolBus architecture, components are considered to be black boxes that transform and generate ATerms. This is an ideal starting point for testing, since it is very easy to test each component in isolation. As we will see in Section 2.4, we will frequently use formal techniques and executable specifications to prototype components. Such prototypes can be tested by applying them to appropriate test cases. Sometimes, the prototype will be replaced by a more efficient implementation. In those cases, an interesting back-to-back testing strategy becomes feasible: we can compare the behavior of the prototype with the behavior of the more efficient implementation of that component.

A final issue to be considered here is the debugging of ToolBus-based applications. We have to deal with concurrent and distributed execution, and components that may have been implemented in different languages. The problem here is how to provide a uniform debugging framework while still reusing native debuggers for language-specific debugging. Olivier (1997) describes the ToolBus Integrated Debugging Environment (TIDE) that is based on an abstract, event-driven, model for debugging. Typical events are calling a procedure and reaching or setting a break point. A source-code viewer and a variable inspector are examples of available generic tools that are based on this abstract debugging model. For each language L that is used for the implementation of a component, the abstract events can be implemented by the native L debugger. In this manner, a unified debugging view can be provided for all components while still relying on existing debuggers. TIDE itself has also been implemented as a component-based system and uses the ToolBus for coordination.

2.4 Construction-time

The run-time architecture just sketched, implies that construction is needed of (*i*) the coordination description; (*ii*) individual components. We will make no assumptions about the relation between the construction environment and the run-time environment. They may be completely disjoint, partially overlap, or even be identical. In a similar spirit, we do not assume that the construction-time environment is the same for each component.

We will now describe how to make *coordination descriptions*, and how to make *components*, with emphasis on the special case of language-oriented components.

Constructing coordination descriptions The construction of the coordination descriptions for the ToolBus amounts to writing process expressions describing the desired cooperation between components. Currently, we provide tools for static type checking of process expressions, and for the debugging and tracing of their execution. In principle, formal verification of process expressions is conceivable, but we have not yet worked on applications that really needed this.

Constructing components In a first approximation, we have nothing to say about the construction methods used for building components. Any software engineering technique or programming language can be used at the component level as desired. From an evolutionary perspective it is important to note that each component may be in a different evolutionary phase.

We have more to say about construction methods for components when they have something to do with language-processing such as, for instance, parsing, checking, compiling or transforming programs in existing or new languages. This is less restrictive than it may seem at first sight: in many problem areas one can follow a language-oriented approach since there is some domain-specific language (DSL) that characterizes essential aspects of the application domain, see Section 3.2.

The Asf+Sdf Meta-Environment (Klint 1993, van Deursen, Heering & Klint 1996, van den Brand, van Deursen, Klint, Klusener & van der Meulen 1996) is our preferred language prototyping tool. It is an interactive programming environment generator that takes a language definition as input (including a definition of the syntax of the desired language and optionally other operations on programs in the language such as, for instance, interpretation, compilation or transformation) and generates corresponding tools as output. From the syntax definition of the desired language various components are generated: a GLR parser, a syntax-directed editor, a pretty printer, optional traversal functions, and optional program analysis functions. For the operations defined on programs, efficient term-rewrite engines are generated. It will not come as a surprise that all these components can be connected to the TOOLBUS and that all intermediate data (e.g., parse trees) are represented as ATerms.

3 CASE STUDIES

3.1 Conversion tools for system renovation

The construction-time versus run-time dichotomy is also useful when building tools for the conversion of legacy systems. Consider building a tool for the automatic conversion between COBOL versions or dialects. In principle, one can write a single, monolithic, tool that performs the desired conversion. From an evolutionary perspective, however, such a monolithic tool is undesirable, since it will require major modifications when either a minor variation in the input language occurs (e.g., a vendor-specific language extension) or additional conversion steps are needed. Another disadvantage is that the conversion steps already implemented in the tool cannot be reused for other purposes.

Van den Brand, Sellink & Verhoef (1997*c*, 1997*b*, 1997*a*) describe a more evolutionary approach. During the construction phase of the conversion tool, the whole conversion process is first split-up in a number of smaller steps (e.g., parse, add end-if's, remove goto's, simplify conditions, pretty print). Next, each step is specified in ASF+SDF and the corresponding component is generated automatically. During the run-time phase of the tool, the individual components are tied together with a coordination architecture.

Since, the components are generated from specifications, they can more easily be adapted to changing circumstances. This is particularly true for the parsing component. As an aside, we like to mention here that the parser generation techniques we use are based on GLR parsing and can thus handle arbitrary context-free grammars (Rekers 1992). In contrast to more restrictive approaches (e.g., LL, LR), this opens up the possibility to develop *modular grammar descriptions* that can be composed in various ways. This is another example, of our component-based approach.

3.2 Domain specific languages

A DSL provides a conceptual framework and a notation that can be used to concisely express problem solutions in a particular application domain. Using compiler technology, texts written in a DSL can be compiled into, for example, executable code, calls to library routines, database transactions, or HTML forms. In this way, the DSL captures the essential knowledge of the application domain itself, while the corresponding DSL compiler captures the knowledge of the underlying infrastructure. Some characteristic domains where DSLs play a rôle are: (*i*) finance (e.g., modeling specific product classes such as, for instance, interest-based products, insurances, or option trading). (*ii*) electronic commerce (e.g., message formats, work flow); (*iii*) telecommunications (e.g., protocols, configuration of telephone exchanges); (*iv*) testing (e.g., test scripts); (*v*) multi-media (e.g., authoring scripts, content descriptions).

The knowledge that is necessary for a DSL and its compiler can come from two sources: a detailed domain analysis of the application area, or from reengineering a legacy system. In the former case, mainly human domain experts are involved in the design of the DSL. In the latter case, an existing legacy system is first analyzed and possibly transformed into a collection of reusable parts. These parts then form the starting point for a domain expert while designing the DSL. Existing knowledge is hence reshaped into a DSL. This illustrates the close relationship between the use of DSLs for forward engineering and for reverse engineering.

Arnold, van Deursen & Res (1995) describe the language RISLA that is intended for the definition of interest-based products. Using existing domain knowledge and a good library of COBOL routines, RISLA was designed and prototyped by means of the ASF+SDF Meta-Environment. Products defined with RISLA are compiled into COBOL programs that can be executed in a traditional administrative environment. From an evolutionary point of view it is interesting to observe that the first RISLA prototype was re-implemented, for efficiency reasons, using standard techniques (Lex and Yacc). After a few years, the language had to be redesigned in order to incorporate the experience gained with it. This turned out to be impossible with the existing, efficient, implementation and the redesign was again based on ASF+SDF. The current production version of RISLA uses ASF+SDF directly.

The use of DSLs has various advantages. First, the domain knowledge embodied by the DSL is easily available thus promoting its (re)use. This leads to explicit descriptions of the essential content only and leads to a speed-up of the software development process and better time-to-market. Second, the DSL compiler insulates the user from unnecessary details and severing machine dependencies. As a result, changes in the software and hardware infrastructure have to be accommodated only once in the DSL compiler. After re-compilation of all DSL programs, the desired infrastructure-related changes have been effectuated without making a single change to the DSL programs themselves. Contrast this with the traditional situation that domain-related knowledge and infrastructure-related knowledge are intermixed in the program code. The use of DSLs thus leads to greater flexibility and less maintenance (van Deursen & Klint 1998).

In large organizations one can identify several, distinct, application areas that are amenable to automation by means of DSLs. Each application area will have its own DSL, which is used to generate components. The use of a coordination architecture for interconnecting the generated components is sketched by van Deursen & Klint (1998).

3.3 Renovation of the ASF+SDF Meta-Environment

Van den Brand, Kuipers, Moonen & Olivier (1997) describe the renovation of the ASF+SDF Meta-Environment and apply an evolutionary, component-based, approach to our own tool suite.

At construction-time, formal specifications (in ASF+SDF) are used to specify the behavior of various components (e.g., syntax-directed editor, program repository, query engine) and component-generators (e.g., parser generator, pretty printer generator, ASF+SDF compiler). When efficiency dictates this, individual components that have been prototyped with ASF+SDF can be replaced by more efficient ones that are, for instance, implemented in C or replaced by an existing tool. Note that DSLs (Section 3.2) are used for describing, syntax, semantics, pretty printing, coordination, and user-interfaces.

At run-time, the TOOLBUS is used to connect components (the ones that have been implemented manually as well as the ones that have been generated) and they can be combined in many ways. In this way, tailored systems can be constructed easily.

4 PERSPECTIVE

We will now briefly summarize the software engineering approach presented here (Section 4.1) and discuss the relation with other approaches (Section 4.2). We conclude with a number of research topics: how to find components for reuse (Section 4.3), and the rôle of cost models for software (Section 4.4).

4.1 An emerging software engineering methodology

The evolutionary approach we have sketched in this paper can be characterized by the following principles:

- System parts are decoupled by separating *computation* from *coordination.*
- A process-oriented, formal, method describes the coordination of components.
- A common intermediate representation standardizes the exchange of information between components.
- Testing and debugging of components is standardized.

- A wide variety of software engineering techniques is used to construct components. Where applicable, we use formal specifications to design and prototype components.
- Each component of a system may be in a different phase of its evolutionary life-cycle and may be replaced anytime.
- The creation of distributed, heterogeneous, systems is accommodated.

We have illustrated all these principles with techniques and examples from our own experience. The same principles can, however, also be applied based on other technology.

4.2 Relation with other approaches

Contrary to popular belief, object-oriented languages do not form a panacea for software engineering in general and reuse in particular. Frakes & Fox (1995) show that reuse does not depend on the implementation language used. In particular, of the two OO languages appearing in their survey (SmallTalk and C++) no positive effect on reuse was found. Yourdon (1993) gives arguments for this. This explains why we have stressed the importance of heterogeneity: being able to combine programs written in different languages.

Although communication infrastructures like, for instance, CORBA (OMG 1996) also enable building multi-language applications, they do not provide the strong separation between coordination and computation as we do.

Contrary to many software engineering frameworks or approaches based on formal techniques, we take a very liberal stand: our evolutionary, component-based, approach is neutral to and can be combined with many different—formal or informal—methodologies and techniques.

4.3 Identifying components for reuse

Recalling the discussions on code scavenging as a technique for reuse (Section 1.2) and on forward engineering versus reverse engineering (Section 2.1), we observe that there are two approaches to define components for reuse: (*i*) the *a priori* definition of parameterized components, and (*ii*) the *post facto* recovery of components from existing code.

Parameterized data types, such as modules and components, have a long history in computer science and they have been applied successfully in many projects. Defining a parameterized data type anticipates the reuse of the data type by establishing which parts can vary (the parameters) and which parts are fixed (all other aspects of the data type). Depending on the context, type constraints may be imposed on the instantiation of the parameters. Parameterized data types correspond, for example, to generics in Ada, and templates in C++. Reuse of parameterized data types is hindered by the fixed parameters—the data type could lend itself to more forms of reuse than can be expressed by the fixed set of parameters—and by limitations of static type constraints that forbid dynamic forms of polymorphism.

Parameterized components can also be recovered from existing code. One approach is to analyze existing code and search for strong correlations between data and code (van Deursen & Kuipers 1997). The outcome of such an analysis can be either code restructuring or conversion of the existing code into class definitions in an object-oriented language. Mayrand, Leblanc & Merlo (1996) describe the use of metrics for finding function clones. Another approach is to manually "frame" existing code fragments into reusable components to which parameters are added as needed (Bassett 1996). Various fragments from the original code can then be regenerated from a single, reusable, frame by means of formalized low-level editing operations. In this way the set of reusable frames can slowly grow, thus decreasing the size of the code base that has to be maintained.

We conclude that techniques stemming from research on parameterized data types (type constraints, syntax-directed view on modules) and from the reengineering field (clustering, framing) could be fruitfully combined to open new perspectives on finding and defining reusable components.

4.4 Usage-based cost models for components

Cox (1996) argues that there is no economic incentive for building high-quality, reusable, software components. In his view, the mere notions of owning, selling and buying electronic property—such as software—form the key problem. The current pay-per-copy schemes for software distribution prevent easy experimentation with new software products and make it hard to establish a realistic price for each product that is based on its inherent characteristics such as, for instance, functionality, implementation effort, or lines of code. Instead, the distribution of software should be free and payment should be based on pay-per-use schemes. He proposes a new commercial infrastructure that opens up the potential for using large systems that are composed of many components, where each use of a component directly generates revenues for its owner/implementor. Since components can easily be replaced by better (and cheaper) ones, a competition between component manufacturers will emerge.

We agree with Cox's vision and observe that the software engineering approach we propose in this paper is compatible with it: pay-per-use schemes can easily be incorporated in the coordination architecture we have described. Issues to be addressed are security and confidentiality of the payment transactions and compatibility with emerging standards for electronic commerce.

5 ACKNOWLEDGMENTS

Section 3.2 is based on a text from a research proposal co-authored by Arie van Deursen.

REFERENCES

Arnold, B.R.T., A. van Deursen & M. Res (1995), An algebraic specification of a language for describing financial products, *in* 'ICSE-17 Workshop on Formal Methods Application in Software Engineering', IEEE, pp. 6–13.

Baeten, J.C.M. & C. Verhoef (1995), Concrete process algebra, *in* 'Handbook of Logic in Computer Science, Volume IV, Syntactical Methods', Oxford University Press, pp. 149–268.

Baeten, J.C.M. & W.P. Weijland (1990), *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press.

Basili, V.R., G. Caldiera & G. Cantone (1992), 'A reference architecture for the component factory', *ACM TOSEM* **1**(1), 53–80.

Bassett, P.G. (1996), *Framing Software Reuse*, Yourdon Press, Prentice-Hall.

Bergstra, J.A. & P. Klint (1996*a*), The ToolBus coordination architecture, *in* P.Ciancarini & C.Hankin, eds, 'Coordination Languages and Models', Vol. 1061 of *LNCS*, pp. 75–88.

Bergstra, J.A. & P. Klint (1996*b*), The discrete time ToolBus, *in* M.Wirsing & M.Nivat, eds, 'Algebraic Methodology and Software Technology', Vol. 1101 of *LNCS*, Springer-Verlag, pp. 286–305.

van den Brand, M.G.J., A. van Deursen, P. Klint, S. Klusener & E. van der Meulen (1996), Industrial applications of ASF+SDF, *in* M.Wirsing & M.Nivat, eds, 'Algebraic Methodology and Software Technology', Vol. 1101 of *LNCS*, Springer-Verlag, pp. 9–18.

van den Brand, M.G.J., M.P.A. Sellink & C. Verhoef (1997*a*), Control flow normalization for COBOL/CICS legacy systems, Technical Report P9714, University of Amsterdam, Programming Research Group.

van den Brand, M.G.J., M.P.A. Sellink & C. Verhoef (1997*b*), Generation of components for software renovation factories from context-free grammars, *in* I.Baxter, A.Quilici & C.Verhoef, eds, 'Proceedings of the Fourth Working Conference on Reverse Engineering', pp. 144–153.

van den Brand, M.G.J., M.P.A. Sellink & C. Verhoef (1997*c*), Obtaining a COBOL grammar from legacy code for reengineering purposes, *in* M.Sellink, ed., 'Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications', Electronic Workshops in Computing, Springer Verlag. To appear.

van den Brand, M.G.J., P. Klint & C. Verhoef (1996), Core technologies for system renovation, *in* K.Jeffrey, J.Král & M.Bartošek, eds, 'SOFSEM '96: Theory and Practice of Informatics', LNCS, Springer-Verlag, pp. 235–254.

van den Brand, M.G.J., P. Klint & C. Verhoef (1997), 'Reverse engineering and system renovation – an annotated bibliography', *ACM Software Engineering Notes* **22**(1), 57–68.

van den Brand, M.G.J., T. Kuipers, L. Moonen & P. Olivier (1997), Implementation of a prototype for the new ASF+SDF meta-environment, *in* M.Sellink, ed., 'Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications', Electronic Workshops in Computing, Springer Verlag. To appear.

Cederqvist, Per (1993), *Version Management with CVS*, Signum Support AB, Box 2044, S-580 02 Linkoping, Sweden.

Chikofsky, E.J. & J.H. Cross (1990), 'Reverse engineering and design recovery: A taxonomy', *IEEE Software* **7**(1), 13–17.

Cleaveland, J. Craig (1988), 'Building application generators', *IEEE Software* **5**(4), 25–33.

Cox, B. (1996), *Superdistribution: Objects as Property on the Electronic Frontier*, Addison-Wesley.

van Deursen, A., J. Heering & P. Klint (1996), *Language Prototyping: An Algebraic Specification Approach*, Vol. 5 of *AMAST Series in Computing*, World Scientific Publishing Co.

van Deursen, A. & P. Klint (1998), 'Little languages: Little maintenance?', *Journal of Software Maintenance* . To appear.

van Deursen, A. & T. Kuipers (1997), Finding classes in legacy code using cluster analysis, *in* S.Demeyer & H.Gall, eds, 'Proceedings ESEC/FSE 97 Workshop on Object-Oriented Reengineering', Technical Report TUV-1841-97-10, pp. 1–5.

Fayad, M.E. & D.C. Schmidt (1997), 'Special issue on object-oriented application frameworks', *Communications of the ACM* **40**(10).

Frakes, W.B. & C.J. Fox (1995), 'Sixteen questions about software reuse', *Communications of the ACM* **38**(6), 75–87.

Hoare, C.A.R. (1996), How did software get so reliable without proof?, *in* M.-C.Gaudel & J.Woodcock, eds, 'Proceedings of the Third International Symposium of Formal Methods Europe: Industrial Benefit and Advances in Formal Methods', Vol. 1051 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 1–17.

Horowitz, Ellis, Alfons Kemper & Balaji Narasimhan (1985), 'Survey of application generators', *IEEE Software* **2**(1), 40–54.

Kelly, K. (1994), *Out of Control: The New Biology of Machines, Social Systems, and the Economic World*, Addison-Wesley.

Klint, P. (1993), 'A meta-environment for generating programming environments', *ACM TOSEM* **2**(2), 176–201.

Krueger, C.W. (1992), 'Software reuse', *ACM Computing Surveys* **24**(2), 131–183.

Lientz, B.P. & E.B. Swanson (1980), *Software Maintenance Management—A Study of the Maintenance of Computer Application Software in 487*

Data Processing Organizations, Reading MA: Addison-Wesley.

Lim, W.C. (1994), 'Effects of reuse on quality, productivity, and economics', *IEEE Software* **11**(5), 23–30.

Mayrand, J., C. Leblanc & E.M. Merlo (1996), Experiment on the automatic detection of function clones in a software system using metrics, *in* S.Bohner & C.A, eds, 'International Conference on Software Maintenance', IEEE, pp. 244–253.

McConnell, S. (1993), *Code Complete*, Microsoft Press.

McConnell, S. (1996), *Rapid Development*, Microsoft Press.

McIlroy, M.D. (1969), Mass produced software components, *in* P.Naur & B.Randell, eds, 'Software Engineering', pp. 138–150.

Naur, P. & B. Randell, eds (1969), *Software Engineering—Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, NATO Science Committee, Garmisch, Germany, 7–11 October, 1968.

Olivier, P. (1997), Debugging distributed applications using a coordination architecture, *in* D.Garlan & D. L.Métayer, eds, 'Coordination Languages and Models', Vol. 1282 of *LNCS*, pp. 98–114.

OMG (1996), *CORBA: Architecture and Specification*, Object Management Group (OMG).

Prieto-Diaz, R. & P. Freeman (1987), 'Classifying software for reusability', *IEEE Software* 4(1), 6–16.

Rekers, J. (1992), Parser Generation for Interactive Environments, PhD thesis, University of Amsterdam.

Reutter, J. (1981), Maintenance is a management problem and a programmer's opportunity, *in* A.Orden & M.Evens, eds, '1981 National Computer Conference', Vol. 50 of *AFIPS Conference Proceedings*, AFIPS Press, Arlington, VA, pp. 343–347.

Ritchie, D.M. & K. Thompson (1974), 'The UNIX time-sharing system', *Communications of the ACM* **17**(7), 365–375.

Yourdon, E. (1993), *Decline and Fall of the American Programmer*, Prentice-Hall.

Yourdon, E. (1996), *Rise and Resurrection of the American Programmer*, Prentice-Hall.

6 BIOGRAPHY

Paul Klint is research group leader at the Centrum voor Wiskunde en Informatica, and professor of computer science at the University of Amsterdam. His research interests include programming environment generators, domain specific languages, coordination architectures and the integration of forward and reverse engineering techniques for software.

Chris Verhoef is an assistant professor at the University of Amsterdam. His research interests are software engineering, system renovation, reverse engineering, maintenance, and theoretical computer science.