Reusing Software: Issues and Research Directions

Hafedh Mili, Fatma Mili, and Ali Mili

Abstract—**Software productivity has been steadily increasing over the past 30 years, but not enough to close the gap between the demands placed on the software industry and what the state of the practice can deliver [22], [39]; nothing short of an order of magnitude increase in productivity will extricate the software industry from its perennial crisis [39], [67]. Several decades of intensive research in software engineering and artificial intelligence left few alternatives but software reuse as the (only) realistic approach to bring about the gains of productivity and quality that the software industry needs. In this paper, we discuss the implications of reuse on the production, with an emphasis on the technical challenges. Software reuse involves building software that is reusable by design and building *with* reusable software. Software reuse includes reusing both the products of previous software projects and the processes deployed to produce them, leading to a wide spectrum of reuse approaches, from the building *blocks* (reusing products) approach, on one hand, to the *generative* or reusable *processor* (reusing processes), on the other [68]. We discuss the implication of such approaches on the organization, control, and method of software development and discuss proposed models for their economic analysis.**

Software reuse benefits from methodologies and tools to:

1) **build more readily reusable software and**
2) **locate, evaluate, and tailor reusable software, the last being critical for the building blocks approach.**

Both sets of issues are discussed in this paper, with a focus on application generators and OO development for the first and a thorough discussion of retrieval techniques for software components, component composition (or bottom-up design), and transformational systems for the second. We conclude by highlighting areas that, in our opinion, are worthy of further investigation.

Index Terms—**Software reuse, managerial aspects of software reuse, software reuse measurements, building reusable components, OO software development, software component retrieval, adapting reusable components.**

I. Introduction

Despite several decades of intensive research, the routine production of software under acceptable conditions of quality and productivity remains an unfulfilled promise. While a great deal of progress has been achieved in understanding the mechanics of constructing a program from a specification, little progress has been achieved in improving the practice of software development accordingly. This predicament stems, in our opinion, from two premises:

- First, a problem of scale: most of our current knowledge in program construction deals with minute details about semantics of programming languages and correctness formulas; while this knowledge is enlightening and instructive, it is rather inadequate to deal with the current pressures on the software industry (in terms of productivity and quality).
- Second, a problem of emphasis: the problem of scale could in principle be tackled with automated tools if it were not for the fact that the most crucial decisions that must be taken in a program construction process, such as the choice of algorithms, control structures, and data structures, are also the most difficult to formalize—hence to automate.

As a result, a wide gap exists nowadays between the demands placed on the software industry (by a society that is increasingly dependent on software and increasingly intolerant of software failure) and what the state of the practice in the industry can deliver; also the brief history of the field abounds with instances of failure [19], [38], [67].

Software reuse offers a great deal of potential in terms of software productivity and software quality, because it tackles the above issues adequately: By dealing with software products at the component level and by focusing on arbitrarily abstract descriptions of software components, it addresses the question of scale; on the other hand, by dealing with software design at the architectural level, rather than the coding level, it addresses the question of emphasis. However, several factors hinder reuse, including the infancy of software development as a scientific [44] or engineering discipline [144], inadequate training in software development in general and software reuse in particular [159], inadequate management structures and practices [59], and the lack of methodologies and tools to support software reuse or software development in general [47]. In this paper, we discuss the most important of these issues and focus on the methodological and technical aspects.

It is customary to categorize software reuse work based on what is being reused (the *object of reuse*) or on the method of reuse (see, e.g., [83] and [68]), the two being closely related. It is customary to distinguish between two general categories of reuse approaches, the *building blocks* approach, which is based on reusing software development *products*, and the *generative* or *reusable processor* approach, which is based on reusing the *process* of previous software development efforts, often embodied in computer tools (processor) that automate part of the development life cycle [68]; these are but two extreme approaches on a continuum involving different mixes of product and process reuse [148]. We refer to both products and processes as *reusable assets*. Reuse approaches raise a number of issues that may be divided into issues related to developing *reusable assets* and issues related to *developing* with *reusable assets*. Under the former set of issues, we focus on

Manuscript received April 1992; revised August 1993.

H. Mili is with the Département d'Informatique, université du Québec à Montréal, Boite Postale 8888, Succ "A", Montréal, Québec, H3C 3P8 Canada.

F. Mili is with the School of Engineering and Computer Science, Oakland University, Rochester, MI 48309-4401.

A. Mili is with the Department of Computer Science, University of Ottawa, Ottawa, Ontario K1N 6N5 Canada.

IEEECS Log Number S95009.

OO software development, as an enabling technology for developing reusable building blocks, and application generators as an example of a commercially successful application of the generative approach. Developing with reusable assets raises issues related to providing methodological and computer support for:

1) locating reusable assets,
2) assessing their relevance to the current needs and
3) adapting them to those needs.

Such issues are anywhere from secondary to irrelevant to the reusable processor end of the spectrum, but are central to the building blocks end of the spectrum. Under adaptation, we discuss a number of techniques for automating the integration and maintenance of reusable components, with an emphasis on techniques other than those offered by object orientation, which are discussed separately, along with other OO principles.

In the next section, we attempt to motivate and define software reuse, and provide a typology of software reuse research, to be used throughout the paper. In Section III, we discuss the overall impact of software reuse on the production of software, starting with the organizational and methodological impact of reuse on the development of software, and then discuss cost/benefit models of software reuse. Sections IV and V focus on the technical challenges and research solutions involved in building reusable software assets and building *with* reusable software assets, respectively. We conclude in Section VI by outlining areas and issues that, in our view, deserve further attention in the research community.

II. A Framework for Software Reuse

A. Motivations

Software productivity has been steadily rising for the past 30 years [160]. However, even with the steady rise in the number of computer professionals [22], it has not kept up with the rising demand for developing new ever more complex software systems and for maintaining existing software [22], [103]. While current software production management practices leave room for improvement [15], nothing short of an order of magnitude increase in programmer productivity will extricate the software industry from the current crisis [67]. According to Boehm, the only factor that can yield that kind of productivity leverage is the number of software source instructions that have to be developed to deliver a given functionality [22]: Instead of searching for ways of writing code faster, we have to look for ways of writing less of it. Automatic programming, whereby a computer system is capable of producing executable code based on informal, incomplete, and incoherent user requirements, is decades away, if ever possible [136]. That leaves us with software reuse as the only realistic, technically feasible solution: We could reuse the processes and products of previous development efforts in order to develop new applications.

Intuitively, savings occur with software product reuse because reused components do not have to be built from scratch. Further, overall product quality improves if quality components are reused. With software process reuse, productivity increases to the extent that the reused processes are automated, and quality improves to the extent that quality-enhancing processes are systematized. Further, there is plenty of duplication in the applications being developed and maintained nowadays, and hence plenty of room for reuse. In 1984, for example, the U.S. software market offered some 500 accounting programs, 300 payroll programs, 150 communication programs, 125 word-processing packages, etc. [77]; the figures are probably higher today. In the early eighties, Lanergan and Grasso estimated that 60% of business applications can be standardized and reused [85]. Generally, potential (estimated) and actual reuse rates range from 15% to 85% (see, e.g., [59], [103]). Existing experience reports suggest that indeed good—sometimes impressive—reuse rates, productivity and quality increases can be achieved (see, e.g., [12], [13], [73], [100]). However, successes have not been systematic (see, e.g., [59, [133]), and a lot of work remains to be done both in terms of "institutionalizing" reuse practice in organizations and in terms of addressing the myriad of technical challenges that make reuse difficult [83].

B. The Object of Reuse

The idea of formal software reuse, as first introduced by McIlroy in his 1968 seminal paper [104], entailed the development of an industry of reusable source-code software components and the industrialization of the production of application software from off-the-shelf components. Software reuse is now understood to encompass all the resources used and produced during the development of software (see, e.g., [43], [50], [133]). Different researchers proposed different categorizations of reusable knowledge, but by and large, most classifications rely on one of three factors or a combination thereof:

1) stage of development at which the knowledge is produced and/or used,
2) level of abstraction (e.g., abstract versus concrete/implemented) and
3) nature of knowledge (e.g., artifacts versus skills).

Jones identified four types of reusable artifacts [77]:

1) *data reuse*, involving a standardization of data formats,
2) *architectures reuse*, which consists of standardizing a set of design and programming conventions dealing with the logical organization of software,
3) (detailed) *design reuse*, for some common business applications and
4) *program reuse*, which deals with reusing executable code. In addition to product/artifact reuse, Horowitz considered various kinds of reuse based on the utilization of very high-level program-producing systems [68].

Three general classes of systems that have been commonly recognized by researchers are:

1) reusable program patterns [19], [68], whereby code or design patterns are used to instantiate specific code fragments or designs, as in application generators or the Programmer Apprentice's clichés [137],

2) *reusable processors* [68], which are interpreters for executable high-level specifications and
3) *reusable transformation systems* [19], [68], whereby some development activities have been embodied in more or less formal transformations (see, e.g., [17], [118], [123]).

Krueger proposed a multilevel categorization of reusable information based on levels of abstraction, where reusable items of level i are, by and large, abstractions of reusable items of level $i-1$, and thus managed to account, more or less easily, for reuse approaches as diverse as source code-scavenging and very high-level languages within the same "abstraction hierarchy" [83].

The above categorizations account for application domain knowledge only to the extent that such a knowledge is embedded in the artifacts (e.g., code and designs) or processes (e.g., application generators). Freeman proposed a five-level hierarchy of reusable *software development knowledge* in which domain knowledge is represented explicitly:

1) environmental knowledge,
2) external knowledge,
3) functional architectures,
4) logical structures and
5) code fragments [50].

The classification corresponds somewhat to the software life cycle, where the last three levels map to the products of system design, detailed design, and coding. The first two (environmental and external) are typically used to derive a particular system's specifications from the user requirements. Freeman distinguished between user- (consumer-) related domain-dependent requirements and developer- (supplier-) related, technology-dependent requirements and characterized the needed knowledge as shown in Table I.

TABLE I
A BREAKDOWN OF ENVIRONMENTAL AND EXTERNAL SOFTWARE DEVELOPMENT KNOWLEDGE [50]

Level	Supplier-Related Knowledge	Customer-Related Knowledge
Environmental	**Technology transfer knowledge**: consists of knowledge about such things as the organizational impact of software technology, personnel training, computer literacy, and so forth.	**Utilization knowledge**: describes the business context in which the software product will be used.
External	**Development knowledge**: deals with the planning and management of software projects such as cost and schedule estimation, test plans, benchmarking, and others.	**Application-area knowledge**: deals with the underlying models for the application domain.

It is the environmental level of software development knowledge that is explicitly lacking from similar life cycle-based categorizations of reusable information. In his 1987 paper, Freeman identified the reuse of environmental knowledge as one of the long-term research goals in software reuse [50]. We know of no research effort that has attempted or is attempting to formalize the reuse of such knowledge since. One area that has been getting considerable attention recently, however, is the reuse of application domain knowledge under the form of *domain models* (see, e.g., [5], [50], [91], [132], [148]). Domain models serve three major purposes:

1) helping developers *understand* an application domain,
2) serving as the starting point for systems analysis (e.g., by specializing the domain model) and
3) providing an application-dependent categorization/classification of existing reusable components (of later development stages) so that opportunities for reuse can be identified as early in the development process as possible [5], [130], [132].

Domain models should identify:

- the entities and operations on those entities that are common to the application domain,
- relationships and constraints between the entities and
- "retrieval cues," i.e., properties of objects that are likely to be used by developers in the process of searching for reusable components [5], [132].

We know of few research efforts that include *declarative* domain models that support all three functions described above (see, e.g., [5], [91]). Neighbors's DRACO system [121] and Simos's work on ASLs [148] achieve much of the same goals by developing domain-dependent specification languages that embody an application domain's common objects and operations

In the next section, we propose our own ontological categorization of reusable knowledge. Our categorization is geared toward highlighting the *paradigmatic* differences between the various reuse methods and abstracting what we consider to be inessential differences between various reusable assets (e.g., code reuse versus design reuse).

C. The Method of Reuse

We adopt the transformational systems' view of software development as a sequence of transformations and/or translations of the description of the desired system from one language (level i description) to another (level $i + 1$ description) as shown in Fig. 1. Three levels of knowledge are used in this translation:

1) knowledge about the source domain (level i),
2) knowledge about the target domain (level $i + 1$), and
3) knowledge about how objects (entities, relations, structures) from the source domain map to objects in the target domain.

For a given level, the knowledge can be seen in linguistic terms, as consisting of a domain language, and a set of expressions known to be valid. The domain language consists of

domain entities (or classes) and domain structures. The description of the various entities and structures can be based on an enumeration of legal entities and structures, or based on a set of properties that must be satisfied by either (e.g., consistency checks, composition rules), or a mix of the two. We refer to the description methods as enumerated and compositional, respectively. The descriptions of past problem instances constitute the expressions that are known to be valid.

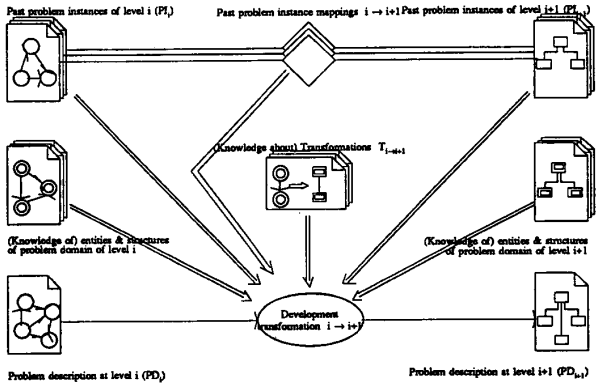

Figure 1. A categorization of reusable knowledge

The mapping knowledge consists of a set of transformation rules, from level i to level $i+1$, and a set of known mappings between problem instances of level i and problem instances of level $i+1$. The transformation rules[1] embody what is usually referred to as process reuse or *skill reuse* (see, e.g., [133]). We shall refer to them as the transformation grammar. Note that this formalism does not distinguish between declarative knowledge and procedural knowledge as we feel the distinction to be mainly a representation issue.

Typically, development consists of, first, describing (specifying) the problem at hand in the language of level i to obtain a description PD_i and, second, transforming that description into one at level $i+1$ (PD_{i+1}), supposed to be the target description language (e.g., executable code). With reuse, one would want to avoid having to *manually*:

1) specify *completely* the problem at hand and/or
2) transform the entire specification of level i into level $i+1$.

Thus, reusable assets include all the kinds of knowledge involved in the development transformation ($DT_{i \rightarrow i+1}$), which can be thought of as the result of applying a generic level-independent problem-solving method on the relevant knowledge sources. The various reuse approaches can be categorized based on:

1) the extent to which the language of level i covers the problem domain of level i and
2) the extent to which the mapping knowledge ($T_{i \rightarrow i+1}$) covers all the entities and structures (i.e., all the valid expressions) of the domain of level i.

1. These rules do not only ensure syntactic correctness of the result description of level $i+1$, but also the preservation of some properties and the satisfaction of some "development constraints."

Finer characterizations may be based on the kind of language description used, along the enumerated versus compositional dimension. Table II shows the characteristics of some of the approaches commonly referred to in the literature. As we go down the rows of Table II, we move from what is generally referred to as the *building blocks* approach to increasingly automated *generative* approaches. Automation requires the complete "cover" of the source domain language (level i) *and* the completeness of the mapping knowledge $i \rightarrow i+1$. In other words, automation is possible if we can express all new problems in terms of problems, or combinations of problems, that have already been solved. We comment below on the various approaches separately.

With source code components, a new problem is solved by composing solutions to subproblems. A complete cover of level i domain would mean that all the components that one may need have been developed, or, more astutely—but equally unrealistic—a set of components has been developed such that every problem can be reduced to subproblems that these components can solve. Notwithstanding the issue of finding such a decomposition/reduction, which can be as challenging as solving the original problem analytically from scratch (see Section V.B), the number of required components is most probably prohibitive [83]. That number depends on:

1) the breadth of the application domain and
2) the composition technique used.

With source code components, composition often takes place "too late" in the software life cycle,[2] limiting the range of behaviors that can be obtained from a set of components to variations on functional composition, as supported by traditional module interconnection languages (see, e.g., [129]) or programming languages. Source code components approaches that support composition of components at a higher level of abstraction yield a greater range of behaviors (see, e.g., [78], [149]). Software schemas are similar to source code components, except that the reusable artifacts are defined at a higher level of abstraction, allowing for a greater range of instantiations (through partial generation) and compositions. Further, the added parameterization makes it possible to build complex, yet generally useful structures (see, e.g., [16]). However, the artifacts are still not meant to cover all the needs of the application domain, and finding and expressing the right compositions are still challenging design problems.

With the remaining three approaches, the source domain language covers the application domain. Transformational systems fall short of automation because the mapping knowledge is incomplete or non-deterministic: A transformational system needs developer assistance in selecting among applicable—and perhaps *objectively* equivalent—transformations [123]. The transformational approach can be used in conjunction with source code components to assist in the modification and integration of such components in new applications [113]. Full

2. Booch's C++ components include 18 implementations of dequeues corresponding to all the possible combinations of choices of
- the concurrency control algorithm,
- the memory allocation algorithm, and
- the ordering algorithm [149].

TABLE II
A CATEGORIZATION OF COMMON REUSE APPROACHES

	Language Level I			Mapping Knowledge			
Approach	**Life Cycle Stage**	**Covering**	**Description Type**	**Covering**	**Description Type**	**Spectrum**	**Examples**
Source code components (see [50], [83])	mostly design	partial	compositional (mostly)	partial	compositional (mostly)	wide spectrum	RSL [27], REBOOT [116], and a number of other "nameless" tools and approaches (e.g. [85], [131], [161]). Object-orientation, seen as a development methodology for reusable components, is discussed in §IV.C. Problems related to the use of such components are discussed in various subsections §V.
Software schemas (see e.g. [83], or referred to as *reusable program patterns* in [77])	mostly design	partial	compositional (mostly)	partial (mostly)	compositional (mostly)	wide spectrum	The programmer's apprentice [137], the PARIS system [80], and Basset's *frame-based software engineering*, in which an application could be *completely* specified and generated using frames [16]. Software schemas are briefly discussed in the context of OO technology §IV.C).
Reusable transformation systems (see e.g. [19], [50])	Software specifications	complete	compositional	partial	compositional	wide-spectrum	A somewhat outdated survey of transformational systems is given in [123]; their potential for quality-preserving maintenance and reuse has been recognized by a number of researchers, including Feather [45], Arango et al. [4], and Baxter [17]. They are discussed in more detail in §V.C.
Application generators (see e.g. [83])	User requirements complete	complete	enumerated (mostly)	complete	narrow, domain-specific	narrow, domain-specific	Unix's Yacc, a number of commercial tools in business information processing (see e.g.[69] for a survey), a number of user interface building frameworks (see e.g. [119] for a survey), etc. Discussed in more detail in §V.B.
Very high-level languages [83], reusable processor [77], etc	Software specifications	complete	*Emphatically* compositional	complete	compositional	depends on the system	Simos' ASL are application-specific languages [148], PAISLey [162] SETL [82] and others are based on application-independent mathematical and computational abstractions. T}.TE.LP.ls 2.ce Table 2. A categorization of common reuse approaches..LP.sp.PP

automation is achieved with application generators and very high-level languages. With very high-level languages, automation is possible at the cost of code efficiency and design quality; very high-level languages are not intended to implement production quality software. Automation is possible with application generators because of a restriction of the application domain.[3] The restriction has the added advantage of making it practical to enumerate a set of template software specifications (or the corresponding software "solutions") parameterized directly with user requirements

It is fair to say that as we go down Table II, the focus shifts from components to composition, and the language for expressing compositions moves up in terms of abstraction. This corresponds closely to Simos's "reuse life cycle," which prescribes an evolution of reuse approaches within organizations, following the maturing of both the application domain *and* the expertise of developers within that domain [148].

3. No application generator available today can build a corporate information system. However, big chunks of such systems (e.g., report generators) can be generated using application generators [34].

The next section deals with the non-technical effects of software reuse on the production of software, including,

1) its effects on the organizational structure of software producing organizations and on the software life cycle,
2) measuring reuse effectiveness, both in technical and economic terms and
3) some reported case studies.

Section IV deals with issues related to building reusable knowledge, with a focus on source code components and application generators. Section V deals with issues related to building new applications *with* reusable knowledge. Such issues are, for the most part, trivial or irrelevant to the application generators and very high-level languages approaches. The discussion will thus be geared toward the building blocks end of the spectrum, and we address issues related to component retrieval, composition, and adaptation. Transformational systems will be discussed to the extent that they help adapt reusable components in a time-saving, quality-preserving way.

III. Software Reuse and the Production of Software

Software reuse provides some feasible remedies to the current software crisis, but many questioned the suitability of existing management practices, organizational structures, and technologies to support software reuse. There is a general agreement that a rethinking of software manufacturing is needed. There is also agreement that the required changes are managerial, cultural, and technical in nature, as was the case for other engineering disciplines [15], [39], [144]. There is no consensus, however, as to the nature and scope of changes, both because the changes involve some yet to be proven management techniques and structures and because the proposed technological answers are different. In this section, we first discuss the effect of software reuse on the organization of software development processes (Section III.A); these changes depend on the reuse paradigm used, along the building blocks versus generative spectrum. Next (Section III.B), we discuss ways to measure software reuse and its impact on productivity and quality. We conclude in Section III.C by discussing the relation between the qualitative effects and measurable effects of software reuse and the challenges that stand in the way of comparing the effectiveness of the various reuse approaches.

A. Software Reuse and Software Engineering

It is fair to say that technological innovations in software development contributed to enhancing software reusability, starting with high-level programming languages, up to structured and modular programming, up to design and analysis notations and methodologies. The same cannot be said about the organization and management of software organizations, which are at best reuse-neutral when they do not hinder reuse practice. We organize our discussion of the changes required and implied by reuse practice into,

1) new organizational structures (e.g., staffing structure),
2) new process models (life cycles) and
3) punctual methodological changes.

A.1. New Organizational Structures

Software reuse relies on the availability of a base of reusable software in all forms (Section IV). Wegner argues that software companies should treat software as capital goods and their organization, including team structures and cost imputations, should reflect that [155]. This is true whether we are dealing with the building blocks approach or with the generative approach: in both cases we have to divert resources, both human and financial, into building a common base of reusable software assets to be amortized over several uses, be they application generators or source code components. It is widely accepted that, in addition to the typical project team structure of software organizations, a team responsible for building and maintaining a base of reuse capital is needed. Different authors proposed different divisions of labor between project teams and "reuse capital" teams. Within the building blocks approach, the component library team would, minimally, be responsible for packaging (e.g., documenting) and controlling the quality of what gets added to the reuse base [131]. The library team could also play an active role in *creating* reusable software of all forms. Barnes studied the economic models for two such arrangements [10]:

1) a pure producer-consumer relationship between the library team and project teams, where the library team is solely responsible for producing reusable components, and
2) a shared arrangement where project teams contribute to and consume what is in the library.

Caldieri and Basili [28] proposed a more software factory-like approach [40]. In their model, project teams do no programming (see Fig. 2). They are responsible for requirements and design specifications—which they submit to the *experience factory*—and for integration and integration testing [28]. The experience factory's activities can be divided into:

1) *Synchronous activities,* which are activities initiated following requests from project teams, and can range from a simple look-up to building the required components from scratch. Such activities are subjected to project teams' schedules.
2) *Asynchronous activities*, consisting of creating components that are likely to be requested (anticipating future demands), or reengineering components generated by the synchronous activities to enhance their reusability.

In [12], Basili et al. report on experiences at the Software Engineering Laboratory (SEL), funded and operated by the University of Maryland, NASA, and the Computer Sciences Corp., in which the above structure has evolved over the years.

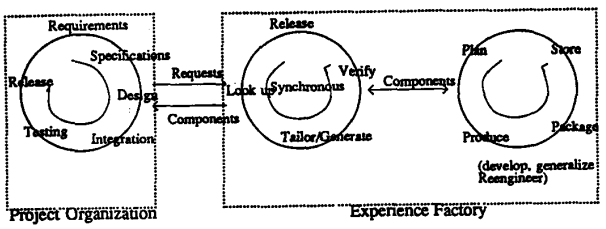

Figure 2. Reuse framework and organization. From [28].

The experience factory was responsible mainly for *process* (vs. *product*) development and reuse [12], [13]. Over a period of five years, reuse rates increased from 26% to 96%, the cost per delivered statement decreased by 58%, a 138% increase in productivity[4] —and the number of errors decreased by a factor of four [12]. It is not clear how a pure producer-consumer relationship between the experience factory and the project teams would have worked. With the building blocks approach, there are a number of motivational and managerial challenges to putting into place such a structure, including putting the

4. The experiments reported in [12] used a project implemented in Fortran as a baseline for errors and cost. Subsequent projects were implemented in Ada. The numbers mentioned here used the first year data with Ada projects as a baseline, instead of the Fortran project. When the Fortran project is used as a baseline, we obtain smaller decrease of cost of delivered statement (35% instead of 58%), but a greater decrease in error rates (a factor of eight, instead of a factor of four), which is to be expected.

most qualified developers[5] in the least satisfying tasks (experience factory) [28] and redistributing responsibility and control of individual projects in a way that may diffuse accountability. Such a division of labor is less problematic with the application generators approach where the skills required from *application developers* are markedly different from those required of *application generator developers*: The former have to be versed in the application domain, possibly end users, while the latter have to be both domain experts *and* software development experts [89]. Less revolutionary divisions of labor exist in more traditional organizations where the job of system administrators and support staff often evolves into building and supporting custom-tailored development tools—embodying reusable processes—or at Japanese software factories, based essentially on a tighter management and measurement of software activities and products (see, e.g., [99], [100]).

A.2. New Process Models

A software life cycle is a model for organizing, planning, and controlling the activities associated with software development and maintenance [124]. For the most part, a life cycle identifies development tasks and identifies and standardizes intermediary work products (deliverables) and review and evaluation criteria. The known life cycles may be classified based on the kind of development tasks and work products involved, and the organization of such tasks. For example, the waterfall life cycle, the spiral model [21], and to some extent prototyping, all involve some measure of analysis, design, coding, and testing. However, while the waterfall life cycle implies that an entire system is analyzed before any part of it is designed or implemented, both the spiral model and prototyping prescribe the analysis $\longleftrightarrow$ testing cycle on system increments [1]. Newer development paradigms usually shorten the analysis $\longleftrightarrow$ testing cycle by automating one or more steps along the cycle [2]. When we talk about software reuse, there are two life cycles to consider

1) the life cycle for developing reusable assets and
2) the life cycle for developing *with* reusable assets.

Issues to consider include whether the two life cycles are different and whether the availability of a base of reusable assets modifies the underlying life cycle. This depends on both the reuse approach used—along the building blocks $\longleftrightarrow$ generative dimension—and on the development methodology used.

With the building blocks approach, both the reusable assets and the products developed with them are software components. Reusable components may be developed either concurrently or separately from specific product development, corresponding closely to the synchronous and asynchronous activities, respectively, of the experience factory in Caldieri and Basili's model (see above). When they are developed concurrently, they follow the same life cycle as non reusable components, except that greater care may go into building and packaging them. When a separate activity is set aside for developing them, we talk about *domain analysis* and *domain engineering*. One of the major inputs of domain analysis is a set of already developed systems within the domain, whose common features are identified, abstracted, implemented, and then packaged [132]. The identification and abstraction of common features can take place at the earliest development stage for which there is adequate documentation. For example, if good quality analysis documents for the existing systems are available, the common features can be recognized at the analysis level. If not, one needs to look at existing designs or even code fragments, perform some measure of reverse-engineering to recover requirements of individual systems/components, identify common features, abstract them, and forward-engineer them

Building new applications with reusable components need not follow a radically different life cycle from building new applications without reusable components (see, e.g., [6], [27], [76], [131]). One of the criticisms leveled at the waterfall life cycle is that each life cycle stage is mainly influenced by the previous stages (top-down), while the existence of reusable components requires some sort of a look-ahead procedure to identify opportunities for reuse and take advantage of them [66], [148]. We believe this to be mainly a documentation issue: Reuse has traditionally meant reuse of small code fragments that have little or no life cycle documentation; if analysis information were stored in components libraries, for example, analysts could identify opportunities for reuse at the analysis level without looking at the actual code of reusable components. The point has been made, though, that OO software development, the reuse methodology par excellence, requires a mix of top-down and bottom-up approaches[6] [66]. This is explained by the premise that an OO development life cycle needs to combine application and domain engineering in order to attain reuse objectives [66]. The application engineering part of the life cycle proceeds in a top-down fashion from requirements gathering to high-level system design. Domain engineering consists of building "clusters" (libraries or layers [107]) of classes, starting with the lowest level (building blocks) which would most likely be needed no matter what the final system design is like, and moving up to application-specific classes, looping back on system design or even analysis [66]. Other OO-induced life cycle changes have been proposed in the literature that are motivated by considerations other than reuse, such as managing the risks inherent in switching to a new development technology [125].

The situation is markedly different with the generative approach. Application generators, which experienced some commercial success, have been studied in the literature (see, e.g., [89], [98]). However, by and large, the process of build-

5. In the SEL at the Univ. of Maryland, the experience factory was mainly staffed by researchers [12]. However, they spent their time mainly collecting and analyzing data and proposing process enhancements, rather than developing variants of sort algorithms or stack structures!

6. The terms top-down and bottom-up are used in software engineering to mean two things. They may refer to the direction used to go through the stages of a life cycle. For the waterfall life cycle, top-down means going from requirements to testing/integration. The terms are also used to refer to construction paradigms, the analytical (top-down) versus synthetic (bottom-up). Things get confused because synthesizing systems from components suggests that detailed design is done before system design, e.g., and the two meanings overlap.

ing "reusable processors" has not earned a lot of attention, perhaps because most of the executable specification languages are research prototypes. A notable exception is Simos's work on application-specific specification languages (ASL), and what he calls *domain life cycle* [148]. He sees ASLs as the culmination of the maturation of an application domain, or of an organization's expertise in that domain. The maturation starts with small reusable code components and moves toward more abstract representations and more complex constructs until an entire application domain is "covered" (Section II.C) [148]. The effect of using the generative approach for software development is much easier to assess: The generative approach shortens traditional life cycles through automation (see, e.g., [98]). Application generators, for example, obviate the need for specifying the software requirements, designing, coding, and testing, of big chunks of applications [89]. Executable specification languages and transformational systems obviate the need for designing, coding, and testing, but developers still need to produce precise formal specifications of the desired system [1].

A.3. Effects on Development Methods

Software development can be considered as a problem-solving activity, the problem being that of finding a software implementation that satisfies a set of user requirements. Cognitive scientists and AI theorists alike consider recall as an essential part of human problem-solving (see, e.g., [88], [141]). Broadly speaking, when faced with a problem, we first perform a "rote recall" to see if we haven't solved the problem before [88]. When that fails, we start looking for *analogical* (similar) problems that we might have already solved and adapt their solution to the problem at hand [29]. When that fails, we fall back on general analytic problem-solving knowledge and skills [88]. Traditional development methodologies (e.g., SA/SD [124]) are analytical in nature and fall back immediately on general problem-solving knowledge and heuristics such as divide and conquer and successive refinements. Researchers recognize that "informal reuse" (i.e., in developer's head) has always been taking place, whereby the base of reusable knowledge is "acquired" individually by developers through experience (see, e.g., [11]). To some extent, "formal" software reuse in general and the building blocks approach in particular recognize the earlier recall-based phases of problem-solving and aim at formalizing them and providing computer support for them.

Challenges to supporting reuse within development methodologies include:

1) identifying reuse tasks and the skills required to perform those tasks (see, e.g., [95]),
2) providing methodological and tool support for these tasks (see, e.g., [113], [116]), and
3) integrating reuse activities into the normal workflow of developers (see, e.g., [47] and [113]).

The reuse tasks depend heavily on the reuse approach used along the building blocks vs. generative axis. With the generative approach, the reuse tasks consist of specifying the desired application in a high-level language (executable specification language, 4GL, etc.), and the required cognitive skills need not be different from those required of traditional development methods. With the building blocks approach, developers try to build a system that satisfies a set of requirements by using as many existing components (or developing as little code) as possible. For any part of the target system, developers must (see, e.g., [47], [95], [113], [135]):

1) formulate the requirements of the part in a way that supports retrieval of potentially useful reusable components,
2) understand the retrieved components and
3) if the retrieved components are sufficiently "close" to the needs at hand and are of sufficient quality, then adapt them.

If no component is found that matches perfectly or closely the given requirements, developers may fall back on general-purpose analytical heuristics to decompose the system (or part thereof) into smaller parts for which steps 1 to 3 may be reiterated [135].

The search and retrieval problem benefits from a large body of work in the area of document retrieval and will be discussed in more detail in Section V.A. For the time being, we note that in the context of reuse, we need more than an algorithm that tries to match a requirement to a *single* component; we need a retrieval system that is capable of *synthesizing* a set of building blocks into a single component that satisfies the developer's requirement. This is what is commonly referred to as bottom-up development. This is perhaps the most challenging problem in the building blocks approach, and where computer assistance is much needed. Yet, there have been few concrete proposals (see, e.g., [62], [79]). Component synthesis and aggregation is discussed in more detail in Section V.B.

Component/program understanding represents an important part of both the mental effort and the cost factor, in reuse [47], [95] and maintenance[7] [103]. Component understanding can mean three things:

1) understanding what it does,
2) understanding how it does it and
3) understanding how to modify it in such a way that it does something a little different.

In a reuse—and maintenance—context, some abstract (implementation independent) component documentation should accommodate step 1, obviating the need for reusers to browse through actual code—i.e., obviating the need for step 2. For components whose evolution/adaptation and extension has been properly planned, the amount of knowledge needed for step 3 can be very small, compared to what it would take to explain how the component works, i.e., step 2; the knowledge required for step 3 corresponds to documenting what Krueger called the *variable part* of component abstractions [83]. It is reasonable to assume that if a component is to be modified in an unanticipated (or not properly parameterized) fashion, one might need to delve into the minute details of the component, and the knowledge required for step 3 may be comparable to that required for step 2. However, studies have shown that reusers are able to edit and adapt components

7. A 1979 study done at IBM revealed that "maintainers" spend at least 30% of their time trying to understand the code to be modified [103].

with only a sketchy understanding of how they work [95]; whether that is desirable or not is another issue.[8]

Program understanding involves the recognition of high-level abstract patterns amid complex and detailed structures. Studies have shown that experts and novices use different approaches to program understanding[9] suggesting that reusers and maintainers may need training in program understanding or the support of tools that help them understand programs [47], [95], [103].

Component understanding is the first step toward component adaptation. Unplanned component adaptation constitutes a textbook case in *analogical problem-solving* [29]. Analogical problem-solving is used when the mapping from problem to solution cannot be characterized intensionally, but such that some <problem,solution> pairs are known for which elements in the solution space (software artifacts) can be traced to elements in the problem space (requirements). A new problem NP is solved by first matching it to a known problem-solution pair <KP, KS>, and then using the difference between NP and KP to infer the difference between KS and the actual solution of NP (NS). Analogical problem-solving is unsound in the sense that a problem solution NS is not guaranteed to solve the problem NP; if we modify a component using this kind of reasoning, we lose any assurances that the modified version does what it says (verification), and what we want it to do (validation). It is *inherently* unsound because it relies on an incomplete knowledge (partial *extension*) of the problem → solution mapping. This kind of technique is only used in the constrained context of transformational systems (see, e.g., [17] and Section V.C) or for informal software artifacts for which there is not much else that can be done (see, e.g., [94]).

In terms of tool support and integration, there is a fairly wide consensus that tools for reuse tasks should integrate seamlessly into CASE environments (see, e.g., [47], [94], [113], [116]). Typical reuse functionalities such as search, copy, and edit should be available to developers in a modeless fashion, and should not distract them from their normal workflow (see, e.g., [113]). Broadly speaking, reuse-oriented CASE environments should be viewed as problem-solving aids, to be used as extensions of developer's mental workspace, rather than a rigid formalism requiring constant translation back and forth to that mental workspace. This entails, among other things, enabling developers to custom-tailor their development environments and providing them with *proactive* development aids/tools [47]. The former is made possible by offering fine-grained development functionalities which developers may combine and sequence at will [116]. The latter remains a research goal, although some knowledge-based systems made some headway in that direction (see, e.g., [147]).

8. Such reuse does away with the quality incentive, and may cost resources for debugging and testing.

9. It was found that experts classify program segments along functional lines, while novices classify program segments along syntactical/superficial similarities. Also, experts use a mix of a bottom-up phase, collecting enough clues to formulate a hypothesis (a pattern), followed by a top-down predictive/verification phase during which they check whether the remaining clues fit in the pattern, while novices use a straight bottom-up strategy, trying to understand programs one line at a time [95].

B. Measuring Software Reuse

Economic considerations are at the center of any discussion of software reuse. Indeed, the most vaunted advantages of software reuse are:

1) an increase of the productivity of software development, which translates directly into monetary terms and
2) an improvement of the quality of the products, which may mean less corrective maintenance, easier perfective maintenance, greater user satisfaction, and so forth, all of which translate into monetary gains.

There are also different costs associated with software reuse, both capital setup (up-front) costs and proportional costs (cost-per-use). Further, different technical approaches to reuse have different investment and return on investment profiles (see, e.g., [42], [148]). Economic models and software metrics are needed that quantify the costs and benefits of reuse. Only recently have researchers started to tackle this problem (see, e.g., [7], [10], [11]). Such studies will not only help convince management of the advantages of software reuse—in case there are any—but will also guide the choice of the technical approaches, and improve the management of the introduction of reuse work methods within organizations [12], [15].

Traditional software metrics that estimate (predict) or measure (after the fact) effort, size, and the relation between them (productivity) need to be amended to account for software reuse. For example, reusable components that accommodate several uses tend to be bigger in size than a version that accommodates a single use, and more complex (see, e.g., [7], [97]). Further, reuse practice presents managers and developers with choices whose implications have to be measured at the organization, project, and task levels. We recognize three such decisions:

1) the decision to launch an organization-wide software reuse program (a long-term, capital investment-like decision [11], [52], [128]),
2) the decision to develop a reusable asset (a *domain engineering decision* [52]) and
3) the decision to (re)use a reusable asset in an application currently under development (an *application engineering decision* [52]).

In the next three sections, we discuss the work relevant to these decisions. Because of the dependencies between some of the metrics and models, we proceed in reverse order. We conclude in Section III.D by discussing the weaknesses of the existing methods and suggesting areas for research.

B.1. Reuse Instance Costs

A reuse instance means different things whether we are talking about the building blocks approach or the generative approach. In the context of the building blocks approach, a reuse instance is a point in the development where a developer has the option of building a component from scratch, but chooses instead to *try* to reuse a component from the library. With the generative approach, a reuse instance corresponds to an entire project life cycle, or a significant part thereof, as the decision to reuse—in this case, generate—modifies the life

cycle in significant ways (see Section III.A). Models appropriate for the generative technology are needed that estimate (or measure) the cost of generative development (see, e.g., [98], [154]) and that compare it to the cost of more traditional development (see, e.g., [114]). In this section, we focus on studies that dealt with the building blocks approach.

Barnes and Bollinger recognized the existence of two kinds of building blocks reuse, namely, *black box reuse*, whereby the component is integrated in its host environment without modifications, and *white box reuse*, whereby the component is adapted and integrated into its host environment [11]. The average cost of attempting reuse can be formulated as follows:

$$[Search + (1-p) \times Development]$$

where *Search* is the cost of performing a search operation on the database, *Development* is the cost of developing the component from scratch, and p is the probability that the component is found in the database. The reuse option is attractive only if:

$$[Search + (1-p) \times Development] < Development,$$

$$\text{or } Search < p \times Development.$$

To favor reuse, we must have an adequate coverage of the library (large p) and make sure that developers can, quickly, either find the component they need or be fairly confident that it does not exist. Obviously, the more complex the reusable component, the more worthwhile it is for a developer to keep searching.

In the context of white box reuse, the developer must weigh the cost of producing a component from scratch against the cost of attempting to reuse one, possibly after modifying it. The average cost of developing with intent to reuse can be formulated as follows:

$$[Search + (1-p) \times (ApproxSearch + q \times Adaptation + (1-q) \times Development)]$$

where p is the probability that the component is found in the database, q is the probability that a satisfactory approximation of the component can be found, *ApproxSearch* is the cost of performing the approximate search, *Search* is the cost of performing an exact search operation on the database, *Development* is the cost of developing the component from scratch, and *Adaptation* is the cost of adapting the component to its host environment [11]. The reuse option is attractive if:

$$Search + (1-p)\, ApproxSearch + (1-p)\, q\, Adaptation \leq (p + (1-p)\, q)\, Development \quad (1)$$

If we consider that the fact that a satisfactory approximation of the component is found means that *Adaptation* $\leq$ *Development*,[10] then a sufficient (but not necessary) condition for reuse to be attractive is given by:

$$Search + (1-p)\, ApproxSearch \leq p\, Development \quad (2)$$

which means the overall cost of search, whether a satisfactory component is found or not, is less than the savings that actually result from those $(100 \times p)$ % cases where a satisfactory component is found.

This inequality has to be understood in the context of experimental evidence to the effect that the cost of adapting a component for the purpose of software reuse jumps very fast as the portion of code to be modified goes up [23]; e.g., the cost of modifying 20% of the code of a component is estimated at near 90% the cost of developing the component from scratch [23]. Margono and Rhoads argued that adaptation costs depended on whether a component was reused within or across application domains and on whether a component was developed in-house or acquired externally [97]. It is fair to say that, in general, white box reuse is cost-effective if it is restricted to those cases where modifications are very minor or already planned and/or parameterized. That being said, inequality (2) can be used as a baseline for developing component libraries and retrieval systems, where we should replace p above by $p \times$ recall,[11] which represents the probability that a component exists that satisfies the needs *and* that is found by the retrieval system. Putting more components in the library increases its coverage (p), but may increase search time (*Search* and *ApproxSearch*) by returning more irrelevant components that need to be studied by developers. Putting in bigger components (higher development costs) increases also the cost effectiveness of the library. We discuss the marginal costs of adding a component to a library in the next section.

We conclude our discussion by pointing out that a developer who is fairly familiar with the contents of a component library can locate what she/he needs more quickly and knows when not to bother even looking. This has the effect of reducing the cost of individual searches (*Search* and *ApproxSearch*) and their relative frequency, which in case of perfect knowledge about the contents of the library, go down from 1 to p for exact search and from $1-p$ to q for approximate search.

B.2. Building a Reusable Asset

Building a reusable asset represents a more or less major investment, depending on the reuse approach used. With the building blocks approach, building components is a regular, recurring activity, whose implications, positive or negative, are minor. By contrast, building a generator is an extraordinary and *costly* decision, on which the success or failure of a reuse program may depend. For the case of application generators, the biggest challenge is to recognize opportunity: When is a generator appropriate [98]. This depends on both the stability of the application domain and the number of systems that need to be developed and maintained within that domain (see, e.g., [34], [89]). The second question has to do with the extent of application development that should be automated. Levy argued that deciding the coverage of the generator should be

10. A study by Woodfield and Embley suggested that developers would not consider reusing if they estimate the cost of adaptation to be 70% or higher than the cost of developing from scratch [159]. They also found that developers systematically underestimate adaptation effort by about 15%, which means that what they perceive to be 70% may actually be 85%. Thus, all in all, developers are reasonably trustworthy as far as ensuring that they don't adapt reusable components in cases where they should develop from scratch.

11. Simply put, the *recall* of a search on a retrieval system is the probability that a relevant item to the search is retrieved by the system. The recall of a retrieval system is the statistical average over a sample of representative searches/queries.

based on rational economic decisions, namely, on the marginal costs of automating an extra ε % of applications, relative to the marginal benefits expected from automating that extra ε % [89]. In particular, he noted that the 20/80 rule holds, namely, incrementally automating the development of applications gets much harder as we come close to full automation [89]. The cost increments have to be measured against (amortized over) the number of systems to be developed and maintained during the "life expectancy" of the generator, which depends on the stability of the application domain.

With the building blocks approach, the decision to build a reusable component should take into account several cost factors:

1) the initial cost of development,
2) the direct and indirect costs of including the component into a library of reusable components,
3) the cost of integrating and/or adapting the component and
4) the expected usage frequency of the component.

Barnes argued that organizations should consider acquiring reusable components from other vendors, and the decision should be purely economical. As a rule of thumb, build reusable components in-house for local expertise, and purchase reusable parts[12] in external expertise. But how to estimate the cost of developing a reusable component? There is a wide consensus that reusable components cost more to develop than nonreusable components with comparable functionality, but estimates range from 50% more [128] to twice the cost or more [97]. The extra cost could be due to a more demanding requirements identification stage (*domain analysis*), lengthier or more complex code[13] (see, e.g., [7]), or more demanding testing and packaging. Balda and Gustafson explored a COCOMO-like empirical cost model for software projects that accounts for both reusing reusable components and developing reusable components [7]. They argued that reusable components tend to be longer and more complex than their nonreusable counterparts, and that the differences depend on the application domain, but offered no detailed breakdown of the extra costs [7]. Rhoads and Margano tracked software projects in which reuse—mainly *within* project—was a priority and found that 60% of overhead costs for building reusable components were incurred during the detailed design of the components [97]. In their study, reusable components were built as a byproduct of application development, and not in the context of a stand-alone domain engineering activity, for which different cost profiles may hold.

Once a reusable component is built, it needs to be included in a repository/library of reusable components. In addition to the obvious (and negligible) costs associated with storage and degraded time performance, there are a number of insidious retrieval costs that are more significant and harder to measure. For a thorough assessment of the result of adding a reusable component to an existing library, we have to see the effects on the reuse instance cost equation:

$$[Search + (1-p) \times (ApproxSearch + q \times Adaptation + (1-q) \times Development)]$$

In principle, adding a component increases the coverage of the component library and thus increases both probabilities p and q and modifies the averages *Adaptation* and *Development* (depending on the new component size relative to the average component size in the library and the average component size "outside the library"). It will also probably increase the costs *Search* and *ApproxSearch*. For instance, with document retrieval systems, there is a three-way trade-off between *recall*, *precision*,[14] and simplicity of the encoding and search strategies [142]. Increasing the size of the document collection degrades the performance of the retrieval system both in absolute terms (e.g., for the same precision level, the user has more irrelevant items to examine) and in relative terms (e.g., "higher resolution" encoding is required to describe components, and thus more complex queries are required to retrieve them with equal precision[15]). It is widely recognized in the literature that bigger libraries are not necessarily better (see, e.g., [73]). Thus, components should be added only after very careful consideration (see, e.g., [28]) and should be taken out of the library, if they have poor reuse record; in [73], Isoda reports on an experimental reuse program at NTT where components were withdrawn from libraries if they haven't been used in three years. In that same experiment, where the components in the library ranged in size from 50 lines or fewer to several thousand lines, it was found that modules of 50 lines or fewer accounted for 48% of the reuse instances and 6% of the reuse volume, while modules 1,000 lines or larger accounted for only 6% of the reuse instances, but of 56% of the reuse volume [73]. Unfortunately, no statistical distribution of module size is provided in [73], but we would not be surprised if pulling those small components out of the library would have actually increased its effectiveness, whereby the loss of coverage is more than offset by enhanced search performance.

B.3. Setting Up a Reuse Program

The question is not so much whether to set up a reuse program or not, but how. There are a number of intertwined organizational and technological choices to be considered, with different cost/benefit characteristics, and managers must have the tools to evaluate and compare them. In this section, we discuss the most salient choices, and any reuse-specific measurables (or measures) proposed in the literature that are relevant to these choices. As shown in Section III.A, reuse practice benefits from new organizational structures and managerial practices. Accounting for such changes in the cost/benefit analysis would be no different from that in any business process re-engineering effort, and won't be discussed below. We organize our discussion around the steps of a reuse adoption process proposed by Davis [42]:

12. This explains in part why mathematical and statistical packages have gained wide acceptance in the software market: Few companies have an in-house mathematician or statistician.

13. E.g., using conditional compilation (extra code) or more parameterization (more complex) to offer several variants of the same functionality.

14. Precision is the average ratio of retrieved and relevant items out of the retrieved items.

15. The readers can convince themselves of the above using intuitive information-theoretic arguments: to encode *and* distinguish between (*precision*) n items, we need codes of length Log(n). The more items we have, the longer the codes.

Initiate reuse program development: This step includes identifying organizational objectives (e.g., productivity and quality objectives) and reuse opportunities [42]. An organization may be active in different application domains, and the reuse potential in each of these domains must be estimated. It is widely recognized that MIS applications are fairly stable and high reuse rates are possible, while systems software and programming environments, e.g., offer few opportunities for reuse (see, e.g., [85], [100]). However, there is no easy way of finding out how much reuse is possible within an application domain without actually doing it over a period of years or performing some extent of domain analysis.

Define reuse program: This step includes:

1) defining the scope/coverage of the reuse program,
2) establishing "reasonable" reuse targets and
3) identifying alternative reuse adoption strategies.

Scoping the reuse program consists of choosing an application domain, or a subdomain thereof, that offers the most reuse potential, the lowest risks, the fastest returns on investment, etc. Once the scope is identified, organizations must establish reuse objectives that they can attain with reasonable effort, depending on a self-assessment of their managerial and technical processes [42]. Davis proposed a *reuse capability model* which defines reuse objectives in terms of three measures:

1) *reuse proficiency*, which is the ratio of the value of the actual reuse opportunities exploited to the value of potential reuse opportunities,
2) *reuse efficiency*, which measures how much of the reuse opportunities targeted by the organization have actually been exploited and
3) *reuse effectiveness*, which is the ratio of reuse benefits to reuse costs [42].

Note that all three measures assume that a reuse program is already in place. Davis pointed out that these measures are not metrics that organizations must be able to calculate at the outset, but are objectives to be attained once a program has started [42].

As mentioned above, it is difficult to precisely quantify the reuse potential of an application domain, and thus, reuse proficiency is only an indicative measure. There has been some interest in the literature for measures of reuse *efficiency*, although mostly as target reuse rates, i.e., as a target percentage of reused code in new projects (see, e.g., [12], [99], [100]). However, there are a number of problems in measuring reuse rates by comparing code sizes, as reflected by the sometimes surprisingly low productivity increases that resulted from impressive reuse rates (see, e.g., [59]). First, there are difficulties in applying such measures for the generative approaches to reuse, where the generated code does not necessarily correspond to what a developer might write, either in style or in size [34]. Second, as shown earlier, reusable components tend to be larger than their nonreusable counterparts, inflating the percentage of reused code within projects. This is exacerbated in the case of the black-box reuse of modules that offer several functionalities: One cannot separate the needed features from those that are not needed (e.g., with OO components) and count them separately. Third, there are also difficulties with defining what constitutes an instance of reuse (see, e.g., [128]): A reusable component that is imported (used) in several client modules should be counted only once. To alleviate these problems, functional (versus size) metrics, such as *function points*, could be used instead. For each project developed under the reuse program, let fp_{tot} and fp_{new} be the function points of the entire project, and of the *new code* developed *for* the project, respectively; the *functionality reuse rate* may be defined as: $\frac{fp_{tot} - fp_{new}}{fp_{tot}}$. The trouble with such a measure is that a function points count cannot be entirely automated. Further, while function points are additive for coarse-grained modules,[16] they may lose significance when we are dealing with low-level components.

Finally, *reuse effectiveness* can be measured directly—and globally—from observables. A naive approach would consist of measuring productivity levels before and after the introduction of reuse. Productivity can be measured as the time average of the ratio of delivered functionality per expended resources. Because reuse involves both proportional recurring costs and one-time fixed costs, productivity studies must necessarily account for different amortizing schedules and account, implicitly or explicitly, for various product line life expectancies. Most of the work on metrics and economic models for software reuse takes into account the time-varying aspects of productivity and explores different return on investment scenarios (see, e.g., [11], [52], [89], [128]).

Analyze reuse adoption strategies: The identification of reuse objectives (in the previous step) suggests a number of candidate reuse adoption strategies, whose costs and benefits are analyzed at this step. An adoption strategy may be seen as a combination of a technical approach and a deployment strategy (e.g., starting at the project level vs. department level, pace of introduction of the technology, etc.). For example, the building-blocks approach may be suited to a low-investment and low-risk, incremental reuse adoption strategy. It also has some inherent limitations in terms of attainable reuse efficiency and effectiveness. A generative approach, on the other hand, supports a high-risk, high-payoff strategy.

Plan reuse adoption strategy: Based on the comparative analysis of the various adoption strategies, one or a combination of strategies may be chosen. At this stage, a detailed deployment plan is produced. Decisions such as how much of the reusable domain to cover the first year, the pace of acquiring the reusable assets, etc. are made here. Detailed cost models such as those discussed in Section III.B.1 and Section III.B.2 are needed.

Implement and monitor reuse program: Monitoring involves collecting data to support the various metrics.

In summary, setting up a reuse program is a major capital investment decision and has been recognized as such by a number of researchers. The economic models proposed in the

16. I.e., are such that for a given two modules $M1$ and $M2$, $FP(M1 + M2) = FP(M1) + FP(M2)$.

literature address fairly adequately the economics of reuse at the organization level and at the project level, by integrating a set of *elementary* cost variables in an encompassing model (see, e.g., [11], [52], [128]). However, it is often those elementary cost variables, or the observables used to derive them, that are hard to measure or interpret, undermining the forecasting or explanatory abilities of these models.

B.4. Discussion

A lot of progress has been achieved on the analysis of the software reuse processes and the derivation of cost estimation models for these processes, and a great deal more is needed. We feel that future efforts should be concentrated on addressing the following aspects:

In relation to reuse instance costs: We need more precise effort and cost models for adapting reusable software. Existing empirical evidence suggests that small changes—which defeat the quality advantage—require substantial efforts [23]—defeating the productivity advantage. We need a better breakdown of those efforts (e.g., trying to understand the code versus implementing the actual change) to focus technical research on those aspects that are most costly. We also need a better characterization of which adaptation efforts are costly and which are not. For example, changing the type of a parameter of a procedure is probably less costly than changing the outcome of a control sequence. Such knowledge may help us develop better techniques for modularizing and parameterizing reusable components and computer tools to support the adaptation process (see, e.g., [94]). We also need a finer characterization and a better integration of retrieval costs in the cost equation (Sections III.B.1 and III.B.2).

In relation to the cost of building reusable assets: It is widely recognized that reusable components are costlier to develop than their nonreusable counterparts. However, there is no agreement over how much more, and there are very few studies about the distribution of "reusability overhead" (see, e.g., [7]); more are needed. There is already recognition in the literature that the extra cost depends on the domain (see, e.g., [7], [97]). Other factors could include the parameterization range, the implementation technique, and associated adaptation/instantiation techniques, etc.

In relation to project-level and organization-level measures: This is perhaps the area where most work is needed. First, we need more accurate and *practical* measures of reuse rates. As shown above, code reuse rates are difficult to measure accurately and do not reflect either effort or savings. Further, they apply only to code reuse and cannot be used to measure design reuse, e.g. We showed that functional metrics are useful, but impractical. We could ignore reuse rates (a means) altogether and look directly at productivity gains (an end). But then, how much of the productivity gains are due to reuse, how much are due to process improvement? How much are due to enhanced communication between developers because teams get smaller? For example, the greatest productivity gains with 4GL tools occur for those projects that become small enough to handle for a single developer [98]. These are not moot questions because we need precise indicators to help us improve those aspects of the reuse plan that can (or should) be improved.

Until (most of) these concerns are properly addressed, there can be no objective basis for comparing different reuse approaches, especially those that fall on different segments of the building blocks ←→ generative axis. The various approaches discussed in the remainder of this paper will only be compared for the extent to which they address specific issues. Where appropriate, we will guesstimate their likely relative effectiveness, but we will not, and cannot, go any further.

IV. Acquiring Reusable Assets

We saw in Section II.B that all the artifacts, both used and produced, and the processes of past software development activities are reusable. We choose the word "acquire" to encompass purchasing, building, and various degrees of reengineering or otherwise transforming existing assets. We discussed the economics of acquiring reusable assets in Section III.B.2. In this chapter, we deal with the technical aspects of acquiring reusable assets. We first discuss general issues related to the acquisition and packaging of reusable assets, with a focus on building blocks. In Section IV.B, we discuss application generators as an example of a commercially successful application of the generative approach. In Section IV.C, we discuss OO software development, as an enabling technology for developing reusable blocks.

A. Overview

What makes a software component reusable? We see reusability as a combination of two attributes, *(re)usefulness*, which means that the component addresses a common need, or provides an often requested service, and *usability*, which means that the component is of good enough quality and easy enough to understand and use for new software developments. The two are often at odds because the generality of a component (its usefulness) entails abstracting the details specific to its individual uses, which often means that these details have to be somehow put back in to use the component, making it less usable. New abstraction techniques in programming and design enable us to reach new optimums but do not change the basics of the trade-off (see, e.g., [72], [83]). This is part of what makes the development of reusable assets more challenging than that of custom-made components.

Acquiring reusable components involves various mixes of new developments and use of existing assets/raw resources, depending on their usability, reusefulness, and desired level of computer support for *unit reuse* tasks (search, understanding, and adaptation/integration). Approaches that rely on existing resources include:

- providing access to existing "assets," which could be as simple as grouping existing computer files in publicly accessible directories, or providing indexing and search tools, browsers, etc. (see, e.g., [113]),
- re-engineering and preemptive maintenance (enhancing the maintainability and the reuse worth of components),

- reverse engineering (recovering "implicit" development knowledge).

or, more generally, transforming available software knowledge that is otherwise too specific (usable but not reuseful) or too diffuse (reuseful but not usable) into a level of abstraction that makes it (re)useful and usable.

Domain analysis and *engineering* may involve any or a combination of the above approaches to identify the basic entities, relations, and operations of the application domain (see, e.g., [132]). Domain analysis is a relatively new activity, and there is some disagreement as to what it involves, both in terms of process/activities and in terms of outputs/work products. However, most researchers agree that a critical (and notoriously difficult) step in domain analysis is the identification of the boundaries of the domain [5], [101], [130]. Lest we oversimplify, domain analysis follows a process similar to that in developing specific software systems. Namely, it involves requirements, analysis, and the production of domain-wide reusable components [5], [101], [130]. The outputs, however, differ from traditional system development in that reusable components typically include standards and guidelines (i.e., *semantic knowledge*), as well as generic, but concrete components such as domain models (i.e., generic functional architectures), generic design architectures and templates, and even generic code fragments [130]. For the case of DRACO [121] and ASLs [148]), however, the output of domain analysis is a domain-dependent executable specification language that embodies the domain objects and operations on those objects.

One of the limitations of "recycling" existing components is that the quality requirements for reusable artifacts exceed those for custom-developed components, and few of the existing components will qualify to be included in a base of reusable assets or will be worth expending effort on. Thus "recycling" is only cost-effective if it can be automated, fully, or to a large extent [28]. Indexing, searching and browsing tools play an important role by organizing existing software knowledge for the purposes of (as an input to) domain engineering (see, e.g., [113]), but do not provide/generate components that are directly usable. In our own work, we built a set of tools that extract a structured representation of software components suited for a reuse-driven CASE tool from diverse and disconnected sources of documentation [113]. However, the added value provided by such tools remains to be proven in a practical setting [113].

In the remainder of this section, we will focus on methods for building new reusable assets, namely, application generators and OO components. Some of the issues related to indexing, retrieval, and browsing, as they relate to software components, will be discussed in various subsections of Section V.A. The interested reader can consult the literature on reverse-engineering; a good starting point is the Jan. 1990 issue of *IEEE Software*.

B. Building Application Generators

Generally speaking, an application generator may be defined as *a tool or a set of integrated tool, that inputs a set of specifications and generates the code of an application within an implementation language*. What distinguishes application generators from compilers of high-level and very high-level languages or automatic programming systems are the "specifications" or "programs" input by the developer, which are:

1) partial—the tool completes them by a set of domain-dependent reasonable defaults and
2) partially or totally nonprocedural—declarative, graphical, etc. [98].

Martin enumerated a number of mostly behavioral properties that application generators should exhibit, including

1) user-friendliness,
2) usable by nonprofessional programmers,
3) support for fast-prototyping,
4) applications take an order of magnitude less time to develop than with traditional development, etc. [98].

It is next to impossible to give a more precise operational definition of what constitutes an application generator without excluding known classes of application generators. This is due to the fact that the specification language used—and hence the generation technique—depends very heavily on the application domain. For the same reasons, it is difficult to design a development methodology for application generators that is appropriate for all application generators, and the development of application generators *in general* received little attention in the literature; by contrast developing *with* application generators has received a fair amount of attention (see, e.g., [114], [154]). The material presented below is based mostly on the work of Levy [89] and Cleaveland [34], describing work at AT&T Bell Labs.

Viewed as translators, applications generators have a fairly standard architecture (system design). Further, the programming techniques for implementing translators (detailed design) are well-understood and fairly standardized. In fact, the design and implementation of translators are so well-understood and standardized that application generators themselves can be built using application generators [34]! The major difficulties in building generators reside in:

1) recognizing cases when they are appropriate [34], [89],
2) defining their requirements, in terms of defining the input language, the output language, and the "transformation grammar" [34], [89] and
3) validating their outputs, i.e., verifying that the code generated does what it is supposed to do [72].

The first two difficulties are methodological in nature. Defining the input language involves striking the proper balance between a language that is sufficiently abstract to be usable by noncomputer experts, but also concrete enough so that executable code can be efficiently generated. Validating the generated code poses a number of technical—and theoretical—challenges [72].

Levy identified a coarse three-step methodology for developing with application generators (what he calls *metaprogramming* [89]):

1) identifying the requirements of the generator,
2) building the generator and
3) using the generator.

Cleaveland proposed a breakdown of the requirements phase into six subphases briefly summarized below [34]:

Recognizing domains: This step consists of assessing whether an application generator approach is appropriate or not. According to Levy, applications generators are appropriate for applications that embody a "complex synthetic set of rules" [89]; complex in the sense that no notation is known within which they can be described succinctly, and synthetic in the sense that they are man-made. This entails that the rules cannot be had right the first time, and they will keep evolving. This makes it appropriate for prototyping. Or, if we look at the full half of the cup instead, application generators are needed when several similar systems have to be built and maintained. This makes it suitable for stable and well-understood application domains. Cleaveland proposed a number of "appropriateness heuristics" including [34]:

1) recognizing recurring patterns between applications (code, design, architecture),
2) a "natural" or "emergent" separation between the functional (declarative) requirements and the implementation (procedural) of applications, or
3) a fairly systematic procedure to go from one to the other.

Defining domain boundaries: This consists of identifying the parts of applications that will be generated, the parts that will have to be built by hand, and the interfaces between the two [34]. There is a trade-off between the range of applications that can be built with the generator (breadth) and how much of these applications will be automated (depth); the decision should be based on economic considerations [34], [89].

Defining an underlying model: This step consists of defining an abstract computational model for the application domain. It is abstract in the sense that it does not depend on a particular implementation technique. Different computational models are appropriate for different application domains [89]. For example, a computational model appropriate for reactive systems could be finite state machines, while one appropriate for database applications could be relational calculus or algebra. Computational models are important for consistency, understandability, and validation [34]. They also make it easier to systematize the implementation of a generator and the generation of a family of generators.

Defining the variant and the invariant parts: the invariant part of an application family consists of the implementation details of the application and all of the defaults assumed by the generator; the variant part consists of those aspects that the developer has to specify. The variant part includes input specifications as well as *code escapes* [34]. Code escapes are used when a part of the application cannot be captured—concisely or at all—within the computational model; they defeat some of the advantages of generators (maintainability at the specification level, traceability, testability, etc.) and should be avoided whenever technically possible [34] and economically justifiable [89].

Defining the specification input method: The input method is essentially the user(developer)-interface of the generator. Input methods depend on the underlying computational model and the target user (developer) community. Input methods include:

1) textual inputs (expressions),
2) graphical inputs (e.g., for user-interface builders [119]),
3) interactive template-filling, etc. [34].

Defining the products: Generators can generate programs, documentation, tests programs or data, and even input to other generators [34]. Issues such as packaging for readability and/or integration and performance, e.g., are important for code fragments [34].

A major concern with application generators is their testing: checking that they do generate the correct code. One of the ways programs are usually tested is by comparing their actual outputs to expected outputs. With program generators, we are not certain that the expected output is correct: it, itself, has to be tested. This additional level of indirection makes it that much harder to validate generators [72]. The problem is more acute than with traditional high-level language compilers which translate imperative code into imperative code, and where there is an easier correspondence between source code and—nonoptimized—target code.

C. Object Oriented Programming

In the past decade, object oriented programming has come to be considered a panacea to all computing aches. Software engineers view object orientation (OO) as the answer to their numerous and intractable problems: enhancing software quality, reusability, and providing a seamless development methodology (see, e.g., [35], [38], [106]). Database researchers recognize that OO allows modeling the semantic *behavior* of data by encapsulating data with the procedures that manipulate them [25]. In the knowledge-based systems arena, OO reincarnates old ideas such as procedural knowledge representation, inheritance, and distributed control [141]. While researchers may not agree on the specifics of the tenets of object orientation, there is a fairly wide consensus that it is an enabling technology for creating interchangeable and reusable software components. We first provide a brief tutorial on OO. Next, we discuss reusability issues across the OO life cycle, i.e., analysis, design, and programming. This is by no means a survey of OO research; our focus is on those aspects of OO that make reuse inevitable, possible, or difficult.

C.1. Object Orientation 101

The concept of "object" in programming was introduced by Dahl and Nygaard in their language SIMULA [41]. SIMULA was designed as a language for simulating dynamic physical systems. Physical objects were modeled by structures containing state variables and procedures used to manipulate them. Using today's jargon, we would say that objects are compilation units that encapsulate data with the procedures that manipulate them. One of the advantages of such structures, from a programming language point of view, was to separate the

visibility of variables from their lifetime, i.e., a variable could be active outside the scope of its visibility. This is the basic idea behind *information hiding*. Information hiding enables us to build modules that are easier to understand and more reusable. Because of information hiding, "objects" can only be manipulated through *public interfaces*—sometimes called *protocols* or simply *interfaces*—i.e., a set of procedures that are "publicly" visible. This makes it possible to change the implementation details of an object without affecting its clients.

Intuitively, a *class* is (the description of) a collection of objects that share the same data structure and support the same operations. The description of a class includes a data template and a definition of the operations supported by the objects—called *instances*—of the class. In some OO languages and modular languages (e.g., Modula and Ada), a distinction is made between the *specification* of a class (e.g., *package specification* in Ada) and its *implementation* (e.g., *implementation module* in Modula). Typically, the specification of a class corresponds to its public interface.[17] An *abstract class* is a class that has a specification but no implementation. *Overloading* makes it possible for several classes to offer/implement the same operations; the compiler disambiguates operation references using the parameter/operand types. *Polymorphism* makes it possible for a variable to hold objects belonging to different classes. *Dynamic binding* delays the resolution of operation references until run-time when the actual type of the variable is known; this allows for greater flexibility in programming [106]. Overloading and polymorphism make it possible to develop general-purpose client code that is indifferent to the *reimplementation and extension* of server code. Classes can be organized along *hierarchies* supporting different kinds of "inheritance." The parallel with natural taxonomies, whereby a natural category "inherits" a number of properties from its ancestors, is tempting, sometimes useful, and often misleading [72]. For the time being, let us just say that inheritance in programming languages is a built-in code-sharing mechanism that, without polymorphism and dynamic binding, would not be much different from various module import mechanisms in traditional languages.

In addition to its programming significance, OO is also a modeling paradigm. As a computational model, OO represents a significant departure from traditional process-oriented modeling approaches in which there is a clear divide between process and data. In process-oriented approaches, data are viewed as static entities, whose *domain-dependent* dynamic semantics are buried into processes which embody *application-dependent* tasks. Complexity in modeled systems is then reflected in the procedures. By contrast, OO encapsulates data with their *domain-wide* dynamic semantics, and complexity in the modeled systems is reflected in the data instead. Presumably, this makes for partial models (components) that are reusable across various applications within the same domain [106]. Further, because procedures evolve faster than data in domains, OO models tend to be more resilient to change [106].

17. This is the case in the Modula and Ada families of languages. In C++, however, the specification must list the procedures and data variables that are not visible outside, but say so.

The modeling potential of OO found its way into analysis and design (see, e.g., [140], [146], [157]). OO proponents argue that OO models, in addition to their reusability and resilience to change, are easier to understand and to communicate to end-users (see, e.g., [35]). Typically, OO analysis is concerned with the derivation of two views of a system:

1) a *static* or *structural* model, describing the objects of the domain and their relationships and
2) a *dynamic* or *behavioral* model, describing the functional and control aspects of the system as embodied in individual object operations and interobject interactions (see, e.g., partial surveys in [46], [158]).

Objects that have the same properties and exhibit the same behavior are grouped in classes. Class hierarchies start taking shape where classes that share *application-significant* data and *application-meaningful external behavior* are grouped under more *general* classes. Identifying generalizations of classes at this level has several advantages, including:

1) enhancing the understandability of the models by reducing the number of *independent* concepts that an analyst/user has to deal with,
2) providing a cross-check with data dictionaries to enforce consistency within the model and
3) identifying opportunities for code reuse [35], [140], [157].

The last is justified by the intuitive realization that similar requirements in terms of external behaviors—an analysis-level product—generally lead to similar implementations.

Object oriented design binds domain-level classes—a requirement—into computational structures that, in addition to "implementing" the required functionality, maximize code sharing and satisfy environmental and performance constraints (see, e.g., [31], [70], [140]). *System* (architectural) design includes partitioning a system into subsystems and/or layers, choosing an overall control paradigm (e.g., event-driven versus hierarchical), and distributing data and processing (see, e.g., [140]). *Class* (detailed) design includes:

1) representation issues (e.g., of attributes, associations, and collections),
2) algorithms, which are tightly coupled into representation issues, and
3) object control paradigm [31], [140], [146].

There are a number of advantages to keeping design (and implementation) class hierarchies close to analysis-level hierarchies, including:

1) traceability [36],
2) conceptual clarity (see, e.g., [39]),
3) reuse of interfaces (see, e.g., [37]) and
4) potential for reusing *application-meaningful computations*—by contrast to structure-manipulation operations which are *inherently* representation/implementation dependent.

Methodologists recognize that in some cases, environmental considerations may dictate different representations for behaviorally similar classes, leading to either suboptimal code

reuse/sharing or, if we insist, unsavory class hierarchies (lots of cancellations, unsafe inheritance, methods having awkward names, etc. [37]). They also suggest looking into alternatives to inheritance (e.g., *delegation*) that achieve the same goals [140]. In general, the transition from design to implementation is fairly straightforward. For the case of control-intensive (e.g., real-time) applications, the transformation can even be automated (see, e.g., [105]).

C.2. Reusability Issues in Object Oriented Analysis

The proponents of OO attribute a number of qualities to OO analysis and to the resulting models, most of which are supposed to favor reuse. We will discuss these as well as other tenets of OO analysis that may impact reuse positively or negatively.

An often-cited advantage of OO analysis and OO models is what Hoydalsvik and Sindre called *problem orientation* [70], i.e., the models are cast into terms of the problem domain. This makes models easily communicable to the target user community and favors greater user involvement in development and hence greater satisfaction with the final product (see, e.g., [35], [36]). We share the view that this is only true in data-rich, processing-poor application domains where objects are intuitive and easy to identify [3] and where most of the processing consists of associative data access; these are domains where more traditional data modeling techniques are already known to be more appropriate than process-oriented techniques [46]. In control-intensive applications, objects are synthetic (artificial) service providers rather than natural data holders, and most of the complexity is embedded in the dynamic model, which uses the same notations as those used in process-oriented techniques.

A second related advantage of OO models is their—presumed—resilience to evolution. Presumably, in application domains, processes change more often and faster than the entities of the domain, and hence a model structured around the data of the problem domain will be more stable [106]. To this, we add the fact that:

1) information hiding minimizes the impact of data structure changes and
2) hierarchical classification enables us to handle data specialization and extension quite handily.

Lubars et al. set out to test the claim that OO models are stable [90]. They define model stability in terms of three properties:

1) *localization*, i.e., changes should be localized in the model, even if they require considerable rework in a localized area,
2) *conservative extension*, meaning that the effect of a change on the work *already done* should be minimal, i.e., we should, as much as possible, extend existing work but not redo it and
3) *model independence*, in the sense that changes to structural (data) models have little impact on behavioral models, and vice-versa.

The authors modeled the ATM application using Rumbaugh et al.'s object modeling technique (OMT [140]) and considered two "small change" scenarios to assess the stability properties. They observed that the structural model—called object *model* in OMT—was well-behaved, but that the behavioral models[18] were not. They also observed that the models were somewhat interdependent because in one scenario, changes to the behavioral models led to revising the object model [90]. The authors recognized that theirs was not a controlled experiment and that no definitive conclusions could be drawn. We believe that some of the difficulties were specific to OMT, and to data-driven methods in general,[19] but concur with their observation that ease of evolution may conflict with ease of description. The authors mentioned two modeling "tricks" that would have stabilized the models:

1) the use of abstract classes to leave room for future specialization or factoring of existing classes and
2) the use of "mixins"[20] to separate concerns and reuse them independently;

both techniques have no meaning to the end user [90].

A third advantage of OO analysis is that it lends itself naturally to *domain analysis* (versus single application analysis) and thus leads to more widely reusable components. This is attributed as much to the notation as it is to the process. For example, once it has been recognized that the class **CheckingAccount** is part of an application, it is difficult not to think of operations to deposit, withdraw, and give balance, even when the application at hand requires only one or two operations. Further, data-driven methodologies (e.g., OMT [140]) explicitly prescribe that analysts should rely on their knowledge of an application domain to complement the statement of the problem as a source for identifying the relevant objects/classes. However, this approach has been criticized for being open-ended, i.e., analysts do not know when to stop adding objects and classes that may be relevant to the domain, but could be irrelevant to the application at hand, and thus unduly burdening the project at hand (see, e.g., [74], [125]). *Use-case driven* or *scenario-driven* methodologies are supposed to alleviate this problem by focusing on the objects that participate in useful system behavior (see, e.g., [74], [138]). And finally, generalization enables developers to factor out the shared data and behavior between classes in (abstract) classes which are even more reusable than the actual (concrete) classes. We find it surprising that, despite the importance of the analysis-level hierarchical organization of classes on the OO development life cycle, and the long-established research tradition in classification in artificial intelligence, there have been relatively few efforts to provide automated or semi-automated tools for building or maintaining class hierarchies

18. OMT uses several complementary notations to represent behavior, including *event flow diagrams* to represent messages exchanged between objects, and *Harel statecharts* for individual objects to represent objects' individual behaviors.

19. A method is considered to be data-driven if modeling starts by identifying the *objects* of the domain/application, before analyzing any desirable behavior the system has to have.

20. Broadly speaking, *mixins* correspond to "mixing in" the features inherited from two different superclasses (multiple inheritance), each representing a view of the class.

(see, e.g., [18], [37], [55], and [56] for a brief literature survey on classification techniques in OO analysis). Notable weaknesses of existing hierarchical classification methods include the fact that most methods do not take into account behavior [56]. Further, nearly all classification methods are limited to "naive" factorizations in the sense that they assume that every attribute or operation is defined in only one place, ignoring redefinition (extension or specialization) at lower levels; this leads to factorizations that are either oversimplifications (unsound) or suboptimal (incomplete) [56].

Some aspects of OO analysis have also been criticized for *hindering* reuse or for underusing the potential of OO for reusability. One of the thorniest problems resides in the specification of interobject behavior (see, e.g., [3], [46], [158]). An unspoken corollary of OO is that any behavior that an object system may exhibit must be attached to an object/class within the system. This forces us to specify—and implement—the interaction between two objects as an operation on one of the two, i.e., it forces us to assign responsibility for a behavior involving two objects to one of the two objects. If that behavior (or the interaction underlying it) is contextual (specific to) a given application, the object made responsible for the joint behavior is not reusable across applications [109]. This led a number of researchers to propose dynamic entities, other than state-bearing application objects, that embody interobject behavior (e.g., *relationships* [139], *constraints* [109], *contracts* [65], etc.). Recognizing the need for such constructs has not made the identification of "behavioral boundaries" between objects any easier (see, e.g., [3], [109], [158]).

A second set of problems deals with the related issues of view modeling and multiple inheritance. Multiple inheritance occurs when a class has two or more *nonhierarchically related* ancestor classes. Multiple inheritance may occur in natural taxonomies and has been supported by a number of knowledge representation languages (e.g., KL-ONE [24] and its derivatives). Further, automatic hierarchical classification (class factoring) algorithms that avoid redundancy may generate *lattices* rather than trees (see, e.g., [37], [56]). Multiple inheritance, which is a powerful modeling concept, becomes a programming language nightmare when the transition is made to implementation. Further, it often results from integrating different *roles* that objects may play within the same application. Forcing all the roles on the same class definition has a number of disadvantages [63]. First, the models tend to be hard to read and understand. For instance, generalization relationships become hard to understand as a class may descend from two unrelated superclasses, each representing a different role. Further, a unified nomenclature must be found for all the roles, inevitably losing meaning and significance. Second, this leads to suboptimal reuse as the individual roles cannot be reused (extended and/or specialized) independently [63]. The problem of providing different views of objects has been addressed in OO programming languages for some time (see, e.g., [145], and C++'s three visibility/access modes for class features [152]). Views as a modeling concept have been getting more attention recently (see, e.g., [63], [122]).

C.3. Reusability Issues in Object Oriented Design

In OO software development the same basic set of concepts is used to describe the products of analysis, design, and implementation [66]. Presumably, this helps make a seamless transition between analysis and design (see, e.g., [31], [35]). The advantages of such a seamless transition are numerous and have been thoroughly documented[21] (see, e.g., [35] and Section IV.B.1). It has been known for some time that the mere use of an object notation is not sufficient to ensure a seamless transition from analysis to design, and that additional care must be taken to ensure that it is (see, e.g., [31], [81]). For instance, it is widely recognized that OO design involves more than adding detail to analysis-level models, and analysis-level class hierarchies may need to be reorganized to take into account environmental and performance factors (see, e.g., [66], [111], [140]). This may lead to design models where classes that have the same external (application-meaningful) behavior are no longer hierarchically related, leading to suboptimal code reuse [111]. Worse yet, if we insist on maximizing code reuse, we may end up with two hierarchically related classes that do not share application semantics. This may lead to unsavory class hierarchies, with lots of cancellations, unsafe inheritance, awkward method names, unpredictable behavior, et. [37], [81]. Most methodologies recognize the problem, but don't do much about it beyond suggesting aggregation/delegation as an alternative to inheritance for code sharing (see, e.g., [140]).

We believe that concerns for reusability and safety need not be contradictory if we view class design as consisting of two *distinct* and possibly *asynchronous* activities:

1) the development of computational structures that support generic, application-independent *structure* manipulations with given performance characteristics and
2) choosing, for a given application-meaningful analysis-level class, the *structure* that best matches its requirements.

To some extent, the above problems are due to the fact that computational structures are essentially designed one application class at a time and are, in a sense, *prematurely* bound to the semantics of application classes by both data types and names, i.e., before they can be refined and reused. Shlaer and Mellor recognize this problem, and the design phase of their methodology includes three steps:

1) building a system-wide design policy,
2) building mechanisms and code templates to support this policy and
3) populate the templates with analysis-level models [105], [146].

However, while they insist that code templates are highly reusable, they provide no formal mechanisms for their reuse.[22]

21. Hoydalsvik argued that because analysis models should be problem/application-oriented and design models should be solution-oriented, then any methodology that claims to bridge the two is not doing analysis correctly [70].

22. The CASE tools that support Shlaer and Mellor may offer specific functionalities that support editing existing templates.

In [111], we proposed a design framework that makes it possible to:

1) reuse interfaces and application-meaningful behavior between design-level classes that do not share the same representation and
2) reuse representation and structure manipulation code to the fullest, without jeopardizing the interface conformance of two hierarchically related design-level classes.

Our solution to the first challenge relies on:

1) defining a flavor of inheritance restricted to *method* inheritance, excluding memory structure[23] and
2) enforcing a strict discipline in designing/coding application-meaningful logic in a way that does not bind it to the representation.[24]

Our solution to the second challenge relies on:

1) defining reusable design templates, including data structure definition and *manipulation* and
2) developing a mechanical procedure for "instantiating" a design-level class by mapping an analysis-level class over a design template [111].

Our design templates may be seen as generic data structures parameterized by both data component types and data component (field) names. A design template may be *extended* by adding new data components and/or operations. Because data component names are meaningless parameters, developers need to specify parameter mappings in case of ambiguity or multiple inheritance/extension [111].

We have not had reliable practical experience with our methodology to ascertain its effectiveness. For example, it is not clear how much of an application's logic can be coded (in an object-flavored PDL [111]) without referring to an internal representation. Further, from a theoretical point of view, thorny subtyping issues with generic types are made even worse by the name parameterization [111]. It does build, however, on the proven principle that greater reuse can be achieved by delaying binding component specification to component realization/implementation and exemplifies the progressive move from a pure building-blocks approach, to one that includes some generation (see, e.g., [149] and Section II.C).

C.4. Reusability and Object Oriented Programming

Naturally, any impediments to reuse that may appear during analysis and design persist when the components are actually implemented. Implementation further binds components to a particular programming language and style, inevitably reducing their applicability. There is disagreement among methodologists whether language considerations should come into play during design or not (see, e.g., [140] vs. [31]). This is an important question because different languages support different sets of OO features (e.g., typed vs. untyped polymorphism, single versus multiple inheritance), with different reusability characteristics that may tip design trade-offs. For example, if we want to maximize code-sharing in a language that does not support multiple inheritance, it is not enough to "linearize" a lattice that maximizes code-sharing; in order to keep the same class hierarchy at design and implementation, we have to consider language features at the design level [37], [56].

In addition to the loss of reusability due to language power, some of the very basic tenets of OO programming go against reusability. First, encapsulation and information hiding replace the traditional stamp coupling between modules, by common coupling[25] *within* modules [156]: All the methods within a class are common-coupled via the structure of the class. Further, inheritance in OO programming languages violates encapsulation and information hiding [150]. For instance, in most OO languages, a class has access to all the implementation details of its superclasses. Thus, methods can be—and often are—written in such a way that they depend on the implementation details of their superclasses: When those change, the entire class hierarchy beneath them may be affected. The C++ language addresses this problem by providing three access levels for attributes and methods:

1) *private* attributes and methods are accessible only to the methods of the defining class,
2) *protected* attributes and methods are accessible to the methods of the defining class and its subclasses and
3) *public* attributes and methods which are accessible to all objects/methods [152].

These access levels are further modulated through the accessibility/visibility of the subclass relationship itself: A client object A may be prevented from using knowledge that server B is a subclass of C, thereby preventing him from using facilities (e.g., operations) that B inherited from C [152]. Meyer argued that inheritance in class libraries is a mechanism that is useful only to the component builder, but that the component user (client) need not and should not be aware of inheritance relationships [107]. When the programming language does not have the built-in mechanisms to discipline the use of inheritance, programmer discipline ("*always* use access functions to read/write objects' attributes") or language preprocessors are needed [111].

C.5. Current Trends

As object technology matures, the distinction between what it *guarantees*—no matter what—what it *enables*—if additional guidelines are used—and what it cannot *deliver*, becomes clearer. Object oriented *programming* guarantees very little, in and of itself. It is mostly an adequate packaging technology for reusable components. The major obstacles and opportunities for building reusable components remain at the analysis and

23. Theoretically, this flavor is nothing but subtyping. From a programming language point of view, it is a mix of class inheritance and delegation.

24. Roughly speaking, we use a stricter version of the *private* mechanism as used in C++. Our version makes sure that data fields/structures are private to the methods used to access them; no other method, even ones belonging to the same *class*, can access them [111].

25. Two modules are *stamp-coupled* if they interact through a *visible* complex data structure, as when a procedure invokes another one whose parameters include a record. Two modules are *common-coupled* if they interact through a *hidden* complex data structure, as when two procedures access the same "global" complex data structure. Common-coupling is less desirable than stamp-coupling because the dependency between the modules is hidden, making changes to the modules error-prone.

design level. Research efforts in analysis and design shift from notations to processes. There are marked weaknesses in analysis and design heuristics and increasing demands for more formal processes, with verifiable properties (see, e.g., [3], [31], [70]). One of the still open, yet fundamental issues that have a direct bearing on the reusability of components has to do with the identification of object's behavioral boundaries and the elicitation and representation of interobject behavior (see, e.g., [3], [157], [158]). Succinctly put, given a high-level specification of the behavior of an object system (whose component objects may be known or not), how to distribute the behavior among component objects in a way that maximizes a given quality criterion—in this case, reusability. The answer to this and related questions may build on existing work on formal specification techniques for reactive systems [64]. It is also becoming clear that not all global behaviors can be effectively distributed among objects (see, e.g., [70]) and there is increasing interest in multi-paradigm programming, e.g., combining logic and OO programming (see, e.g., [96]).

A second major thrust in OO research was motivated by practical experience, as it quickly became clear that classes and methods are too fine-grained reuse units to provide any substantive leverage and bigger reusable units need to be considered. For instance, objects seldom offer any interesting behavior on their own and it is often in combination (interaction) with other objects that any useful functionality is achieved [75]. The idea of object *frameworks* [43], as design-level collections of interacting and *interchangeable* objects, captured significant attention recently.[26] The idea of reusable software (micro-)architectures is not new. However, object orientation's abstraction, parameterization and code-sharing mechanisms support elegant ways for developing and using *frameworks* (see, e.g., [72]). OO frameworks have been developed and used successfully in the area of graphical user interfaces (see, e.g., [43], [58], [110]). A lot of attention has been devoted recently to the documentation of frameworks, both formally, in terms of specifying interobject interactions (see, e.g., [65]), and informally, to describe their applicability and illustrate their use (see, e.g., discussions about *pattern languages* [51], [75], [87]). It is interesting to note that, according to Krueger's classification of reuse approaches [83], the emphasis on OO frameworks moves the reuse of OO software from a pure component-oriented approach to a combination of "software schemas" and "software architectures" approach, i.e., occurring at a higher level of abstraction and providing much greater reuse leverage [83].

V. Building With Reusable Software

In this section, we discuss issues related to developing *with* reusable software. We focus on issues related to the building-blocks approach because, as mentioned in Section III.A.2 and Section III.A.3, the generative approach does not affect much the steps that it does not automate. We will discuss in turn component retrieval, component composition and component adaptation. In component retrieval, we look at the problem of matching a set of requirements for a component to a database of component descriptions. The matching seeks a *monolithic* component that satisfies the requirements. With component composition, the matching seeks a *combination* of components (such as functional composition) that satisfies the requirements. Finally, we shall discuss component adaptation from the perspective of transformational systems.

A. Software Component Retrieval

Given a set of requirements, the first step in building with reusable components consists of finding a component that satisfies the requirements, either in its present form, or modulo minor modifications. When the number of components in the library is large, developers can no longer afford to examine and inspect each component individually to check its appropriateness. We need an automated method to perform at least a first-cut search that retrieves an initial set of potentially useful components. Such a method would match an encoded description of the requirements against encoded descriptions of the components in the library. The choice of the encoding methods, for both the requirements and the components, and of the matching algorithms involves a number of trade-offs between cost, complexity, and retrieval quality. We start by formulating the retrieval problem from a software reuse perspective. Next, we discuss some of the trade-offs involved in the choices mentioned above. Finally, we briefly describe a representative subset of related work in the literature.

A.1. The Component Retrieval Problem

We can formalize the component retrieval problem as follows. We make the distinction between a problem space and a solution space, where the problem consists of the developer's needs. We can further divide the problem space into:

1) *actual problem space*, as opposed to,
2) problem space as understood by the *developer* and
3) *query space*, which consists of the developer's perceived need's translation into a "query" that the component retrieval system can understand (see Fig. 3).

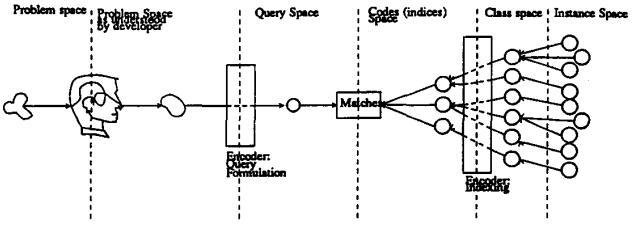

Fig.3. A model of component retrieval.

We use the term *query* to mean an expression of the developer's need. It can be as simple as a string fed into a string search command or as complex as a Z specification of the component. Depending on the level of expertise of the developer, his/her understanding of the need can be as close to the

26. For example, IEEE's *Computer* started a new department called *Frameworks* in the March 1994 issue. An entire conference is devoted to the documentation of frameworks. The seven-year-old Software Frameworks Association is a self-help nonprofit organization. For info, e-mail to info@frameworks.org.

actual need as possible. Also, depending on the expressiveness of the language used to formulate queries, the developer (or some other agent) can translate those needs as accurately as possible.

From the solution space end, we can identify three subspaces:

1) component instances space,
2) component classes space and
3) codes/indices space.

The component instances space consists of components some of which may be equivalent in some respect. For example, two stable in-situ sorting algorithms are *functionally equivalent*. Two quicksort programs coded in *exactly* the same way, but such that variable names differ are *functionally equivalent* and *performance equivalent*—they generate the same machine code. Within the class space, these two components will be represented by the same class. The class space is the space of equivalence classes. The equivalence relationship may include functional equivalence (same input-output relation), performance equivalence (same time and space requirements), etc. The codes space consists of the descriptions of the component classes using an encoding or *indexing* language. In practice, the encoding step inevitably results in a loss of information. In the best case, "indexing" encodes only a subset of the properties of a component class, as when the encoded description of a program module does not state *all* the potential uses of the program. Depending on the relative size of that subset, indexing would project more or fewer distinct component classes into the same codes (or indices). Worse yet, the encoding can assign properties to classes that don't have them.

Matchers compare an encoded description of the developer's needs (query) to the encoded descriptions of the components in the library. The languages for describing queries and components should be identical or homomorphic. Given the number of translations and their complexity, there are ample opportunities for inaccuracy and ambiguity, and it is a wonder any needs get satisfied! Any one of the above steps have kept information retrieval (IR) researchers busy for years. A full survey of IR research is beyond the scope of this paper. We explore below the relation between encoding languages and methods and the corresponding matching algorithms, in general, and from a software reuse perspective.

A developer query may be seen as a predicate Q(.) that retrieves software components X such that Q(X) is true. Each class C in the class space is characterized by a description represented as a predicate D_C(.) such that *all* (and *only*) instances Y of the class C are such that D_C(Y) is true. As mentioned above, encoding generally results into a loss of information by replacing the actual description D_C of a class C by a "simpler" description $\overline{D}_c(\cdot)$. Accordingly, retrieval takes place by comparing Q(.) (the developer query) to the *approximate descriptions* of component classes ($\overline{D}_c(\cdot)$s). We distinguish between two basic matching approaches:

1) *partial-order-based retrieval*: The retrieval algorithm returns classes of items such that D_C(.) **LT** Q(.), for some partial-order relation **LT**. Notice that in practice $\overline{D}_c(\cdot)$ will be used for comparison. If **LT** is a logical implication, the algorithm returns classes C such that D_C(.) → Q(.). In this case, for all X such that D_C(X) ($\overline{D}_c$(X)) is true, we are assured that Q(X) is true.
2) *distance-based retrieval*: The retrieval algorithm returns classes of items such that Dist(D_C(.), Q(.)) (Dist $\overline{D}_c(\cdot)$, Q(.))) is smallest, where Dist is generally a metric that satisfies the following property:

$$[D(.)\ \mathbf{LT}\ Q(.)] \rightarrow [\mathrm{Dist}(D(.), Q(.)) = 0]$$

Intuitively, Dist. measures the extent by which $D_c(\cdot)$ ($\overline{D}_c(\cdot)$) fails the partial-order relation. Needless to say, in either case, the quality of retrieval depends on the quality of indexing, i.e., the relation between $D_c(\cdot)$ and $\overline{D}_c(\cdot)$.

Independently of the retrieval used, queries seldom return software components that fit the needs exactly, especially when those needs are not very precise in the beginning. Thus, most likely, the components retrieved will have to be adapted, and the assessment stage consists of evaluating the retrieved components for the efforts required to modify them. We consider assessment to be an integral part of retrieval. For instance, developer queries should be seen from the following perspective:

> Find components that satisfy the functional requirements Q(.) *OR are easy to modify so that they satisfy Q(.).*

In other words, a binary predicate IsEasyToAdaptTo(., Q(.)) should be appended, implicitly or explicitly, to any developer query. What makes a component Y easy to adapt to the set of requirements Q(.)? There are two kinds of criteria that make a component easy to adapt:

1) some general criteria related to the intrinsic quality of the component to be adapted and
2) differences between the retrieved component and the requirements Q(.).

In other words, the predicate IsEasyToAdaptTo(., Q(.)) can be seen as the conjunction of two predicates:

$$\mathit{IsEasyToAdaptTo}(X, Q(.)) \equiv \mathrm{EasyToModifyInGeneral}(X) \wedge [\mathrm{CostOfImplementingDifference}(D_{\text{class of X}}(.), Q(.)) \text{ on X is small}]$$

where $D_{\text{class of X}}$(.), or D_X(.) for short, is the description of X's properties.

Criteria that make a component easy to understand and adapt include the complexity of the component (size, cyclomatic complexity, number of inputs and outputs, etc.) and the quality of its documentation [28], [131], [159]. Estimating the cost of implementing the difference between D_X(.) and Q(.) on X poses two major difficulties. First and foremost, we have to develop a procedure that, based on the difference between two descriptions D_X(.) and Q(.), tells what kind of changes need to be incurred to components with description D_X(.) so that they fit in (satisfy) the description Q(.). In fact, *distance-based retrieval is fairly useless if the measure Dist doesn't correlate*

somehow to the amount of effort required to adapt the component; this issue is discussed in more detail in the next section. Second, we need a way to estimate the costs of making various kinds of changes to a program. Changing a program may involve modifying its interface, its (internal) structure, or both. Clearly, the cost of adaptation depends on the scope of change (interface alone, versus internal logic) and extent of the change. As mentioned in Section III.B.4, more work is needed in this area.

A.2. Evaluating Retrieval Performance

Traditionally, retrieval quality is measured by recall and precision. Recall measures the ratio of number of *retrieved* and *relevant* items to the total number of *relevant* items in the information/knowledge base. Precision measures the ratio of the number of *retrieved* and *relevant* items to the number of *retrieved* documents. Such measures are only adequate for partial-order-based retrieval, which assumes that relevance is a Boolean function, and have been widely criticized for this reason. We add to this the concern for estimating the effort required to adapt a component that doesn't match the developer's requirements. In this section, we study the properties that the encoding schemes and retrieval algorithms need to have to address these problems.

First, it is interesting to discuss the conditions under which we can achieve 100% recall and 100% precision. With partial-order-based retrieval, perfect precision (only items that *truly* satisfy the query are returned) implies that indexing/encoding should be such that:

$$(\forall C)\left[\left(\overline{D}_c(\cdot)\ \mathbf{LT}\ Q(\cdot)\right) \rightarrow \left(D_c(\cdot)\ \mathbf{LT}\ Q(\cdot)\right)\right] \qquad \text{(A1)}$$

The reader can check that condition (A1) is satisfied if for all C, $D_c(\cdot)\ \mathbf{LT}\ \overline{D}_c(\cdot)$. If the partial order is logical implication, the condition $\left[D_c \rightarrow \overline{D}_c\right]$ means *that all that have been encoded ("said") about classes is accurate*. We say that the encoding is *sound*. For perfect recall, encoding should be such that:

$$(\forall C)\left[\left(D_c(\cdot)\ \mathbf{LT}\ Q(\cdot)\right) \rightarrow \left(\overline{D}_c(\cdot)\ \mathbf{LT}\ Q(\cdot)\right)\right] \qquad \text{(A2)}$$

Condition (A2) is satisfied if for all C, $\overline{D}_c(\cdot)\ \mathbf{LT}\ D_c(\cdot)$. If **LT** stands for logical implication, this says that *all that is true (and functionally significant) about a class C, and possibly more (erroneously), has been encoded in* $\overline{D}_c(\cdot)$. Not surprisingly, to achieve perfect recall and precision, we need equivalence between the actual intension of classes ($D_C(.)$) and their encoding ($\overline{D}_c(\cdot)$); *logical* equivalence if **LT** is logical implication. In practice, neither is possible, as mentioned in the previous section. With distance-based retrieval, we don't need logical equivalence but the encoding process should be such that the ranking produced by *Dist* using $\overline{D}_c(\cdot)$ is similar to that which would have been produced using the actual $D_C(.)$. The following must hold:

$$\begin{aligned}&(\forall C_1, C_2)\\ &\Big[\Big(Dist\big(\overline{D}_{c_1}(\cdot), Q(\cdot)\big) < \\ &Dist\big(\overline{D}_{c_2}(\cdot), Q(\cdot)\big)\Big) - \rightarrow \\ &\Big(Dist\big(D_{c_1}(\cdot), Q(\cdot)\big) \le Dist\big(D_{c_2}(\cdot), Q(\cdot)\big)\Big)\end{aligned} \qquad \text{(B1)}$$

$$\begin{aligned}&(\forall C_1, C_2)\\ &\Big[\Big(Dist\big(D_{c_1}(\cdot), Q(\cdot)\big) \le \\ &Dist\big(D_{c_2}(\cdot), Q(\cdot)\big)\Big) \rightarrow \\ &\Big(Dist\big(\overline{D}_{c_1}(\cdot), Q(\cdot)\big) \le Dist\big(\overline{D}_{c_2}(\cdot), Q(\cdot)\big)\Big).\end{aligned} \qquad \text{(B2)}$$

Condition (B1) means that if an item C_1 is presented to the developer before item C_2, then it is truly more relevant[27] than C_2. Condition (B2) means that if C_1 is more relevant than C_2, then it will be presented to the developer before C_2. The reader can check that because, for a given query, *Dist* defines a total order on the class space, the two conditions are actually equivalent.

Another way of interpreting condition (B1) (or (B2)) is to say that encoding is monotonic, or, introduces a consistent bias. A cautionary note is, however, in order:

> **In a reuse context, the thoroughness of component encoding is limited by the developer's willingness to formulate long and precise queries**; there is no point in encoding every bit of relevant information about a component if a developer barely has the patience for typing string search regular expressions!

We now look at the issue of estimating the cost of adapting a component that does not match exactly the needs of the developer. Notwithstanding cases where a component fails to match the query on nonessential (nonfunctional) properties, distance-based retrieval implicitly assumes that the measure *Dist* somehow correlates to the effort required to adapt the component to match the query. It is fair to assume that the effort is related to the extent of structural changes needed to change the component. A component may be described by either its function (input-output relation, the "what") or its structure (the "how"). Typically, a developer queries the library for a component that fulfills a given purpose ("what"), i.e., by specifying its functional properties rather than by specifying or sketching its structure; the former being, supposedly, easier than the latter. Hence, for matching purposes, component classes are described by their functional properties. Therefore, *in order to estimate the structural changes needed to apply to a component that fails a developer query, we need to have a model of the relation between functional requirements (function, types of inputs/outputs, etc.) and component structure.*

27. The left-hand side of (B1) has strict inequality (<) because we don't care what happens for cases where $Dist\big(\overline{D}_{c_i}(\cdot), Q(\cdot)\big) = Dist\big(\overline{D}_{c_i}(\cdot), q(\cdot)\big)$, since encoding naturally maps distinct classes to equivalent codes.

The kind of knowledge needed to model a structure-function relationship is not much different from that needed in automatic programming. For instance, if we can characterize the structures that implement a given function, we are only one step away from generating those structures automatically based on the specification of the function! It is extremely difficult to characterize such structures, in no small measure because several algorithms, e.g., can implement the same function, and the same algorithm can implement several "functions," depending on the data it manipulates. However, we should mention that automatic programming systems do not generate *all the possible programs* that can satisfy a given set of requirements; one that does suffices. Further, we do not need to characterize the full range of (function, structure) pairs, but rather the structural modifications associated with "small" functional differences. In other words, if FS is the mapping that associates to a function f a set of structures $\{s_i\}_i = \mathrm{FS}(f)$, we do not need a characterization of FS, but rather of $\mathrm{FS}(f + \Delta f) - \mathrm{FS}(f)$. This reduces the problem to finding types of functional differences that can be accommodated by (measurably) small structural differences,[28] knowing that several kinds of structural differences can accommodate a given functional change. In the next section, we will comment on the extent to which the encoding and retrieval methods discussed address this problem in one form or another.

A.3. A Survey of Existing Approaches

Existing approaches to software component retrieval cover a wide spectrum of component encoding methods and search or matching algorithms. The encoding methods differ with respect to their soundness, completeness, and the extent to which they support an estimate of the effort it takes to modify a component. Striving for any of these qualities makes encoding more complex and costlier. It also makes it possible to support more sophisticated retrieval, provided queries of equal richness and expressiveness to that of the encoding scheme are used. In practice, there is a limit to how complex queries can be for component search and reuse to be worthwhile (Section III.B.1). Accordingly, overly complex encoding schemes are wasteful unless reusers are provided computer assistance in formulating equally complex queries. The approaches discussed below strike different balances between complexity and cost on one hand and retrieval quality on the other. Further, some are immediately practicable and have been used in a production setting, while others are mere theoretical explorations. We discuss three major classes of encoding and retrieval approaches, by increasing order of complexity:

1) *text-based* "encoding" and retrieval,
2) *lexical descriptor-based* encoding and retrieval and
3) *specification-based* encoding and retrieval.

With text-based encoding and retrieval, the textual representation of a component is used as an implicit functional descriptor: Arbitrarily complex string search expressions supplied by the reuser are matched against the textual representation (see, e.g., [113]). The main advantage of such an approach is related to cost: No encoding is required, and queries are fairly easy to formulate. Its disadvantages have been thoroughly investigated in the information retrieval literature [20]. Simply put, plain-text encoding is neither sound nor complete. Short of a full-fledged language understanding system that takes into account the context, the presence of a concept (term or phrase) in the text does not imply that the component is about that concept (e.g., "Unlike quicksort, this procedure..."). Conversely, the absence of a concept from the text does not mean that the component is *not* about that concept, as different developers and documenters may use a different terminology.

Plain text encoding and search, and variants thereof, have been used in a number of software libraries (e.g., [48], [93], [113]), alone or in conjunction with other search methods, and had fairly good recall and precision rates. In a controlled experiment performed at the Software Productivity Consortium, Frakes and Pole found that more sophisticated methods (see below) had no provable advantage over plain text retrieval in terms of recall and precision [49]. However, they found that developers took 60% more time than with the best method to be satisfied that they had retrieved all the items relevant to their queries. This accounts for both the speed with which individual search statements/expressions can be formulated and the number of *distinct* search statements that had to be submitted to answer the same query. With traditional document retrieval systems such as library systems, longer search times are a mere annoyance. *In a reuse context, bigger search times can make the difference between reusing and not reusing* (Section III.B.1).

With lexical descriptor-based encoding, each component is assigned a set of *key phrases* that tell what the component "is about." We could define a two-place predicate *IsAbout* (.,.), where key phrase K is assigned to a component (or component class) C iff *IsAbout*(K, C). If C is assigned a set of phrases $\{K_1, ..., K_n\}$, then *IsAbout*(C, K_1) Λ...Λ *IsAbout*(C, K_n) is true, and the one-place predicate *IsAbout*(., K_1) Λ...Λ *IsAbout*(., K_n) may be considered as the description of the component (class) C (Section V.A.2). Typically, subject experts inspect the components and assign to them key phrases taken from a *predefined vocabulary* that reflects the important concepts in the domain of discourse (see, e.g., [6], [27], [113], [131]). Notwithstanding the possibility of human error and the coarseness of the indexing vocabulary, such encoding is sound, as opposed to plain-text encoding. Further, because a key phrase need not occur in a component's textual representation to be assigned to it, it is also *more* complete[29] than plain text encoding. Reusers formulate their queries as Boolean expressions of key phrases. Let Q = E(K' *sub* 1, ..., K' *sub m*), a Boolean expression with terms K'_1, ..., K'_m. A component C with key phrases K_1, ..., K_n is considered relevant to the Boolean expression (query) Q iff *IsAbout*(., K_1) Λ...Λ *IsAbout*(., K_n) $\rightarrow$ *E*(*IsAbout*(., K'_1), ..., [*sAbout*(., K'_m)), or, equivalently, if K_1 Λ...$K_n \rightarrow Q = E(K'_1, ..., K'_m)$. *Boolean retrieval*, as it is called, corresponds to what we called partial-order-based retrieval.

28 For example, we can compare graphical representations of programs obtained through data and control flow analysis, see, e.g., [86].

29 An encoding scheme can be considered complete only if it says everything of consequence about the component; that is hard to define.

In practice, the method presented above is refined in many ways. In one refinement, instead of using the one generic relation *IsAbout*(.,.) between components and descriptors, several specific relations are used such as *HasFunction*(.,.), *ApplicableToDomain*(.,.), *OperatesOn*(.,.), etc. [27], [112], [131]; this is commonly referred to as *multifaceted classification* in the information retrieval literature, where each facet corresponds to a relation, and the descriptions for each facet are logically ANDed. For example, if we use the facets *HasFunction* and *OperatesOn*, a routine that sorts both arrays and linked lists may be described by the one-place predicate [*HasFunction*(., Sorting)] Λ [*OperatesOn* (., Array) Λ *OperatesOn*(., LinkedList)]. Similarly, reuser queries are now formulated using a conjunction of Boolean expressions, one for each facet on which the reuser wishes to search.

A second set of refinements amend the retrieval algorithm itself to handle approximate matches. We illustrate them for simple (single-facet) indexing; extending them to multifaceted indexing is fairly straightforward. First, if there is a partial order between key phrases themselves, the partial order may be used to extend queries. Assume for example that the key phrases are organized in a taxonomy. Let K_1 and K_2 be two phrases such that K_1 has an "is-a" relation with K_2. By definition, any component C that is about K_1 (*IsAbout*(C, K_1)), by default, is also about K_2 (*IsAbout*(C, K_2)). However, the reverse is not true. Thus, in the process of looking for components that are about K_2, the ones that are about K_1—and K_2's descendants in general—would also do. This approach is used in MEDLINE [102], an on-line medical literature retrieval system operated and maintained by the (U.S.) National Library of Medicine. Two additional refinements turn Boolean retrieval—which is partial-order-based—into distance-based retrieval. The first ranks components by decreasing number of key phrases that match phrases from the query [142]. The second method is used when key phrases are organized in a taxonomy (see, e.g., [134]) or a weighed semantic net in general (see, e.g., [131]). The former has been used in the European ESPRIT Practitioner (software reuse) project (see, e.g., [113]). The latter has been more widely used in software libraries (see, e.g., [27], [48], [131]). In SoftClass, a prototype CASE tool with integrated support for reuse, we implemented three classes of lexical descriptor-based component retrieval algorithms that combine the above features with weighed facets and a number of fuzzy bells and whistles [112]. We are currently setting up a series of experiments to compare the different methods. However, we don't expect significant improvements to result from some of the refinements mentioned above.

Lexical descriptor-based encoding and retrieval suffers from a number of problems. First, an agreed (or agreeable) vocabulary has to be developed. That is both labor-intensive and conceptually challenging. Sorumgard et al. reported a number of problems developing and using a classification vocabulary [151]. They experienced known problems in building indexing vocabularies for document retrieval, including trade-offs between precision and size of the vocabulary and the choice between what is referred to as precoordinated or post-coordinated *indexing*,[30] with the confusion that may result from mixing the two [151]. Software-specific challenges include the fact that one-word or one-phrase abstractions are hard to come by in the software domain [83], [151].

Further, it is not clear whether indexing should describe the computational semantics of a component, as in "this procedure returns the record that has the highest value for a float field, among an array of records," or its application *semantics*[31] as in "this procedure finds the highest paid employee within the employees file" [151]. Characterizing computational semantics is important for reuse across application domains. However, reusers may have the tendency to formulate their queries in application-meaningful terms. Formal specification methods suffer from the same problem, but to a lesser extent, since application semantics show up specifically as terminal symbols in the specification language. Finally, neither the encoding mechanism nor the retrieval algorithm lend themselves to assessing the effort required to modify a component that does not match the query perfectly. This is so because the descriptors have externally assigned (linguistic) meanings and bear no relationship to the structure of the components. For example, what does it mean for a component *C* to have the function *Sort*, i.e., what does it mean to have *HasFunction*(*C*, Sort)? it only means what we wish the symbol "Sort" to mean to us, and any relation between two symbols has to be posited by us, rather than proven by a formal proof system.

From the reuser's point of view, a familiarity with the vocabulary is needed in order to use a component retrieval system effectively [142]; a hierarchical (e.g., taxonomical) organization of the key phrases and proper browsing tools can alleviate the problem significantly [113]. Further, queries tend to be fairly tedious to enter, especially for the case of multifaceted encoding. In SoftClass, where software components are grouped into component categories, each with its own facets, queries are entered by filling out a simplified component template that stands for the prototypical component the developer wishes to retrieve [113]; the filled out template is then internally translated into a Boolean query and matched against the component base. While this format does not handle all kinds of queries efficiently, we believe that it handles the most common queries efficiently [113].

Specification-based encoding and retrieval comes closest to achieving full equivalence between what a component is and does and how it is encoded. With text and lexical descriptor-based methods, retrieval algorithms treat queries and codes as mere symbols, and any meaning assigned to queries, component codes, or the extent of match between them is external to the encoding language. Further, being natural language-based, the codes are inherently ambiguous and imprecise. By contrast, specification languages have their own semantics within which

30. (Very) roughly speaking, with precoordinated indexing, several phrases for the same facet are to be interpreted *disjunctively*, while with post-coordinated indexing, they should be interpreted *conjunctively*; see [112].

31. Using our approach to object-oriented design (seec Section IV.C.3 and [111]), generic design *templates* would be described by their computational semantics. Application data *structures* would be characterized *explicitly* by their application semantics, and implicitly, through the generic design template to which they map, by their computational semantics.

the fitness of a component to a query can be formally established [32], [108], [161], and "mismatches" between the two can be formally interpreted [108], [161]. Typically, the formal specification-based methods correspond to what we called partial order-based retrieval, using a partial-order relationship between specifications. This partial order is often used to preorganize the components of the library to reduce the number of comparisons between specifications—an often prohibitively costly operation [108], [115]. The methods discussed in the literature differ in the expressiveness of the specification language. Also, different retrieval algorithms take advantage more or less fully of the power of the specification language. The subset of the language used for retrieval *often* has no effect on recall, but degrades precision, as in using operations' signatures instead of using signatures *and* pre- and postconditions.

In [108], A. Mili et al. describe a method for organizing and retrieving software components that uses relational specifications of programs and refinement (contravariance) ordering between them. Any given program is correct with respect to (satisfies) the specification to which it is attached, as well as the specifications that are "above" it. Hence, a specification retrieves the programs attached to it, as well as those attached to specifications that are "below" it. A theorem prover is used to establish a refinement ordering between two specifications. Two forms of retrieval are defined: *exact retrieval*, which fetches all the specifications that are more refined than a reuser-supplied specification, say K, and *approximate retrieval*, which is invoked whenever the exact retrieval fails, and which retrieves specifications that have the biggest "overlap"[32] with K [108]. One nice property about approximate retrieval is that, while it does not directly assess the effort required to modify a component, the difference actually means something (e.g., a program is found that gives the desired outputs for a subset of the inputs) and may suggest a way of modifying the returned programs to make them satisfy the requirements or use several of them in combination [108].

Chen et al. proposed a similar approach that uses algebraic specifications for abstract data types and an *implementation* partial ordering between them [32]. Reusable components, which may be seen as abstract data types, are specified by both their signature and their *behavioral axioms*. However, while the *implementation* relationship takes into account the behavioral axioms, the retrieval algorithm uses only signatures, modulo a renaming of the "types" of the components to match those of the query [32]; the authors did envision using an "interactive system [read semiautomatic] for algebraic implementation proofs" [32]. Moorman-Zaremski and Wing propose an approach based exclusively on signature matching [161]. The major advantage of their approach is that the information required for matching can be extracted directly from the code. They first define *exact matches* between function signatures, to within parameter names, and then define module signatures and partial matches between modules using various generalization and subtyping relationships [161]. They too envision taking into account behavioral specifications in future versions, using LARCH (cf. [60]) specifications—which would then have to be encoded manually.

None of the formal specification-based methods we know about addresses directly (or successfully) the issue of assessing the effort required to modify a component returned by approximate retrieval (inexact match). Further specification-based methods that include *behavioral* specifications (and not just signatures) suffer from considerable costs. First, there is the cost of deriving and validating formal specifications for the components of the library (see also [115]). This cost is recoverable because it could be amortized over several *trouble-free* uses of the components and is minimal if specifications are written *before* the components are implemented, which is the way it should be (and seldom is) done. The second cost has to do with the computational complexity, if not outright undecidability, of proof procedures. This cost can be reduced if actual proofs are performed only for those components that match a simplified form of the specifications, e.g., the signature; not much else can be done about the inherent complexity of proof procedures or their undecidability without sacrificing specification power. Last but not least, there is the cost for the reuser to write full-fledged specifications for the desired components. Because there is no evidence that specifications are either easier or shorter to write than programs, reusers need motivations other than time-savings, or computer assistance, to write specifications for the components they need. We believe that formal specification-based matching will remain a theoretical curiosity for the time being and will integrate only in the more formal development methods that address application domains such as reactive and real-time systems.

B. Component Composition

Under component composition, we address two dual sets of issues:

1) Given a set of components, and a schema for composing them, check that the proposed composition is feasible (*verification*) *and* satisfies a given set of requirements (*validation*); we refer to this as the composition verification and validation problem and
2) given a set of requirements, find a set of components within a component library whose combined behavior satisfies the requirements; we refer to this as the bottom-up design problem.

The first problem benefits from a large body of work that is not often associated with reuse. A thorough coverage of this problem is beyond the scope of this paper. We will be content to highlight the general issues and describe a representative sample of work in this area (Section V.B.1). The second problem, discussed in Section V.B.2, benefits from work on verification and validation of compositions, but presents challenging search problems of its own.

32. The overlap between two specifications is determined using the "meet" lattice operation [108].

B.1. Composition Verification and Validation

Component composition verification and validation poses three challenges:

1) designing a language for describing compositions of components that lends itself to verification and validation,
2) performing verification and
3) performing validation.

There are two general methods for describing compositions of components. If we think of a component as consisting of a specification and a realization (or a set of realizations, see, e.g., [83]), then composition may occur either at the specification level or at the realization (implementation) level (see also Section V.B.3). Specification languages usually provide built-in composition operators with well-defined semantics. For example, with relational specifications, any of the set theoretic and relational operations may be seen as a composition. In this case, we might say that the set of specifications is closed under composition, and verifying or validating a composition of specifications or validating it against a target specification is not different from verifying any individual specification or validating it against another specification. With regard to validation, we can expect the same challenges discussed for specification-based component encoding and retrieval (Section V.A.3). The problem with specification-level composition is that it is often difficult to characterize specification-level manipulations by manipulations on the actual realizations (programs) of these specifications [108].

When composition takes place at the component realization level, we obtain a (much) smaller range of behavioral compositions, but we are assured that these compositions are feasible without additional development. Component compositions are usually described using the so-called *module interconnection languages* [61], [129]. A module may be seen as having an *internal structure*, consisting of a set of data structures and a set of procedures that reference them, and an *external interface* specifying the external entities the module depends on and the internal entities the module exports. Module interconnection languages describe component (module) compositions by specifying:

1) the *obligations* of the individual participants and
2) the *interactions* between the components.

The specification of the obligations of the individual components consists, minimally, of the signatures of the operations they need to support; this is similar to Ada's constrained generics, where generic packages list the operations that type parameters have to support [106]. It could also include the specification of the *behavior* of the operations. This is the approach followed in Helm et al.'s *contracts* [65] and a number of algebraic specification-based interconnection languages such as Goguen's library interconnection language (LIL [57]) and other derivatives of OBJ or LILEANNA [153], e.g., an application of LIL's concepts to *ANN*otated *A*da packages. One of the interesting features of LIL is that obligations are specified in terms of *theories*, and a given module (in this case, an abstract data type) may satisfy a theory in different ways, called *views* [57]. This has the advantage of ignoring operation names during verification, by focusing on their behavioral semantics instead.

The specification of the interactions between the components varies from simple call dependencies [61] to a full-fledged behavioral specification including interaction logic, aggregate-wide preconditions, postconditions, invariants, etc. (see, e.g., [61], [65], [109]). Behavioral interactions between components can also be specified *implicitly* in logic-based (or logic-flavored) languages. One such language is MELD, an OO language developed by Kaiser and Garlan [78]. In MELD, classes are represented by *features*. Methods are represented by semideclarative[33] constructs called *action equations*. When the same methods are implemented by two features, their action equations are merged. In case the merge creates dependencies, a topological sort determines which action equations are to be executed first [78]. This constitutes, in our opinion, MELD's most interesting feature for reuse by composition as it automates code-level integration. Validating the behavior of a composition of modules against a desired behavioral specification is generally a difficult problem [84], [163]. One of the major difficulties is due to the fact that it is difficult to get a closed form expression for the behavior of the aggregate. This is due to the fact that the language used for describing compositions is different from that used for specifying individual components (see, e.g., [163]).

B.2. Bottom-Up Development

Top-down development consists of decomposing the requirements for a module into requirements for a set of (simpler, more reusable, etc.) submodules and patterns of interactions between them. Informal requirements analysis and specification methods use informal heuristics to guide the decomposition process. Formal methods use various reduction and factoring mechanisms to decompose specifications (see, e.g., [30], [33]). In both cases, the decomposition is guided by properties that the component submodules and their patterns of interactions have to have. For non-trivial requirements, a virtually limitless number of solutions of equal quality could be found. With bottom-up development, the major requirement is that the decomposition yields specifications for which components have already been developed. This is generally a very difficult problem.

Enumerating compositions of components within a library "until one is found that has the desired behavior" seems unthinkable at first. In practice, a number of practical and theoretical considerations can make the number of compositions to explore manageable. In [62], Hall describes a component retrieval method that explores combinations of components when none of the individual components matches the user query. Users specify the desired behavior by giving an example input-output pair $<I, O>$. The idea of retrieving components based on the output they *actually* produce when executed on a user-supplied input was first proposed by Podgurski and Pierce [127]. Hall extends Podgurski and Pierce's work by

33. Action equations are essentially predicates. However, they do contain some control information such as iteration and sequencing.

exploring compositions of components when no single component is found that returns O given I. For example, if all the components in the library are single-parameter functions $f_1, ..., f_n$, then if for all i, $f_i(I) \neq O$, we try out $f_j(f_i(I))$, for all $1 \leq i, j \leq n$, and if none is found that returns O, we try three levels deep function compositions, etc. [62]. Hall showed that, in general, the number of compositions of components of depth d or less is doubly exponential in d, i.e., of the form $O(n^{(n^d)})$. However, a number of techniques help reduce that number considerably, without missing any potential solutions. Type-compatibility requirements considerably restrict the range of possible compositions. More dramatic results are obtained by dynamic programming: When generating compositions of depth d ($f_{i_1} o \ ... \ of_{i_d}$), apply new components to all the combinations of distinct return *values* of depth $d - 1$, rather than all combinations of distinct *programs* of depth $d - 1$.

Hall tested his algorithm on a library of 161 Lisp functions. The retrieval system itself is written in Common Lisp. He limited the depth of compositions to three, with the level three functions limited to those that have a single input. Fifteen queries took an average of 20 seconds, and a maximum of 40 seconds, running on a SUN SPARC II. In one example, a query provoked 2,400 component executions instead of a potential 10^{16} executions [62]. While more testing is needed to assess the efficiency of the algorithm, the processing times remain reasonable for a reasonably large library and show that the method could indeed be computationally practical. However, as Hall pointed out, this method could not be applied to retrieving nonexecutable reusable components [62]. A less serious engineering difficulty has to do with multiplatform libraries, components that raise exception while being tested out or loop endlessly, and components with side-effects, all of which pose challenging but tractable engineering problems [62].

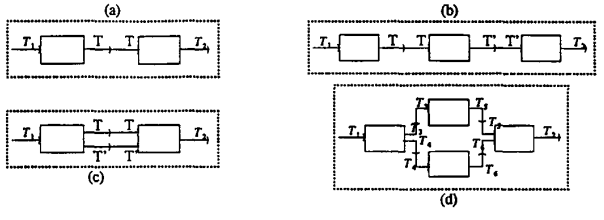

Fig. 4. Alternative compositions that could be considered by the composition retrieval algorithm.

We have started work on a combination of Hall's work and Moorman-Zaremski and Wing's work on signature matching [161]. Assume that a developer is looking for a component that takes an input variable of type T_1 and produces an output variable of type T_2. Failing to find such a component, an algorithm could find any of the compositions shown in Fig. 4, that "concatenates" components based on type compatibility between their inputs and outputs. The *function realization problem*, as we called it, consists of finding all the compositions of functions (signatures) that consume no more than the inputs specified by the developer's query and produce at least the outputs specified by the developer's query [112]. We showed in [112] that the *set cover problem*, which is known to be NP-complete [54], could be reduced to the function realization problem. Worse, we know of no heuristic that guarantees a solution value within a constant factor of the optimum cover of a set [54], which suggests that none could be found for the function realization problem either. Fortunately, finding out whether a function signature has a realization or not can be done in polynomial time [112]; bear in mind, though, that a realization does not necessarily exhibit the desired behavior. Because several realizations could be equivalent, we defined a "minimal" form for realizations called *normal realization*, where each function is needed (i.e., without it the composition would not be a realization), and where each function has the minimal possible depth, i.e., is "called as soon as all its inputs are available." We developed an algorithm for finding normal realizations and implemented it in Lisp [112].

Our method has the advantage of not requiring component execution, and like Moorman-Zaremski and Wing's method [161], the information required for search can be extracted from components themselves. However, programming language types alone can be hopelessly nondiscriminatory. A library written in a weakly typed language (e.g., C) is likely to have a handful of types, and the algorithm will have a dismal precision. Application-oriented definitions of types can sharpen the search but may miss out on some valid realizations [62]. We are currently testing the algorithm on a library of data manipulation functions (string manipulation, data conversion, etc.). While we do not expect a good precision, we are hoping that an inspection of the results will help us recognize classes of realizations or subrealizations that should be pruned out of the search, thereby increasing the efficiency of the algorithm. However, we expect most of the gains to come from using richer semantics for types and type compatibility, and we are pursuing work in that direction as well.

C. Adapting Reusable Components

We use the term adaptation to refer to what happens to a component between the time a decision is made to reuse it and the time it has become part of the product. We recognize three potential subtasks,

1) what Krueger called *selection* [83]: if a reusable component has a variable part or explicitly enumerated alternative implementations, select (the) one that is appropriate for the problem at hand,
2) *modification*: in case the component or any of its variants cannot be used as is and
3) *integration*, which is essentially a verification step that checks whether the component is compatible with its environment.

One of the major differences between selection and modification is that with selection, the changes to the component have been planned ahead of time. This is generally done using various parameterization and abstraction techniques and will be discussed briefly below. With *modification*, the changes are often unanticipated or poorly planned. As mentioned in Sections III.A.3 and III.B.1, modifying reusable components may defeat both the quality and the productivity advantages of reuse. Hence, it should be automated as much as possible to save

time and ensure that the modifications are quality-preserving. We discuss modification in the context of transformational systems. As for the integration of reusable components, what is not addressed by module interconnection languages discussed in Section V.B.1 is not specific to the reusability of the components and will not be discussed further here.

C.1. Selection

Two commonly used selection mechanisms are specialization and instantiation of abstract software components. Abstraction has been supported by programming language constructs for some time [143]. At the most basic level, declaring program constants or using variable-dimension arrays is a form of parameterized programming. Conditional compilation is another more sophisticated form of parameterized software, whereby different code sequences are compiled based on a number of system and environmental parameters. In this case, adaptation consists simply of setting the right environmental parameters. In general the mechanisms involved depend on the nature and complexity of the parameters, ranging from a simple compile or linkage-time binding (e.g., of an unresolved reference to a type T to a specific type), to a mix of substitutions, conditional compilation and code generation as in [16], and template-based approaches in general (see, e.g., [85], [137]).

Object oriented languages support a number of abstraction and selection mechanisms, including generic classes, abstract classes, and metaclasses with metaprogramming [72]. *Genericity* supports the development of complex data structures with parameterized component types. For example, one could define a generic list structure LIST[T] whose node values (or "data" fields) are of a generic type T that supports comparison operators. In this case, selection consists of using the (e.g., declaring a variable of) data type LIST[< >] by specifying an actual type instead of the parameter T. The obligations of the type T may be specified explicitly in the specification of the generic type (called constrained *genericity* [106]) or implicitly based on what types will actually compile or execute. With *abstract classes* (Section IV.B.1), selection consists of choosing among several concrete subclasses that *conform* to the behavior of the abstract class, or creating a subclass of our own, to address the specific needs of the application at hand [72]. Note that subclassing alone does not guarantee that a class conforms to its superclasses, i.e., that the types they implement are in a subtype relationship. We showed in Section IV.B.3 how design-level considerations may lead to situations where subclasses do not implement subtypes, and vice-versa. Unfortunately, few programming languages (e.g., Eiffel [106]) ensure that subclasses implement types that are in a subtype relationship, and subclassing remains essentially a code-sharing mechanism, with the problems we discussed in Sections IV.B.3 and IV.B.4. Finally, the use of *metaclasses with metaprogramming* may be seen as an OO packaging (design) for program generators (Section IV.2) and will not be discussed further here; the reader is referred to [72] for a more thorough discussion.

C.2. Modification

Modification is required when a retrieved component has to be *reworked* to accommodate the needs of the application at hand. The need for modification may become clear during retrieval: The *encoded description* of the component does not match perfectly the query. Alternatively, a closer inspection of a component whose encoded description did match the query may reveal inadequacies. The latter case is possible because encoding is often incomplete: The encoded description leaves out some functional properties of the component. We saw in Section V.A.2 that the mismatch between a query and the encoding of a component may be more or less revealing as to the (extent of the) changes that need to be incurred to the component, depending on the completeness of the encoding. For the purposes of presentation, we consider the two situations as instances of the same problem: Given the specification of a desired component S_D, the "closely matching" specification S_C of an existing component C, and its realization R_C (i.e., implementation), find the realization for the desired component—call it R_D; the additional problem of working our way back from differences between encoded (partial) description used for retrieval to actual functional specifications raises similar issues because in both cases we have to walk our way back, upstream of an information-losing mapping (see Fig. 5).

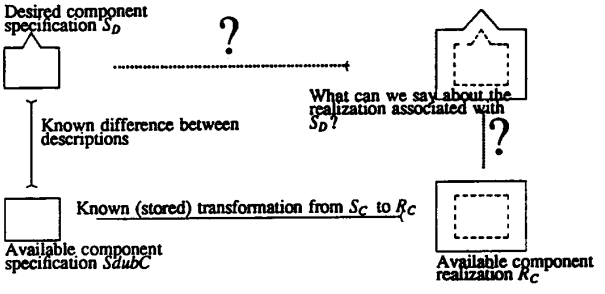

Fig. 5. The component modification problem.

This problem is best understood in the context of a transformational view of software development. Software development is seen as a (possibly long) sequence of transformations, starting with more or less formal specifications, leading eventually to executable code [1]. Fig. 6 shows the typical software life cycle for transformational systems. In such systems, software development consists of two major steps:

1) deriving *formal* specifications from user requirements—if the transformations to be applied are to do any substantive work and
2) applying a set of transformations on the formal specifications, gradually building a computer program, in either a target programming language or in a readily translatable language [123].

An important characteristic of transformational systems is the potential for maintaining software systems at the specifications level [8]. According to this view, the complexity of software development lies not in the individual transformations, but in applying the "right" transformation when several alternatives

are possible. Existing transformational systems provide varying levels of support for selecting the "right" transformations, ranging from simply enacting/executing transformations chosen by the developer to full automation [8], [123].

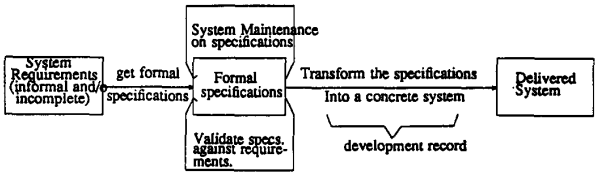

Fig. 6. Typical software life cycle in transformational systems. Adapted from [1].

Regardless of the "intelligence" of the transformations (e.g., their knowledge about their own appropriateness), the transformational approach has the following advantages:

1) relieving developers from labor-intensive, knowledge-poor tasks,
2) virtually eliminating clerical errors,
3) ensuring correctness of the resulting programs by construction and
4) maintaining a record of development choices, their rationale, or both for maintenance purposes.

It is this last characteristic of transformational systems that concerns us most in this paper, namely, the potential for software reuse. Maintenance has been recognized by a number of researchers as a particular form of reuse (see, e.g., [14]). Balzer et al. recognized transformational systems' potential for enhancing software reuse, whereby reusable components are maintained and modified at the specification level rather than at the implementation level [8]. While a number of researchers have recognized the importance of recording development decisions for reuse purposes, the transformational approach to software development makes *explicit computer-supported* use of those decisions to maintain/reuse existing software.

The transformational approach makes maintenance and reuse easier, not only because it makes development easier in general, but also because "similar" inputs (e.g., formal specifications or any other intermediary form) *often* call for the same transformations to be applied. Cases where the same sequence of transformations cannot be replayed, e.g., the preconditions of one of the transformations fail to hold as a result of a modification, developer intervention is only needed from that point onward. Baxter studied the commutativity and dependencies between transformations to minimize the scope of modifications [17]. Take the example of a program P that was derived from a specification S using the chain of transformations $T_n \, o \, ... \, o \, T_i \, o \, ... \, o \, T_1$ and assume that S was modified into S′ such that all transformations up to, not including, T_i (i.e., $T_1, ..., T_{i-1}$) were applied successfully. Normally, a developer would have to intervene to choose an alternative transformation to T_i and proceed from that point to the end. However, if it is known that some transformation T_j, for $i < j \leq n$, commutes with (thus, independent of) T_i, then it could be moved (applied) ahead of T_i, and the developer would have fewer transformations to consider. Generally speaking, replaying design histories is not foolproof. The level of confidence in the replayed process depends on the knowledge embodied/used in selecting transformations (e.g., completeness and soundness of preconditions) and the responsiveness of the transformation selection algorithm to specification changes. One can easily imagine a case where an innocuous change in specifications might require a significant change in program structure to maintain a similar level of performance.

Transformational systems have been criticized by some researchers for their limited range of applicability [123]. The programs generated with this approach were mostly toy examples, as was the case with other AI-oriented automatic programming systems [9]. While, in principle, the transformational approach is not limited to small programs, the amount of knowledge that needs to be encoded to handle large software systems is prohibitively large. Most of the earlier transformational systems embody basic, domain-independent, *programming knowledge*. More recent efforts such as the DRACO system [121] support domain-specific specification languages, and transformations embody some form of domain knowledge. Software reuse research may well benefit from relaxing the formal correctness-preserving nature of transformations and fromusing more heuristic rules such as analogical reasoning [29], especially when we deal with informal or poorly structured software products (see, e.g., [94]). Alternatively, we could settle for localized or partial transformational approaches, as opposed to ones covering the entire specification → program cycle.

To the extent that software reuse benefits from automatically propagating software changes across development stages, work on configuration management systems and program dependencies is eminently relevant. Configuration management systems are concerned primarily with maintaining the integrity of software systems and the interoperability of components as they undergo change [120]. Minimally, such systems help localize the effects of changes [92]. On a different scale, work on program dependencies is concerned primarily with the *local* effects of change, typically within a procedure. Typically, data and control flows within a program are analyzed, thereby identifying the parts that depend on a particular datum/control statement. Such analyses support reuse in many ways:

1) localizing the effects of changes [117], [126], thus guiding reusers in the process of adapting retrieved components to their needs,
2) simplifying program structures [71], which enhances program readability and understandability and
3) "slicing" programs to extract specific functionalities [53], [71], in case the retrieved component does more than what is required.

The latter is an interesting dual to reuse by composition (see Section V.B). We should mention, however, that reliable flow analysis depends on a number of restrictive assumptions, such as the absence of side-effects and "global variables" (see, e.g., [117], [126]). On the positive side, they help increase the reuse worth of code fragments by automating some of the code modification tasks and do not require the availability of design or analysis information. A combination of *macroscopic* con-

figuration management and *microscopic* program flow analysis can help reduce the cost of maintaining and reusing software.

VI. Summary and Discussion

Reuse is the default problem-solving strategy in most human activities [88], and software development is no exception. Software reuse means reusing the inputs, the processes, and the outputs of previous software development efforts. Software reuse is a means toward an end: improving software development productivity and software product quality. Reuse is based on the premise that *educing* a solution from the statement of a problem involves more effort (labor, computation, etc.) than *inducing* a solution from that to a similar problem, one for which such efforts have already been expended. While the inherent complexities in software development [26] make it a good candidate for explorations in reuse, it is far from obvious that actual gains will occur. The challenges are structural, organizational, managerial, and technical. In this paper, we discussed some of the most important issues, with an emphasis on the technical ones.

Economic considerations, and cost/benefit analyses in general, must be at the center of any discussion of software reuse. Notwithstanding differences between reuse approaches along the building blocks—generative dimension, it is useful to think of software reuse research in terms of attempts to minimize the average cost of a reuse occurrence (see Section III.B.1):

$$[Search + (1 - p) \times (ApproxSearch + q \times Adaptation_{old} + (1 - q) \times Development_{new})]$$

where *Search*(*ApproxSearch*) is the average cost of formulating a search statement to a library of reusable components, and either finding one that matches exactly (approximatively) the requirements, or be convinced that none exists, $Adaptation_{old}$ is the average cost of adapting a component returned by approximate retrieval, and $Development_{new}$ the average cost of developing a component that has no match, exact or approximate, in the library. For reuse to be cost-effective, the above must be smaller than:

$$p \times Development_{exact} + (1\text{-}p) \times q \times Development_{approx} + (1 - p) \times (1 - q) \times Development_{new})$$

where $Development_{exact}$ and $Development_{approx}$ represent the average cost of developing custom-tailored versions of components in the library that could have been used as is or adapted, respectively. Note that all these averages are time averages and not averages on individual components, i.e., a reusable component will be counted as many times as it is used.

Work on developing reusable software aims at maximizing p (probability of finding an exact match) and q (probability of finding an approximate match)—i.e., maximizing the coverage of the application domain—and minimizing $Adaptation_{old}$ for a set of common mismatches, i.e., packaging components in such a way that the most common mismatches are handled easily. Increasing p and q does not necessarily mean putting more components in the library; it could also mean putting components that are more frequently needed. Because adding components increases search costs (see Section III.B.1), we could use a two-pronged approach:

1) identify components that are generally useful and
2) try to cover the same set of needs with fewer components.

Identifying the components that are generally useful is sometimes called *domain analysis* and is an important activity for both application generator development (Section IV.B) and OO software development (Section IV.C). Covering the same set of needs using fewer components involves two paradigms:

1) *abstraction*, essential to application generators, and very important to OO software development and
2) *composition*, which is central to OO software development.

Composition supports the creation of a virtually unlimited number of aggregates from the same set of components and reduces the risk of combinatorial explosion that would result from enumerating all the possible configurations (cf. Section II.C and [149]). In general, the higher the level of abstraction at which composition takes place, the wider the range of systems (and behaviors) that can be obtained (see Section V.B). The combination of abstraction and composition provides a powerful paradigm for constructing systems from reusable components and constitutes the major thrust behind research in OO frameworks. It also exemplifies the ways in which software reuse addresses the scalability and focus issues in software engineering (see Section I).

Work on developing *with* reusable software aims at minimizing the cost of search (exact and approximate) and the cost of adaptation. Minimizing the cost of searches involves a number of trade-offs between the cost of formulating searches (*Search* and *ApproxSearch*) and the quality of the retrieval. For instance, the coverage probabilities p and q above should be replaced by smaller probabilities to take into account the less than perfect recall of search methods (see Section V.A). Further, a search method that is not precise (i.e., returns irrelevant components) increases the cost of finding a component by forcing the developer to examine irrelevant components. As a rule of thumb, given a fixed amount of effort to be spent on formulating queries, we can achieve higher recall values only at the expense of lower precision, and vice-versa. To enhance both recall and precision, more effort should be spent formulating queries. We discussed a range of approaches that strike different balances between query complexity and retrieval quality (c.f. Section V.A.3). However, there is an inherent practical limit to how complex queries can be, beyond which developers will not bother searching. As for adaptation, empirical evidence showed that the cost of modifying components in non-anticipated ways goes up very quickly with the scope and extent of the modifications [23]. The two major cost factors are:

1) understanding what changes need to be made and
2) verifying and validating the component after the change.

Transformational systems reduce the first cost by enabling developers to make changes directly at the requirements level and reduce the second cost by propagating such changes in a—mostly—correctness-preserving way [17] (cf. Section V.C).

How far can we reduce the cost of reuse occurrences? If we achieve full coverage ($p = 1$) and develop a query language that is perfectly precise and that has perfect recall—that is called a specification language!—then we have achieved wide-spectrum automatic programming! The generative approach and the building blocks approach to software reuse approach full coverage from two different, but complementary directions. The generative approaches often have a perfect coverage within a subarea of the application domain and need to be extended "horizontally" to cover the entire domain. In order to maintain performance characteristics (e.g., code optimality), different models/generators may be needed to cover a given domain. Conversely, the building blocks approach has the potential to cover an entire domain, but only sparsely so. To fill in the gaps, so to speak, abstract language constructs (e.g., module interconnection languages) are often added, yielding a coarse application-specific specification language whose atoms are *concrete* application components. As language constructs are added and increasingly abstract representations of components are used, we move progressively toward generative approaches based on very high-level languages [148]. Finally, it is interesting to note that in the context of the building blocks approach, perfect retrieval and effortless adaptation are only possible if the relation between specifications and implementations has been completely formalized. To some extent, software reuse turns the automatic programming problem into several optimization subproblems, allowing us to tackle software automation piecewise.

ACKNOWLEDGMENTS

Some of the material in Sections III.A.3 and IV.C benefited from discussions and joint work with Robert Godin (professor of computer science, University of Québec at Montréal), Gregor Bochmann (professor of computer science, University of Montréal), and Piero Colagrosso (Bell Northern Research).

Some of the background material for Section V.B was collected and compiled by Hassan Alaoui, a PhD student; the complexity results and algorithms for finding function realizations based on types are due to Odile Marcotte (professor of computer science, University of Québec at Montréal).

H. Mili was supported by grants from the Centre de Technologie Tandem de Montréal (CTTM), a division of Tandem Computers Inc., Cupertino, Calif, the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the Fonds pour la Création et l'Aide á la Recherche (FCAR) of Québec, and Centre de Recherche en Informatique de Montréal (CRIM) through the *MACROSCOPE* initiative and the Québec Ministry of Higher Education's *SYNERGIE* programme (IGLOO Project).

F. Mili was supported by grants from the National Science Foundation, and the School of Engineering and Computer Science, Oakland University.

A. Mili was supported by grants from NSERC and the School of Graduate Studies and Research of the University of Ottawa (Ottawa, Canada).

REFERENCES

[1] W.W. Agresti, "What are the new paradigms?," *New Paradigms for Software Development*. ed. W.W. Agresti, pp. 6-10, IEEE, 1986.

[2] William W. Agresti, "Framework for a flexible development process," *New Paradigms for Software Development*, William W. Agresti, ed., pp. 11-14, IEEE, 1986.

[3] M. Aksit, and L. Bergmans, "Obstacles in OO software development," *Proc. OOPSLA '92*, Vancouver, B.C., Canada, Oct. 18-22, 1992.

[4] G. Arango, Ira Baxter, P. Freeman and C. Pidgeon, "Software maintenance by transformation," *IEEE Software*, pp. 27-39, May 1986.

[5] G. Arango, "Domain engineering for software reuse," PhD thesis, Dept. Information and Computer Science, Univ. of California, 1988.

[6] S.P. Arnold and S.L. Stepoway, "The reuse system: Cataloguing and retrieval of reusable software," *Proc. COMPCON S'87*, IEEE CS Press, pp. 376-379. 1987.

[7] D.M. Balda and D.A. Gustafson, "Cost-estimation models for the reuse and prototype software development," *ACM SIGSOFT*, pp. 42-50, July 1990

[8] R. Balzer, T. Cheatham Jr., and C. Green, "Software technology in the 1990s: Using a new paradigm," *Computer*, Nov. 1983, pp. 39-45

[9] R. Balzer, "A 15 year perspective on automatic programming, *IEEE Trans. Software Engineering*, pp. 1,257-1,268, Nov. 1985.

[10] B. Barnes, T. Durek, J. Gaffney, and A. Pyster, "A framework and economic foundation for software reuse," *Proc. Workshop Software Reusability and Maintainability*, 1987.

[11] B.H. Barnes and T.B. Bollinger, "Making reuse cost-effective," *IEEE Software*, vol. 8, no. 1, pp. 13-24, Jan. 1991.

[12] V.R Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora, "The software engineering laboratory—an operational software experience," *Proc. 14th Int'l Conf. Software Engineering*, Melbourne, Australia, pp. 370-381, May 11-15, 1992.

[13] V.R. Basili and S. Green, "Software process evolution at the SEL," *IEEE Software*, pp 58-66, July 1994.

[14] V.R. Basili, "Viewing maintenance as reuse-oriented software development," *IEEE Software*, vol. 7, no. 1, pp. 19-25, Jan. 1990.

[15] V. R. Basili, and J.D. Musa, "The future engineering of software: A management perspective," *IEEE Computer*, , vol. 24, no. 9, pp. 90-96, Sept. 1991.

[16] P.G. Bassett, "Frame-based software engineering," *IEEE Software*, pp. 9-16, July 1987.

[17] I.D. Baxter, "Design maintenance dystems," *Comm. ACM*, vol. 35, no. 4, pp. 73-89, Apr. 1992.

[18] P. Bergstein and K.J. Lieberherr, "Incremental class dictionary learning and optimization," *Proc. ECOOP '91*, Geneva, Switzerland, pp. 377-395.

[19] T.J. Biggerstaff, and C. Richter, "Reusability framework, assessment, and directions," *IEEE Software*, pp. 41-49, July 1987.

[20] D. Blair and M.E. Maron, "An evaluation of retrieval effectiveness for a full-text," *Document-Retrieval System Comm. ACM*, vol. 28, no. 3, pp. 289-299, Mar. 1985.

[21] B. Boehm, "A spiral model of software development and enhancement." *Computer*, vol. 21, no. 5, pp. 61-72, May 1988.

[22] B. Boehm,, "Improving software productivity," *IEEE Software*, pp. 43-57, Sept. 1987.

[23] B.W. Boehm, "Megaprogramming," *Keynote speech, ACM Computer Science Conf.*, Phoenix, Ariz., Feb. 1994,

[24] R.J. Brachman and J.G. Schmolze, "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9, pp. 171-216, 1985.

[25] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams, "The GemStone data management system," *Object Oriented Concepts, Databases, and Applications*. pp. 283-308, W. Kim, ed., Addison Wesley 1989.

[26] F. Brooks, "No silver bullet: Essence and accidents of software engineering," *Computer*, , pp. 10-19, Apr. 1987.

[27] B.A. Burton, R.W. Aragon, S.A. Bailey, K.D. Koehler, and L.A. Mayes, "The reusable software library," *IEEE Software*, pp. 25-33, July 1987.

[28] G. Caldiera, and V.R. Basili, "Identifying and qualifying reusable software components," *Computer*, vol. 24, no. 2, pp. 61-70, Feb. 1991.

[29] J. Carbonell, "Derivational analogy in problem solving and knowledge acquisition," *Proc. Int'l Machine Learning Workshop*, Monticello, Ill., June 1983.

[30] D. Carrington, D. Duke, I. Hayes, and J. Welsh, "Deriving modular designs from formal specifications," *Software Engineering Notes, Proc. First ACM SIGSOFT Symp. Foundations of Software Engineering*, vol. 18, no. 5, pp. 89-98, Los Angeles, Calif., Dec. 7-10, 1993.

[31] D. de Champeaux, D. Lea, and P. Faure, "The process of OO design," *Proc. OOPSLA '92*, Vancouver, B.C., Canada Oct. 18-22, 1992

[32] P.S. Shicheng Chen, R. Hennicker, and M. Jarke, "On the retrieval of reusable components," *Selected Papers from the Second Int'l Workshop on Software, Reusability Advances in Software*, ReuseLucca, Italy Mar. 24-26, 1993.

[33] S.C. Cheung, and J. Kramer, "Enhancing compositional reachability analysis with context constraints," *Software Engineering Notes, Proc. First ACM SIGSOFT Symp. on the Foundations of Software Engineering*, vol. 18, pp. 115-125, Los Angeles, Calif., Dec. 7-10, 1993.

[34] C.T. Cleaveland, "Building application generators," *IEEE Software*, July 1988, pp. 25-33.

[35] P. Coad and E. Yourdon, *Object Oriented Analysis*. Prentice Hall, 1991, second edition.

[36] P. Coad, "RE: On the purpose of OO analysis," *Proc. OOPSLA'93*, Washington, D.C. Sept. 26 - Oct. 1, 1993.

[37] W.R. Cook, "Interfaces and specifications for the Smalltalk-80 collection classes," *Proc. OOPSLA'92*, 1992, Vancouver, B.C., Canada, Oct. 18-22.

[38] B.J. Cox, *Object Oriented Programming: An Evolutionary Approach*. Reading, Mass..: Addison Wesley, 1987.

[39] B.J. Cox, "Planning the software revolution," *IEEE Software*, vol. 7 , no. 6, pp. 25-35, Nov. 1990.

[40] M.A. Cusumano, "The software factory: A historical interpretation," *IEEE Software*, pp. 23-30, Mar. 1989.

[41] M.B. Dahl and K. Nygaard, "Simula common base language," Technical report S-22, Norvegian Computing Center, 1970.

[42] T. Davis, "The reuse capability model: A basis for improving an organization's reuse capability," *Advances in Software Reuse, Selected Papers from the Second Int'l Workshop on Software Reusability Advances in Software Reuse*, pp. 126-133, Lucca, Italy Mar. 24-26, 1993.

[43] L.P. Deutsch, "Design reuse and frameworks in the Smalltalk-80 programming system," *Software Reusability*, vol. II , A.J. Perlis, ed., ACM Press, 1989.

[44] E.W. Dijkstra, "On the cruelty of really teaching computer science," *Comm. ACM*, vol. 32, no. 12, pp. 1,398-1,404, Dec. 1989.

[45] M.S. Feather, "Reuse in the context of a transformation-based methodology," *ITT Proc. Workshop on Reusability in Programming*, pp. 50-58, 1983.

[46] R.G. Fichman and C.F. Kemerer, "Object oriented and conventional analysis and design methodologies: Comparison and critique," *Computer*, vol. 25, pp. 22-39, Oct. 1992.

[47] G. Fischer, "Cognitive view of reuse and design," *IEEE Software*, pp. 60-72, July 1987.

[48] W.B. Frakes and B. A. Nejmeh, "An information system for software reuse," *Software Reuse: Emerging Technology*, IEEE CS Press, 1990, pp. 142-151.

[49] W. B. Frakes and T. Pole, "An empirical study of representation methods for reusable software components," Tech. Report, Software Productivity Consortium, Herndon, Va. May, 1992.

[50] P. Freeman, "Reusable software engineering: Concepts and research directions," *Tutorial: Software Reusability*, P. Freeman, ed., pp. 10-23, 1987.

[51] R.P. Gabriel, "The failure of pattern languages," *J. Object Orientedriented Programming*, pp. 84-88, Feb. 1994.

[52] J. E. Gaffney and R.D. Cruickshank, "A general economics model of software reuse," *Proc. 14th Int'l Conf. Software Eng.*, pp. 327-337, ACM Press , Melbourne, Australia, May 11-15, 1992.

[53] K.B. Gallagher and J.R. Lyle, "Using program slicing in software maintenance, *IEEE Trans. Software Engineering*, vol. 17, no. 8, pp. 751-761, Aug. 1991.

[54] M. Garey and D Johnson, *Computers and Intractability*. San Francisco: Freeman, 1979.

[55] S. Gibbs, D Tsichritzis, E. Casais, O. Nierstrasz, and X. Pintado, "Class management for software communities," *Comm. ACM*, vol. 33, no. 9, pp. 90-103, 1990.

[56] R. Godin and H. Mili, "Building and maintaining analysis-level class hierarchies using galois lattices," *ACM SIGPLAN Notices, OOPSLA '93 Proc.*, vol. 28, pp. 394-410, Washington, D.C. Sept. 26 - Oct. 1, 1993,.

[57] J.A. Goguen, "Reusing and interconnecting software components," *Computer*, pp. 16-28, Feb. 1986,.

[58] A. Goldberg, "Information models, views, and controllers," *Dr. Dobb's*, July 1990.

[59] G. Gruman, "Early reuse practice lives up to its promise," *IEEE Software*, pp. 87-91, Nov. 1988.

[60] J.V. Guttag, J.J. Horning, and J.M. Wing, "An overview of the Larch family of specification languages," *IEEE Software*, vol. 2, no. 5, pp. 24-36, Sept. 1985.

[61] P. Hall and R. Weedon, "Object oriented module interconnection languages,"*Selected Papers from the Second Int'l Workshop on Software Reusability Advances in Software Reuse*, IEEE C S Press, pp. 29-38, Lucca, Italy, Mar. 24-26, 1993.

[62] R.J. Hall, "Generalized behavior-based retrieval," *Proc. 15th Int'l Conf. Software Eng.*, ACM Press, pp. 371-380, Baltimore, Md., May 17-21, 1993.

[63] W. Harrison and H. Ossher, "Subject-oriented programming: A critique of pure objects," *SIGPLAN Notices Proc. OOPSLA '93*, vol. 28, no. 10, pp. 411-428, Washington D.C., Sept. 26 - Oct. 1, 1993.

[64] B. Harvey, H. Kilov, and H. Mili, "Specification of behavioral semantics in OO information modeling: Workshop report," *OOPS Messenger Addendum to the OOPSLA '93 Proc.*, ACM Press.

[65] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying behavioral compositions in OO systems," *Proc. OOPSLA '90*, ACM Press, Ottawa, Canada, Oct. 22-25, 1990. ,

[66] B. Henderson-Sellers and J.M. Edwards, "The OO systems life cycle," *Comm. ACM*, vol. 33, no. 9, pp. 143-159, Sept. 1990.

[67] R.C. Holt, T. Stanhope, and G. Lausman, "Object oriented computing: Looking ahead to the year 2000," ITRC TR-9101, Apr. 1991, Information Technology Research Center, Univ. of Toronto.

[68] E. Horowitz and J.B. Munson, "An expansive view of reusable software," *IEEE Trans. Software Engineering*, vol. 10 , no. 5, pp. 477-487, 1984.

[69] E. Horowitz, A. Kemper, and B. Narasimhan, "A survey of application generators," *IEEE Software*, vol. 2, no. 1, Jan. 1985.

[70] G. M. Hoydalsvik and G. Sindre, "On the purpose of OO analysis," *Proc. OOPSLA '93*, ACM Press, pp. 240-255, Washington, D.C., Sept. 26-Oct. 1, 1993.

[71] J. C. Huang, "State constraints and pathwise decomposition of programs," *IEEE Trans. Software Engineering*, vol. 16 , no. 8, pp. 880-898, Aug. 1990.

[72] Y. Intrator and H. Mili, "Getting more out of your classes: Building families of programs in OOP," Tech report no. 234, Dept. Maths and Computer Science, Univ. of Quebec at Montreal, May 13, 1994.

[73] S. Isoda, "Experience report on a software reuse project: Its structure, activities, and statistical results," *Proc. 14th Int'l Conf. Software Engineering*, pp. 320-326, Melbourne, Australia, May 11-15, 1992.

[74] I. Jacobson, *Object Oriented Software Engineering: A Use Case Driven Approach*. ACP Press, 1992. ,

[75] R.E. Johnson, "Documenting frameworks using patterns," *Proc. OOPSLA '92* , ACM Press, pp. 63-76, Vancouver, B.C., Oct. 18-22, 1992.

[76] G. Jones, "Methodology/Environment Support for Reusability," Software Reuse: Emerging Technology, Will Tracz, ed., IEEE CS Press, pp. 190-193, 1990.

[77] T. Capers Jones, "Reusability in programming: A survey of the state of the art," *IEEE Trans. Software Engineering*, vol. 10, no. 5, pp. 488-494, Sept. 1984.

[78] G.E. Kaiser and D. Garlan, "Melding software systems from reusable building blocks," *IEEE Software*, pp. 17-24, July 1987.

[79] K.C. Kang, "A reuse-based software development methodology," *Software Reuse: Emerging Technology*, Will Tracz, ed. , pp. 194-196, IEEE CS Press 1990.

[80] S. Katz, C. Richter, and K.-S. The, "PARIS: A system for reusing partially interpreted schemas," *Proc. Ninth Int'l Conf. on Software Eng.*, pp. 377-385, 1987.

[81] G. Kiczales and J. Lamping, "Issues in the design and documentation of class libraries," *Proc. OOPSLA '92 SIGPLAN Notices*, ACM Press, vol. 27, no. 10, pp. 435-451, Vancouver, B.C, Oct. 18-22, 1992.

[82] P. Kruchten, E. Schonberg, and J. Schwartz, "Software prototyping using the SETL programming language," *IEEE Software*, vol. 1, no. 4, pp. 66-75, Oct. 1984.

[83] C.W. Krueger, "Software reuse," *ACM Computing Surveys*, ACM Press, vol. 24, no. 2, pp. 131-183, June 1992.

[84] S.S. Lam and A.U. Shankar, "Specifying modules to satisfy interfaces: A state transition system approach," *Distributed Computing*, vol. 6, pp. 39-63,1992.

[85] R.G. Lanergan and C.A. Grasso, "Software engineering with reusable designs and code," *IEEE Trans. Software Engineering*, vol. 10, no. 5, pp. 498-501, Sept. 1984.

[86] J. Laski and W. Szermer, "Regression analysis of reusable program components," *Selected Papers from the Second Int'l Workshop on Software Reusability Advances in Software Reuse*, IEEE CS Press, pp. 134-141, Lucca, Italy, Mar. 24-26, 1993.

[87] D. Lea, "Christopher Alexander: An introduction for OO designers," *Software Engineering Notes*, vol. 19, no. 1, pp. 39-46, Jan. 1994.

[88] D.B. Lenat, R.V. Guha, K. Pittman, D. Pratt, and M. Shepherd, "CYC: Toward programs with common sense," *Comm. ACM*, special issue on Natural Language Processing, vol. 33 , no. 8, pp. 30-49, Aug. 1990.

[89] L.S. Levy, "A metaprogramming method and its economic justification," *IEEE Trans. Software Engineering*, vol. 12, no. 2, pp. 272-277, Feb. 1986.

[90] M. Lubars, G. Meredith, C. Potts, and C. Richter, "Object oriented analysis for evolving systems," *Proc. 14th Int'l Conference on Software Engineering*, pp. 173-185, May 11-15, 1992 Melbourne, Australia, ACM Press.

[91] M.D. Lubars, "Wide-spectrum support for software reusability software reuse: Emerging technology," pp. 275-281, ed. W. Tracz, IEEE CS Press, 1990.

[92] Luqi, "A Graph Model for Software Evolution," *IEEE Trans.* Software Engineering, vol. 16 , no. 8, pp. 917-927, Aug. 1990.

[93] Y.S. Yoelle S. Maarek, D.M. Berry, and G.E. Kaiser, "An information retrieval approach for automatically constructing software libraries," *IEEE Trans. Software Engineering*, vol. 17, no. 8, pp. 800-813, Aug. 1991.

[94] N.A. Maiden and A.G. Sutcliffe, "Exploiting reusable specifications through analogy," Special issue on CASE, *Communications of the ACM*, vol. 35 no. 4, pp. 55-64, Apr. 1992.

[95] N.A. Maiden and A. Sutcliffe, "People-oriented software reuse: the very thought," *Proc. Second Int'l Workshop on Software Reuse, Computer*, pp. 176-185, PressLucca, Italy March 24-26, 1993.

[96] J.H. Maloney, A. Borning, and B.N. Freeman-Benson, "Constraint technology for user-interface construction in ThingLab II," pp. 381-388, *Proc. OOPSLA '89*, ACM Press, Oct. 1989.

[97] J. Margono and T.E. Rhoads, "Software reuse economics: cost-benefit analysis on a large-scale ada project," *Proc. 14th Int'l Conference on Software Engineering*, pp. 338-348, May 11-15, Melbourne, Australia.

[98] J Martin, "Fourth Generation Languages—Volume I: Principles," Prentice-Hall 1985.

[99] Y. Matsumoto, "A Software Factory: An Overall Approach to Software Production Tutorial: Software Reusability," pp. 155-178, ed. P. Freeman, IEEE Press, 1987.

[100]Y. Matsumoto, "Experiences from software reuse in industrial process control applications," *Selected Papers from the Second Int'l Workshop on Software Reusability Advances in Software Reuse*, pp. 186-195, Lucca, Italy, March 24-26, 1993 IEEE CS Press.

[101]R. McCain, "A software development methodology for reusable components *Proc. 18th Hawaii Conference on Systems* Sciences, Hawaii, Jan. 1985.

[102]D.B. McCarn, "MEDLINE: an introduction to on-line searching," *J. American Society for Information Science*, vol. 31 , no. 3, pp. 181-192, May 1980.

[103]C. McClure, "The three R's of software automation: re-engineering, repository, *Reusability,"* Prentice-Hall, 1992.

[104]D. McIlroy, "Mass produced software components," *Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, pp. 88-98, eds. J. M. Buxton, P. Naur, and B. Randell, Petrocelli/Charter, New York 1969.

[105]S. Mellor, "The Shlaer-Mellor Method," *Tutorial notes,* OOPSLA'93, Washington, D.C. Sept. 26 - Oct. 1, 1993 ACM Press.

[106]B. Meyer, "Object oriented software construction," ed. Prentice-Hall Int'l, 1988. ,

[107]B. Meyer, "Lessons from the design of the eiffel libraries," *Communications of the ACM,*. vol. 33, no. 9, pp. 69-88, Sept. 1990.

[108] A. Mili, R. Mili, and R. Mittermeir, "Storing and retrieving software components: a refinement-based approach," *Proc. 16th Int'l Conf. on Software Engineering*, Sorrento, Italy, May 1994.

[109]H. Mili, J. Sibert, and Y. Intrator, "An OO model based on relations," *J. Systems and Software*, vol. 12, pp. 139-155, 1990.

[110]H. Mili, A. E. El Wahidi, and Y. Intrator, "Building a graphical interface for an OO tool for software reuse," *Proc. TOOLS USA '92*, ed. B. Meyer, Aug. 2-6, 1992, Santa Barbara, Calif.

[111]H. Mili and H. Li, "Data abstraction in softclass, an OO case tool for software reuse," *Proc. TOOLS '93*, pp. 133-149, Santa-Barbara, CA Aug. 2-5, ed. B. Meyer, Prentice-Hall.

[112]H. Mili, O. Marcotte, and A. Kabbaj, "Intelligent component retrieval for software reuse," *Proc. 3rd Maghrebian Conf. on Artificial Intelligence, and Software Engineering*, pp. 101-114, Apr. 11-14, 1994, Rabat, Morocco.

[113]H. Mili, R. Rada, W. Wang, K. Strickland, C. Boldyreff, L. Olsen, J. Witt, J. Heger, W. Scherr, and P. Elzer, "Practitioner and SoftClass: A *Comparative Study of Two Software Reuse Research P*rojects," *J. Systems and Software*, vol. 27, May 1994.

[114]S.K. Misra and P.J. Jalics, "Third-generation versus fourth-generation development," *Software*, vol. 5, no. 4, pp. 8-14, July 1988.

[115]Th. Moineau, and M.C. Gaudel, "Software reusability through formal specifications, *Proc. 1st Int'l Workshop on Software Reusability*, Universitaet Dortmund 1991.

[116]J.M. Morel and Jean Faget, "The REBOOT environment," *Selected Papers from the 2nd Int'l Workshop on Software Reusability Advances in Software*, pp. 80-88, ReuseLucca, Italy, March 24-26, 1993, IEEE CS Press.

[117]M. Moriconi, and T.C. Winkler, "Approximate reasoning about the semantic effects of program changes," *IEEE Trans. Software Engineering*, vol. 16, no. 9, pp. 980-992, Sept. 1990.

[118]J. Mostow, M. Barley, "Automated reuse of design plans," *Proc. Int'l Conf. on Engineering* Design, Boston, MA 1987.

[119]B.A. Myers, "User-Interface Tools: Introduction and Survey," *Software*, pp. 15-23, Jan. 1989, Special issue on user interfaces.

[120]K. Narayanaswamy, W. Scacchi, "Maintaining configurations of evolving software systems," *IEEE Trans. Software Engineering*, vol. 13, no. 3, pp. 324-334, March 1987.

[121]J.M. Neighbors, "The DRACO approach to constructing software from reusable components," *IEEE Trans. Software Engineering*, pp. 564-574, Sept. 1984.

[122]H. Ossher and W. Harrison, "Combination of inheritance hierarchies," *SIGPLAN Notices, Proc. OOPSLA' 92.*, vol. 27 no. 10, pp. 25-40, Oct. 18-22, 1992, Vancouver, B.C., Canada,

[123]H. Partsch and R. Steinbruggen, "Program transformation systems," *Computing Surveys*, vol. 15 no. 3, pp. 399-236, Sept. 198.

[124]L. Peters, "Advanced Structured Analysis and Design," ed. R.W. Jensen, Prentice Hall, 1987.

[125]M. Pittman, "Lessons learned in managing OO development," *Software*, pp. 43-53, Jan. 1993.

[126]A. Podgurski and L.A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. Software Engineering*, vol. 16, no. 9, pp. 965-979, Sept. 1990.

[127]A. Podgurski and L. Pierce, "Retrieving reusable software by sampling behavior," *ACM Transactions Software Engineering and Methodology*, vol. 2 no. 3, pp. 286-303, July 1993.

[128]J.S. Poulin and J.M. Caruso, "A reuse metrics and return on investment model," *selected papers from the 2nd. int'l workshop on software reusability advances in software*, pp. 52-166, ReuseLucca, Italy, March 24-26, 1993, IEEE CS Press.

[129]R. Prieto-Diaz and J.M. Neighbors, "Module interconnection languages, *J. Systems and Software*, vol. 6, no. 4, pp. 307-334, Nov. 1986.

[130]R. Prieto-Diaz, "Domain analysis for reusability," *Proc. COMPSAC '87*, pp. 23-29, 1987, IEEE Press.

[131]R. Prieto-Diaz and P. Freeman, "Classifying software for reusability *Software*, pp. 6-16, Jan. 1987.

[132]R. Prieto-Diaz, "Integrating domain analysis and reuse in the software development process *Proc. 3rd Annual Workshop on Methods and Tools for Reuse*, CASE Center, Syracuse University, Syracuse, N.Y. June 13-15, 1990.

[133]R. Prieto-Diaz, "Status Report: Software Reusability," *Software*, vol. 10 no. 3, pp. 61-66, May 1993.

[134]R. Rada, H. Mili, E. Bicknell, and M. Blettner, "Development and application of a metric on semantic nets *IEEE Trans. Systems, Man, and Cybernetics*, Jan./Feb. 1989, vol. 19 , no. 1, pp. 7-30.

[135]C.V. Ramamoorthy, V. Garg, and A. Prakash, "Support for reusability in Genesis," *IEEE Trans. Software Engineering*, vol. 14 , no. 8, pp. 145-1154, Aug. 1988.

[136]C. Rich and R. Waters, "Automatic programming: myths and prospects *Computer*, pp. 40-51, Aug. 1988.

[137]C. Rich and R. Waters, "The programmer's apprentice: A research overview," *Computer*, pp. 1-25, Nov. 1988.

[138]K.S. Rubin A. Goldberg, "Object behavior analysis," *Communications of the ACM*, vol. 35, no. 9, pp. 48-62, Sept. 1992.

[139]J. Rumbaugh, "Relations as semantic constructs in an OO language," *Proc. OOPSLA '87*, pp. 466-481, Oct. 4-8, 1987, ACM Press.

[140]J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, "Object oriented modeling and design," Prentice Hall, 1991.

[141]D. Rumelhart and D. Norman, "Representation in memory," Center for Human Information Processing, La Jolla, Calif.

[142]G. Salton and M. McGill, "Introduction to Modern Information Retrieval," McGraw-Hill, New York, 1983.

[143]M. Shaw, "Abstraction techniques in modern programming languages," *Software*, pp. 10-26 Oct. 1984.

[144]M. Shaw, "Prospects for an Engineering Discipline of Software," *Software*, vol. 7 , no. 6, pp. 5-24, Nov. 1990.

[145]J.J. Shilling and P.F. Sweeny, "Three steps to views: Extending the OO paradigm," *Proc. ACM Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pp. 353-361, New Orleans, Louisiana, Oct. 1989, ACM.

[146]S. Shlaer and S. Mellor, "*Object Oriented* systems analysis: Modeling the world in data," Yourdon Press: Englewood Cliffs, N.J. 1992.

[147]B.G. Silverman, "Survey of expert critiquing systems: practical and theoretical frontiers," *Communications of the ACM*, vol. 35, no. 4, pp. 06-127, Apr. 1992.

[148]M.A. Simos, "The domain-oriented software life cycle: Toward an extended process model for reusability software reuse: emerging technology," pp. 354-363, ed. Will Tracz, IEEE CS Press, 1990.

[149]M. Sirkin, D. Batory, and V. Singhal, "Software components in a data structure precompiler, *Proc. 15th Int'l Conf. on Software Engineering*, pp. 437-446, Baltimore, Maryland, May 17-21, 1993, ACM Press.

[150]A. Snyder, "Encapsulation and inheritance in OO programming languages," *Proc. OOPSLA '86*, pp. 38-45, Sept. 1986, Portland, Oregon.

[151]L.S. Sorumgard, G. Sindre, and F. Stokke, "Experiences from application of a faceted classification scheme," *Selected papers from the 2nd Int'l Workshop on Software Reusability Advances in Software*, pp. 116-124, ReuseLucca, Italy, March 24-26, 1993, IEEE CS Press.

[152]B. Stroustrup, "The C++ programming languages," Addison-Wesley 1986.

[153]W. Tracz, "LILEANNA: A parameterized programming language," *Selected Papers from the 2nd Int'l Workshop on Software Reusability Advances in Software*, pp. 66-78, ReuseLucca, Italy, March 24-26, 1993 IEEE CS Press.

[154]J. Verner and G. Tate, "Estimating size and effort in fourth-generation development," *Software*, vol. 5, no. 4, pp. 5-22, July 1988.

[155]P. Wegner, "Varieties of reusability tutorial: Software reusability, ed. Peter Freeman , pp. 24-38, 1987.

[156]P. Wegner, "Dimensions of OO modeling, OO computing," *Computer*, CS Press, vol. 25 no. 10, pp. 2-20, Oct. 1992.

[157]R. Wirfs-Brock, B. Wilkerson, and L. Wiener, "Designing OO software," Prentice-Hall: 1990.

[158]R. Wirfs-Brock and R.E. Johnson, "Surveying current research in OO design," *Communications of the* ACM, vol. 33, no. 9, pp. 105-124. , Sept. 1990.

[159]S.N. Woodfield, D.W. Embley, and D.T. Scott, "Can programmers reuse software," *Software*, July 1987, pp. 52-59.

[160]E. Yourdon, "Decline & fall of the american programmer," Prentice-Hall: Englewood Cliffs, N.J, 1992.

[161]A.M. Zaremski and J.M. Wing, "Signature matching: A key to reuse software engineering, *Notes, 1st ACM SIGSOFT Symp. on the Foundations of Software Engineering*, vol. 18, pp. 582-190, 1993.

[162]P. Zave, W. Schell, "Salient features of an executable specification language and its environment," *IEEE Trans. Software Engineering*, vol. 12, no 2, pp. 312-325, Feb. 1986.

[163]P. Zave and M. Jackson, "Conjunction as composition," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 4, pp. 379-411, Oct. 1993.

Hafedh Milli is an associate professor of computer science at the University of Quebec in Motreal, Canada. He holds an engineering diploma from the Ecole Central de Paris, Paris, France, which was awarded in 1984, and a PhD in computer science from George Washington University, Washington, DC, which he earned in 1988.

His research interests include object orientation, software reuse, information retrieval, and knowledge representation. He has been leading or participating in a number of government- and industry-sponsored (BNR, IBM, DEC, Tandem Computers, CAE Electronics, DMR Group, National Bank of Canada, etc.) R&D projects in the area of object orientation and software reuse.

He is founder and president of INFORMILI, Inc., a computer services company that specializes in training and consulting in OO and software reuse.

Fatma Mili is an associate professor at Oakland University, Rochester, Michigan. Her research interests are in software engineering, formal methods, and scientific databases. She holds a doctorate degree from the University Pierre et Marie Curie in France.

Prof. Mili is a member of the ACM and the IEEE Computer Society.

Ali Mili holds a PhD from the University of Illinois at Urbana-Champaign (earned in 1981). He earned a doctorat es science d'etat from the University of Grenoble, France, in 1985.

His research interests are in software engineering, ranging from the technical to the managerial aspects of the discipline.

His latest book, coauthored by J. Desharnais and F. Mili, deals with the mathematics of program construction. It is published by Oxford University Press, New York.