An Assessment and Analysis of Software Reuse

TED J. BIGGERSTAFF

Microelectronics and Computer Technology Corp.
Austin, Texas

1. Introduction . . . 1
 1.1 Hyperboles of Reuse . . . 2
 1.2 Key Factors Fostering Successful Reuse . . . 4
2. Software Reusability Successes . . . 10
 2.1 Fourth-Generation Languages (LSR to VLSR) . . . 10
 2.2 Application Generators (VLSR) . . . 13
 2.3 Forms Designer Systems (LSR to VLSR) . . . 15
 2.4 Interface Developer's Toolkits . . . 18
 2.5 The Software Factory (MSR to LSR, Process-Oriented Reuse) . . . 20
 2.6 Emerging Large-Scale Component Kits (LSR) . . . 22
 2.7 User-Oriented Information System (LSR to VLSR). . . . 23
 2.8 Application-Specific Reuse (LSR to VLSR) . . . 25
 2.9 Designer/Generators (LSR to VLSR) . . . 27
3. Examples of Reuse Implementation Technologies . . . 30
 3.1 Classification and Library Systems . . . 30
 3.2 CASE Tools . . . 31
 3.3 Object-Oriented Programming Systems . . . 33
4. Effects of Key Factors . . . 38
 4.1 Relationships among the Reuse Factors . . . 38
 4.2 A Quantitative Model of the Relative Amount of Integration Code . . . 41
5. Futures and Conclusions . . . 53
 5.1 Futures . . . 53
 5.2 Conclusions . . . 54
6. References . . . 54

1. Introduction

Software reusability (Biggerstaff and Perlis, 1984; Biggerstaff and Richter, 1987; Freeman, 1987; Tracz, 1987, 1988; Biggerstaff and Perlis, 1989; Weide *et al.*, 1991) is not a "silver bullet"* (Brooks, 1987), but is an approach that under special circumstances can produce an order of magnitude improvement in software productivity and quality, and under more common

* The phrase "silver bullet" is jargon that refers to a panacea for software development.

circumstances can produce less spectacular but nevertheless significant improvements in both. This chapter will examine several aspects of reuse: (1) reuse hyperboles that lead to false expectations, (2) examples of reuse successes, (3) the factors that make these examples successful, (4) the relationships among these factors, (5) in particular, the relationship between reuse technologies and their potential for productivity and quality improvement, and (6) the quantitative relationship between the key factors and the resultant reuse benefits.

1.1 Hyperboles of Reuse

After listening to a series of speakers, each promising additive cost decreases that were summing suspiciously close to 100%, one wag was heard to comment, "If this keeps up, pretty soon our internal software development activities will be so efficient that they will start returning a profit." As in this story, software reusability hyperboles often strain credulity. Unfortunately, software reusability hyperbole is more seductive than software reusability reality.

There are several major reuse hyperboles that reflect some measure of truth but unfortunately overstate the profit of reuse or understate the required qualifications and constraints.

- *Reuse technology* is the most important factor to success. This is an aspect of the silver bullet attitude and is typified by statements like: "If I choose Ada, or Object-Oriented programming or an application generator then all other factors are second- and third-order terms in the equation that defines the expected improvement. Success is assured." However, this is seldom completely true. While the technology can have very high impact (as with application generators for example), it is quite sensitive to other factors such as the narrowness of the application domain, the degree to which the domain is understood, the rate of technology change within the domain, the cultural attitude and policies of the development organizations, and so forth. Yes, the technology is important but it is not always primary nor even a completely independent factor.
- Reuse can be applied everywhere to great benefit. This is another aspect of the silver bullet attitude that one can apply reuse to any problem or application domain with the same expectation of high success. The reality is that narrow, well-understood application domains with slowly changing technologies and standardized architectures are the most likely to provide a context where reuse can be highly successful. For

example, well-understood domains like management information systems (MIS) and business applications, user interfaces, narrowly defined product lines, numerical computation, etc. all, to a greater or lesser extent, have these qualities and reuse has flourished in these environments. Reuse has failed in new, poorly understood domains.
- Reuse is a hunter/gatherer activity. Making a successful reuse system is largely an intellectual activity of finding the right domain, the right domain standards, the infrastructure, and the right technical culture. It is not simply a matter of going out into the field and gathering up components left and right. Casually assembled libraries seldom are the basis of a high payoff reuse system. Successful reuse systems are crafted to accomplish a set of well and narrowly defined company or organizational goals. Too general a set of goals (e.g., we need a reuse system) or too general a domain (e.g., we need components that support all of our functional needs) usually lead to a low payoff. The hidden truth in this attitude is that populating a reuse library is largely fieldwork and that the "gold" is in the domain. But the success comes through problem driven harvesting, establishing domain standards to enhance component interconnectability and careful adaptation of the harvested components to those interconnection standards.
- We can have reuse without changing our process. Reuse is sensitive to many cultural, policy and environmental factors. An anti-reuse attitude within an organization, a process that is inconsistent with reuse or a weak, unsupportive infrastructure (software and process) can doom a potentially successful reuse effort.

Given that we reject these hyperboles, let us look at the reality of software reuse. In the broadest sense, software reuse is the formalization and recording of engineering solutions so that they can be used again on similar software developments with very little change. Hence, in one sense, the software reuse process institutionalizes the natural process of technology evolution. Consider the evolution of commercial software products. Successful companies often maximize their competitiveness by focusing on product niches where they can build up their technological expertise and thereby their product sets and markets, in an evolutionary fashion. For example, over a period of years, a company might evolve a line editor into a screen editor and then evolve that into a word processor and finally evolve that into a desktop publishing system. Each generation in such an evolution exploits elements of the previous generations to create new products and thereby build new markets. In an informal sense, such a company is practicing reuse within a product niche. The companies that formalize and institutionalize this process are truly practicing reuse. Since this definition of reuse is independent of any specific enabling technology (e.g., reuse libraries or application generators), it allows us to take a very broad view of reuse, both in the range of potential component types that can be reused (e.g., designs, code, process, know-how, etc.) as well as in the range of technologies that can be used to implement reuse.

The success of a reuse strategy depends on many factors, some of them technical and some of them managerial. While we will attempt to point out management factors that foster or impede reuse, we will largely focus on the technology of reuse.

In the next subsection, we hypothesize a number of factors or properties that we believe foster successful software reuse. Then in the following sections of the chapter, we will examine several reuse successes and the role that these factors played in those successes. Finally, we attempt to build a qualitative model that describes the interrelationship among the factors and a quantitative model that describes the effects of two of the key independent technology factors on the payoff of software reuse. In the end, we hope to leave the reader with a good sense of the kinds of reuse approaches and technologies that will lead to success and those that will not.

1.2 Key Factors Fostering Successful Reuse

Some of the key factors that foster successful reuse are:

- Narrow domains
- Well-understood domains/architectures
- Slowly changing domain technology
- Intercomponent standards
- Economies of scale in market (opportunities for reuse)
- Economies of scale in technologies (component scale)
- Infrastructure support (process and tools)
- Reuse implementation technology

Narrow domains: The breadth of the target domain is the one factor that stands out above all others in its effect on productivity and quality improvement. Typically, if the target domain is so broad that it spans a number of application areas (often called *horizontal reuse*) the overall payoff of reuse for any given application development is significantly smaller than if the target domain is quite narrow (often called *vertical reuse*). The breadth of the target domain is largely discretionary, but there is a degree to which the reuse implementation technology may constrain the domain breadth. There is a range of implementation technologies, with *broad-spectrum* technologies at one end and *narrow-spectrum* technologies at the other. Broad-spectrum technologies (e.g., libraries of objects or functions) impose few or no constraints on the breadth of the target domain. However, narrow-spectrum technologies, because of their intimate relationship with specific domain niches, do constrain the breadth of the target domain, and most often constrain target domains quite narrowly. In general, narrow-spectrum implementation technologies incorporate specialized application domain knowledge that amplifies their productivity and quality improvements within some specific but narrow domain. As an example, fourth-generation languages (4GLs) assume an application model that significantly improves the software developer's ability to build MIS applications but is of no help in other domains such as avionics.

Even though there is a restrictive relationship only at one end of the spectrum (between narrow target domains and narrow implementation technologies), in practice there seems to be a correlation between both ends of the spectrum. Not only do narrow-spectrum technologies, perforce, correspond to narrow target domains but broad-spectrum technologies often (but not always) correspond to broader domains.

The key effect of domain breadth is the potential productivity and quality improvement possible through reuse. Reuse within very narrow domains provides very high leverage on productivity and quality for applications (or portions of applications) that fall within the domain but provides little or no leverage for applications (or portions of applications) that fall outside the domain. For example, an application generator might be used to build MIS applications and it would give one very high leverage on the data management portion of the application but it would not help at all in the development of the rest of the application. Luckily, MIS applications are heavily oriented toward data management and therefore, such reuse technologies can have a significant overall impact on MIS applications.

Broad-spectrum technologies, on the other hand, show much less productivity and quality improvement on each individual application development but they affect a much broader class of applications. Generally speaking, the broad-spectrum technologies we are going to consider can be applied to virtually any class of application development.

In the succeeding sections, we will often use the general terms *narrow-spectrum reuse* and *broad-spectrum reuse* to indicate the breadth of the domain without any specific indication of the nature of the implementation technology being used. If the breadth of the implementation technology is important to the point, we will make that clear either explicitly or from context.

Well-understood domains/architectures: The second key factor affecting the potential for reuse success is the level of understanding of problem and application domains, and the prototypical application architectures used within those domains. Well-understood domains and architectures foster successful reuse approaches and poorly understood domains and architectures almost assure failure. Why is this? Often as a domain becomes better and better understood, a few basic, useful, and successful application architectures evolve within the domain. Reuse systems can exploit this by reusing these well-understood architectural structures so that the software developer does not have to recreate or invent them from scratch for each new application being developed. However, if such application architectures have not yet evolved or are not known by the implementing organization, it is unlikely they will be discovered by a reuse implementation project.

The fact that the problem domains in which narrow-spectrum reuse has been successful are well-understood domains is not coincidental. In fact, it is a requirement of a narrow-spectrum reuse technology. This observation points up a guideline for companies that intend to build a narrow spectrum reuse system to support application development.

> To successfully develop a narrow-spectrum reuse technology, say an application generator or a domain-specific reuse library, the developer must thoroughly understand the problem and application domain and its prototypical architectures in great detail before embarking on the development of a reuse system for that domain.

There is a three-system rule of thumb—if one has not built at least three applications of the kind he or she would like to support with a narrow-spectrum technology, he or she should not expect to create a program generator or a reuse system or any other narrow-spectrum technology that will help build the next application system. It will not happen. One must understand the domain and the prototypical architectures thoroughly before he or she can create a narrow-spectrum reuse technology. Hence, the biggest, hardest, and most critical part of creating a narrow-spectrum technology is the understanding of the domain and its prototypical architectures.

Slowly changing domain technology: Not only must one understand the domain but the domain needs to be a slowly changing one if it is to lend itself to reuse technology. For example, the domain of numerical computation is one in which the underlying technology (mathematics) changes very little over time. Certainly, new algorithms with new properties are invented from time to time (e.g., algorithms allowing high levels of parallel computation) but these are infrequent and the existing algorithms are largely constant. Thus, if an organization makes a capital investment in a reuse library or an application generator for such domains, they can amortize that investment over many years. Rapidly changing domains, on the other hand, do not allow such long periods of productive use and, therefore, do not offer as profitable a return on the initial investment.

Intercomponent standards: The next factor is the existence of intercomponent standards. That is, just like hardware chips plug together because there are interchip standards, software components, and especially narrow-spectrum technology components plug together because there are analogous intercomponent standards. These standards arise out of an understanding of the problem domains and the prototypical architectures. The narrower the domain, the narrower and more detailed the intercomponent standards. In very broad domains, these standards deal with general interfaces and data (e.g., the format of strings in a string package), whereas in a narrow domain the standards are far more narrowly focused on the elements of that domain (e.g., in an "input forms" domain, the standards might specify the basic data building blocks such as field, label, data type, data presentation form, and so forth).

This factor suggests that certain narrow spectrum reuse technology strategies will not work well. For example, if one intends to build a library of reusable software components, the strategy of creating a library and then filling it with uncoordinated software components, will lead to a vast wasteland of components that do not fit together very well. Consequently, the productivity improvement will be low because the cost to adapt the components is high. The analogy with hardware manufacturing holds here. If two software components (or chips) are not designed to use the same kinds of interfaces and data (signals), extra effort is required to build interface software (hardware) to tie them together. This reduces that payoff gained by reuse and also tends to clutter the design with Rube Goldberg patches that reduce the resulting application's maintainability and limit its ability to evolve over time.

Economies of scale in market: Another important factor is the economies of scale in the "market," where we are using the term market in the broadest sense of the word and intend to include the idea that the total coalition of users of a component, regardless of the means by which they acquire it, is the market for that component. Thus, economies of scale in the market means that any reuse technology should be driven by a large demand or need. One should be able to identify many opportunities to apply the reuse technology to justify its development (or purchase) and maintenance. If you are only going to develop one or two applications, it seldom pays to develop (or purchase) a reuse technology for the target application. This is not to say that informal, ad hoc or opportunistic reuse, which is not organizationally formalized, should not be exploited. The point is that if an institutionalized reuse technology costs a company a lot to develop and maintain, it should return a lot more in savings to that company. One way to gauge that return beforehand is to consider the opportunities for reuse.

Economies of scale in technologies: There are also economies of scale in the technologies themselves, in the sense that, the larger the prefabricated component that is used by the reuse technology, the greater the productivity improvement for each use. And it is this increase in size of components that tends to force the narrowing of the technology domain. Thus, the size of the prefabricated component, the narrowness of the application domain, and the potential productivity improvement are all positively correlated.

Because the scale of the components is so important and the fact that scale correlates to other important properties of reuse technologies, we introduce some broad terminology that draws on the hardware component analogy. *Small-scale* components are defined to be from 10 to 100 lines of code, i.e., $O(10^1)$ LOC; *medium-scale* components are those from 100 to 1000 lines, i.e., $O(10^2)$ LOC; *large-scale* from 1000 to 10,000 lines, i.e., $O(10^3)$ LOC; *very large-scale* from 10,000 to 100,000 lines, i.e., $O(10^4)$ LOC; and *hyper-scale* above 100,000 lines, i.e., greater than $O(10^5)$ LOC. The sizes that we choose are somewhat arbitrary and flexible because we are most interested in the relative properties of the reuse systems that rely on the different scales of components. Therefore, the numbers should not be taken too literally but rather should provide a loose categorization of component sizes.

Carrying the hardware analogy further, we use the term SSR (small-scale reuse) to refer to those technologies that tend to use small-scale components on the average. SSR is motivated by the hardware term SSI (small-scale integration). Similarly, MSR, LSR, VLSR, and HSR are medium-scale, large-scale, very large-scale and hyper-scale reuse technologies. While reuse technologies are not, strictly speaking, limited to a particular scale, they seem to most easily apply to a characteristic scale range. For example, libraries of functions tend toward small scale and medium scale not because it is impossible to build large and very large function-based components, but rather because of the lack of formal support for large-scale design structures (e.g., objects or frameworks) in functionally based programming languages. Any such large-scale design structure falls outside of the functional language formalism and must be manually enforced. Experience has shown that manual enforcement tends not to be very successful. It is generally easier to use other reuse implementation technologies (e.g., true object-based languages) that provide formal mechanisms to enforce and manage these larger-scale structures.

Infrastructure support: Another important factor is an organization's infrastructure. Most reuse technologies (and especially the narrow spectrum technologies) pay off best when they are coordinated with an existing, well-defined, and mature software development infrastructure (process). For example, an organization that uses computer-aided software engineering (CASE) tools is better positioned to exploit the reuse of design information than one that does not. CASE tools provide a (partially) formal notation for capturing such designs. And if an organization is already trained and using CASE tools, the additional effort to integrate a library of reusable designs into the process is significantly less than it would be otherwise.

Reuse implementation technologies: One factor that can effect the degree of success of a reuse approach is the implementation or enabling technology that one chooses. For many narrow spectrum approaches to reuse, the technology is intimately tied to the approach and it makes more sense to discuss these technologies in the context of the discussions of the specific approaches. We will do this in the following section. On the other hand, broad-spectrum implementation technologies are not tied to any specific reuse approach, even though they are quite often used for broad-spectrum reuse, and so we will mention a few instances of these technologies here and discuss their values.

- Libraries: Library technology is not a primary success factor but its value lies largely in establishing a concrete process infrastructure that fosters reuse by its existence more than by its functionality. If an organization's first response to a reuse initiative is to build a library system, then they probably have not yet thought enough about the other more important factors.
- Classification systems: The main value of classification systems is that they force an organization to understand the problem and application domain.
- CASE tools: Their value lies in establishing a representation system for dealing with designs and thereby including reusable components that are more abstract (and therefore, more widely reusable) than code.
- Object-oriented programming languages: Their main value is in the perspicuity of the representation and its tendency to foster larger and more abstract reusable components (i.e., classes and frameworks) than

in earlier languages (i.e., functions). Further, the object-oriented representation tends to lead to clearer, more elegant and more compact designs.

In summary, reuse success is not a result of one technology or one process model or one culture. It is a result of many different mixtures of technologies, process models, and cultures. We can be guided by a few general principles that point in the direction of success and warn us away from surefire failures, but in the end, the details of success are defined by hard technical analysis and a strong focus on the application and problem domains. I suspect that there is an 80/20 rule here—the domain has an 80% effect and all of the rest has a 20% effect.

2. Software Reusability Successes

Now let us consider some cases of successful reuse and analyze them in the light of these success factors.

2.1 Fourth-Generation Languages (LSR to VLSR)

Among of the earliest rapid software development technologies to appear and ones that can be bought of the shelf today are fourth-generation languages (4GLs) (Gregory and Wojtkowski, 1990; Martin, 1985; Martin and Leben, 1986a, b). These are quite narrow technologies that apply most specifically to the domain of MIS and business applications. The entities that are being reused in these two cases are the abstract architectural structures (i.e., design components) of MIS applications.

The typical 4GL system provides the end user with some kind of high-level capability for database management. For example, a high-level query from the end-user is often translated into an application database transaction that generates a report. The report may be a business form, a text-based report, a graph, a chart, or a mixture of these elements (see Fig. 1).

4GLs are typically very high-level languages that allow you to talk to the database system without all of the overhead that you would have to use if you were writing an equivalent COBOL program. In a COBOL program you might have to allocate memory and buffers to handle the results from the query. You might have to open the database, initiate the search, and so forth. In contrast, 4GL languages typically do all of those things for you. They provide a language that requires you to talk only about the essential database operations. For example, Fig. 2 shows a sequential query language (SQL) query that selects a part number from a table of all parts, such that the weight of the associated part is less than 700 pounds.

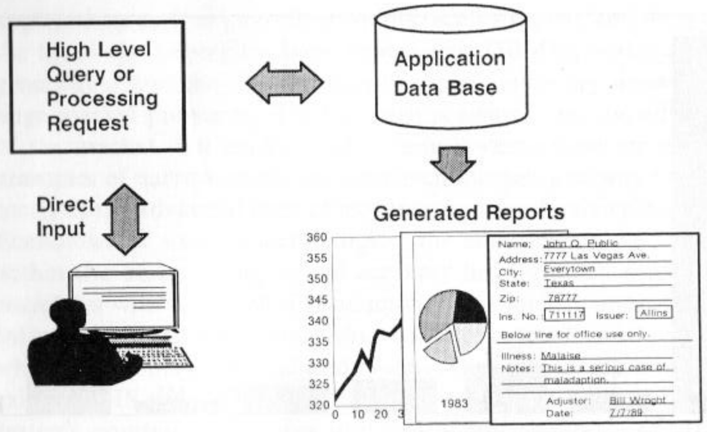


FIG. 1. Fourth-generation languages (4GLs).

```
SELECT Part#
    FROM PART
    WHERE PartWeight < 700
```

FIG. 2. Typical 4GL Query (in SQL).

These languages provide you with quite an increase in productivity because of the reduction in the amount of information necessary to perform an operation. Figure 3 illustrates this reduction by comparing the number of bytes required to express a benchmark query in COBOL against the number of bytes required in various 4GLs (Matos and Jalics, 1989). Of course, the exact number of bytes needed to express any given query will vary but the relative sizes represented in this chart are pretty typical. The typical proportions are from 11 to 22 times more characters to express a query in COBOL than in a 4GL. Since the number of bytes required is directly proportional to the amount of work required to create the query, it is an order of magnitude easier to perform database queries and generate reports in 4GLs than in COBOL or other high-level languages.

Now let us look at this example of reuse against the properties that we proposed:

- Narrow domains: clearly, the domain is quite narrow in that it applies to the data management and user interface aspects of MIS and business systems in general. Importantly, this domain is a large part of each such application and therefore, the overall payoff for each application

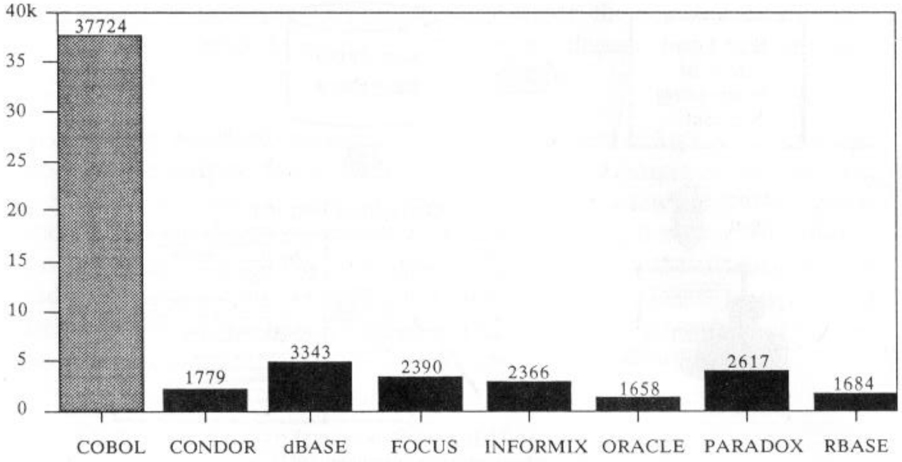


FIG. 3. Comparison of source code volumes.

can be quite large. Over all business applications, the variance is quite large but one can expect the reduction in effort to range typically between 25% and 90%. It is not atypical for 90% or more of the application to be directly handled by the 4GL, thereby allowing the application to be created for one tenth of the cost of building the system with a conventional high-level language. Defects are similarly reduced.

- Well-understood domains/architectures: the data management and user interface architectures within this application domain have been increasingly better understood and standardized for the last 25–35 years, and consequently they have evolved into standard subsystems that are common to many if not most of the application programs in the domain. DBMSs (database management systems) and 4GLs are two of the concrete manifestations of that ongoing understanding and standardization process.
- Slowly changing domain technology: the underlying hardware and software technologies have changed slowly enough that they can be largely hidden by lower-level system layers, e.g., DBMSs.
- Intercomponent standards: the DBMSs form a set of hardware-hiding standards and the 4GLs impose an additional set of application logic-hiding standards. If we looked inside of various 4GL systems we would likely find other finer-grained standards that allow the subsystems of the 4GL to fit together easily.
- Economies of scale in market: the MIS and business system market is probably one of the largest application markets that exist today. Virtually every business of any size at all requires some set of computer applications such as payroll, accounts receivable, etc. and these are only the tip of the iceberg for large companies. DBMSs, 4GLs, application generators, and the like are simply the evolutionary answer to these huge market pressures. It is the huge pressures and the advanced age of the market that explains why these systems were among the first examples of narrow-spectrum reuse technologies and why they are currently at the advanced level of maturity and productivity improvement.
- Economies of scale in technologies: the components, i.e., subsystems within the 4GLs, being reused are very large-grained pieces and this correlates with the level of productivity and quality improvement.
- Infrastructure support: the infrastructure support is largely in place when these tools are introduced because the tools are built to fit into an existing MIS shop. They are fitted to the kinds of hardware, operating systems, computer languages, and typical process models of MIS shops. This makes their adoption quite easy.

2.2 Application Generators (VLSR)

Application generators form another class of reuse technology that is similar to the 4GL class but varies in the following ways:

1. Generators are typically used to generate application systems that will be used many times whereas 4GLs generate programs or queries that are often one-of-a-kind.
2. Application generators tend to be more narrow than 4GLs, often focusing on a narrow application family, whereas 4GLs tend to focus on a broader application domain containing many application families. For example, compiler builders, like YACC and Lex, are application generators for building applications in the parser and lexical analyzer families.

While it is a research prototype rather than a production system, the GENESIS system (Batory, 1988; Batory *et al.*, 1989) is a good example of an application generator that is pushing the productivity and quality improvement limits. GENESIS (see Fig. 4) is for DBMSs what compiler builders are for compilers. GENESIS *generates* database management systems. While many application generators can be purchased off the shelf today, GENESIS is still in its research phase but, nevertheless, is interesting because it illustrates how far generator technology can be pushed. How does GENESIS work?

The GENESIS user specifies a set of requirements that characterize the kind of DBMS desired. In a typical specification, the user specifies (1) a data

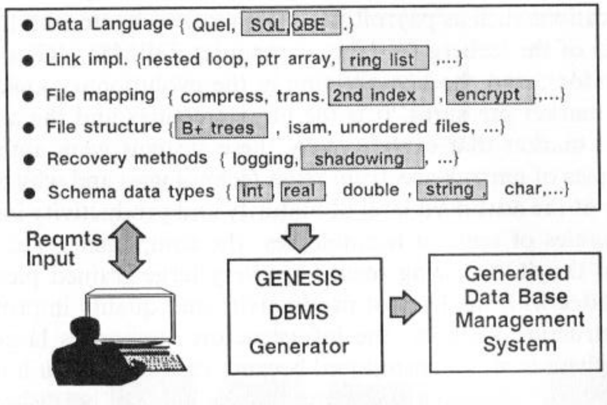

• Data Language { Quel, SQL, QBE ,}
• Link impl. {nested loop, ptr array, ring list ,...}
• File mapping { compress, trans. 2nd index , encrypt ,...}
• File structure { B+ trees , isam, unordered files, ...}
• Recovery methods { logging, shadowing , ...}
• Schema data types { int , real double , string , char,...}
Reqmts Input
GENESIS DBMS Generator
Generated Data Base Management System

FIG. 4. Genesis application generator system.

language, e.g., Sequel and/or QBE; (2) the database link implementation, e.g., a ring list; (3) the file mapping, e.g., secondary indexes and encryption; (4) the file structures, e.g., B-trees, ISAM, unordered files, etc.; (5) the recovery methods, for example, logging, shadowing, etc.; and (6) the data type schemas, e.g., *ints* and *reals* and *strings*, etc. GENESIS then generates a database management system to those specifications. So if one wants to generate a DBMS that has exactly the same functionality as Ingress, that can be done by specifying the particular requirements of Ingress.

Typically, application generators provide productivity that is one or two orders of magnitude better than hand coding. While the only problem that GENESIS solves is the creation of database management systems, it is highly productive at this. I can generate a 40,000-plus line DBMS in about 30 minutes of work. So, application generators can give you *very* high productivity for *very* narrow domains. What is more, the quality of the code is *very* high. Typically, a bug that is due to the generator turns up about as frequently as bugs in a good, mature compiler.

Now let us look at this example of reuse against the factors:

- Narrow domains: this is one of the narrowest domains—DBMS—and the productivity and quality improvements over hand coding from scratch are consequently exceptionally high. In this case, we can experience several orders of magnitude improvement in productivity and quality—40,000 lines of debugged code in less than 1 hour of work by using GENESIS versus four or five people for several years to build an equivalent target system from scratch.
- Well-understood domains/architectures: DBMSs have the advantage that hundreds of researchers have been working for over 20 years to work out the theoretical basis of these systems. That background work has turned an impossible task (i.e., building a GENESIS system 20 years ago) into one that is just hard (i.e., building GENESIS today).
- Slowly changing domain technology: DBMS technologies are relatively stable overtime although they do seem to go through periodic technology shifts such as moving from hierarchical to relational and more recently to Object-Oriented DBMSs. However, within any given DBMS model, the change is relatively slow and within the older technologies (hierarchical and relational) fundamental advances now seem almost nonexistent.
- Intercomponent standards: GENESIS would be impossible without the development of a set of well-defined module interconnection standards that allow the system to plug together various modules with different operational properties but having the same standardized connection interface.
- Economies of scale in market: since GENESIS is a research project, it is not yet clear whether or not there really are economies of scale for GENESIS per se. Nevertheless, the typical application generator arises because of a "market pressure" (sometimes within a single company) for the facility.
- Economies of scale in technologies: the prefabricated GENESIS components are typically several thousand lines of (parameterized) code and if one considers the additional generated code, GENESIS is in the VLSR technology range.
- Infrastructure support: as with 4GLs, application generators are fitted to the kinds of hardware, operating systems, computer languages, and typical process models that already exist within MIS shops, making their adoption quite easy.

2.3 Forms Designer Systems (LSR to VLSR)

Another kind of reuse technology is forms designers, which are variously called screen painters or designers. These systems attack the problem of developing a forms-based user interface for a business application. Most businesses are rife with forms, e.g., invoice forms, that are used as an integral part of the business operation. Consequently, they are ubiquitous in many business applications and are therefore a prime candidate for a reuse technology. Forms designers and screen painters allow one to quickly generate a forms-based user interface by using a set of predefined building block components. Figure 5 presents a conceptual overview of forms designers.

A forms designer's form representation is visually like a paper-based form and it is used by the application program to request input from, and present

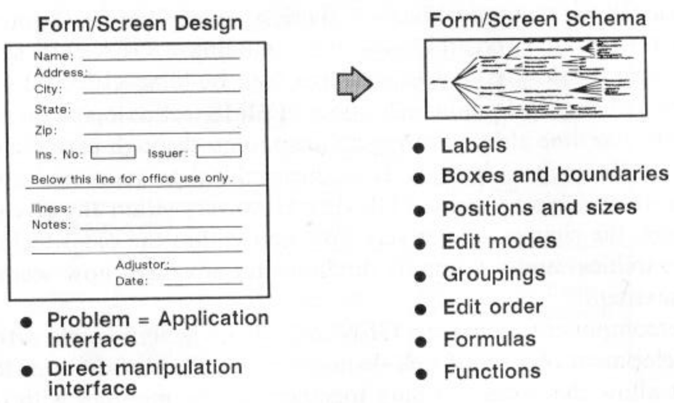


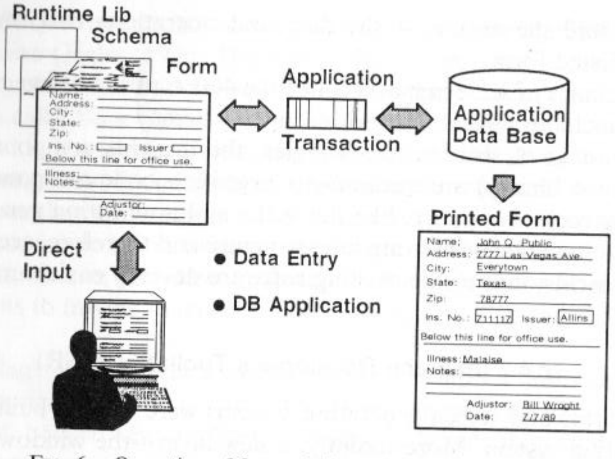
FIG. 5. Creation of a form-based application interface.

output to the end-user. Users create forms by a direct manipulation interface permitting them to draw the form on the screen, complete with labels, fields, borders, and so forth, exactly the way they want it to look to the application user. Then, the form design is turned into an internal schema that specifies the form's labels, boxes, boundaries, colors, fields, and their positions. The schema may also specify editing modes. For example, numbers may be the only valid input for a date or price field. It specifies the edit order of the fields, i.e., the order in which the cursor sequences through the fields as the end user presses the tab or return key. The schema may also allow formulas that functionally relate certain fields to other fields. For example, the gross pay field in a work record form could be calculated as the product of the field containing the salary per hour times the field containing the number of hours worked.

Once these forms are created, they are used by an application program as a way to request input from or present output to the user as shown in Fig. 6. In the use phase, the application program loads the schema and a run-time library that manages the form's data and user interaction. The run-time library handles the presentation of the form to the user, the interaction with the user, the editing of the fields, and the transformation of the form's data into some kind of transaction or internal record that can be used by the application program.

Once the data are entered into the form by the end-user, the form's data fields are typically converted into a data record or database transaction, which may produce a variety of business side effects e.g., inventory being ordered, an invoice generated, etc.

The properties of this domain are:



Name: John Q. Public
Address: 7777 Las Vegas Ave.
City: Everytown
State: Texas
Zip: 78777
Ins. No.: Z11117 Issuer: Allins
Below this line for office use.
Illness: Malaise
Notes:
Adjustor: Bill Wroght
Date: 7/7/89

FIG. 6. Operation of form within an application program.

- Narrow domains: this domain—forms-based user interfaces—is quite narrow but constitutes a smaller portion of the application (i.e., only the user interface portion) than 4GLs typically do. Therefore, it leads to a somewhat smaller but by no means inconsequential developmental cost and defect reduction. Depending on the overall complexity of the application, one might expect a typical developmental cost and defect reduction to be in the 5–25% range. When a forms designer is incorporated into a 4GL, which is common, the overall improvement jumps significantly and an order of magnitude decrease in total developmental cost and number of defects is common.
- Well-understood domains/architectures: like 4GLs, this technology has been evolving for years and the methods and architectures are well and widely known.
- Slowly changing domain technology: this technology has been largely stable for years with only minor evolutionary changes arising from advances in monitor technology (e.g., high resolution, bitmapped, color, etc.) and the associated interface software (e.g., graphical user interfaces (GUI) and windowing interfaces). Much of this evolutionary change can be and has been isolated by software layers within the screen designers that abstract out the essential properties of the monitor hardware and the interface software.
- Intercomponent standards: the screen designer tool establishes a wide range of standards including what types of data can be accommodated in fields, how the field and label information is encoded for the run-time routines, what kinds of editing operations are available to the

user, and the nature of the data and operations that result from a completed form.

- Economies of scale in market: like the 4GLs, this is a huge marketplace that includes most MIS and business systems.
- Economies of scale in technologies: the reusable components (in the run-time library) are medium- to large-scale code components.
- Infrastructure support: like the 4GLs and application generators, this technology fits the existing infrastructure and therefore, accommodates easy inclusion into the existing software develop environment.

2.4 Interface Developer's Toolkits (VLSR)

Early forms and screen generation systems were usually built directly on the operating system. More recently, a new layer—the window manager—has been introduced and applications are now built on top of these windowing systems. This has given rise to another kind of reuse facility—the interface developer's toolkit. Interface toolkits are analogous to forms designers and screen painters but address a broader range of applications. More to the point, they allow one to build applications with GUIs. Interface toolkits provide libraries of facilities out of which one can build a rich variety of user interfaces, including interfaces for form designers and screen painters. Figure 7 presents a conceptual overview of the interface developer's toolkit.

Like the form and screen designers, interface toolkits are designed for developing user interfaces. They tend to be built on top of whatever standard window interface is supplied with the operating system. They also provide

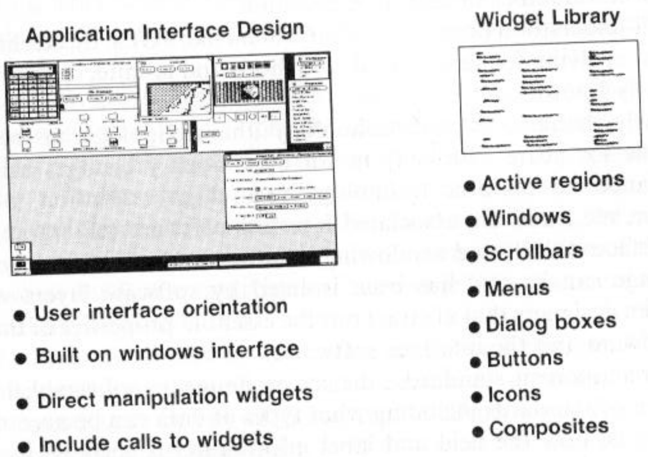

Application Interface Design
Widget Library
• User interface orientation
• Built on windows interface
• Direct manipulation widgets
• Include calls to widgets
• Active regions
• Windows
• Scrollbars
• Menus
• Dialog boxes
• Buttons
• Icons
• Composites

FIG. 7. Interface developer's toolkit.

a number of direct manipulation *widgets* (Nye and O'Reilly, 1990)—to use the X windows (Heler, 1990; Hinckley, 1989; Nye, 1988; Scheifler *et al.*, 1988; Young, 1989) terminology—that can be included in the application interface via calls. So a typical widget library would provide active regions, a region sensitive to mouse events; window objects with all of the necessary window management functionality; scrollbar objects so that windows can show canvases that are actually much larger than the windows themselves and allow the user to scroll to the unseen portions of the canvas; menus of various kinds, such as pull-down, pop-up, sticky, etc.; dialog boxes for data input; buttons to invoke functions; icons to represent suspended programs; etc.

An advantage of an interface toolkit is that it ensures a uniform look and feel for all applications built with it. This allows the end-user to have a pretty good idea of how a new application operates based only on his or her previous experience with other applications.

One of the first uses of this idea was in Xerox PARC's *Alto* personal computer system (Xerox, 1979). Later, the same idea was used in the Xerox Star Workstation (Xerox, 1981). The first widespread, commercialization was in the Apple computer's MacApt interface builder's kit. More recently, such toolkits have been built for various windowing systems. X-Windows appears to be the emerging window standard for the Unix/workstation world and several X-based toolkits are available, such as Interviews, Motif, and Openwindows. One can also purchase similar toolkits for other operating systems and machines types. Toolkits for the PC (personal computer) market are built on top of Microsoft Windows (TM Microsoft), OS/2 Presentation Manager (TM IBM), etc. The market for interface toolkits is growing rapidly at this time.

The properties of this approach are much the same as the forms designers with a few differences. First, because the applications using this approach tend to cover a broader range of domains than just MIS or business systems, the user interface is typically a much smaller part of the overall target application program and therefore, while the payoff in absolute terms is large, the decrease in developmental costs and defect levels is often proportionally smaller over the whole developmental effort than with 4GLs and screen designers.

While this technology is reasonably well understood and standards are being formalized, it is not as mature as the forms interface, and therefore it is still evolving new subdomains. One subdomain is the recent emergence of the GUI generator, the analogue of the forms designers. GUI generators, which recently appeared on PCs and are just beginning to appear on workstations, allow one to design the screen interface through a direct manipulation editor (Sun Microsystems Corporation, 1990). They allow one to graphically construct the interface design using high-level objects like menus, panels, dialog boxes, and events. These tools are more complicated than forms designers because so much more of the application code must come from and be customized to the application by the software engineer rather than just being standard run-time functions loaded from a library. Thus, these tools must allow a lot more custom programming to occur. As these interface designers emerge and evolve, we can expect more and more of the application creation to be taken over by them and consequently, a further decrease in development costs and defect levels.

2.5 The Software Factory (MSR to LSR, Process-Oriented Reuse)

Another reuse approach is the *software factory*, a process that treats software development more like conventional manufacturing than design. Consequently, reuse plays a large role. The software factory concept has been perfected to a high art by a number of Japanese companies. Toshiba's software factory (Fig. 8) (Cusumano, 1989, 1991; Matsumoto, 1989) is a typical example of this kind of reuse. Their domain is real-time process control software for industrial applications, e.g., heavy steel rolling mills. This is MSR (with the potential to evolve into LSR), where the components are not just code, but are artifacts drawn from across the full life cycle. That is, they include requirements, design, and code. In Toshiba's case, these components are specified in formal languages, stored in a reuse repository, and reused in the course of developing customized versions of the products in the product family that they serve. Because the domain is narrow—i.e., a product family—each new version of the product represents only a modest variation on the stored components. That is, every heavy steel rolling mill is different in small ways, such as equipment kind and numbers, mill dimensions, and equipment placement. Such differences can be accommodated with only small changes in the requirements, designs, and code components. This is accomplished by a "copy and edit" approach. Interrelated requirements, design, and code components are retrieved from the repository and manually modified to accommodate each new process control system.

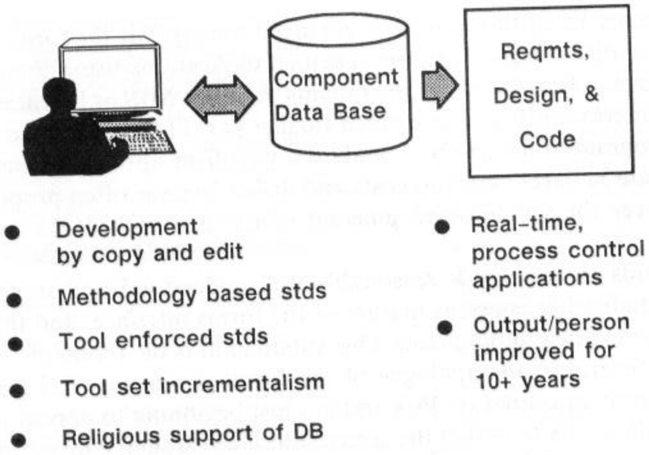

Component
Data Base
Reqmts,
Design, &
Code
• Development by copy and edit
• Methodology based stds
• Tool enforced stds
• Tool set incrementalism
• Religious support of DB
• Real-time, process control applications
• Output/person improved for 10+ years

FIG. 8. Toshiba's software factory.

Because the process is so highly formalized—e.g., through the existence of design languages—it is easy and natural for standards to arise. Further, these standards are enforced by the existence of tools. Both the tools and the associated standards grow and evolve together in an incremental way over the years. Finally, the software factory has a strong commitment to the support and maintenance of the repository system.

In Toshiba's case, the formalized languages, supporting tools, and associated standards form the foundation of the formalized software factory process and provide significant opportunities for leverage on productivity and quality of the code. Between 1976 and 1985 the Toshiba software factory increased its software development productivity from an equivalent of 1390 lines of assembly source code per month to 3100 per month, thereby achieving a cumulative productivity increase of approximately 150%. During the same period, they were able to reduce the number of faults to between one quarter and one tenth of the number that they were experiencing at the beginning of the period (Cusumano, 1989). Other Japanese companies with software factory models (e.g., NEC and Fujitsu) have shown similar improvements in productivity and quality.

What key properties of the software factory model foster reuse success?

- Narrow domains: in this case, the domain is extremely narow (i.e., a product family) leading to the opportunity for reusing very large-scale pieces. However, the measured payoff is more modest suggesting that the degree of customization required for each such component may mitigate the improvement.
- Well-understood domains/architectures, slowly changing domain technology, intercomponent standards, and economies of scale in market: these properties all favor reuse. The domain is a product family that has been perfected over the years leading to a stable, well-understood architecture with well-developed intra-application standards. The very nature of the business establishes an inertia that slows the change of the problem domain's technology. While this is not a huge market, it is clearly a sufficiently large market to make investment in reuse technology worthwhile. In short, these companies have determined that

reuse makes business sense, which is the best measure of the value of applying this technology.

- Infrastructure support: the operational character of these companies provides a nurturing context for such techniques. The strong emphasis on process and the inclination to cast the software development into a manufacturing metaphor provide an infrastructure in which this approach to reuse has a strong opportunity for success.

2.6 Emerging Large-Scale Component Kits (LSR)

Now let's do a little bit of prediction and look at a set of development technologies that are just beginning to emerge. You cannot buy these technologies today, but in a few years you probably will be able to. I believe that interface toolkits will spawn the development of other complementary toolkits that contain larger-scale components—components that are much more oriented to specific application domains and more complex than widgets. This is an example of the emerging field of vertical reuse.

In some sense, large-scale components are an extension of the widget notion but one that is more specialized to particular application domains. (Domain specialization is an inevitable consequence of the growth of component sizes.) For example, desktop publishing is an application domain that is mature enough to supply such components, e.g., fonts, pixel images, graphs, and various kinds of clip art. Spreadsheets are another kind of component that may be included in various applications. What is left to do is to establish standards for their representation that transcend their particular application niche. Once this is done, clip art and spreadsheet frameworks can be imported into and used together within a single application program. Today such integration would be difficult. As transcendent standard representations emerge for these and similar component classes, it will become relatively easy.

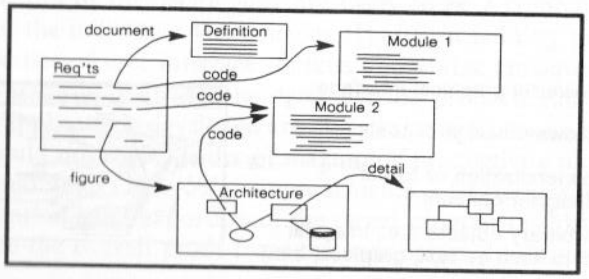
The large-scale component notion is enabled by object-oriented (Cox, 1986; Elis and Stroustrup, 1990; Goldberg and Robson, 1983; Meyer, 1988; Saunders, 1989; Stroupstrup, 1986, 1988) technology in that objects represent a good implementation mechanism. They are finer grained than programs but larger grained than functions or subroutines. Another important characteristic is that objects *hide* the details of their implementations thereby, allowing them to be more easily moved into a new application context without introducing conflicts between the implementation of the object and its surrounding context. Thus, this property makes them more like black boxes that can be plugged in where needed.

This proposed approach has many of the same properties as interface toolkits adjusted to account for bigger components in narrower domain niches. We would expect that the component library would be a compilation of components from a number of mostly independent subdomain niches and the average payoff for each application developed using that library would reflect the degree to which the subdomains addressed the typical functionality in the application programs being developed.

2.7 User-Oriented Information System (LSR to VLSR)

Another kind of toolkit seems likely to emerge, one which is more specialized than DBMSs but less specialized than 4GLs or forms designers. This toolkit, illustrated in Fig. 9, is likely to be a combination of hypertext systems (Bigelow, 1987; Biggerstaff and Richter, 1987; Conklin, 1987; Gullichsen *et al.*, 1986; Smith and Weiss, 1988); frame systems (Brachman and Schmolze, 1985; Fikes and Kehler, 1985; Finin, 1986a, b); object-oriented programming languages (Cox, 1986; Ellis and Stroustrup, 1990; Meyer, 1988; Saunders, 1989; Stroustrup, 1986, 1988); and object-oriented databases (Kim and Lechovsky, 1989). Once again, while you can purchase packages that provide *some* of the characteristics of the desired technology, you cannot purchase a technology with all of the properties that I foresee. Nevertheless, I believe that in a few years you will be able to.

What is happening in this area is a convergence of these four technologies. First, hypertext technologies allow one to deal with various kinds of unstructured data and link those data together in arbitrary ways. Hypertext systems are extending the nature of the user interface, but in another sense, they are also extending database technology.



- **Hypertext (also called hypermedia) systems**
- **AI frame and rule based systems**
- **Object oriented programming systems**
- **Object oriented DBMS's**

FIG. 9. Emerging user-oriented information systems.

The second technology that is part of this evolution is frame- or rule-based systems. These systems organize related sets of information into frames, which are very much like objects. They often provide a rule subsystem whereby the end-user can express inferencing operations on the frames.

The third technology is object-oriented programming systems, which provides an elegant, disciplined, and lucid way to organize programs and their data.

And finally, object-oriented DBMSs are beginning to address the problem of "persistence" of an application's object-oriented data. Persistence addresses the following question: "How does one store large sets of object-oriented data such that it persists from application execution to application execution, and so that many applications can share it in a database-like mode?"

Figure 10 summarizes the properties that such systems will have. These systems will have a powerful graphical interface that operates on a rich hypermedia-based information subsystem. This subsystem will be a toolkit with lots of facilities that can be quickly and easily integrated into individual application programs. It will have browser-navigators available so that one can navigate through the information. They will be highly flexible browser toolkits for building browsers that can be customized to the needs of specific application programs. In some sense, this technology is really a generalization of the forms designer and GUI generator concepts where the toolkit allows complex information to be projected into a greater variety of output forms and media. That is, we have moved beyond business forms and interfaces, and project the information into arbitrary graphical forms. Similarly, we will be offered a wide variety of output media including graphics, sound, full motion video, etc. We can already see elements of multimedia beginning to emerge in the PC marketplace.

- Powerful graphical interface
- Browser/navigator toolkits
- Generalization of forms designer concept
- Arbitrary objects (i.e., irregular data such as text, graphics, etc.)
- Arbitrary linkages
- Inference
- Very large scale data bases of objects
- OODBMS 's with persistence and sharing

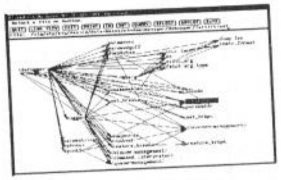


FIG. 10. Properties of user-oriented information systems.

These systems allow one to operate with arbitrary objects and irregular data. Thus, one can intermix text, graphics, speech, animation, etc., and they can link this information together in rather arbitrary ways. That is, one can take a graphics diagram and put a link from any place in the diagram to any other node (i.e., object) in the database. It is not the kind of thing that typical databases allow one to do very well, because they are designed to deal with regular data; i.e., fixed-length items that fit nicely into predefined tables with predefined relationships, i.e., tables and relationships that are processed in a very regular fashion. Hypertext data are not regular, are not of fixed length, do not fit well into predefined tables or predefined relationships, and are not processed in a regular fashion.

Another property of such systems is that they will allow you to do inferencing on the information. For example, one might want to write a rule that checks to see if there is a link between two frames or objects and if there is, execute an operation on the data base.

The merging of object-oriented programming environments and object-oriented databases will allow large systems of objects or frames to persist from one application execution to the next. Thus, applications will be dealing with and sharing large sets of objects over months or years. Such object sets are typically too large to be loaded completely in memory with any given application. Therefore, applications must have the capability to "page" some subset of the object network into memory to be operated on. The object-oriented DBMSs must keep a faithful corespondence between the object images that are in an application program's memory and the object images that reside in the database.

The properties of this technology are likely to be a combination of the properties of the individual technologies. That is, it is likely to have many of the properties of user interface toolkits and 4GLs. However, it is likely that the leverage of these technologies will be proportionally much less because the applications developed are likely to grow in size and complexity. Thus, we would not expect order of magnitude productivity improvements, but rather midrange (20%–50%) improvements. The parkinsonian growth of the application specific portion of the target programs is likely to significantly reduce the overall profit from the user-oriented information system reuse.

2.8 Application-Specific Reuse (LSR to VLSR)

The narrowest kind of reuse is the kind that is focused on a specific application family. The software factory is an example of one implementation of this idea.

In some sense, the application-specific reusable components concept is an extension of the large-scale component concept that we talked about earlier. The main difference is that application-specific reusable components tend to be larger in scale and oriented toward a narrower class of applications, often a family of closely related products. We would not expect to find application-specific reusable components in the commercial marketplace. They are just too specialized. However, we would expect companies to develop such components internally as a way to enhance their ability to deliver variants of products quickly and cheaply.

As a consequence of the increased scale and focus, these components typically provide greater leverage or payoff than large-scale components. But application-specific reusable components are only feasible in well-understood domains where there already exists a high level of domain competence in an organization. That is, if an organization has developed enough avionics systems to thoroughly understand the typical architectures of these systems, then it might be able to create a set of application-specific reusable components for avionics. If not, it would be almost impossible to create such a set because of the large amount of expertise that must be acquired.

If an organization is going to develop a set of application-specific reusable components, it must analyze the domain. The organization must determine what set of components will be of greatest benefit to its product sets. One way is to look at the components of previously developed systems and harvest them for the component library. Of course, some energy will have to be invested to generalize these components and make them more reusable, but that is easy in comparison to the overwhelming alternative of creating all of the components from scratch. The results of this domain analysis should include (1) a vocabulary of terms that cover all important problem domain concepts and architectural concepts, (2) a set of standard data items that range across the whole domain and serve as the inputs and outputs for the components, and (3) a set of abstracted designs for the reusable components that will be used to construct the target applications. These results are generalizations of the concepts, data, and components found in existing systems and establish a framework of intercomponent standards that are important to component reusability.

For a reuse library to be successful, it must be established on a rich and well-defined set of intercomponent standards. That is, one must make sure that the set of components derived from the domain analysis will plug together easily and can be reused in new applications without a lot of effort to adapt them. The data items, which are standard across all of the components in the library, are the key concrete manifestation of these intercomponent standards. Without such a framework of intercomponent standards, a reuse library has a high probability of failing. With such a framework, the chances of success increase significantly.

Such a framework of intercomponent standards is critical to all reuse efforts, but they become a more and more important factor as the scale of the components increases. Hence, application-specific reuse with its large and very large components, amplifies the importance of such standards.

This need to analyze domains via the analysis of existing programs, is spawning a new class of tools, which we call *design recovery tools*. These are really a generalization of *reverse engineering tools*. While reverse engineering tools are largely aimed at porting or cloning existing programs, design recovery tools are aimed, in addition, at helping human beings to understand programs in human-oriented terms. Operationally, reverse engineering tools are largely concerned with the extraction of the formal information in programs (that information expressible via programming language formalisms) whereas design recovery tools are more concerned with establishing a mapping between the formal, implementation structures of a program and the semiformal, domain-specific concepts that humans use to understand computational intentions. For example, consider the mapping from an array of C structures to the architectural concept *process table*. I have coined the term "the concept assignment problem" to describe the problem of creating such mappings and the concept assignment problem is the central problem being addressed by design recovery tools.

The understanding developed with the aid of design recovery tools serves several purposes beyond simply porting application programs, purposes such as re-engineering, maintenance, domain analysis, and reuse library population. While the subject of design recovery and the related subjects of re-engineering, maintenance, reverse engineering, and domain analysis are highly important to reuse, they are beyond the scope of this chapter. Suffice it to say that these are all critically important subjects to organizations engaged in reuse.

2.9 Designer/Generators (LSR to VLSR)

Another class of reuse facilities currently being developed in research laboratories are the designer/generator systems. These systems add abstract design components to the reusable libraries to push the reuse activities back into the design cycle. Further, they add rules plus a rule-driven shell around the reuse libraries to allow some automation of the design process. The rules define how to specialize (i.e., add the missing details to) the design components and how to interconnect various design components to form a complete target program. Designer/generator systems are typically mixed-initiative systems with the software engineer providing requirements, missing designs, and a large dose of intelligent decision making. By this technique, the systems go from simple requirements and specifications directly to code.

In essence, they emulate the kind of design construction process that human designers perform. If the libraries are well populated in the domain of interest to the software engineer, the development of a system is like a game of 20 questions with the user giving an initial set of requirements and specifications and then after that only participating when the system gets stuck or needs more information. The end product is executable code for the desired application.

The ROSE reuse system (Lubars, 1987; Lubars, 1990; Lubars, 1991) is an example of a designer/generator system (see Fig. 11). It is a prototype that was developed to experiment with this kind of semiautomated reuse system. ROSE has two libraries—one of design schemas and one of algorithms—both of which are expressed in forms more abstract than code. ROSE takes a specification of the target system in the form of a data flow diagram built from abstract data types and abstract operations. It attempts to design the target system from this specification by choosing design schemas from its reuse library to fill out the lower levels of the design. The specifications that it starts with are ambiguous in the sense that most details of the target system are not determined by the initial specifications. Thus, the system develops the details of the design by four mechanisms: (1) choosing candidates for lower-level design schemas from the design library; (2) inferring design details via constraints attached to the designs; (3) transforming and specializing pieces of the developing design by using transformation rules (i.e., design rules) from the library; and (4) soliciting information from the software engineer when it gets stuck. Once the design has been worked down to atomic design elements, it is still more abstract than code and goes through another step which maps (i.e., compiles) the design into algorithms specified in some specific programming language.*

If the library is reasonably well populated within the target domain, much of the target program's development is automated and a working program of a few hundred lines of code can be produced in 10–15 minutes of work. If the library is incompletely populated, then the process becomes progressively more manual depending on the level of design library population. With a completely empty library, the system behaves much like a conventional CASE system and requires about the same level of effort as developing the target program with a CASE system.

In the case of designer/generator technologies, most of the key factors that we have identified with successful reuse systems are defined more by the nature of library components than by the designer/generator technology itself. In theory at least, one can populate the libraries with elements of any

* The experimental version of ROSE produces target application programs in three languages: C, Pascal, and Ada.

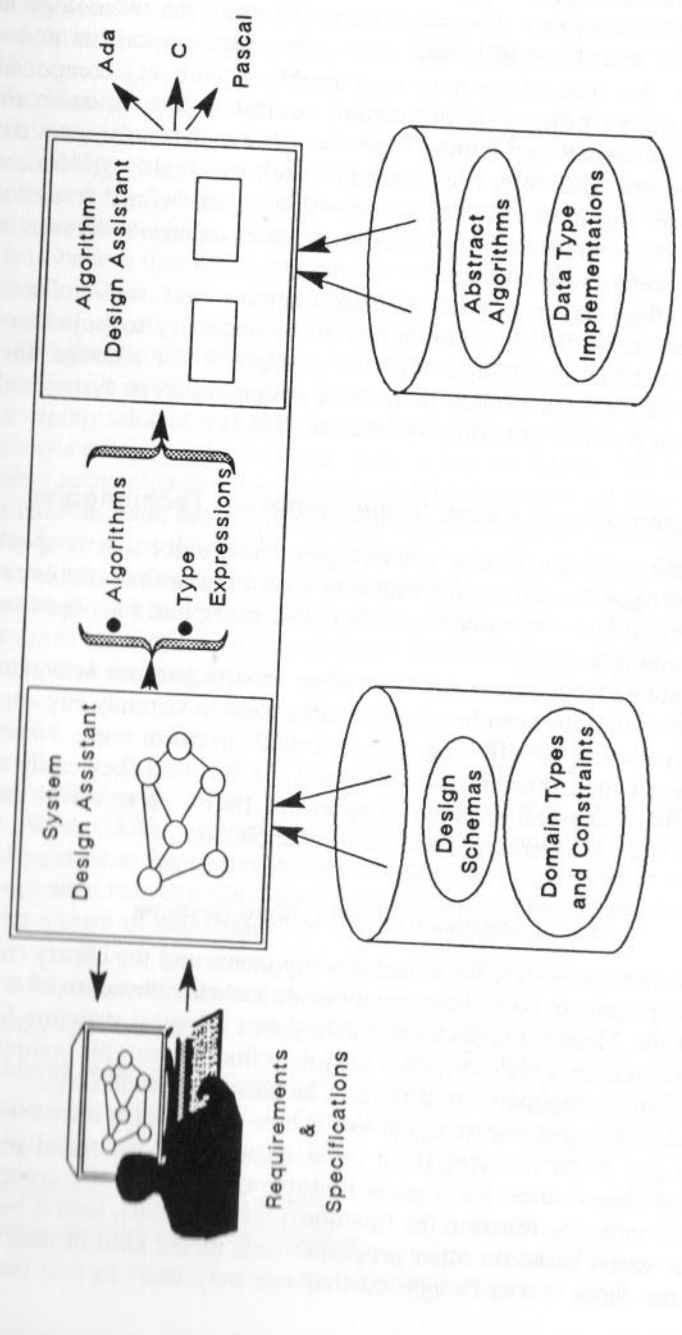


FIG. 11. ROSE reuse system.

scale and populate them completely enough to build large percentages of the target applications out of reusable parts. To date, the technology has not been tested with large-scale and very large-scale components and we speculate that this technology may have problems with big components within a regimen of nearly full automation because such a situation may impose large inference requirements on the system. Therefore, to date, designer/generators only have been shown to work reasonably well for components between medium and large scale within a well-defined framework of domain/architecture standards and constraints. It remains to be seen how well this technology will scale up.

This technology is best suited for very narrow and well-understood domains because of the large amount of effort necessary to populate the reuse libraries. In fact, the large effort to populate ROSE's design library led to the creation of a project to build a design recovery system called DESIRE (Biggerstaff, 1989; Biggerstaff *et al.*, 1989).

3. Examples of Reuse Implementation Technologies

This section considers generic technologies that are not strictly speaking reuse technologies but are implementation technologies that enable reuse: (1) classification and library systems, (2) CASE tools, and (3) object-oriented programming systems.

These enabling technologies are themselves broad-spectrum or horizontal technologies in that they can be used to enable reuse in virtually any application domain and enable either narrow- or broad-spectrum reuse. Nevertheless, because of their inherent generality and the fact that they easily allow the specification of small reusable components, they tend to orient toward broad-spectrum or horizontal reuse in their application.

3.1 Classification and Library Systems

The classification system for reusable components and the library (repository) system used to hold those components are two elements of a reuse infrastructure. These elements largely help define a logical structure for the application domain, which simplifies the job of finding reusable components and identifying components that need to be added to the library.

A key classification research problem is how to organize the overall library. For any given domain, there often is no single canonical or ideal hierarchical classification for a given reusable component. If a component is classified under the function (or functions) it implements, then it becomes difficult to access based on other properties such as the kind of data that it operates on. Since it was recognized that one may want to find the same component based on different properties, classification schemes have evolved that take the library science approach of allowing a component to be found based on any of a number of its properties (called "facets") (Prieto-Diaz, 1989).

The library system itself is a rather minor, though conspicuous, element of the reuse infrastructure. Its role can be viewed in two ways: (1) as a piece of technology that is key to the success of the reuse effort, or (2) as a piece of infrastructure whose main value is in establishing a process context and thereby enhancing the value of associated reuse technology. The author tends to believe that the second view is closer to the truth and that too much emphasis on the technical importance of the library system can lead one to focus too little on other more critical elements of the reuse project.

To put the above notion in concrete terms, when a company is setting up a reuse effort, it is often easier to build a library system than to try to understand exactly what is really needed and what kind of technology best fits the company's environment. In some cases, a manual library system may be a perfectly acceptable solution initially and the key technical innovations may lie in choosing and analyzing the appropriate domains. If a reuse proposal is only focused on the design of the library system, then it is quite possible that too little thought has been given to other more important aspects of the problem such as the specific domain and components to be reused. The library is not unimportant. It is just not the first thing to think about when planning and creating a reuse system.

3.2 CASE Tools

Figure 12 characterizes CASE systems (Chikofsky, 1988, 1989; Fisher, 1988). Most CASE systems focus largely on the problem of drafting, i.e., providing engineering drawings that describe the design of software systems. They are most effective when used to design large-scale systems being developed by a team of designers.

- Mostly automated drafting systems
- Diagrams
- Shared repository
- Document generation
- Consistency checking (weak)
- Prototyping (e.g., screen layout)
- Code generation (but NOT creation)

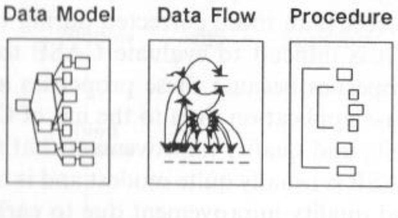


FIG. 12. Characterization of CASE systems.

CASE systems provide several kinds of diagrams that support various design methodologies. For example, they typically provide diagrams that describe data relationships, e.g., data flow diagrams that show how data flow through the various parts of the system, and procedural representations from which the source code can be derived, sometimes semiautomatically. In addition, they typically provide a shared repository and help in managing the diagrams; document generation capabilities for including design diagrams in design documents; and various kinds of analyses that report on the state of the design. They often do some weak consistency checking, e.g., verifying that the right kind of boxes are connected to the right kind of arrows. Some CASE tools provide limited prototyping capabilities such as screen layout facilities. With these tools, one can design the screen interface to the target system and then generate the interface code, much like the forms designers discussed earlier.

The major benefit of using a CASE tool is that the evolving design is recorded formally, e.g., in data flow diagrams, statecharts, predicate calculus, etc. The real value of CASE tools arises out of using these design representations as a working model during the development process. The act of using design formalism forces many design issues out into the open that would otherwise remain hidden until late in the design process. Moreover, it uncovers omissions in the design. But the most important effect is the migration of the design model into the heads of the designers. After all, it is the in-head knowledge that one uses during the whole developmental process.

Productivity improvement with CASE tools is often modest. Some savings result because design updates are easy with CASE tools and because the design and the code are integrated and often managed by the CASE system. But overall, the direct savings are modest.

The major, but indirect, benefits of CASE systems come during the testing and maintenance phases. Because the details of the target design are expressed early, the errors and the defects can be seen and detected early. This tends to lead to a higher-quality system with fewer defects to correct after the system has been delivered. The productivity improvement arises largely because postdelivery defects cost two orders of magnitude more to correct than those corrected during the design phase.

It is difficult to evaluate CASE tools against our proposed set of reuse properties because these properties are more sensitive to the nature of the reuse application than to the use of CASE tools. Consequently, the productivity and quality improvement that result strictly from the reuse aspects of CASE is usually quite modest and is often overshadowed by the productivity and quality improvement due to early defect detection.

An inherent value of CASE tools to reuse applications is the infrastructure support that CASE tools provide to the reuse process. Another inherent value of CASE tools is that they tend to foster reuse of software *designs* in addition to reuse of code. Since designs are more abstract than code, they tend to have a higher opportunity for reuse and thereby have a higher payoff potential.

3.3 Object-Oriented Programming Systems

Object-oriented systems (Cox, 1986; Ellis and Stroustrup, 1990; Goldberg and Robson, 1983; Meyer, 1988; Saunders, 1989; Stroustrup, 1986, 1988) impose a structure on a program by developing a graph of related *classes* of objects (see Fig. 13). For example, one could define a *rectangle* as the class of displayable, graphic objects that are rectangular and relate it to its *superclass* of "graphic object," i.e., the class of all displayable, graphic things. Further, one could define a *subclass* of rectangle called a *window*, i.e., a displayable, graphical rectangle that has additional properties and behaviors over those of a rectangle. For example, a window is a displayable rectangle that can accept input from the keyboard and mouse, and produce output within its rectangular aperture. One could design other subclasses (i.e., specializations) of graphic objects such as circle, ellipse, and so forth.

As shown in Fig. 14, each such class corresponds to some set of real-world objects. For the user interface classes, the real-world objects might be graphical manifestations that are drawn on a computer screen. For example, a rectangle could be part of a line drawing; or with certain additional characteristics, it might be a window; or with even more specialized characteristics, it might be a *tiled window*—i.e., a window with panes in it; or it could be a *browser*, i.e., a window that knows how to display graphs; and so forth.

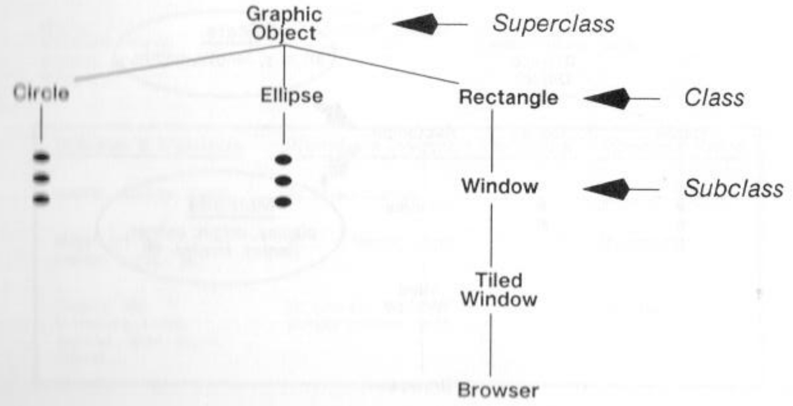


FIG. 13. Example class hierarchy.

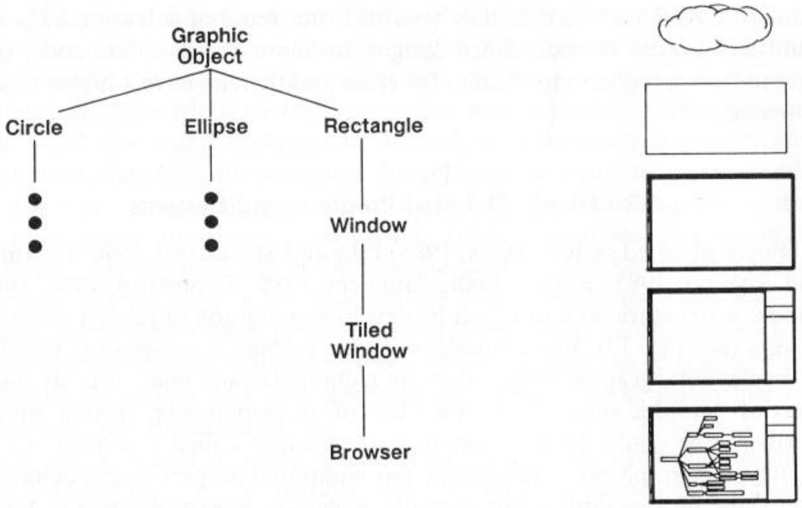


FIG. 14. Classes and real-world objects.

Each class has two kinds of information associated with it, as shown in Fig. 15. One is state information that defines an instance of the class. For example, a rectangle class would have *instance variables* x and y that define the position of its upper-left corner. Further, it would have instance variables that define its length and width.

The second kind of information associated with a class is a set of so-called *methods* that define the behavior of that class. These methods manage the state information defined by the instance variables. Examples of such methods are *display*, which draws the rectangle on the screen; *origin*, which returns the (x, y) position of the rectangle on the screen; and so forth.

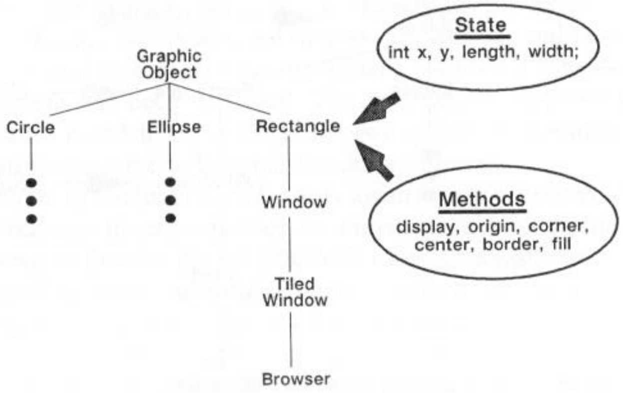
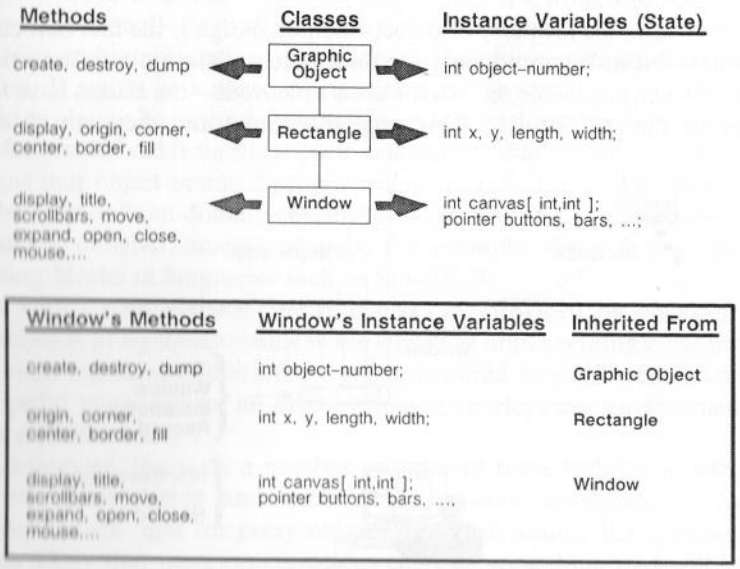


FIG. 15. Structure of classes.

One of the important object-oriented concepts is *inheritance*, which is also called *subclassing* and is illustrated in Fig. 16. The idea is that if I already have the definition of a rectangle and want to define something that is a specialized instance of rectangle, like a window, all I have to do is specify the additional data (i.e., instance variables) and behavior (i.e., methods) of a window over that of a rectangle. In other words, a window is something that has all of the same state information as a rectangle but, in addition, has some state specific to it. For example, a window might have a canvas containing a set of pixels to be displayed. Further, it might have a list of control facilities like buttons and scrollbars.

In addition to the extra state information, a window may have additional methods. And it might also replace some of the rectangle's methods (e.g., the display method of rectangle is replaced by the display method of window in Fig. 16).

To put it more abstractly, classes represent the definition of the state and the behavior of the set of all objects of a given type. An individual member of that set of objects is called an *instance* of the class or alternatively, an *object*.


Methods
Classes
Instance Variables (State)
create, destroy, dump
Graphic Object
int object-number;
display, origin, corner, center, border, fill
Rectangle
int x, y, length, width;
display, title, scrollbars, move, expand, open, close, mouse,...
Window
int canvas[int,int]; pointer buttons, bars, ...;

Window's Methods	Window's Instance Variables	Inherited From
create, destroy, dump	int object-number;	Graphic Object
origin, corner, center, border, fill	int x, y, length, width;	Rectangle
display, title, scrollbars, move, expand, open, close, mouse,...	int canvas[int,int]; pointer buttons, bars, ...	Window

FIG. 16. Subclassing and inheritance.

An instance is implemented as a data record that contains all of the state information that describes the individual object. This is illustrated in Fig. 17. Thus, a tiled window instance record would contain all of the state information unique to tiled windows, plus all the state information inherited from window, plus all the information inherited from rectangle, plus all the state information inherited from graphic object. The record containing all of that state information represents an instance of a tiled window.

Now, when a method is called, it performs some operations that use or change the state information in the instance record. Examples of messages are display yourself, change your size, move the canvas under the window aperture, and so forth.

One of the most important properties of object-oriented systems is that they impose an extra layer of design discipline over conventional languages. They allow one to formally express additional information about the architectural organization of a system beyond what one can express in a typical high-level language such C or FORTRAN. More to the point, they insist on that architectural information. They insist that one cast the design of a system in terms of a set of related classes that correspond in a natural way to the real-world entities that the system is dealing with. This discipline helps one to develop a cleaner and more elegant design structure, in the main, because it forces the designer to explicitly think about the real-world entities and their interrelationships, and this enhances the reusability of the resulting classes.

Another valuable property of object-oriented design is the fact that classes are natural reusable components. Because much of the state information is "hidden"—i.e., accessible only to the class's methods—the classes have fewer constraints that tie them to their original context and they can be easily relocated and reused in new contexts in other programs. And because classes are conceptually larger-grained components than functions, their reuse tends to provide better productivity and quality improvement than reuse of functions.

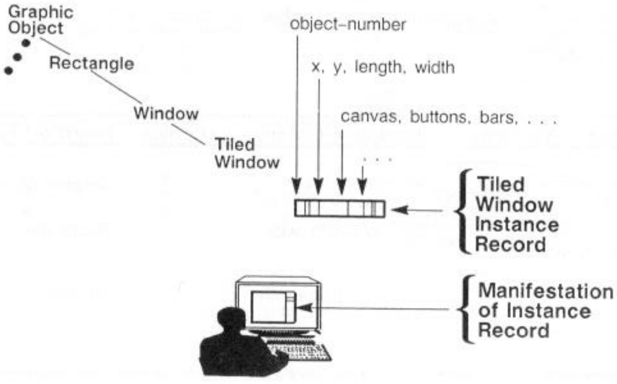

Graphic Object
Rectangle
Window
Tiled Window
object-number
x, y, length, width
canvas, buttons, bars, . . .
. . .
Tiled Window Instance Record
Manifestation of Instance Record

FIG. 17. Instances of classes.

On the other hand, from a reuse perspective, classes are still relatively small-grained components and one would really like even larger-scale reusable components. Fortunately, object-oriented systems also provide a platform for creating larger-scale reusable components, called frameworks. A *framework* is a set of classes that taken together represent an abstraction or parameterized skeleton of an architecture.

The final benefit of object-oriented development is inheritance. It reduces the amount of programming effort necessary. Because one already has some functionality defined in an existing class, building a specialization of that class is much simpler than starting from scratch. Some of the data and some of the functions are already written and can be inherited from the superclass.

The reuse benefits of object-oriented programming systems are analogous to the reuse benefits of CASE systems. That is, the productivity and quality benefits are more sensitive to the existence of a reuse infrastructure than the fact the object-oriented programming is involved. Nevertheless, we must admit that object-orientation mitigates toward somewhat larger-scale reuse that function orientation and therefore, there is a tendency toward improvements in productivity strictly due to the object-orientation. Even so, object-oriented languages are inherently broad spectrum and tend to most easily enable small- or medium-scale component reuse. Therefore, the productivity and quality gains due strictly to the object orientation tend to be modest. It would be a guess substantiated only by intuition but these gains would probably be in the 5–10% range. Additional productivity and quality benefits are derived from the reduction in defects that accrue from the cleaner designs that object-oriented programming styles foster. Still further benefits can be derived from domain-specific facilities that particular object-oriented languages or environments provide, for example, the rich user interface building blocks in languages such as SmallTalk.

As with CASE systems, the infrastructure provided by object-oriented languages is of significant value in the course of implementing reuse libraries. Although significant additional work is required to implement a complete and useful reuse system, an object-oriented development environment provides a head start.

In summary, there are a number of different reuse technologies that can improve productivity and quality in software development. Not all approaches are right for every organization, but among the approaches, it is very likely that most organizations can find something that will fit their culture and needs. There are no magic wands or silver bullets that will give an organization orders of magnitude improvement over *all* the software that it develops. But there are a number of individual approaches which if used conscientiously within receptive contexts, will provide significant increases in productivity and quality.

4. Effects of Key Factors

The objective of this section is to explore the relations among the reuse success factors and in the course of this exploration, to develop an analytical model that quantifies the relationship between some of the key factors and the productivity and quality benefits that they produce. We will also explore—in an intuitive manner—the relationship between specific reuse technologies and their potential for productivity and quality improvement.

4.1 Relationships among the Reuse Factors

Cost is arguably the most important metric in software development and it can be viewed as the sum of the following costs:

- Cost of developing new custom components for the target software.
- The taxes on the reused components (i.e., the amortized costs to develop and maintain the reusable components).
- Cost to assemble the components into a system (i.e., integration or "plumbing" costs).
- Cost to remove the defects from the target system, which breaks down into two costs: (1) cost of removing defects from the component software and (2) cost of removing defects from the integration software, i.e., the plumbing software.

Figure 18 shows how these various costs are affected by the key factors that we used to characterize successful reuse systems. We can see that among the independent factors, the degree to which the domain is understood, the breadth of the domain chosen, and the specific kind of reuse technology have an effect on three key, dependent factors: (1) the amount of reuse within the target application, (2) the scale of the components being reused, and (3) the intercomponent connection standards. These in turn affect several elements of the total cost. The larger the amount of reuse (i.e., the larger the proportion of the application built out of reusable components), the less one has to spend on developing new components for the target application. Similarly, the larger the proportion of reused components in an application, the less one has to spend on removing defects, because the reused components have fewer defects to start with. The number of defects in a reusable component generally decreases the more the component is reused.

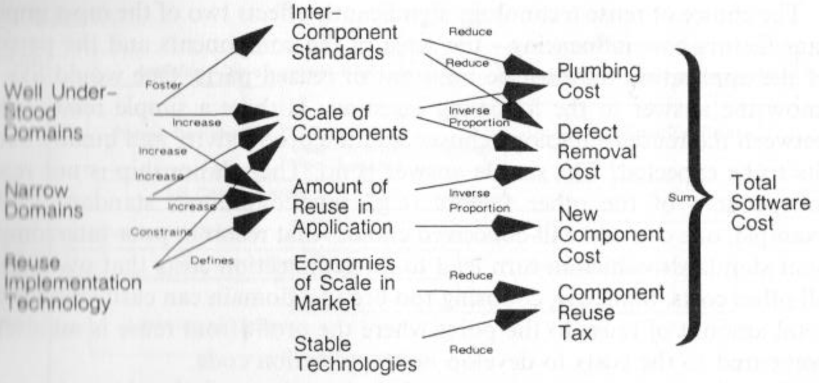


FIG. 18. Relationships among key factors and cost.

The scale of components typically affects the cost to assemble the components. Assembly of larger-scale components requires less plumbing and introduces fewer plumbing errors, both of which reduce costs. This is the same kind of cost reduction phenomenon seen in hardware: it is cheaper to build a device out of very large-scale integration (VLSI) components than out of small-scale integration (SSI) components.

Finally, intercomponent standards reduce plumbing costs mainly by reducing the amount of specialized code that must be developed to hook components together. The more highly standardized the interconnections, the less effort it requires to assemble the applications out of components. The following section will examine this phenomenon analytically.

Figure 18 should make it clear that the final effect on the software cost is wrought by a mixture of technology and business decisions. While it is important to carefully consider exactly what reuse technology is right for the organization and problem at hand, one must keep in mind that the effects of the best reuse technology can be nullified by ill-considered business decisions. For example, a poor choice of an application domain (e.g., one that the organization knows little about or one that is rapidly evolving), or a decision to accommodate too broad a part of the application domain, can overwhelm any productivity or quality improvement potential provided by the reuse technology. Therefore, while we focus much of our attention in this chapter on reuse technologies, successful reuse can only be achieved through good technology and business decisions.

The choice of reuse technology significantly effects two of the most important factors cost influencing—the scale of the components and the percent of the application that can be built out of reused parts. One would like to know the answer to the following question: Is there a simple relationship between the reuse technology chosen and the productivity and quality benefits to be expected? The simple answer is no. The relationship is not really independent of the other factors (e.g., intercomponent standards). For example, one can make ill-conceived choices that result in poor intercomponent standards, which in turn lead to interconnection costs that overwhelm all other costs. Similarly, choosing too broad a domain can easily reduce the total amount of reuse to the point where the profit from reuse is minuscule compared to the costs to develop new application code.

Nevertheless, intuition suggests that there is a relationship, given the assumption that choices for other factors are reasonable. We will assume a reasonably good domain choice with stable technology and components that have a high potential for reuse. Given these assumptions, there does seem to be a rough relationship between the technology chosen, the scale of the components implied by that technology, and the percent of the target application that is constructed out of reused components. And since the dollar savings to be realized from reuse correlates directly with the percent of the target application that is constructed out of reused components, we will express the benefits of reuse in terms of the potential percent of reuse in the target applications rather than dollars. Figure 19 is the author's perception of the relationship among technologies, component scale, and the percent of the target application that can potentially be built out of reusable components.* It is intended solely as a conceptual description and is not for estimation purposes. To this writer's knowledge, no one has yet done the empirical research necessary to establish a relationship between technology choices and the productivity and quality improvements.

All other things being equal, technologies that fall in the upper right-hand portion of the diagram have the potential to provide large improvements in productivity and quality; i.e., generally more than 50% cost reduction. Those in the lower left-hand portion can provide 0–20% cost reduction, and those elsewhere in the chart are probably somewhere in between.

However, let me remind the reader once again that this is at best an intuitionally based relationship that suggests potential, not one that guarantees specifics. It is easy in specific cases to manipulate the other factors to completely confound the relationship. Now let us take an analytical look at the relationship between some of the reuse factors.

* Since each of the technologies shown in Fig. 19 allows quite a bit of implementation flexibility, they are drawn as boxes to indicate ranges along both axes.

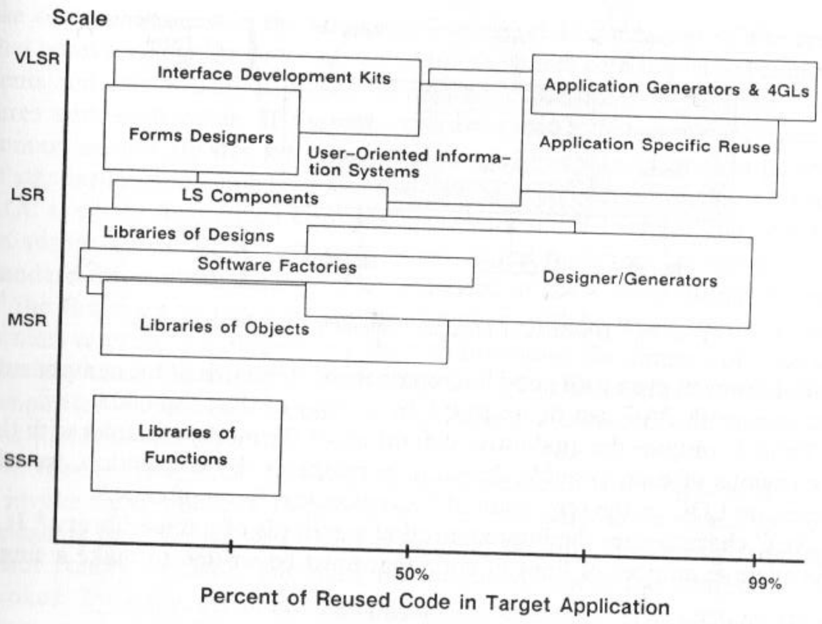


FIG. 19. Productivity and quality improvement estimating heuristic.

4.2 A Quantitative Model of the Relative Amount of Integration Code

This section introduces an analytical model to predict the effects of component scale and intercomponent standards on the plumbing costs and thereby, on the eventual profit wrought by reuse. We will do this by examining the amount of code required to connect the reused components into a target application program for various levels of component scale and intercomponent standards.

4.2.1 Definitions

Figure 20 defines the key model variables. Specifically, we want to determine PAC—the proportion of the total code in the application that is committed to connecting the reused components—because PAC is proportional to the overhead costs associated with reuse. The desired situation is where PAC is a very small proportion of the total code, ideally near zero. As predicted by our earlier qualitative analysis, this ideal is approached in the case of large components with good intercomponent standards. We will see that in the case of poor intercomponent standards, PAC can exceed 0.7 (i.e., 70%) of the total code in the target application, whereas with good intercomponent standards and relatively large components, PAC approaches

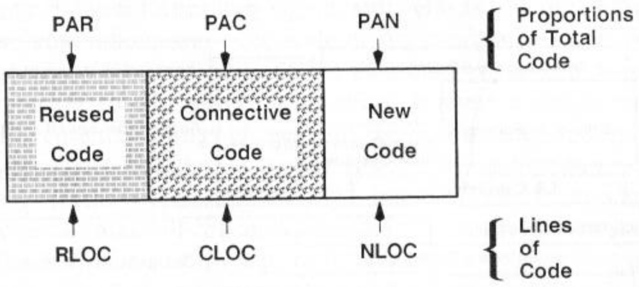


FIG. 20. Divisions of program containing reused components.

zero. However, even with good intercomponent standards, if the components are too small, PAC can be up to 0.5 (i.e., 50%) of the total code.

Table I contains the qualitative definitions of the model variables with the dimensions of each variable shown in parentheses. By convention, we will often use LOC in the text as an abbreviation for "lines of code."

ACC characterizes the interconnection standards of a reuse library.* It is the average number of lines of code that must be written to make a single

TABLE I

DEFINITIONS OF VARIABLES IN MODEL

Inputs characterizing library	
ACC	Average connectivity complexity (LOC/Connection)
SC	Average scale (LOC)
Inputs characterizing target application	
AFI	Average Fan-In (Connections)
NLOC	Number of lines of new code (LOC)
RLOC	Number of lines of reused code (LOC)
Outputs	
CLOC	Number of lines of connection code (LOC)
NRC	Number of reused components in target application (no. of components)
P	Ratio of new LOC to reused LOC (dimensionless)
PAC	Proportion of connection code in target application (dimensionless)
PAN	Proportion of new code in target application (dimensionless)
PAR	Proportion of reused code in target application (dimensionless)
TLOC	Total number of lines of code in target application (LOC)

use of a component in the target application. It is a measure of the code that wires together the new code and data structures with the reused components and data structures, as well as the reused components and data structures with each other. If the data structures for conceptually equivalent domain entities are standard across the whole library and there is a pattern of standardization among the function interfaces, object protocols, etc. then ACC is small. As a trivial example, consider a set of routines that operate on strings where all strings used by all of the functions are stored in a standard format. The amount of code needed to use a string output by one of the functions as input to another function will be small. If the string formats required are different for the two functions, the amount of code to interface them will be significantly larger. While this example is trivial in comparison to real components, it illustrates the nature of the standards that we are discussing.

In the best case, ACC is a single statement, which is the statement used to invoke the component. In the real world, this is seldom the case. Usually, the calling interface requires different forms of the data or requires data that is not readily available but must be computed before the component is invoked. Typically, the plumbing code characterized by ACC includes such things as computation of required data; reorganization of existing data structures (e.g., transforming a zero-end-marker string into a length-tagged string); the creation of new data structures required for input, output, or operation; database creation; database operations; file operations; and so forth. This connectivity code can be extensive if the various data-related components hew to widely different standards.

Although the ideal for ACC is one, it is often not achieved. An example serves to illustrate this. In order to reuse an existing parser, one often has to write a postprocessor that transforms the parse tree computed into a new form that fits the context of a different system. Often other computational extensions also need to be made for the new context. All of this code must be written from scratch and contributes to the average connectivity complexity for each of the components within the reuse library.

The second model input variable that characterizes the reuse library is SC, the average scale (i.e., size in LOCs) of the components in the library.

The other key inputs are defined by the target application program. AFI is the average number of connections required for a typical component. Each such connection requires ACC lines of code on the average.

An example is in order to clarify the true nature of and relationship between ACC and AFI. Even though in the model we are considering average connections, a concrete example using individual connections and plumbing code will make the relationship clearer. Let us suppose that

* The ACC characterizes the expect number of lines of code needed to make a connection to a component. It is an average computed over many uses of a set of data-related components within different applications and is a convenient characterization of expected connectivity properties. It is not meaningful with respect to specific individual components or specific individual applications. It is the computed average over all components in a library and over a large number of reuse experiences of the total lines of connectivity code required in those applications divided by the total number of connections required in those applications.

$f(x, y, z)$ is a reusable function. The plumbing code required to integrate f into a program consists of two parts: (1) the set of code that packages and formats the inputs to f—for example, x, y and some global data structure g—and later unpackages and reformats any outputs of f—e.g., z and the data within g; and (2) the code that makes the data transfers happen—e.g., a call statement or a process spawn. If the packaging/unpackaging code is the same for every use of f in the program, then one can write functions to do the packaging and unpackaging, and amortize that code over the many invocations of f in the new program. On the other hand, if we have several distinct kinds of uses of f, each requiring packaging/unpackaging code that is so different that we cannot use a single set of functions to do the packaging/unpackaging, then we must amortize each distinct set of packaging/unpackaging code over its set of uses and use the average of those to compute ACC. Thus, only in the simplest case do the lines of code counted by ACC correspond to a specific programming artifact (e.g., a subroutine or function) within a target program. More generally, ACC represents some proportion of such artifacts averaged over many uses.

The next two input variables define the number of lines of code in a target application program that are reused (RLOC) and new (NLOC).

From these model input variables, we calculate CLOC, the number of lines of code required for connection of the reused components into the target application. TLOC (the total number of lines of code in an application) can also be calculated from these inputs as can the various proportions of code types in the application—PAR (reused), PAC (connection), and PAN (new). The average number of components in a target application—NRC—can be computed from these variables. We are most interested in how PAC changes as we vary our assumptions about the degree of intercomponent standardization and the relative scale of the components.

We introduce another variable P, which is the ratio of new code to reused code. This ratio is useful because we are less interested in the absolute magnitudes of NLOC, RLOC, and CLOC than in the relative proportions of these quantities and how those proportions change under differing sets of assumptions.

The variables ACC, SC, AFI, RLOC, CLOC, and NLOC are ripe for empirical studies to characterize various reuse libraries and compare the library characterizations against the results of reusing components from those libraries in real application programs. This would provide some measure of goodness for reusable libraries and eventually result in standards against which reuse libraries could be measured.

4.2.2 *The Model*

The following equations define the relations among the variables.

$$\mathrm{TLOC} = \mathrm{NLOC} + \mathrm{RLOC} + \mathrm{CLOC} \tag{4.1}$$

$$\mathrm{PAR} = \frac{\mathrm{RLOC}}{\mathrm{TLOC}} \tag{4.2}$$

$$\mathrm{NRC} = \frac{\mathrm{PAR} * \mathrm{TLOC}}{\mathrm{SC}} = \frac{\mathrm{RLOC}}{\mathrm{SC}} \tag{4.3}$$

$$\begin{aligned}\mathrm{CLOC} &= \mathrm{NRC} * \mathrm{AFI} * \mathrm{ACC}\\ &= \frac{\mathrm{RLOC} * \mathrm{AFI} * \mathrm{ACC}}{\mathrm{SC}}\end{aligned} \tag{4.4}$$

$$\mathrm{P} = \frac{\mathrm{NLOC}}{\mathrm{RLOC}} \tag{4.5}$$

$$\mathrm{PAC} = \frac{\mathrm{CLOC}}{\mathrm{TLOC}}. \tag{4.6}$$

Now we work PAC into a form that is more amenable to approximation.

$$\mathrm{PAC} = \frac{\mathrm{NRC} * \mathrm{AFI} * \mathrm{ACC}}{\mathrm{TLOC}}.$$

Using the first form of Eq. (4.3)

$$\mathrm{PAC} = \left(\frac{\mathrm{PAR} * \mathrm{TLOC}}{\mathrm{SC}}\right) * \left(\frac{\mathrm{AFI} * \mathrm{ACC}}{\mathrm{TLOC}}\right)$$

which allows us to cancel out the absolute quantity TLOC leaving

$$\mathrm{PAC} = \frac{\mathrm{PAR} * \mathrm{AFI} * \mathrm{ACC}}{\mathrm{SC}}. \tag{4.7}$$

We want Eq. (4.7) in a form that involves only AFI, ACC, SC, and P, so we reformulate PAR.

$$\begin{aligned}\mathrm{PAR} &= \frac{\mathrm{RLOC}}{\mathrm{TLOC}}\\ &= \frac{\mathrm{RLOC}}{\mathrm{RLOC} + \mathrm{NLOC} + \mathrm{CLOC}}.\end{aligned}$$

Using Eq. (4.4) for CLOC, we get

$$\begin{aligned}
\text{PAR} &= \frac{\text{RLOC}}{\text{RLOC} + \text{NLOC} + \left(\dfrac{\text{RLOC} * \text{AFI} * \text{ACC}}{\text{SC}}\right)} \\
&= \frac{\text{RLOC} * \text{SC}}{\text{SC} * \text{RLOC} + \text{NLOC} * \text{SC} + \text{RLOC} * \text{AFI} * \text{ACC}} \\
&= \frac{\text{RLOC} * \text{SC}}{\text{RLOC} * (\text{SC} + \text{SC} * \text{P} + \text{AFI} * \text{ACC})} \\
&= \frac{\text{SC}}{\text{SC} + \text{SC} * \text{P} + \text{AFI} * \text{ACC}} \\
&= \frac{\text{SC}}{\text{SC} * (1 + \text{P}) + \text{AFI} * \text{ACC}}. \qquad (4.8)
\end{aligned}$$

Substituting Eq. (4.8) into Eq. (4.7), we get

$$\text{PAC} = \left(\frac{\text{SC}}{\text{SC} * (1 + \text{P}) + \text{AFI} * \text{ACC}}\right) * \left(\frac{\text{AFI} * \text{ACC}}{\text{SC}}\right).$$

Canceling our SC, we have a form that is good for approximation analysis.

$$\text{PAC} = \left(\frac{\text{AFI} * \text{ACC}}{\text{SC} * (1 + \text{P}) + \text{AFI} * \text{ACC}}\right). \qquad (4.9)$$

Now let us consider three cases:

1. a library with poor interconnection standards
2. a library with good interconnection standards but small components
3. a library with good interconnection standards and relatively large components

For case 1, we define a library with poor standards as one in which ACC is equal to SC. In other words, it takes about as much code to make an interconnection to a reused component as is in the reused component itself, on the average. Substituting SC for ACC in Eq. (4.9) and canceling SC, gives us

$$\text{PAC} = \frac{\text{AFI}}{(1 + \text{P}) + \text{AFI}}. \qquad (4.10)$$

Notice that the size of the component does not appear because of our destandardization assumption. This is not just an interesting theoretical case. Anecdotal evidence suggests that it often happens that SC and ACC are nearly the same in libraries of casually assembled components.

Figures 21a and 21b show two views of how PAC is affected by various values of AFI and P for case 1. Notice in Fig. 21b in particular that where P = 0, in the limit, PAC approaches 1.0 as AFI approaches infinity. However, for all P, we see that the proportion of connection code grows as AFI grows. While for large Ps the *relative amount* of connection code decreases, it does so only because the relative amount of reused code is diminishing. This relative decrease in PAC is not cause for rejoicing, because the absolute amount of work may still be substantial. More to the point, the amount of work necessary to reuse code can be more than the amount of work required to rewrite the reused code from scratch. Looking at the PAC/PAR ratio,

$$\frac{\text{PAC}}{\text{PAR}} = \text{AFI} = \text{Fan-In}$$

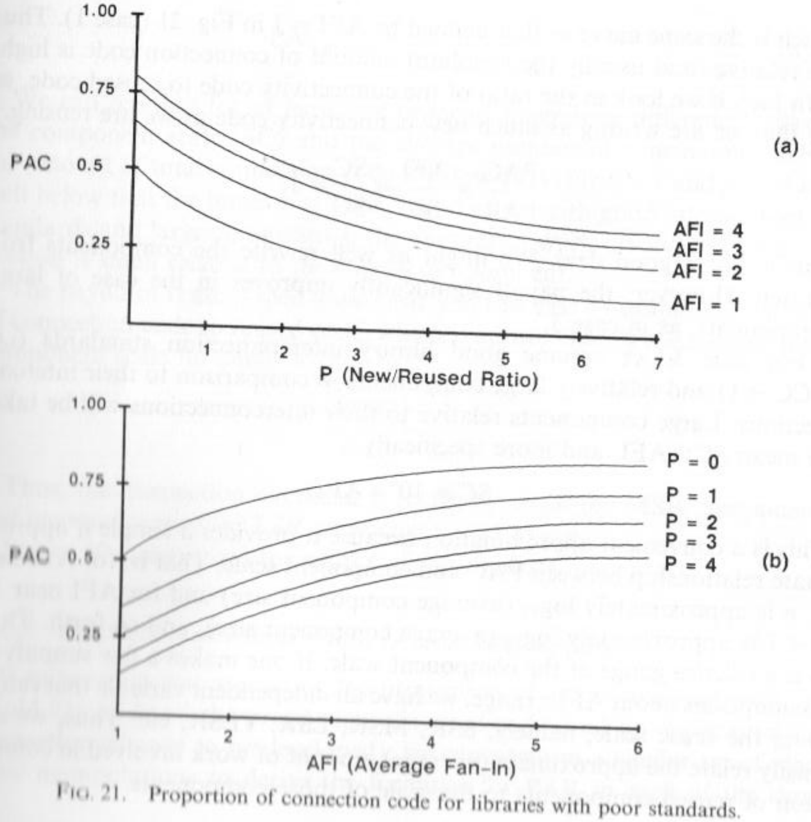


FIG. 21. Proportion of connection code for libraries with poor standards.

we see that since the fan-in must be at least 1, we always have to do at least as much work to connect the reused components as we would do to rewrite the reused code from scratch and if the fan-in is greater than 1, we have to do more. Admittedly, case 1 is a boundary case, but we must remember that there is a neighborhood around this case where reuse does not really payoff and one needs to structure their strategy to avoid this neighborhood.

Case 2 is a library with good standards but relatively small components. We define good standards to mean that ACC = 1. Thus, Eq. (4.9) becomes

$$\mathrm{PAC} = \frac{\mathrm{AFI}}{\mathrm{SC} * (1 + \mathrm{P}) + \mathrm{AFI}}. \tag{4.11}$$

We define small components to mean that SC = AFI, or in other words, the size of the connection network for a component is about the same as the size of a component. This produces

$$\mathrm{PAC} = \frac{1}{2 + \mathrm{P}} \tag{4.12}$$

which is the same curve as that defined by AFI = 1 in Fig. 21 (case 1). Thus, the relative (and usually the absolute) amount of connection code is high.

In fact, if we look at the ratio of the connectivity code to reused code, we see that we are writing as much new connectivity code as we are reusing.

$$\frac{\mathrm{PAC}}{\mathrm{PAR}} = \frac{\mathrm{AFI}}{\mathrm{SC}} = \frac{\mathrm{SC}}{\mathrm{SC}} = 1.$$

This is not a good deal. We might as well rewrite the components from scratch. However, the payoff significantly improves in the case of larger components, as in case 3.

For case 3, we assume good library interconnection standards (i.e., ACC = 1) and relatively large components in comparison to their interconnections. Large components relative to their interconnections will be taken to mean SC≫AFI, and more specifically

$$\mathrm{SC} = 10^n * \mathrm{AFI}.$$

This is a convenient approximation because it provides a simple if approximate relationship between PAC and component scale. That is, for AFI near 1, n is approximately $\log_{10}$ (average component size) and for AFI near 10, $n + 1$ is approximately $\log_{10}$ (average component size), and so forth. Thus, n is a *relative* gauge of the component scale. If one makes a few simplifying assumptions about AFI's range, we have an independent variable that ranges over the reuse scale, namely, SSR, MSR, LSR, VLSR, etc. Thus, we can easily relate the approximate (average) amount of work involved in connection of reused components to the scale of those components.

Using this approximation, Eq. (4.9) becomes

$$\mathrm{PAC} = \frac{\mathrm{AFI}}{10^n * \mathrm{AFI} * (1 + \mathrm{P}) + \mathrm{AFI}}.$$

Canceling out AFI, we get

$$\mathrm{PAC} = \frac{1}{10^n * (1 + \mathrm{P}) + 1}. \tag{4.13}$$

For $n = 1, 2, 3, \ldots$, we get

$$\mathrm{PAC}_{n=1} = \frac{1}{10 * \mathrm{P} + 11} \qquad \mathrm{PAC}_{n=2} = \frac{1}{100 * \mathrm{P} + 101}$$

$$\mathrm{PAC}_{n=3} = \frac{1}{1000 * \mathrm{P} + 1001}$$

and so forth. Thus, for $n > 0$,

$$\mathrm{PAC}_{\mathrm{approx}} = \frac{1}{10^n * (1 + \mathrm{P})}. \tag{4.14}$$

We can see that for at least one order of magnitude difference between the component scale (SC) and the average number of connections (AFI), the amount of total connection code is below 10% (for $n = 1$ and $p = 0$) and well below that for larger n's. Thus, for libraries with good interconnection standards and large components, the amount of work involved in interconnection is small relative to the overall development.

The payoff of reuse is seen quite clearly in this case by examining the ratio of connection code to reused code, which is approximately the inverse of the component scale for small AFI.

$$\frac{\mathrm{PAC}}{\mathrm{PAR}} = \frac{1}{10^n}.$$

Thus, the connection overhead is relatively small for MSR components and inconsequential for LSR components and above.

Figure 22 summarizes the results of this analysis.

4.2.3 Proportion of Reuse Code (Actual and Apparent)

If rather than just examining the proportion of interconnection code, we would like to know the proportion of reused code (and by implication the proportion of code to be developed), we can perform a similar set of algebraic manipulations to derive the formulas for PAR in each of the three

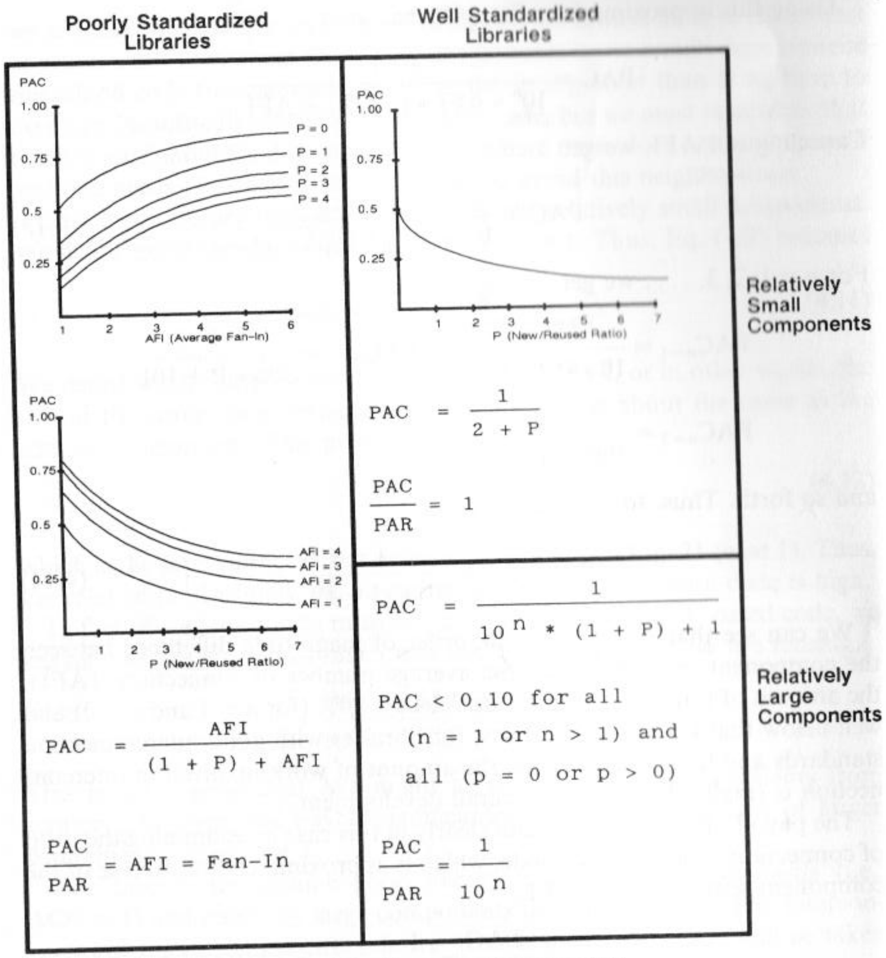


FIG. 22. Summary of case analysis

cases considered earlier. The results of these derivations are:

CASE 1: $$PAR = \frac{1}{(1 + P) + AFI}$$

CASE 2: $$PAR = \frac{1}{2 + P}$$

CASE 3: $$PAR = \frac{10^n}{10^n * (1 + P) + 1}.$$

The formula for case 3 is fairly complex to compute and it would be convenient to have a simpler approximation to PAR for case 3. The apparent proportion of application reuse (APAR) is a useful approximation. APAR is defined as

$$APAR = \frac{RLOC}{RLOC + NLOC}$$

which can be expressed as

$$APAR = \frac{1}{1 + P}.$$

In other words, APAR ignores the connection code, assuming it to be small. Obviously, this approximation only works for some situations. The question is, Under what circumstances is this a good approximation of PAR? Figure 23 shows the APAR curve in comparison with the PAR curves for case 2 and several parameterizations of case 1. It is clear from this figure that APAR is generally not a good approximation for either case 1 or case 2. However, for case 3, APAR is a pretty good approximation under most parameterizations. For $n >$ or $= 2$, the connectivity does not significantly alter the percent of reused code and APAR is a good approximation. For $n = 1$, the worst case is when $p = 0$, and even in this case, the difference is only about 0.08. The remaining integral values of p (greater than 0) differ by no more than 0.02. For $n = 0$, the formula reduces to case 2. This leads

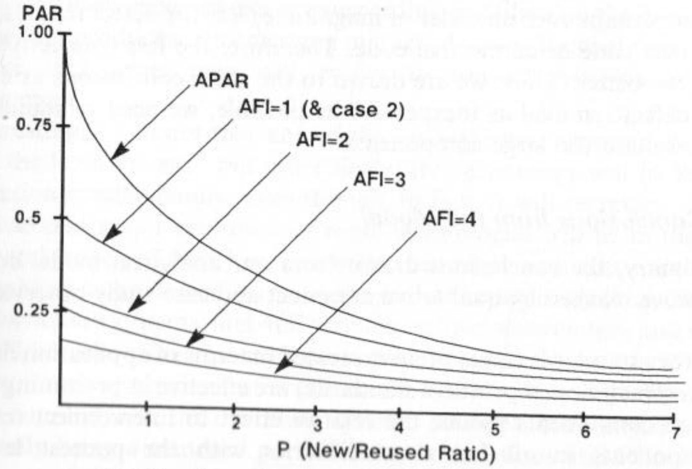


FIG. 23. Proportion of reuse code (apparent and real).

to the following rule of thumb:

> If on the average, the component scale (SC) is one or more orders of magnitude greater than AFI (the average interconnection fan-in) and the reuse library is well standardized (ACC is near 1), the connectivity code has no appreciable effect on the reuse proportions and APAR is a good approximation for PAR.

4.2.4 *Effects on Defect Removal*

In the previous sections, we have focused largely on the excessive plumbing costs that arise from poorly standardized libraries and small components. The analytical model also has cost avoidance implications with respect to defect removal that may be as great or greater than the cost avoidance that accrues from well-designed reuse regimes.

The important facts to note are:

- Since reused code has significantly fewer defects than new code, defect removal from reused code is usually significantly cheaper than from new code. It is not unusual for there to be anywhere from several times to an order of magnitude difference between these costs.
- Since connective code is new code, it will exhibit the higher defect rates and therefore, higher defect removal costs than reused code.

When considering the effects of reuse regimes on defect removal, the conclusions are the same as when considering the effects of reuse regimes on basic development, i.e., make the connective code be as small as possible, thereby making PAR as large as possible. Each line of reused code will cost several times (and perhaps even an order of magnitude) less for defect removal than a line of new code or connective code. Therefore, the less connective code we have, the better. Thus, we are drawn to the same conclusions as above: to make defect removal as inexpensive as possible, we need to standardize our libraries and use large components.

4.2.5 *Conclusions from the Model*

In summary, the conclusions drawn from our analytical model confirm those that we reached by qualitative argument and case study observations:

- Library standards (most often expressed in terms of application domain data structure and protocol standards) are effective in promoting reuse.
- Large components reduce the relative effort to interconnect reusable components in all but those libraries with the poorest level of standardization.

Therefore, the conclusion must be to develop large components (which tend toward domain specificity) and use a small set of (domain-specific) data structures and protocols across the whole library of components.

5. Futures and Conclusions

5.1 Futures

If I try to predict the future evolution of reuse, I see two major branches—vertical reuse and horizontal reuse—that break into several minor branches. In the vertical reuse branch, I see large-scale component kits becoming the "cut-and-paste" components for end-user application creation. That is, more and more applications will be constructed by using folders of facilities that are analogs of the *clip art* that is so widespread in today's desktop publishing. Of course, such end-user programming will have inherent limitations and therefore, will not replace professional programming, only change its span and focus.

The other major evolutionary branch within vertical programming evolution will be the maturation of application-specific reuse, which will evolve toward larger-scale components and narrower domains. This technology will be used largely by the professional programmer and will probably focus mostly on application families with a product orientation. Even though the productivity and quality improvements will be high, as with all vertical reuse technologies, the motivation in this case will be less a matter of productivity and quality improvement and more a matter of quick time to market. More and more software companies are succeeding or failing on the basis of being early with a product in an emerging market. As they discover that reuse will enhance that edge, they will evolve in toward reuse-based product development.

Interestingly, I doubt that any of the vertical reuse approaches will long retain the label "reuse," but more likely, the technology will be known by application specific names, even though, in fact, it will be reuse.

The second major evolution of reuse technologies will be in the area of horizontal reuse and here I see two major branches—systems enhancements and enabling technologies. As technologies like interface toolkits, user-oriented information systems, and 4GL-related technologies mature and stabilize, they will become more and more part of the operating system facilities. This is not so much a statement of architecture, in that they will probably not be tightly coupled with the operating systems facilities, but more a matter of commonly being a standard part of most workstations and PCs. In fact, a litmus test of the maturity of these technologies is the degree to which they are considered a standard and necessary part of a delivered computer. One can see this kind of phenomenon currently happening with the X windows system. Within 10 or so years, it will probably be difficult and unthinkable to buy a workstation or PC that does not have some kind of windowing interface delivered with it.

The other major branch of horizontal reuse is the set of reuse enabling technologies. More and more these technologies will merge into a single integrated facility. The object-oriented language systems and their associated development environments (i.e., the integrated debuggers, editors, profilers, etc.) will be integrated with the CASE tools such that the design and source code become an integral unit. The CASE tools themselves will be enhanced by designer/generator systems to allow them to do increasingly more of the work for the designer/programmer by using reuse technologies and libraries. Finally, I expect to see both the CASE tools and programming language development environments merge with reverse engineering, design recovery, and re-engineering tools and systems. These reverse engineering, design recovery, and re-engineering tools all support the population of reuse libraries as well as the analysis, understanding and maintenance of existing systems. Without such systems, the reuse libraries will largely be empty and the technology impotent. These are the systems that allow an even more primitive kind of reuse, that of bootstrapping previous experience into formal reusable libraries and generalized reusable know-how.

Thus, while horizontal reuse and vertical reuse will evolve along different paths, both will move from independent tool sets to integrated facilities and consequently their leverage will be amplified.

5.2 Conclusions

There are no silver bullets in software engineering, and reuse is not one either, although it may come as close as anything available today. While not a silver bullet or cure-all, it does provide many opportunities for significant improvements to software development productivity and quality within certain well-defined contexts. If one understands where its works well and why, it can be a powerful tool in one's arsenal of software development tools and techniques.

REFERENCES

Arango, G. (1988). Domain Engineering for Software Reuse, Ph.D. dissertation, University of California at Ivine.

Batory, D. S. (1988). Concepts for a Database System Compiler, ACM PODS.

Batory, D. S., Barnett, J. R., Roy, J., Twichell, B. C., and Garza, J. (1989). Construction of File Management Systems from Software Components. COMPSAC.

Bigelow, J., and Riley, V. (1987). Manipulating Source Code in Dynamic Design. HyperText '87 papers.

Bigelow, J. (1988). Hypertext and CASE. *IEEE Software* **21**(3), 23–27.

Biggerstaff, T. J., and Perlis, A. J., eds (1984). Special Issue on Reusability. *IEEE Transactions on Software Engineering*, **SE-10**(5).

Biggerstaff, T. J. (1987). Hypermedia as a Tool to Aid Large-Scale Reuse. MCC Technical report STP-202-87; also published in "Workshop on Software Reuse," Boulder, Colorado.

Biggerstaff, T. J., and Richter, C. (1987). Reusability Framework, Assessment, and Directions. *IEEE Software*.

Biggerstaff, T. J., and Perlis, A. J., eds (1989). "Software Reusability" (two volumes). Addison-Wesley/ACM Press.

Biggerstaff, T. J. (1989). Design Recovery for Maintenance and Reuse, *IEEE Computer*.

Biggerstaff, T. J., Hoskins, J., and Webster, D. (1989). DESIRE: A System for Design Recovery. MCC Technical Report STP-021-89.

Brachman, R. J., and Schmolze, J. G. (1985). An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* **9**, 171–216.

Brooks, F. P. (1989). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* **22**(7).

Chikofsky, E. J. ed. (1988). Special Issue on Computer Aided Software Engineering. *IEEE Software*.

Chikofsky, E. J., ed. (1989). Computer-Aided Software Engineering. *IEEE Computer Society Press Technology Series*.

Conklin, J. (1987). Hypertext: An Introduction and Survey. *IEEE Computer*.

Cox, B. (1986). "Object-Oriented Programming: An Evolutionary Approach." Addison-Wesley.

Cross, J. H., II, Chikofsky, J., and May, C. H., Jr. (1992). Reverse Engineering. In "Advances in Computers," Vol. 35 (Marshall Yovitz, Ed.) Academic Press, Boston.

Cusumano, M. A. (1989). The Software Factory: A Historical Interpretation. *IEEE Software*.

Cusumano, M. A. (1991). "Japan's Software Factories: A Challenge to U.S. Management." Oxford University Press.

Ellis, M. A., and Stroustrup, B. (1990). "The Annotated C++ Reference Manual." Addison-Wesley.

Fikes, R., and Kehler, T. (1985). The Role of Frame-Based Representation in Reasoning. *Communications of the ACM*, **28**(9).

Finin, T. (1986a). Understanding Frame Languages (Part 1). *AI Expert*.

Finin, T. (1986b). Understanding Frame Languages (Part 2). *AI Expert*.

Fisher, A. S. (1988). "CASE: Using Software Development Tools." Wiley.

Freeman, P. (1987). Tutorial on Reusable Software Engineering. *IEEE Computer Society Tutorial*.

Goldberg, A., and Robson, D. (1983). "Smalltalk-80: The Language and Its Implementation." Addison-Wesley.

Gregory, W., and Wojtkowski, W. (1990). "Applications Software Programming with Fourth-Generation Languages." Boyd and Fraser Publishing, Boston.

Gullichsen, E., D'Souza, D., Lincoln, P., and The, K.-S. (1988). The PlaneTextBook. MCC Technical Report STP-333-86 (republished as STP-206-88).

Heller, D. (1990). "Xview Programming Manual." O'Reilly and Associates, Inc.

Hinckley, K. (1989). The OSF Windowing System. *Dr. Dobbs Journal*.

Horowitz, E., Kemper, A., and Narasimhan, B. (1985). A Survey of Applications Generators. *IEEE Software*.

Kant, E. (1985). Understanding and Automating Algorithm Design. *IEEE Transactions on Software Engineering* **SE-11**(11).

Kim, W., and Lechovsky, F. H. eds. (1989). 'Object-Oriented Concepts, Databases, and Applications," Addison-Wesley/ACM Press.

Lubars, M. D. (1987). Wide-Spectrum Support for Software Reusability. MCC Technical Report STP-276-87, (1987) also published in "Workshop on Software Reuse," Boulder, Colorado.

Lubars, M. D. (1990). The ROSE-2 Strategies for Supporting High-Level Software Design Reuse. MCC Technical Report STP-303-90, (to appear). Also to appear in a slightly modified form in M. Lowry and R. McCartney, eds., "Automating Software Design," under the title, Software Reuse and Refinement in the IDEA and ROSE Systems. AAAI Press.

Lubars, M. D. (1991). Reusing Designs for Rapid Application Development. MCC Technical Report STP-RU-045-91.

Martin, J. (1985). "Fourth-Generation Languages: Volume II. Principles." Prentice-Hall.

Martin, J., and Leben, J. (1986a). "Fourth-Generation Languages—Volume II. Representative 4GLs." Prentice-Hall.

Martin, J., and Leben, J. (1986b). "Fourth-Generation Languages—Volume III. 4GLs from IBM." Prentice-Hall.

Matos, V. M., and Jalics, P. J. (1989). An Experimental Analysis of the Performance of Fourth Generation Tools on PCs. *Communications of the ACM* **32**(11).

Matsumoto, Y. (1989). Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. In "Software Reusability" (T. J. Biggerstaff and A. Perlis, eds.). Addison-Wesley/ACM Press.

Meyer, B. (1988). "Object-Oriented Software Construction." Prentice-Hall.

Neighbors, J. M. (1987). The Structure of Large Systems. Unpublished presentation, Irvine, California.

Norman, R. J., and Nunamaker, J. F., Jr. (1989). CASE Productivity Perceptions of Software Engineering Professionals. *Communications of the ACM* **32**(9).

Nye, A. (1988). "Xlib Programming Manual." O'Reilly and Associates, Inc.

Nye, A., and O'Reilly, T. (1990). "X Toolkit Intrinsics Programming Manual." O'Reilly and Associates, Inc.

Parker, T., and Powell, J. (May 1989). Tools for Building Interfaces. *Computer Language*.

Pressman, R. S. (1987). "Software Engineering: A Practitioner's Approach—2nd Ed." McGraw-Hill.

Prieto-Diaz, R. (1989). Classification of Reusable Modules. In "Software Reusability—Volume I" (T. J. Biggerstaff and A. Perlis, eds.). Addison-Wesley.

Rich, C., and Waters, R. (1989). Formalizing Reusable Components in the Programmer's Apprentice. In "Software Reusability" (T. J. Biggerstaff and A. Perlis, eds.). Addison-Wesley/ACM Press.

Rowe, L. A., and Shoens, K. A. (1983). Programming Language Constructs for Screen Definition. *IEEE Transactions on Software Engineering*, **SE-9**(1).

Saunders, J. H. (March/April 1989). A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*.

Scheifler, R. W., Gettys, J., and Newman, R. (1988). "X Windowing System: C Library and Protocol Reference." Digital Press.

Selby, R. W. (1989). Quantitative Studies of Software Reuse. In "Software Reusability" (T. J. Biggerstaff and A. Perlis, eds.). Addison-Wesley/ACM Press.

Smith, J. B., and Weiss, S. F. eds. (1988). Special Issue on Hypertext. *Communications of the ACM* **31**(7).

Stroupstrup, B. (1986). "The C++ Programming Language." Addison-Wesley.

Stroupstrup, B. (May 1988). What is Object-Oriented Programming? *IEEE Software* 10–20.

Sun Microsystems Corporation (1990). "OpenWindows Developer's Guide 1.1 User Manual." Sun Microsystems.

Tracz, W. ed. (July 1987). Special Issue on Reusability. *IEEE Software*.

Tracz, W. ed. (July 1988). Tutorial on Software Reuse: Emerging Technology. *IEEE Computer Society Tutorial*.

Wartik, S. P., and Penedo, M. H. (March 1986). Fillin: A Reusable Tool for Form-Oriented Software. *IEEE Software*.

Weide, B. W., Ogden, W F., and Zweben, S. H. (1991). Reusable Software Components. In "Advances in Computers" (M. C. Yovits, ed.)

Xerox Corporation (1979). "Alto User's Handbook." Xerox Palo Alto Research Center, Palo Alto, California.

Xerox Corporation (1981). "8010 Star Information System Reference Guide." Dallas, Texas.

Young, D. A. (1989). "X Window Systems Programming and Applications with Xt." Prentice-Hall.