SOFTWARE HOUSES

A "software house" is an organization that develops customized or bespoke software for a customer. This general definition, depending on the interpretation attached to the terms "customer" and "organization," spans the software industry spectrum from large, multinational software development concerns to the information technology departments of various government/industry organizations. From the outset, it should be noted that with the software industry's gradual maturity, the notions of software reuse and software house are inextricably intertwined. This is due largely to mass production and tight time-to-market that have become dominant factors in modern large-scale software production.

Conceptually, a software house in the data processing industry is a place where the entire range of software development activities are handled in a seamlessly integrated and centrally managed manner. Such activities of product development include: requirements analysis, requirements specification, design, implementation, phase-specific reviews, module and system testing, software quality assurance, software reliability modeling, cost/effort/schedule estimation, configuration management and version control, use of software tools, and the important and pervasive issue of the judicious reuse of various software artifacts as well as the potential use of existing domain-specific software packages and components.

The scope and structure of a software house also depends on the scale of software being developed. Software development can generally be divided into software development in-the-large and software development in-the-small. Most of the

techniques, methodologies, and tools of software engineering apply to software development in-the-large. Of course, we should keep in mind that there is no industry-wide consensus, nor is there a standard definition for "small" or "large" in this context.

The goal here is to consider situations as diverse as the following: a small company that accepts orders from customers to build software; a multi-million or multi-billion dollar company that builds software based on market analysis, customer response, and so on; and a relatively small data processing department in a company, whose primary product is something other than software, developing in-house software needed by the company. The differences are based on several factors including the nature and degree of interaction between customer(s)/developer(s) during development, whether the resulting software is generic or special purpose, the cost of software development, and the issues of upward compatibility and portability.

What makes an article on software houses different from a typical technical paper is the fact that, due to the nature of the topic, a discussion of software houses ought to be expository in nature. There is no research, per se, reported or research area, per se, conducted or funded under the rubric of software houses. While being speculative and providing a vision is certainly called for, the bulk of the work reported must necessarily be the repeatable and reliable state of the practice in the software industry. However, the notion of a software factory has received some attention from researchers. And, being tangentially related to the topic of software houses, it behooves us to mention the work on software factories here.

In his book on software factories, Johnson (1) discussed the issues involved in software development and maintenance from the perspective of the information technology practitioners and for the consumption of information technology managers. Rockwell and Gera (2) presented a conceptual reference model for software factories, which they defined as industrial software production. By focusing on the communication requirements of large-scale software facilities, their model provides a plausible way of describing existing systems and processes, as well as comparing and evaluating any proposed alternatives and enhancements. Their model integrates technical and organizational issues, while treating project management concerns in the context of long-term capital planning.

The rest of this article is organized as follows: The next section is the overview section that delineates the general position of software houses in the overall software development life cycle. The two main sections follow the overview section and discuss the technical and managerial aspects of software houses. The categorization of the relevant issues into technical and nontechnical may seem rather arbitrary at times. The intent, however, is not presenting a precise taxonomy. The goal is to paint as comprehensive a picture of the software houses as possible. The final section contains some concluding remarks dealing with general trends and directions with respect to software houses.

OVERVIEW

During the early years of software evolution (1950s to 1960s), application software was custom-designed and had relatively limited distribution (3). The second era (mid-1960s to late 1970s) was characterized by the use of product software (i.e., programs developed to be sold to one or more customers), thus leading to the advent of "software houses" that were charged with developing software for widespread distribution in a multidisciplinary market.

A software house has been defined as a company that offers both general and specialized software packages for sale to computer system owners, or a company that develops software for customers on a contractual basis (4). Another definition offered for a software house is an organization that develops customized software for a customer, as contrasted with a software publisher that develops and markets software packages (5).

Software houses are typically modeled after the software factory concept. A factory implies (a) industrial scale, (b) the ability to guarantee production quality, (c) budgeting, (d) scheduling, and (e) the use of capital investment to make the production process more cost effective (6). However, software consists of uniquely designed and complexly structured sets of assertions, instructions, and decisions, all of which must be negotiated, codified, analyzed for consistency, and validated for effectiveness in a constantly changing environment. Enterprises are discovering that their corporate survival depends increasingly on their ability to master the notion of a software factory and make the mysterious software process predictably manageable (7). Typically, what is needed to achieve such a goal is a well-understood generic production process, which would allow the management to interpret the daily events in terms of established targets and to respond to them accordingly, and a well-integrated set of technical tools to support the process.

In order to meet the ever increasing demand for software, modern software houses specialize in specific areas. The diversity of the target audience is enormous. Software houses can be found concentrating on diverse areas such as the apparel industry, hospitality fields, legal issues, debt recovery, medical data acquisition, and aircraft-guided missile systems.

Software houses, whether they are small or large, relatively modern or old, generally use off-the-shelf products to develop software. Alternatively, highly specialized software houses are unique in their development processes and environments, platforms, and market segments. Software houses can be found based in the United States, Europe, and elsewhere. They variously render consulting and/or software development services and resolve managerial/technical problems for production, services, or technology companies.

Software as a system undergoes a complete system development life cycle. For a software house, regardless of the particular process model utilized, this means involvement in activities and issues such as (a) understanding the customer's requirements and the scope of the product, (b) knowing how the customer's organization works and what other systems are being used there, (c) estimating the degree of change that the proposed product would bring and anticipate the possible resistance to that change, (d) estimating cost (human and computing resources) and quote reasonable prices and delivery dates, (e) handling liabilities and other legal issues, (f) reaching a contractual agreement that exactly defines what is to be delivered and when, (g) breaking the system down into components, (h) programming considerations such as the choice of an implementation language and the associated data

structures and algorithms, (i) decisions regarding the size and composition of the development teams, (j) the existence and adequacy of test plans ensuring the precise working of the system, (k) usability tests and user-friendliness issues, and (l) user interface design considerations.

TECHNICAL ASPECTS

The technical aspects to be discussed in this section are not generally the issues that one would encounter in a technical journal on the state of the art in a single area of software engineering. This is due to the encompassing nature of the topic of software houses. All the same, some of the relevant aspects include: software architecture, design patterns, software components, software classification, software reuse, component packaging, interface compatibility, component composition, quality of a software system as a function of the quality of its components, cost estimation for a software system based on the actual or estimated costs of its constituent components, and a theoretical model for software houses.

Among the above-mentioned topics, the prominent ones are: reuse, software components, and software classification. Three of the four subsections that make up the rest of this section take a more in-depth look at these three major issues that essentially define a software house. It should be noted that there is some inevitable overlap in the coverage. For instance, a discussion of software reuse will be incomplete without a reference to software components. The last subsection in this section addresses the rest of the major technical issues germane to software houses.

Reuse

Productivity, adaptability, simplicity, maintainability, and reliability are among the main issues concerning the construction of software in a software house. In today's marketplace, a competitive software house is one that can reduce product delivery time, increase the diversity of its products, enhance interoperability among its products, and conform to the standardization of software components. Reuse is an emerging practice in software development that favorably affects the above-mentioned issues.

Effective reuse practice has exhibited significant documented benefits, far more than other related ongoing activities addressing process improvement, in enhancing the software development process. Software reuse is the reapplication of a variety of components of existing systems to a new and similar system. Obviously, this is a definition for code reuse, which concerns only one of the possible reusable software artifacts. All products of the software development process, such as requirement analysis documents, architectural designs, detailed designs, planning documents, test plans, and test cases, can potentially to be reused (8).

A software house has in its core a production team consisting of those who engineer software and write code. Generally, software engineers/programmers are constantly called upon to produce more and more software, and thus their productivity continues to be an increasingly pressing problem. Recent surveys and papers on the reusability of software indicate that software reuse is a major way to boost software engineering productivity.

The result of one survey showed that 40% to 60% of all code is reusable from one application to another, 60% of the design and code on all business applications is reusable, and 75% of program functions is common to more than one program. The survey also indicated that only 15% of the code found in most programs is unique and novel to a specific application (8). Over the life cycle of a system, reuse can provide many advantages over the traditional approach of "designing from the ground up." Such advantages include:

1. Shortening development cycles and lowering production costs for future development efforts by reusing existing components.
2. Improving system reliability by reusing proven components and reducing the need for system testing.
3. Reducing life-cycle maintenance costs by reducing the "ripple" effect of changes caused by revised requirements.
4. Enabling a software house to partially recover its investment in existing software systems when developing new software products and undertaking new design efforts (9).

Reuse and Software Houses. Software components are the main commodity of a software house. Like any business, an inventory of the main commodity is considered one of the *assets.* Thus, a library of software components is one of the major assets of a software house. Regardless of whether a software house employs the services of a number of subcontractors, develops all software in-house, utilizes off-the-shelf components, or develops any hybrid mixture of the above, it must develop and maintain a library of components for current and future business.

Existence of business modeling and domain-specific systems is considered to be one of the prime factors in the success of attempts at initiating and nurturing reuse programs and reusable component library systems. About 85% of the reuse reported at IBM has come from domain-specific libraries (10). The selection of a specific appropriate domain, followed by performing domain analysis on that domain to develop specific domain modules, is one of the primary factors in the success of a reuse project. One of the major problems of the software factory initiative in Japan's NTT was the inappropriateness of the target domain (11). The REBOOT project, conducted under the auspices of a multinational consortium of software development firms in Europe, emphasizes the domain-oriented method as a "secure way" to produce reusable components (12).

Reuse Methodologies. There is no single, widely accepted definition for reuse that is applicable to all phases of the software development process (from specification to design to implementation documents) and all levels of program translation/compilation (from source code to assembly code to templates to object code, and everything else in between). In order to reap the benefits of reusability, a software house needs to concern itself with reuse at all levels and stages. Reuse has a distinct definition for each of the above-mentioned phases and levels. Moreover, the approaches and techniques applied to them can be quite different.

Generally, the complexity of the applicable reuse technique increases as it moves from the specification level to the code and object code levels. On the positive side, the time and space efficiency of the application of reuse techniques improves in the same direction. Specification and design levels are at a higher level of abstraction than the other reuse levels, and therefore their potential for accommodating reuse is greater and adaptation to new applications can be simpler. However, the reuse process of specification and design levels ultimately involves coding (either manually or system generated), testing, and debugging. On the other hand, reuse at the code level, especially in the form of black-box reuse, essentially eliminates recoding and unit testing; hence it is more economical, given that a large collection of reusable code components is organized in a software library.

Regardless of the level of reuse, the design of a successful reuse environment in a software house should provide answers to the questions of what is the unit of reuse (or, what are the units of reuse) and how to put the units together, and it also should address the following tasks:

1. Identifying and providing access to software components based on user requirements (locating).
2. Facilitating component modification or development (customization).
3. Providing facilities to store, retrieve, and integrate reusable components efficiently (configuration and version management) (13).

Component-Based Software Development. Component-based software engineering (CBSE) or component-based software development (CBD) is an emerging software development approach that has become the focal point of the reuse community. CBD has the potential to significantly impact how software houses conduct their business, and eventually to revolutionize the software industry. Since 1995, several models have been proposed by major software companies and software industry consortia as de facto standards for CBD (3).

CORBA (common object request broker) published by OMG (the Object Management Group) provides a number of services that enable objects (reusable components) to communicate with other objects in a system. OLE (object linking and embedding), which is part of COM (component object model) published by Microsoft, defines a standard structure for reusable components. Other proposed tools, models, and standards that facilitate CBD include ActiveX, Sun's JavaBeans, and Visual Basic.

As mentioned earlier, a successful CBD needs to be domain-specific. Furthermore, components should have standard interfaces and should be portable and interoperable across different applications within the same domain. The CBD process modifies the traditional software life cycle in the following respects (14):

1. Capturing the domain requirements: capturing domain information with focus on commonality and variability of the current, future, and potential requirements, users, and constraints.
2. Performing a robust analysis: identifying objects of analysis, and systematically presenting their commonality and variability.
3. Designing the component-based subsystems: adopting a design model that is suitable for the implementation environment.
4. Constructing and testing the component-based subsystem.
5. Packaging the component-based system according to the required configuration.

Reuse Obstacles. Lack of a clear, unified, and standard long-term strategy has hampered the full development, deployment, and general acceptance of software reuse (15). Absence of organizational commitment to reuse can compromise the efficacy and lessen the overall productivity of a software house. Traditional approaches to software development lack focus on (a) planned reuse, (b) system development from an integrated perspective (i.e., from an organizational perspective rather than from a single stand-alone application perspective), and (c) strategic long-term organizational advantage.

Having a stable and well-understood software asset base is crucial to achieving successful software reuse in a software house. Among nontechnical obstacles to reuse, a notable one is *the not invented here* (NIH) factor. Two of the other reuse hindrances are (a) lack of organization-wide incentives for development of reusable components and (b) licensing requirements and copyright laws. The list of technical obstacles includes, but is certainly not limited to, inadequate specification technology for reuse, lack of standard formal models, and scarcity of uniform design notations.

Software Components

A software house must deal with software components at various levels. A previous section on software reuse addressed the notion of software artifacts as components. The next section, which discusses classification and identification of software components, addresses the cataloging of software components. This section attempts to shed some light on the creation of software components (as a result of a reclamation process based on the dissection and decomposition of existing software systems) and the use of software components (through interfacing and composition).

Megaprogramming is the term commonly used in reference to construction and engineering of software systems from existing components, as contrasted with software development by coding one instruction at a time. The analogy is obviously industrial mass production techniques. The main goal is to reduce time-to-market and improve the reliability and maintainability of the final product. The economics of scale indicate, if not dictate, that megaprogramming is indeed the future of software houses and the software marketplace.

The software components industry, for which the term megaprogramming has been coined, requires the creation and existence of proven and well-defined components that are implemented according to software composition principles (16). The rest of this section presents a conceptual and formal framework for developing reusable software components that leverage the compositional capabilities of megaprogramming languages.

The megaprogramming enterprise model consists of (a) component production and component assembly governed by software architecture principles and standards and (b) a bro-

kerage that supervises the overall product line (by performing cost analysis, feasibility analysis, configuration management, etc.) and releases the product to the end users. Megaprogramming implies the adoption of a process model (e.g., document-based waterfall model or risk-based spiral model) that not only does not discourage reuse, but also promotes the black-box reuse approach (i.e., reuse with little or no modification).

A conceptual framework is defined that distinguishes among three aspects of software components (6):

1. The *concept* or abstraction that the component represents.
2. The *content* of the component or its implementation.
3. The *context* under which the component is defined (or what is needed to complete the definition of a concept or content within a certain environment).

The *concept* represented by a reusable software component is an abstract description of "what" the component does. Concepts are identified through requirements analysis or domain modeling as providing the desired functionality for some aspect of a system. A concept is realized by an interface specification and a description of the semantics associated with each operation. The *content* represented by a reusable software component is an implementation of the concept or "how" a component does "what" it is supposed to do. It assumes that each reusable software component may have several implementations that obey the semantics of its concept. The *context* represented by a reusable software component depends on understanding and expectations based on familiarity with previous implementations.

These three aspects of a software component make the following assumptions about their environment:

1. There is a problem space (specification domain) that can be decomposed into a set of concepts (or objects, if one prefers using an object-oriented paradigm).
2. There is a solution space that is characterized by the contents (implementations) of the concept.
3. The solution space is populated by several different implementations (or parameterized implementations) that can be instantiated by different contexts within the solution space.

With the purpose being the development of useful, adaptable, and reliable software modules with which to build new applications, the following three requirements (6) should be addressed by a component-centered model of a system:

1. Components must be *useful;* that is, they must meet the high-level requirements of at least one concept necessary to design and implement a new software application.
2. Components must be *adaptable;* that is, they must provide a mechanism such that modules can be easily tailored to the unique requirements of an application.
3. Components must be *reliable;* that is, they must accurately implement the concept that they define.

Each component is basically made up of code plus interface specifications. The problem of code development is generally more tractable than the problem of specifying precise, unambiguous, and generalizable interface specifications. The software industry is in the process of developing the requisite technologies to define a formalism for interfaces, so that software components could interoperate smoothly.

Component composition in the megaprogramming technology has inherent risks rooted in the causes of component integration failures. Such causes include incompletenesses and inconsistencies involving data, control, timing, and implicit assumption.

Classification and Identification of Software Components

The capability to classify and store as well as to identify and locate software components is an important activity in a software house. Classification schemes are essential for setting up and maintaining a software library. A software library is a changing and growing collection of all of the modules that have been certified as reusable components. In order to be able to catalog and subsequently access those components, it is preferable that they be organized by attributes that define software structure, environment, function, implementation, and the like (13).

A different approach in classification and identification of software components is the application of abstraction methodologies. Abstraction has been applied extensively to help manage the intellectual and conceptual complexity of the software development process. Abstraction plays a focal role in the selection, customization, and integration phases of constructing a software system utilizing software components that are stored in a component repository.

Classification Principle. A classification system for software components is built based on a classification principle or schema. According to Prieto-Diaz (17): "Classification is grouping like things together. All members of a group, or class, produced by classification share at least one characteristic that members of other classes do not. Classification displays the relationships among things and among classes of things." A classification schema is a tool to produce systematic order based on a controlled and structured index vocabulary (13). A classification schema must be capable of expressing *hierarchical* and *synthetical* relationships. Hierarchical relationships are those that express subordination or inclusion relationships. The synthetical relationships are those relationships that are made to relate two or more ideas belonging to two or more hierarchies. Classification schemas are typically hierarchical with synthetical classification depicted as compound classes.

A classification schema can be arranged in two principal ways: enumerative and faceted. The hierarchical enumerative method recursively divides knowledge into subclasses until it covers all possible compound classes. A typical example of enumerative hierarchy is the Dewey decimal classification used in the classification of subjects in Library Science (18). The synthetical faceted method builds up relations from subject statements of documents. This type of relation is synthesized from two or more concepts that exist in different hierarchies. In the faceted method, the elemental component classes of subject statements are extracted, listed, and stored, and their generic relationships are displayed.

A classifier using the faceted schema has to represent a desired subject in the assembled form of elemental classes (a compound class). This process is called *synthesis,* the organized group of elemental classes are called *facets,* and the members/items of the facets are called *terms.* Facets within a faceted scheme are ranked by citation order corresponding to their significance to the user requirements. Therefore, when classifying, the most relevant term in a classification description is selected from the facet most relevant to the user (18).

With the enumeration schema, classes are typically prepared for a user. While the user or classifier of a faceted schema must synthesize the multielement classes. This feature of a faceted schema makes it easier to expand, thus making it more flexible, precise, and suitable for dynamic and expandable environments as compared to an enumerated schema (13).

Software Classification. Software components can be described by their function, procedure, and implementation details, among other things. Prieto-Diaz and Freeman (13) suggested that a characterization of the functionality (what it does) and the environment (where it does it) of a software component would suffice for classification. Burton and Aragon (19) used algorithm description, documentation, testing, and version management plus functionality and environment as classification attributes. Prieto-Diaz and Freeman (13) suggested the following attributes for faceted classification: function, object, medium, system type, functional area, and setting.

The Prieto-Diaz and Freeman classification method actually employs a *controlled vocabulary* technique for indexing software. They have used this technique to avoid duplicate and ambiguous descriptors of software components. For example, a software component described as

⟨`move, words, file`⟩

could also be described as

⟨`transfer, names, file`⟩

Describing code using controlled vocabulary is not problem prone for an audience that is not composed of information specialists. A term thesaurus could be used to gather all synonyms under a single concept, and one term that expresses the concept best would be the representative term (17).

Conceptual Closeness. When dealing with a faceted classification system, the problem of where to insert a new component presents itself. This is a problem about the attributes used to characterize a software component. To decide which terms are closer to each other, the idea of a conceptual graph, to measure closeness among terms in a facet, can be used (17). A conceptual graph is defined as an acyclic directed graph in which the leaves are terms and the nodes are considered as supertypes. Supertypes represent general concepts relating two or more terms. The weights of the edges are assigned by the user. Smaller weights represent the closeness of the terms to the supertype.

The concept of closeness measurement could be utilized during retrieval. In cases where query for a term cannot match any descriptor, a retrieval system can check the nearby terms for related items. It is time-consuming to construct a conceptual graph with more than few terms. However, the basic graph structure doesn't change much during the expansion of the collection of software components, and it tends to remain stable. Conceptual graph construction can be considered a substantial but one-time effort. Regardless, once constructed, a conceptual graph would need tuning as users provide feedback on retrieval performance.

Domain Analysis. To make the faceted classification scheme a more efficient method for a software house, the *domain analysis* methodology is recommended. This section provides an introduction to domain analysis and its application to classification and software reuse. According to Arango: "Domain analysis is a knowledge intensive activity for which no methodology or any kind of formalism is yet available" (20).

Domain analysis is an activity that happens even before the system analysis phase of the software development life cycle, and it creates a domain model to support the system analysis. This information/model could be used in the subsequent phases of the software development process. In the domain analysis process, "information used in developing a software system is identified, captured, and organized with the purpose of making it reusable when creating a new system" (17). Domain analysis could play an active role in the creation and organization of a software factory. Matsumoto (21) reported the successful application of domain analysis in the development of software factories.

The domain analysis process can be incorporated into the software development process. A simplified three-step domain analysis procedure to advance reuse is as follows:

1. Identification of reusable entities
2. Abstraction or generalization of those entities
3. Classification and cataloging for further reuse

Based on the above procedure, Prieto-Diaz (17) proposed a procedural model for domain analysis. Using the faceted classification schemes, his methodology is "to create and structure a controlled vocabulary that is standard not only for classifying but also for describing titles in a domain-specific collection" (17).

In the context of domain analysis, Arango (20) sees reuse as a learning system. In his proposed model, software development is a self-improving process which draws from a knowledge source that is named *reuse infrastructure,* and it is integrated with the software development process. Reuse infrastructure consists of domain-specific reusable resources (i.e., components in particular and assets in general) and their description. In Arango's reuse environment, by employing the reuse infrastructure and utilizing the specification of the software to be built, an implementation of the desired software is constructed. Then, the software thus produced is compared against the input of the system (i.e., the specification of the system).

There are three particular functions that are crucial for reuse infrastructure. These functions (17) are the abstractions of the duties of:

1. A librarian (making assets accessible to potential users)
2. An asset manager (controlling asset quality)

3. A reuse manager (facilitating the collection of domain analysis relevant data and coordinating all reuse operations)

Assets are those entities (documents, deliverables, and components) in a software development life cycle that are potentially reusable.

The typical process resulting from the integration of conventional software development and domain analysis is as follows:

1. Reusable resources are identified and added to the system.
2. Reuse data are gathered and fed back to the domain analysis process for tuning the domain models and updating the resource library.

The newly developed system can then be used to refine the reuse infrastructure (17).

Other Technical Issues

In this subsection some of the rest of the major technical aspects of software houses are briefly examined.

Quality Assurance. Quality assurance is concerned with checking both product and process quality (22). Software quality is more than verification and validation, especially as far as software houses are concerned. Quality assurance encompasses software attributes such as maintainability, reliability, and portability. A quality assurance plan should explicitly identify the quality attributes that are most significant for a particular project, and it should provide principles and guidelines as to how these attributes can be judged.

Some software houses have quality assurance departments that are responsible for the end-to-end quality of the internal documents and deliverables as well as of the final documents and deliverables. Other software houses address the pervasive issue of quality in the context of total quality management (TQM) (3). TQM is basically a multistep process that involves the function of quality auditing through the use of metrics and/or reviews.

Cost/Effort Estimation. The most visible undesirable aspects of software development in general and software houses in particular have been notorious schedule slippages and cost overruns (22). The explanations or justifications that are offered include incorrect or inaccurate design, frequent and undocumented changes, dynamically changing run-time environment, unexpected technological changes, personnel turnover, and of course imperfect estimation tools. Having reasonably accurate estimates of the interrelated parameters of person-month, cost, and schedule are critical for coming up with a systematic and structured approach to software development and maintenance as well as for the nontechnical issues of time-to-market, public image of the software house, and so on.

Software Architecture. Focus on software architecture and design patterns is a relatively new trend in institutional software development (23,24). Developing software based on reusing existing software architectural primitives is gradually gaining popularity. Although there are no widely used software development models or environments with software architecture as an integral part yet, there is little doubt about the increasing recognition of the importance of software architecture in developing software in software houses.

Strictly speaking, abstractions of recurring patterns in software design are called design patterns or frameworks; whereas software architecture, algorithms, and data structures constitute the design phase of the conventional software development life cycle. However, the two notions are sufficiently related to be treated together. The working definitions that have been offered in the literature for software architecture and patterns include the following:

1. An abstraction of information about components and connectors.
2. Structural issues, which are part of the design phase in the conventional life cycle, including:
   2.1. organization of a system as a composition of components,
   2.2. global control structures,
   2.3. protocols for communication, synchronization, and data access, and
   2.4. assignment of functionality to design elements.
3. A solution to a recurring problem in a specific context or environment.
4. Definition of a system in terms of computational components and interactions among those components.

Some of the examples of software architectures and patterns are: pipes and filters, abstract data types and object-oriented inheritance hierarchies, event-based broadcast/response systems and implicit invocations, layered and monolithic systems, repository-based development, table-driven applications, and state changes and state machine models. The current focus of workers in the general area of software architecture is on evaluation of existing architectures using metrics, methods of specification of new architectures using architecture description languages, extraction of the architecture of existing systems for reverse engineering, and visualization of architectures.

MANAGEMENT ASPECTS

It is a widely expressed concern that the present approaches, techniques, and methodologies of constructing and acquiring large and complex customized or bespoke software-based systems are unsatisfactory. The reason for this concern is that the resulting systems never fully meet, and probably never can meet, the requirements of the users when they come on line. Improving this situation would have significant implications for both the technical and management/commercial aspects of software development. A useful approach for meeting the challenge would be to establish a set of guidelines (25) to assist a software house in developing software.

The nontechnical or management aspects of software houses include legal issues, commercial aspects, training approaches and techniques, customer support, TQM and the attendant notion of quality assurance, and the issue of standards for various software products, deliverables, and

processes. Understandably, some of the management issues overlap with the technical issues, a good example being quality assurance. The difference is typically based on the perspective that can include measurement, enforcement, and interpretation.

The following two sections contain a discussion of the most significant management issues of software houses. The first section addresses the general area of legal concerns in dealing with software development. The second section briefly discusses the issue of process assessment and improvement.

Legal Aspects of Software Development

With the proliferation of computers, the market for software has grown exponentially. The capital investment in software and software houses, together with the fierce competition for segments of the huge software market, has inescapably resulted in a number of legal entanglements. It is a fact that there is a plethora of legal issues that are directly or indirectly related to software construction and software houses. However, the goal here is to keep the discussion as close to the technical issues as possible.

One of the major technical aspects of software houses that can significantly impact productivity is software reusability. In the software engineering community, many perceive that legal issues surrounding systematic reuse can discourage or even prohibit software reuse (26). Although legal issues are less of an impediment to software development using in-house software components, there are significant legal considerations for reuse among organizations.

Legal Protections for Software. Current legislation does not consist of dedicated laws that address software and software reuse specifically and explicitly (12). However, there are legal protections for software that are adapted from intellectual property law. Intellectual property is any product of the human thought processes that has some intellectual, informational, or economic value (26,27). Although ideas, in and of themselves, are not protectable as intellectual property, their expression or embodiment in a tangible object will afford them some form of intellectual property protection (28). There are four basic forms of intellectual property protection in the United States: patent law, copyright law, trade secret law, and trademark law (29).

A patent protects "novel and nonobvious" inventions and gives the inventor exclusive rights to make, use, or sell the invention. A copyright is a legal device that provides the author of a literary work the right to control how that work is used. Trade secret laws protect information about some intellectual property interest that is maintained as a proprietary secret and can potentially provides the owner with a commercial advantage. Trademark laws protect original names, words, phrases, logos, or other symbols that are used to distinguish products in the marketplace (26,29,27).

An overview of the different types of intellectual property law is provided below. Since trademarks are primarily marketing tools, trademark law will not be explored further in this subsection.

Patents. A patent for an invention gives the patent owner a statutory monopoly on the invention. Upon issuance of a patent for an invention, the patent owner has the exclusive right to make, use, license, or sell the invention for 17 years. The laws explicitly prohibit unauthorized copying of the expression of the idea or unauthorized use of any product that contains the invention. Patent laws protect the idea, as well as the invention itself, against independent creation. The invention must satisfy the four requirements of patentability before a patent is granted: novelty, utility, disclosure, and nonobviousness.

Although patent law provides the ultimate protection for intellectual property, there are several reasons that the software industry has not adopted the patent law as the primary means of protecting software. The cost, in terms of both money and time, has steered many away from this form of protection. The patent examination process, during which the patent examiner evaluates the invention against the four patent conditions, is a lengthy one. It can take between 18 months and 3 years (sometimes more) to obtain a determination (27). In today's fast-moving software world, the length of the examination process is typically longer than the market life of most software products. Perhaps the biggest inhibitor to software patent protection is that most programs do not qualify for a patent because they are unable to satisfy the nonobviousness requirement (27).

The unpatentability of most computer programs, as well as the long lead times and expense associated with the patent examination process, have compelled software developers to look for alternative means to protect their software. The copyright system provides another legal mechanism for protecting software.

Copyrights. A copyright is a legal means for protecting software from unauthorized copying or misappropriation. A copyright provides the author of a protected work a set of exclusive rights to the expression of ideas within that work. A copyright owner has the following exclusive rights to his/her work:

1. To reproduce the work.
2. To prepare derivative works.
3. To distribute copies of the work.
4. To perform the work publicly.
5. To display the work publicly.

Unlike patent law, the copyright laws do not protect the ideas contained within a protected work. Rather, they merely the expression of those ideas, and a copyright does not protect an author from independent creation of the same expression from another author. In fact, a copyright does not extend to any idea, procedure, process, system, method of operation, concept, principle, or discovery.

There are three prerequisites to full copyright protection, as outlined below:

1. *Fixation.* The work must be fixed in some tangible medium of expression or representation; the embodiment of the work must be sufficiently permanent or stable so that it can be perceived, copied, or otherwise communicated.
2. *Originality.* The work must be an original work of the author; that is, the work must have been created independently.
3. *Creativity.* A minimal amount of creativity is required for a copyright.

Once these three criteria are satisfied, copyright protection is automatic for that work. The term of this protection lasts for the lifetime of the author plus 50 years; for works made for hire, the term is 75 years from publication or 100 years from creation, whichever is shorter.

Note that a copyright notice is not required to obtain protection. Although copyright registration is inexpensive ($20) and is granted with little examination, a copyright owner does not have to file a copyright application with the Copyright Office. The ease of obtaining copyright protection for software provides a distinct advantage over patent protection.

Trade Secrets. A trade secret is any formula, pattern, device, or compilation of information that provides a competitive advantage over one's competitors in business. Trade secrecy laws give the holder of a trade secret only one exclusive right: Others may not obtain the secret through unlawful or improper means. In other words, trade secrets are protected only against unauthorized appropriation through unfair practices.

Although trade secret protection is provided under state laws rather than federal law, the basic elements of trade secrecy do not vary significantly from state to state. There are three requisite conditions that must be satisfied for a trade secret to be afforded legal protection: novelty, secrecy, and value. Trade secret protection is extended as long as the information is kept secret. Accidental, negligent, or intentional disclosure is one way to lose trade secret protection. This form of protection does not prevent or prohibit others from learning the secret through fair means, such as independent discovery or lawful reverse engineering.

Analogous to the copyright system, this form of protection offers some advantage over patent protection. The legal costs associated with obtaining a patent are much higher; the only significant costs associated with trade secrecy are the expenses of establishing appropriate external and internal controls to protect the secret, such as execution of nondisclosure agreements and distribution or licensing agreements (28). Trade secrecy can be used in conjunction with copyright. Software and related artifacts can be afforded both forms of protection at the same time. This combination may provide maximum protection possible under current intellectual property law.

Impact on Software Development. Anyone who is associated with a software development project, especially if it is within the context of a software house, needs to be aware of the various forms of intellectual property protection available for software. The developer should know whether his/her work qualifies for protection. More importantly, the developer needs the ability to determine whether he or she is infringing upon someone else's patent or copyright and to determine whether his or her actions are violating trade secret laws. Companies, employers, and managers of software development teams also need to be aware of the legal issues related to intellectual property as it applies to software. By law, an employer can be held liable for infringement, even if he or she is unaware that the infringement took place (28).

Legal Issues on Software Development for Reuse. In this subsection the focus is on legal considerations of developing software systems from software components in a software repository. The various phases of development (analysis, design, implementation, and testing) are considered individually from the legal perspective.

Analysis Phase. Obtaining domain knowledge from a repository provides the development team an introduction to the problem domain and provides a basis for determining the customer's requirements (12). The use of domain knowledge poses no legal barrier to the developer, unless that knowledge is protected as a trade secret. Regardless of the product or software development phase in which that product is used, if trade secret protection is extended to the information obtained from a reuse library, the software development team must ensure that adequate precautions are taken to prevent unauthorized disclosure.

Although copyright protection may be extended to the form or expression of the domain knowledge, the information itself (i.e., the facts about the domain) is not protected. The use of knowledge available to the general public does not violate copyright law. In fact, the copyright system is designed to encourage the use and advancement of such knowledge. The reuse of functional and nonfunctional requirements can provide significant savings in development costs. Reusing functional requirements has the same legal interpretation as reusing domain knowledge (30).

Design Phase. The purpose of this phase is to identify how the new application or system will satisfy the functional and nonfunctional requirements specified during the analysis phase. This phase can be divided into two major subphases:

1. Architectural design, where the system is described in terms of parts of the whole.
2. Detailed design, where each of the system parts is described in full.

Both of the subphases require some legal consideration. Based on the results of *Apple v. Microsoft,* it appears that reusing standard architectures and frameworks is relatively safe in terms of legal liability. Because the goal of implementing standards is to move toward a standard, any use or "copying" of a standard architecture or framework is relatively risk free.

The detailed design phase may not be as free of risk. In *Whelan v. Jaslow,* the court defined the expression of a computer program to be its "structure, sequence, and organization" (31). The expression and, therefore, the detailed design specifications can be afforded copyright protection. In reusing design descriptions, it is recommended that the development team obtain permission to copy or adapt the components.

Implementation Phase. In the implementation phase of the software development process, the detailed design obtained from the previous phase is translated into source code. The source code, some of which is directly reused or adapted from reusable components, is integrated to form a new, complete, and fully functioning product. The reuse of source code warrants special legal attention. Whenever source code is reused or adapted, permission to do so must be obtained from the appropriate copyright holder. This must be done even for test versions or prototypes that are never intended to be sold.

If a software development team opts to reuse a design but not necessarily all the source components that were created as a result of the original design, the outcome of the afore-

mentioned case indicates that this is legally accepted by the courts (31). During this phase, the developers should ensure that appropriate notices are embedded in the new application. Upon the initiation of the new application, a message containing intellectual property information should be displayed. In addition, the team should ensure that this information is included with each distinct component of the system. Although comment fields are an excellent way to document intellectual property information for each module, this information is lost once the module is converted to object code.

Testing Phase. The purpose of the testing phase is to ensure that the new application or system functions correctly. Although testing is an integral part of the entire system life cycle, the focus of the testing is to determine if the system satisfies its specifications—that is, all of the functional and nonfunctional requirements. The reuse of test components (i.e., test plans, scenarios, etc.) in the testing phase may require some attention.

The components that are reused in the testing phase should be treated much like the reuse of components in the analysis phase of software development. The reuse of test cases, test data, and test environments does not pose a legal risk to the developer. Analogous to the use of domain knowledge and functional requirements, protection may apply to the form of this information but not to the information itself. When a software development team reuses test documents, the team must ensure that appropriate actions are taken in the creation and publication of new test documents. These documents should be handled as any other traditional literary work, such as a novel or screenplay.

Process Assessment and Improvement

For a software house to be successful in the market, it is essential that it should deliver quality software products that fulfill the combined expectations of the customer. This is particularly true for smaller software enterprises. It is reasonable to assume that the quality of a complex software product can only be predicated on having a mature software development process. The quality of software development process in an organization must be assessed, preferably regularly, to determine the capability of the organization and to initiate process improvement, if necessary.

For assessment of standalone software houses or software houses that are part of a larger parent organization, several methodologies have been have been proposed. Examples include the CMM (capability maturity model), BOOTSTRAP, and the new standard ISO 15504 formerly called SPICE (software process improvement capability determination). For small enterprises, however, sophisticated and high-priced assessment methodologies that depend on several external consultants are not entirely viable. A more economically and technically feasible approach would be to conduct initial self-evaluations and periodic follow-up structured reviews and interviews (32).

While software process improvement is the main goal of various software development assessment approaches, a rank-ordering of software houses can have other uses as well. For instance, a large-scale software development contract can be awarded based on such a rank ordering. The experiences of a government team in determining the winner of a major software contract using the Software Engineering Institute's software capability evaluation program using the CMM has been reported in the open literature (33). The contractor-selection process can have as an integral part a number of on-site and post-on-site evaluation activities.

CONCLUDING REMARKS

There are a number of emerging trends and technologies for software houses. Generally, these trends and techniques are not innovative and revolutionary, but instead they have simply become mature enough to be incorporated into the methodical and systematic software development processes utilized in a software house.

The most important innovation is concurrent engineering, which can be interpreted as both (a) software development in parallel (extraction of the parallelism inherent in the specification, design, and implementation, and exploiting it to streamline and expedite the development process) and (b) parallel software development (developing software for parallel platforms). The latter obviously offers more challenges since it essentially encompasses the former. Software engineering for parallel and distributed systems generally deals with the identification of problem-domain and solution-domain parallelism as well as the optimal utilization of the concurrency in the specification and design phases.

The software development process in a software house can be enhanced not only by effectively utilizing the concurrency inherent in a software problem/solution and the platform, but also by utilizing the internet/intranet network services via the use of object-oriented design techniques (such as design patterns, information hiding, and layered modularity) and object-oriented language features (such as inheritance, parameterized types, templates, abstract classes, and dynamic binding).

BIBLIOGRAPHY

1. J. R. Johnson, *The Software Factory: Managing Software Development and Maintenance,* Wellesley, MA: QED Information Sciences, 1989.
2. R. Rockwell and M. H. Gera, The Eureka software factory core: A conceptual reference model for software factories, *Proc. Softw. Eng. Environ. Conf.,* Reading, UK, July 1993, pp. 80–93.
3. R. S. Pressman, *Software Engineering: A Practitioner's Approach,* 4th ed., New York: McGraw-Hill, 1997.
4. *Webster's New World Dictionary of Computer Terms,* 3rd ed., New York: Webster's New World, 1988.
5. A. Freeman, *The Computer Glossary: The Complete Illustrated Desk Reference,* 5th ed., New York: AMACOM American Management Association, 1991.
6. J. J. Marciniak, *Encyclopedia of Software Engineering,* New York: Wiley, 1994.
7. B. J. Cox, Planning the software industrial revolution, *IEEE Softw.,* **7** (6): 25–33, 1990.
8. W. J. Tracz, Software reuse: Motivators and inhibitors, *Proc. COMPCON87,* San Francisco, CA, 1987, pp. 358–363.
9. C. W. Krueger, Software reuse, *ACM Comput. Surv.,* **24** (2): 131–183, 1992.
10. J. S. Poulin, Populating software repositories: incentives and domain-specific software, *J. Syst. Softw.,* **30** (3): 187–199, 1995.

11. S. Isoda, Experience report on software reuse projects: its structure, activities, and statistical results, *Proc. Int. Conf. Softw. Eng.,* Melbourne, Australia, 1992, pp. 320–326.
12. E. Karlsson (ed.), *Software Reuse: A Holistic Approach,* New York: Wiley, 1995.
13. R. Prieto-Diaz and P. Freeman, Classifying software for reusability, *IEEE Softw.,* **4** (1): 6–16, 1987.
14. I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse,* New York: ACM Press and Addison-Wesley, 1997.
15. M. K. Zand and M. H. Samadzadeh, Software reuse: Current status and trends, Invited Editorial, *J. Syst. Softw.,* **30** (3): 167–170, 1995.
16. M. D. McIlroy, Software components, *IEEE Softw.,* **1** (4): 2–23, 1984.
17. R. Prieto-Diaz, Classification of reusable modules, in T. Biggerstaff and A. Perlis (eds.), *Software Reusability: Concepts and Models,* Vol. I, New York: ACM Press, 1989, pp. 99–123.
18. B. Buchanan, *Theory of Library Classification,* London: Bingley, 1979.
19. B. A. Burton and R. W. Aragon, The reusable software library, *IEEE Softw.,* **4** (4): 25–33, 1987.
20. G. Arango, Domain engineering for software reuse, Ph.D. thesis, Comput. Sci. Dept., Univ. of California, Irvine, CA, 1988.
21. Y. Matsumoto, A software factory: An overall approach to software production, in P. Freeman (ed.), *IEEE Tutorial on Software Reusability,* Los Alamitos, CA: IEEE Computer Society Press, 1987, pp. 155–178.
22. I. Sommerville, *Software Engineering,* 5th ed., Reading, MA: Addison-Wesley, 1996.
23. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline,* Upper Saddle River, NJ: Prentice-Hall, 1996.
24. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley, 1995.
25. A. Kemp, Software procurement and superconcurrent engineering, *Comput. Control Eng. J.,* **5**: 299–303, 1994.
26. T. R. Huber, Reducing business and legal risks in software reuse libraries, *Proc. 3rd Int. Conf. Softw. Reuse: Adv. Softw. Reusability,* Los Alamitos, CA: IEEE Computer Society Press, 1994.
27. S. Fishman, *Copyright Your Software,* Berkeley, CA: Nolo Press, 1994.
28. F. L. Cooper III, *Law and the Software Marketer,* Englewood Cliffs, NJ: Prentice-Hall, 1988.
29. I. H. Donner, Intellectual property protection for multimedia applications (part 1): So many flavors, so little time, *IEEE Comput.,* **28** (7): 92–93, 1995.
30. N. Carr and M. K. Zand, Legal aspects of software development with reuse, *Proc. 12th Int. Conf. Comput. Appl.,* Tempe, AZ, March 1997.
31. J. Drezel, *What Is Protected in a Computer Program? Copyright Protection in the United States and Europe,* Weinheim, Germany: VCH, 1994.
32. P. Grunbacher, A software assessment process for small software enterprises, *Proc. 23rd EUROMICRO Conf. (EUROMICRO97): New Front. Inf. Technol.,* Budapest, Hungary, 1997, pp. 23–128.
33. D. Rugg, Using a capability evaluation to select a contractor, *IEEE Softw.,* **10** (4): 36–45, 1993.

MANSUR H. SAMADZADEH
Oklahoma State University

MANSOUR K. ZAND
University of Nebraska—Omaha