Dividing the Software Pie

J. Craig Cleaveland
Janet A. Fertig
George W. Newsome

Systematic software reuse, or multiuse, is a key to increasing the productivity and quality of software development. In the past 20 years, reuse has experienced many failures and few successes. Many technological, organizational, and cultural obstacles have been placed in its path. A critical step to increasing software reuse is to recognize that a new division of labor is required, one in which component developers create reusable components and product developers compose products from these components. Changing organizational structure and software development processes to nurture these roles is challenging. Once these roles are recognized and established, however, standard abstraction techniques and other software reuse technologies can help separate the concerns of component developers and product developers. This paper illustrates the separation of concerns by examining its application to interfaces, a particularly difficult area in which these concerns are traditionally intertwined.

Introduction

If cars and computers can be designed from components, why not software? The emerging software component industry, still in its infancy, may someday provide a basis for extensive software reuse. Until that day arrives, organizations can foster their own internal version of such an industry. Too often, organizations approach software reuse as if it were just another technique, tool, or skill to add to their list. Over and over again, the software reuse literature warns us about the pitfalls of such an approach.

The second section of this paper, "A New Division of Labor," presents the way in which labor is divided between component developers and product developers. Experience teaches us that division of labor is at the heart of software reuse (see Panel 2), and no techniques, tools, or skills alone can substitute for it. Other industries have clearly separated development roles. For example, computer components (such as disk drives and power systems) are developed independently from computer systems (composed from computer components). Software designers need to maintain the same separation. They must also make a clear distinction between developing software components for a variety of products and composing products from software components.

Decisions are the fundamental unit of effort in the software development process. The third section of this paper, "Decisions, Decisions, Decisions," describes the decision process and also introduces some domain engineering concepts, standardization versus parameterization, and platforms.

For software reuse to work effectively, the concerns of component developers and product developers must be separated. The fourth section of this paper, "Separation of Concerns," describes the technological support needed to do that. Fortunately, standard software engineering practices and abstraction techniques are suitable for this purpose.

This paper concludes by illustrating how abstractions can be applied in the particularly thorny area of interfaces.

Reuse versus Multiuse. Despite the efforts of researchers to infuse the term "software reuse" with a very specific meaning, it is popularly associated with any kind of sharing. The word "reuse" implies using something not originally intended for the purpose at hand, thereby getting something for (almost) nothing. Instead of "free software," however, one often encounters obstacles that require changing the software in ways that the original designers did not intend. This effort almost always constitutes salvaging the item in question—that is, copying and modifying an artifact, resulting in the creation of many similar artifacts. Each artifact then needs separate maintenance, which reduces the benefit of sharing. Thus, salvaging reaps only part of the potential benefits of reuse, and in some cases the costs may outweigh the benefits. Salvaging may be the only option available when software is not designed for reuse.

Opportunistic reuse is the unplanned or serendipitous reuse of software.[1] This paper introduces a new term, "multiuse," to avoid the ambiguity that the word reuse connotes. Multiuse implies design for multiple uses. Although the term reuse may seem to imply only opportunistic reuse and salvaging, the software community generally defines the term reuse to mean sharing something across multiple products, independently of how it was designed, created, or reused. Therefore, multiuse is a form of reuse, sometimes called "systematic reuse."[2]

Multiuse has three major principles:

- Components are designed for a range of products, rather than a single product.
- Products are designed with reuse in mind, by favoring a compositional, rather than decompositional, approach to design.
- A distinction is made between the common parts of a component, which are not touched or salvaged, and variable parts, which can be customized for a range of products.

The first two principles imply distinct development roles with different concerns, even when the same individual or organization performs those roles. The last principle implies a need for separation of concerns between these development roles.

Panel 1. Abbreviations, Acronyms, and Terms

API	application program interface
CAST	configuration and synthesis tool
component	a distinct part of a product. Components may be created from first principles every time by the organization building the product, reused from other products, or obtained from a component supplier.
commonality	a decision about how members in a domain are alike
dependency	a decision or assumption about the environment in which a component is built or executed
domain	a defined and delimited subject area. In software, an application area or a set of related features. For instance, performance monitoring, user interfaces, and protection switching are domains.
GPN	Global Public Networks, an AT&T business unit
IDL	interface definition language
MIL	module interconnection language
MTE	multiuse test environment
multiuse	design and implementation of a component for a range of products, also called systematic reuse
OO	object oriented
product	a system built by an organization and provided to a customer in return for revenue. It may include manufactured goods, software-only goods, and items that are passed through AT&T. Some products may be components of other products.
SWCB	Software Conveyor Belt team
systematic reuse	design and implementation of a component for a range of products, also called multiuse
variability	a decision about how members in a domain differ

A New Division of Labor

By its nature, multiuse implies a new division of labor. In a multiuse process, components are developed to be used in a range of products, and products are assembled from such components. Because these two different activities have different goals, concerns, and priorities, they are given two distinct labels: component development and product development, as shown in Figure 1. This division of labor occurs whenever it becomes too expensive to develop products from first principles every time.

Component Developers. Component developers are concerned with the range of products that might use their components. They must understand what is common and sharable across that range, and also what varies and must be customized. Domain analysis determines which portions are shared, called *commonalities*, and which portions are variable, called *variabilities*. Domain implementation is the creation of multiuse components that satisfy commonalities and variabilities. Each multiuse component comes with a customizing process that allows product developers to "customize" the component for their product. Together, domain analysis and domain implementation make up a process called domain engineering.[3] Component developers are solution experts whose knowledge about solutions within a particular domain is captured, packaged, and reused.

Product Developers. Product developers are the problem experts; they know the customers and the problems that require solutions. They define and build products for specific customers or markets. Rather than decomposing a design into unique components that probably must be built anew, product developers "compose" products from multiuse components—that is, they design products with reuse in mind. They also customize each multiuse component by using a prescribed customizing process provided by the component developer. This may simply mean selecting the variations needed for their product.

As with any new division of labor, this leads to more specialized tasks, particularly among component developers who specialize in specific domains. Specialization increases productivity, even without the benefits of software reuse, by allowing the large-scale reuse of domain knowledge and expertise across an organization.

The Need for Separating Concerns. The concerns of component developers and product developers were not previously recognized as being worthy of separation. To work efficiently, an organization must allow the right people to make decisions at the right time without getting in each other's way. If an organization cannot separate these concerns into independent decision-making capabilities for component developers and product developers, its atmosphere becomes too chaotic. Some difficulties perceived as organizational obstacles to reuse may really be symptomatic of a technical failure to separate concerns. While this might suggest the need for new technology or tools, such tools can only be as good as the model they present. A model of how the necessary concerns are separated must be defined before methods and tools are proposed to support it. (See Panel 3.)

Panel 2. Reuse and Culture

The ideas presented in this paper originate in work performed by the Software Conveyor Belt (SWCB) team, formed in 1992 by the executives of the former Transmission Systems Business Unit (TSBU), now part of the Global Public Networks (GPN). SWCB was charged with developing multiple-use software strategies across the organization.

As is the case in many reuse initiatives, the SWCB discovered early that the primary barriers to widespread reuse are cultural and organizational, not technical. However, the experience of the SWCB team suggests that the oft-cited barriers of "not-invented-here" and concerns over quality are not significant issues, as borne out by the results of a recent industry survey.[14] Instead, they learned that the chief nontechnical challenges facing any organization wanting to carry out planned or systematic reuse are those facing any business partnership.

Even when reuse is mandated, the challenges are essentially those of a business partnership and they must be met by the same approaches used in developing external partnerships. These include:

- Recognizing that a partnership is required,
  - Alignment of business goals;
- Establishing the context of the partnership,
  - Equitable (not necessarily equal) benefits,
  - Accountability versus risk, among all partners,
  - Commitment and predispositions; and
- Taking the actions necessary to support the partnership,
  - Fostering shared knowledge, mutual dependency, and organizational linkages.

Clearly, it is easier to talk about a partnership than to create one.[15] However, the success of any systematic reuse initiative depends on it.

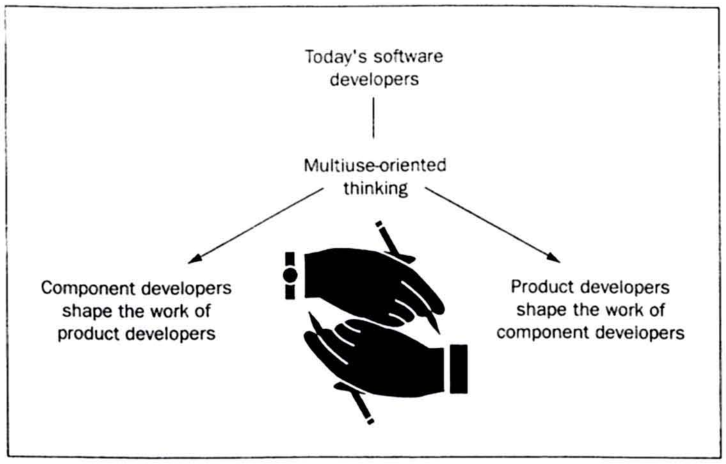
Today's software developers
Multiuse-oriented thinking
Component developers shape the work of product developers
Product developers shape the work of component developers

Figure 1. In the multiuse model, the role of today's software developer is split into two closely coupled, but distinct, roles. The influence that each role has on the other makes it difficult to achieve the separation required to reap the benefits of multiuse. Special attention must be paid to achieve this separation.

Decisions, Decisions, Decisions

Software development is a series of decisions. Decisions are made throughout the software life cycle. Decisions are made about requirements, software architecture, interfaces, algorithms, data representations, programming languages, operating systems, host and target environments, tools, test and debug strategies and tools, and even the intricate details of line-by-line coding. According to Brooks,[4] the hard thing about software is making the decisions, not coding them. It is difficult enough for a single team to handle and coordinate a complex set of decisions. It becomes more difficult when teams of component developers and product developers must manage and coordinate these decisions, particularly if the teams are separated in time, as well as place and organization.

Just as it is necessary to distinguish between component developers and product developers, it is also important to distinguish between the decisions to be made during component development, and those to be made later, during product development (see Figure 2). During component development time, the component developer makes decisions about a component for a range of products. In particular, the component developer attempts to distinguish clearly between commonalities and variabilities. A commonality is a decision made by the component developer about how a component will work in all products. A variability is a decision about the component that the product developer or product user will make.

Variabilities. There are two important kinds of variabilities: build-time variabilities and run-time variabilities. A *run-time variability* is often implemented as additional parameters to functions and procedures. Less often, run-time variabilities are implemented as "resource files," which are read in at run time and then interpreted. *Build-time variabilities* are often implemented with macros or program generators. Both kinds of variabilities make a component reusable in a wider range of products.

Commonalities and Dependencies. *Commonalities* are decisions (or assumptions) made by the component developer about the environment in which a component is built, tested, installed, or operated. These are also called *dependencies,* because the component is dependent on such decisions. For example, a component that uses the UNIX* operating system is dependent on it. Dependencies are potential barriers to software reuse. Some dependencies, such as UNIX dependencies in an environment in which everyone uses UNIX, may never be a problem, while other dependencies, such as a C++ program in an organization that exclusively uses Smalltalk, may block potential software reuse in most circumstances. Dependencies are safe or reasonable if they do not present barriers for a range of products.

Dependencies include hardware interfaces, processors, operating systems, databases, programming languages, interfaces, communication mechanisms, build and test environments, and requirements. A dependency can only be created by the component developers (which

Panel 3. The Need for Organizational Separation

What are the implications for organizations in recognizing the different roles at work in creating a product based on multiuse? Specifically, is it better for the distinct roles of component developer and product developer to be performed by the same individual, different individuals in the same organization, or by separate organizations?

Without these different roles, decisions about the product would become embedded in the components. Conversely, the product would depend on decisions governed by components, because the components are considered white boxes to the product developers. The skills required by component developers are those of domain solution technology, while the skill required of product developers is the ability to choose component sets that can fulfill some other purpose. It is reasonable to suppose that combining the roles in one individual will lead to the same lack of separation as was previously observed. When wearing the component developer's hat, the developer will be inclined to embed product-specific functionality into the component; when wearing the product developer's hat, the developer will find it difficult to avoid taking advantage of internal component knowledge. When the roles are given to different individuals in the same organization, it is difficult to separate the different business goals of the component builders and product builders, because an organization tends to address a single goal.

Further impetus for organizational separation of roles results from recognizing the nature of software implied by the multiuse model. In this model, software components, though intangibles, represent capital investment, and not just cost. The factory metaphor is appropriate, because any factory requires an investment to allow the rapid production of products. Component building can be seen as the tooling-up phase, while product development becomes software production. Labor-intensive software, in which software is built from first principles every time, is a strategy for reducing short-term investment. A capital-intensive approach, such as multiuse, is a strategy for reducing long-term costs. An appropriate balance between short-term and long-term goals can only be achieved if there is also a balance in funding between product development and component development. This balance of funding is typically maintained through sometimes minor organizational separation of product development and component development.

For these reasons, it is difficult to combine the roles and still maintain the essential separation. A business partnership between distinct organizations, although difficult to establish and maintain, provides the separate, clearly delineated roles and relationships that are needed.

include domain analysts), because the decisions they make affect many products. On the other hand, the decisions made by product developers are limited to a single product.

Commonalities are not simply dependencies to be avoided. On the contrary, commonalities are the only way of sharing something across a range of products. A component with no commonalities has nothing shared across a range of products, and therefore has no economic value as a reusable piece of software. Similarly, a component with few commonalities has little economic value. Commonalities represent what is sharable or reusable across a range of products. Increasing commonalities increases the economic leverage of a component.

Tradeoffs Between Commonalities and Variabilities. A classic tradeoff exists between commonalities and variabilities. Each decision is potentially a commonality or a variability. Commonalities increase economic leverage at the possible expense of shrinking the product range, while variabilities expand product range at the expense of limiting shared parts. How does one determine whether a decision should be a commonality or a variability? Many decisions are obviously one or the other, but some could go either way. So the question becomes: Should we standardize or parameterize?

Standardizing creates a commonality, whereas parameterizing creates a variability. Domain analysis guides these decisions. However, a domain analysis divorced from the organization cannot easily determine which decisions should become new organizational standards. Such domain analysis efforts necessarily err on the side of variabilities, thus making the component more generally useful, but also more expensive, with a smaller common sharable portion. Attaining a more efficient solution requires a tighter coupling between the domain analysis effort and the organizational commitment to supporting the effort and any resulting standards.

For example, to make reusable components that are simpler and cheaper to construct, an organization may decide to incur the costs and pains of standardizing on an operating system for all its software products. If, on the other hand, an organization decides on an isolated domain analysis effort, it may be forced to build a more complex component that can be adapted to a variety of operating systems.

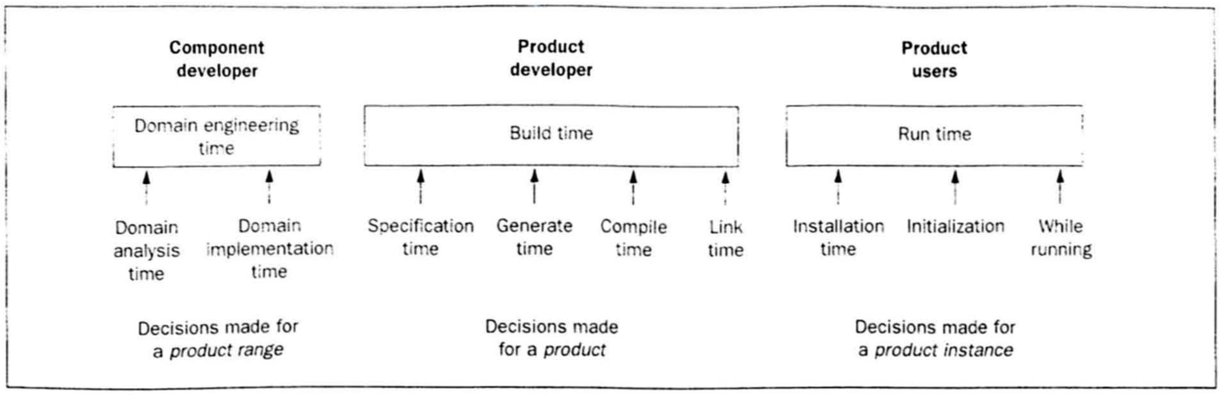

Figure 2. Decision times during the course of a product's life cycle. To have the maximum effect, decisions about a product must be made at the appropriate time and in a way that does not constrain other decision makers. The earlier in the life cycle a decision is made, the greater the impact it will have on the product.

Balancing Platforms with Commonalities and Variabilities. A platform effort begins with a collection of integrated components for building products, typically including "infrastructure" components such as hardware processors, operating systems, communication mechanisms, and databases. From there it evolves over time to incorporate various combinations of the following:

- A set of rules and constraints such as error handling and fault management strategies, programming styles, or communication protocols;
- An architectural framework, patterns, and standard interfaces between domains of an application area; and
- A variety of application components that can be customized for a range of products.

Some previous AT&T platform efforts have suffered from too many commonalities, leaving insufficient variabilities to support the necessary range of products that an organization needs. Components (and platforms) with few variabilities generally become more useless as they grow larger. As a consequence, some platform efforts remain stuck by providing only a collection of integrated components for building products, and are unable to extend into the other combinations listed above.

Combining the strengths of platforms (creating standards) with the strengths of domain engineering (creating components that can be customized) makes it possible to achieve a more optimal balance between commonalities and variabilities. Domain analysis efforts tied to such platforms can create multiuse components more effectively, with a higher percentage of common parts. One AT&T Business Unit, Global Public Networks (GPN), is organizing around platforms and other multiuse assets. It is therefore ideally poised to reap the benefits of this approach. The GPN Platform Group develops, maintains, and supports platform assets and is independent of the GPN product organizations. The platforms organization and products organization form a business partnership committed to multiuse. This relationship is more likely to survive if their technical concerns can be separated and the costs and benefits for each organization can be clearly identified.

Given that organizations can determine who will decide what, the next logical question is this: How can those decisions be distributed in a manner that will not create chaos?

Separation of Concerns

Separating concerns involves creating abstractions, determining which parts of them are variable and fixed, and then implementing the abstractions, as described in the sections that follow.

Abstractions. Abstraction is the major technique for making software more general, flexible, and reusable. Wegner states that abstraction and reusability are two sides of the same coin,[5] and Krueger states that abstraction is the essential feature in any reuse technique.[6] Raising the level of abstraction increases the potential for reuse. Successful abstractions, however, are difficult to create; they represent the major technical obstacle to successful reuse.

Abstraction is often portrayed as the ability to separate what is important from what is not important. Unimportant details are suppressed to highlight the essential features, permitting users of the abstraction to focus their attention on the relevant aspects of solving a problem. This portrayal, however, is "user-centric." An essential aspect of software abstraction is missing: the separation of concerns between development roles. Abstraction is not so much a matter of separating important details from unimportant details, but of separating a user's concerns from an implementor's concerns. So the words "important" and "unimportant" relate to the user of the abstraction, not the abstraction itself. To avoid this

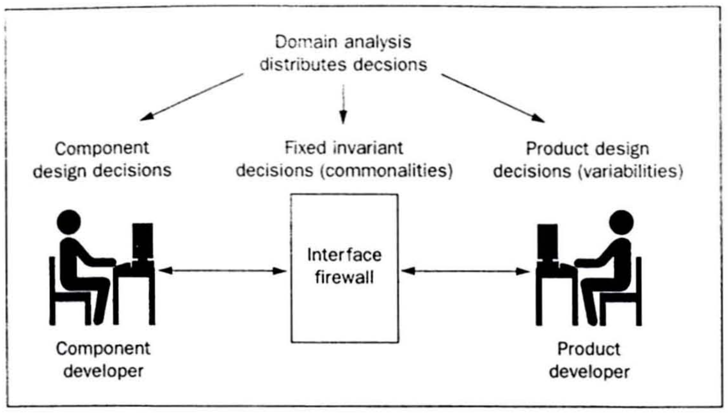
Domain analysis distributes decsions
Component design decisions
Fixed invariant decisions (commonalities)
Product design decisions (variabilities)
Interface firewall
Component developer
Product developer

Figure 3. Creating an interface involves several decision-making roles. Domain analysis determines which decisions are fixed, or invariant, and then distributes the variabilities to either the component developer or the product developer. Decisions are distributed in a way that allows each type of developer to make decisions independently of the other.

bias, software abstraction is viewed as more purely a separation of concerns between these two different development roles.

Every Abstraction Has Two Sides. A software abstraction is an interface that has two sides. The interface, now popularly called an application program interface (API), defines the invariant, or fixed, part of an abstraction, which neither side can change. The user, or product developer, side is the set of decisions that a user makes and expresses as a specification. The implementor, or component developer, side is the set of decisions that an implementor makes and expresses as an implementation. The abstraction separates these two sets of decisions and "hides" each set of decisions from the other. The implementor can make implementation decisions without affecting the user's decisions. The term "hidden" means that the user does not need to know or care about decisions made by the implementor. (In this sense the word "hidden" is a misnomer and should more properly be called "black box.") The same property is true in reverse. The implementor does not need to know or care about decisions made by the user. Both the user and the implementor do need to know about the interface itself, which represents the invariant rules, properties, and characteristics by which both user and implementor must abide (see Figure 3).

The creator of the abstraction —as opposed to the user or implementor—determines which decisions belong to which development roles. Each decision that is made by the user (or product developer) is a parameter of the abstraction. The process of creating an abstraction is thus sometimes called parameterization, because a designer takes something specific and makes it more general by adding a parameter to the abstraction.

Implementing Abstractions. Abstractions take many forms. Simple physical separation, such as the practice of isolating machine or operating system dependencies in separate code files, is often enough to achieve significant reuse. The user's responsibilities include providing the appropriate modifications to the separated code files so that a piece of software can be adapted to a new machine or operating system. The remaining files, which belong to the implementor, are not touched by the user.

Beyond simple physical separation are a wide variety of software techniques for achieving abstraction. The classic and oldest method is functional, or procedural, abstraction, sometimes called subroutines in assembly languages or FORTRAN. The parameters of the function are the parameters of the abstraction. In general, the more parameters a function has, the more abstract it becomes. A sort routine without any parameters sorts a particular array. A sort routine with an array parameter can sort any array, and is thus more reusable. A sort routine with an array parameter and a comparison operator not only sorts any array, but will also sort it in various ways.

Like functions, macros have parameters. Unlike functions, however, macros are expanded at compile time. Macros are typically used to implement build-time variabilities, whereas functions are used to implement run-time variabilities. The distinction between macros and functions is often overdone in programming languages, forcing programmers to decide at coding time whether to use a macro or function (C++ methods and the Ada programming language functions are exceptions). Partial parameterization is a technique that allows some subset of a function's parameters to be "executed/expanded" at compile time. Such a feature permits variabilities to be decided at either run time or build time for little cost.

Data abstraction and object-oriented (OO) techniques separate the use of data from how it is represented and manipulated. OO has been so successful that advocates sometimes equate it with reuse, although it is agreed that it is neither sufficient nor necessary to achieving software reuse.[7] Some have even argued that reuse is not an essential feature of OO.[8]

A second aspect of OO that contributes to reuse is *inheritance*, which allows common or general aspects of an object to be separated into a more general class that is shared by all its subclasses. Inheritance is an elegant approach to achieving reuse through physical isolation. It is not intrinsic to OO; rather, it is a general technique applicable in other situations as well. For example, the nmake[9] program uses directories of source files as if they were a hierarchy of classes.

Specification-driven techniques, such as application generators, little languages, application-oriented languages, and 4GLs, separate the concerns of product developers and component developers by providing a specification language to express the decisions made by a product developer.[10] The specification language is typically domain specific—that is, constructed for the specific needs of a particular domain—and may be textual, graphical, database, expert system, or spreadsheet. The component developer provides the tools and translators that take the specification information provided by the product developer and generate a variety of outputs, including code to implement the desired component.

Standards are playing an increasingly important role in the development of multiuse software. Standard operating systems (such as POSIX), communication mechanisms (such as CORBA), programming languages (such as ANSI C), and graphical user interfaces (such as the X Window System*) make it much easier to write reusable software. Standards are also abstractions that provide a well-defined interface between users and implementors of standardized services. External standards, such as those described above, provide the necessary incentive for organizations to adopt standards that lead to more efficient development of multiuse software. Standards make formerly unsafe dependencies a bit safer, at least at the time that the standard is adopted. When the current standard is superseded by a new one, previously safe dependencies once again become unsafe. Standards should therefore be adopted with care.

A developer might have the greatest set of components in the world, but still be unable to compose them into a system. This could be true for a variety of reasons, including overlapping functionality among components and interface and protocol mismatches. Architectural frameworks manage this problem by providing standard domains and interfaces between collections of components. An architectural framework separates the concerns among software architects, product developers, and software component developers by clearly indicating who is responsible for what. The software architects provide generalized frameworks and patterns[11] of interaction between components.

Abstraction, in all its myriad forms, is the chief technique for achieving separation of concerns. In the concept of information hiding,[12] internal design decisions of a component should not be visible to the outside world. This enables the component developer to freely change the design decisions without recoding any software external to the component. Correspondingly, designers should keep external design decisions out of a component so that product developers can freely change the product design decisions without recoding a component's internals. Information hiding works in both directions, as shown in Figure 4, protecting not only the component developer, but also the product developer.

The next section explores in depth an area in which separation of concerns plays a key role for enabling software reuse.

Interfaces—An Example of Separation of Concerns

Consider the following line of code in some arbitrary component *X*:

```
IpcSend(ErrorHandler, ErrorCode);
```

This line of code represents an interface between *X* and the `ErrorHandler` components. Developers often think of interfaces as single decisions, but they involve

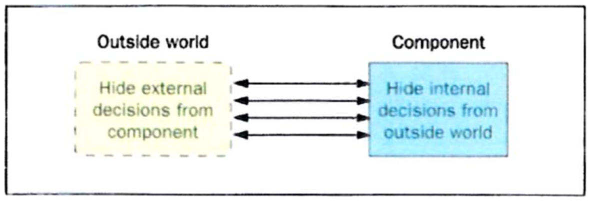
Outside world
Component
Hide external decisions from component
Hide internal decisions from outside world

Figure 4. Traditionally, information hiding has dealt with concealing the internal details of a component's construction from its users. This multiuser model also hides details of the world in which the component will be used from the component.

many decisions, most of them independent of one another. For example, the line of code shown on the previous page results from the following decisions:

How should we send the message?	`IpcSend`
Where should we send the message?	`ErrorHandler`
What is the type of message?	Type of `ErrorCode`
What is the content of the message?	Value of `ErrorCode`
When should we send the message?	Position in code

A single developer cannot make all these decisions alone. Many decision-makers enter the fray, including software architects, platform developers, component developers, and product developers. Most code embeds nearly all these decisions directly into the component.

Typically, information about all these interface decisions are tightly intertwined with component functionality. The variety of communication mechanisms in general use is staggering: function calls, object methods, remote procedure calls, interprocess communications, interprocessor communications, shared memory, global variables, files, and so on. Choosing one of these mechanisms and embedding it into the code of a component creates yet another dependency. Choosing the destination of a message and embedding it into component code creates yet another dependency. Whether the destination is the name of a function or the name of another component, a dependency will be created. It is no wonder that reusing such components is difficult. The component depends on decisions made about the whole system. A developer may want to reuse a component, but may be thwarted because it seems impossible to extract it from the web of product decisions built into it.

Traditional development processes, such as the waterfall or the spiral process, drive these decisions from the wrong end. In a traditional development process, the software architecture precedes component development. Software architects make decisions about the set of components and their interconnections. Architects frequently make early decisions about the components' physical location and the communication mechanisms for connecting them. This information is all readily available to the component developers, who frequently are also the architects. It is therefore not surprising that this knowledge finds its way into the internals of the components and results in the previous `IpcSend` example. Product architecture decisions become buried and invisibly intertwined in the software components.

A multiuse process must avoid this mixing of concerns in the development of multiuse components. A component developer should not know, and should not care, about the final product's configuration or communication mechanisms. It may be said that the performance, or even the behavior, of the whole product is affected by these choices. That is exactly the point. The product is affected; the components need not be so affected. Another important point is to recognize that one size may not fit all. There is no requirement that states that only one component can perform a given function. The size/speed tradeoff, in general, is incorrectly made by component developers. In hardware design many devices perform the same function using different technology. The tradeoffs are usually speed/cost/heat dissipation, and the product developer chooses the component technology that best fits the product need. So it should be with software components. This implies that it is feasible to think of families of software components from which product developers can choose.

To be more specific, an interface includes data, control, representation, mechanisms, and subtle assumptions about data relationships. An interface exposes details of the component to the world. In addition, and perhaps more troublesome, the interface exposes some

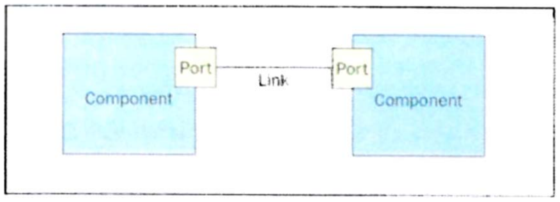

Figure 5. Ports specify information flow into and out of the component, while links specify the mechanistic details of information flow between components. Component developers are mainly concerned with ports, and component developers with links.

details of the world to the component. These issues are difficult because the interface is the place where component developer decisions meet product developer decisions. It is important to allow component developers to make their decisions in a way that does not prevent product developers from making their decisions later. Component developers need to have control over data representation and messages. Other decisions, such as communication mechanism and connection, must be delayed until a product developer chooses to make them.

The issue of data representation is important because traditional code is frequently organized around data representations rather than around data semantics. In particular, the representations are frequently fixed by assumptions of underlying transport mechanisms. This is a common path along which mechanism details find their way into components. Component developers must be free to choose appropriate representations without being constrained by product builder decisions that have not yet been made.

Ports and Links. The separation of concerns introduced earlier in this paper can be achieved using the concept of ports and links. A *port* simply describes what information flows into or out of a component. A *link* specifies which ports are connected together and what type of mechanism is used for the connection. Component developers specify ports and relationships between ports on the component. Product developers specify links. Most code written today does not distinguish between ports and links, and this leads to the intertwined interface code described earlier.

Rewriting the `IpcSend` example using ports and links results in something like:

```
ErrorPort(ErrorCode);
```

with the understanding that the product developer provides the details of what `ErrorPort` means. In this way, the component developer does not influence where the message is going, or how it is getting there—that is left to the product developer.

A module interconnection language (MIL) separates the concerns of component and product developers by providing languages for describing interfaces and product configurations that are different from the programming language used to build the components. MILs increase the range of possible configurations and the modularity and multiuse of software components. They do this by separating the language for programming individual components (programming in the small) from the language for configuring products from predefined components (programming in the large).[13] This separation allows the implementation language for each component to be chosen based on the needs of that component alone.

Typically, a MIL will provide language elements to describe the message part of the interface, often called the interface definition language (IDL), as well as language elements to describe the interconnections between interfaces. In some MILs, interfaces are called ports and interconnections are called links (see Figure 5).

MILs describe static relationships between components. It is probable that MILs will be extended to not only encode static topology, but also to encode sets of choices from which a topology can be selected. Adding constraint descriptions moves the MIL towards being a framework description, and frameworks, as this paper describes, are at the heart of multiuse platforms.

GPN is conducting a trial of MILs. One MIL, called the configuration and synthesis tool (CAST), has been used for building a number of multiuse components that can now be fitted to a variety of infrastructures and communication mechanisms. Tools such as CAST can generate almost all the glue code needed to make the desired interconnections between multiuse components and the rest of the system. Another tool, called the multiuse test environment (MTE) uses the MIL specification to enable interactive and batch testing of multiuse components or configurations of components. In addition, a MIL specification is a description of the product architecture, a rigorous specification that can be processed and checked for consistency even before the components have been constructed. This specification enables developers to detect classic interface mismatches early.

Summary and Final Note

This paper has described some of the steps of discovery leading to the construction of a model of multiuse based on reducing dependencies. In this model, strong separation of the concerns of component developers and product developers is essential to avoid placing interdependencies into components. Technologies such as CAST and MTE can also enable systematic, rather than opportunistic, reuse of component sets that are large enough to promote significant economic leverage. Defining platforms and standards can also help achieve this. Dependencies are further reduced by balancing standardization (commonalities) and parameterization (variabilities) of the component. A domain engineering process is critical to accomplishing this.

Technology can help achieve the necessary separation of concerns, but technology alone is not enough. The concerns of component and product builders are sufficiently disjointed to warrant organizing around them. From an organizational perspective, promoting multiuse is no more or less than establishing new internal business partnerships. It is this need for partners that makes multiuse easy to achieve technologically, but so difficult to carry out organizationally.

Acknowledgments

The authors would like to acknowledge David M. Weiss of AT&T Bell Laboratories for many constructive discussions during the work that led to this paper.

References

1. S. Wartik and R. Preito-Diaz, "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches," *International Journal on Software Engineering and Knowledge Engineering*, Vol. 2, No. 3, September 1992, pp. 403-431.
2. W. B. Frakes and S. Isoda, "Success Factors of Systematic Reuse," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 14-19.
3. R. Preito-Diaz and G. Arango, eds. *Domain Analysis and Software Systems Modeling*, IEEE CS Press, Los Alamitos, California, 1991.
4. F. P. Brooks, "No Silver Bullet," *Computer*, Vol. 20, No. 4, April 1987, pp. 10-19.
5. P. Wegner, "Varieties of reusability," reprinted in *Tutorial: Software Reusability*, edited by P. Freeman, IEEE Computer Society Press, 1987.
6. C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, Vol. 24, No. 2, June 1992, pp. 131-183.
7. M. L. Griss and M. Wosser, "Making reuse work at Hewlett-Packard," *IEEE Software*, Vol. 12, No. 1, January 1995, pp. 105-107.
8. T. Bryant and A. Evans, "OO oversold: Those objects of obscure desire," *Information and Software Technology*, Vol. 36, No. 1, January 1994, pp. 35-42.
9. S. Cichinski and G. S. Fowler, "Product Administration through Sable and Nmake," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 59-70.
10. J. C. Cleaveland and C. Kintala, "Tools for Building Application Generators," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 46-58.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
12. D. L. Parnas, "On Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, December 1972, pp. 1053-1058.
13. F. DeRemer and H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 2, June 1976, pp. 80-86.
14. W. B. Frakes and C. J. Fox, "Sixteen Questions About Software Reuse," *Communications of the ACM*, Vol. 38, No. 6, June 1995, pp. 75-87, 112.
15. J. C. Henderson, "Plunging into Strategic Partnerships: The Critical IS Connection," *Sloan School of Management Review*, Spring 1990.

(Manuscript approved February 1996)

J. Craig Cleaveland *is a former member of technical staff in the Global Communications Software Platform Development Department at AT&T Bell Laboratories in the Merrimack Valley Works, North Andover, Massachusetts. He pioneered the use of specification-driven tools and techniques, and is the author of two books on data types and formal methods for specifying programming languages. He received a B.S. in mathematics, and M.S. and Ph.D. degrees in computer science, all from the University of California at Los Angeles. Mr. Cleaveland, who joined AT&T in 1982, recently left the company.*

Janet A. Fertig *is head of the OC 48 Software and Systems Engineering Department at AT&T Network Systems in the Merrimack Valley Works, North Andover, Massachusetts. She is responsible for FT-2000 development and is interested in improving the effectiveness of software development. Ms. Fertig joined AT&T in 1980, after receiving a B.S. and M.S. from The University of Arizona, Tucson, and a Ph.D. from the University of Virginia, Charlottesville, all in systems engineering.*

George W. Newsome *is a member of technical staff in the Global Systems Engineering and Architecture Department at AT&T Network Systems in Holmdel, New Jersey. He works on network management, distributed computing, and ITU standards. Mr. Newsome received a B.Sc. in electrical engineering from the University College, London, UK. He is a chartered engineer and a member of the IEE. He joined AT&T in 1974.*