HEWLETT PACKARD

Building Object-Oriented Instrument Kits

Martin L. Griss, Robert R. Kessler*
Software Technology Laboratory
HPL-96-22
February, 1996

component, domain-specific kits, framework, object-oriented, OO methods, software reuse, WAVE

Quick development of related instrument applications requires a new approach—combine techniques of systematic software reuse and component-based software with object technology and RAD. Reusable domain-specific frameworks, components, glue languages, and tools must be designed to work together to simplify application construction. Previous papers define such compatible workproducts as domain-specific kits and describe a framework for their analysis and comparison. Using this kit model, we analyze the Hewlett-Packard Visual Engineering Environment (VEE) as a domain-specific kit. Contrasting these analysis results with those for Microsoft's Visual Basic® (VB), we can identify the key features required for an experimental next-generation version of such an instrument kit. We gain maximum benefit by integrating systematic OO methods to develop reusable architecture, framework, and components with a flexible, domain-specific visual application assembly environment. Using Objectory™ as the design tool and VB as the implementation vehicle, we have prototyped an open, component-oriented VEE-like environment, the WAVE Engineering System. WAVE is a flexible kit that supports high levels of software reuse. We illustrate how object technology, reuse, and RAD methods are combined by applying WAVE to a simple LEGO™ instrumentation kit implemented in VB. We describe some results of our experiments and what we have learned about kit construction using OO methods and VB implementation. This description will enable the reader to use the concept of kits to structure and package reuse deliverables, and also indicate how technologies such as VB can be used as a powerful starting point.

Internal Accession Date Only

* *University of Utah*, Salt Lake City, Utah.
To be published in *Object Magazine*, April/May 1996.
© Copyright Hewlett-Packard Company 1996

Building Object-Oriented Instrument Kits

Martin L. Griss

Hewlett-Packard Laboratories
1501 Page Mill Rd.
Mail Stop 1U-16
Palo Alto, CA 94304-1126
griss@hpl.hp.com

Robert R. Kessler

University of Utah
3190 M.E.B.
Department of Computer Science
Salt Lake City, UT 84112
kessler@cs.utah.edu

Abstract

Quick development of related instrument applications requires a new approach -- combine techniques of systematic software reuse and component-based software with object technology and RAD. Reusable domain-specific frameworks, components, glue languages, and tools must be designed to work together to simplify application construction. Previous papers define such compatible workproducts as domain-specific kits and describe a framework for their analysis and comparison. Using this kit model, we analyze the Hewlett-Packard Visual Engineering Environment (VEE) as a domain-specific kit. Contrasting these analysis results with those for Microsoft's Visual Basic ® (VB), we can identify the key features required for an experimental next-generation version of such an instrument kit. We gain maximum benefit by integrating systematic OO methods to develop reusable architecture, framework, and components with a flexible, domain-specific visual application assembly environment. Using Objectory ™ as the design tool and VB as the implementation vehicle, we have prototyped an open, component-oriented VEE-like environment, the WAVE Engineering System. WAVE is a flexible kit that supports high levels of software reuse. We illustrate how object technology, reuse, and RAD methods are combined by applying WAVE to a simple LEGO ™ instrumentation kit implemented in VB. We describe some results of our experiments and what we have learned about kit construction using OO methods and VB implementation. This description will enable the reader to use the concept of kits to structure and package reuse deliverables, and also indicate how technologies such as VB can be used as a powerful starting point.

1. Introduction

Our goal is to build many different, yet similar, measurement systems rapidly. To accomplish these goals for application family development, we combine Rapid Application Development (RAD) techniques, domain-specific kits for reuse, objects for their maintenance and robustness capabilities and three-tier architecture for distribution. A disciplined approach, using object-oriented (OO) analysis and design methodology is key to structuring cost-effective, robust, and easy to maintain systems. All of these must be integrated into a complete and consistent system. However, none of this can be accomplished without some constraints. Our main constraint is that full OOA/OOD is too complicated and time consuming for high-level RAD by domain-experts and thus must be adapted to be usable in this context.

In the rest of the paper, we first review the role of systematic reuse and object technology, and describe our target of heterogeneous, three-tier distributed measurement system. Then we describe how we are integrating disciplined OO methods and pragmatic visual construction environments. We then review domain-specific kits and the kit analysis framework. Using that framework, we analyze an existing

domain-specific tool from the instrument domain (HP-VEE), comparing it with Visual Basic as an open, domain-generic system. Combining these techniques with the three-tier architecture has resulted in a system we call WAVE, implemented as an example domain-specific visual construction environment extending VB. We briefly discuss our experience using WAVE and Objectory in a kit-based software engineering class at the University of Utah. Finally, we discuss what we have learned and our conclusions.

1.1 Rapid Construction of Instrument Application Families

Of particular interest to Hewlett-Packard and its customers is the ability to rapidly and cost-effectively construct measurement systems of various kinds. These systems combine instruments (sensors and actuators), computation for control and analysis, enterprise information systems, and networks for distributed integration of instruments and various enterprise systems. These systems are conceptually very similar and can often be viewed as a family of related systems.

Examples include a variety of manufacturing systems, test systems, automotive test systems, chemical analysis systems, and electronic home control systems. These systems are characterized by an increasingly large number of available instrument components and the increasing use of PCs with Microsoft Windows 95® or Windows NT™, networked to other machines. Pressure is increasing to make rapid building of customized systems easier, systems that can be easily changed by various domain experts. These characteristics are shared by many other kinds of application domains, although we will focus on measurement systems in this paper.

A key desire has been to create software components that can be reusable yet are easy to combine with other reusable elements into a complete system. In this paper, we describe how we have explored and combined aspects of systematic software reuse, component-based software, and object technology to address some of these issues.

1.2 Software Reuse Is Crucial

Systematic reuse is an organizational approach to product development in which software assets are purposefully created to be reusable [Griss93]. These assets are then consistently used and maintained to obtain high levels of reuse, allowing the organization to produce quality software products rapidly and effectively. Systematic reuse is widely believed the best way to significantly improve the software process for the construction of product families, by improving quality, reducing development and maintenance costs, and decreasing time to market [Frakes94].

To establish an effective reuse program, a systematic approach is needed to change process, organization, technology, and many other factors. The scope of these changes pervades the software development organization. Success with reuse requires attention to both technical and non-technical factors, such as process, organization, management, architecture, and tools. In addition to these organizational changes, technology needs to be modified for effective use [Griss95].

1.3 Object Technology Can Help

As discussed in [Griss95], one of the most frequently stated reasons for selecting object technology (OT) is to enhance component and design reuse. Software reuse is a widely desired benefit of employing objects. Encapsulation, inheritance, and polymorphism are powerful tools to produce reusable components and frameworks.

But simply adopting OT will not ensure effective systematic reuse. Too many object-oriented reuse efforts fail, often because of a focus on technology alone. Without an explicit agenda for reuse, OT will not lead to reuse [Pittman93]. Nevertheless, some form of OT is surely the most promising enabling technology for large-scale component reuse.

Furthermore, to obtain significant amounts of reuse, several aspects of OT must be modified or used very carefully [Lorenz93]. We need support for domain engineering to identify components and architecture for product family design and implementation. We need distinct software engineering processes for separate component Creator and Utilizer organizations. We must augment most of today's OT methods with a reuse-oriented macro process and reuse-supportive architecture [Griss96]

Our work at HP on domain-specific kits [Griss94,Griss95b,Griss95c] has led us to add kit-like features to the Objectory ™ method we use, incorporating several other techniques to produce flexible, customizable, layered, modular environments. Modifications and additions specifically support domain engineering, and reuse-oriented object model factoring and packaging [Griss95d]. Distinct software engineering processes support component engineering and application engineering. Kit-like additions include problem-oriented scripting languages. Simple system and component generators in the form of Wizards help mask the complexity.

1.4 Distributed Measurement Systems in a Heterogeneous, Three-tier Environment

We are prototyping an open, extensible, distributed software environment, "Sensorium," for the development, distributed control, and enterprise-wide/Internet-wide integration of measurement systems. In this application domain, some of the data servers and client servers are actually (programmable) instruments. Our user community is interested primarily in using PC-based Microsoft tools and components, including Visual Basic, Visual C++, Microsoft Office, and Microsoft BackOffice, as well as in connecting them to HP-UX systems. Our target implementation environment is a three-tier system with a tight coupling of Windows NT and HP-UX workstations working in concert (see Figure 1). The NT systems are used as the front-end user interface. NT servers and HP-UX machines typically handle the various business objects and back-end database processing services of the three-tier approach.

Some services will be provided as World Wide Web- (WWW-) interfaces running on the clients with WWW-compatible business objects on the WWW server, while others will integrate client interfaces into MS Office on the client with business objects on the client or business object server. CGI scripts or Java[1] are used to define business objects, running on a data or client server. Java will also be downloaded to program certain instruments. Application partitioning determines whether business objects reside on the data server, business server, or in client space (e.g. Java and Visual Basic objects). Remote OLE/COM helps to hide this distinction.

[1] Java is a highly portable, interpretive dialect of C++, used predominantly as a WWW applet programming language, though it was originally designed as a form of instrument programming language at Sun Miscrosystems.

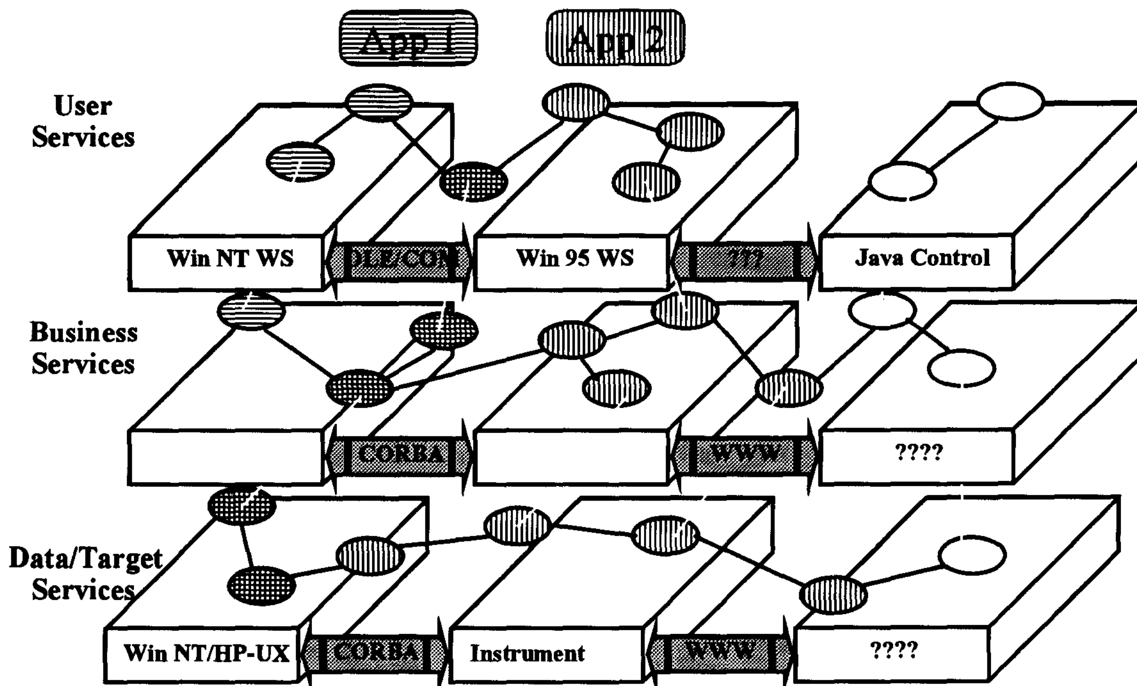


Figure 1: Three-tier model with Windows, HP-UX, and network resources.

2. Integrating OO Methods with Visual Construction

In our work, we have been guided by a vision of an integrated, multi-paradigm approach to rapid, yet disciplined, application development, as shown schematically in Figure 2. The triangular shape is meant to show how we use a kit approach to resolve the tension between the use of disciplined OO/model-based methods to define complex systems (on the left), and a more pragmatic prototyping and implementation approach (on the right). Linthicum[Linthicum96] ascribes some of this tension to pressure caused by short time frames; little time is left to do disciplined OOA/D and reuse in a traditional iterative RAD cycle. Our goal is a development approach in which these approaches coexist more effectively.

The layers roughly correspond to elements associated with a domain-specific kit, discussed further in section 3: framework, components, languages, and tools. Lower layers correspond to more-conventional implementation technologies, used perhaps by systems programmers, while upper levels are closer to the needs of domain experts and use more domain-oriented tools. From left to right, we show the transition from requirements analysis to implementation. At the bottom level, the full power of a complete OO method is used to analyze requirements, design the system architecture and infrastructure, and implement the appropriate framework. At the next level up, this OO model and the framework interfaces are used as a basis to develop compatible components, using a lighter-weight component-oriented OO method, with coding templates and component generators, driven by these models and interfaces to simplify component construction. The "conceptual distance" from analysis to implementation is thus decreased, and the overall process should be simpler for the component builder..

Finally, at the top level, we use a very-light-weight OO modeling language, or even a domain-specific OO-based modeling language, which can be closely connected to a RAD visual programming or interpretive domain-specific language to customize and glue components together. One might have no visible modeling at all, simply assembling icons drawn from a domain-specific palette. One might also use some non-code models to develop and evaluate aspects of the applications that have certain overall properties that are hard to obtain directly in code. In some cases, model-based application generators will

present the modeling and implementation options in a conversational way, perhaps in the form of a "fill in the blank" parameterized use-case.

We thus use full-blown, disciplined, reuse- and kit-adapted OOA/OOD methods to develop systems architecture and to develop components, business objects, and key subsystems using model and code templates shaped by the architectural work.

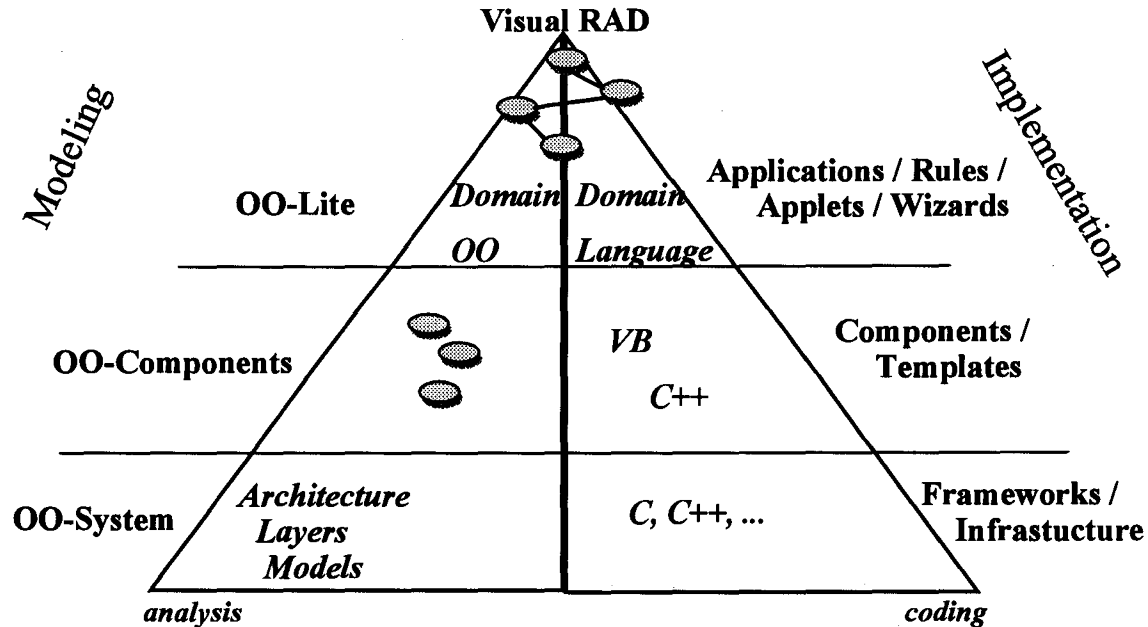


Figure 2: Our model for rapid, iterative problem analysis and coding

2.1 Visual Assembly and Rapid Prototyping

We use a RAD-like combination of lightweight OO methods, prototyping, and problem-specific visual environments to selected, customize, and combined components into applications. We imagine using some Wizard-like technique that allows the user to customize use-cases or object models to then drive the assembly of components. We are exploring the use of the new capabilities of the Microsoft Open Integrated Development Environment (IDE) and other Visual Basic extension mechanisms to develop these custom environments as natural extensions to Visual Basic itself. An important issue is developing techniques that allow the interfaces and business object templates developed using Objectory to connect to the appropriate VB service interfaces and IDE handles, so that Visual Basic can be used as a form of domain-oriented configuration or scripting language.

The first step was to extend Visual Basic with a problem-oriented visual programming capability that will allow one to visually and rapidly select, customize, and glue together business objects to create custom applications. WAVE, described in section 4, is our first step in this direction. Several other such extensions might be needed to support the different tasks, and to produce tools peaked for scripting, gluing, business rule editing, or business rule process/workflow modeling.

Visual Basic itself is used to provide the graphical interface portion of the applications (the user service of the three-tier application), while the extensions provide the integration with the business objects in the second tier and the database objects in the third tier.

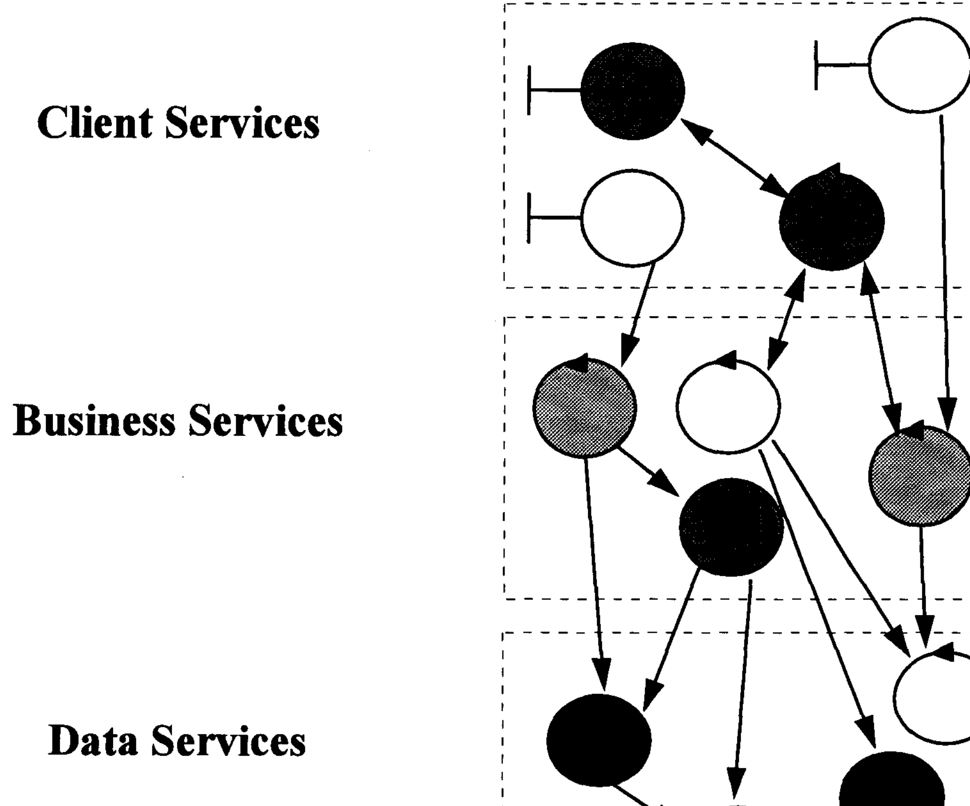
2

2.2 Using OO Methods for Systematic, Kit-based Reuse

As described previously [Griss95], domain analysis is the starting point for the systematic design of a reuse architecture and reusable assets for a family of related applications. Example systems, user needs, domain expertise, and technology trends are analyzed to identify and characterize common and variable features of the domain. While an OOA/OOD systems representation method, such as Objectory, can be used to express appropriate variability for reuse, it cannot explicitly support all the needed styles of variability. The modeling capabilities must match the more powerful parameterization, linguistic, and generative implementation techniques. Examples include Basset's NETRON/CAP frame representation [Bassett91,Bassett95], our domain-specific kits [Griss95d], and other techniques for generating implementations [Batory94.

In our work, we use a lightweight version of Objectory to construct the reusable components and define the system architecture and infrastructure. Figure 3 shows how Objectory interface, control, and entity objects naturally partition to fit the three-tier architecture. A collaborative project with Ivar Jacobson (described in a column [Griss96] and tutorial [Griss95d]) is developing reuse architecture and process extensions to OOSE[Jacobson92] and OO Business Engineering [Jacobson94]

During the OOSE analysis phase, we partition the system functionality into interface, control, and entity objects, which we loosely assign to the client, business, and data services layers, as shown in Figure 3 in



general and in Figure 4 for WAVE. We have some choices as to where we place certain functionality, such as instruments and robots.

2.3 Using Current OO CASE Tools to Generate and Combine Kit Elements

Several OO CASE tools, such as Rational's ROSE and Objectory SE, Select Tool, and Platinum's Paradigm Plus, have a customizable code generation capability. This has been used by the various tools to generate C++ and Visual Basic from the models. It has also been used to generate IDL interfaces. These generated code and interface files are naturally consistent with the models. This capability can be used to

generate components and framework infrastructure for a kit. We believe it can even be used to generate complete applications for subsequent customization.

Some of the tools also provide a reverse engineering capability, such that if the generated code is manually changed in an appropriate way, it can be parsed and the changes can be reflected back in the model. This capability can be used to support some amount of RAD-like iteration, using for example Visual Basic as a glue code and GUI builder. The changed code can then be reintegrated with the models.

Finally, some of these tools support customizable notation, OLE control of the tools, and access to the underlying model database. This capability could make possible the construction of model-driven generators that could be called from the Visual Basic IDE environment to aid in creating specialized components. In principle, it should be possible to build a relatively domain-specific environment, with domain-specific icons on the model palette, driven by the underlying OO modeling system.

In our own work, we have chosen to first build the visual programming environment, WAVE, as an extension of Visual Basic, as described in section 4. This separate environment is then linked to these other tools for the model connections.

3. Domain-specific Kits

3.1 Background

The kit concept has evolved over several years. It originated with the observation that many application construction systems can offer very high productivity by using a form of hybrid reuse, combining component composition with some form of system generation [Biggerstaff87]. The kit concept was derived by studying the characteristics of these successful systems [Griss93]. We started by looking at features of simple software games such as the Pinball Construction Kit and The Incredible Machine®. Fischer's domain-oriented design environments at the University of Colorado had many interesting tool concepts [Fischer92]. We also looked at larger product systems such as the ATMS manufacturing process software toolkit from Philips and several Hewlett-Packard instrument toolkits. These instrument toolkits have instrument definition languages, domain-specific environments, and loadable components. The key to improved productivity using each of these systems was the presence of several kinds of software elements, such as components, frameworks, problem-oriented languages, and other domain-specific tools, working together to provide a domain-specific environment.

We also studied a variety of problem oriented and scripting languages, such as Technikron's TDL, Microsoft's VBA, and Osterhout's TK/TCL [Osterhout94]. These are used to glue together and customize components, and they use appropriate domain-specific extensions. NETRON CAP also displays kit-like features[Bassett91,Bassett95]. In NETRON, an application is described by a hierarchical set of frames. Each frame references lower-level frames and may includes explicit "editing" instructions to modify, include, or exclude lower frames. Frames also inherit default behavior and parameters from higher-level frames. The frames are processed by a special generator to create appropriately customized target workproducts, such as code, data, or documents.

In the following sections, we describe the concept of kits and kit analysis, which we then apply to Visual Basic and the HP-VEE instrument environment.

3.2 Domain-specific Kits Package Reusable Workproducts

We define a domain-specific kit [Griss94] as a set of compatible, reusable "elements" that work well together to make it easy to build a family of related applications in some domain. A kit developer uses domain engineering methods to structure the application domain, and kit engineering methods to build domain-specific components, framework, and tools [Griss95b]. The kit can then be used by application developers to rapidly build one or more applications in this domain. While kits typically make visible the

distinction between components and framework, several domain-specific tools, languages, and generators hide complexity.

A domain-specific kit may include several kinds of kit elements:

- **Components**—Well-documented, tested, and packaged sets of compatible software assets of various types, such as C functions, C++ objects, test files, specifications, documentation, and examples.
- **Framework**—An instantiation of the architecture and interfaces, providing common services and core functionality, and enabling component integration, interoperability and customizability.
- **Language(s)**—Notation to combine or customize components and adds functionality. A language could be general-purpose, a flexible generic scripting language, or a highly domain-specific textual or visual language.
- **Exemplary Application(s)**—Prepackaged, ready-to-run applications (or subsystems) built with the kit, which also serve as a base for incremental creation of other applications.
- **Environment and Tools**—Component browsers, glue and customization language editors, application builders and generators, and macro recording tools that aid in quick assembly of complete applications.

By observing available application technologies from this "kit" perspective, one can see new ways to use them effectively. The different elements of the kit work together to support the various developers and users o f the technology. While these kit-like features are extremely useful to the practical delivery of reuse, most object-oriented frameworks unfortunately do not systematically include integrated problem-oriented languages, generators, or other domain-specific environment extensions. It is critical to consider kit user experience and their skills when designing a framework and components, so that they are not too complex for the intended utilizers Generally, application-oriented domain experts will greatly benefit from a custom problem-oriented language and generators; many find inheritance and C++ too complex. If generators or domain-specific languages are used, they must be designed together with the other kit elements.

3.3 Analysis of VB as a Kit

In a previous *Object Magazine* article [Griss95a], we described how Visual Basic could be viewed as a kit. We use the kit analysis framework [Griss95c], a set of attributes, and a tabular method to examine the kit elements and their characteristics. We describe the analysis of VB and VEE, which led to our experiment of implementing WAVE, a VEE-like kit as an extension of VB. Of some importance are which elements are present, which are domain-specific, how complete and extensible the kit is, and how well the domain-specific metaphor is supported by framework and tools.

VB is a major part of Microsoft's application development and customization strategy [Udell94]. It plays a pervasive role in conjunction with C++ and other languages and is packaged in several distinct ways, either as the embedded customization language, Visual Basic for Applications (VBA), or as a series of more powerful systems.

Visual Basic includes the following important kit features:

- **Domain**—The domain is very generic, targeted primarily toward any graphical user interface-based application, although one can use the system to write non-graphical applications too.
- **Components**—VB components (VBX or OCX) comprise several compatible parts, including icons, code, help file, Windows resource, property sheet, event procedure stubs, and OLE interface.
- **Framework**—The VB framework is built on the Windows API. The framework is generic but enforces a Windows GUI, event-driven communication, and prototypical component structure.
- **Language**—VB's interpretive, generic programming/scripting Basic language glues components together and specifies behavior as method bodies and global procedures. The language includes

extensions for objects, methods, properties, event-driven programming, OLE access, and external routines. Property sheets associated with each component are a form of "template language."

- **Exemplary Applications**—Hundreds of applications are included, and many more are available from numerous other sources.
- **Environment and Tools**— VB's development and execution includes a visual "builder" and conversational generators, called Wizards. Using browsers and editors, the developer customizes the component's appearance and behavior, changing graphical attributes, properties, event actions, and method bodies.

VB is complete in the sense that it is a full programming language. VB's power comes from the careful balance between a rather simple object and component model and its easy-to-use, interpretive environment. A large variety of built-in and third-party components and add-on tools, plus its openness, add to its appeal. Our analysis and experimentation with both VB 3.0 and VB 4.0 suggest that VB is extensible enough to be the basis for a family of compatible, domain-specific kits.

3.4 Analysis of VEE as a Kit

The Hewlett-Packard Visual Engineering Environment for Windows [Hensel93] (HP-VEE or VEE HP) is an iconic programming language whose primary domain is engineering problems. It is especially well suited to allow the user to collect data from instruments, analyze/filter it, and then display the results in some meaningful format. The use of a domain-specific, iconic programming language means that the user need not write "conventional" programs. VEE is therefore a domain-specific solution that permits instrumentation engineers to quickly build complex control systems.

- **Domain** -- HP-VEE focuses on engineering problems that interact with instruments.
- **Components** -- Primary are the instrument drivers (instruments). These pieces of software provide a graphical and functional description of how one interacts with an instrument. Many instruments (or instrument-like components) are supplied with the VEE distribution, but VEE also allows the user to create instruments using the "wizard-like" HP Driver Writer Tool and associated Compiler.
- **Framework** -- A dataflow model transfers vectors of instrument data for display and processing.
- **Language** -- A visual, iconic environment using wired connections along which the data flow.
- **Exemplary applications** -- Are presented as sample applications that also may be cut and pasted into new programs.

VEE is reasonably complete, since any reasonable program can be written in the graphical programming language. However, writing non-instrument applications (e.g., access to databases, interrupt-driven serial I/O, complex computations) using VEE is very clumsy. On the openness/extensibility front, one can certainly add new instruments as long as they fit the VEE model of instrument interaction; however, other extension capability is very limited. VEE provides limited access to other programs.

3.5 Allocation of Domain Specificity to Kit Elements

Not all kit elements must be individually domain specific to produce a useful domain-specific environment. Alternative technology choices lead to different degrees of domain specificity for the kit elements and the environment as a whole, and different degrees of kit extensibility and interoperability. For example, with a generic framework and glue language, combining different domain-specific components and kits is relatively easy. Further, some components and subsystems can be generated using distinct domain-specific tools, without disturbing other parts of the kit. Each kit element (even in one kit) can be implemented using different technologies and mechanisms, and can be parameterized and customized in different ways. Overall, the kit must capture and present to the user the appropriate programmability in a metaphor that is as convenient and domain specific as possible. Component and

tool mechanisms and packaging can be carefully designed to allow an appropriate balance among integration, coherence, domain specificity, and extensibility.

Given the importance of domain specificity and openness, we compared these aspects of the various kit elements of VB and VEE from an instrument systems perspective. This really is at the core of the decision of how to extend either. The key elements are summarized in the following table.

	Domain Specificity	Openness
Components	VEE: *many instrument drivers and displays*	VB: *OCX mechanism, rich set of OCXs*
Framework	VEE: *Dataflow engine, measurement vectors*	VEE & VB: *mechanisms rigidly built in*
Glue	VEE: *wiring, dataflow, control interconnect*	VB: *general-event Basic, DLL, OLE access*
Environment	VEE: *visible flow on wires*	VEE: *hard to extend, only DDE, can be further extended by rebuilding* VB: *loadable OCX mechanism, Wizards, add-ins, menu extensions*

As can be seen, VEE has more domain-specific components and glue language, while VB lacks instrument system domain-specific features but provides a more open, extensible environment. So, to meet our goal of creating a next-generation VEE-like system that integrated better with the enterprise and had more features for custom VB-compatible interfaces, we could either extend VEE to work well with VB or add instrument components and a graphical glue language to Visual Basic.

Following the analysis, we compared the structure, performance, and ease of coding of several small programs inside and outside of the instrument domain. We also evaluated PC software component technology trends (OLE, OCX, and VB evolution) and investigated low-level VEE extensibility experience. This analysis led us to chose the route of exploring the addition of several VEE-like domain-specific features to VB, in the form of specially packaged components, additional tools, and environment extensions.

4. WAVE—Wired Component Assembly in Visual Basic

In this section, we describe a prototype measurement systems kit, Wired Component Assembly in Visual Basic (WAVE), for a simple instrumentation domain. For reasons we will explain below, we have chosen to create our kit as an extension to Visual Basic. The first prototype is operational, using Objectory for system and component design. We will connect Wave to the Objectory model as described above.

4.1 Why we Chose VB as the Basis for an OO Kit

Why are we using VB to build OO kits, when one might imagine that SmallTalk, TCL, C++, or Java might be the more natural choice? Our kit project started in mid-1994, when many instrument system developers really wanted the power, flexibility, and openness of VB. Instrument Basic, HP-VEE, and increasingly VB, were popular with instrument engineers. They were interested in a system that would deliver the functionality of VEE in a VB-compatible context. Prior use of C++ to build instrument system frameworks had shown that it was disappointingly difficult for most instrument engineers to use and extend. The large variety of VBX and OCX components and OLE-accessible MS Office applications made VB the obvious choice.

To appeal to these potential users we used VB. We started with the belief that VB would be OO "enough." Certainly VB has a refined and accessible component model. As explained in [Udell94], to support component reuse, VB uses parameterization, generation, aggregation, instead of inheritance, to

tailor specific components. VB 3.0 had very strong encapsulation and a form of polymorphic method dispatch. While it had no inheritance and was limited in user-defined methods, we were able to use a simple Wizard to generate specialized components from a more generic component. VB 4.0 is even more effective. It provides user-defined classes with methods and properties (still no inheritance), a more extensible environment, and better integration into the Windows environment by allowing construction of DLLs and local and remote OLE automation servers.

4.2 WAVE: An Experimental VEE/VB Instrument Kit

The analysis of VEE and VB led to the following description of our target kit, called WAVE (Wired Assembly and Visual Execution). WAVE combines the best features of VB and VEE:

- **Domain** -- Instrument control; extension to other domains is easy by replacing the components.
- **Components** -- Targeted toward the instrument control domain. These components can be defined with Visual Basic or other languages supporting a DLL-based model.
- **Framework** -- VEE-like dataflow via the transfer of vectors of instrument data.
- **Language(s)** - Visual, iconic environment using wired connections, and VB as scripting language.
- **Exemplary applications** - Many samples to control the various devices.
- **Tools**- We anticipated the need for one Wizard to create visual wrappers and connectors for components.

We decided from the outset to attempt to stay entirely within VB, despite its known deficiencies for very large programs. Our goal was to push it to its limits and thus gain an understanding of its strengths and weaknesses. The strength of the resulting kit comes from its implementation in VB being highly open and extensible. Only if performance became critical, or if certain features were inaccessible, would we descend into C or C++, or use the Windows API [Appleman93].

One of our first needs was to choose an appropriate set of instruments to control. We decided to focus on the electronic control of devices constructed with computer-controlled LEGO® parts. These parts offer many of the same types of problems and constraints as standard instruments (direct, real-time sending of commands and receiving/processing data), yet demonstrate a level of visual expressiveness that help to cement the subliminal message that "software components should be as easy and fun to use as LEGOs."

Our resulting system is the WAVE graphical programming environment. WAVE allows the construction of "click and wire" programs that interact with the user, robots, motors, and sensors. Figure 4 shows how the WAVE objects fit the typical three-tier architecture shown in Figure 3. The top, client-services level consists of the applications written with WAVE to control the instruments. In addition, several other client applications interact directly with the instruments—for example, a program that allows the user to manually control the motors and devices of an instrument system. The business objects are the high- and low-level components manipulated by the WAVE visual construction. Some of these components simply turn on a device or read a sensor value. Higher-level components control a robot to rotate to a correct location or allow a sensor subsystem to determine the color of a LEGO brick. We currently have two data services, one a database that stores information related to the specific application scenario and the other a program we call the *box server*, which interacts with the attached devices. The box server accepts higher-level commands (e.g., "sense this device for 10 seconds and return an array of sensor values") and turns them into low-level operations (e.g., "send the appropriate byte stream commands to turn on a motor at port 3 at speed 4"). The OOSE use-case and analysis model for the box server is shown in Figure 5 and Figure 6, showing the various parts that interact clients and instruments.

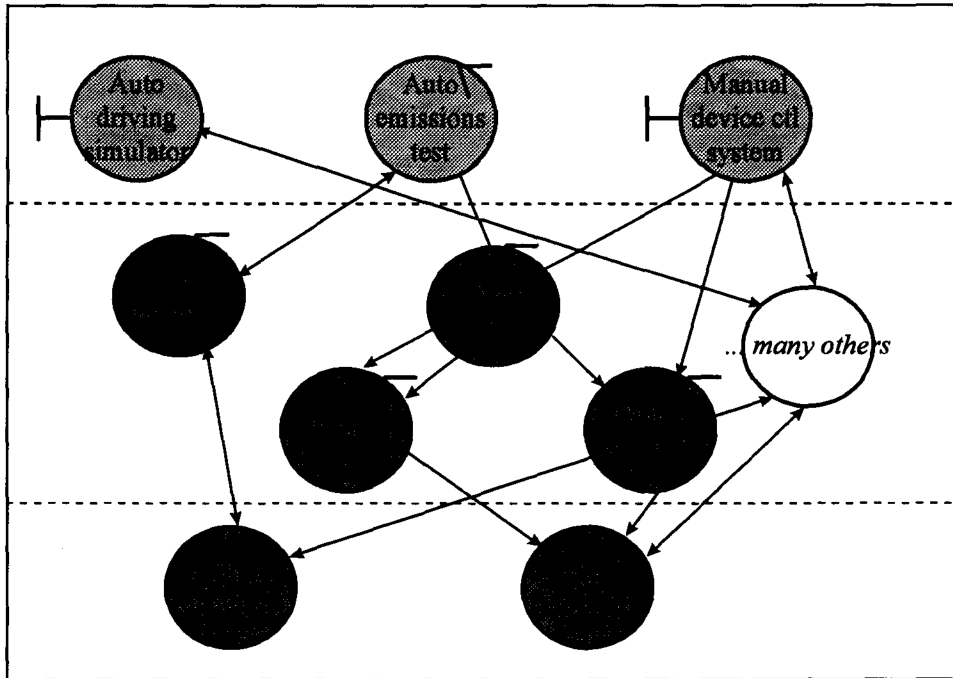


Figure 4: The WAVE architecture

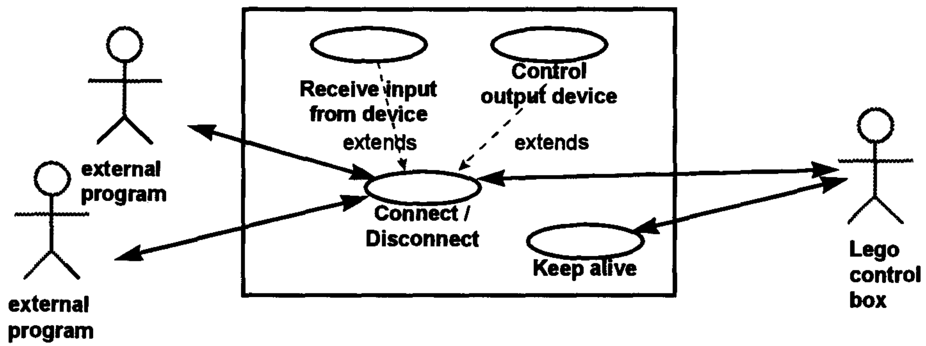


Figure 5: Use-case model for the box server part of WAVE

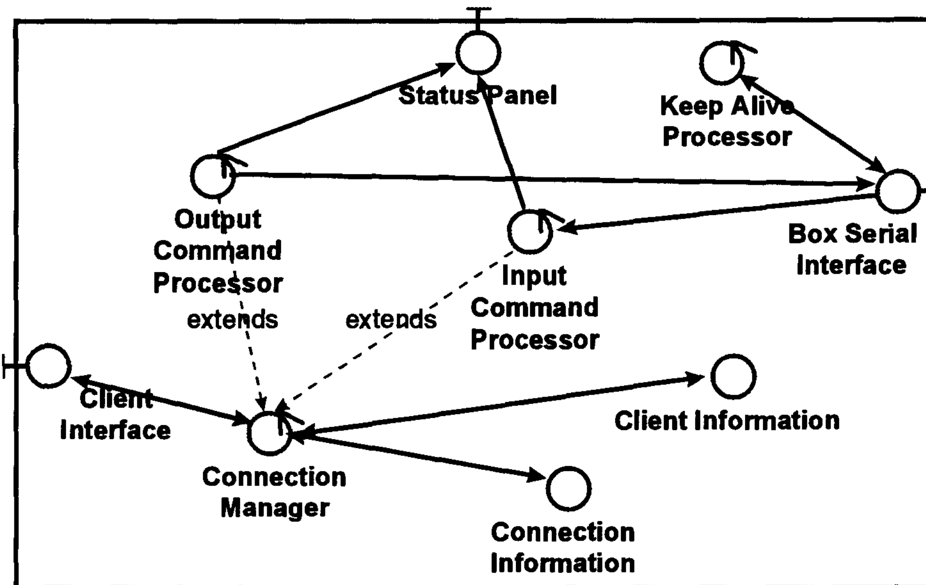
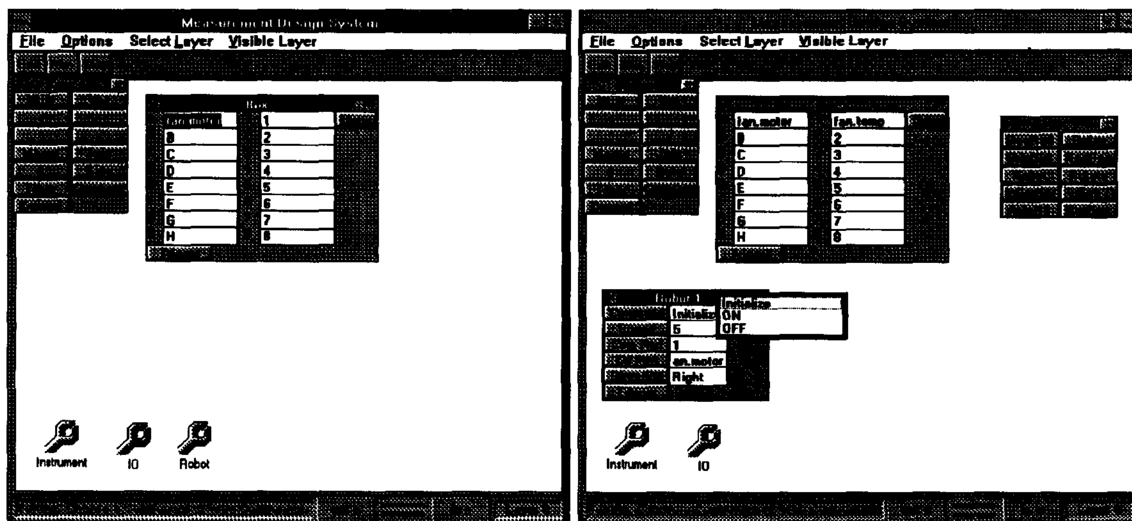


Figure 6: Analysis model for the WAVE box server

4.2.1 An example WAVE program

Figure 7 shows an example of designing a simple fan control program with WAVE that will turn the fan motor on for 5 seconds. We start with an empty layer and several palettes of components. In Figure 7a, we select the box component, which allows us to assign names to the Control Lab devices. Once named, devices then be accessed using the object-oriented or VB style of prefixing the name with "`box.`" ("`box.fan_motor`" in this example). In Figure 7b, we have selected a "robot" control component, which allows us to send commands to an output device. We then choose the type of command from the pop-up menu. Notice that several of the input fields have been filled in with values, including the control device ("`box.fan_motor`"), which references the fan motor device connection as defined in the box component. Figure 7c shows addition of a delay component with the delay time set to 5,000 milliseconds. Finally, Figure 7d shows the all of the components, including wires. At this point, pressing the **Run** button allows the program to execute. During the execution, each component's top menu bar is highlighted while that component is executing.



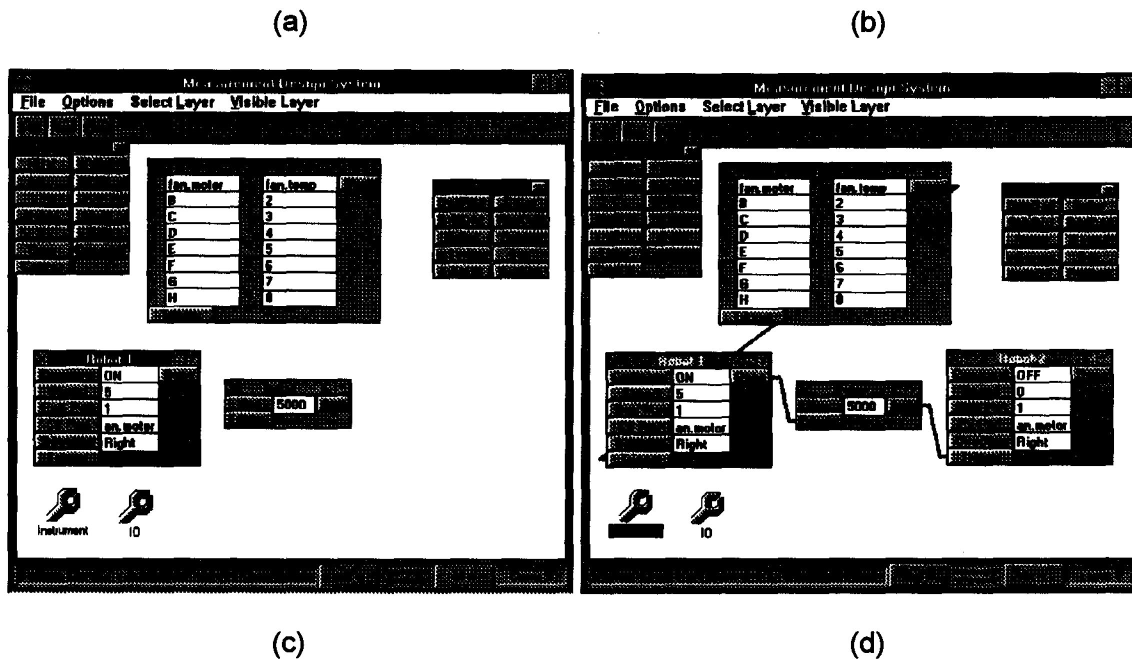


Figure 7: Designing a fan control using WAVE.

4.2.2 Experience with WAVE in a Kit-based Software Engineering Class

Based on our experience using Objectory and VB while constructing WAVE, we decided to use Objectory, WAVE, and VB as the basis of a component-based software engineering class at the University of Utah. We are teaching a three-quarter-long senior undergraduate software engineering class emphasizing reuse. In the first quarter (fall 1995), we had students use WAVE to control simple LEGO-based robots. Then they used Visual Basic directly to control other instruments (scanners and turtles built from LEGO and Fischerteknik) using some of the WAVE components. A lightweight subset of Objectory (essentially OOSE [jacobson92]) was used for requirements-capture and high-level design (see http://cs.utah.edu/~cs451). Coyne and DuToit[Coyne96] describe similar experiences using Objectory in a one-semester software engineering class at CMU. While not focused on reusable components, they did a subsystem decomposition and implementation in C++.

In the second quarter, begun January 1996, we are building a larger manufacturing system. Students are using more of Objectory and the Objectory CASE tool to design the subsystems and components and to explore alternative user scenarios. Informal domain engineering sharpens the definition of components, which are being implemented in VC++ and VB. Objectory models are being used to design and document the reusable components produced by some groups for use by others. A separate graduate student project is extending WAVE as a visual assembly interfaces for the more robust components.

Students are learning the concepts of disciplined software reuse-based software engineering, with components, frameworks, and kits. They relish the power that is available from rapid program construction using component-oriented environments such as WAVE, Visual Basic, and OLE.

4.3 WAVE Next Steps

So far, we have only used the OO architecture and component models informally. The next step is to formalize this process to produce components that are consistent with the Visual Basic OCX structure yet are shaped by the OO framework. We are designing the next generation of WAVE based on an augmented VB IDE. We will use the OO models to drive a visual assembly tool and component wizard. We are

exploring several routes for such extensions in the form of design add-ins, Wizards, component manager extensions, and WAVE-like wired programming. These components can then be rapidly assembled and wired together by WAVE. Some customization of these components or business rules can be written in Visual Basic itself or in Java.

5. Conclusions

5.1 Refined Concept of Kits

The project was very helpful in understanding several of the tradeoffs that must be made in developing a kit. In particular, our kit analysis showed that VEE was indeed a domain-specific kit, but it lacked enough openness to make it easy to move into the enterprise and also allow more flexible design of the user interface portions of an instrument control program. VB solves both of these limitations with its facilities for design of the user interface and its easy integration into the enterprise, but it lacked any domain specificity in terms of instrument control. Taking the best parts of each, we added graphical programming capabilities and a framework for instrument component creation to Visual Basic, resulting in an excellent second-generation prototype. WAVE still needs more tuning for performance, most likely involving conversion of some parts into C/C++.

5.2 Recommendations

We are convinced that one should deliver domain-specific reuse in the form of a kit, not just as components and framework. Domain-specific languages and tools dramatically simplify the task presented to the domain expert. With an open kit, the user is able to start application building at the most domain-specific level, yet descend to more analysis or more code as needed.

We are also convinced that it is quite feasible to combine disciplined OO analysis and design methods with a more visual, RAD-like environment. The design and evaluation of the basic mechanisms and some components are well under way. We have informally carried out the transformation from Objectory model to components and code, but we have yet to demonstrate full integration in operation.

We find that that VB is attractive for demonstrating the kit concept and for implementing a variety of educational and practical kits. Students were able to grasp the concepts rapidly and build quite sophisticated applications. Students were quite comfortable with the combination of a lightweight OO modeling preceding a cycle of RAD-like implementation in VB.

6. Acknowledgments

The authors are grateful to the many Hewlett-Packard and Utah staff members who reviewed earlier drafts of this paper and made significant contributions to the work, including Randy Coverstone, Regan Fuller, Jon Gustafson, Michelle Miller, Keith Moore, Christian Mueller-Plantiz , Marc Tischler, Kevin Wentzel, and Lorna Zorman. We appreciate the support provided by Fischertechnik, Hewlett-Packard, LEGO, Microsoft, and Objective Systems AB (now Rational), which made various parts of the project possible.

7. References

Appleman93 D Appleman, *Visual Basic Programmer's Guide to the Windows API*, 1993, Ziff-Davis Press.

Bassett91 PG Bassett, *Software Engineering for Softness,* American Programmer, 4(3), March 1991.

Bassett95 PG Bassett, *To Make or Buy? There is a Third Alternative*, American Programmer, 8(11), November 1995.

Batory94 D Batory et al., *The GenVoca Model of Software System Generators*. IEEE Software, 11(9), September 1994.

Biggerstaff87 T Biggerstaff and C Richter, *Reusability Framework, Assessment and Directions*, IEEE Software, 4(2), March 1987.

Coyne96 RF Coyne and AH DuToit, *Teaching Model-based Software Engineering*, Object Magazine, January 1996.

Fischer92 G Fischer, *Domain-Oriented Design Environments*, Proceedings of the Seventh Knowledge-Based Software Engineering Conference, IEEE, September 1992.

Frakes94 W Frakes and S Isoda, *Success Factors of Systematic Reuse*, IEEE Software, 11(5), September 1994.

Griss93 Martin L. Griss, *Software Reuse: From Library to Factory*, IBM Systems Journal, 32(4), November 1993.

Griss94 ML Griss and K Wentzel, *Hybrid Domain-Specific Kits for a Flexible Software Factory*, Proceedings of SAC '94, March, ACM, 1994.

Griss95 ML Griss, *Systematic Software Reuse—Objects and Frameworks are Not Enough*, Object Magazine, February 1995.

Griss95a ML Griss, *Packaging Software Reuse Technologies as Kits*, Object Magazine, October 1995.

Griss95b ML Griss and RR Kessler, *Visual Basic does LEGO*, Proceedings of VBITS 95, San Francisco, Visual Basic Programmers Journal, March 1995.

Griss95c ML Griss and KD Wentzel, *Hybrid Domain-Specific Kits*, Journal of Systems and Software, September 1995.

Griss95d ML Griss and I Jacobson, *The Reuse-Driven Software Engineering Business—Architecture and Process for Systematic OO Reuse*, OOPSLA-95 Tutorial, Austin, TX, October 1995.

Griss96 M Griss, *Systematic Software Reuse - A year of Progress*, Object Magazine, February 1996.

Hensel93 R Hensel, *Cutting your Test Development Time with HP-VEE - An Iconic Programming Language*, Hewlett-Packard Professional Books, Prentice-Hall, 1993.

Jacobson92 I Jacobson, M Christerson, P Jonsson and G Overgaard, *Object-Oriented Systems Engineering: A Use-case Driven Approach*, Addison-Wesley, 1992.

Jacobson94 I Jacobson, M Ericsson and A Jacobson, *The Object Advantage - Business Process Reengineering with Object Technology*, Addison-Wesley, 1994.

Linthicum96 DS Linthicum, *Breaking OO out of the "time box,"* Object Magazine, January 1996.

Lorenz93 Mark Lorenz, *Facilitating Reuse using OO Technology*, American Programmer, August 1993.

Osterhout94 JK Osterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, 1994.

Pittman93 Matthew Pittman, *Lessons Learned in Managing Object-Oriented Development*, IEEE Software, 10(1) January 1993.

Udell94 J Udell, *Component Software*, Byte Magazine, May 1994.

8. Biographies

Dr. Martin L. Griss is a Laboratory Scientist at Hewlett-Packard Laboratories, Palo Alto, where he researches object-oriented reuse and measurement system kits. As HP's *reuse rabbi,* he led work on software reuse process, tools, software factories, and the systematic introduction of software reuse into HP's divisions. He is an Adjunct Professor at the University of Utah working with Professor Kessler on reuse-based software engineering education. He was previously director of the Software Technology Laboratory and has over 25 years of experience in software engineering research. He has authored numerous papers and reports on reuse, writes a reuse column for Object Magazine, and is active on several reuse program committees. He was until 1982 an Associate Professor of Computer Science at the University of Utah, where he worked on portable Lisp compilers and interpreters and on computer algebra systems.

Dr. Robert R. Kessler has been on the faculty of the University of Utah since 1983; he currently holds the position of Associate Professor of Computer Science and Acting Chairman. Prior to a 1994-1995 sabbatical at Hewlett-Packard Research Labs, Professor Kessler was director of the Center for Software Science, a research group working in nearly all aspects of system software for sequential and parallel/distributed computers. He has interests in parallel and distributed systems, portable Lisp systems, architectural description–driven compilers, software engineering, and general systems software. He has experience with practical business software as well as academic software development and most recently has been working on component-oriented software development using Visual Basic and OLE. Professor Kessler completed his first textbook, *Lisp, Objects, and Symbolic Programming*, in 1988. He is currently Co-Editor-in-Chief of the International Journal on Lisp and Symbolic Computation.