'Systematic' Versus 'Accidental' Reuse in Japanese Software Factories

Michael A. Cusumano

MIT Sloan School of Management

WP# 3328-BPS-91
Draft: September 9, 1991

Address Correspondence to:

Michael A. Cusumano
Associate Professor
MIT Sloan School of Management
Room E52-555
Cambridge, MA 02139

Tel. (617) 253-2574

ABSTRACT

A major topic in software engineering is how to reuse or recycle product components across more than one system or project, making it unnecessary to keep specifying, building, and testing new products completely from scratch. While reuse may seem to be primarily a *technological* problem, this article illustrates how reusability also needs to be seen as a *managerial and organizational* problem. This is because the recycling of specifications, detailed designs, and actual code, as well as development tools, engineering methods, and test cases, can either be *systematic* -- occurring as frequently as possible due to management planning as well as provision of s ·pport technologies and incentives -- or *accidental* -- occurring infrequently and in an ad hoc manner due to the lack of formal planning, support, or personnel incentives. The focus of the discussion is on techniques and tools used in Japanese software factories, especially Toshiba, which in the mid-1980s was routinely delivering software systems with nearly 50% reused code.

INTRODUCTION

A topic in software engineering as well as in product development management in general has continued to arouse interest among academics and practitioners: how to reuse or recycle product components across more than one system or project, making it unnecessary to keep specifying, building, and testing new products completely from scratch.[1] The real problem is how build, deposit, and then find existing components so that reuse does not cost more than making new components, and final products with a lot of reused designs or code do not contain too many compromises in performance. Reuse may thus seem to be primarily a *technological* problem, depending, for example, on the structure of specific software components and their suitability for different application contexts, user requirements, or hardware platforms. This article, however, draws on examples from Japan to demonstrate that, in software as in other engineering domains, reusability also needs to be seen as a *managerial and organizational* problem. This is because the recycling of specifications, detailed designs, and actual code, as well as development tools, engineering methods, and test cases, can either be *systematic* -- occurring as frequently as possible due to management planning as well as provision of support technologies and incentives -- or *accidental* -- occurring infrequently and in an ad hoc manner due to the lack of formal planning, support, or personnel incentives.

Of course, if every product that a development group tries to build is totally different, then reuse is either impossible or limited to elementary process knowledge, such as general project-management expertise, or very common subroutines, such as screen handlers or date-conversion modules. For a series of projects where reuse is possible, however, many firms still find that reuse does not occur to the same extent as there exist opportunities based on redundancies in customer requirements and program functions. Because of the potential benefits to productivity and quality from reusing proven components, how to raise reusability to both technological and managerial or organizational limits has become a major strategic concern for software producers. This is especially true in Japan, where there persists a small number of available software packages, severe shortages of skilled programmers, and large demand for labor-intensive custom software (including programs embedded in hardware) where many of the required functions are not unique to each project.

REUSE CONCEPTS AND TECHNICAL SUPPORT

Reusable Components: Just as reuse of experience embodied in methods and tools is a common engineering and manufacturing practice, reuse of software components has its analog in any conventional industry where engineers design standardized, interchangeable parts that factories mass produce and then use in identical or even somewhat different end products. The notion of recycling pieces of software first occurred to programmers in the 1950s who found they did not have to keep rewriting common subroutines, such as for converting temperatures or sorting different types of data. By the mid-1960s, packages of reusable Fortran routines were commercially available from IBM. The idea of making and using reusable components gained further popularity during the 1968 NATO conference on software engineering when M.D. McIlroy of AT&T suggested that governments and companies establish "components factories" to create mass-producible modules to serve as "black boxes" for new programs. Producers would benefit by building new programs from existing components that represented efficient designs, while users would have products that, even though containing some standardized parts, were tailored to their needs.[2] Although McIlroy referred primarily to reusable subroutines at the code level, reuse has gradually come to include data types, architectures, designs, whole programs, modules, and other elements (such as tools, methods, and experience), in addition to executable code, which often proved most difficult to reuse across different applications or computer systems.[3]

Benefits and Costs: While only a few empirical studies of reuse exist because of the difficulty of compiling, analyzing, and revealing company data, available research indicates that reuse has a positive impact on productivity, at least when designs and code are not changed much before they are redeployed.[4] Reuse prior to coding seems particularly valuable because writing requirements and detailed designs often accounts for a much larger portion of development costs than coding. Developers can achieve large savings in testing as well if they build reusable modules as independent subsystems and then thoroughly test these before recycling. Reusing debugged designs and coded modules can also reduce long-term maintenance costs, often the most expensive part of software development. In addition, reuse has the potential of leveraging good designs across more than one project. Most reuse proponents cited in the literature have even argued that, since research on software tools and methodologies has brought only limited gains, reusability, including design reuse and automatic program generation, remained the only way to achieve large advances in productivity.[5]

On a small scale, prior to the 1980s, firms met McIlroy's goal with the increasing use of packaged subroutines and modifiable packaged software, as well as with the gradual introduction of operating systems such as UNIX and programming languages that supported subroutine reuse and portability of code and tools across different hardware architectures. But, despite some progress, such as at Raytheon in the late 1970s, extensive reuse appeared not to become a major objective of software-engineering practice in the United States until a revival in the early 1980s in academic circles and at firms such as the ITT Corporation. In the meantime, Japanese applications-software producers, led by Toshiba, began pursuing reusability as a primary emphasis in the mid-1970s. A study of 40 Japanese and U.S. software systems published in 1990 found that Japanese reuse levels were at least twice those in the U.S. sample (averaging 18% compared to under 10% of delivered lines of code), although the averages were not as large as sometimes claimed.[6] Nor did the Japanese achieve high levels of reuse until after they accumulated better component libraries and support tools, and introduced systematic management practices and incentives.

Reuse in Japan and elsewhere has made slow progress because of various costs and constraints. On the technical side, many factors influenced whether a particular project can recycle existing components, such as the match in application or function of the existing software compared to the new design requirements; the particular language and the characteristics of the target computers; and the program structure and degree of modularity, or how independent modules are of other modules or subsystems in the original program based on the construction of the interfaces between various layers in a complete system (the new application software, subsidiary applications packages, utilities, the operating system, and the hardware). Structured techniques also teach developers to conceptualize software systems anew in a top-down manner, from large functions to smaller tasks, and to write programs for specific applications. Reusing designs, code, or architectures, on the other hand, requires conceptualization and construction around pieces of existing software, and thus acceptance of possible compromises in system features or performance.

On the organizational side, writing and documenting components for reuse generally requires extra time that individual customers might not want or should not want to pay for. Project managers, and project members, have good reason to resent this extra time, unless they see opportunities to save time and money in the future. For example, systematic recycling of designs or code required standardization of module interfaces to fit components together, whether or not these standards prove useful or necessary for a given product. Developers need common tools, such as libraries to store programs and documents, and reusable-parts retrieval systems, to make it easier to find and understand pieces of software someone else had written. In designing software for potential reuse, designers have to think of more possible applications and more carefully document their work.

Reuse thus involves potential tradeoffs in product uniqueness and costs in the short term on any individual project, even though recycling promises higher productivity and even quality over a series of projects by eliminating the need to reinvent designs or continue to do coding, documenting, and testing for similar applications. In fact, while many software developers and customers might prefer uniquely designed programs, the widespread use of software packages, as well as several studies indicating that somewhere between 40% and 90% of the code applications producers delivered in a given year consisted of similar functions, indicated there was vast potential for more reuse.[7]

Reuse Support: In addition to emphasizing and teaching abstraction techniques, Japanese firms introduced several tools during the 1980s to support abstraction in general as well as the specific reuse of system architectures, data types, designs, and coded modules or whole programs. Hitachi, for example, began using in the early 1980s a tool called EAGLE (Effective Approach to Achieving High Level Software Productivity), which helped applications programmers build software from reusable modules as well as structure new designs and code for reuse. Company engineers claimed that users could develop 60% of the application programs customers demanded from 22 patterns, all accessible in the EAGLE database and modified only slightly for individual applications.[8]

EAGLE users first analyze data items and interrelationships and catalog these in a data-dictionary database. Next they register the system-design and program specifications in another database. At this point, EAGLE searches a pattern and parts library for existing components the system might reuse. This makes it possible to assemble some programs almost entirely from the library, although most programs written with the tool contain reused patterns and parts as well as new designs and routines. EAGLE next generates an outline of the program from the detailed (module-level) specifications and then produces a source program. Programmers can edit the source program to add particular functions wanted by individual users. Finally, EAGLE automatically generates test commands and carries them out in conversational Japanese. During 1984-1986, Hitachi refined EAGLE to allow it to handle PL/1 and a Japanese specifications language, in addition to COBOL. The new version of the tool also lets customers design their own menus (within certain limits). With these improvements, Hitachi began using EAGLE to construct database and data-communications programs, along with common business programs. In terms of performance, according to Hitachi's internal audits, programs designed with EAGLE generally showed a 2.2-fold improvement in productivity (measured by lines of code per programmer and manpower costs in a given time period). EAGLE also shifted more effort into system design and substantially reduced time necessary for testing.

NEC in 1984 began using a similar tool set called SEA/I (System Engineering Architecture/I). Its three subsystems consist of an Empirical Information Base (EIB), a collection of support tools, and a standardized methodology covering system proposals, design, implementation, testing, and installation. The EIB provides set formats and ready access to previously written system definitions, layout designs, system and program structures, program modules ("source parts"), tested programs, and test data. Rather than requiring one development methodology for all customers, SEA/I also specifies five approaches.

First, the tool users can adopt a conventional life-cycle model and move, sequentially with the usual iterations, from requirements definition through implementation and testing. A second approach relies on SEA/I's software CAD capabilities for prototyping, design, and then semi-automated generation of program components (automation except for some special coding). A third calls for building new software from reusable design specifications and coded modules, with minimal modifications but customization in the sense that SEA/I users can offer different configurations of the components for different customers; this approach minimizes testing, since the program components come from a library of fully tested items. A fourth model addresses the development of customized systems where SEA/I helps users build a prototype and finished program without changing individual modules but by offering the customer a combination of existing applications packages; in this case, the new program needs little if any coding or testing. The fifth calls for the use of SEA/I tools to analyze, restructure, and re-document existing programs built before the introduction of SEA/I, to facilitate enhancements, reuse, or maintenance of these older software systems.

Fujitsu has a comparable but larger set of reuse-support tools, including several dedicated to particular applications. These include PARADIGM (for general business applications primarily in COBOL), ACS-APG (a specialized version of PARADIGM that supports applications control structure and program generation for more complex on-line transactions processing systems), and BAGLES as well as BAGLES/CAD (specialized versions of PARADIGM for generating banking applications software). BAGLES/CAD is particularly significant as a simplified, menu-based tool that allows users with little or no knowledge of software programming to produce executable banking programs of considerable size and complexity.

A sample of Japanese R&D for reuse support comes from work originating with

Japan's Fifth-Generation Computer Project. A group within Fujitsu's central research laboratories has developed an experimental programming-support system that includes an English-like specification language mechanically translated into predicate logic formulas, and a logic-based system to retrieve reusable software modules, stored by function, from a modules library. The library stores specifications for each module, coded in PROLOG, which the tool compares with requirements to identify functionally equivalent modules reusable for particular parts of a new program. The tool goes beyond conventional reuse-support systems by adding superior retrieval and verification capabilities. Earlier reuse-support methods located modules by matching specifications or code, whereas the PROLOG system makes it possible to identify modules with similar functions even if the specifications do not match in a conventional search process. Another feature of the tool, which supports reuse as well as maintenance, is an "explanation generator" that analyzes code and produces English-like explanations of the program logic by comparing the code with preexisting templates (skeletons) of explanations stored in a separate database.

CASE STUDY: THE TOSHIBA SOFTWARE FACTORY[9]

Factory Structure and Process: Toshiba established a software factory in 1977 to develop real-time process-control software embedded in hardware systems made at the company's Fuchu Works, located in the western outskirts of Tokyo. In the late 1980s, the Works had approximately 7,500 employees primarily in four areas: Information Processing and Control Systems, Energy Systems, Industrial Equipment, and Semiconductors (Printed Circuit Boards). Product departments within the divisions corresponded roughly to 19 product lines; each department contained sections for hardware and software design as well as for manufacturing, testing, quality assurance, and product control. Approximately half the Fuchu Works' employees were software developers, with about 2,300 working in the Software Factory, assigned from the product departments for particular projects.

Following this matrix structure, Toshiba had no single manager of its Software Factory, although the factory contained a permanent staff spread over five sections. Four provided administrative support for software production, relying on systems analysts as well as programming and testing personnel from product departments. The fifth maintained the tool set, called the Software Workbench (SWB), helped R&D teams develop new tools, and operated the SWB Service Center, the file-storage room, and other related SWB facilities. It also coordinated software quality-assurance plans for the entire factory and provided assistance to keep these plans on track, in addition to collecting data to evaluate productivity, reuse, and software reliability.

Individual software systems were huge. The average applications program consisted of about 4 million equivalent-assembler source lines (EASL) of code; the range was 1 to 21 million EASL. Projects (including hardware and software components) generally took three years to complete. Four or five systems analysts normally did high-level design and worked full-time on one project until completion. Another 10 to 15 engineers did detailed design, while 70 to 80 programmers completed the coding and debugging. To divide and then coordinate their activities, Toshiba broke down the software production process into distinct phases, following a life-cycle model common to both hardware and software products: requirements specification and design, manufacturing, testing, installation and alignment, and maintenance. Prescribed procedures and specific tools from the SWB system provided support for each phase.

While having analysts, designers, and programmers in the same departments encouraged communication between the different groups, Toshiba also relied on a formalized process for requirements specification and design, which it broke down into two parts. Part I included the customer's specific objectives as well as constraints such as cost and time, and particular methodologies the customer wanted Toshiba to follow. Systems analysts outside the factory drew up these system designs. Part II, a more precisely structured document done after analysis of the Part I requirements, outlined the overall functions of the program and simulated its operation to generate performance parameters that Toshiba used to negotiate prices and other contract elements with the customer. Designers already assigned to the project and physically located in the Software Factory drew up these Part II specifications. On some occasions, such as when they did not want to share too much proprietary knowledge with Toshiba, customers wrote their own Part I specifications and then the Software Factory turned these into code.

Reuse Promotion: Toshiba's strategy for accommodating rising demand for software and ensuring a high level of productivity and quality, despite variations in individual skills, was to build products from standardized components reassembled or combined with new components. This approach -- the systematic creation and reuse of reusable software parts -- lay at the heart of Toshiba's concept of *factory* production for software. Toshiba did not solve all problems related to reusability; it did not, for example, utilize very formalized or sophisticated methods for classifying reusable software components and recycling them across different product families. Nonetheless, the factory simplified or restricted problems to manageable areas, and provided incentives both for project managers and personnel to write reusable software and reuse it frequently, at least within similar application domains. How the factory promoted reuse in the face of organizational as well as technological constraints illustrates the strategic management of technology, integrating product planning, engineering tools and techniques, personnel training and incentives, as well as management control systems.

Approximately half the software Toshiba's factory delivered in 1985 (the last year for which Toshiba has made data public) consisted of reused code and designs, including some applications packages developed in product departments to serve as major components of customized systems and designs or code with minor modifications (usually no more than 20% of the contents of an individual module). The other half of delivered software was new, mainly written for an individual customer or application, again including some tailored packages. The SWB system collected data on all reused elements, converted designs to equivalents in generated code, and paid particular attention to a simple output measure: the number of reused lines converted to EASL code in a delivered system.

Table 1 provides a breakdown of the three categories into which the approximately 50% reused software fell. One category consisted of packages of design skeletons, called "white-box" parts, kept in department program libraries. These described functions common to applications within a particular domain, such as nuclear-power plant control systems, or steel-mill process control systems, and ranged in size from 1,000 to 10,000 EASL. Software developers in the factory often merely chose the right package and filled in blank slots for different customers; code generators produced much of the actual code. Developers could also modify the designs. Another portion consisted of relatively large utility programs that worked in between operating systems and industry-specific applications packages to control communications, database management, and other basic functions; or tools and other embedded sub-programs generally usable in a variety of systems. Toshiba deposited these components, which ranged in size from 10,000 to 100,000 EASL, mainly in department libraries but sometimes in the factory library system. The final category (about 10% of delivered code) consisted of black-box modules, usually no more than

3,000 EASL in size. These common subroutines, also accessible from across product departments through a central factory library, covered functions demanded in most systems the factory built, such as for managing general-purpose displays or converting temperature data.

For the other 50% of delivered software that Toshiba wrote from scratch for individual customers, in addition to SWB tools and conventional languages such as Fortran, factory departments were beginning to deploy very-high level (4th-generation) application-specific languages for systems design that eliminated coding as a separate task. An example is POL (Problem-Oriented Language), which the nuclear power-plant department relied on extensively to design components specific to individual customers. Unlike conventional computer languages, POL relied on menus and tables with blank spaces representing control logic for various functions (found primarily in nuclear power plants, however, limiting the use of this language to one application domain). Engineers filled in the blanks and a compiler produced executable code. Another tool that worked with POL, RRDD (Reverse Requirements Definition and Documentation System), generated updated documentation automatically, in the event personnel changed parts of an existing program.

In order to build programs around existing components as much as possible, Toshiba required projects to draw up plans, called "repeat maps," at the requirements-analysis and module-design phases. Systems analysts produced the first map by comparing the main subsystems they wanted to build with existing packages in the department and factory libraries. After inserting the appropriate packages into the system under construction, designers in the factory drew up another set of repeat maps, identifying specific modules to reuse or modify and new components needed to implement requirements.

The organization Toshiba created to promote reuse and overcome short-term concerns of project managers and development personnel relied on software reusing parts steering committees along with software reusing parts manufacturing departments and software reusing parts centers (Figure 1). Product departments formed steering committees for different areas (with different members, depending on the application) to determine if customers had a common set of needs suitable for a package, and then allocated funds from the Fuchu Works' budget for these special projects. Some packages were utilities usable in different departments, although most served specific applications. For example, PODIA (Plant Operation by Displayed Information and Automation), a package created in the department for nuclear power plants, covered all the basic functions common to these systems and accounted for about half the software the department delivered.

The reusing parts manufacturing departments and parts centers, at the project, product-department, and factory levels, evaluated new software and documentation to make certain it met factory standards; after certification, engineers registered the software in department or factory reuse databases (libraries). Registered items required a key-word phrase to represent the functionality of the part or correspond to a specific object, as well as a brief "description for reusers" that explained the part's basic characteristics. These descriptions came in a code-like format, with specific names such as "slab" or "roller" converted to generalized notations like "MOVING_OBJECT." The cataloging procedures also required engineers to identify parts they expected to reuse frequently, such as common subroutines, and those they did not, such as job-oriented applications packages they might retrieve once at the beginning of development rather than daily. The factory kept the frequently reused software (source code and functional abstracts) on easily accessible disk files and the less frequently reused software on magnetic tape.

Evaluation criteria to determine which software parts were good enough for reuse focused on measures such as fitness, quality, clarity, abstractness, simplicity, coupling level (with other modules), completeness, "human interface" (module identification and algorithm descriptions), software interface, performance (response time), and internal configuration of the module. These criteria supported more specific, factory-oriented guidelines: The contents of a module (objects, relationships between objects, algorithms) had to be easily understandable to users who did not develop the code. The interfaces and requirements to execute the software (other code needed, language being used, operating system, automatic interrupts, memory needed, input/output devices, etc.) had to be clearly specified. The software had to be portable (executable on various types of computers) or transferable (modifiable to run on different computers, if not designed to be portable). Finally, the software had to be retrievable in a program library by people who were not familiar with it.

Toshiba had tools similar to Hitachi's EAGLE, NEC's SEA/I, and Fujitsu's PARADIGM, ACS-APG, and BAGLES/CAD to support reuse of designs and coded modules, as well as other functions. A corporate R&D group, formally attached to the Fuchu Works, developed these and more advanced tools. One example is a tool that facilitates the labelling and retrieving of reusable modules, relying on a special language Toshiba developed, called OKBL (Object-Oriented Knowledge-Based Language). A menus asks users a series of questions to define precisely what type of part they needed. For example, if developers want to see what functional modules are available for a particular application, they can enter the library for that application and then type "function" when asked for "Super-class." If the desired function is to scan analog data from a measuring instrument, they can type "scan" when the system asks for the subclass and "analog" when it asks for the kind of data. The tool then specifies the method of scanning, providing choices under other subclasses.

While the OKBL tool appeared relatively easy to use and will probably become more useful as Toshiba refines it, factory personnel, especially those having worked several years in the factory, seemed to rely mainly on manual techniques and experience -- printed catalogues of reusable software and knowledge gained from prior efforts -- to find software in libraries appropriate for new projects. Since Toshiba organized most reusable software as large packages in department libraries, and many departments used only a few packages to build most of their systems, developers quickly became familiar with the contents of different packages and do not seem to require much tool support. On the other hand, Toshiba reused only about 10% of delivered software across product departments. Thus better methods and tools to index and retrieve software components in a more generic fashion seems important to increase reuse further because this will make more designs, code, or packages accessible to members of different product departments who do not have a personal familiarity with either the application or existing software.

Even though Toshiba still had room to improve inter-departmental reuse, within the departments, management relied on an integrated set of incentives and controls to encourage project managers and personnel to take the time to write reusable software parts and reuse them frequently. At the start of each project, managers agreed to productivity targets that they could not meet without reusing a certain percentage of specifications, designs, or code. Design review meetings held at the end of each phase in the development cycle then checked how well projects met reuse targets, in addition to schedules and customer requirements. At the programmer level, when building new software, management required project members to register a certain number of components in the reuse databases, for other projects. Personnel also received awards for registering particularly valuable or frequently reused modules, and they received formal evaluations from superiors on whether they met their reuse targets. The SWB system, meanwhile, monitored reuse levels as well as deviations from targets both at the project and individual levels, and sent regular reports to managers.

To implement reuse objectives at the level of module design and coding, as opposed to the level of system design, Toshiba relied on another methodology, called "50SM" (50 Steps/Module).[10] As suggested by the name, the basic concept involved limiting the number of lines of code (steps) in one module to 50 or less (about one page), making the parts easier to understand and redeploy. The 50SM method covered three kinds of modules -- procedural (subroutines, functions, macros, etc.), data (files, variables or constants in memory, interface data, etc.), and packages (abstract data types, library programs, etc.). The factory also required a "technical description formula" to outline the external and internal module specifications as well as inter-module relationships. The 50SM presented a constraint in that the technique primarily supported structured design and reuse in procedural languages such as C, FORTRAN, COBOL, PASCAL, and Ada, rather than the use of newer object-oriented or logic-programming languages, which, for some applications, had advantages. Furthermore, in practice, only about half of new modules met the 50-line limit, although the remainder were close, usually within 100 to 150 lines. Nonetheless, this simple technique helped make reusable code and designs understandable, and worked well with tools such as code generators and editors.

Toshiba management reinforced its reusability strategy through training of new personnel in program development and in maintenance. In-house educational courses showed employees how to build software starting at higher levels of abstraction (requirements and design) and then working downward, which managers claimed increases the number of reused modules and the reuse frequency of a reused module.

At the same time, in program design, Toshiba trained personnel to abstract data, define standardized interfaces and parameters, and follow the factory procedures for cataloging and documenting. Furthermore, even if projects did not reuse particular programs, managers felt the same techniques for design, testing, documentation, and library registration that aided reuse simplified software maintenance, perhaps the most costly part of software development for programs with a long life span and frequent changes. It might even be the case that savings in maintenance alone made the extra effort required for reusability worthwhile.

Performance Improvements: Table 2 shows gross software productivity for the Fuchu Works and the Toshiba Software Factory from 1972 to 1985, and reuse rates in the programming or coding phase as well as new code estimates from 1977, when the software factory opened. Particularly striking is the rise in productivity (delivered equivalent-assembler source lines per person per month, excluding operating systems, utilities, and other basic software) and the obvious impact of increasing reuse, measured at the code level. Productivity rose from 1,390 lines per person per month in 1976 to over 3,100 in 1985, while reuse levels (lines of delivered code taken from existing software) increased from 13% in 1979 to 48% in 1985. About 60% of the code delivered in 1985 was in Fortran, 20% in high-level problem-oriented languages, and 20% in assembler; the equivalents of these are expressed in assembler, using Toshiba's internal conversion coefficients.

In terms of improvement rates, in the five years prior to the start of the factory, productivity gains appeared erratic, even dropping 12% in 1975. Fuchu software developers improved output 13% between 1972 and 1973, but nominal productivity in 1976 was still no higher than the 1973 level. In contrast, output per worker rose dramatically in 1978, the first full year of factory operations, while productivity doubled the 1975 level by 1981. Productivity improvements slowed considerably after 1981, but still averaged 6% or 7% annually. Production of new code actually followed a declining trend, since reusing more code required more time to read and modify the recycled parts, as well as more time to write code (and designs) for reuse. Toshiba's ability to recycle code rose dramatically in some years, such as between 1982 and 1983, but rose less rapidly afterward, stopping at just under 50% in 1985. The general leveling off of gross productivity improvements appeared to relate directly to a leveling off in reuse increases, although more data is needed to establish this trend more precisely.

Despite slowing gains in productivity and reusability, output per employee at the Software Factory already appeared high by the mid-1980s. The 3,130 lines of EASL source code per month per employee in 1985 translate into approximately 1,000 lines of Fortran-equivalent code. This monthly productivity estimated in Fortran equalled the Japanese average reported by Cusumano and Kemerer in their survey of Japanese and U.S. software projects, also adjusted for language differences, and exceeded by a large margin the U.S. adjusted average of about 600 Fortran-equivalent lines per work-month. Other adjustments to the Toshiba numbers produce figures that, while still laudable, are not quite as dramatic as nominal productivity suggested. Subtracting reused code from the 1985 data (which is not completely appropriate, since reusing code systematically requires time to write for reuse and to implement reuse), Toshiba employees averaged a still impressive 1,600 EASL or around 500 lines of new Fortran code per month. Adjusting for estimated overtime of 70 hours per month by recalculating for a 160-hour month, Toshiba personnel in 1985 delivered about 2,200 EASL (700 Fortran-equivalent), including reused code, and about 1,100 EASL (370 Fortran-equivalent), subtracting reused code. But while this latter number did not suggest that Toshiba had a huge advantage over other firms, it was still impressive given the size and complexity of the systems Toshiba delivered, the low level of defects, and the fact that half the factory employees were high-school graduates (albeit *Japanese* high-school graduates, with basic calculus, statistics, and a variety of science courses and other college-level material behind them).

Of course, data on productivity and reusability, as well as quality, are difficult to compare across firms. It is also difficult to measure reuse of designs, which is often more important than reuse of executable code, since designs can be modified more easily for different applications and machines. (The Toshiba Software Factory in 1985 claimed to reuse about one-third of its design documentation and generated much code more-or-less automatically from detailed designs, often in a flow-chart form, although it is not clear how design reuse affected the delivered EASL numbers.) In any case, since Toshiba appears to have collected its numbers consistently, its data provide some sense at least of changes in productivity and reuse over time at the factory. The numbers thus suggest strongly that (1) the factory emphasis on reusability doubled nominal productivity levels, and (2) reusability involved some costs, such as in overhead as well as in a decline in new-code productivity. On this last observation, Toshiba's internal studies of reuse rates, number of lines in modules changed when reused, and overall output per person, indicated that productivity was significantly improved only if about 80% of a module was reused without changes. If only 20% was used unchanged, the impact on overall productivity was negative. Between 20% and 80%, there was no noticeable impact on productivity. (Hitachi and Fujitsu reported similar findings in their studies of reuse and productivity.)

CONCLUSION: A SPECTRUM OF REUSE STRATEGIES

Reusability, and reuse-promotion strategies, actually fell across a spectrum. This ranged from no reuse, because of no commonality in applications or stability in program architectures and functions, to various degrees of accidental or ad hoc reuse and then to systematic reuse, with categorizations of software going beyond application domain libraries to indexing software modules and designs by functional content (Table 3). Any software producer might achieve occasional reuse if individuals remember what software they or other groups had built in the past that resembles what they want to do in the present. But factory approaches such as Toshiba's promoted more frequent and systematic reuse by planning across a series of projects as well as devising tools, libraries, reward and control systems, and training techniques to maximize the writing of reusable software components and their recycling as often as possible, at least within similar product families. In theory, design for reuse constituted an investment in proven components that developers and customers should want to recycle. In practice, reuse promotion faced technological as well as organizational obstacles whose solution required not merely technical analysis and support, but also management planning along with controls and incentives *above the level of the individual project*.

The best departments in Toshiba and other Japanese firms also managed reuse in relatively flexible ways. Developers mixed packages with new code -- thus giving customers the option of buying a semi-customized product rather than either a standardized package or a fully-customized system. Support tools emphasized the reuse of designs as much or more than executable code; this made it easier to modify features for different applications or reuse software across incompatible operating systems and hardware architectures. In addition, writing new designs and code for reusability, and depositing new parts in reuse libraries, created a continually expanding inventory of proven components that developers and customers should want to recycle. Raising reusability to the level of a factory policy also seemed to counteract objections by programmers to reusing other people's work and by project managers to absorbing overhead costs associated with designing, documenting, and testing software for general usage rather than for a specific application. In addition, Japanese companies made reuse-support systems available to in-house developers, subsidiaries and subcontractors, as well as users of their hardware, thus increasing the dissemination of these tools and techniques along with shifting some of the custom-programming burden to users.

Still, one must not overestimate the extent and nature of reuse in Toshiba or other Japanese firms. Like their counterparts around the world, Japanese facilities mainly confined the recycling of components to similar product families and used relatively simple classification techniques and support tools. They also reused most of their software in systems with relatively stable architectures and functions, such as control systems for power plants and automated factories, or common businesss applications for mainframe computers. As a result, opportunities for greater reuse clearly existed in Japan, especially across different types of products with technologies that identify the functional content of software components. But whatever the future of reuse in Japan or elsewhere, their accumulation of practical experience, as well as a steadfast commitment to exploiting reusability to improve both productivity and quality, suggested that Japanese software factories would remain among industry leaders in promoting systematic rather than accidental reuse.

Table 1: Approximate Breakdown of Reused Software in Toshiba

Breakdown		Comments
100%	TOTAL SYSTEM	Delivered lines of custom applications software for an individual project, excluding basic systems software. Size: 1,000,000 to 21,000,000 EASL
50%	REUSED SOFTWARE	
	(White-Box Designs)	Applications-specific packages or subsystems of design skeletons. Written, documented, and registered for reuse in product-department application libraries. Size: Usually 1000 to 10,000 EASL
	(Utilities, Tools)	Applications-specific utilities, tools, or other special programs imbedded in delivered software. Written, documented, and registered for reuse in product-department libraries mainly but also in factory libraries. Size: Usually 10,000 to 100,000 EASL.
	(Black-Box Modules)	Coded subroutines common to most software made in the factory. Written, documented, and registered for reuse from a central factory library. Approximately 10% of reuse. Size: Usually up uo 3000 EASL.
50%	NEW SOFTWARE	Plant- or customer-specific software, not considered reusable but written, documented, and registered in a similar manner for maintenance. Often written with advanced fourth-generation languages that generate code from menus or tables of application functions or specifications.

Source: Cusumano 1991, p. 260.

Table 2: Performance at Toshiba Software Factory

Notes:	EASL	=	Debugged and Delivered Equivalent Assembler Source Lines Per Programmer Per Month, Averaging All Projects in the Factory and Including All Phases and Manpower (Requirements Analysis through Maintenance)
	Index	=	Based on 1972 EASL productivity estimate (1230)
	Change	=	Percent increase or decrease over previous year
	Reuse %	=	Percent of delivered lines of code taken from existing software with little or no modifications (usually no more than 20% of a given module)
	New Code	=	EASL x [(100 - Reuse %)/100]
	Quality	=	Defects Per 1000 Lines of Delivered Code Converted to EASL

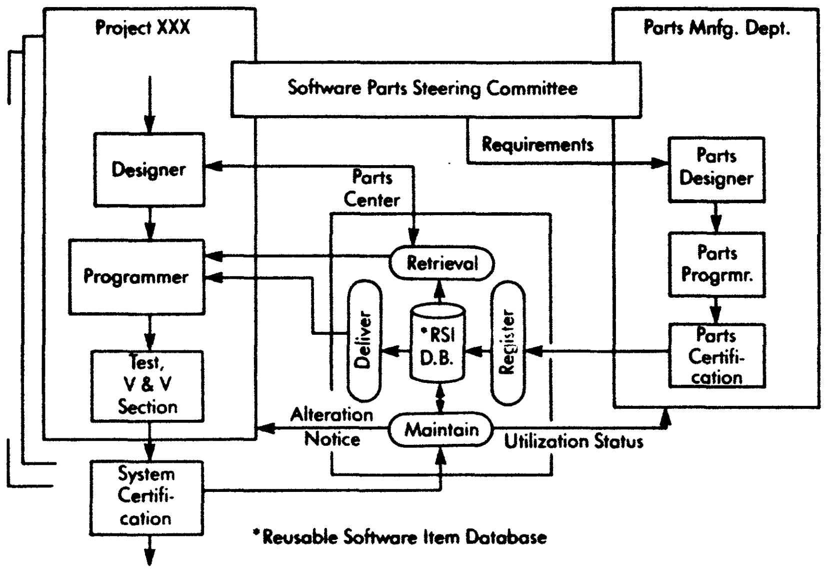
Year	Total EASL Delivered Per Person Per Month	Index/Change (100)	(%)	Reuse %	New Code (EASL)	Defects Per 1000 EASL	Employees (All Phases)
	PRE-FACTORY ESTIMATES:						
1972	1230	100	--	Data Not Available			
1973	1390	113	+13				
1974	1370	111	- 2				
1975	1210	98	-12				
1976	1390	113	+15				
	POST-FACTORY ESTIMATES:						
1977	Data Not Available						
1978	1684	137	--	--	--	7 to 20	1200
1979	1988	162	+18	13	1730		1500
1980	2072	168	+ 4	16	1740		1700
1981	2443	199	+18	29	1735		2050
1982	2595	210	+ 6	26	1920		2100
1983	2763	225	+ 7	41	1630		2150
1984	2931	238	+ 6	45	1612		2250
1985	3130	254	+ 7	48	1612	0.2 to 0.05	2300

Source: Cusumano 1991, p. 240.

Table 3: The Reusability Spectrum

Reuse Levels:	
None	No Commonality Among Projects No Stability in Program Architectures and Functions No Design Planning or Management for Multiple Projects No Reuse Support Tools And Libraries No Reuse Promotion Organization And Incentives Little or No Measurable Reuse
Some	Some Commonality Among Projects Some Stability in Program Architectures and Functions No Design Planning or Management for Multiple Projects No Reuse Support Tools And Libraries No Reuse Promotion Organization And Incentives Occasional But Still Ad Hoc or Accidental Reuse
More	Much Commonality Among Projects More Stability in Program Architectures and Functions Some Design Planning or Management for Multiple Projects Reuse Support Tools And Libraries For Application Domains No Reuse Promotion Organization And Incentives More Frequent But Not Maximum Reuse
Most	Much Commonality Among Projects Much Stability in Program Architectures and Functions Design Planning and Management for Multiple Projects Reuse Support Tools And Libraries By Software Content Reuse Promotion Organization And Incentives Systematic and Maximum Reuse

Figure 1: Toshiba's Reusability Promotion System



Source: Matsumoto 1987, p. 173. Reproduced with permission.

REFERENCES

1. Literature dealing with reuse is vast and growing. For two recent collections of studies, see Peter Freeman, Tutorial: Software Reusability, IEEE Computer Society Press, Washington, D.C., 1987, which includes many papers from a special issue on reuse in the September 1984 issue of IEEE Transactions on Software Engineering; and Ted Biggerstaff and Alan Perlis, Software Reusability: Volume I, Concepts and Models and Volume II, Applications and Experience, Addison-Wesley, Reading, Mass., 1989. Recent issues of IEEE Software (January 1991, January 1990, March 1987, January 1987) also contain useful articles on reusability.

2. M.D. McIlroy, "Mass Produced Software Components," in Peter Naur and Brian Randell, eds., Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Scientific Affairs Division, NATO, Brussels, January 1969.

3. Useful categorizations of types of reuse can be found in T. Capers Jones, "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 488-494; Ted Biggerstaff and Charles Richter, "Reusability Framework, Assessment, and Directions," IEEE Software, March 1987, pp. 41-49.

4. See, for example, Richard W. Selby, "Quantitative Studies of Software Reuse," in Biggerstaff and Perlis, Volume 2, pp. 213-234.

5. Most of the authors in the Freeman and Biggerstaff and Perlis anthologies are of this opinion.

6. See Michael A. Cusumano and Chris F. Kemerer, "A Quantitative Analysis of U.S. and Japanese Practice and Performance in Software Development," Management Science, Vol. 36, No. 11, November 1990, pp. 1384-1406.

7. These numbers reflect studies done at TRW as well as NEC, Fujitsu, Toshiba, and Hitachi between the late 1970s and the mid-1980s.

8. The following description of reuse-support tools and techniques at Hitachi, NEC and Fujitsu is based on technical articles and reports from these companies as well as site visits and interviews conducted during 1986-1989. For specific references and this and the following section, see Michael A. Cusumano, Japan's Software Factories: A Challenge to U.S. Management, Oxford University Press, New York, 1991.

9. The primary sources for this section are a series of interviews with and articles written by the founder of the Toshiba Software Factory, Dr. Yoshihiro Matsumoto, and other engineers and managers at the factory during 1986-1989. The specific sources are listed in Cusumano 1991, pp. 218-272. A particularly useful article in English is Yoshihiro Matsumoto, "A Software Factory: An Overall Approach to Software Production," in Freeman 1987, pp. 155-178.

10. Kazuo Matsumura et al., "Trend Toward Reusable Module Component: Design and Coding Technique 50SM," Proceedings of the Eleventh Annual International Computer Software & Applications Conference -- COMPSAC, IEEE Computer Society Press, Los Washington, D.C., 1987, pp. 45-52.