Incentive compatibility and systematic software reuse

Robert G. Fichman [a,*], Chris F. Kemerer [b,1]

[a] *Wallace E. Carroll School of Management, Boston College, 452B Fulton Hall, 140 Commonwealth Avenue, Chestnut Hill, MA 02167-3808, USA*
[b] *Katz Graduate School of Business, University of Pittsburgh, 278a Mervis Hall, Pittsburgh, PA 15260, USA*

Abstract

Systematic software reuse has emerged as a promising route to improved software development productivity and quality. Many large corporations have initiated systematic reuse programs, and many reuse frameworks have been developed to guide organizations in these efforts. Yet, in spite of this, systematic reuse in practice has been difficult to achieve. In this paper we argue that a key inhibitor has been the incentive conflict inherent in traditional programs of reuse. We reach this conclusion based on an analysis of interview data gathered from 15 projects across eight different sites in a company once viewed as a leader in the reuse movement. We found that one key contributor to the absence of widespread systematic reuse in this firm was a perception among project teams that reuse was incompatible with prevailing project team priorities and incentives, such as to complete projects on time and within budget. Based on this finding, we undertake a survey of different approaches to establishing reuse described in the literature, and analyze them to determine whether incentive incompatibility is inherent in the nature of software reuse for larger organizations. We conclude that it is not, and provide guidance on how such organizations can design an incentive-compatible program of reuse, i.e., one that generates a climate in which developers and teams view reuse as having a more favorable "value proposition" according to the prevailing incentives operating at the team level. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Software development; Systematic software reuse; Incentive compatibility

1. Introduction

Systematic software reuse represents a promising innovation in software development practice (Kim and Stohr, 1998; Krueger, 1992; Mili et al., 1995). Pervasive reuse of existing software components can lead to reductions in software development cost and cycle time, and can also promote software quality. In anticipation of these benefits, many leading edge high-technology companies undertook initiatives beginning in the early 1990s to promote systematic reuse within their firms (Griss and Wosser, 1995; Isoda, 1995; Joos, 1994; Poulin et al., 1993). In addition, some comprehensive frameworks for instituting systematic reuse were developed, including the Synthesis Method (Davis, 1993; Wartik and Davis, 1999) and the STARS program (Rada and Moore, 1997). Even so, it has been suggested that systematic software reuse remains difficult to achieve in practice (Card and Comer, 1994; Kim and Stohr, 1998; Mili et al., 1995; Rine and Sonnemann, 1998).

We conducted a study of the current state of software reuse at a large organization often mentioned in the reuse literature, examining 15 projects from eight different sites. We found that, while considerable local and informal reuse was reported on many of these projects, the goals of *formal* and *systematic* reuse had not been achieved. Perhaps most telling, the corporate level goal of inter-project, as-is reuse of components was rare in the projects examined. Given that the selection criteria for the study favored more reuse-oriented projects, this suggests a lower level of systematic reuse among the general population of projects at our data site.

During our investigation we talked to dozens of software professionals. Among other things, we posed the question of what was inhibiting systematic reuse both on particular projects and more broadly within this organization. Some of the more common observations were the following:

1. The top priorities on projects are to get them done on time and within budget, and reuse is seen to conflict with these priorities, thus creating disincentives to pursue reuse.

---

*Corresponding author. Tel.: +1-617-552-0471; fax: +1-617-552-0433.

E-mail addresses: fichman@bc.edu (R.G. Fichman), ckemerer@katz.pitt.edu (C.F. Kemerer).

[1] Tel.: +412-648-1572; fax: +412-648-1693.

2. Reuse decisions are delegated to individual developers who usually do not know what is available to potentially be reused.

3. "Not invented here" is the natural state for many programmers.

4. When assets are known to exist that meet the functional requirements of the project at hand, they often do not meet the operational or performance needs of the project (e.g., for rapid execution, for high reliability, for compact code).

5. Reuse across team boundaries raises difficult coordination and ownership issues that have not been resolved.

6. Version management issues have yet to be addressed.

This is a fairly typical list of reuse inhibitors (Griss, 1993; Isoda, 1995; Kim and Stohr, 1998; Mili et al., 1995) and we expect that many organizations attempting to establish systematic reuse today are struggling with these same sorts of issues. In this paper we focus on the first inhibitor, and consider what it would take to design a reuse program that would be viewed as more *incentive compatible*. The prevailing priorities at our data site, as with many firms, are to keep to schedules and budgets. Actions that conflict with these priorities, such as taking time and resources to build for reuse (especially when there is high uncertainty about how much time and resources will be required) create an incentive conflict, because promotions and other rewards are still based on keeping to schedules and budgets. We believe this inhibitor is especially problematic because, like many large organizations with dispersed software teams, this company has a prevailing culture that emphasizes project team autonomy and accountability for results. Because teams are afforded discretion in how they meet project priorities, they are less likely to adopt methods seen to conflict with these priorities. Thus, we believe that the incentive conflict created by *traditional* programs of reuse – focusing as they do on long-term objectives of the *organization as a whole* rather than the more immediate objectives of *individual project teams* – goes to the heart of the matter as for why broader reuse efforts here and elsewhere have not been sustained.

The contributions of this research are threefold. First, we present a snapshot of the state of reuse in one of the more prominent advocates of systematic reuse from the early 1990s (Section 2). Most experience reports portray reuse in an optimistic light, and while there is considerable value in understanding exemplars of successful reuse, we believe that it is also valuable to report on and derive lessons from situations where reuse did not unfold as had been expected. Next, we offer a new perspective on the issue of incentive compatibility and reuse (Section 3). Much has been written about the potential incentive conflict introduced by reuse, and many kinds of program "overlays" to counter incentive problems have been proposed (e.g., paying developers for supplying reusable components). However, we know of no other research that considers in detail the possibility that a program of reuse can be structured in a way to *avoid* many incentive problems in the first place. In our analysis we highlight the special role of costs associated with *reuse failure* – an idea notably lacking in prior discussion of reuse costs and benefits – in promoting incentive problems. Finally, we analyze alternative options for how to set up a reuse program along three key dimensions – the organizational structure, the process model, and the funding model – and analyze the options for each dimension on the issue of incentive compatibility (Section 4).

While the inspiration for our analysis has been findings from our data site, we believe it can be generalized to other large companies with dispersed and autonomous software development teams seeking the benefits of systematic reuse.

2. Overview of reuse at the research site

In the early 1990s, our case study site funded several initiatives to help institute formal and systematic software reuse. These initiatives included the development of standard reuse measures and associated economic models, initial funding for reusable "parts centers", development of an infrastructure for sharing components, development of incentive programs for supplying and using reusable components, and establishment of two corporate-level bodies to guide overall reuse policies. These initiatives were expected to replace the accidental, informal methods of locating, copying, and adapting software with a formal, systematic process of exchanging quality components that could be used as-is.

In 1996, we conducted a study of 15 recent software projects at eight sites (five in the US and three abroad). The main goals of the study were twofold: to identify the best of current practices related to software reuse, and to gather data to develop an activity-based costing model for software reuse (Fichman and Kemerer, 2002).

Before each site visit we administered a questionnaire to participating teams. This questionnaire covered the purpose of the project, staffing, development methods and tools, extent and nature of reuse, and project management practices. Once on site, we conducted semi-structured interviews with each project team to expand on and clarify material from the pre-visit questionnaires. Interviews would typically be held first with developers, then with the project manager, and finally with the project manager's supervisor, for a total of about 4 hours of interviews per project. In some cases we also interviewed non-project participants who nevertheless had insights into the practice of reuse at the site or more broadly within the firm. These included interviews with the champion for object-oriented technology at one site, the development methods coordinator at another, and

the reuse champion at a third. A summary was developed for all interviews and reviewed by the respondents for accuracy and completeness. In addition, with the participant's permission, most interviews were tape-recorded.

Based on an analysis of our questionnaire and interview data, we found that, while considerable local and informal reuse was reported on many of these projects, little progress had been made towards the goal of *formal* and *systematic* reuse throughout the corporation, and corporate level support for reuse had waned. Although the technology infrastructure was still operational, the internal policy groups and reusable parts centers had been defunded, "pay for components" schemes had been phased out, and in general, project teams were at liberty to practice as much or as little reuse as they saw fit. Our respondents cited two main reasons for phasing out of corporate-level initiatives in support of reuse: lower than expected participation, and staffing cuts resulting from the general downsizing of the firm in the early 1990s. As mentioned, a perception that aggressive pursuit of reuse would conflict with project priorities and corresponding incentives was the most frequently articulated inhibitor of reuse among software developers.

The actual practice of reuse among all but a few of the teams we studied may best be described as "opportunistic", a term coined by Davis to describe a state where there are few guidelines, processes, measures or tools in place to encourage or support reuse per se (see Appendix A, where the projects are identified as Project "A", "B" and so forth.) The inter-project, as-is reuse of components – the main goal of the broader reuse initiatives within this firm – was rarely practiced in the projects examined. Only three projects produced any components for use by other teams, and only three reused components from centralized libraries (beyond those that came bundled with the programming environment).

Even in spite of the absence of a *systematic* reuse process, some of the projects obtained substantial reuse in one form or another. We observed some common elements that tended to be present:
1. Strong philosophy of reuse from project managers.
2. Availability of good quality, directly applicable reusable assets that could not be ignored.
3. Small core development team (4–8 people).
4. Well understood, narrowly defined functional requirements.
5. Limited resources available to the team for reinvention.
6. Strong central architect who acts as reuse guardian.
7. Absence of special performance requirements (e.g., for efficient execution).

Many respondents though the time might be ripe to revisit the reuse issue now that better tools were available and more was known about reuse in general. It is not at all uncommon for early adopters to fail in their initial attempts to introduce a new way of doing things

(Fichman and Kemerer, 1997, 1999; Rogers, 1995). What distinguishes successful innovators in this event is the ability to "fail forward" – that is, use failure as learning opportunity to guide subsequent initiatives (Leonard-Barton, 1989). We believe that organizations with similar software development environments to our research site can succeed with systematic software reuse, but should reconsider how to achieve this objective. In the remainder of this paper, we present an argument for designing a reuse program that focuses squarely on incentive compatibility.

3. Incentive compatible reuse

The traditional view of reuse is concerned with designing a program that will *maximize the aggregate net benefits of reuse* – even if the program "runs at a loss" from the perspective of many individual projects. This idealized view is well presented by most reuse economic models (Banker and Kemerer, 1992; Barnes and Bollinger, 1991; Bollinger and Pfleeger, 1990; Frakes and Fox, 1996; Gaffney and Durek, 1989; Gaffney and Cruickshank, 1992; Henderson-Sellers, 1993; Lim, 1994; Pfleeger, 1996; Pfleeger and Bollinger, 1994; Poulin et al., 1993; Schimsky, 1992). It is true that any organization contemplating reuse must develop some assurance that reuse will pay off in aggregate. However, there is a danger in not looking further, because the adoption of reuse is, in reality, a two-part decision process (Leonard-Barton, 1988). The first is the organizational level decision to institute systematic reuse. The second is the decision to practice reuse at lower levels of the organization.

A well-established principle from the literature on innovation diffusion is that an adopting unit (a person, team, etc.) will only willingly adopt an innovation when it is viewed as providing advantages over current practices (Rogers, 1995). While the internal diffusion of an innovation does not conform to all of the assumptions of the classical diffusion model, we believe that the natural forces of diffusion will nevertheless hold considerable sway over whether a reuse program becomes successful.

Therefore, while from the perspective of the first level decision makers it is acceptable for some – or even most – projects teams to experience a net loss owing to the practice of reuse so long as aggregate net benefits are maximized, we believe this will not sit well with the project teams themselves in most organizations. We submit that unless a reuse program is designed so that reuse has a high probability of benefiting typical projects from the start – *from the point of view of those on the project teams themselves* – the practice of reuse is much less likely to be diffused throughout the organization.

As a result, we suggest that the traditional idealized view described above should be augmented or replaced

Table 1
Traditional vs. incentive-compatible reuse

Idealized reuse management heuristics	Incentive-compatible management heuristics
Encourage reuse where it can be practiced most efficiently from an organizational standpoint	Encourage reuse where it has the highest probability of a net positive local payoff
Expect teams to willingly assume the extra cost and risk associated with reuse production and consumption	Seek ways to shift the extra costs and risks of reuse production and consumption out of individual projects
Strongly encourage only the most leverageable forms of reuse – black-box, inter-organizational – over the least leverageable	Recognize and encourage all forms of reuse that have net benefits at a project level, including less leverageable forms (e.g., within-project factoring to isolate common code, and reuse of components previously developed by the project team itself)

with what we refer to as an *incentive-compatible* view. Under the incentive-compatible view, the goal of a reuse program is to maximize the probability that, *for any given project*, the benefits to that project will outweigh the costs as perceived by the project team. In Table 1 below, we contrast the traditional and incentive-compatible approaches by listing out some general heuristics consistent with each approach. In the sections to come, we use the incentive-compatible heuristics to guide specific choices on options for setting up a reuse program.

3.1. Incentives and control

Incentive problems occur in organizations when – because of differing goals – the values and preferences of people doing the work are inconsistent with those of management. For example, more senior managers may prefer actions that produce reusable components benefiting other teams in the future, while project teams may prefer actions that ensure the most positive outcomes on their own projects (Banker and Kemerer, 1992). And indeed, a lack of incentives to build for reuse was a frequently articulated inhibitor by our respondents, owing to a perception that reuse might conflict with getting projects done on time and within budget. As one project manager put it: "What is the value proposition for developers?"

There are two basic approaches to dealing with incentive problems in the adoption of new technologies. One is to attempt to implement the technology in such a way that it is most compatible with prevailing priorities and incentives. A second approach, the one more commonly advocated in the literature on reuse, is to pursue an idealized vision for how the technology should work, and to rely on *control systems* to counter whatever incentive problems may arise while implementing this vision. A body of theory – referred to as *control theory* – describes how to design control systems that minimize the total cost of incentive problems (Eisenhardt, 1985). Three basic approaches are available: (1) behaviorally based, (2) outcome-based, and (3) socialization-based. The behavioral approach monitors and rewards specific behaviors viewed as desirable. For example, a salesperson would be rewarded for showing up on time, being friendly to cus-

tomers, and so forth. In the context of reuse, this approach suggests rewarding individuals and/or teams based on their levels of reuse produced and consumed. This approach is often advocated in the literature (Isoda, 1995; Poulin, 1995) and has been employed at the data site in the past (i.e, the "pay for components" schemes).

The outcome-based approach measures and rewards the final outcomes desired by management. Returning to the sales example above, an outcome-based approach would pay salespeople on commission. The outcome-based approach to encouraging reuse would measure the aggregate, long-term productivity, quality, and cycle time benefits resulting from reuse and reward teams based on those outcomes. Outcome-based rewards have advantages, but often are not feasible either because outcomes are too difficult to measure or lie too much outside the control of the workers. It appears that both problems apply in the case of reuse, which may explain the lack of examples of this approach in the literature.

Under the socialization approach, management goes to the source of the problem and seeks to change the values of workers to be consistent with those of management (Ouchi, 1980). This approach to encouraging reuse suggests actions such as indoctrinating developers on the organizational-level benefits of reuse, and emphasizing that reuse is "just part of the job" like good design or thorough testing. This approach is frequently mentioned in the reuse literature as a precondition to establishing reuse (Goldberg and Rubin, 1995b; Meyer, 1995b).

All three of these approaches – behaviorally based, outcome-based, and socialization-based – assume that some particular structure of work is given and necessarily leads to severe goal misalignments. This is indeed the case with the idealized view of reuse, and may serve to explain the amount of attention in the reuse literature devoted to countering incentive problems (Isoda, 1995; Poulin, 1995). However, we believe that attempting to introduce a radical new process and at the same time attempting to manage severe incentive conflicts is a high risk undertaking for most organizations. Furthermore, little is known about the efficacy in practice of the various control systems that have been proposed for software reuse. For example, there seems to be a growing

awareness that the practice of paying individual developers cash bonuses for reusable parts may be counter productive, as, for example, it may encourage a mercenary attitude on the part of developers (Goldberg and Rubin, 1995a). This is consistent with observations from our own research site. One manager said the incentive program put in place at his site seemed to work well at first, but ended up a "disaster". Developers eventually started trying to game the system, and managers felt that it was sending the "wrong message" to teams, i.e., that reuse was something outside the normal course of a developer's responsibilities. It also created resentments because the highest reusing teams were those using new technology, so teams using old technology felt they were at a disadvantage. At another site, a manager noted that developers found the incentive program's implicit assumption that reuse was not just an ordinary part of the development process somewhat "insulting", because "of course they reuse whatever it makes sense to reuse". At a third a manager explained that while the program succeeded in generating a large number of components for inclusion in the reuse library, there were complaints about the quality of the resulting components, and the difficulty of reusing them.

The foundation of the incentive-compatible view we advocate here, by contrast, holds that work should be structured, where possible, so that *existing* values of developers naturally lead to the outcomes desired by management. When this has been done, it is not necessary to rely on sophisticated control systems with unproven efficacy. Assuming that all has been done that can be done to establish an incentive-compatible approach, we believe more modest control systems can be employed as complements to help minimize the costs of remaining goal misalignments. However, this is much different from relying on strong control systems to counter severe incentive problems from the start.

3.2. The uneven distribution of reuse costs and benefits as a source of incentive problems

It is the uneven distribution of reuse costs and benefits across projects that generates the bulk of incentive problems in reuse at the project team level. Much work has been done to identify the range of reuse costs and benefits (see for example, Poulin et al., 1993, as summarized in Table 2). Following Barnes and Bollinger these costs and benefits are divided into two parts: *reuse production*, or the generation of reusable parts, and *reuse consumption*, or the reuse of those parts (Barnes and Bollinger, 1991). Table 2 vividly illustrates the potential incentive problem of reuse at the project team level, as least when idealized reuse policies are employed. The costs of reuse are substantial, immediate, certain, and, for the most part, local to the project team. The benefits of reuse come later, are uncertain, and often accrue to another team (Banker and Kemerer, 1992).

3.3. Costs of reuse failure

While the costs of reuse have been described in the literature, curiously, none of the formal cost/benefit models adequately address a particular kind of cost, namely, the costs incurred when a developer attempts reuse of an existing component and fails. We refer to these costs as the *costs of reuse failure*. Failure can mean the component is not reused in the end, or the cost of reusing exceeds the cost of developing the component from scratch. Only in the Mili et al. model are any failure costs explicitly included, specifically, the cost of failed component searches (Mili et al., 1995). We believe that the costs of reuse failure have particularly strong implications for incentives.

The lack of attention to reuse failure costs is curious, because even in a well-run reuse program, reuse failure is apt to be a very common event. Frakes and Fox (1996) identify seven reuse "failure modes", each corresponding to a step in the process where reuse can fail. These include: (1) reuse not attempted, (2) part does not exist, (3) part not available, (4) part not found, (5) part not understood, (6) part not valid; and (7) part not integratable. They point out that even if the probability of any particular failure – given the absence of prior failure – is only 10%, the joint probability of failure overall is greater than 50% for each attempt. They then go on to argue that organizations should do a Pareto-style analysis to identify which failure modes are most prevalent, and target resources to reduce the incidence those failure modes.

This analysis is an excellent start, but does not account for the fact that although all failures have the same end *result* (i.e., a missed reuse opportunity) they do not all have the same *cost* to the reuse consumer. Failures that occur late in the process of attempting reuse are far more expensive than those that occur early in the process. Resistance due to the cost of reuse failure was mentioned by several projects at the data site. As one manager put it: "after a few bad experiences, you tend to revert to an anti-reuse attitude".

Failures that occur late in the reuse process are obviously costly in terms of the programmer's time. But there are other costs as well. These failures are also costly in terms of motivation and accountability. It is especially disheartening to face developing a component from scratch after all, once the reuse attempt has failed. The possibility of reuse failure also increases the variability of reuse related outcomes. This makes it difficult for programmers to predict the effort and time required to fulfill project objectives, and therefore, managers lose some of their ability to hold programmers accountable for meeting objectives.

Table 2
Reuse costs and benefits[a]

Reuse costs	Reuse benefits
Reuse producer costs	*Reuse producer benefits*
Cost of performing cost–benefit analysis	Added revenue due to income from selling reusable information
Cost of performing domain analysis	Added revenue from fees or royalties resulting from redistribution of
Cost of designing reusable parts	information
Cost of modeling/design tools for reusable parts	
Cost of implementing reusable parts	*Reuse consumer benefits*
Cost of testing reusable parts	Reduced cost to design
Cost of documenting reusable parts	Reduced cost to document (internal)
Cost of obtaining reuse library tools	Reduced cost to implement
Cost of added equipment for reuse library	Reduced cost to unit test
Cost of resources to maintain reuse library	Reduced cost to design tests
Cost of management for development, test and library	Reduced cost to document tests
support groups	Reduced cost to execute testing
Cost of producing publications	Reduced cost to product publications
Cost of maintaining reusable parts	Added revenue due to delivering product sooner to market place
Cost of marketing reusable parts	Reduced maintenance costs
Cost of training in software reuse	Added revenue due to improved customer satisfaction with product quality
	Reduced cost of tools
Reuse consumer costs	Reduced cost of equipment
Cost of performing cost-benefit analysis	Reduced cost to manage development and test
Cost of performing domain analysis	
Cost of locating and assessing reusable parts	
Cost of integrating reusable parts	
Cost of modifying reusable parts	
Cost of maintaining modified reusable parts	
Cost of testing modified reusable parts	
Fees for obtaining reusable parts	
Fees or royalties for reusing parts	
Cost of training on software reuse	

[a] See Poulin et al. (1993).

With the incentive-compatible view of reuse, the goal is not to reduce the incidence of the most prevalent failures (as suggested by Frakes and Fox), but *to reduce the incidence of those reuse failures that are the most costly to reuse consumers*. The example below illustrates the point.

To simplify the example, the seven failure modes are aggregated into three, as follows:
1. Reuse candidate not sought (corresponds to Frakes and Fox failure mode 1);
2. Reuse candidate not found (corresponds to Frakes and Fox modes 2, 3, and 4);
3. Reuse candidate cannot be used (corresponds to Frakes and Fox modes 5, 6, and 7).

Suppose the costs of deciding whether to seek a reuse candidate, the cost of searching for a candidate, and the cost of figuring out whether a candidate can be used are each $100. Then suppose an organization has a choice between two scenarios, which we will refer to as Scenario I and Scenario II (see Table 3). Finally, suppose both scenarios have the same overall reuse success rate (50%), but a different distribution of failure hazards. (The failure hazard at step n is the probability of failure in that step given it did not occur in steps 1 through $n-1$.) Scenario I has an increasing hazard of failure across each step: the chance of failing to seek reuse is 10%, the chance of not finding a candidate given one is

sought is 20%, and the chance of not being able to use any candidates given one or more are found is 30%.

Scenario II has a decreasing hazard of failure, from 30% to 20% to 10%. The overall the chance of success is the same for both scenarios $((1-0.1)*(1-0.2)*(1-0.3) = (1-0.3)*(1-0.2)*(1-0.1) = 0.5)$, and the "cost of success", is of course, the same on average $(100+100+100 = \$300)$. The difference is, under Scenario II, the expected "cost of failure" is 50% higher (\$111 versus \$75). This example actually understates the case, since the cost of deciding to attempt reuse is much less than searching, which is less than establishing whether a candidate that has been found is actually reusable.

It may be argued that this analysis is simply another example of the intuitive notion that if one is destined to fail, it is better to fail early. But, consider how organizations pursuing traditional approaches to systematic reuse often proceed. They invest in a centralized library facility. They set up incentive programs to encourage people to contribute anything that might be useful, but typically do not require extensive documentation, certification, and so forth. They end up with, in the words of one of our case study respondents, not a reuse library but with a "reuse junkyard". Since developers are reluctant to look in this junkyard, the organization sets up explicit reward systems to encourage them to do so, and

Table 3
Reuse failure modes

	Step 1	Step 2	Step 3	Cumulative	
Scenario I – increasing hazard					
Failure hazard in step N	0.10	0.20	0.30		
Failure probability in step N	0.10	0.18	0.22	0.50	(overall chance of failure)
Cost attempting step N	$100	$100	$100		
Cumulative cost if failure occurs in step N	$100	$200	$300		
Expected value of failure cost for each step	$10	$36	$65	$110.80	(expected total cost of failure)
Scenario II – decreasing hazard					
Failure hazard in step N	0.30	0.20	0.10		
Failure probability in step N	0.30	0.14	0.06	0.50	(overall chance of failure)
Cost attempting step N	$100	$100	$100		
Cumulative cost if failure occurs in step N	$100	$200	$300		
Expected value of failure cost for each step	$30	$28	$17	$74.80	(expected total cost of failure)

perhaps mandates some percent reuse consumption on projects (Cusumano and Kemerer, 1992; Isoda, 1995; Poulin, 1995). All of these actions lead to the "Type I" increasing hazard scenario – e.g., comparatively higher probability of finding "something" but a comparatively lower probability that "something" will work out in the end – and correspondingly, higher costs of reuse failure.

The incentive-compatible approach (to be discussed in the next section) would suggest a much different philosophy, where the library is built up slowly and carefully, parts are of high quality and well documented, and there are variants of the same functionality to support different performance requirements. This approach would be more likely to lead to the decreasing hazards scenario, and correspondingly lower costs of reuse failure for any given average probability of reuse success.

4. Dimensions of a systematic reuse program

Several authors have identified the main dimensions that comprise a systematic reuse program (Davis, 1993; Griss, 1993; Mili et al., 1995). In one representative framework, Kim and Stohr (1998) highlight six dimensions: organizational structural issues, software reuse process issues, issues related to reuse economics, developer behavior issues, software reuse technologies, and legal and contractual issues.

In this section, we present and analyze several options along the first three of these dimensions, which we label: (1) the organizational model for reuse (where and how the work of reuse is performed), (2) the reuse production model (where in the development lifecycle reusable components are produced) and, (3) the model for reuse funding. We have already discussed the fourth dimension, behavioral issues (Section 3.1), and we believe the remaining two dimensions do not have compelling implications for incentive compatibility, other than more advanced tools and a more favorable legal environment will unambiguously encourage reuse.

An assumption that underlies our discussion is that an organization has in place at least a basic toolset to support component sharing. Although library tools are not sufficient to ensure systematic reuse (Griss, 1993), without a toolset to support the storage and retrieval of components across organizations, it is unlikely that significant inter-team reuse will occur (Adams, 1991). It is also assumed that an organization has at least a minimal set of reuse metrics in place, otherwise there will be no basis for evaluating and directing reuse efforts (Frakes and Fox, 1996).

4.1. Reuse organizational model

The organizational model for reuse specifies who does the actual work related to reuse, and how. At one extreme is the *library* approach, where application teams do virtually all the work related to reuse, supported only by a passive library of components (Griss, 1993). At the other extreme is the *reuse factory* model, where a reuse team performs all development of components, and application teams are limited to assembling the results (Caldiera and Basili, 1991). Various intermediate forms have also been suggested, including the *product center* and *expert services* models described by Goldberg and Rubin (1995b), and the *team producer* model (Fafchamps, 1994). Finally, we have devised a model of our own, based on observations at our research site, which we label the *curator* model. We will discuss these models in increasing order of the extent to which the work of reuse is performed by dedicated reuse specialists rather than application project teams, i.e., (1) library, (2) curator, (3) product center, (4) expert services, (5) team producer, (6) reuse factory.

Of the projects observed, 11 of 15 adopted some elements of the library model. Of the remainder, two were porting projects with no particular organizational model for reuse (Projects A and B), one exhibited elements of the curator model (Project N), and one exhibited elements of the team producer model (Project O).

4.1.1. Library model

Under the library model, centralized libraries of components are set up and managed by one or more reuse specialists (Griss, 1993; Isoda, 1995; Poulin, 1995). Applications teams do all the work of developing reusable components, then offer them to the library. Offered components are accepted either "as-is", or with minimal certification. Applications teams do all the work on the consumption side of reuse as well. They browse the library for potential components, judge their applicability and reusability, and then select and adapt components judged to be promising.

4.1.2. Curator model

The curator model has the same basic structure as the library model, but has a different relationship to applications teams. As with the library model, one or more reuse specialists are assigned the task of managing repositories of components. But, in addition, the reuse specialists are given a strong quality certification role, a marketing agenda, and a budget for funding component acquisition. Under this model, the primary sources of reusable assets are not random components of unknown quality volunteered by project teams, but rather commissioned works from respected and willing producers. Teams that are in a particularly good position to produce reusable components are encouraged to make proposals to the curator team, which if accepted, will result in funding to defray the extra costs of developing reusable components.

In addition to funding commissioned works, the reuse team has a marketing role. Rather than the passive role of a librarian that answers questions if asked, the curator team has responsibility for finding out what applications teams are up to, and delivering information about what kinds of components are stocked in the repository to application teams. With this model the full costs of reuse consumption still fall on application teams, but with two important differences compared to the library model. First, the library contains a smaller number of higher quality components. Therefore the search costs are decreased, as are the odds of other reuse failure modes (e.g., "part not valid"). Second, due to the marketing efforts of the curator, potential reusers should have at least some idea of what kinds of components reside in the repository. This also reduces costs, particularly failure mode 1 (reuse not attempted where it should have been). Therefore, the curator model should substantially reduce the costs of reuse consumption (and of reuse failure) borne by project teams.

4.1.3. Product center model

The product center model calls for the creation of a reuse center staffed by reuse engineers and other reuse specialists (e.g., librarians, toolsmiths, methodologists) (Goldberg and Rubin, 1995b). This center is responsible for identifying and evolving a formalized reuse process model and for developing and maintaining a robust reuse infrastructure. The center is also responsible for seeing to it that application developers receive adequate training about reuse in general, and the sanctioned reuse process in particular.

In day to day operations, the product center has primary responsibility for identifying, acquiring, certifying, classifying and storing reusable components. Components may be acquired externally, from applications teams, or occasionally be developed by the product center staff. Tools are developed to support the storage and retrieval of components. Under this model, the full costs of reuse consumption – and in some cases, reuse production – fall on applications teams, who are expected to view development with and for reuse as a standard part of their software development process. This model is similar to our site's "parts center" initiative of the early 1990s.

4.1.4. Expert services model

The expert services model is structurally similar to the product center model (Goldberg and Rubin, 1995b). It, too, calls for the creation of a reuse center, with the same basic set of responsibilities. There is, however, a crucial difference. Under the expert services model, reuse engineers from the reuse center are actually "loaned out" to applications development teams to assist them with the practice of reuse. On the reuse consumption side, reuse engineers supply knowledge of what is potentially reusable, and assist in any necessary adaptations. On the production side, the reuse engineers identify where the project has the best opportunities to produce new components, and assists with the task of making those components reusable. At the end of the project, the reuse engineers take back knowledge of the domain, new assets to be made reusable, and knowledge of which existing assets are good "as is" and which need to be fixed or enhanced. This is also when reuse specialists might develop additional versions of components to satisfy different performance criteria. Since the responsibility for reuse is concentrated in the hands of a cadre of reuse specialists, this model does not require a sophisticated reuse infrastructure or a large scale reuse training program as does the product center model. Goldberg and Rubin found this model to be most successful of the five they observed practice (Goldberg and Rubin, 1995b).

4.1.5. Team producer model

Under both the team producer (Fafchamps, 1994) and reuse factory (Caldiera and Basili, 1991) models, specialized teams do the work of producing components. The main differences between the two models are the extent of centralization, and the assumed pervasiveness of reuse. Under the team producer model, decentralized reuse teams are created. Each team has its own supervi-

sor, and resides on the same organizational level as two or more application teams. The reuse team and the application teams they serve report through the same higher level manager. Under this model, as much or as little reuse is practiced as the higher level manager sees fit. This model has been the most successful of the four models employed by Hewlett-Packard (Fafchamps, 1994).

4.1.6. Reuse factory model

Under the reuse factory model – unlike with the team producer model – reuse is assumed to be ubiquitous. Applications are developed entirely by assembling reusable components (Caldiera and Basili, 1991). Reuse specialists employed in the reuse factory develop *all* software components, and application development teams are limited to two functions: (1) communicating project requirements to the reuse factory, (2) assembling the components supplied by the reuse factory.

4.1.7. Evaluation of organizational models on incentive compatibility

We believe that the library, product center, and factory models are inappropriate from an incentive-compatibility standpoint. The library model places the whole burden of reuse on projects. In addition, according to many commentators, the library model is destined to fail because it tends to generate large numbers of poorly documented components of questionable quality (Griss, 1993; Isoda, 1995). This is backed up by the early experiences of our case study site. The product center model, besides placing the full burden of reuse on project teams, assumes an organization already has systematic reuse as part of its standard development practice and culture. The factory model takes the reuse vision to its furthest extreme. While, in principle, excellent from a reuse cost standpoint, it is only feasible in situations where minimal levels of market responsiveness and project team autonomy are acceptable. In addition, the ability to lure the best people into staffing such an organization has been questioned (Mili et al., 1995).

The expert services and team producer model are the most promising choices from an incentive-compatibility point of view. First, both result in an efficient style of reuse, because they rely on reuse specialists to do most of the work of reuse. These specialists know how to develop for and with reuse, and have regular interaction with repositories of components. Second, both models

shift most of the costs and risk of reuse consumption and production outside application projects. The main disadvantages of the expert services model are the high costs of the reuse center itself, and possible resistance from application teams to the idea of having reuse specialists "sit in" on their projects. The disadvantage of the team producer approach is it requires a special set of circumstances: multiple teams reporting through the same manager that operate in a domain with a high potential for reusability.

Both the expert services and team producer models have high indirect costs, and require major structural changes to a development organization. When neither of these models are feasible, an organization might consider the curator model, a model that we rate as neutral on incentive-compatibility. Teams still bear all the schedule risk of attempting to produce components for reuse, but they receive external funding for the out-of-pocket costs, and some of the costs of reuse failure are actively managed. Only the most promising reuse opportunities are pursued, and the library only contains well documented, high quality parts.

4.2. Reuse production model

Closely linked to the organizational model for reuse are decisions about when and how to produce internally developed components for reuse. Two interrelated decisions determine the production model for reusable components. The first is whether to follow an a priori approach, where developers make decisions about potential for reusability based on an analysis of the domain, and build components to be reusable from the start, or the a posteriori approach, where the assessment of reusability and efforts to actually make components reusable occur after they have been developed and used in at least one instance (Meyer, 1995a). The second decision is where in the lifecycle process efforts to make components reusable should occur: before, during, or after the project in which components or their underlying abstractions are first encountered (Henderson-Sellers and Pant, 1993). Based on our review of the literature and site practices, we have identified five options for reuse production. These map to the above decisions as shown in Table 4 below.

Of the studied projects, eight (Projects A–H) had little or no emphasis on producing reusable components and did not conform to any of the above models. Of the

Table 4
Lifecycle models for reuse production

	Pre-project	In-project	Post-project
A priori domain analysis	1. Pre-project domain analysis	2. In-project domain analysis	
A posteriori generalization		3. In-project generalization	4. Post-project generalization
			5. Next-project generalization

remainder, six projects primarily employed the in-project generalization approach (Projects I–M) and one employed in-project domain analysis (Project N).

4.2.1. Domain analysis approaches

Pre-project domain analysis corresponds to the conventional notion of domain analysis as articulated by Prieto-Diaz and others (Moore and Bailin, 1991; Prieto-Diaz and Arango, 1991). Under this model, an attempt is made to identify abstractions and develop reusable work products with potential application to a family of projects residing in the same domain. These work products are supplied to application projects, where they may be further specialized or used "as is". We observed no instances of this model at the 15 case projects. In-project domain analysis is the same as pre-project domain analysis, except that it is initiated after the project has begun, based on the discovery of a compelling opportunity to develop broadly reusable components.

4.2.2. Generalization approaches

In-project generalization corresponds to the notion of building for reuse as espoused by the OO community (Meyer, 1995b). The idea here is that in the natural course of the project developers will take the extra time and effort required to make components generic and reusable, and the delivered system will incorporate these components. Post-project generalization is a model where, after project deliverables have been completed, the project team (or some other team) harvests those components that appear to have high potential for reusability and then makes those components reusable. Next-project generalization is the same as post-project generalization, except that decisions about reusability potential are made in the context of some subsequent project. To implement this approach, two kinds of reuse libraries are maintained, a primary library containing components already generalized for reuse, and a second library containing components judged to have potential reusability, but which have yet to be made reusable (Henderson-Sellers and Pant, 1993). Subsequent projects then search both libraries, and when a suitable candidate is found in the secondary library, the team undertakes the work required to make the component reusable.

4.2.3. Evaluation of production models on incentive compatibility

At first glance, pre-project domain analysis seems like an attractive option from an incentive-compatibility point of view. The costs of producing components are shifted out of any particular project, and in theory, the benefits of consuming components are available to all subsequent application projects with reasonable costs of consumption. Nevertheless, based on the literature and comments our respondents, it appears this is *not* an ef-

fective general model for production of components because it is difficult to develop components that are truly reusable outside the context of a particular project. As a result, domain analysis tend to produce the wrong components, or, the right components, but in a form that makes them difficult to reuse. The net result are large libraries that contain, in effect, irrelevant or low quality components – a situation which leads to high costs of reuse consumption and high costs of reuse failure by project teams.

Both the in-project domain analysis and in-project generalization appear to be poor choices from the incentive-compatible view – unless an organizational model is adopted that provides dedicated reuse specialists (e.g., expert services or team producer). This is because the full costs of reuse production are borne by individual projects. One of the most frequent comments we heard from project teams regarding barriers to reuse was that they did not have the time or resources to make components reusable during the course of application projects. It has been estimated that it costs from 11% to 380% more to make a component reusable compared to the single use cost (Lim, 1994). Furthermore, since the extent of these costs usually cannot be predicted in advance, in-project reuse production necessarily increases the variance of project outcomes. Some projects will make investments in reuse that fail to produce useable work products. Even if the expert services organizational model is employed, and reuse specialists do most of the work related to reuse production, inevitably some of the cost of reuse production will spill over to the project team.

Therefore, we believe that post-project and next-project generalization are the best choices for an incentive-compatible reuse production model. The main advantage of the post-project approach is that developers do the work of making components reusable when those components are fresh in their minds, and only a single library of high quality components need be maintained and searched. The main disadvantage is that the effort is expended without knowing when, or whether the resulting components will ever be reused. The main advantage of the next-project approach is that "just in time" is employed to the production of components, and as a result, the primary reuse library contains only components proven to be reusable in at least one instance. The main disadvantage is that extra effort to make components reusable occurs on the critical path of subsequent projects. Also, there is no assurance that a developer familiar with candidate components will be available when the time to generalize them has arrived.

4.3. Funding and cost management

In a traditional development environment, a direct relationship exists between the deliverables produced by a project and most of the activities required to produce

those deliverables. As a result, funding is simply matter of charging project sponsors for the direct effort associated with development. Systematic reuse breaks down this traditional relationship, because many costs – producing and certifying reusable components, maintaining reuse libraries, performing domain analysis to name a few – benefit multiple projects. As a result, if systematic reuse is to be supported over the long term, new approaches to funding and management are required that reflect the inherently multi-project nature of reuse.

Based on a review of the literature, we identified four alternative approaches to funding the indirect recurring costs of reuse:
1. Traditional overhead funding.
2. "Tax" funding.
3. Payments for components (and services).
4. Activity based costing/activity based management.

The vast majority of the projects we studied (A–M) had no separate funding for the costs associated with reuse, but rather absorbed these costs into the direct costs of the project itself. Of the remaining two projects, one obtained overhead funding for the costs of developing reusable components (Project N) and one initiated the practice of charging other teams for use of the components they had developed (Project O).

4.3.1. Overhead funding

The simplest approach to funding the indirect costs of reuse is simply to view this cost as an overhead expense, and to pool it with other overhead expenses. All personnel and other expenses relating to reuse efforts not tied directly to some particular project would be paid for from overhead accounts.

4.3.2. Tax funding

The "tax" funding approach is similar to overhead funding. Under this approach, the indirect costs of reuse are paid for by a flat tax levied on all applications projects (Goldberg and Rubin, 1995b;Meyer, 1995b). The basis for the tax could be either as a percentage of total direct project costs, or total direct labor costs. Projects are charged this tax whether they practice reuse or not, but can be granted a partial "tax rebate" to the extent that they contribute reusable components (Meyer, 1995b).

4.3.3. Pay-for-components

A third option suggested in the reuse literature is to charge applications teams directly for the components and services they consume (Meyer, 1995b; Poulin et al., 1993). One challenge here is that many of the costs of setting up a reuse program and acquiring or developing a base of components are incurred well before the opportunities for cost recovery through sales of those components. Therefore some mechanism, such as a *cost sharing bank* must be devised to fund the up-front costs of reuse (Bollinger and Pfleeger, 1990).

4.3.4. Activity based costing

We have devised an alternative approach to funding and managing the indirect costs of reuse based on the principles of activity based costing (ABC) (Estrin, 1994; Ness and Cucuzza, 1995; Turney, 1991). This approach is described in more detail in a related article (Fichman and Kemerer, 2002). Here we only summarize and comment on the approach.

ABC works as follows. Rather than viewing products as being direct "consumers" of overhead and indirect resources, the repetitive activities required to actually produce products are introduced to stand in between products and costs. These activities are viewed as the consumers of overhead, and in turn, products are viewed as consuming activities. ABC provides a systematic method for mapping overhead costs to activities and from activities to products. Under this approach teams would be charged according to the degree of consumption of indirect reuse support activities.

4.3.5. Evaluation of funding models

The overhead funding model is quite consistent, in principle, with an incentive-compatible approach, because all of the costs covered by overhead are incurred whether or not reuse is practiced on a particular project, and are therefore extracted from the project level cost/benefit equation for reuse. Nevertheless, we feel traditional overhead funding is not viable long term because it provides no basis for evaluating and managing the effectiveness of indirect reuse activities, and because it sends the message that reuse activities are not part of the value-added process of software development.

The tax funding approach preserves and extends the incentive-compatible advantages of the traditional overhead funding approach, and counters some of the disadvantages. As with overhead funding, indirect reuse costs are incurred whether or not reuse is practiced, so these costs are extracted from the project level reuse cost/benefit equation. In addition, this approach actively encourages the supply of components, because the tax rebates improve the project level cost/benefit equation. And because these rebates benefit the team and project's client rather than any one individual, they avoid the mercenary quality of individual incentives. Nevertheless, as was the case with overhead funding, this approach provides no basis for evaluating and managing the efficiency and effectiveness of indirect reuse activities. Therefore, we believe that sooner or later it must be replaced with one of the remaining two remaining options.

We think the pay-for-components approach holds promise once systematic reuse has been established, because it brings market forces to bear on the problem of managing reuse costs. However, unless the components are actively supported and marketed, as under the team producer model or expert services models, we believe this a poor initial approach to funding reuse,

Table 5
Three key dimensions of systematic software reuse

Organizational model	Production model	Funding and cost management
1. Library	1. Pre-project domain analysis	1. Overhead
2. Curator	2. In-project domain analysis	2. Reuse tax
3. Product center	3. In-project generalization	3. Pay-for-components
4. Expert services	4. Post-project generalization	4. Activity based costing
5. Reuse Factory	5. Next-Project generalization	
6. Team producer		

Table 6
Summary evaluation of the dimensions of systematic reuse

Dimension	Most incentive compatible	No strong relationship	Least incentive compatible
Organizational model	Expert services, team producer	Curator	Library, product center, reuse factory
Production model	Post-project generalization, next-project generalization	Pre-project domain analysis, in-project domain analysis/generalization (if performed by reuse specialists)	Pre-project domain analysis, in-project domain analysis/generalization (if performed by application team)
Funding	ABC, reuse tax	Overhead, pay-for-components (if actively marketed and supported)	Pay-for-components (if not actively marketed and supported)

Table 7
Dimensions of software reuse on projects

Projects	Organizational model	Production model	Funding and cost management
A, B	None (porting a product)	No production of components	None
C, D, E, F, G, H	Library (some elements)	In-project generalization (low)	None
I, J, K, L, M	Library (some elements)	In-project generalization (medium-high)	None
N	Curator (some elements)	In project domain analysis	Overhead
O	Team producer (some elements)	In project generalization, next project generalization	Pay-for-components

because from a incentive-compatible view, charging for components strongly discourages the consumption of those components.

The advantages and disadvantages of the ABC approach are similar to those for the pay-for-components approach. The main advantage is that the approach is sustainable over the long term because it provides mechanisms for evaluating and managing the efficiency and effectiveness of indirect reuse activities, and it provides a mechanism for allocating the costs of reuse to projects that benefit from these indirect activities. The main disadvantage is that an ABC program is itself difficult and costly to establish. Also, projects are, in a fashion, charged for reuse consumption, although ABC can be employed with a lesser emphasis on cost allocation, and a greater emphasis on activity improvement.

4.4. Summary

We began this section with the argument that organizations characterized by dispersed, autonomous software development teams – like our data site and many other large software organizations – should adopt a incentive-compatible approach to establishing systematic software reuse. We described what it means for a reuse program to be incentive-compatible, then described alternative options for three key dimensions of a systematic software reuse program (summarized in Table 5 below). Finally, we analyzed the attractiveness of the various options for each dimension from a incentive-compatible point of view. Our conclusions about which options were most consistent and inconsistent with the incentive-compatible approach are summarized in Table 6. Although a few of the projects we studied contained one or two incentive-compatible elements, most did not – and all included at least some elements inconsistent with project philosophy (see Table 7).

5. Summary and conclusions

Like many leading edge technology-based companies, our data site embarked upon an ambitious plan for instituting systematic reuse in the early 1990s. Key elements of this plan included establishment of new organizations to create and evolve their overall reuse policies; establishment of reusable parts centers with supporting tools to provide an infrastructure for sharing components; development of incentive programs to encourage the production and consumption of reusable components; and development of standard reuse measures and associated economic models to judge progress in achieving program goals.

We conducted a study of 15 recently completed software development projects to find out, among other things, the current status of reuse there, and the extent to which these initiatives had achieved intended results. We found, contrary to our initial expectations, that the practice of reuse is informal and ad-hoc, and tended to be driven by the immediate needs and opportunities of individual projects. The inter-project, as-is reuse of components – the main objective of the reuse initiatives of the early 1990s – was especially rare among the teams we studied. Furthermore, we found that larger reuse initiatives had, for the most part, been discontinued.

After talking to dozens of software developers and managers, we concluded that the overall reuse program was incompatible with the prevailing software development culture and incentives – one that emphasized project team autonomy and accountability for results on individual projects – and therefore had the cards heavily stacked against it from the start. This observation lead us to reconsider whether the conventional reuse wisdom – that systematic software reuse requires that software teams adopt an organizational level point of view – was, in fact, correct. Based on a detailed analysis of alternative approaches to establishing systematic reuse, we concluded that the conventional wisdom should be amended, and that companies should strongly consider an incentive-compatible philosophy when designing a program of systematic software reuse.

We then set about developing some guidelines for organizations that might wish to establish a incentive-compatible approach to reuse. We began by defining what it means for software reuse to be incentive-compatible, and developed the novel idea that a key goal of this approach is to effectively manage the costs of reuse failure. We then developed a framework laying out three dimensions of systematic reuse, each with several options, and analyzed the alternative options within each dimension to determine which were the most incentive-compatible. Along the way we introduced a novel organizational model for reuse we labeled the "curator" model.

We expect that many organizations may be in the process of evaluating – or, like our site, reevaluating – alternative approaches to achieving systematic software reuse. We believe that for many of these organizations, the key to success with reuse will mean giving serious thought to structuring the program of reuse to minimize incentive compatibility problems.

Acknowledgements

Appendix A. Projects featuring software reuse

Prj	Description technologies, use of OO	Staff, size, technologies, use of OO	Reuse highlights		
			Description	Consumption	Production
A	Develops a commercial middleware product designed to allow transparent access to data in a heterogeneous database environment	Staff: 12 at peak; Effort: 288 staff months; Techs: 'C'; OO: none	Reuse mainly takes the form of porting. Product was built using workproducts (code, test cases, interface design) from a pre-existing related product	High	None
B	Develops a commercial product that provides a mainframe-like Cobol environment on workstations	Staff: 4–8; Effort: 48 staff months; Techs: 'C'; OO: none	Reuse mainly takes the form of porting and factoring. Reused 100 KLOC from a front end parser. Reused design and module structure	High	None
C	Develops Unix-based workstation tools for internal I/O subsystem teams. Provides single user interface to the toolset independent of compiler and platform	Staff: 9; Effort: 45 staff years; Techs: C and various home grown facilities; OO: used extensively, shallow hierarchies	Beyond use of some standard generic classes maintained in a team library, reuse is confined to a few developers that have the natural inclination to seek reuse	Medium (from team class library)	Low (to team library)

Prj	Description technologies, use of OO	Staff, size, technologies, use of OO	Reuse highlights		
			Description	Consumption	Production
D	A "clean-up" project to improve the integration of a database administrator toolset and to provide a consistent look and feel	Staff: 18 (department); Effort: 20 staff years (our est.); Techs: C++; OO: employed extensively, emphasis on encapsulation	Reuse mostly from toolset library and a domain specific set of "common classes" maintained by the team	Medium (mainly GUI classes from toolset library, team library)	Low (to team library)
E	Develops a generalized interface to be used by multiple protocol stacks and device drivers	Staff: 6 avg., 8 peak; Effort: 40 staff months; Techs: OO enhanced 3GL; OO: employed extensively	Goal was to develop a set of functionality need by multiple parts of the product (i.e., within product code factoring). Results not generically reusable	Low (from centrally maintained collection classes)	High (team library)
F	Develops facilities to help applications run in a clustered environment. One of about 80 subsystems in a large product family	Staff: 5; Effort: 40 staff months; Techs: OO enhanced 3GL; OO: employed selectively; one level of inheritance	Reuse mostly takes the form of shared OS services and macros, and some internal factoring	Medium (from labwide library)	Low (to team library)
G	Develops enhancements to an OS workload manger (i.e., to dynamically start and stop server address spaces for client work)	Staff: 18; Effort: 18 staff months; Techs: Case tool, a 3GL; OO: employed selectively via the CASE tool	Reuse mostly takes the form of internal factoring of code (via inheritance) and reuse of CASE tool components and centrally maintained macros	Medium (from labwide library, generator)	Low (to team library)
H	Develops new functionality to a TP monitor to allow a single point of control for monitoring a network of computer clusters	Staff: 7 avg., 14 peak; Effort: 35 staff years; Techs: C, C++, DSOM; OO: employed selectively. Two levels of subclasses typically	Reuse from toolset library and internal factoring	Low (from toolset library)	Low (to team library)
I	Develops a new release of a distributed data archival, backup and recovery system running in client–server architecture across all major client and server platforms	Staff: 8 avg., 15 peak; Effort: 480 staff months; Techs: C++ compilers, class libraries; OO: mainly for platform specific code and GUIs. One level of inheritance	Main reuse goal to minimize platform specific-code, and confine platform-specific code to separate subsystems. Produced reusable components for GUI and other common services	Low (GUI classes from toolset library)	Medium (to team library)
J	A large project to implement an OS on a new platform	Staff: 250; Effort: 750 staff years; Techs: C++, many custom tools; OO: employed extensively, emphasis on encapsulation	Reuse takes many forms: common services, inheritance reuse, interface reuse	Medium (from team libraries)	Medium (to team libraries)

Prj	Description technologies, use of OO	Staff, size, technologies, use of OO	Reuse highlights		
			Description	Consumption	Production
K	Develops a set of TCP/IP and Internet related applications for Windows platforms	Staff: 8 avg., 12 peak; Effort: 20 staff years (our est.); Techs: C++; Windows MFC, Java, Visual Basic others; OO: employed extensively	Reuse mostly takes the form of incidental sharing of code among programmers. In one instance another team was able to reuse a 10 KLOC component this team had developed	Low (from team library)	Medium (to team library, once to other team)
L	Adds a data access layer to the data site's C++ product to allow OO programmers to access legacy tables in relational databases, and have them appear as persistent objects	Staff: 6–7 (core team); Effort: 18 staff years (our est.); Techs: C++, SQL; OO: employed extensively, 3–4 levels of inheritance	Reuse from toolset library (common classes), team specific libraries. Also developed a code generator	High (from toolset library, team library)	High (to team libraries, code generator)
M	Develops new functionality to allow companies using Smalltalk to encapsulate mainframe transactions	Staff: 5 avg., 7 peak; Effort: 90 staff months; Techs: Small-talk; OO: employed extensively	Extensive reuse from toolset library. About 75% of new classes were sub-classed from existing classes (not including "object")	High (from toolset library)	High (to team library)
N	Develops two components for a new version of an OS, the recovery manager function and the logger function	Staff: 30 avg., 35 peak; Effort: 180 staff years; Techs: OO enhanced 3GL; OO: employed selectively. Inheritance used increasingly sparingly as time went on	Early in the project, it was recognized that much of the domain was generic. Got funding to develop generic reuse library.	None	High (generic library created for other teams)
O	Takes functionality needed in several components of an OS communication server and develops it as common services to be used by those components	Staff: 5 avg.; 8 peak; Effort: 40 staff months; Techs: C++; OO: employed extensively	Intended from the start to develop components to be used by other parts of the product (i.e., within product code factoring). Decided to make components be more generically reusable. Components are "in the plan" for six other products	Low (from toolset library)	High (for other teams)

References

Adams, S., 1991. The economics of software reuse (panel). Paper presented at the Third Annual Conference on Object Oriented Programming, Languages and Systems (OOPSLA), Phoenix AZ.

Banker, R.D., Kemerer, C.F., 1992. Performance evaluation metrics for information systems development: a principal-agent model. Information Systems Research 3 (4), 379–400.

Barnes, B.H., Bollinger, T.B., 1991. Making reuse cost-effective. IEEE Software 8 (1), 13–24.

Bollinger, T.B., Pfleeger, S.L., 1990. Economics of reuse: issues and alternatives. Information and Software Technology 32 (10), 643–651.

Caldiera, G., Basili, V.R., 1991. Identifying and qualifying reusable software components. IEEE Computer 24 (2), 61–70.

Card, D., Comer, E., 1994. Why Do So Many Reuse Programs Fail? 11 (9), 114–115.

Cusumano, M., Kemerer, C.F., 1992. A quantitative analysis of US and japanese practice and performance in software development. Management Science 36 (11), 1384–1406.

Davis, T., 1993. The reuse capability model: a basis for improving an organization's reuse capability. IEEE Transactions on Software Engineering, 126–133.

Eisenhardt, K. (1985). Control: organizational and economic approaches. Management Science 31 (2).

Estrin, T.L., 1994. Is ABC suitable for your company?. Management Accounting 75 (10), 40–45.

Fafchamps, D., 1994. Organizational factors and reuse. IEEE Software 11 (9), 31–41.

Fichman, R.G., Kemerer, C.F., 1997. Object technology and reuse: lessons from early adopters. IEEE Computer 30 (10), 47–59.

Fichman, R.G., Kemerer, C.F., 1999. The illusory diffusion of innovation: an examination of assimilation gaps. Information Systems Research 10 (3), 255–275.

Fichman, R.G., Kemerer, C.F., 2002. The activity based costing for component based software development. Information Technology and Mangement, forthcoming.

Frakes, W.B., Fox, C.J., 1996. Quality improvement using a software reuse failure modes model. IEEE Transactions on Software Engineering 22 (4), 274–279.

Gaffney Jr., J.E., Durek, T.A., 1989. Software reuse – key to enhanced productivity: some quantitative models. Information and Software Technology 31 (5), 258–267.

Gaffney, J.E.J., Cruickshank, R.D., 1992. A general economics model of software reuse. Paper presented at the International Conference on Software Engineering, Melbourne, Australia.

Goldberg, A., Rubin, K.S., 1995a. Failing with Objects, Succeeding with Objects (Decision Frameworks for Project Management). Addison-Wesley, Boston, MA.

Goldberg, A., Rubin, K.S., 1995b. Succeeding with Objects (Decision Frameworks for Project Management). Addison-Wesley, Boston, MA.

Griss, M.L., 1993. Software reuse: from library to factory. IBM Systems Journal 32 (4), 548–566.

Griss, M.L., Wosser, M., 1995. Making Reuse Work at Hewlett-Packard. IEEE Software 12 (1).

Henderson-Sellers, B., 1993. The economics of reusing library classes. Journal of Object Oriented Programming (July–August), 43–49.

Henderson-Sellers, B., Pant, Y.R., 1993. Managing reuse across the system lifecycle. Object Magazine (December).

Isoda, S., 1995. Experiences of a software reuse project. Journal of Systems and Software 30 (3), 171–186.

Joos, R., 1994. Software reuse at motorola. IEEE Software 11 (9).

Kim, Y., Stohr, E.A., 1998. Software reuse: survey and research directions. Journal of Management Information Systems 14 (4), 113–147.

Krueger, C.W., 1992. Software reuse. ACM Computing Surveys 24 (2), 131–183.

Leonard-Barton, D., 1988. Implementation characteristics of organizational innovations. Communication Research 15 (5), 603–631.

Leonard-Barton, D., 1989. Implementing new production technologies: exercises in corporate learning. In: Von Glinow, M.A., Mohrman, S. (Eds.), Managing Complexity in High Technology Industries: Systems and People. Oxford University Press, Oxford.

Lim, W.C., 1994. Effects of reuse on quality, productivity, and economics. IEEE Software 11 (5), 23–30.

Meyer, B., 1995a. Nature and Nurture: Making Reuse Succees, Object Success. Prentice-Hall, New York.

Meyer, B., 1995b. Object Success. Prentice-Hall, New York.

Mili, H., Mili, F., Mili, A., 1995. Reusing software: issues and research directions. IEEE Transactions on Software Engineering 21 (6), 528–562.

Moore, J.M., Bailin, S.C., 1991. Domain analysis: framework for reuse. In: Prieto-Diaz, R., Arango, G., (Eds.), Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, CA, pp. 179–203.

Ness, J.A., Cucuzza, T.G., 1995. Tapping the full potential of ABC. Harvard Business Review 73 (4), 130–138.

Ouchi, W.G., 1980. Markets, bureaucracies, and clans. Administrative Science Quarterly 25 (1), 129–141.

Pfleeger, S.L., 1996. Measuring reuse: a cautionary tale. IEEE Software 13 (7), 118–127.

Pfleeger, S.L., Bollinger, T.B., 1994. The economics of reuse: new approaches to modeling and assessing cost. Information and Software Technology 36 (8), 475–484.

Poulin, J., Caruso, J., Hancock, D., 1993. The business case for software reuse. IBM Systems Journal 32 (4), 567–594.

Poulin, J.S., 1995. Populating software repositories: incentives and domain-specific software. Journal of Systems Software 30 (3), 187–199.

Prieto-Diaz, R., Arango, G., 1991. Introduction and overview: domain analysis concepts and research directions. In Prieto-Diaz, R., Arango, G. (Eds.), Domain Analysis and Software System Modeling. IEEE Computer Society Press, Los Alamitos, CA, pp. 9–32.

Rada, R., Moore, J., 1997. Standardizing reuse. Communications of the ACM 40 (3), 19–23.

Rine, D.C., Sonnemann, R.M., 1998. Investments in reusable software: a study of software reuse investment success factors. Journal of Systems and Software 41 (1), 17–32.

Rogers, E.M., 1995. Diffusion of Innovations, fourth ed. The Free Press, New York.

Schimsky, D., 1992. Software reuse – some realities. Vitro Technical Journal 10 (1).

Turney, P.B., 1991. Common Cents. Cost Technology, Hillsboro, OR.

Wartik, S., Davis, T., 1999. A phased reuse adoption model. Journal of Systems and Software 46 (1), 13–23.

Robert G. Fichman is an assistant professor of information technology at the Carroll School of Management, Boston College. He received his BS and MS degrees in Industrial and Operations Engineering from the University of Michigan and his Ph.D. in Information Technologies from the MIT Sloan School of Management. His research analyzes the distinctive patterns of diffusion exhibited by IT innovations, and draws implications for innovation suppliers and adopters on how to better manage the introduction of new technologies. He has published in *IEEE Computer*, *Information Systems Research*, *Sloan Management Review*, *Management Science*, and other scholarly journals. Prior to getting his Ph.D., Dr. Fichman worked for several years as an IS applications development manager for a leading telecommunications company, and as an IT industry consultant.

Chris F. Kemerer is the David M. Roderick Chair in Information Systems at the Katz Graduate School of Business, University of Pittsburgh. Previously, he was an Associate Professor at MIT's Sloan School of Management. He received the B.S. degree from the Wharton School at the University of Pennsylvania and the Ph.D. degree from Carnegie Mellon University. His current research interests include management and measurement issues in information systems and software engineering, and he has published articles on these topics in a number of professional and academic journals, including *Communications of the ACM*, *IEEE Computer*, *IEEE Software*, *IEEE Transactions on Software Engineering*, *Information Systems Research*, *Management Science*, *MIS Quarterly*, *Sloan Management Review*, and others. He is a former Principal of American Management Systems Inc., a Virginia-based software development and consulting firm. Dr. Kemerer serves or has served on the editorial boards of the *Annals of Software Engineering*, *Communications of the ACM*, *Empirical Software Engineering*, *IEEE Transactions on Software Engineering*, *Information Systems Research*, the *Journal of Organizational Computing*, the *Journal of Software Quality*, and *MIS Quarterly*. He is currently the Departmental Editor for Information Systems at *Management Science*.