VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

No Name: Just Notes on Software Reuse

Robert Biddle, Angela Martin, and James Noble

Technical Report CS-TR-03/11
July 2003

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

No Name: Just Notes on Software Reuse

Robert Biddle, Angela Martin, and James Noble

Technical Report CS-TR-03/11
July 2003

Abstract

In the beginning, so our myths and stories tell us, the programmer created the program from the eternal nothingness of the void. In this essay, we recognise that programs these days are like any other assemblage, and suggest that in fact programming has always been about reuse. We also explore the nature of reuse, and claim that Components themselves are not the most important consideration for reuse; it is the end product, the composition. The issues still involve value, investment, and return. But pervasive reuse promotes a change in the method of construction of the program, and in the program itself.

Publishing Information

A version of this paper is to be presented as part of the Onward! stream at ACM Oopsla 2003, in Anaheim California, USA.

Author Information

Robert Biddle and James Noble are members of the academic staff in the School of Mathematical and Computing Sciences and the School of Information Management at Victoria University of Wellington. Angela Martin is a PhD candidate at Victoria University of Wellington.

No Name: Just Notes on Software Reuse

Robert Biddle, Angela Martin, James Noble
Victoria University of Wellington, New Zealand.
{robert,angela,kjx}@mcs.vuw.ac.nz

ABSTRACT

In the beginning, so our myths and stories tell us, the programmer created the program from the eternal nothingness of the void. In this essay, we recognise that programs these days are like any other assemblage, and suggest that in fact programming has always been about reuse. We also explore the nature of reuse, and claim that Components themselves are not the most important consideration for reuse; it is the end product, the composition. The issues still involve value, investment, and return. But pervasive reuse promotes a change in the method of construction of the program, and in the program itself.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Reuse models

General Terms

Design

Keywords

Software Reuse, Components, Object-Oriented Programming

Figure 1: Name Story — This essay concerns components, reuse, value, and beauty. All photographs were taken at No Name Building Recyclers, Wellington, New Zealand, with their kind permission.

1. PROGRAMMING

In the beginning, so our myths and stories tell us, the programmer created the program from the eternal nothingness of the void. Whether it is Stallman typing teco macros and wearing out the shift keys; Chuck Moore typing backwards at Kitt Peak; Goldberg, Deutsch, Robson et. al. in the parclands of California; billg hunched over Allen's Altair emulator; Bill Joy's VAX crashing and deleting vi multibuffer support for the next ten years; Gabriel doing *The Right Thing* at Lucid; or Larry Wall doing whatever: 'tsall good.

All of our subtribes have programmer heros: wearing tramping boots in the machine room. Mythic figures who bestrode the earth in the beginning. But we are no longer in the beginning, and programmers no longer write programs like the Don Knuth of our fables — typing all the source of TEX from memory into a single teletype, compiling without any syntax errors, printing *The Art of Computer Programming* from that first binary.

Programs these days are like any other assemblage — films, language, music, art, architecture, writing, academic papers even — a careful collection of preexisting and new components (if we had the money and could afford them) pieced together not to tell some overwhelming story, but to get the job done, meet the deadline, get paid and go home.

The goals, the stepwise refinements, the structure charts, the Unified Modelling Language diagrams, the problems, the solutions — although they may be obvious at the start — soon splinter and fragment because the program is no longer one grand design, but a whole series of little stories, user stories even, little parts, pieces, components, objects, classes, files, modules, load-time objects, runtime checks, aspects, subjects, packages, assemblies, bat files, dot files, app files, some large, some small, something old, something new, something borrowed, something blue, some typed, some generated, some stolen, some bought, some crufted together automatically by emacs macros, some downloaded from Google, some from an expensive framework library given away free with a box of breakfast cereal, some larger, some smaller, and the only commonality being their lack of commonality, lack of anything that marks them as being part of the program except the indisputable fact that if you delete any one of them the program might stop working.

Number One: Every Number One song ever written is only made up from bits from other songs. There is no lost chord. No changes untried. No extra notes to the scale or hidden beats to the bar. There is no point in searching for originality.
—Jimmy Caulty and Bill Drummond, *The Manual* [5]

---

2. KNUTH

The most important lesson I learned during that past nine years is that *software is hard*; and it takes a long time. From now on I shall have significantly greater respect for every successful software tool that I encounter. My original plan was to spend one year working on algorithms for typography; but that was in the spring of 1977, and I soon found that much more work would be needed to finish the job. Even so, if my health continues to be good, I think it will turn out that those nine years were not wasted, because they will have improved my efficiency enough that I'll be able to recover the time during the next decade or so. Most importantly, those nine years were surely not wasted, because I learned an enormous number of things that will "feed" the theoretical work that I do in the future.

...

The amount of technical detail in a large system is one thing that makes programming more demanding than book-writing. Another is that programming demands a significantly higher standard of accuracy. Programs don't simply have to make sense to another human being, they must make sense to a computer. For example, I write the entire TEX compiler and desk-checked it before I did any debugging on a machine. At that point I had in my hands a document of some 500 pages, containing the program and an informal proof of its correctness. If I had submitted the program to human referees for verification, they would presumably have found a few problems that could readily be fixed, after which I might have published my program as a theoretical demonstration that "there exists a way to compile TEX." But of course the computer was a much sterner taskmaster than any human referees would be; therefore I had to spend another five months of intense activity before my program actually ran well enough for me to believe that it did the right things.

(Chapter 9, Theory and Practice III)

One of the main reasons I've chosen to speak about Theory and Practice this morning is that I've spent the past 12 years working on a project that has given me an unusual opportunity to observe how theory and practice support each other.

...

What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD. From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to find that the writing of programs for TEX and for METAFONT proved to be much more difficult than all the other things I had done (like proving theorems or writing books).

(Chapter 10, Theory and Practice IV)

— Donald E. Knuth, *Selected Papers on Computer Science* [16]

Figure 2: Sink Story — These components are easily recognisable, but note they are kept in pairs because users typically want to acquire matched pairs.

Figure 3: Sink Story — Other components are less recognisable, but play important roles in creating larger assemblies, and knowledgeable users may combine them as appropriate.

101-ISM: The tendency to pick apart, often in minute detail, all aspects of life using half-understood pop psychology as a tool.
— Douglas Coupland, *Generation X* [9]

3. CONSTANTINE

We recycle so many things, from grocery bags to toner cartridges, why not recycle code? Why not reuse designs and models rather than always starting from scratch? The rewards of reuse seem to be enormous. What code is cheaper to write than code you don't have to write at all? With higher levels of reuse supported by larger component libraries, we might double or triple effective productivity. All we have to do is change the whole culture of software development and maybe the personalities of programmers.

Reuse is hardly a new idea. The lowly subroutine was conceived so that the same instructions did not have to be written out each time a particular calculation was needed.

Then, what is the problem? Unfortunately, most programmers like to program. Some of them would rather program than eat or bathe. Most of them would much rather cut code than chase documentation or search catalogs or try to figure out some other stupid programmer's idiotic work. Software developers develop things; users use them. Other things being equal, programmers design and build from scratch rather than recycle.

...

Reusable component libraries have been around for almost as long as people have been programing. The first to yield to reuse were math routines, followed soon by input-output. Except for the sheer joy or perversity of doing it, no applications or tool developer writes their own sine-cosine routines anymore.

. . .

If it takes the typical programmer more than 2 minutes and 27 seconds to find something, they will conclude it does not exist and therefore will reinvent it.

. . .

The payoff is finding the component you need in a reasonable time and then finding that it can be readily used or adapted for your use. Every time this happens, you are being reinforced for it. The habit of initially consulting the library or repository becomes ingrained without being tied to increasing the number of green stamps you get or your quarterly code bonus.

— Larry L. Constantine, *The Peopleware Papers: Notes on the Human Side of Software* [6](pp. 141–145)

4. COOPER

The primary side effect of code reuse is that large portions of most programs exist not because some interaction designer wanted them to exist, but because some other programmer already did the work on someone else's budget. Much of the software that we interact with exists for the sole reason that it existed before.

. . .

A fascinating aspect of the imperative to reuse code is the willingness with which programmers will adopt code with a questionable pedigree. Some inexperienced programmer will hack out the first interaction idea that pops into his head, but once it is written, that piece of code becomes the basis for all subsequent efforts because it is so aggressively reused.

In Windows, for example, the really experienced programmers built the internal processing of the operating system, but the first sample applications that showed third-party developers how to communicate with the user were written by a succession of summer interns and junior coders at Microsoft. The Windows internal code has been upgraded and rewritten over six major releases, and it has steadily improved. However, an embarrassingly large number of popular applications have in their hearts long passages of program code written by 21-year-old undergraduates spending a summer in Redmond. The same is true for the Web. Amateur experimenters hacked out the first Web sites, but those who followed cloned those first sites, and their sites were cloned in turn.

As you can see, there is a clear conflict of interest between what the user needs and what the programmer needs. We anticipate conflict of interest in countless activities and professions, and we have built in safeguards to curb its influence. While judges and lawyers have skills in common, we never let lawyers adjudicate their own cases. We never let basketball players referee their own basketball games. The conflicting interests are clearly visible, yet we consistently let

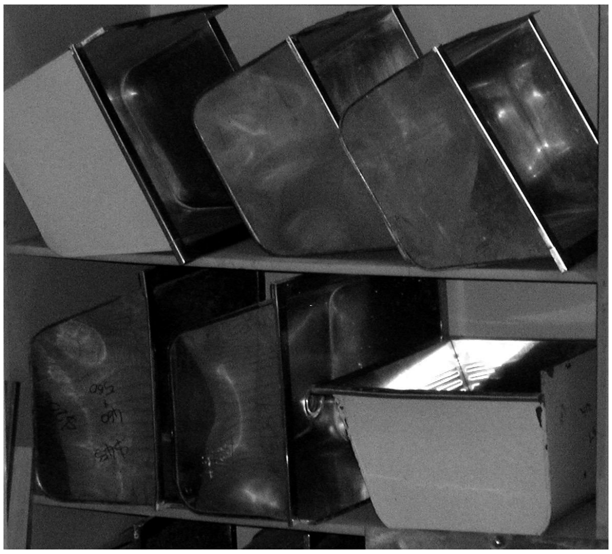

Figure 4: Sink Story — Large components may be acquired separately without any related components included.

programmers make design decisions based on personal implementation considerations.

— Alan Cooper, *The Inmates are Running the Asylum* [7](pp.106–108)

5. GATES

INTERVIEWER: In a company like Microsoft, where you have 160 programmers, how do you go about creating an environment where you can develop successful programs?

GATES: One way is to have small project teams, typically four or five people, and one of those people has to have the proven ability to really absorb a program. And when that lead person is uncertain about something, he or she should be able to discuss it with even more experienced programmers.

Part of our strategy is getting the programmers to think everything through before they go to the coding phase. Writing the design documents is crucial, because a lot of simplification comes when you see problems expressed as algorithms. They're kind of in the smallest form then, where you can see what the overlap is.

Another important element is code review, making sure you look through the code and see if senior people can provide hints about how to do something better. And you have to review similar projects that have gone super, super, well; programmers can look at how those other people performed previously, and get ideas from other projects about how to improve their own program.

— Bill Gates interviewed by Susan Lammers, *Programmers at Work* [18](p. 74)

Figure 5: Sink Story — Alternatively, large components may be acquired with additional matching components included and already assembled.

Figure 6: Sink Story — Entire packages of components are also available with all related components included and assembled, ready for installation.

CANNIBALIZE 1. To take salvageable parts from (as a disabled machine) for use in building or repairing another machine. . . . 3. To use or draw on material of (as another writer or an earlier work) [a volume . . . that not only ∼ previous publications but is intended itself to be cannibalized — R.M. Adams]. 4. To make use of (a part take from one thing) in building or repairing something else.

Webster's Ninth New Collegiate Dictionary (Springfield, Mass.: Merriam-Webster, 1990). Quoted by Rem Koolhaas and Bruce Mau, in *S,M,L,XL* [17]

---

6. LANGUAGE

Programming language has always been about reuse.

A computer programme, like a concert programme, might consist of a single sequence. But from the earliest experience of programming, we made up language to make programming easier by supporting reuse.

Selection and iteration allow instructions to be reused, executed again without re-specification, subject to varying conditions. A stored program computer allows a computer program to be specified once, and then stored for later use and reuse.

Procedures allow a body of instructions to be named and then reused from different contexts. Procedure parameters allow a body of instructions to be customised to suit the needs of differing calling contexts. Procedural encapsulation allows procedures to be reused without regard to their implementation.

Records allow data structures to be named and then reused to specify different data with the same structure.

Classes and objects allow data structures to be bound with procedures that use them, so that both data and procedures can be reused together. Encapsulation of classes and objects means they can be reused according to their specification, and without regard to their implementation.

Figure 7: Window Story — Large composite components are available ready to install, though some adjustments may be necessary to achieve the desired result.

Types allow checking the specification of names and their later use and reuse to determine if the use or reuse is appropriate.

Late binding of procedure names allows calling contexts to be reused by calling different bodies of instructions in different circumstances. Late binding of class or object names allows the contextual frameworks to be reused by working data structure with different implementation and behaviour.

And macros, and generators, and aspects, and so on. Programming language has always been about reuse; programs work in this way. Software wants to be reused.

7. LOVELACE

> Those who may desire to study the principles of the Jacquard-loom in the most effectual manner, viz. that of practical observation, have only to step into the Adelaide Gallery or the Polytechnic Institution. In each of these valuable repositories of scientific illustration, a weaver is constantly working at a Jacquard-loom, and is ready to give any information that may be desired as to the construction and modes of acting of his apparatus. The volume on the manufacture of silk, in Lardner's Cyclopædia, contains a chapter on the Jacquard-loom, which may also be consulted with advantage.
>
> The mode of application of the cards, as hitherto used in the art of weaving, was not found, however, to be sufficiently powerful for all the simplifications which it was desirable to attain in such varied and complicated processes as those required in order to fulfil the

Figure 8: Window Story — Also, smaller sub-components are available for assembly into composite structures. Note that these components are presented in a way so as to assist the user in locating and selecting the correct item.

purposes of an Analytical Engine. A method was devised of what was technically designated backing the cards in certain groups according to certain laws. The object of this extension is to secure the possibility of bringing any particular card or set of cards into use any number of times successively in the solution of one problem. Whether this power shall be taken advantage of or not, in each particular instance, will depend on the nature of the operations which the problem under consideration may require. The process is alluded to by M. Menabrea, and it is a very important simplification. It has been proposed to use it for the reciprocal benefit of that art, which, while it has itself no apparent connexion with the domains of abstract science, has yet proved so valuable to the latter, in suggesting the principles which, in their new and singular field of application, seem likely to place algebraical combinations not less completely within the province of mechanism, than are all those varied intricacies of which intersecting threads are susceptible. By the introduction of the system of backing into the Jacquard-loom itself, patterns which should possess symmetry, and follow regular laws of any extent, might be woven by means of comparatively few cards.

Those who understand the mechanism of this loom will perceive that the above improvement is easily effected in practice, by causing the prism over which the train of pattern-cards is suspended to revolve backwards instead of forwards, at pleasure, under the requisite circumstances; until, by so doing, any particular card, or set of cards, that has done duty once, and passed on in the ordinary regular succession, is brought back to the position it occupied just before it was used the preceding time. The prism then resumes its forward rotation, and thus brings the card or set of cards in question into play a second time. This process may obviously be repeated any number of times.

— Augusta Ada King, Countess of Lovelace, *Annotated translation of Sketch of the Analytical Engine Invented by Charles Babbage, by L. F. Menabrea of Turin, Officer of the Military Engineers* [8](Note C)

Figure 9: Window Story — The repository also stores reusable elements needed to built components. These are typically not available to users, but are recovered and retained by the repository management to facilitate creation of components suitable for users.

NOW DENIAL: To tell oneself that the only time worth living in is the past, and that the only time that may ever be interesting again is the future.
— Douglas Coupland, *Generation X* [9]

8. TURING

Outline of Logical Control

We also wish to be able to arrange for the splitting up of operations into subsidiary operations. This should be done in such a way that once we have written down how an operation is done we can use it as a subsidiary to any other operation.

. . .

When we wish to start on a subsidiary operation we need only make a note of where we left off the major operation and then apply the first instruction of the subsidiary. When the subsidiary is over we look up the note and continue with the major operation. Each subsidiary operation can end with instructions for this recovery of the note. How is the burying and disinterring of the note to be done? There are of course many ways. One is to keep a list of these notes in one or more standard size delay lines (1024), with the most recent last. The position of the most recent of these will be kept in a fixed TS, and this reference will be modified every time a subsidiary is started or finished. The burying and disinterring processes are fairly elaborate, but there is fortunately no need to repeat the instructions involved, each time, the burying being done through a standard instruction table BURY, and the disinterring by the table UNBURY.

— Alan Turing, *Proposed Electronic Calculator* [34]

9. WILKES, WHEELER, AND GILL

Closed Subroutines

A "closed" subroutine is one which is called into use by a special group of orders incorporated in the master routine or main program. It is designed so that when its task is finished it returns control to the master routine at a point immediately following from that which it was called in.

. . .

The library catalog used in the Laboratory is drawn up in two sections. One gives a concise specification of the purpose of each subroutine together with sufficient information to enable a programmer to make use of it; this includes information about the operating time and storage space occupied. The second section gives the orders of each subroutine in full. The catalog is contained in loose leaf books so that new sheets can be inserted as new subroutines are added to the library.

— Maurice V. Wilkes, David J. Wheeler and Stanley Gill, *The Preparations of Programs for An Electronic Computer: With special reference to the EDSAC and the use of a library of subroutines* [37]

As soon as the EDSAC began to work I called a meeting of those interested in development of programming methods — it would have been premature to call them programmers — and we constituted ourselves into a committee to establish a library of such subroutines in order that every user should not have to start from the bottom. At first we thought of the library as containing subroutines of the type just mentioned and subroutines for the computation of elementary functions; later it became clear that it could be expanded in various directions, notably by inclusion of subroutines for performing some of the standard operations of numerical analysis. It was so clear to me that we should base our system of programming on a library of subroutines that I was somewhat surprised a few years later to find that not everyone had gone this way.

— Maurice V. Wilkes, *Memoirs of a Computer Pioneer* [36]

10. NYGAARD AND DAHL

The syntax for this new language feature was easy to find. The "links" could be declared separately, without any information about the other process classes which use link instances as a prefix layer. Since the processes of these other process classes were at the same time both "links" and something more, it was natural to indicate this by textually prefixing their declarations with the process class identifier of this common property, namely "link". These process classes would the be "subclasses" of "link".

It was evident that when prefixing was introduced, it could be extended to multiple prefixing, establishing hierarchies of process classes. (In the example, "car" would be a subclass of "link", "truck" and "bus" subclasses of "car".) It was also evident that this "concatenation" of a sequence of prefixes with a main part could well be applied to the action parts of processes as well.

Usually a new idea was subjected to rather violent attacks in order to test its strength. The prefix idea was the only exception. We immediately realised that we now had the necessary foundation for a completely new language approach, and in the days which followed the discovery we decided that:

1. We would design a new general programming language, in terms of which an improved SIMULA I could be expressed.
2. The basic concept should be *classes* of *objects*.
3. The prefix feature, and thus the subclass concept, should be a part of the language.
4. Direct, qualified references should be introduced.

— Kristen Nygaard and Ole-Johan Dahl, *The development of the SIMULA languages* [25]

Figure 10: Window Story — The repository will use recovered components to assemble composite components suitable for users. Sometimes the repository will also incorporate new components as well.

Figure 11: Window Story — Complete composite components are offered ready for users to install. Note that such components may have important and distinctive features, and so they are displayed prominently to make these features evident to potential users.

11. PARNAS

> In discussions of system structure it is easy to confuse the benefits of a good decomposition with those of a hierarchical structure. We have a hierarchical structure if a certain relation may be defined between the modules or programs and that relation is a partial ordering. The relation we are concerned with is "uses" or "depends upon." It is better to use a relation between programs since in many cases one module depends upon only part of another module (. . .). It is conceivable that we could obtain the benefits that we have been discussing without such a partial ordering, e.g. if all the modules were on the same level. The partial ordering gives us two additional benefits. First, parts of the system are benefited (simplified) because they use the services of lower levels. Second, we are able to cut off the upper levels and still have a usable and useful product. For example, the symbol table can be used in other applications; the line holder could be the basis of a question answering system. The existence of the hierarchical structure assures us that we can "prune" off the upper levels of the tree and start a new tree on the old trunk. If we had designed a system in which the "low level" modules made some use of the "high level" modules, we would not have the hierarchy, we would find it much harder to remove portions of the system, and "level" would not have much meaning in the system.
>
> . . .
>
> We have tried to demonstrate by these examples that it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are

likely to change. Each module is then designed to hide such a decision from the other. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules.

— David Lorge Parnas, *On the criteria to be used in decomposing systems into modules* [26]

12. LISKOV AND ZILLES

What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details. In the case of programming, the use which may be made of an abstraction is relevant; the way in which the abstraction is implemented is irrelevant. If we consider conventional programming languages, we discover that they offer a powerful aid to abstraction: the function or procedure. When a programmer makes use of a procedure, he is (or should be) concerned only with what it does — what function it provides for him. He is not concerned with the algorithm executed by the procedure. In addition, procedures provide a means of decomposing a problem — performing part of the programming task inside a procedure, and another part in the program which calls the procedure. Thus, the existence of procedures goes quite far toward capturing the meaning of abstraction.

Unfortunately, procedures alone do not provide a sufficiently rich vocabulary of abstractions. The abstract data objects and control structures of the abstract machine mentioned above are not accurately represented by independent procedures. Because we are considering abstraction in the context of structured programming, we will omit discussion of control abstractions.

This leads us to the concept of abstract data type which is central of the design of the language. An *abstract data type* defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type.

We believe that the above concept captures the fundamental properties of abstract objects. When a programmer makes use of an abstract data object, he is concerned only with the behaviour which that object exhibits but not with any details of how that behaviour is achieved by means of an implementation. The behaviour of an object is captured by the set of characterizing operations. Implementation information, such as how the object is represented in storage, is only needed when defining how the characterizing operations are to be implemented. The user of the object is not required to know or supply this information.

— Barbara Liskov and Stephen Zilles, *Programming with abstract data types*, [19]

13. JOHNSON AND FOOTE

One of the most important kinds of reuse is reuse of designs. A collection of abstract classes can be used to express an abstract design. The design of a program is usually described in terms of the program's components and the way they interact. For example, a compiler can be described as consisting of a lexer, a parser, a symbol table, a type checker, and a code generator. An object-oriented abstract design, also called a framework, consists of an abstract class for each major component. The interfaces between the components of the design are defined in terms of sets of messages. There will usually be a library of subclasses that can be used as components in the design.

...

Frameworks are useful for reusing more than just mainline application code. They can also describe the abstract designs of library components. The ability of frameworks to allow the extension of existing library components is one of their principal strengths. Frameworks are more than well written class libraries. A good example of a set of library utility class definitions is the Smalltalk Collection hierarchy. These classes provide ways of manipulating collections of objects such as Arrays, Dictionaries, Sets, Bags, and the like. In a sense, these tools correspond to the sorts of tools one might find in the support library for a conventional programming system. Each component in such a library can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent. A framework, on the other hand, is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific. Frameworks can be built on top of other frameworks by sharing abstract classes.

...

Frameworks provide a way of reusing code that is resistant to more conventional reuse attempts. Application independent components can be reused rather easily, but reusing the edifice that ties the components together is usually possible only by copying and editing it. Unlike skeleton programs, which is the conventional approach to reusing this kind of code, frameworks make it easy to ensure the consistency of all components under changing requirements. Since frameworks provide for reuse at the largest granularity, it is no surprise that a good framework is more difficult to design than a good abstract class. Frameworks tend to be application specific, to interlock with other frameworks by sharing abstract classes, and to contain some abstract classes that are specialized for the framework. Designing a framework requires a great deal of experience and experimentation, just like designing its component abstract classes.

— Ralph E. Johnson and Brian Foote, *Designing reusable classes* [14]

Figure 12: Door Story — Most components are presented in categories so as to enable the user to locate desired components quickly. The common categories are firstly usage, and then style.

14. COMPOSED SOFTWARE

> *"One thing can be stated with certainty: components are for composition."*
>
> — Clemens Szyperski, Introduction to *Component Software — Beyond Object-Oriented Programming* [32]

This is true: components are certainly for composition; they may even be the best things to compose; but they are not the only things that can be composed; and they are not the only things to consider when thinking about software reuse.

Components themselves are not the most important consideration for reuse; nor is the process of composing things together the most important consideration: rather, it is that the end product is a *composition*, that is, something that is made up of many things, not just one thing.

But 'Components' are not the only things that are composed. By 'Components' (with the capital 'C' and the inverted commas) we mean modern, object-oriented/component-oriented units of software that have been specially designed to be composed, that ideally are specified, abstracted, reliable, trustable, binary, marketable, and above all, substitutable, *fungible*.

Focusing on software compositions — on the *Composed Software* that is the end product of a composition process — shows that it is made up of more than just these idealised 'Components'. Some of the software will be old software made new, born again, changed and broken to become part of a new creation. Some of the software will be freshly written to provide important new behaviour or meet critical quality requirements. Some of the software will be mundane, egoless, glue code, written or stolen or grown to tie other pieces together but providing no behaviour or qualities of its own. And, yes, some of the software will be shiny, new, modern, encapsulated components, purchased from a perfect-bound web page catalogue, printed on glossy paper decorated with happy pictures of imaginary programmers from central casting, cavorting on their days off.

In the traditions of Saussure, structural linguistics, semiology, semiotics, we can consider artifacts to be constructed along two dimensions or axes: the axis of combination (technically the syntagm), and the axis of selection or substitution (technically the paradigm). The axis of selection gives a menu of choices — components that can be made visible in catalogues — that can be consulted during the composition process. The axis of combination gives the structure of the end product — the composed software in this case — but is visible only when the composition process is complete.

Traditional software componentry falls on the paradigmatic axis: programmers select components to compose into their programs. In Visual Basic, for example, the VBX market included many small vendors selling slightly different versions of scrolling table controls: a programmer could choose one, incorporate it into the program, and substitute a different component should the first prove inappropriate.

On the other hand, less traditional (but more successful) forms of software components, such as software frameworks, programming languages, and middleware infrastructures, fall on the syntagmatic axis. Considering Visual Basic itself, the Microsoft Foundation Classes, J2EE, each of these frameworks establishes the form of a single syntagm — all the programs built using these frameworks have the same underlying structure. This kind of reuse has been classified as *context reuse*, to distinguish it from the traditional *component reuse* technologies [3]. Within such a framework, developers can paradigmatically substitute their own components,

Figure 13: Door Story — Some components are distinguished because they are superior in quality and distinctive in design. These are presented separately, firstly to display their quality, and secondly because their design often features unique subcomponents which are best kept assembled together. Note that the special display assists users to locate distinctive components, and assists the repository to encourage adoption of the high value components.

choose components supplied free with the frameworks, or obtain from outside suppliers — but, the framework itself sets the overall shape and tone of the software.

In software, successful syntagmatic composition — context reuse resulting in Composed Software — provides much more value than paradigmatic selection – component reuse of software 'Components'. Does this mean 'Components' are worthless? Of course not! You cannot compose software without anything to compose: you must have some assets before you can leverage them to produce something new. But Composed Software is composed from many things more than just 'Components'. The surplus value of a composition lies in the end product — the value of the whole composition must surpass the cost of the parts — or there is no economic benefit to composition or reuse. The focus on 'Components', then, from McIlroy [21] via [10] to Szyperski [32], is fundamentally mistaken. What we are interested in, where the money is, is Composed Software, not software components.

Personality: So why don't all songs sound the same? Why are some artists great, write dozens of classics that move you to tears, say it like it's never been said before, make you laugh, dance, blow your mind, fall in love, take to the streets and riot? Well, it's because although the chords, notes, harmonies, beats and words have all been used before their own soul shines through; their personality demands attention.
— Jimmy Caulty and Bill Drummond, *The Manual* [5]

---

15. DEVELOPMENT

In traditional software development, the customer is absent after initial determination of requirements. Accordingly, the developers as suppliers are left having to make many decisions for themselves. The customer waits for the result.

This can create all kinds of problems, as the developers may misinterpret the customer intentions and supply functionality that is not needed, or not supply functionality that is needed.

Beck's ideas for Extreme Programming (XP) [2] address issues in software development, and in particular concentrates on the relationship between a software development team and the customer. As part of the process in XP, Beck emphasises an "on-site customer", where a customer representative is directly available for consultation and negotiation with the development team.

In software reuse, the issues involve the relationship between two software developers: one acting as supplier and the other acting as customer. In reuse, it is the supplier who is typically absent. Accordingly, it is the customer developer who is left having to make many decisions for themselves. This also can create problems, as the customer developer may misinterpret the supplier intentions, and may decide functionality is present when it is not, or may decide functionality is not present when it is.

The problem is the inverse of the one addressed in XP: the problem is the off-site supplier, and there has been much consideration of what to do. It turns out there are many reasons for failure. And as our technology has improved, we have succeeded in some measure, but then defined the problem in an ever narrower way. As we have succeeded in software reuse, so the phrase "software reuse" has come to refer to those even smaller ways in which we fail.

Many of the issues that remain are economic and organisational. Wider economic practice regards bringing together suppliers and customers as a serious concern, and offers us a variety of strategies to consider. For example, we might adopt centralised planning and management. Alternatively, we might facilitate a marketplace. For suppliers and customers, the issues still involve value, investment, and return.

Figure 14: Door Story — Standard components are displayed to facilitate quick selection of candidate components. The key to this process is the indexing information that is displayed prominently on each component. Some components have special features indicated with diagrams.

16. SCHMIDT

Why Software Reuse has Failed Historically

. . .

In theory, organizations recognize the value of systematic reuse and reward internal reuse efforts. In practice, many factors conspire to make systematic software reuse hard, particularly in companies with a large installed base of legacy software and developers. In my experience, non-technical impediments to successful reuse commonly include the following:

- Organizational impediments – e.g., developing, deploying, and supporting systematically reusable software assets requires a deep understanding of application developer needs and business requirements. As the number of developers and projects employing reusable assets increases, it becomes hard to structure an organization to provide effective feedback loops between these constituencies.
- Economic impediments – e.g., supporting corporate-wide reusable assets requires an economic investment, particularly if reuse groups operate as cost-centers. Many organizations find it hard to institute appropriate taxation or charge-back schemes to fund their reuse groups.
- Administrative impediments – e.g., it's hard to catalog, archive, and retrieve reusable assests across multiple business units within large organizations. Although it's common to scavenge small classes or functions opportunistically from existing programs, developers often find it hard to locate suitable reusable assets outside of their immediate workgroups.
- Political impediments – e.g., groups that develop reusable middleware platforms are often viewed with suspicion by application developers, who resent the fact that they may no longer be empowered to make key architectural decisions. Likewise, internecine rivalries among business units may stifle reuse of assets developed by other internal product groups, which are perceived as a threat to job security or corporate influence.
- Psychological impediments – e.g., application developers may also perceive "top down" reuse efforts as an indication that management lacks confidence in their technical abilities. In addition, the "not invented here" syndrome is ubiquitous in many organizations, particularly among highly talented programmers.

— Douglas C. Schmidt, *Why Software Reuse has Failed and How to Make It Work for You* [31]

17. POULIN

What to Measure As Reuse

We will now look at various categories of software, discuss why or why not each might count as reuse, and then give a recommend solution. Most of the conclusions hinge on whether or not we expect someone to write that software.

As we discuss these issues and reach our conclusions, we will define what it means when we refer to a Reused Source Instruction (RSI). Some of the choices may seem clear, but none have gone without controversy. For each code category, *someone has published a report claiming the category represents reuse!*

Product Maintenance: New Versions
. . . We do not count product maintenance as reuse.

Use of Operating System Services
. . . We do not count the use of the operating system as reuse.

Use of High-Level Languages
. . . We do not count use of high-level languages as reuse.

Use of Tools
. . . We do not count the use of tools as reuse.

Use Versus Reuse of Components
. . . We only count the first use of a component as reuse.

Use of Commercial Off-The-Shelf Software
. . . We do not count COTS products as reuse.

Ported Software
. . . We do not count porting as reuse.

Application Generators
. . . We do not count generated code as reuse.

Code Libraries
Use of utility libraries

. . . We do not count software from utility libraries as reuse.

Use of local utility libraries
. . . We may count software from utility libraries as resue.

Project and domain specific libraries
. . . We count software from project and domain specific libraries as reuse.

Corporate resue libraries
. . . We count software from reuse libraries as reuse.

Use of Modified Software
. . . We do not count modified components as reuse. [1]

Applying the Counting Rules to Object-Oriented Software

Recent articles and publications cite reuse as a principle benefit of object-oriented technology; however, object-oriented projects face the same issues as any other development project when it comes to defining what to count as "reused" software. Experience reports from OO projects routinely state impressive reuse levels and benefits due to reuse. However, we cannot trust these reports without understanding what the reports counted, e.g., inheritance or polymorphism. As with most experience reports, recent work in OO metrics fails to address this issue. Part of the problem comes from counting "internal" reuse in OO reuse metrics: in other words, using your own code . . . . This common and unfortunate misconception within the OO community really confuses many unwary practitioners.

— Jeffrey S. Poulin, *Measuring Software Reuse: principles, practices, and economics models* [29]

OPINION PARALYSIS: The tendency, when given unlimited choices, to make none.
— Douglas Coupland, *Generation X* [9]

---

18. JACOBSON, GRISS, JONSSON

For example, the transition from *no reuse* to *informal code reuse* (sometimes called leverage or cloning), in which chunks of code are copied, adapted slightly, and then incorporated into the new system, occurs when developers:

- are familiar with each other's code, and trust each other
- feel the need to reduce time to market, even though they would prefer to rewrite the software

This strategy works – for a while. Development time is reduced, and testing is often less tedious than with totally new code. But as more products are developed using this approach, maintenance problems increase. Multiple copies of the software, each slightly different, have to managed. Defects found in one copy have

[1] Throughout this paper we use ellipses to mark where we have left out words or passages to improve clarity: in this case we wish to note that Poulin does present arguments for each of these points, but it is the points themselves that we wish to highlight.

Figure 15: Door Story — Surprisingly, the repository also includes unused components. In this case, the components had been obtained for a new system but later found to be surplus to requirements. Rather than being discarded, they are offered to other users. In fact, the repository also offers custom-built components if users find no already-assembled components that meet their requirements.

to be found and fixed multiple times. This often leads to a *black box code reuse* strategy, in which a carefully chosen instance of code is reengineered, tested, and documented for reuse. All projects are then encouraged or required to reuse just this copy without modification.

This works well for a while, and then the issue of dealing with changes to satisfy an increasing number of reusers arises. Should everyone be "forced" to use only the standard version? Should multiple versions be maintained? Should adaptation be allowed? Who decides? Should test files and designs also be reused? Who will train and educate component reusers? All of these issues lead to the creation of a *managed workproduct reuse process*, in which the creation and reuse of components is explicitly managed and supported by a distinct organization.

Beyond this point, to get higher levels of reuse and more coverage of the lifecycle, it is important to move to *architected reuse*, to explicitly architect the components and the systems that will use them. This is the only way to insure that components fit together. The development and use of a common architecture involves even more organizational commitment and structure.

— Ivar Jacobson, Martin Griss, and Patrik Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, [13](p. 22)

19. MCCLURE

Selecting Reusable Components

Rationale

The system development and maintenance processes can be speeded up and aided by reusing various types of reusable components. Other reasons for using reusable components are: improved system quality, reduced system development and maintenance costs, and reduced risk of project failure. This technique ensures that system builders will search for all types of reusable components in all the likely sources.

Critical Issues

Buidling application systems from reusable components is based on the assumptions that reusable components exisit somewhere, they are reasonably easy to find and understand, and they are of good quality. If the time to search for candidate reusable components is too long in the system builder's eyes, the system builder will opt for building the component from scratch rather than reusing an existing ocomponent regardless of how well the reusable component fits the current need. A Reuse Library and a Reuse Catalog that are well organized with a classification scheme and supported by search and retrieval tools are essential to ensuring that reusable components are actually used in application system development.

— Carma McClure, *Software Reuse Techniques: Adding Reuse to the System Development Process* [20](pp.201–202)

Professional: The professionals of the city are like chess players who lost to computers. A perverse automatic pilot constantly outwits all attempts at capturing the city, exhausts all ambitions of its definition, ridicules the most passionate assertions of its present failure and future impossibility, steers it implacably further on its flight foward. Each disaster foretold is somehow absorbed under the infinite blanketing for the urban.

— Rem Koolhaas and Bruce Mau, *S, M, L, XL* [17]

---

20. TRACZ

Have you every tripped down the primrose path of least resistance, commending yourself for building a new program by salvaging someone else's software, only to be startled by the harsh reality that things were not as great as you planned?

True, you thought you were building on someone else's successes, but you had not counted on inheriting their mistakes or finding out, too late, that what you thought you could reuse "as is" required a lot more effort that you had planned. The software you were trying to salvage might be good, but for what? You budgeted time and staff to salvage or carry over code from the last project, only to find that it didn't work as advertised, if the fact that it worked was advertised at all.

Ignoring blatant errors of commission, the subtle errors of omission are the ones that really require the most effort to overcome (e.g., failure to document implementation decisions or failure to test for certain pathological conditions). The software might work well in the narrow context for which it was designed, but taken out of its specific domain the software suddenly becomes brittle — in other words reuseless.

...

Sometimes, programmers should let old code die a natural death rather than spend any effort trying to revive it. As many of us have learned from experience, plenty of reuseless code is lying around (one might argue that a lot of it was useless code in the first place). Not that most code is reuseless (or more important, needs to be created as reuseless), but software not specifically designed for reuse is simply more difficult and costly to reuse. Similarly code designed for reuse (reuseful code) might cost 30% to 200% more to develop, document, and test, but subsequent reuse costs 20% to 40% less than rewriting.

Making software reusable exacts a cost in experience and effort. Creating reusable interfaces requires insight in seeing how software has been used in the past and envisioning how it might be used in the future. Further-more, because the most important quality of reusable software, is that it be quality software, emphasis should be placed on thoroughly specifying, testing, and certifying that the software has achieved a certain level of operational and documentation quality. Only then will programmers be willing to invest their time to consider its reuse.

— Will Tracz, *Confessions of a Used Program Salesman: Institutionalizing Software Reuse* [33](pp. 73–74)

21. FOWLER

This tension between builders and designers happens in building too, but it's more intense in software. It's intense because there is a key difference. In building there is a clearer division in skills between those who design and those who build, but in software that's less the case. Any programmer working in high design environments needs to be very skilled. Skilled enough to question the designer's designs, especially when the designer is less knowledgeable about the day-to-day realities of the development platform.

...

One way to deal with changing requirements is to build flexibility into the design so that you can easily change it as the requirements change. However, this requires insight into what kind of changes you expect. A design can be planned to deal with areas of volatility, but while that will help for foreseen requirements changes, it won't help (and can hurt) for unforeseen changes. So you have to understand the requirements well enough to separate the volatile areas, and my observation is that this is very hard.

Now some of these requirements problems are due to not understanding requirements clearly enough. So a lot of people focus on requirements engineering processes to get better requirements in the hope that this will prevent the need to change the design later on. But even this direction is one that may not lead to a cure. Many unforeseen requirements changes occur due to changes in the business. Those can't be prevented, however careful your requirements engineering process.

So all this makes planned design sound impossible. Certainly they are big challenges.

...

Two of the greatest rallying cries in XP are the slogans "Do the Simplest Thing That Could Possibly Work" and "You Aren't Going to Need It" (known as YAGNI). Both are manifestations of the XP practice of simple design.

The way YAGNI is usually described, it says that you shouldn't add any code that will only be used by a feature that is needed tomorrow. On the face of it this sounds simple. The issue comes up with such things are frameworks, reusable components, and flexible design. Such things are complicated to build. You pay an extra up-front cost to build them, in the expectation that you will gain back that cost later. This idea of flexibility up-front is seen as a key part of effective software design.

However, XP's advice is that you not build flexible components and frameworks for the first case that needs that functionality. Let these structures grow as they are needed. If I want a money class today that handles addition but not multiplication, then I build only addition into the Money class. Even if I'm sure I'll need multiplication in the next iteration, and understand how to do it easily, and think it'll be really quick to do, I'll still leave it till that next iteration.

Figure 16: Door Story — Sometimes components are offered still assembled with related components from a previous context. The repository prefers to present the entire composition, instead of disassembling the unit, where such larger structures may be valuable to users.

One reason for this is economic. If I have to do any work that's only used for a feature that's needed tomorrow, that means I lose effort on features that need to be done for this iteration. The release plan says what needs to be worked on now. Working on other things for the future is contrary to the developers' agreement with the customer. There is a risk that this iteration's stories might not get done. Even if this iteration's stories are not at risk, it's up to the customer to decide what extra work should be done — and that might still not involve multiplication.

— Martin Fowler, *Is Design Dead?* [12](pp.5–9)

22. BROOKS

> The development of the mass market is, I believe, the most profound long-run trend in software engineering. The cost of software has always been development cost, not replication cost. Sharing that cost among even a few users radically cuts the per-user cost. Another way of looking at it is that the use of n copies of a software system effectively multiplies the productivity of its developers by n. That is an enhancement of the productivity of the discipline and of the nation.
>
> The key issue, of course, is applicability. Can I use an available off-the-shelf package to perform my task? A surprising thing has happened here. During the 1950's and 1960's, study after study showed that users would not use off-the shelf packages for payroll, inventory control, accounts receivable, etc. The requirements were too specialized, the case-to-case variation too high. During the 1980's, we find such packages in high demand and widespread use. What has changed?
>
> Not really the packages. They may be somewhat more generalized and somewhat more customizable than formerly, but not much. Not really the applications, either. If anything, the business and scientific needs of today are more diverse and complicated than those of 20 years ago.
>
> The big change has been in the hardware/software cost ratio. The buyer of a $2-million machine in 1960 felt that he could afford $250,000 more for a customized payroll program, one that slipped easily and nondisruptively into the computer-hostile social environment. Buyers of $50,000 office machines today cannot conceivably afford customized payroll programs; so they adapt their payroll procedures to the packages available. Computers are now so commonplace, if not yet so beloved, that the adaptations are accepted as a matter of course.
>
> . . .
>
> I still remember the jolt I felt in 1958 when I first heard a friend talk about building a program, as opposed to writing one. In a flash he broadened my whole view of the software process. The metaphor shift was powerful, and accurate. Today we understand how like other building processes the construction of software is, and we freely use other elements of the metaphor, such as specifications, assembly of components, and scaffolding.
>
> The building metaphor has outlived its usefulness. It is time to change again. If, as I believe, the conceptual structures we construct today are too complicated to be specified accurately in advance, and too complex to be built faultlessly, then we must take a radically different approach.
>
> Let us turn nature and study complexity in living things, instead of just the dead works of man. Here we find constructs whose complexities thrill us with awe. The brain alone is intricate beyond mapping, powerful beyond imitation, rich in diversity, and self-renewing. The secret is that it is grown, not built.
>
> — Frederick P. Brooks Jr., *No Silver Bullet* [15](pp.197–198)

23. ENVIRONMENT

On the consequences of composing modules into systems: Given a program to be "written"; to be "built"; to be "grown"; how should this be undertaken when much bespoke software is too expensive? Equally as important: what will the final program look like?

The process of composing modules into systems is qualitatively different from the structured approach of decomposing systems into modules. In a step-wise decomposition process, we begin with an idea of what the system should do, a problem to be decomposed; and we then make our decomposition, proceeding in a rational manner, isolating the important design decisions in their own modules, removing upward dependencies, following the grain of the domain. This is best described as proceeding top-down: from a problem to a program, in contrast to the bottom-up method of just coding a program without thinking first [27]. All this is well understood, and underlies many modern software development methods, from Dijkstra and Wirth, through Constantine and Mills, Dahl and Nygaard, Wirfs-Brock and Booch.

The process of composing modules into systems is also qualitatively different from that of building systems step by step. In the extreme, agile, or lean approaches we begin with a Person, (rather than a Problem) and mututally explore the domain until that Person's patience or prosperity runs out, building a program along the way. Agile aproaches build programs a small slice (or experimental spike) at a time: the first story, task, use case is implemented running vertically all the way from the top to the bottom of the system, iterating to widen this slice out to the whole system. Rather than trusting in a "Big Design Up Front", we write code but then revise the program's design, refactoring the program to achieve exactly the same kind of design qualities (compression, consiseness, clarity, lack of dependency, resonance with the domain) promoted by the structured approaches to programming, design, and analysis. All this is also well understood, as practiced and promoted by Beck [2], Fowler [12], and other advocates of the "agile" approach. Reuse is not one of the Twelve Practices of Extreme Programming, nor one of the Values from the Glowing Whiteboard of Agile Software Development [1].

The process of composing modules into systems, then, is quite different from either of these approaches. We care about Assets — Components, Software, Libraries, Modules, Frameworks, Configuration, Media, Images, Quotations, Music, Stuff — rather than Persons or Problems [4]. We proceed by aglomerating these Assets, remembering for you wholesale, justaposing, hammering, chucking them in, filling the spaces with smaller things, with glue, with strings and sealing wax, and other fancy stuff. We can take existing "programs" as "components", or perhaps begin our program by copying one or more existing programs and then adding more components, adjusting their configurations, or modifying them only where absolutely necessary (or where it looks like fun). There can be a complete absense of any traditionally-recognised forms of coding, design, or analysis, no refactoring, no stories, not top down, not bottom up, not horizontal or vertical, but random splodges of function based only upon whatever comes to hand.

One irony of object-oriented programming, extreme programming, agile development, lean coding, aspect-oriented separation of advanced meta-concerns, *et cetera*, is that the qualities and kinds of programs they aim to create are all exactly the same — disregarding minor differences of fashion in underlying programming languages. High cohesion, low coupling, low redundancy, high functionality. The topology and structure of great modern programs is remarkably similar — the THE Operating System, Sketchpad, Emacs, Unix, Smalltalk, the Wiki, TeX.

What we now think of as a program — Eä: the program that is, all-the-program-there-was, what we have put together and chosen to take as program, what we separate from all that is arbitrarily not-program — is this agglomeration, this collage, this text. Some components are bigger than others; some components are more important than others; and yet — there is no big story, there is no "main component", there is no main routine — or if there is something claiming to be a main component, there will be many staking that claim.

Early programs were written by one programmer. All kinds of software engineering — whether the standards and documentation of heavyweight processes, or the panopticon of agile development — attempt to keep up this fiction, the program should look as if it was written by one programmer. What we now think of as a program looks as if it was written by many people, at many different times, in many different languages, as if these people were competitors, as if they hated each other, as if they are didn't care, and were not unashamed.

In other words: pervasive reuse promotes a change in the method of construction of the program, and in the program itself.

After all, the etymology of "Hacker" is: "someone who makes furniture with an axe" [11].

OBSCURISM: The practice of peppering daily life with obscure references (forgotten films, dead TV stars, unpopular books, defunct countries, etc.) as a subliminal means of showcasing both one's education and one's wish to disassociate from the world of mass culture.
— Douglas Coupland, *Generation X* [9]

---

24. MCILROY

The Market

Coming from one of the larger sophisticated users of machines, I have ample opportunity to see the tragic waste of current software writing techniques. At Bell Telephone Laboratories we have about 100 general purpose machines from a dozen manufacturers. Even though many are dedicated to special applications, a tremendous amount of similar software must be written for each. All need input-output conversion, sometimes only single alphabetic characters and octal numbers, some full-blown Fortran style I/O. All need assemblers and could use macro-processors, though not necessarily compiling on the same hardware. Many need basic numerical routines or sequence generators. Most want speed at all costs, a few want considerable robustness.

Needless to say much of this support programming is done sub-optimally, and at a severe scientific penalty of diverting the machine's owners from their central investigations. To construct these systems of high-class componentry we would have to surround each of some 50 machines with a permanent coterie of software specialists. Were it possible quickly and confidently to avail ourselves of the best there is in support algorithms, a team of software consultants would be able to guide scientists towards rapid and improved solutions to the more mundane support problems of their personal systems.

In describing the way Bell Laboratories might use software components, I have intended to described the market in microcosm. Bell Laboratories is not typical of computer users. As a research and development establishment, it must perforce spend more of its time sharpening its tools, and less using them than does a production computing shop. But it is exactly such a systems-oriented market toward which a components industry would be directed.

The market would consist of specialists in system building, who would be able to use tried parts for all the more commonplace parts of their systems. The biggest customers of all would be the manufacturers. (Were they not it would be a sure sign that the offered products weren't good enough.) The ultimate consumer of systems based on components ought to see considerably improved reliability and performance, as it would become possible to expend proportionally more effort on critical parts of systems, and also to avoid the now prevalent failings of the more mundane parts of systems, which have been specified by experts, and have then been written by hacks.

— M. Douglas McIlroy, *Mass Produced Software Components* [21]

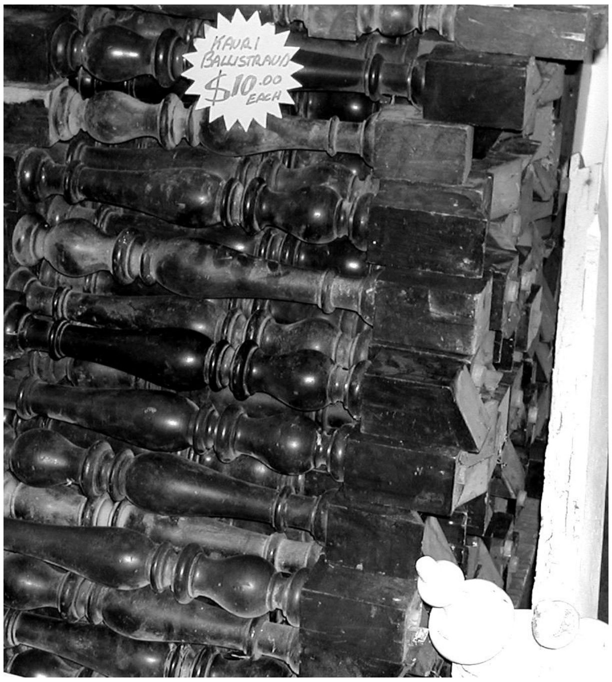

Figure 17: Balustrade Story — Alternatively, sometimes components must be disassembled. In this case, the larger component no longer complied with current legal requirements, and so the repository is forbidden to make the composite component available to users. The component was dismantled, and the smaller sub-components offered for reuse.

25. COX

Software crisis

The gunsmith shop in colonial Williamsburg, Va., is a fascinating place to watch gunsmiths build guns as we build software: by fabricating each part from raw materials and hand-fitting each part to each assembly.

When I was last there, the gunsmith was filing a beautifully proportioned wood screw from a wrought iron rod that he'd forged on the anvil behind his shop, cutting its threads entirely by hand and by eye. I was fascinated by how he tested a newly forged gun barrel — charging it with four times the normal load, strapping it to a log, and letting 'er rip from behind a sturdy shelter — not the least hindered by academia's paralyzing obsession that such testing 'only' reveals the presence of defects, not their absence.

The cottage-industry approach to gunsmithing was in harmony with the economic, technological, and cultural realities of colonial America. It made sense to expend cheap labor as long as steel was imported at great cost from Europe. But as industrialization drove materials costs down and demand exceeded what the gunsmiths could produce, they began to experience pressure to replace the cottage-industry gunsmith's process-centered approach with a product-centered approach; high-precision interchangeable parts to address the consumer's demand for less costly, easily repairable products.

The same inexorable pressure is happening today as the cost of hardware plummets and demand for software exceeds our ability to supply it. As irresistible force meets immovable object, we experience the pressure as the software crisis: the awareness that software is too costly and of insufficient quality, and its development nearly impossible to manage.

Insofar as this pressure is truly inexorable, nothing we think or do will stand in its path. The software industrial revolution will occur, sometime, somewhere, whether our value system is for it or against it, because it is our consumers' values that govern the outcome. It is only a question of how quickly, and of whether we or our competitors will service the inexorable pressure for change.

— Brad J. Cox, *Planning the Software Industrial Revolution* [10]

26. NORMAN

Two Kinds of Market Economies: Substitutable and Nonsubstitutable

The importance of a proper infrastructure goes beyond its impact upon usefulness and intrusiveness. It can determine the entire success and failure of a technology. The lesson of Thomas Edison and his choice of an incompatible infrastructure, both for electricity and the phonograph, leads to a more general lesson about the marketplace.

There are two kinds of economic markets: substitutable and nonsubstitutable. Substitutable goods are products like groceries, clothes, and furniture. Nonsubstitutable goods are invariably infrastructures. The two have very different properties. Most recent books about the business and marketing side of technology miss this distinction, but a company that provides a substitutable good must function very differently than one that provides a nonsubstitutable one.

In a substitutable marketplace, goods of one manufacturer can be substituted for goods of another. This is the classic market-driven economy, where competition prevails. This applies to food and newspapers, to automobiles and television sets. In this marketplace, standard market forces are at work and the market can be shared among competitors. Usually, one company has a substantial lead, but the others can coexist in relative stability. This is a classic case of free market competition. The choice of one substitutable good makes no commitment for the future. The consumer can buy a Pepsi today and a Coke tomorrow: the first choice does not constrain the second.

In a nonsubstitutable market, the required infrastructure means that goods from one manufacturer cannot be substituted for goods of another. This is the marketplace that Edison found himself in with his use of DC electricity over the competition's use of AC. It was the same story with his use of vertically cut cylinders and discs when the competition used laterally cut discs. It is what happened with Beta videocassette recorders.

Once there is a nonsubstitutable market is doesn't matter how good the product is.

— Donald A. Norman, *The Invisible Computer* [24])(pp.116–117)

27. ROBINSON, HOVENDEN, HALL, AND RACHEL

Fordism and postFordism

When Brad Cox [1990] wrote his riposte to Fred Brooks, he proposed as a silver bullet a technical solution based on components, taking a manufacturing viewpoint leaning heavily on the experience of mass-production. This approach is known as Fordism, or Taylorism, and much of software development has been grounded in this view of human industrial activity.

Robin Murray [1989] has given a very readable account of Fordism and post-Fordism. Fordism began at the start of the twentieth century, and has dominated industrial processes since. It has four basic principles:

- standardised products
- repeated tasks having potential for automation
- unautomated tasks analysed using work study methods, to enable the easy training of workers and their easy replacement — this is the scientific management of Fred Taylor, also known as Taylorism.
- production lines with the work moving to the workers.

This method of production had high initial costs and relied upon large volumes of sales, and thus heavy marketing and even the manipulation of the market. Managements were very hierarchical. Labour relations were poor, and turnover of employees was large due to the mechanical nature of the employment. Fordist ideas were not restricted to Europe and America, and were also keenly adopted within the industrialisation programs of the former Communist block.

We see Fordism and Taylorism in the software lifecycle and software development methodologies. While we do not have production lines in computing with detailed specialisation, we do identify different skills of analysis and programming, testing and management, and decompose the work into discrete steps where specialists are applied using their specific skills. The specialisation may further breakdown into programmers skilled with particular programming languages, designers specialised in particular types of systems like communications, database and interfaces, and analysts specialised in particular application areas, from command and control to banking to flight control. The Fordist view is further seen in the preoccupation with metrics and process improvement, the emphasis on control ('you cannot control what you cannot measure').

Alternatives to Fordism arose within retailing and from there moved into manufacturing. Information technology is seen as critically important in these developments, enabling retailers like Sainsburys to track the stocks in their stores and the sales during the day to arrange delivery of just the correct quantities from their warehouses overnight. From there it was a small step to by-pass the warehouses and order directly from the suppliers, so that the suppliers tuned their production to the retailing needs on a daily cycle. The need for the holding of large stocks was removed, this is just-in-time manufacturing.

At the manufacturing end, flexibility was also achieved through information technology, with the lead coming from Japan, from Toyota, reputedly following a visit to the US by Toyota who saw the post-Fordist retailing there. The move was to reskill workers, using methods like quality circles to tap the knowledge of the workers. "In post-Fordism, the worker is designed to act as a computer as well as a machine" (Murray 1989 p272) And the specialist machinery itself was also flexible, being general purpose and capable of being set up for some other job very quickly indeed.

From there the next step was to subcontract wherever possible. The retailing equivalent has been franchising. The practices of Benetton in the textiles and clothing industry is used as an example of this trend, directly employing a few thousand people but indirectly employing tens of thousands through subcontracting of manufacture, and franchising sales. Information technology has been critical in monitoirng sales worldwide, noticing trends, and switching production to meet the orders from shops.

— Hugh Robinson, Fiona Hovenden, Pat Hall and Janet Rachel, *Postmodern Software Development*, [30], also citing Cox [10] and Murray [22].

Modernism: Modernism's alchemistic promise — to transform quantity into quality through abstraction and repetition — has been a failure, a hoax: magic that didn't work. Its ideas, aesthetics, strategies are finished. Together all attempts to make a new beginning have only discredited the *idea* of a new beginning. A collective shame in the wake of this fiasco has left a massive crater in our understanding of modernity and modernization.
— Rem Koolhaas and Bruce Mau, *S, M, L, XL* [17]

---

28. NAUR AND RANDELL

There was a considerable amount of debate on what some members chose to call the 'software crisis' or the 'software gap'. As will be seen from the quotations below, the conference members had widely differing views on the seriousness, or otherwise, of the situation, and on the extent of the problem areas.

David and Fraser: (from their *Position paper*) "There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well."

Hastings: I am very disturbed that an aura of gloom has fallen over this assembly. I work in an environment of many large installations using OS/360. These are complex systems, being used for many very sophisticated applications. People are doing what they need to do, at a much lower cost than ever before; and they seem to be reasonably satisfied. Perhaps their systems do not meet everybody's need, they don't meet the time sharing people's demands for example, but I don't think software engineering should be confused with time sharing system engineering. Areas like traffic control, hospital patient monitoring, etc. are very explosive, but are very distinct from general purpose computing.

Gillette: We are in many ways in an analogous position to the aircraft industry, which also has problems producing systems on schedule and to specification. We perhaps have more examples of bad large systems than good, but we are a young industry and are learning how to do better.

Randell: There are of course many good systems, but are any of these good enough to have human life tied online to them, in the sense that if they fail for more than a few seconds, there is a fair chance of one or more people being killed?

Graham: I do not believe that the problems are related solely to online systems. It is my understanding that an uncritical belief in the validity of computer-produced results (from a batch-processing computer) was at least a contributory cause of a faulty aircraft design that lead to several serious air crashes.

Perlis: Many of us would agree that Multics and TSS/360 have taken a lot longer to develop than we would have wished, and that OS/360 is disappointing. However, perhaps we are exaggerating the importance of these facts. Is bad software that important to society? Are we too worried that society will lose its confidence in us?

Randell: Most of my concern stems from a perhaps over-pessimistic view of what might happen directly as a result of failure in an automated air traffic control system, for example. I am worried that our abilities as software designers and producers have been oversold.

Opler: As someone who flies in airplanes and banks in a bank I'm concerned personally about the possibility of a calamity, but I'm more concerned about the effects of software fiascos on the overall health of the industry.

Kolence: I do not like the use of the word crisis. It's a very emotional word. The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis — sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.

Ross: It makes no difference if my legs, arms, brain and digestive tract are in fine working condition if I am at the moment suffering from a heart attack. I am still very much in a crisis.

Fraser: We are making great progress, but nevertheless the demands in the industry as a whole seem to be going ahead a good deal faster than our progress. We must admit this, even though such an admission is difficult.

Dijkstra: The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.

— P. Naur and B. Randell, *Software Engineering: Report of a conference sponsored by the NATO Science Committee* [23]

Figure 18: Brick Story — Even the most basic of components have a place in the repository. While these components are basic, they are made using using a process that is no longer common. Accordingly, a significant effort was necessary to locate and obtain these basic components.

1968: *Sous le pavé, la plage* (under the pavement, beach): initially. May '68 launched the idea of a new beginning for the city. Since then, we have been engaged in two parallel operations: documenting our overwhelming awe for the existing city, developing philosophies, projects, prototypes for a preserved *and* reconstituted city and, at the same time, laughing the professional field of urbanism out of existence, dismantling it in our contempt for those who planned (and made huge mistakes in planning) airports, New Towns, satellite cities, highways, high-rise buildings, infrastructures, and all the other fallout from modernization. After sabotaging urbanism, we have ridiculed it to the point where entire university departments are closed, offices bankrupted, bureaucracies fired or privatized.

— Rem Koolhaas and Bruce Mau, *S, M, L, XL* [17]

---

29. THE END

Is Software Engineering, as a discipline, a fraud founded upon a lie?

The Lie: There is a 'software crisis' — in truth there never was.

The Fraud: You must to spend lots of money to bridge a 'software gap' — in truth there is no gap.

The Consolation: There is no silver bullet — but there are no werewolves.

So there have been minor problems — from time to time the phone network crashes, a rocket explodes, every desktop computer in the world succumbs to a virus — but we ignore them the way we ignore flat batteries, the inevitably of traffic accidents as we walk across a roadway, or difficulties we have in opening plastic bottles of ketchup.

The dot-com bubble didn't end because of a 'software gap', because programmers couldn't write proper software fast enough. Rather the reverse: programmers were able to write software so easily that businesses began to pay for software that was pointless, and therefore worthless.

Software engineering has failed, but software is a success.

What is now becoming clear is that software — irrespective of how it is written, built, or grown — has a structure more like a natural system than a mathematical construction. The relationships between the objects in a program is more like that of the words in a novel, the cells in an organism, or the leaves on a tree, than a series of propositions linked by *modus ponens* [35, 28]. Programs will more easily yield their secrets to critical theory, biology, or botany, than to modal logic or complexity theory.

No program is an island: every program contains a trace of every program that has ever been written, or that will ever be. So where does this leave us, at the end of Software Engineering? We are immeasurably richer due to all the software in the world. We draw upon this software whenever we need to create something 'new': programming is — and always has been — re-use, re-programming. We have the makings of a new discipline: the natural science of composed software; the normal science of the study of all the software in the world.

Crisis: What if we simply declare that there is no crisis — redefine our relationship with the city not as its makers but as its mere subjects, as its supporters?

More than ever, the city is all we have ...
— Rem Koolhaas and Bruce Mau, *S, M, L, XL* [17]

Figure 19: Fountain Story — Marcel Duchamp showed the importance of context in art. Even recycled ready-made components have beauty and value. The component repository helps to bring the beauty and value of components to new users in new contexts.

30. REFERENCES

[1] Agile Alliance. Manifesto for agile software development. Available at `http://agilemanifesto.org`, 2003.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Robert Biddle and Ewan Tempero. Understanding the impact of language features on reusability. In Murali Sitaraman, editor, *Proceedings of the Fourth International Conference on Software Reuse*, pages 52–61, Orlando, USA, April 1996. IEEE Computer Society.

[4] Robert Biddle and Ewan Tempero. Towards asset based software engineering. In *Ninth Workshop on Institutionalizing Software Reuse (WISR9)*, 1999.

[5] Jimmy Caulty and Bill Drummond. *The manual: How to Have a Number One the Easy Way*. KLF Publications, 1988.

[6] Larry L. Constantine. *The Peopleware Papers: Notes on the Human Side of Software*. Prentice-Hall Inc., 2001.

[7] Alan Cooper. *The Inmates are Running the Asylum*. Sams Publishing, 1999.

[8] Augusta Ada King Countess of Lovelace. Annotated translation of sketch of the Analytical Engine invented by Charles Babbage, by L. F. Menabrea, of Turin, Officer of the Military Engineers. (Available at `http://www.fourmilab.ch/babbage/sketch.html`), 1843.

[9] Douglas Coupland. *Generation X: Tales for an Accelerated Culture*. St. Martins Press, 1992.

[10] Brad J. Cox. Planning the software industrial revolution. *IEEE Software*, November 1990.

[11] Eric S. Raymond (Editor). The jargon file. Available at `http://www.jargon.org`, 2003.

[12] Martin Fowler. Is design dead? In *Extreme Programming Examined, Giancarlo Succi and Michele Marchesi, Editors*, pages 3–18. Addison-Wesley, 2001.

[13] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.

[14] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[15] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, anniversary edition edition, 1995.

[16] Donald E. Knuth. *Selected Papers on Computer Science*. Cambridge University Press, 1996.

[17] Rem Koolhaas and Bruce Mau. *S,M,L,XL, O.M.A., Small, Medium, Large, Extra-Large, Office of Metropolitan Architecture*. The Monacelli Press, second edition, 1998.

[18] Susan Lammers. *Programmers at Work*. Microsoft Press, 1986.

[19] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on very high level languages*, pages 50–59, 1974.

[20] Carma McClure. *Software Reuse Techniques: Adding Reuse to the System Development Process*. Prenctice-Hall Inc., 1997.

[21] M. Douglas McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report on a conference sponsored by the NATO Science Committee on Software Engineering*, pages 138–155. Scientific Affairs Division, NATO, Brussels, 1969.

[22] Robin Murray. *Fordism and Post-Fordism*, pages 167–276. Academy Editions, 1992.

[23] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. NATO Scientific Affairs Division, Brussels, 1969.

[24] Donald A. Norman. *The Invisible Computer: Why Good Products Fail, The Personal Computer is so Complex, and Information Appliances are the Solution*. MIT Press, 1999.

[25] Kristen Nygaard and Ole-Johan Dahl. The development of the simula languages. In *The first ACM SIGPLAN conference on History of programming languages*, pages 245–272, 1978.

[26] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[27] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, 1986.

[28] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry of object-oriented programs. *Communications of the ACM*, 2004. To appear, draft available at: `http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-02-30.abs.html`.

[29] Jeffrey S. Poulin. *Measuring Software Reuse: principles, practices, and economic models*. Addison-Wesley Longman Inc., 1997.

[30] Hugh Robinson, Fiona Hovenden, Pat Hall, and Janet Rachel. Postmodern software development. *Computer Journal*, 41(6), 1998.

[31] Douglas C. Schmidt. Why software reuse has failed and how to make it work for you. *The C++ Report*, 1999.

[32] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Inc., 1997.

[33] Will Tracz. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley Publishing Inc., 1995.

[34] Alan Turing. Proposed electronic calculator. (Available at `http://www.alanturing.net/turing_archive/archive/index/aceindex.html`), 1946.

[35] S. Valverde, R. Ferrer Cancho, and R. V. Sole. Scale-free networks from optimal design. Available at `http://www.santafe.edu/sfi/publications/wpabstract/200204019`.

[36] Maurice V. Wilkes. *Memoirs of a Computer Pioneer*. MIT Press, 1985.

[37] Maurice V. Wilkes, David J. Wheeler, and Stanley Gill. *The Preparations of Programs for An Electronic Computer: With special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press, 1951.

PARASITE What happens when a critical essay extracts a "passage" and "cites" it? Is this different from a citation, echo, or allusion within a poem? Is a citation an alien parasite within the body of its host, the main text, or is it the other way around, the interpretative text the parasite which surrounds and strangles the citation which is its host?

— J. Hillis Miller, as quoted in Gregory L. Ulmer, "The Object of Post-Criticism", in *The Anti-Aesthetic: essays of Postmodern Culture*, ed. Hal Foster (Seattle: Bay Press, 1989), as quoted in Rem Koolhaas and Bruce Mau, *S,M,L,XL* [17]
