Techniques for Software Maintenance

Kostas Kontogiannis
Department of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece

Abstract

Software maintenance constitutes a major phase of the software life cycle. Studies indicate that software maintenance is responsible for a significant percentage of a system's overall cost and effort. The software engineering community has identified four major types of software maintenance, namely, corrective, perfective, adaptive, and preventive maintenance. Software maintenance can be seen from two major points of view. First, the classic view where software maintenance provides the necessary theories, techniques, methodologies, and tools for keeping software systems operational once they have been deployed to their operational environment. Most legacy systems subscribe to this view of software maintenance. The second view is a more modern emerging view, where maintenance is an integral part of the software development process and it should be applied from the early stages in the software life cycle. Regardless of the view by which we consider software maintenance, the fact is that it is the driving force behind software evolution, a very important aspect of a software system. This entry provides an in-depth discussion of software Q1 maintenance techniques, methodologies, tools, and emerging trends.

INTRODUCTION

Software maintenance is an integral part of the software life cycle and has been identified as an activity that affects in a major way the overall system cost and effort. It is also a major factor for affecting software quality. Software maintenance is defined by a collection of activities that aim to evolve and enhance software systems with the purpose of keeping these systems operational. The field of software maintenance was first discussed in a paper by Canning,[1] where different software maintenance types where implicitly presented. However, it was due to a paper by Swanson[2] where the terms and types of software maintenance were first explicitly defined in a typology of maintenance activities.[2]

In the following years, the software community realized the importance of the field, and the Institute of Electrical and Electronics Engineers (IEEE) published two standards Q2 in this area. The IEEE standard 610.12-1990) and the updated standard 1219–l998 identify four major types of software maintenance.[3,4] The first type is referred to as Corrective Software Maintenance, where the focus is on techniques, methodologies, and tools that support the identification and correction of faults that appear in software artifacts such as requirements models, design models, and source code. The second type is Perfective Software Maintenance, where the focus is on techniques, methodologies, and tools that support the enhancement of the software system in terms of new functionality. Such enhancement techniques and methodologies can be applied at the requirements, design, or source code levels. The third type of software maintenance is referred to as Adaptive Software Maintenance and refers to activities that aim to modify models and artifacts of existing systems so that these systems can be integrated with new systems or migrated to new operating environments. A fourth type of software maintenance is Preventive Software Maintenance. Preventive Software Maintenance deals with all other design time and development time activities that have the potential to deliver higher-quality software and reduce future maintenance costs and effort.[5] Examples of Preventive Software Maintenance include adhering to well-defined processes, adhering to coding standards, maintaining high-level documentation, or applying software design principles properly. In general, preventive maintenance encompasses any type of intention-based activity that allows to forecast upcoming problems and prevent maintenance problems before they occur.[4,6] Preventive maintenance touches upon all the other three types of maintenance and in some respect is more difficult to define boundaries for.[6] Due to broad boundaries of preventive maintenance, in this entry we will mostly focus on core technical and process issues of the first three types of software maintenance, namely, corrective, adaptive, and perfective maintenance. The interested reader can refer to Refs. [2], [6], and [7] for a more detailed discussion on preventive maintenance.

A number of studies have indicated that software maintenance consumes a substantial portion of resources within the software industry. A study by Sutherland[8] estimated that the annual cost of software maintenance in the United States is more than \$70 billion dollars for a total of

Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044348

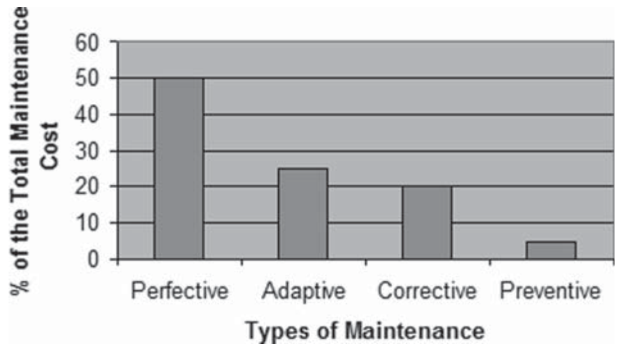


Fig. 1 Proportional cost per maintenance type.

approximately 10 billion lines of code. Considering that it is estimated that there are more than 100 billion lines of code around the world,[9] the annual worldwide cost of software maintenance is estimated at the staggering amount of $700 billion dollars. Furthermore, there have been studies estimating the ratio of software maintenance cost vs. total development cost. These studies indicate that software maintenance costs attribute at least 50% of the total development cost of a software system.[9] Other studies,[10, 11] have estimated that maintenance costs can even range between 50% and 75% of the total development cost. Fig. 1 illustrates the proportional cost for each type of software maintenance with respect to the total maintenance cost.

Software maintenance is the primary process for achieving software evolution. With accumulated experience over the years, a collection of rules and observations were formulated by M. Lehman[12–14] into what is known as the laws of software evolution. These laws relate to observations regarding continuous change, increased complexity, self-regulation, conservation of organizational stability and familiarity, continued growth, and quality degradation. The laws of evolution focus on the observation that in order for large systems to remain operational they must constantly be maintained, and that an unfortunate consequence of continuous maintenance is system quality degradation where systems become complex, brittle, and less maintainable. This phenomenon is referred to as software erosion and software entropy. There is a point when a system reaches a state where regular maintenance activities become very costly or difficult to apply. At that point the system must be considered for reengineering, migration, reimplementation, replacement, or retirement.

In this respect, software maintenance has been traditionally considered as an activity that is applied on the source code of the system and only after the system became operational. However, more recent views consider that software maintenance is an activity that can be applied in all phases of the software life cycle and to a variety of software artifacts. Therefore, maintenance nowadays is not considered as a postdevelopment activity but rather as an activity that is also applied during Greenfield software development.[15] This view also originates from the concepts of iterative, incremental, and unified process models as well as Model Driven Engineering (MDE)[16] that postulate software system development as an incremental process whereby requirements, design, source code, and test models are continuously updated and evolved.

As with every engineering activity, software maintenance must follow a specific prescribed process.[17] A complete maintenance path encompasses the identification, selection, and streamlining of software analysis and software reverse engineering, software artifacts transformation, and software integration.[18] Here, we will attempt to present a unified process description for software maintenance that includes fours major phases, namely, *portfolio analysis/strategy*; *modeling/analysis*; *transformation*; and *evaluation*.

In the *portfolio analysis/strategy* phase, the issues and problems of the system in its current form are identified.

In the *modeling/analysis* phase, software artifacts are denoted and analyzed so that maintenance requirements can be set based on the systems' state and strategy. The *modeling/analysis* phase allows for complete maintenance paths to be defined and planned. More specifically, in this phase, software artifacts are represented and denoted utilizing a modeling language and formalism, and consequently, various models of the existing system [usually models of the source code such as the Abstract Syntax Tree (AST)] are analyzed. The result of this phase is the identification of specific system characteristics that can be used to define maintenance requirements, maintenance objectives, and quantifiable measures for determining whether the results of a maintenance activity when completed will meet the initial maintenance requirements and objectives or not.

In the *transformation* phase, the selected maintenance path is applied utilizing software manipulation and transformation tools.

Finally, in the *evaluation* phase, measurements for evaluating whether the selected maintenance activities have met technical and financial requirements set in the analysis phase are applied.

Having briefly introduced software maintenance as a phase in the software life cycle, we can now proceed to discussing specific techniques, methodologies, and tools that support software maintenance. This entry is organized as follows. In the "Software Maintenance Process" section we discuss the software maintenance process. In the "Software Maintenance Techniques" section we discuss key software maintenance techniques, while in the "Tools, Frameworks, and Processes" section we discuss tools and frameworks for software maintenance. In the "Emerging Trends" section we present emerging trends in the area of software maintenance and in the "Concluding Thoughts" section we provide some final thoughts on this subject.

SOFTWARE MAINTENANCE PROCESS

As an engineering activity, software maintenance should adhere to specific processes. Different research groups and practitioners have considered the problem and have proposed a number of process models for software maintenance.[19] By taking into account the state of the art and practice, we can consider that software maintenance process encompasses four major phases, namely, *portfolio analysis and strategy determination*; *system modeling and analysis*; *artifact transformation*; and finally, *evaluation*.

The *portfolio analysis and strategy determination* phase aims to identify, gather, and evaluate the resources, components, and artifacts that make up a system and consequently assess the current state of the system so that specific maintenance requirements and objectives could be set. Depending on the nature of the problems discovered, the appropriate maintenance strategy and action can then be drafted.

The second phase, *system modeling and analysis*, aims to denote and represent system resources, components, and artifacts in a specific modeling formalism, and allow for the extraction of important information from the system for the purpose of understanding its structure, dependencies, and characteristics.

The third phase, *artifact transformation*, aims to apply various maintenance and transformation techniques to achieve the requirements and the objectives set in the *portfolio analysis* phase.

Finally, the *evaluation* phase aims to apply techniques to assess whether the maintenance requirements and objectives have been met as the result of maintenance operations as well as to assess the quality characteristics of the new system. These phases are applied incrementally and iteratively and not in a piecemeal sequential manner. In this respect, *portfolio analysis and strategy determination* phase feeds results to *system modeling and analysis* phase that in turn produces results that can be fed back to and revise/extend the *portfolio analysis and strategy determination* phase, moving iteratively, incrementally, and gradually through the *transformation* and *evaluation* phases. Fig. 2 illustrates a schematic block diagram of the maintenance process.

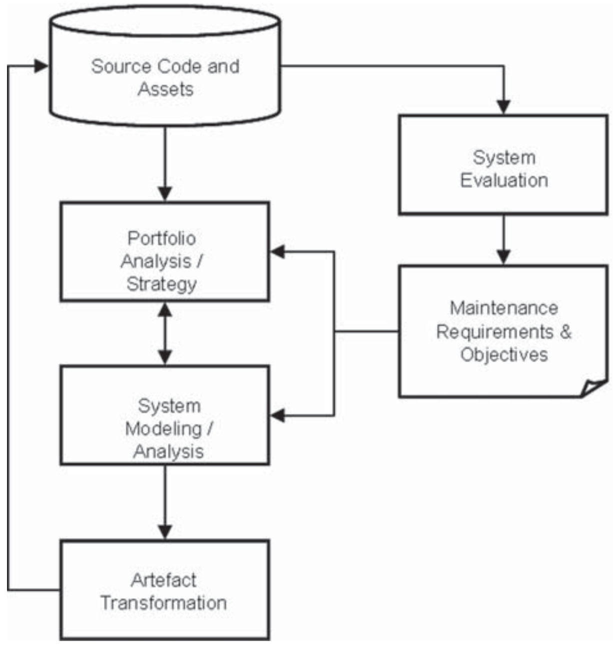


Fig. 2 Software maintenance process. Q18

Portfolio Analysis and Strategy Determination

The objective of portfolio analysis and strategy determination is to assess the system from a financial, technical, and business perspective. The purpose is to compile an inventory of a system's physical objects and its dependencies, to construct an operational profile of the system in terms of its delivered functionality, to calculate an estimate of its operational and maintenance cost and effort, to identify various Key Performance Indicators (KPIs), and to collect satisfaction ratings obtained from the users of the software system. The results of this phase can be used to establish maintenance requirements and determine the appropriate strategy that is required such as whether the system will be maintained (enhanced, ported, corrected), redeveloped, retired, or be kept as is. The sections below discuss in more detail the phases of portfolio analysis and strategy determination.

Software portfolio analysis

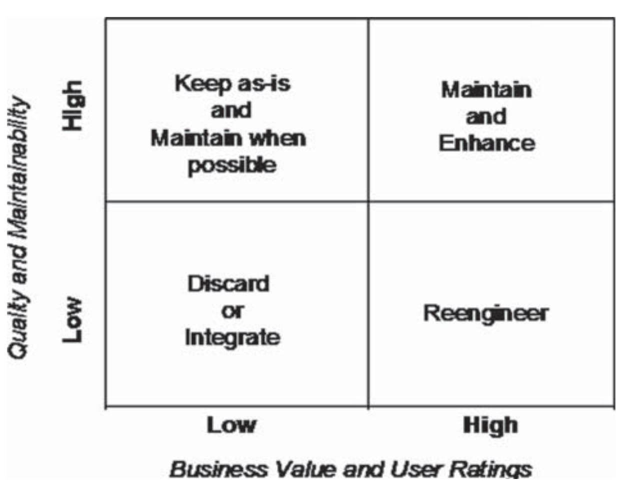
This phase aims to create an inventory of the system's physical objects and resources, evaluate its operational state, and assess the system's role in the overall corporate strategy, mission, and processes.[20] In addition, a compilation of data related to maintenance and operational costs of the system is important in order to evaluate maintenance efforts from a financial perspective. This phase requires static analysis of the source code to compile a record of the system's physical objects, and dynamic analysis of traces to assess the system's behavior against the specified or intended behavior. Compilation of historical and forecast financial data related to the cost of operations over the remaining operational period of the system are also important data to be collected in this phase.

Static analysis tools can provide a wealth of source code-related information such as valuable information with respect to unused code, metrics, poor coding practices, as well as component dependencies. The dynamic analysis tools can provide information with respect to whether the system achieves its functional and nonfunctional requirements including security issues, memory usage, performance degradation trends, and interface bottlenecks. Finally, the compilation of operation and historical maintenance data could provide valuable information with respect to annual change rate of the modules of the system, compilation of software maturity indexes, average time/effort/cost measures for typical maintenance tasks, mean time to failure metrics, mean time to repair metrics, failure intensity metrics, compilation of the number of cumulative system failures, as well as reliability and availability measures and profiles using standard reliability growth models.[21]

Strategy determination

The objective of this phase is to identify maintenance requirements and to devise a strategy with respect to maintenance activities that should be considered, given the current state of the system. The different strategies include restructuring, migration, porting, enhancement, redevelopment, or keeping the status quo, that is, leaving the system as is until its final retirement. In order to make these decisions, the results from the *portfolio analysis* phase are considered in addition to the information obtained from a number of system and environment characteristics that help determine the importance and vulnerability of the system from a mission and business operations perspective.[22–24] These characteristics pertain to system vulnerabilities due to obsolete implementation programming languages and infrastructure used, software mission criticality and anticipated impact in case the system fails, preparedness and the level of technical competency of the organization to undertake a maintenance project, availability of funds supporting the maintenance efforts, and management commitment.

In order to select an overall maintenance strategy one must consider several factors and conduct a thorough technical, financial, and risk assessment of the systems that are to be maintained. For the sake of simplicity, we can consider that a very generic assessment can be based on the system's business value vis-à-vis the ease of change, or on user ratings vis-à-vis the quality coefficient of the system. Fig. 3 summarizes such a high-level software maintenance



Q18 **Fig. 3** Guidelines for selecting a maintenance strategy.

road map from guidelines presented in Ref. [25]. Another guideline is based on questionnaires and detailed technical, economic assessments and management assessments that select the appropriate maintenance strategies for a given system or a family of systems.[24] Finally, yet another guideline that aims to produce a maintenance strategy specifically for migrating legacy systems to Service-Oriented Architecture (SOA) environments is also based on questionnaires and system analysis to establish the migration context, to describe existing capability, and to describe the target SOA state.[26] These different strategies aim to provide answers to whether it makes sense to apply a specific maintenance task, what parts of the system can be reused, what type of changes need to be applied to which components in order to accomplish the maintenance objectives, and how to obtain a preliminary estimate of cost for a given maintenance task or activity.

System Modeling and Analysis

The objective of modeling and analysis is the representation of source code (or even binary code)[27] at a higher level of abstraction using a domain model (schema), and the subsequent analysis of such models so that dependencies between system artifacts can be extracted and modeled.[28] This phase encompasses two major tasks. The first task is referred to as source code representation and utilizes parsing technology to compile a model of the source code that can be algorithmically and mechanically manipulated. The second task is referred to as source code analysis and aims to assist on program and system understanding. The sections below discuss in more detail the phases of source code modeling and source code analysis.

System modeling

System modeling focuses on the construction of abstractions that represent and denote source code, computing environment characteristics, and configuration information at a higher level of abstraction. In particular, the area of source code modeling or source code representation deals with techniques and methodologies to represent information on a software system at a level of abstraction that is suitable for algorithmic processing. System modeling has a profound impact in software maintenance as it affects the effectiveness and the tractability of maintenance activities. The effect of modeling formalisms to software maintenance has been discussed in Ref. [29].

There are two major schools of thought in the source code modeling domain. The first school of thought advocates formal models that not only aim to represent the source code at a higher level of abstraction but also to denote the semantics of the source code on a rigorous mathematical formalism. In this respect, formal properties of the code can be proven using theorem proving or other formal deduction techniques. Approaches that fall in this category include structural operational semantics, denotational semantics, axiomatic semantics, π-calculus, and process algebras.[30–32] A criticism on these approaches is that models are difficult to build and manipulate algorithmically, especially for large industrial systems.

The second school of thought advocates more informal models that are produced from parsing or scanning the code. These models do not necessarily have well-defined formal semantics, but provide and convey rich-enough information so that source code analysis and manipulation of large software systems can be tractably achieved.

It is evident that both approaches have benefits and drawbacks. The intuition behind the first approach that utilizes formal models allows for properties of the source code to be verified, a very important issue for mission-critical systems analysis. However, these approaches do not scale up very well as the complexity and the size of such formal models may become unmanageable for large systems.

Similarly, the intuition behind the second approach that utilizes informal models is that it allows for a "good-enough" analysis of very large systems in a tractable manner. For example, the architectural recovery of a multi-million line system does not require highly formal and mathematical models. The motivation here is to utilize models that can represent massive amounts of data to tractable algorithms so that we can obtain a solution that is useful to software engineers who can proceed with a more detailed and targeted analysis if needed. In this entry, we focus on the use of the latter type of informal models as these are mostly used in practice for the analysis and maintenance of large software systems. These include ASTs, Call Graphs, Program Summary Graphs, and Program Dependence Graphs (PDGs) among others. These representations are achieved by parsing the source code of the system being analyzed at various levels of detail and granularity (i.e., statement, function, file, package, subsystem level). Models can also be denoted by a variety of means such as tuples, relations, objects, and graphs. Regardless of how the models are denoted they have to conform to a schema that is referred to as the *Domain Model*. Domain models can be represented in a variety of ways but most often are represented as relational schemas or as class hierarchies.[33–36]

One specific type of model that is most often used for software analysis and maintenance is the AST. ASTs are tree structures that represent all the syntactic information contained in the source code.[37] Every node of the tree is an element of the programming language used. The non-leaf nodes represent operators, while the leaf nodes represent operands. ASTs suppress unnecessary syntactic details (whitespace, symbols, lexemes, punctuation tokens) and focus on the structure of the code being represented. The AST notation is the most commonly used structure in compilers to represent the source code internally in order to analyze it, optimize it, and generate binary

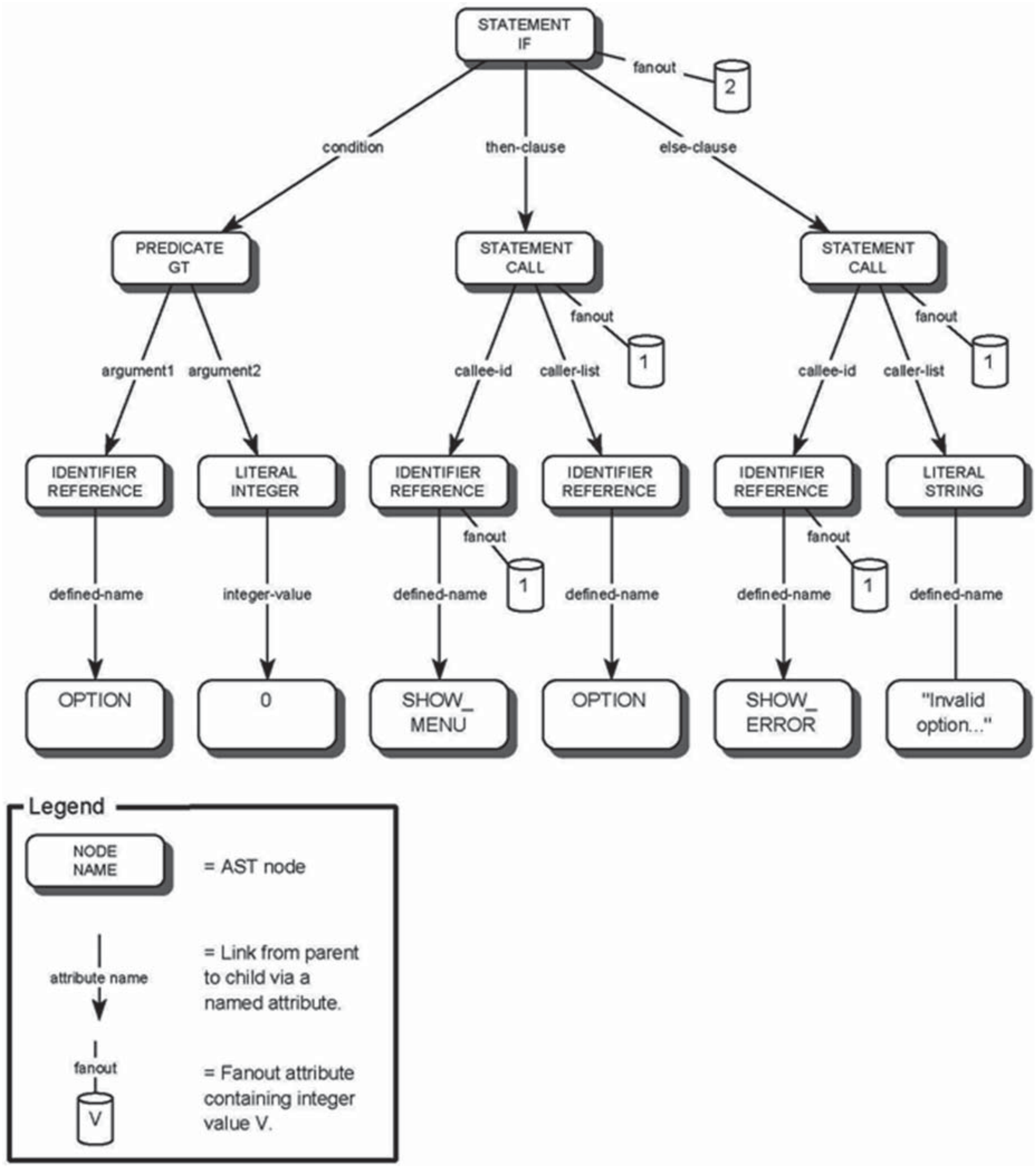
```
Statement
.......Statement_Block
.......Statement_Call
.......Iterative_Statement
.............Statement_While
.............Statement_For
.............Statement_DoWhile
.......Condition_Statement
.............Statement_If
..............................
Expression
.......Identifier_Reference
.......Literal_Integer
.......Predicate_LT
.......Predicate_GT
.......Predicate_Equality
......................................
```

Fig. 4 Sample domain model class hierarchy for the C programming language. Q18

code for a specific platform. In this context, source code modeling aims to facilitate source code analysis that can also be applied at various levels of abstraction and detail, namely, at the *physical level* where code artifacts are represented as tokens, lexemes and ASTs; the *design level* where the software is represented as a collection of modules, interfaces, and connectors; and the *conceptual level* where software is represented in the form of abstract entities, such as objects, Abstract Data Types (ADTs), and communicating processes.

An example of a fraction of a domain model for the C programming language presented as a class hierarchy is illustrated in Fig. 4. More specifically, Fig. 4 illustrates part of a domain model in the form of a hierarchy of classes that denote structural elements of the C programming language. For example, as depicted in Fig. 4, the C programming language has Statements, a subcategory of which is Condition_Statement. A subcategory of Condition_Statement is Statement_If, and so on. Similarly, the language has Expressions, subcategories of which include Predicate_GT, Predicate_LT, etc.

A parser can be used to invoke semantic actions that aim to populate such a domain model and create objects that are associated and form a tree structure as the one depicted in Fig. 5. Such trees are referred to as ASTs and provide a very rich model, which can be used for source code analysis and transformation. Fig. 5 illustrates the Annotated AST that is compliant with the domain model of Fig. 4 and pertains to the following snippet of code:



Q18 **Fig. 5** Sample Annotated Abstract Syntax Tree class.

```
IF (OPTION > 0)
      SHOW_MENU(OPTION)
   ELSE
      SHOW_ERROR("Invalid option..")
```

The Abstract Semantic Graph, or ASG for short, also provides a rich abstract representation of source code text. ASGs are composed of nodes and edges. Nodes represent source code entities, while edges represent relations. Both the nodes and the edges are typed and have their own annotations that denote semantic properties.[38]

The Rigi Standard Format, or RSF for short, is a format for representing source code information. It is a generic, intuitive format that is easy to read and parse. The syntax of RSF is based on entity relation triplets of the form <relation, entity, entity>. An example of an RSF tuple is <calls Function_1 Function_2>. Sequences of these triplets are stored in self-contained files.

Currently, RSF is the base format for the reverse engineering tool Rigi.[34,39]

The Tuple-Attribute Language, or TA for short, is a metamodeling language designed to represent graph information.[40] This information includes nodes, edges, and any attributes the edges may contain. TA is easy to read, convenient for recording large amounts of data, and easy to manipulate. The main use for TA is to represent facts extracted from source code through parsers and fact extractors. In this way TA can be considered to be a "data interchange" format. Other metamodeling languages for representing software artifacts include the FAMIX metamodel used by Moose,[41] the GXL,[36,42] and the Knowledge Discovery Metamodel (KDM)[43] proposed by the Object Management Group (OMG).

Other popular source code representation models include the Control Flow Graphs, Data Flow Graphs, Call Graphs, and PDGs.[37,44] Control Flow Graphs denote the possible flows of execution of a code segment from statement to statement. Data Flow Graphs offer a way to eliminate unnecessary control flow constraints in representing the source code of a system, focusing mostly on the exchange of information between program components (basic blocks, functions, procedures, modules). Call Graphs offer a way to eliminate variations in control statements by providing a normalized view of a possible flow of execution of a program. In Control Flow Graphs, nodes represent source code basic blocks, while edges represent possible transfer of control from one basic block to another. A Call Graph represents invocation information between functions or between procedures. Nodes in the Call Graph represent individual functions or procedures and edges represent call sites and may be labeled with parameter information. Finally, PDGs have been extensively used for software analysis and in particular source code slicing. Nodes in PDGs represent either statements or entry or exit variables in a code fragment, usually this code fragment being the body of a function, while edges represent either control dependencies, or data flow dependencies. Control dependencies in PDGs indicate that one statement may or may not select the execution of another statement (e.g., the condition in a Statement_If may or may not select the execution of the then or the else part of the statement's body). Similarly, data flow dependencies between nodes in PDGs denote that one statement (node) defines the value of a variable and the other statement (node) uses the variable.[44]

In the paragraphs above, we focused mostly on the representation of modeling of source code. Another important dimension is the extraction of models from binary files, an area what is referred to as binary analysis.[27,45]

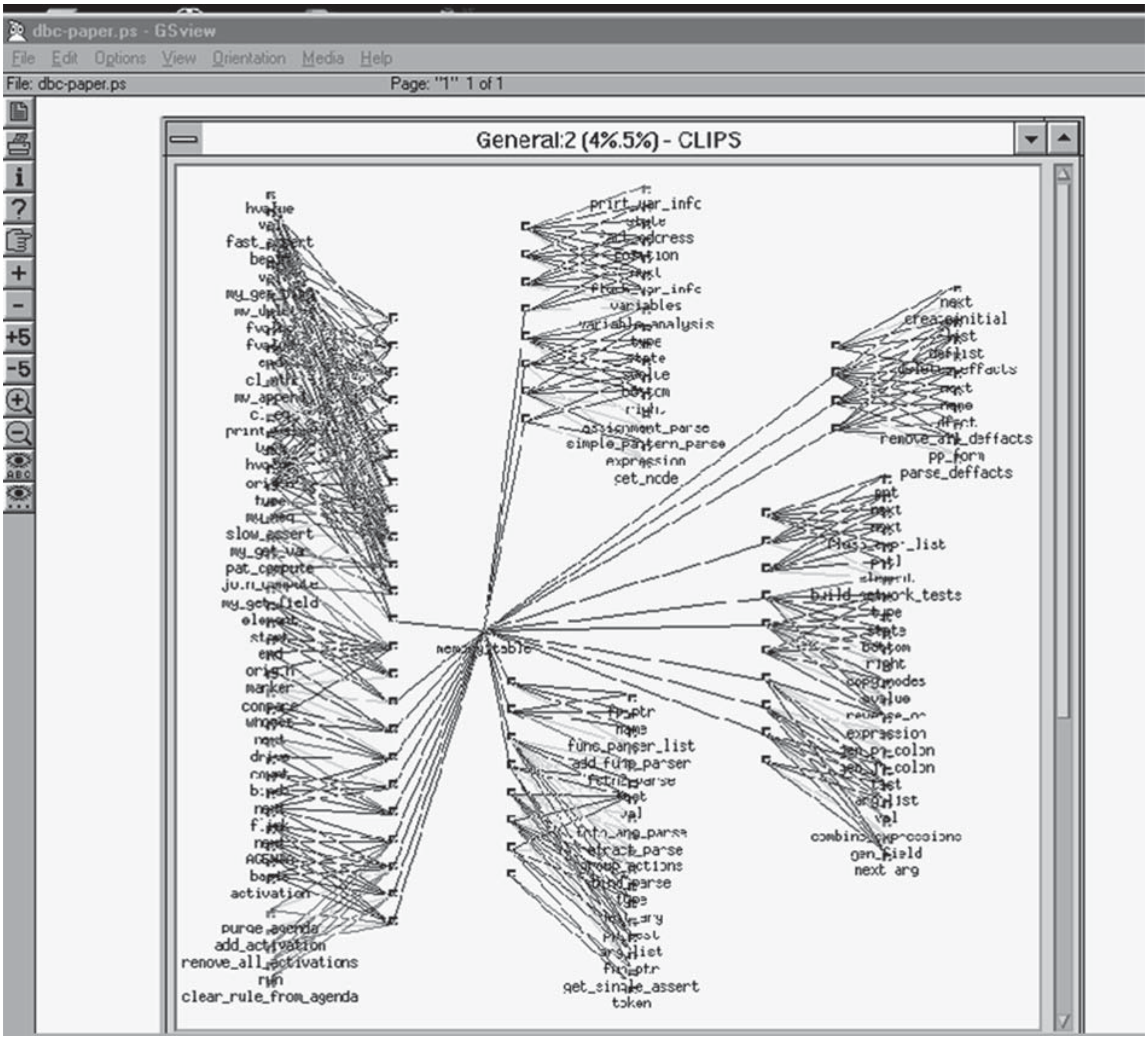
System analysis

Analysis takes two forms: static and dynamic analysis. Static analysis focuses mostly on the analysis of the source code.[46] Dynamic analysis aims at extracting information and models from execution traces.[47–49] The primary focus of the static or dynamic analysis of the system is the compilation of various system views such as architecture views, code views, metrics views, and historical views. These views allow the identification of the system's major components, data and control flow dependencies, data schema structure, configuration constraints, as well as the extraction of various metrics that serve as indicators of the system's quality and maintainability. For system analysis we consider the following important points of interest.

Architectural extraction is one such area of system analysis that deals with the identification of major system components and their dependencies.[50] These components and their dependencies (calls, uses, imports, exports) provide a view of the system at the architecture level. The identified components are composed of collections of functions, data types, and variables. For the formation of the components that constitute an architectural view of the system, clustering is used as the primary technique. Elements such as functions, data types, and variables are grouped together according to some clustering strategy and according to some distance or similarity measure. Architecture extraction analysis can be used for the migration of legacy systems to Network-Centric and to Service-Oriented environments. An example of an extracted architecture is illustrated in Fig. 6, where a large system has been abstracted in the form of an Entity Relationship (ER) graph where nodes are files and edges denote data flow, control flow, and call information between these files. A clustering algorithm has been applied in this ER graph to yield groups of files that share common flows while flows between groups are minimized. These groups illustrated in Fig. 6 can be considered as components of the recovered system architecture.

Another important area of system analysis is the extraction of ADTs.[51,52] The extraction of ADTs encompasses two tasks. The first task deals with the analysis of data types in the source code of the system and the assessment of whether these data types can be considered as ADTs or not. The assessment criteria are based on how extensive is the use of these types in the system, the operations that are associated with these data types, and their relationship with other data types. The second task deals with the identification and attachment of operations to these ADTs as well as with the identification of possible specializations and generalizations among these types. ADT extraction can be used for the migration of systems written in procedural languages to new systems that conform with the Object-Oriented programming paradigm.[53]

Yet another important area of system analysis is slicing.[54,55] Slicing is a technique that allows for the identification of all system elements that are affected by, or affect, a particular system element at a particular location that is referred to as the slicing criterion. Slices can be classified as forward slices and backward slices. The most



Q18

Fig. 6 Snapshot of extracted system architecture.

common use of slicing is source code slicing. In this respect, a slicing criterion is a program statement variable at a specific point. A forward slice is a collection of statements that are affected by the slicing criterion. A backward slice is a collection of statements that affect the slicing criterion. The result of the slice can even be an executable program that includes all the statements in the forward or in the backward slice given a slicing criterion. Slicing is based on the analysis of source code models such as the AST or the PDG.[44]

Another area of system analysis is data flow analysis. Data flow analysis is an area originally proposed in the area of optimizing compilers and in the area of unit testing .[37,56] Data flow analysis is based on how data propagate through the control flow structure of a code fragment. There are several data flow analysis algorithms proposed in the literature. In the context of software maintenance, these algorithms are used for determining the dependencies between statements and for estimating the potential impact of a change when maintaining the source code of a software application. The most frequently used data flow algorithms for software maintenance include the algorithms of reaching definitions, def-use chains, available expressions, and constant propagation. Reaching definitions data flow analysis aims to identify all definitions of a variable that reach a given point, in other words, what are the possible values of a variable at a given program point. Def-use chains data flow analysis aims to identify for a given definition of a variable all the possible uses of it. Available expressions data flow analysis aims to identify the scope in which the value of specific expressions remains unchanged. In this respect, as long as the value remains unchanged, one could potentially replace a complex reoccurring expression with a variable denoting its value. This allows for program simplification and therefore easier maintenance. Finally, constant propagation analysis aims to identify variables that obtain a constant value and all the points where this constant value can be safely used.[37]

Another area is data schema analysis. Data schema analysis is a type of analysis that pertains mostly to data-centric systems that utilize and access transactional databases. To maintain such systems it is not adequate to analyze the source code that pertains to the transaction logic or business logic but it is also important to analyze and identify dependencies between the elements of the data schemas used by the application.[57] Data schema analysis can be used for schema simplification, thus reducing the size and complexity of the data or facilitating schema merging and schema mediation, two very important tasks for system integration and interoperability.

Similarly, the area of metrics analysis is a type of system analysis that is aiming for the compilation of a collection of various software metrics that reflect in quantifiable terms structural properties of the code and of the design of the system in general. These computed metrics can be used in different ways for software maintenance. Some of the most typical uses of metrics are as quality predictors for a specific design,[58] as differential indicators when two versions of a system are compared, or as clone detection techniques.[59,60]

Transformation

The objective of the *Transformation* phase is to apply manual, semiautomatic, or fully automatic techniques that allow for the manipulation of the source code, the event flows, or the database artifacts so that maintenance goals and objectives can be achieved. An example of a particular type of transformation is the architectural repair transformation that is applied at the design or architecture level and aims to repair a legacy system's architectural drifts that may have occurred due to prolonged maintenance operations. The transformation phase can be considered in the context of three distinct activities, namely, *functional transformations*, *process transformations*, and *data transformations*.

Functional transformations can be applied at the design/architecture level or at the source code level. These transformations aim to manipulate code artifacts to repair faults, add new functionality, or adapt/port the system to new platforms and environments. When functional transformations are applied at the design/architecture level they take the form of what is referred to as architectural repair.[61] Similarly, when functional transformations are applied at the source code level they take the form of what is referred to as code transformations.[62] When these transformations do not aim to change the behavior and the functionality of the system (e.g., to increase its quality) they are also referred to as refactorings.[63,64]

Process transformations are applied at the event flow level or at the workflow/process level. Event flows denote how and in which order events are channeled from originator processes (callers) to receiver processes (callees). Similarly, workflows denote the order by which processes and data stores are activated during the system's operation for a given use case or application scenario. When process transformations are applied at the event flow level, we aim to manipulate the way systems/components interact and are part of an area called *event processing*. Examples of event flow transformations include integration with third-party components and integration with monitoring/auditing tools. When process transformations are applied at the workflow or business process level these take the form of IT workflow reengineering or business process reengineering, respectively.[65]

Data transformations are applied at the schema or data instance level with the purpose to analyze and manipulate the logical and physical data in an application. When data transformations are applied at the logical level, these may take the form of data schema manipulations. Similarly, when data transformations are applied at the physical level, these may take the form of manipulation of data streams or even manipulation of individual instances of data.[66] An example of data transformations at the logical level is the alteration of a schema in a database by adding or removing fields and relations. Similarly, an example of data transformation at the physical level is the alteration of data streams from binary to text, or the alteration of a specific data item in a form that can be consumed by a client process.

Evaluation

The objective of the evaluation phase is to assess whether software maintenance activities fulfill the maintenance requirements and the desired maintenance goals set in the strategy determination phase. In this respect, there are two major points of view we can approach the evaluation phase. The first is from a technical standpoint while the second is from a financial standpoint.

Technical evaluation

The selection of the technical evaluation strategy to be used depends on the type of maintenance objective (adaptive, corrective, perfective) and the type of transformation (functional, process, data) that has been applied. One could consider four major technical evaluation strategies. The first is based on static analysis of the source code, the second is based on software testing techniques, the third is based on software metrics, and the fourth is based on feature modeling and analysis. These four strategies are complementary in the sense that each one provides different evidence toward evaluating the effectiveness of the maintenance operations applied.

The first technical evaluation strategy, namely, the static analysis of the source code can be applied when we have ample access to the source code of a system, and we would like to prove that certain structural properties of the source code or the design of the system hold after specific maintenance operations have been applied and completed. This type of evaluation may take the form of identifying that certain dependencies between components have been eliminated or established accordingly, or that certain design patterns or refactorings have been introduced.[67,68] Component dependencies include data and control dependencies, use or exposure of interfaces from one to other components or applications, or the use of common libraries and protocols. Similarly, static analysis techniques can assist toward the evaluation of whether or not a design pattern or refactoring has been or can be introduced in a particular part of a system. In this respect, it has been experimentally shown that the introduction of design patterns or refactorings plays a role toward enhancing the quality of a software system.[69] By verifying statically that such structures have been introduced into the system, software engineers may make assumptions on the quality and extensibility of the new system.

The second technical evaluation strategy is based on testing and can be used whether we have full access to the source code or not.[56,70] Furthermore, different testing techniques can be used for different types of evaluations. For the evaluation of the impact maintenance activities have on a system, software engineers may opt for regression testing techniques applied at the unit level (unit testing), at the component level (integration testing), or that the system level (system and functional testing). Depending how much access we have to the source code of a system, we can choose what type of testing we can apply. When we have full access to the source code we consider white box testing techniques, whereas when we have partial or no access to the source code we apply gray box or black box testing, respectively.

The third technical evaluation strategy is based on metrics and can be used when we have some access or full access to the source code of the system. The metrics techniques can be applied in two ways. First, metrics can be used to determine whether particular components of a software system meet specific structural properties that can be quantified by such metrics. Second, metrics can be used to determine whether specific maintenance operations that have been applied had a differential impact on metrics before and after the specific maintenance operations took place. This differential on metrics values may serve as an indicator that the specific maintenance operations had a positive or negative impact on the system quality. In the related literature there are several different software metrics that have been proposed as indicators of the overall system quality and maintainability.[71] In this respect, Victor Basili in Ref. [72] proposed the Goal-Question-Metric (GQM) approach, where different metrics can be used to assess software systems and processes according to the specific assessment goals and assessment questions that arise from these goals. Examples of metrics that relate to evaluating a software system with respect to maintenance cost and effort include Current Change Backlog, Change Cycle Time from Date Approved and from Date Written, Cost per Delivery, Cost per Activity, Number of Changes by Type, Staff Days Expended/Change by Type, Complexity Assessment, etc.

Other examples of metrics that serve as indicators of the quality and maintainability of a code fragment is the cyclomatic complexity of its Control Flow Graph measured by the McCabe cyclomatic complexity metric, and the estimated degree of the delivered functionality by a code fragment measured by the Function Point metric. The interesting property of the Function Point metric is that it can also be applied at the design or component level of a system and relates to the level of cohesion of a code fragment or a component.[73]

Furthermore, metrics can be combined in linear formulas that serve as predictors of software quality or maintainability.[74] These linear formulas can be developed experimentally using past maintainability data and linear interpolation techniques. For example, in Ref. [58], the software maintainability index (SMI) of a software module is estimated by a linear formula of the form

$$\mathrm{SMI} = 125 - 3.989 * \mathrm{FAN}_{\mathrm{avg}} - 0.954 * \mathrm{DF} - 1.123 * \mathrm{MC}_{\mathrm{avg}}$$

where SMI is the predicted maintainability of a module, $\mathrm{FAN}_{\mathrm{avg}}$ is the average number of calls emanating from all classes/functions of the module, DF is the total number of incoming and outgoing data flows from the module, and $\mathrm{MC}_{\mathrm{avg}}$ is the average McCabe cyclomatic complexity of all methods/functions of the module. According to experimental studies presented in Ref. [75], SMI values below 65 indicate low maintainability, values between 65 and 85 indicate medium maintainability, while values above 85 indicate high maintainability.

The fourth technical evaluation strategy, namely, feature-based analysis aims to collect features from the system (design artifacts or source code artifacts) and attempt to assess how design decisions affect quality attributes. Two of the most well-known techniques in this area are Software Architecture Analysis Method (SAAM)[76] and the Architecture Tradeoff Analysis Method (ATAM).[77] These techniques aim to assess the impact design decisions have on software system quality with emphasis on performance, security, availability, and modifiability. These techniques can also assist on evaluating the trade-off among different design choices. The techniques are based on a structured process that aims to identify and present the scope and goals of the evaluation; perform investigation and analysis of the alternative design decisions; test and prioritize alternatives; and report findings. The techniques can also be used for Greenfield software development[15] early in the life cycle or to analyze existing legacy system architectures.

Financial evaluation

The evaluation of maintenance activities from a financial standpoint takes the form of computing the maintenance and operational cost for keeping the system as is vs. the cost of performing specific changes to the system (adaptive, corrective, or perfective changes), and computing the expected reduced maintenance and operational cost for the new enhanced system if the maintenance operations are applied. In the related literature there are a number of different techniques to perform economic analysis for evaluating maintenance operations from a financial standpoint. Most of these techniques fall into three major categories.

The first category deals with the traditional economic analysis that is based on the concepts of Net Present Value (NPV), Benefit Investment Ratio (BIR), Return on Investment (ROI), and Rate of Return (ROR).[24] These indicators, especially the NPV and BIR indicators, are used to determine which type of maintenance operation will have the highest economic impact or alternatively, what is the economic impact of a selected strategy. The strategy that yields the highest NPV or BIR is preferable from a financial point of view. Similarly, we are also interested in keeping the ROI and ROR indicators as high as possible. In the related literature, the strategy of doing nothing and keeping the system as is to deteriorate toward its retirement is considered the null or status quo strategy. All other strategies can be compared with the null strategy. The NPV of a given strategy is defined as the difference of the Present Value (PV) of the total cost of the status quo strategy and the PV of the total cost of the given strategy. The PV of the total cost of a strategy is defined as the sum of the cost of implementing the strategy and the cost of operating and supporting the new system that has been produced by applying the specific strategy.[24]

The second category is based on prediction methods that estimate the effort savings for support and maintenance in person months for the new updated system vs. the effort for support and maintenance in person months for the old system.[78,79] An example of such an effort prediction model is COCOMO,[80] which takes as input the annual change rate of the system due to scheduled maintenance, the effort to implement a maintenance strategy, and a quality indicator of the system, and yields an estimate for the maintenance effort for this system. As the quality indicator of the system increases, the annual maintenance effort to keep the system operational decreases. In this respect, we aim to select those maintenance operations that have the highest impact on the quality indicator of a system. There are a number of different quality indicators that can be used, such as the Software Maturity Index, and indexes that are based on a linear combination of software metrics.[71,58]

Finally, a third technical evaluation category that is gaining attention over the past few years is based on futures valuation theory and on portfolio valuation theory. Valuation theory is a robust field in finance and proposes techniques to estimate the value of a choice that becomes available with an investment (i.e., a specific maintenance operation).[81] As software maintenance and evolution activities imply new choices for future expansion, adaptation, and correction, futures valuation theory allows for the estimation of the value (i.e., the future benefits) of the alternative investment opportunities (i.e., alternative maintenance operations). The choice with the highest valuation can be considered as the most preferable from a financial point of view. Relevant to the futures valuation approach are techniques that aim to valuate software portfolios. The idea is that alternative maintenance options provide as a result different alternative software portfolios that can be evaluated. The maintenance operation that yields a software portfolio with the highest value among alternatives can also be considered as the most preferable from a financial point of view.[24] A comprehensive list of cost estimation models can also be found at NASA's Cost Analysis Division.[82]

SOFTWARE MAINTENANCE TECHNIQUES

As presented in the sections above, software maintenance is classified by three major types, namely, corrective, perfective, and adaptive maintenance. In this section, we present some of the most frequently used software maintenance techniques per maintenance type. In the area of corrective maintenance we discuss techniques that are applied at different levels of software abstraction and granularity and deal with architectural mismatch repairs, component restructuring, fixing software errors, and identifying possible causes of failures. In the area of perfective maintenance, we discuss techniques that deal with enhancing a system with new functionality, performing software refactorings, and merging software components. Finally, in the area of adaptive maintenance, we discuss techniques that deal with synchronizing business processes with run time applications that implement these processes source code, migrating components to Network-Centric environments, and transliterating software systems to new languages.

Corrective Maintenance

Corrective maintenance is focusing on the application of techniques for understanding, fixing, or remediating problems in the source code or the design of a software system. Corrective maintenance can be thought of as encompassing two phases. The first phase deals with the identification of the fault, or what is also referred to as root cause analysis (RCA). The second phase deals with the correction of the

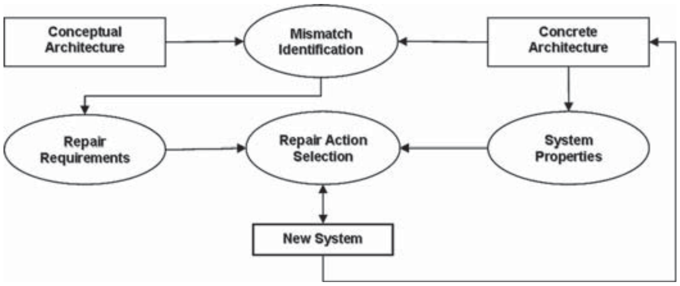


Fig. 7 Architecture repair process. Q18

problem. Unfortunately, there are no prescribed ways to fixing faults as every case is different. However, we could consider frameworks and tools that could assist these fault identification and correction tasks. In this section, we discuss three types of techniques, namely, architecture reconstruction and repair techniques, techniques for the identification of inconsistencies between requirements and run time applications, and environments that assist the identification and correction of software faults.

Architecture reconstruction and repair

The laws of evolution presented by Lehman imply that a system as it is being maintained evolves constantly. Furthermore, as time goes by, its design and architecture gradually erodes. There is a point in time where the system becomes so brittle and inflexible that even simple maintenance tasks are difficult to perform. At this point, one could consider applying corrective maintenance techniques that aim first, to reconstruct or extract the architecture of a system and second, to repair the architecture. Reconstruction takes the form of identifying components in terms of collections of functions, data types, and variables. The identified components should contain entities that exhibit high level of cohesion and are related by strong data and control dependencies. On the contrary, elements that belong to different components should exhibit low coupling. Architectural reconstruction can be achieved by clustering algorithms that can be either unsupervised (depend solely on the algorithm and the similarity distance)[83] or supervised (there are constraints and initial conditions as to how the components look like).[50] The intuition behind clustering techniques for architectural reconstruction is that the software system can be represented at a high level of abstraction as a collection of relations between software artifacts. An example of such a relation is the “calls” relation between functions. Once a collection of relation tuples is used to model the software system at a higher level of abstraction, clustering techniques can be applied to group together software artifacts that have a high number or relations between them, while as a group have minimal relations with other software artifacts that belong to other groups. This is a way to simulate a form of high cohesion and low coupling and therefore assume that the resulting clusters may provide a snapshot view of the as-currently-is system architecture.

Similarly, repair takes the form of identifying drifts between the extracted or *concrete architecture* and the *conceptual* or envisioned architecture. The *concrete* architecture depicts the actual interactions between the modules as implemented in the source code. These module interactions are based on dependencies between program entities (e.g., functions, data types, and variables). The *conceptual* architecture depicts the interactions we believe should exist between the modules following design and other domain-specific principles and information. Fig. 7 illustrates the architectural repair process based on the comparison of the concrete and conceptual architecture of a system. The new system yields a new concrete architecture that is evaluated against properties, constraints, and invariants stemming from the conceptual architecture of the system.

In this context, architectural repair may take two forms: forward architecture repair and backward architecture repair.[61] Forward architecture repair means repairing the *concrete* architecture to match the *conceptual* architecture. Reverse architecture repair means repairing the conceptual architecture to match the concrete architecture. This match takes the form of reconciling the conceptual architecture and the concrete architecture to minimize their anomalies. For this reconciliation to be achieved, a number of transformations can be considered. These transformations aim to eliminate specific differences between the conceptual and concrete architecture.[84,85] Depending on the differences we are interested in eliminating, transformations may take the form of insertions, deletions, and modifications of components and connectors in the concrete architecture to match the conceptual or vice versa. The rationale behind such repairs is to minimize structural and design inconsistencies between what is implemented (concrete architecture) vs. what should have been implemented (conceptual architecture).

Identifying inconsistencies between requirements and run time behavior

In many occasions, either due to prolonged maintenance or due to erroneous design or implementation of a system, the system does not deliver correctly its intended functionality as this is specified in its Software Requirements Specification (SRS) documents.[86] In order to identify whether the system delivers its intended functionality, we need to consider techniques to track a system's run time behavior so as to detect deviations from its requirement specification and identify parts of the system that may be responsible for this deviation.[87] Approaches in this area fall into three main categories: *event-driven*, *goal-driven*, and *pattern-driven*.

Event-driven approaches model assumptions through appropriate formalisms such as the Formal Language for Expression Assumptions (FLEA) or through events that are classified as satisfaction events or as denial events.[88] The intuition behind these approaches is to attempt to formally verify or deny such formal expressions that represent assumptions related to specific functional or non-functional system requirements by using information that is collected as the system runs. More specifically, the system is monitored and when an assumption is violated or a denial event is observed the associated requirements are considered to be in violation and remedial or maintenance actions can be taken. A denial event or a violation of an assumption is associated with the failure of a requirement and consequently with the potential failure of one or more components.[89] Failure hypotheses are associated with remedial actions through predefined requirement/assumption/remedy tuples or through specialized types of analysis such as obstacle analysis.

Goal-driven approaches utilize a model of the system that describes the prerequisites for the system's correct operation in terms of an AND-OR goal tree or a goal graph. The intuition behind these approaches is to attempt to satisfy logical combinations of goals that have to be achieved so that a specific functional or non-functional system requirement can be met. The combinations of these goals that need be satisfied can be collected from a constraint satisfaction or a SAT solver given a collection of goals in an AND-OR tree. When there is a mismatch between the actual and expected behavior, then the subgoals that are associated with this behavior mismatch in the goal tree are considered as potential root cause hypotheses.[90] A problem solver (i.e., a propositional satisfiability—SAT solver)[91] can be used to traverse the goal tree and identify all the combinations of actions or other subgoals that may contribute to this mismatch between the expected and observed system behavior.[92]

Pattern-driven approaches model requirement failures as patterns specified in a pattern language. The intuition behind these approaches is to identify through pattern matching abnormal sequences of events that are interleaved with normal events as the system runs. The abnormal sequences of events are usually associated with different types of possible system failures, attacks, and threats, and are usually modeled as patterns in a pattern language. These patterns are matched against the events that are collected as the system operates. If a pattern is observed, the associated failures and threats are then considered as initial root cause hypotheses. The requirement violations/threats can be associated with components or with remedy actions through prespecified condition-action rules, or tuples.[93]

Fixing faults

When a system operates it is possible at some particular point in time that a fault or bug in the code will be triggered. This fault in the code may be harmless or may trigger an error. An error is defined as the situation in which a system enters a state that is different from the one that is specified. An error may also be harmless or be remediated through exception handling mechanisms so that the user may not even be alerted or may not notice that something went wrong. However, there are situations where errors result in failures. A failure is defined as the situation where the observed behavior of a system is different than the one specified and from the one the user is expecting to observe. One part of corrective maintenance is identifying, tracing, and correcting faults in the code or the configuration of a system once a failure has been observed. The software community has proposed over the years a number of techniques for making the task of fault identification easier[94] once a failure is observed. These techniques include static code analysis, identification and interpretation of antipatterns,[95] as well as tools and techniques to perform historical analysis, bug tracking, and software evolution analysis.[96,97] Static code analysis techniques rely on the analysis of the source code of a system in order to infer characteristics of the program that may lead to unsafe array, garbage collection, or pointer operations. A common technique for these tools is to use symbolic execution and path simulation[98] where program execution paths are simulated using symbolic variables to determine if any software errors could occur. Another static code analysis technique for bug detection is the identification of source code patterns that match known bug patterns.[99] These systems first construct an abstract model of the source code being analyzed and then attempt to match these abstract models against predefined error patterns. Most of these systems are customizable in the sense the users can add their own patterns utilizing a tool-specific formalism. A variation of the above approach is based on the identification of antipatterns and the identification of code "bad smells." Antipatterns are design patterns that are commonly used but are ineffective and counterproductive.[100] Similarly, "bad-smells" are bad design and bad programming implementation patterns that are proven to cause problems related to the performance, reliability, maintainability, and robustness of a software system.[101] Software evolution history analysis tools can also be used to examine information relevant to the evolution of a system, to investigate the rationale behind applied changes and past maintenance operations, and also to store metrics such as complexity metrics, maintainability metrics, failure intensity metrics, and cumulative failure counts. These systems can also be used to associate functional and non-functional requirements with particular components of the system or configuration constraints.[92] Bug tracking systems also play an important role in tracing faults in the code. These systems keep information on open or already remedied faults as well as information on whom and how worked to remedy the fault. These systems not only help software maintainers to focus their attention on a particular component when a failure is observed, but also help users to understand dependencies between system elements. Finally, versioning systems play an important role in understanding how one system version differs from its previous version. In addition to providing the ability to roll back to previous versions, versioning systems also assist maintainers for associating failures with components, as in most cases failures occur due to a previous change, maintenance operation, or alteration in the computing environment or configuration parameters.

Finally, another aspect of corrective maintenance is the need for an infrastructure to assist software maintainers to perform both regression testing once an error is fixed and to perform reliability analysis during and after regression testing runs.[21,56] More specifically, performing regression testing after a maintenance operation has been applied is always important and required. Just passing the prescribed regression tests may not be adequate especially for mission-critical systems. What is also needed is the analysis of the reliability of the system by the utilization of reliability growth models. In this respect, the failure intensity of a system is measured every time after a test suite is applied and an error has been discovered and fixed. Regression testing, integration testing, and system testing should be kept on being applied up to a point where the failure intensity of a system measured in failures per CPU hour of operation falls below a prescribed level. Reliability growth models can be used to predict how many hours of testing are still required for the failure intensity of a system to drop below a specified value.[21] As an example, consider that for some mission-critical systems it has been reported failure intensity rates of 0.1 to approximately 5.5 failures per thousand CPU hours of operation. Depending on the criticality of the mission and the potential impact of a failure (loss of life, financial loss, bad publicity, or mere inconvenience), the guidelines and the acceptable maximum tolerable value may vary.

Perfective Maintenance

Perfective maintenance aims to apply changes in the system in order to increase some of its functional and some of its non-functional quality characteristics. Examples of perfective maintenance operations that deal with the functional characteristics of a software system include adding new functionality, while examples of perfective maintenance affecting the system's quality characteristics include increasing its maintainability, extensibility, portability, security, reliability, and usability. In this section, we focus on some of the most frequently used perfective maintenance techniques that are related to adding new functionality to a software system, merging software components to yield a new component, and refactoring a component or a system to increase some of its quality characteristics.

Adding new functionality

Adding new functionality to a system may take various forms. One form is to integrate the system with other components or applications at the architecture level. Another form is to adapt or extend its source code at the component or class level so that new functionality can be added through specific design patterns. Depending on the level of abstraction and granularity these maintenance operations are applied on, we can differentiate between architecture-level changes and source code-level changes.

At the architecture level, we consider architectural patterns that allow the integration of a system with other systems. The use of facades and wrappers[102] can be used to facilitate the addition of new functionality in the system by integrating the system with other components and applications at the architecture level. Such wrapper components provide standard utility services such as transaction management, message mediation, authentication, access control policies, encryption, etc. In the software engineering literature there are a number of architecture level patterns that have been proposed for extending in a proper way the functionality of a system by altering parts of its design at the architecture level. These include Data Source Architectural Patterns, Object-Relational Structural Patterns, Object-Relational Patterns, Distribution and Concurrency Patterns, and Session State Patterns. A collection of architecture-level patterns can be found in Refs. [103] and [104]. Yet another architectural-level technique that is used to integrate systems with other applications, thus extending the functionality and capabilities of such systems, is the use of Enterprise Service Bus (ESB). The ESB is an architectural abstraction that is usually implemented utilizing middleware technologies and provides fundamental system integration services such as data and protocol mediation as well as message and event processing.[105] Fig. 8 illustrates the deployment and use of a typical enterprise bus. The ESB provides a wealth of infrastructure services such as message queuing, message routing, message mediation, interface adaptation, logging, monitoring, and security. To date ESB is the dominant commercial choice for system integration and interoperability.

At the source code or at class level the addition of new functionality can be achieved in various ways. One way is the addition of new methods to a class or the addition of

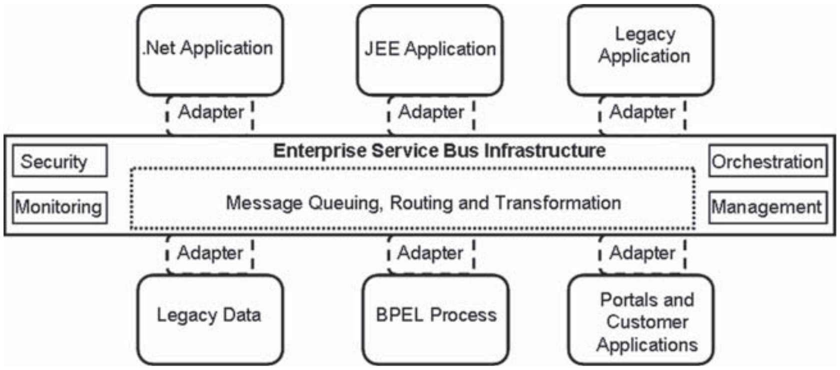


Fig. 8 Schematic of an Enterprise Service Bus. Q18

new classes to a package. This is a technique that requires care, as maintainers have to make sure that key architectural constraints are not violated (e.g., violation of design invariants); the enhanced class or the enhanced component remains cohesive (e.g., new functions operate on same type of data and deliver related to the class/component functionality); there are no side effects while extending the class or the components (e.g., files, databases, resources, or critical regions) are accessed without proper permissions and locks; and the enhanced class or the enhanced component maintain a low level of coupling with the other components (e.g., no unnecessary interfaces are exposed and no data are accessed without the proper use of interfaces). Another method is the use of appropriate design patterns such as the adapter, decorator, proxy, state, and factory method.[106] The use of such design patterns assumes that we have access to the source code and it can be achieved via the use of refactorings and program transformations.

Software merging

In some cases, an application is developed as a result of merging source code of two or more existing systems. This merging phenomenon is also a very common issue when a large software system is developed by many groups that have to merge their components to form the final product. Software merging is also a common issue in product-line type of software development where software versions with possibly common ancestors are merged to form new versions or new products.[107] In this respect, we can consider software merging as a form of customization and an important part of Perfective Software Maintenance. In practice, software merging is handled by tools that support collaborative software development, and software configuration management. These tools aim to address the issue in a tractable and decidable way by limiting the scope of merging, the programming languages supported, and the type of artifacts that can be merged. In general, software merging techniques can be classified as two-way or three-way merging techniques. Two-way merging deals with techniques that attempt to merge two versions of a software artifact without relying on the common ancestor from which the two versions originate. Three-way merging deals with techniques that allow for information in the common ancestor to be used during the merge process. Orthogonal to the classification of two-way and three-way merging, software merging techniques can be further classified as textual, syntactic, semantic, and structural. Textual merging techniques consider source code files purely as text files, and merging takes the form of appropriately merging source code files line by line (line-based merging). Syntactic merging techniques omit unnecessary textual details between the two files that are to be merged, and it will consider that two artifacts cannot be merged only if they produce a non-syntactically valid result. Depending on the data structures used to model the source code, syntactic-based merging can be further classified as tree-based or graph-based. Semantic merging deals with techniques that aim to detect situations where the resulting program is not semantically correct, such as having undeclared variables (static semantic conflicts) or is executing the wrong version of a function (run time semantic conflicts). Finally, structural merging deals with situations where one or both of the versions to be merged have been produced as a result of refactoring operations. The problem arises when the two versions are merged and we cannot decide which of the refactoring operations can be valid in the new merged version. Structural merging is an area for which more research is still needed. A complete survey of software merging techniques can be found in Ref. [108].

Software refactoring

Software refactoring refers to source code transformations that aim to improve the quality of the system without altering its behavior characteristics. Even though refactoring operations do not fix errors or add any new functionality, they facilitate the understandability, maintainability, and extensibility of the code.[109] Examples of refactoring include the transformation of source code within a block into a subroutine, moving a method or an attribute to a more appropriate class, or create more general types to take advantage of class inheritance in Object-Oriented systems.

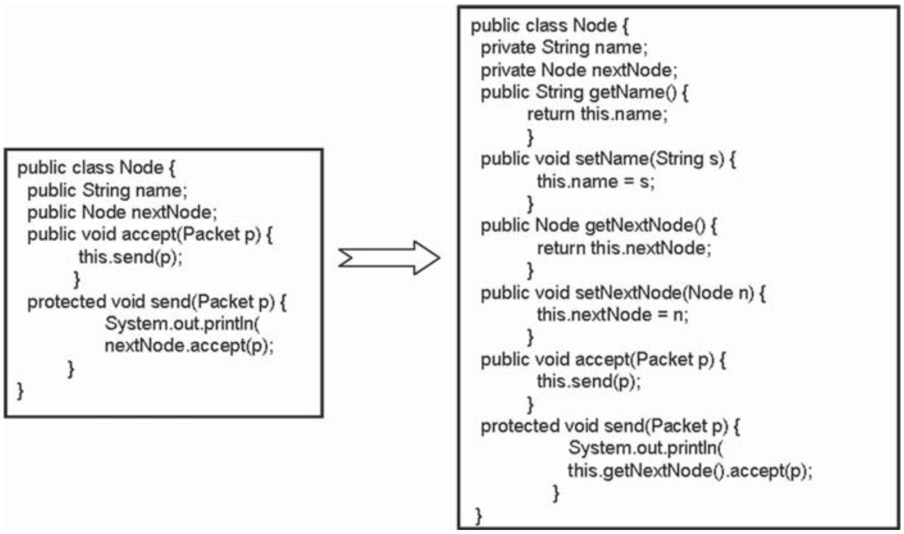


Fig. 9 Sample refactoring operation for the replacement of public data fields.

Fig. 9 illustrates a refactoring operation[109] where a public class attribute becomes private and accessor and mutator methods are added. At the source code level, refactoring takes the form of source code transformations so that design patterns can be introduced and antipatterns be removed. In the related software engineering literature, a number of different standard refactoring transformations and a number of to-be-avoided practices (antipatterns) have been proposed.[95] These catalogs discuss in detail the different types of transformations, their anticipated effects on system quality, and the conditions under which these refactoring operations can be applied or antipatterns can be recognized. Refactoring operations on the source code can be automated to a large extent.[62] These operations focus on raising the level of abstraction so that the system can be more extensible; decomposing the code into more manageable logical units so that the system components and code can be more maintainable and reusable; and improving coding standards (e.g., variable renaming, moving methods, and altering hierarchies) so that the system is easier to understand and modify.

Refactoring operations do not only apply at the source code level of a system but may also apply at the business, architecture, and data levels. At the business level, refactoring takes the form of altering the workflows or business processes in order to introduce proven patterns than can be used to increase throughput, decrease cost, or positively affect KPIs of interest.[110] At the architecture level, refactoring takes the form of component restructuring or the introduction of new components or the integration of the old system with other applications. Usually the objective is to enhance some non-functional and quality characteristics of a system. For example, the objective of architecture refactoring may be to make the system more secure, extensible, robust, or maintainable. At the data level, refactoring operations aim to reorganize the data schema of an application so that the data schema can be more maintainable and more extensible.

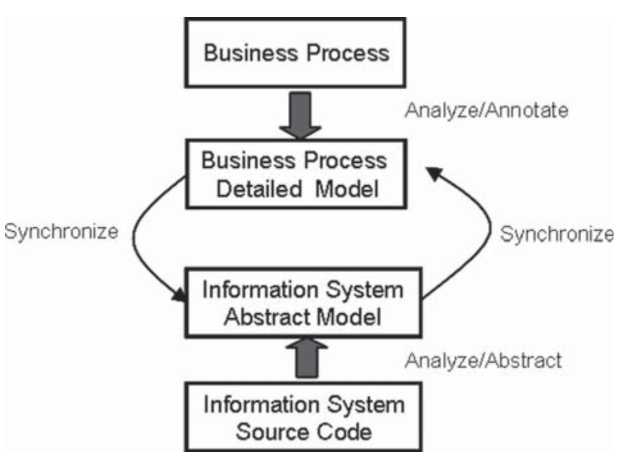
Adaptive Maintenance

Adaptive maintenance techniques allow for the migration, porting, and integration of software systems to new platforms, languages, environments, and third-party applications. Adaptive maintenance may take several forms. In this section, we discuss techniques that deal with three of the most often utilized maintenance scenarios, namely, synchronizing business processes and workflows with run time applications that implement these processes and workflows, migrating software systems to Object-Oriented, Network-Centric, and Service-Oriented environments, and finally migrating legacy components to new languages. Adaptive maintenance is an important part of the software life cycle as it allows for the system to remain operational when the underlying platforms or the operating environments change. A typical example of adaptive maintenance is when parts of the functionality of a legacy system need to be exposed as services (e.g., Web services) to other applications or users. The sections below discuss some of the most often occurring scenarios and techniques of adaptive maintenance.

Synchronizing business processes with run time applications

Software evolves in iterative and incremental steps from its inception to its retirement. The evolution includes change of artifacts at different levels of abstraction, from very abstract ones, such as business process specifications, to very concrete ones, such as source code. A perfective or corrective maintenance operation such as the addition of a new class could directly affect architecture and design models, and through ripple effects, even the requirements specification and the business process models of this system.[111] If each model change is not propagated to all affected models and if changes that are performed in parallel are not coordinated, consistency among artifacts is lost and semantic drift is created.[112–114] In this context, adaptive maintenance may take the form of propagating and synchronizing software models that pertain to different levels of abstraction and used by different stakeholders. This problem is referred to as model synchronization or model coevolution.[115] The solution here is to analyze the code so that an activity type of diagram can be compiled.[116] Such an activity type of diagram will identify for each use case of the system, the related operations, the sequence, and the conditions under which these operations are invoked and the data exchanged between these operations. These models can be created by static code analysis, dynamic analysis, or a combination of both.[117] On the other hand, business processes that implement specific system scenarios should be analyzed and annotated with information regarding the type of data exchanged between the various business process steps; the roles and constraints of each process step; and task descriptions. The intuition behind the analysis and abstraction of the source code with the concurrent annotation of the business processes is to bring source code models and business process models closer so that they can be compared and reconciled. Fig. 10 illustrates the concept of bridging the conceptual gap between high-level business process models and the system's source code. This convergence allows for comparisons to be applied in a more efficient and meaningful manner.

Reconciling annotated business process models with source code models takes the form of identifying differences and similarities between these models. In this respect a number of techniques have been proposed that fall in the general area of model dependence extraction. These techniques include formal concept analysis (FCA) and feature modeling.[118,119]



Q18 **Fig. 10** Bridging the conceptual gap between business processes and source code.

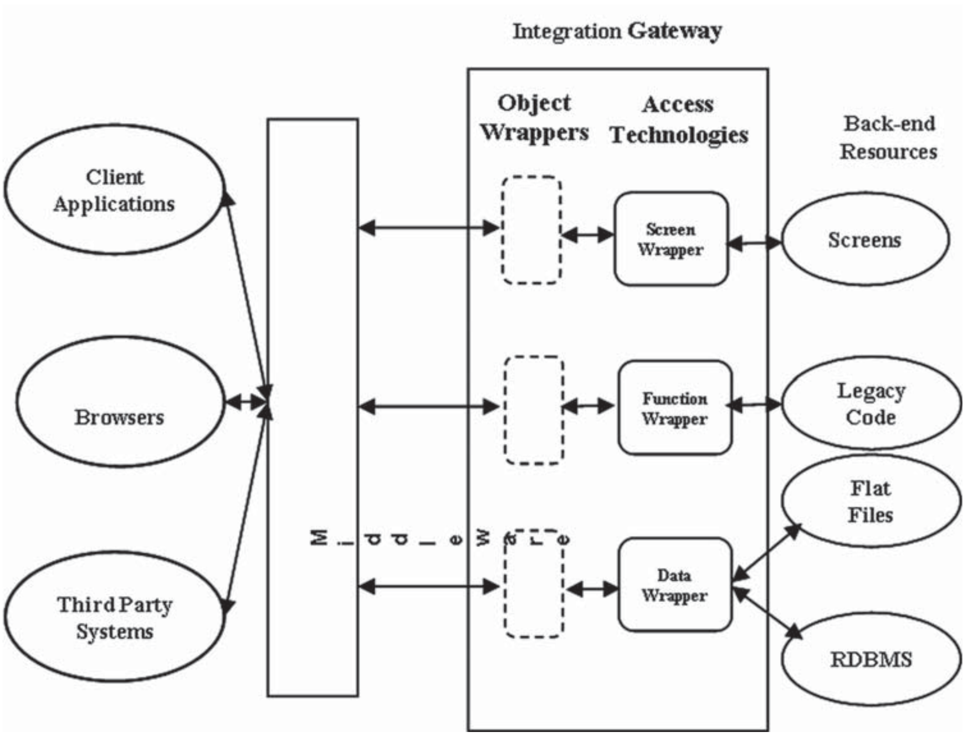
Migrating to Network-Centric Service-Oriented environments

The rapid development of new technologies in the area of Object-Oriented systems and Service-Oriented computing over the past few years necessitated the migration and porting of many corporate legacy applications to Network-Centric environments. This migration may take several forms. One form is wrapping, where legacy components are interfaced or wrapped with components that are responsible for exposing the original legacy interfaces to third-party components over standard application protocols such as http, SOAP, or RMI. Another form is the utilization of a framework such a Object Request Broker (ORB) to allow invocations of native legacy code from remote components written even in a different language than the legacy component being invoked.

Wrapping is a technique based on the creation of new components that provide bidirectional communication between the legacy component and third-party remote components. In one direction these components offer globally visible interfaces to third-party external client components. These globally visible interfaces implement native calls to the legacy component that is being invoked. In this respect, the clients do not need to know how the legacy component is invoked, and the only information they need to know is the visible interface offered by the wrapper component. Furthermore, these wrapper components offer through a specific application and transport layer protocol results back to the requesting client. The migration through wrapping can be achieved through different strategies.[102] These include the use of middleware frameworks such as Java Enterprise Edition JEE5 and Enterprise Java Beans, the use of Extensible Markup Language (XML) wrappers, and the use of proxy components to wrap components and dispatch client's requests to native legacy system calls. The intuition behind the wrapping approach is that the legacy components are minimally altered if any way at all. The wrapping approach provides a reduced migration risk and effort. On the other hand, legacy components are treated often as "black boxes" and there is no deeper understanding of the inner workings of the components to facilitate future evolution. Fig. 11 illustrates such a wrapping scenario for the exposure of legacy code, User Interface (UI) screens, and legacy data as service resources, to client and third-party applications.

In this context, we can differentiate between three major strategy types for migrating a legacy component to a Network-Centric environment. These can be qualified as the black box, the gray box, and the white box strategy.

In the black box strategy, we have access only to the UI of the system. Techniques like screen scraping or dialogue tracing can be used to provide a façade component (the wrapper) that invokes the same UI entities through programmatic and not through user-driven manual means.[120] In this approach, we do not have any access to the code, and we



Q18 **Fig. 11** System integration using wrapping technology.

focus only on exposing the UI functionality through such a façade component that simulates the original UI. The benefit on this approach is that it does not require any analysis of the source code of the system and therefore it is easier, faster, and most economic. The drawback of this approach is that any problems of the legacy system remain hidden and many consider it as an approach where sooner or later any quality problems of the legacy system will eventually surface and the source code analysis will be inevitable. The intuition behind this strategy is that it requires minimal access to the legacy component and has the potential to minimize cost and effort during the migration process.

The gray box strategy assumes that we have some minimal access to the source code of the system. The strategy utilizes an approach that has three major steps. In the first step the system is analyzed using static and dynamic analysis techniques so that the system is broken down to a collection of components that offer specific functionalities.[121–123] The second step is to identify the dependencies of each component with all other components that make up the system, and model the signatures of the interfaces of the components we want to expose to third-party clients. The third step is to create wrapper components that allow for the remote client components to access the legacy components.[124] The benefit of the approach is that it decomposes the legacy system and provides a level of understanding of its inner workings. Another benefit is that the extracted components can be considered as reusable assets that can be utilized in other applications as well. The drawback of the approach is that it requires some level of analysis of the source code and the extraction of dependencies between the extracted components. These dependencies may be very obscure and difficult to extract and therefore, if these dependencies are not fully revealed, there is the possibility that there may be unforeseen side effects when a component is invoked by a third-party external client component. An example could be that external client components may be accessing secure internal databases or updating a database without the proper locking mechanisms. The intuition behind this strategy is to identify reusable assets that can be used as standalone components to form assembly elements of new systems and applications in an organization. In this respect, componentization and access of legacy functionality can be done at a finer level of granularity.

The white box strategy is based on the assumption that we have full access to the source code and it is also based on three steps. In the first step we analyze the legacy source code and we extract an object model. Complex data structures become class types and functions that utilize such data types become methods to the classes generated by the data types. This is a complex process and is referred to as *objectification* of the legacy code. The second step is the automatic generation of Object-Oriented source code from the extracted object model. This is a step that can be automated especially if the target Object-Oriented language shares common features with the legacy source language (e.g., C to C++). The automatic generation of the target Object-Oriented source code may be more challenging if the source and target language do not share many common features (e.g., Fortran to Java). The third step is the use of a middleware framework or the creation of wrapper components that allow the access of methods of individual objects from remote clients. The intuition behind the white box approach is that migration to Network-Centric environments is best done when the source code has been analyzed and restructured so that legacy functionality is encapsulated in classes and selectively exposed in the form of methods. In this respect, the benefit of the white box approach is that it decomposes the legacy application in a very fine level of granularity, thus extending the understanding we have on the business logic encapsulated in the system. This also allows for individual classes and consequently individual objects to be used from external applications as reusable assets. The drawback of the approach is that it requires significant investment in time, effort, and funds to extract an accurate object model from the legacy system and generate new Object-Oriented code. Another drawback is that the original design of the system may be lost and that may create maintainability problems with the staff that is familiar with the old original structure of the system.[53] Fig. 12 illustrates sample transformations to migrate procedural C code to Object-Oriented C++ code. The first and second transformations illustrate how type definitions and structures can be mapped into classes. The third transformation illustrates how C functions can be mapped to methods. In this particular example the return data type and the parameter data type of the original C function are considered to allocate the function as a method to the class that stems from this data type (second transformation).

Once the functionality of a legacy component has become available to remote clients as a service, we can then utilize a number of architectural styles and frameworks to integrate it with other systems and components.[125] The most relevant architectural styles are the layered architecture style and in particular the three-tier architectural style and the implicit invocation style such as the Model-View-Controller and the Event Driven Architecture (EDA) style.

Regardless of the strategy used, these recent advances in software engineering technology allow for various levels of wrapping and mediation to take place in large systems, and middleware frameworks also allow for the seamless access of legacy components or objects in a secure way and with robust transaction management policies from remote clients. In this way, the level of granularity can be arbitrarily raised and wrappers can be further wrapped, components can interface with other components, applications, and databases through mediators, and so on, thus building what is known as Systems of Systems (SoS) or Ultra Large Scale Systems (ULS).[126] ESB technology and Publish/Subscribe protocols can be used to create such large corporate systems by integrating diverse applications, components, and data sources in one seamless environment.[103,127] Fig. 13 illustrates an example three-tier architecture (client side, server side, legacy systems) that is based on the JEE framework and can be used to expose legacy components as services to remote clients.

Migrating to new languages

In many occasions, adaptive maintenance takes the form of migration of a system to a new programming language.

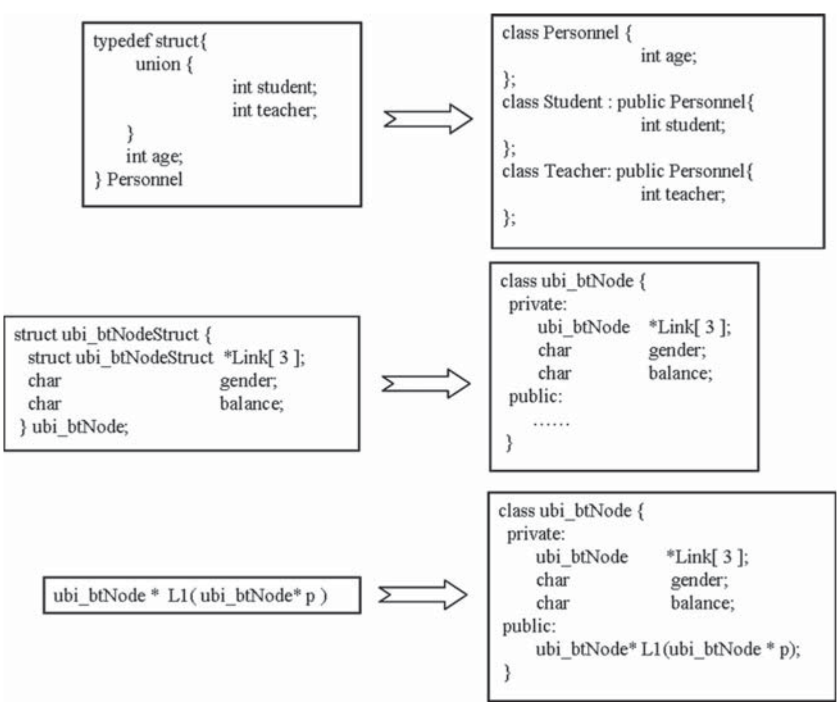


Fig. 12 Sample migration from procedural to Object-Oriented code. Q18

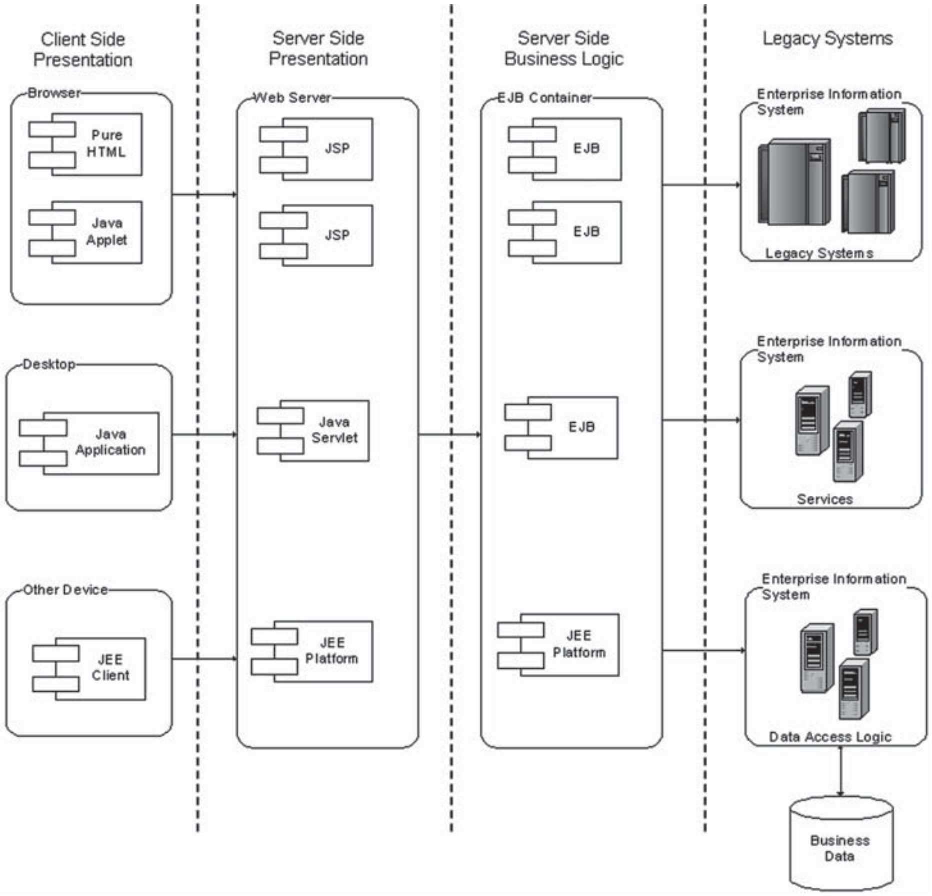


Fig. 13 Block diagram of a typical three-tier Network-Centric architecture. Q18

This may be required for several reasons. The most common reasons include the lack of development, debugging, and testing tools for the old language, the lack of a compiler that runs on modern platforms for the old language, the lack of common libraries for the old language, or even the lack of developers who are familiar with the old language. Migration to a new language usually takes the form of transliteration where we transform a system written in one language to another but without changing the underlying programming paradigm (e.g., from Fortran to C). Less frequently, migration takes the form of transforming a system from one language to another new language in a new programming paradigm (e.g., from imperative to Object-Oriented such as from C to C++). Regardless of the form and type of migration, at the core there is what is referred to as program transformation technology. In this section we focus on classifying and discussing a number of generic source code transformation techniques that fall in two main categories of transformations. The first category is referred to as Model-to-Text transformations. The second is referred to as Model-to-Model transformations.[128]

Model-to-Text-based migration is based on two steps. In the first step, a model of the source code is built. This model is usually the Annotated AST of the source code being migrated. The model is usually represented as a complex data structure in dynamic memory and can be persistently stored if needed in an Object-Oriented or relational database. In the second step, a transformer guided by a transformation control strategy uses the model to generate the text of the source code of the new migrant system. In the related literature, the approaches to generate the target program's source code text have been classified into two categories, namely, visitor-based approaches and template-based approaches. In the visitor-based approaches, a visitor mechanism (e.g., the visitor design pattern) is used to traverse the source model (i.e., the Annotated AST of the source system) and to generate the appropriate target text (i.e., the target source code for the migrant system). The template-based approach is based on collections of templates that represent the target text containing splices of metacode to access the source model and to perform code selection and iterative expansion. These templates form the right-hand side (RHS) of transformation rules. The left-hand side (LHS) of these rules implements logic that accesses the source model and provides data to variables and metacode in the RHS templates. The result of the rule application is the instantiation of the RHS templates to form syntactically valid text that corresponds to the source code of the target migrant system. The intuition behind this approach is to avoid maintaining more than one abstraction models (i.e., one source model of the original system and one abstract model of the migrant system) and be able to immediately generate source code text directly from the abstraction models that represent and denote the original legacy system. In this respect, the benefits of this approach are twofold. First, it does not require the additional complexity of maintaining one source and one target model, thus eliminating the need to define and validate such a target model. Second, it allows for the direct validation of the transformation rules by inspecting the source code as this is directly emitted from the transformer.

Model-to-Model approaches are based on transformers that are applied to a source model (e.g., the Annotated AST) and yield another model that is then used to generate code for the new migrant system. The target model is usually the Annotated AST of the target system. A pretty printer can then be used to generate the target source code. In this respect, Model-to-Model transformations are in fact tree transformations. The frameworks that support such Model-to-Model transformations fall into four main categories, namely, direct manipulation, staged structure-driven, template-based, and graph transformation-based frameworks. The direct manipulation Model-to-Model transformation frameworks offer an application programming interface (API) and a corresponding programming language to manipulate the source models through custom user-defined transformation programs written in the programming language offered by the framework (Lisp, Java, etc.). The staged structure-driven frameworks operate in two steps. In the first step, the skeleton of the target model is created. In the second step, the skeleton target model is filled or instantiated by setting values for attributes and linking various model references. The template-based frameworks utilize template models that contain annotations in the form of metacode or constraints. An application program that implements a control strategy instantiates the template model to yield a concrete model that in turn is used to generate the target source code. The graph transformation-based approach utilizes a graph transformation language to transform the source model to a target model. Triple Graph Grammars and Attributed Graph Grammars constitute the theoretical framework for these types of transformers. The intuition behind the Model-to-Model approaches is that it allows for different versions of the same target system to be generated from a common target model. This common target model is generated by the source model that is an abstraction of the original source code. Once such a target model is created, customized fine-tuned transformers can be used to generate different versions of the target source code (e.g., from PL/I to Fortran 95, or to Fortran 2003).

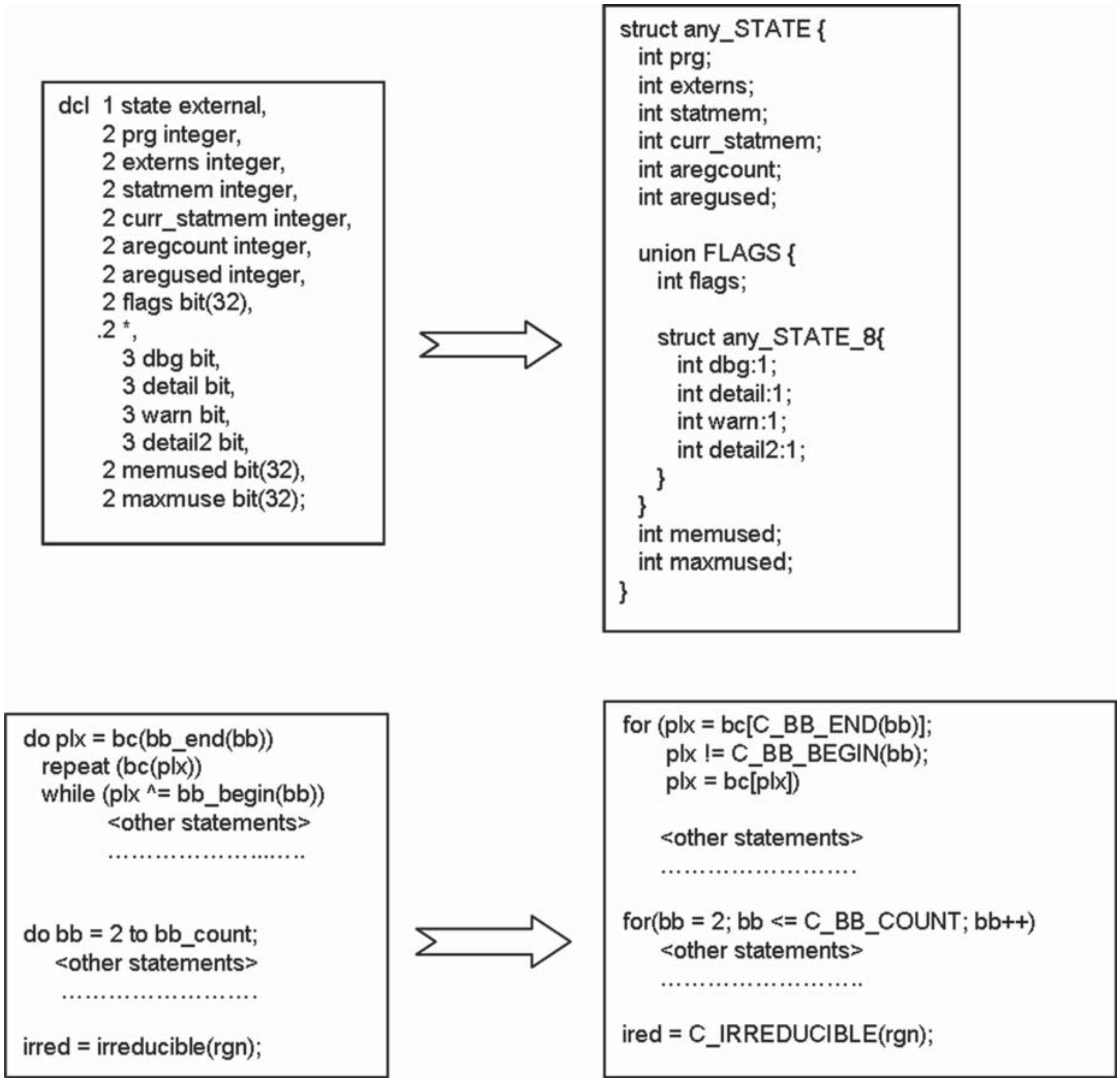
Source code migration has been successfully used for migrating large programs written from one language version to another version, programs written in imperative languages (such as PL/I) to other imperative languages such as C, or programs written from one language in one programming paradigm to another language in another programming paradigm such as C to Java.[129, 130] Fig. 14 below illustrates two examples of PL/I-like code segments automatically migrated to equivalent C code. Equivalence here is evaluated by ensuring that the external system behavior of the migrant system is the same as the one of the original system.

Concluding this section, it is important to mention two points of interest. One is that migration and transliteration do not need to be complete (i.e., produce a fully working system automatically) for the migration to be successful. The strength of the automated approaches is that they allow for the mechanic, uniform, consistent, and fast transformation of the bulk of the system (e.g., 95% of the code). There could be a small part of the generated target system code that is either not syntactically correct or has some semantic errors. These can be revealed by testing (unit, system, functional testing) and the assessment of the reliability of the system using appropriate reliability growth models. Automatic transliterators can generate more that 95% of the original code to the new code. This is a major step as it reduces the migration time and possible manual rewriting errors. The second point has to do with the need for rerresting the new migrant system to make sure that the functional and non-functional requirements for the migrant system indeed hold. Experience shows that this testing phase is probably the most time consuming and expensive in the whole process and should not be underestimated.

TOOLS, FRAMEWORKS, AND PROCESSES

Tools

Software maintenance for large systems without the use of specialized tools would not be possible due to the complexity of the task and due to the size of the source code usually involved. Furthermore, many maintenance tasks would be error prone if performed manually. Tools automate many tedious and repetitive tasks and therefore eliminate the problem or human errors. Examples include repeated transformations, metrics calculation, as well as the extraction of data and control flow dependencies between source code statements or between components. In this respect, there are many tools that have been proposed by the industry and the academia. The implementation view of the architecture of these tools may vary. For example, these may have been implemented using a combination of a pipe and filter architectural style, blackboard style, or an implicit invocation style.[103] However, regardless of the implementation view of the architecture, most tools share a common conceptual architecture. This conceptual architecture is composed of components such as the front-end component that allows for parsing the code; the source code repository component that stores in dynamic memory or in persistent storage the source code model (i.e., the AST); and a number of plug-in components that allow for the analysis and

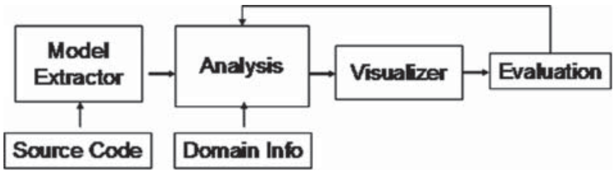


Q18 **Fig. 14** Transliteration example of PL/I-like source code to C source code.

manipulation of the source code model. Fig. 15 illustrates the conceptual architecture of most source code analysis tools. Examples of tools that have been extensively used in the industry and academia include the Rigi[34] and Landscape,[131] and tools that have been developed as research prototypes but have also been extensively used in the industry are the CIA,[35] Genoa,[132] TXL,[133] Columbus,[134] and Bauhaus tools.[135] Finally, in the area of visualization, tools such as Dotty, Shrimp, and Landscape have been extensively used for visualizing various software models. These tools provide a basic list of what is available for software maintenance. Elaborate lists of tools and sites related to software maintenance can also be found in many research sites.[136] Also, prototype tools are regularly demonstrated at related international IEEE and ACM conferences such as the IEEE/ACM International Conference on Software Engineering (ICSE), the IEEE International Conference on Software Maintenance (ICSM), the IEEE Working Conference on Reverse Engineering (WCRE), the IEEE International Conference on Program Comprehension (ICPC), and the IEEE Conference on Software Maintenance and Reengineering (CSMR).

Frameworks

So far, there have been several frameworks and metamodeling languages proposed by different communities. The OMG in a breakthrough proposal defined a core metamodeling language called Meta Object Facility (MOF) that is based on modeling domains and specifying schemas using classes, associations, and inheritance. The simplicity and



Q18 **Fig. 15** Conceptual architecture of software analysis tools.

power of this language allows for software engineers to model the schema of various other modeling languages including UML itself. The interesting dimension is that once a schema has been designed as a MOF model, OMG has defined a standard, called eXtensible Metadata Interchange (XMI), which allows for such MOF models to be represented as XML schemas. This has interesting implications because once a MOF model is instantiated to form a concrete model, then this concrete model can be represented again through XMI as an XML document that complies with the XML schema that stems from the domain model's schema. Once this XML document that describes the concrete model is available, it can then be fed along with its corresponding XML schema to frameworks such as the Eclipse Modeling Framework (EMF). The EMF framework then automatically generates a data structure, representing as Java objects in dynamic memory the XML document that corresponds to the concrete MOF model and provides an API for accessing and manipulating these objects. Also, transformers can be used to transform one EMF model that is compliant to one schema to another EMF model that is compliant to another schema. Such transformers include Atlas Transformation Language (ATL), Attribute-Graph-Grammar System (AGG), and MTF. The impact of the above is that it allows for the development of tools that manipulate models that now are stored as objects in dynamic memory and not as just plain graphs and images. These tools can perform analysis of models, transformations of models, and generation of source code for a new migrant system.

In EMF, MOF follows a multilayer architecture by gradually evolving and extending the metamodeling elements in different levels. An advantage of such approach is that it makes it easier to select a subset of the language for applications that do not require the entire features of the language. Essential MOF (EMOF) is extended by Complete MOF (CMOF), which provides a more comprehensive set of modeling features. EMF is another such modeling facility proposed by the Eclipse community. With respect to transformation languages, Query-View-Transformation, also referred as QVT, provides means for the algorithmic specification of model transformations, pattern matching, traceability, and support for incremental and bidirectional model updates.

As an example of how the above technologies relate to software maintenance, consider the following scenario where a software engineer defines a domain model (a schema) for the AST of a given language (see Program Representation). This schema can be then represented as a Q9 MOF model. A parser can be used to parse the source code of the system being maintained and instantiate this MOF model to create a concrete AST for a given code fragment or the complete system in EMF. This concrete model is essentially a model of the source code of the system and can be analyzed (e.g., compute metrics, identify refactoring operations, and identify dead code) or transformed to yield a new updated source code model. Such model transformation can be achieved using a transformation language such as QVT, ATL, or AGG. This type of maintenance relates to Model-to-Text and Model-to-Model approaches described earlier.

Finally, some commercial tools that facilitate software development and maintenance include the IBM Rational Software Architect, the Poseidon UML framework, and the StarUML framework, where software engineers can use these to perform reverse or forward engineering. In the reverse engineering mode one can extract class diagrams from existing source code. In the forward engineering mode one could export source code models (use cases, class diagrams, sequence diagrams, ASTs) through the XMI standard to EMF compliant models for further processing and transformation from within the EMF environment.

Processes

Software maintenance is a complex, expensive, and time-consuming task that encompasses several risks. As such, it must be planned and organized carefully in a methodological and structured way. At the beginning of this entry, we discussed generic processes for software maintenance. Even though these processes are helpful for understanding the issues and for planning the activities related to software maintenance tasks, they do not provide much of practical benefit unless they are associated with some concrete application guidelines. These guidelines are provided in the form of *handbooks* and *methodologies* that allow software maintainers to select, plan, design, implement, and evaluate various maintenance activities. In this context, we discuss two of the most frequently used process methodologies for software maintenance, namely, the Software Reengineering Assessment (SRA) and the Service Migration and Reuse Technique (SMART).

The SRA methodology is a concrete process methodology proposed by the U.S. Air Force Software Technology Group. The SRA assumes a limited budget and availability of resources as compared to the large software portfolio of an organization. The objective of SRA is to provide a structured methodology for assessing maintenance options and proposing alternatives. SRA is structured around three phases: the technical assessment, the economic assessment, and the management evaluation. In the first phase, the software portfolio of an organization is assessed from a

technical standpoint. For each system, different maintenance techniques are considered and evaluated with respect to its technical merit, impact, risk, and feasibility. The result of the technical assessment phase is a list of systems that must be considered for maintenance along with a list of maintenance options for each such system. The economic assessment evaluates each maintenance option proposed by the technical assessment from a financial point of view. The options are ranked based on their
Q10 corresponding Benefit to Investment Ratio and their NPV. The systems and the corresponding maintenance options with the highest NPV and BIR indicators are evaluated jointly by the technical assessment team, the economic analysis team, and the management team.[24] The systems and the maintenance options that are selected from this joint evaluation are the ones that will be chosen for maintenance.

Another methodology is the SMART that has been developed by the Software Engineering Institute at Carnegie Mellon University. SMART is a technique that can be used to make initial decisions about the feasibility of reusing legacy components as services within an SOA environment. Fig. 16 illustrates the basic tasks of the SMART process.[137] SMART operates in three phases. The first phase is the planning phase where the methodology provides a structured way to gather a wide range of information about legacy components, the target SOA, and potential services. The information is collected using a methodology referred to as the Service Migration Interview Guide (SMIG). The SMIG directs information gathering for establishing the migration context; for describing and assessing the existing capability, that is, the data about the legacy components from the identified stakeholders; and for describing the target SOA state, that is, the data and information for the target SOA system. The second phase is the GAP analysis to understand the differences between the current state of the system and the future planned state of the new SOA system. Once this analysis has been performed, the methodology proceeds in the third phase. The third phase is the strategy determination phase where the methodology assists software maintainers to produce a service migration strategy as the primary product as well as other information related to the state of the legacy system. SMART[26] is also implemented as a tool that allows developers to gather and organize the data required so that informed decisions can be made.

EMERGING TRENDS

As software technologies, programming languages, utility frameworks, computing platforms, and environments evolve at a rapid pace, software maintenance practices must evolve too. In this section, we discuss some of the emerging trends in software engineering that have a direct impact on software maintenance practices. We classify these emerging trends in eight categories, namely, processes, modeling, transformations, evaluation, software visualization, open source, mining software repositories, and SoS.

Processes: Until recently, the majority of software development processes focused on collections of specifications and plans that must be completed before any design or implementation activity starts. These are quite rigid models since they do not allow for any prototypes or implementations to be shared with the end user, until the final release of the system for acceptance testing. As an answer to this problem, over the past few years we see an emerging trend for a new generation of process models that have a profound effect on software maintenance. These models are referred to as *agile processes* that are iterative and incremental in the sense that software is developed and maintained in iterations (versions) considering new functional and non-functional requirements incrementally in each new version. Every iteration cycle is worked on by a team through a full software development cycle, including planning, requirements analysis, design, coding, unit testing, integration testing, and system and acceptance testing when a working version is to be demonstrated to stakeholders. These processes consider source code as the primary artifact and therefore software maintenance becomes a de facto part of the development process and not only an

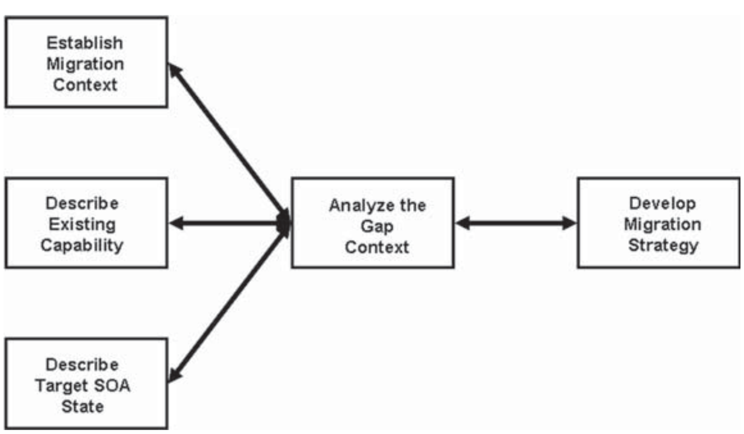


Fig. 16 Basic activities of the Service Migration and Reuse Technique process. Q18

activity that applies only on systems that have been released and are already in operation. Agile methods emphasize face-to-face communication over written documents and are use case-driven and architecture-centric. They also allow for risk avoidance for the early resolution of problems. Agile development has been widely documented in dedicated IEEE/ACM conferences such as Extreme Programming (XP) and Agile. To date, we do not have any concrete frameworks to assess how these agile processes affect system quality or how these processes can be further enhanced to become more efficient for software maintainability purposes. Some of the emerging processes that are expected to have an impact on software maintenance include Agile Unified Process (AUP) XP, Feature Driven Development (FDD), and Open Unified Process (OpenUP).[138]

Modeling: So far, software representation models were based on proprietary metalanguages and custom-made schemas. This created a problem of building maintenance tools that can interoperate. However, over the past few years we see a standardization of the metamodeling languages used to represent software artifacts. The metamodeling language that emerged as a de facto standard is the MOF. Combined with new transformation technologies that aim to manipulate MOF-based models in a programmatic manner, we see an emerging trend where software artifacts such as low-level designs and source code are created automatically as a result of transforming higher level of abstraction models. This has a profound effect on software maintenance as we see gradually moving from the maintenance of source code artifacts to the maintenance of software models with emphasis on techniques to support model synchronization and model coevolution, that is, how models can remain consistent when one of them changes due to maintenance activities. An area that deals with the issues of automatic code generation and the maintenance of software models is MDE that is currently at the forefront of software engineering research.

MDE is a software development methodology that focuses on the creation, interpretation, and transformation of models that represent software artifacts such as use cases, requirements, design specifications, and test models. These models can be used to automatically or semiautomatically generate infrastructure and application source code for a system. MDE creates unique challenges and opportunities for software development and software maintenance. Some of the challenges relate to devising well-defined representations of models and devising transformation techniques for the generation of new models and source code from existing models. In this context of MDE, it is also needed to devise techniques that allow for all these models to remain consistent and synchronized when one or more of them changes due to maintenance activities. Even though MDE may look complicated at first, it promises significant gains in software development productivity, especially when software development relates to enterprise software systems that utilize standard architectural patterns and middleware technologies. Furthermore, from the software maintenance standpoint, MDE offers the possibility of keeping track of the dependencies and the associations between various software artifact models (e.g., requirements, architecture, and low-level design) with the source code of a software system.

Transformations: At the heart of most maintenance operations is the concept of software transformations.[139] These transformations may take the form of modifying the system so that it can be ported to a new environment, or of altering the system to add new functionality, or of refactoring the system so that it can be more maintainable. In the past, software transformations were implemented using proprietary frameworks. Furthermore, transformations could not be easily reused among even similar applications. Over the past few years we observe an emerging trend toward the standardization of such transformation languages and transformation frameworks. These transformation languages and frameworks focus on manipulating MOF models and are based on graph and tree transformations. Examples of such emerging languages and frameworks for model transformation include QVT, the ATL, and the AGG.

Evaluation: As discussed in the previous sections, the evaluation of software maintenance operations falls into two main categories, namely, technical and financial evaluation. In this respect, an emerging trend in evaluating software maintenance and evolution choices from a technical perspective is techniques to establish traceability links that allow for what-if type of impact analysis where the effects of a transformation or a maintenance operation can be better estimated before the operation is applied.[140] Similarly, there are a couple of emerging trends for evaluating alternative software maintenance and evolution scenario and choices from a financial perspective. These include the use of future options valuation and the valuation of customer loyalty that can be gained by software customization operations.

Software visualization: Software systems grow constantly in size and complexity. Even though we have elaborate tools to analyze large software systems, the results produced from these tools would be of limited value if we could not have techniques to present these results to software engineers. For humans, one of the best ways to understand and analyze information is through images. In this respect, software visualization is an emerging field in the area of software maintenance. Software visualization techniques are not new.[141] However, as the complexity of software systems grows, the need for efficient visualization techniques grows even bigger.[142–146] Some of the emerging challenges in this field include dealing with complexity of data, enhancing the usability of visualization tools, and developing collaborative visualization tools. A thorough discussion on research challenges in software visualization can be found in Ref. [147].

Open source: Over the past decade we have witnessed the revolution of open source system development. During this decade we have also gathered valuable information on open source development process models, on the impact some design decisions have on open source quality, and on the evolution history and politics of these systems. Furthermore, we have gathered detailed reports related to faults, fixes, and testing processes. An emerging trend in the area of software maintenance is how to leverage the information gathered from open source system development and maintenance to apply other non-open source projects and how to decide what strategy is best to use for software maintenance. The software engineering community believes that there is still a lot to be learnt from these systems.[148–151]

Mining software repositories: A software system is composed of artifacts that are far richer than just its source code. These artifacts include of course the source code, but also requirements and design documents, evolution data, bug reports, test suites, deployment logs, and archived communications. Over the past decade, and for the purposes of software maintenance and evolution, the research community developed robust technologies to model, store, and process these artifacts in specialized repositories called software repositories.[152–154] As these repositories grew in size and complexity, so did the need to develop efficient and tractable mining and retrieval techniques. Therefore, an emerging field in the area of software maintenance is mining software repositories. In this respect, there are some challenges to address in this field. These include dealing with software repository complexity and diversity of artifacts these repositories store, dealing with consistency of data, dealing with the extraction of high-quality data in a tractable manner, and dealing with unstructured data. A comprehensive discussion on mining software repositories can be found in Ref. [155].

Systems of Systems: This term refers to a collection of task-oriented, large-scale, concurrent and distributed systems that integrate and collaborate to form a new, more complex "metasystem," which offers more functionality and performance than simply the sum of the constituent systems. These SoS, also referred to as ULS, started to appear in the areas of Command, Control, Computers, Communication and Information Intelligence (C4I); Surveillance and Reconnaissance (ISR); as well as in banking and finance. These systems pose unique challenges in the area of software maintenance. Emerging trends in this area include the use of unified models for the representation of these systems and their dependencies; techniques for multilanguage multiplatform system analysis and maintenance; techniques for monitoring RCA and diagnostics of SoS; and techniques that assess the alignment of run time IT infrastructure with business processes. The reader can also refer to a detailed discussion on the emerging area of SoS and ULS that can be found in Ref. [126].

CONCLUDING THOUGHTS

Software maintenance is a very important part of the software development life cycle. In this entry, we discussed a collection of topics that are indicative of this important area of software engineering. Software maintenance is an ever-evolving field and we anticipate it will generate significant challenges, especially as it has started being considered, through agile processes, as an integral part of the Greenfield development processes. As such, we expect to see software maintenance techniques being even more prevalent in the years to come, especially software maintenance techniques to support the development and operations of large-scale multilanguage, multiplatform Service-Oriented systems. We encourage the readers to delve into the references of this entry and investigate further specific technical details of their interest.

REFERENCES Q11

1. Canning, R. That maintenance 'iceberg'. EDP Analyzer **1972**, *10* (10).
2. Swanson, E.B. The dimensions of maintenance. In Proceedings of the 2nd International Conference on Software Engineering, San Francisco, 1976.
3. The Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology, 1990. IEEE Standard 610.12-1990.
4. The Institute of Electrical and Electronics Engineers. IEEE Standard for Software Maintenance, 1998. IEEE Standard 1219–1998.
5. Hunt, B.; Turner, B.; McRitchie, K. Software maintenance implications on cost and schedule. In Proceedings of the IEEE Aerospace Conference, 2008.
6. Chapin, N. Do we know what preventive maintenance is? In Proceedings of 2000 IEEE International Conference on Software Maintenance, 2000.
7. Kajko-Mattsson, M. Preventive maintenance! Do we know what it is? In Proceedings of the IEEE International Conference on Software Maintenance, 2000. Q17
12. Lehman, M.M.; Fernandez-Ramil, J. Software evolution and feedback: theory and practice. In *Software Evolution*; Madhavji, N.H., Fernandez-Ramil, J., Perry, D.E., Eds.; Wiley, 2006.
13. Lehman, M.M.; Belady, L.A., Eds.; *Program Evolution: Processes of Software Change*; Academic Press, 1985.
14. Lehman, M.M.; Ramil, J.F.; Wernick, P.D.; Perry, D.E.; Turski, W.M. Metrics and laws of software evolution—the nineties view. In Proceedings of the 4th International Software Metrics Symposium, 1997.
15. Bruegge, B.; Dutoit, A.H. *Object-Oriented Software Engineering Using UML, Patterns and Java*; Prentice Hall, 2004.
16. Schmidt, D. Model driven engineering. IEEE Comput. 2006. Q15
17. Bennettand, K.; Rajlich, V. Software maintenance and evolution: a roadmap. In Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000.

18. Finnigan, P. et al. The software bookshelf. IBM Syst. J. **1997** *36* (4).
19. Sousa, M.J.C.; Moreira, H.M. A survey on the software maintenance process. In Proceedings of the IEEE Conference of Software Maintenance, 1998.
20. Seaman, C.B. The information gathering strategies of software maintainers. In Proceedings of IEEE Conference on Software Maintenance, 2002.
21. Rook, P. *Software Reliability Handbook*; Centre for Software Reliability, Springer, 1990.
22. Sliwerski, J.; Zimmermann, T.; Zeller, A. HATARI: raising risk awareness. In Proceedings of 10th European Software Engineering Conference, 2005.
23. Brodie, M.L.; Stonebraker, M. *Migrating Legacy System*; Morgan Kaufmann: SanMateo, California, 1995.
24. Software Reengineering Assessment Handbook JLC-HDBK-SRAH Version 3.0, USAF Software Technology Support Center.
25. Arnold, R. *Software Reengineering*; IEEE Computer Society Press: Los Alamitos, CA, 1993.
26. Lewis, G.; Morris, E.; Smith, D.; O'Brien, L. Service-oriented migration and reuse technique (SMART). In Proceedings of IEEE Conference on Software Technology and Engineering Practice, 2005.
27. Cifuentes, C.; Gough, K.J. Decompilation of binary programs. J. Softw. Pract. Experience **1995**, *25* (7), 811–829.
28. Canfora, G.; Di Penta, M. New frontiers of reverse engineering. In International Conference on Software Engineering (ICSE' 07)—Future of Software Engineering Track (FOSE), 2007.
29. Darcy, D.P.; Palmer, J.W. The impact of modeling formalisms on software maintenance. IEEE Trans. Softw. Eng. **2006**, *53* (4).
30. Stoy, J. *Denotational Semantics: The Scott-Straehey Approach to Programming Language Theory*; MIT Press: Cambridge, MA, 1977.
31. Aceto, L.; Fokkink, W.J.; Verhoef, C. Structural operational semantics. In *Handbook of Process Algebra*; Bergstra, J.A., Ponse, A., Smolka, S.A., Eds.; Elsevier, 2001. Q12
32. Milner, R. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
33. Markosian, L.; Newcomb, P.; Brand, R.; Burson, S.; Kitzmiller, T. Using an enabling technology to reengineer legacy systems. Commun. ACM **1994**, *37* (5), 58–70.
34. Müller, H.A.; Klashinsky, K. Rigi a system for programming-in-the-large. In Proceedings of the 10th International Conference on Software Engineering, 1988.
35. Chen, Y.F.; Nishimoto, M.Y.; Ramamoorthy, C.V. The C information abstraction system. IEEE Trans. Softw. Eng. **1990**, *16* (3), 325–334.
36. Winter, A.; Kullbach, B.; Riediger, V. An overview of the GXL graph exchange language. In Proceedings of the Software Visualisation International Seminar, LNCS 2269, Dagstuhl Castle, Germany, May 2002. Springer-Verlag, 2002; 324–336.
37. Aho, A.V.; Sethi, R.; Ullman, J.D.; Lam, M.S. *Compilers: Principles, Techniques, and Tools*; Pearson Education, Inc., 2006.
38. Bell Canada. "DatrixTM Abstract Semantic Graph: Reference Manual v1.4 2000-05-01", 2000.
39. H. A. Müller, K. Wong, and S. R. Tilley. "Understanding software systems using reverse engineering technology." In the Proc. of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (AFCAS), 1994.
40. Holt, R.C. Structural manipulations of software architecture using Tarski relational algebra. In Proceedings of the Working Conference on Reverse Engineering, 1998.
41. Ducasse, S.; Lanza, M.; Tichelaar, S. Moose: an extensible language-independent environment for reengineering object-oriented systems. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET '00), 2000.
42. Holt, R.C.; Schurr, A.; Sim, S.E.; Winter, A. GXL: a graph-based standard exchange format for reengineering. J. Sci. Comput. Program. **2006**, *60* (2), 149–170.
43. http://www.omg.org/technology/documents/modernization speccatalog.htm Q14
44. Horwitz, S.; Reps, T.; Binkley, D. Interprocedural slicing using dependence graphs. ACM TOPLAS **1990**, *12* (1), 26–60.
45. Cifuentes, C.; Simon, D. Procedure abstraction recovery from binary code. In Proceedings of the IEEE on the 4th European Conference on Software Maintenance and Reengineering, 2000.
46. Zheng, J.; Williams, L.; Nagappan, N.; Snipes, W.; Hudepohl, J.P.; Vouk, M.A. On the value of static analysis for fault detection in software. IEEE Trans. Softw. Eng. **2006**, *32* (4).
47. Ritsch, H.; Sneed, H.M. Reverse engineering programs via dynamic analysis. In Proceedings of IEEE Working Conference on Reverse Engineering, 1993.
48. Salah, M.; Mancoridis, S.; Antoniol, G.; Di Penta, M. Scenario-driven dynamic analysis for comprehending large software systems. In Proceedings of IEEE Conference on Software Maintenance and Reengineering, 2006.
49. Safyallah, H.; Sartipi, K. Dynamic analysis of software systems using execution pattern mining. In Proceedings of IEEE Conference on Program Comprehension, 2006.
50. Sartipi, K. Software architecture recovery based on pattern matching. In Proceedings of IEEE Conference on Software Maintenance, 2003.
51. Canfora, G.; Cimitile, A.; Munro, M.; Tortorella, M. Experiments in identifying reusable abstract data types in program code. In Proceedings of IEEE Conference on Program Comprehension, 1993.
52. Yeh, A.S.; Harris, D.R.; Reubenstein, H.B. Recovering abstract data types and object instances from a conventional procedural language. In Proceedings of IEEE Working Conference on Reverse Engineering, 1995.
53. Zou, Y.; Kontogiannis, K. Migration to object oriented platforms: a state transformation approach. In Proceedings of IEEE Conference on Software Maintenance, 2002.
54. Gallagher, K.; Lyle, J. Using program slicing in software maintenance. IEEE Trans. Softw. Eng. **1991**, *17* (8), 751–761.
55. Kagdi, H.; Maletic, J.I.; Sutton, A. Context-free slicing of UML class models. In Proceedings of 21st International Conference on Software Maintenance ICSM, 2005.
56. Jorgensen, P.C. *Software Testing a Craftsman's Approach*; CRC Press, 2006.

57. Castelli. D. A strategy for reducing the effort for database schema maintenance. In Proceedings of IEEE Conference on Software Maintenance and Reengineering, 1998.
58. Muthanna, S.; Kontogiannis, K.; Ponnambalam, K.; Stacey, B. A maintainability model for industrial software systems using design level metrics. In Proceedings of IEEE Conference on Reverse Engineering, 2000.
59. Pfleeger, S.L.; Bohner, S.A. A framework for software maintenance metrics. In Proceedings of 6th International Conference on Software Maintenance, 1990.
61. Tran, J.B. Software Architecture Repair as a Form of Preventive Maintenance. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 1999.
62. Tahvildari, L.; Kontogiannis, K. A software transformation framework for quality-driven object-oriented re-engineering. In Proceedings of IEEE Conference on Software Maintenance, 2002.
63. Baxter, I.; Pidgeon, C.; Mehlich, M. DMS: program transformations for practical scalable software evolution. In Proceedings of the 26th International Conference on Software Engineering (ICSE '04), 2004.
64. Mens, T.; Tourwe, T. A survey of software refactoring. IEEE Trans. Softw. Eng. **2004**, *30* (2).
65. Hauser, R.F.; Friess, M.; Kuster, J.M.; Vanhatalo, J. An incremental approach to the analysis and transformation of workflows using region trees. IEEE Trans. Syst. Man Cybernetics, Part C: Applications and Reviews **2008**, *38* (3).
66. Henrard, J.; Roland, D.; Cleve, A.; Hainaut, J.L. Large-scale data reengineering: return from experience. In Proceedings of IEEE Conference on Reverse Engineering, 2008.
67. Arnold, R.S.; Bohner, S.A. (Eds.). *Software Change Impact Analysis*; Wiley-IEEE CS Press, 1996.
68. Briand, L.C.; Labiche, Y.; O'Sullivan, L. Impact analysis and change management of UML models. In Proceedings of the 19th ICSM, IEEE CS Press, 2003.
69. Kataoka, Y.; Imai, Y.T.; Andou, H.; Fukaya, T. A quantitative evaluation of maintainability enhancement by refactoring. In Proceedings of IEEE Conference on Software Maintenance, 2002.
70. Agrawal, H.; Alberi, J.L.; Horgan, J.R.; Li, J.J.; London, S.; Wong, W.E.; Ghosh, S.; Wilde, N. Mining system tests to aid software maintenance. IEEE Comput. **1998**, *31* (7), 64–73.
71. Schniedewind, N.F. Software quality maintenance model. In Proceedings of IEEE Conference on Software Maintenance, 1999.
72. Basili, V.; Rombach, D. Tailoring the software process to goals and environments. In Proceedings of the 9th International Conference on Software Engineering, Monterey, California, 1987.
73. Pfleeger, S.L. Software metrics: a rigorous and practical approach. *Course Technology*, 2nd Ed.; 1998.
74. Bandi, R.K.; Vaishnavi, V.K.; Turk, D.E. Predicting maintenance performance using object-oriented design complexity metrics. IEEE Trans. Softw. Eng. **2003**, *29* (1).
75. Coleman, D.; Lowther, B.; Oman, P. The application of software maintainability models in industrial software systems, J. Syst. Softw. **1995**, *29*, 3–16.
76. Kazman, R.; Bass, L.; Abowd, G.; Webb, M. SAAM: a method for analyzing the properties of software architectures. In Proceedings of IEEE International Conference on Software Engineering, 1994.
77. Kazman, R.; Klein, M.; Barbacci, M.; Longstaff, T.; Lipson, H.; Carriere, J. The architecture tradeoff analysis method. In Proceedings of IEEE Conference on Engineering of Complex Computer Systems, 1998.
78. Antoniol, G.; Cimitile, A.; DiLucca, A.; DiPenta, M. Assessing staffing needs for a software maintenance project through queuing simulation. IEEE Trans. Softw. Eng. **2004**, *30* (1). Q13
79. Sneed, H.M. Estimating the costs of software maintenance tasks. In Proceedings of IEEE Conference on Software Maintenance, 1995.
80. Boehm, B. *Software Engineering Economics*; Prentice Hall, 1981.
81. Black, F.; Scholes, M. The pricing of options and corporate liabilities. J. Polit. Econ. **1973**, *81* (3), 637–654.
82. National Aeronautics and Space Administration Cost Analysis and Evaluation Division http://www.nasa.gov/offices/pae/organization/cost_analysis_division.html.
83. Doval, D.; Mancoridis, S.; Mitchell, B. Automatic clustering of software systems using a genetic algorithm. In Proceedings of IEEE Conference on Software Technology and Engineering Practice, 1999.
84. Tran, J.B.; Holt, R.C. Forward and reverse repair of software architecture. In Proceedings of IBM Conference Center for Advanced Studies CASCON, IBM Press, 1999.
85. Krikhaar, R. *Software Architecture Reconstruction*. Ph.D. thesis, University of Amsterdam: The Netherlands, 1999.
86. Egyed, A.; Grunbacher, P. Automating requirements traceability: beyond the record and replay paradigm. In Proceedings of 17th International ASE Conference, IEEE CS Press, 2002.
87. Mylopoulos, J.; Chung, L.; Nixon, B. Representing and using non functional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. **1992**, *18* (6), 483–497.
88. Fickas, S.; Feather, M. Requirements monitoring in dynamic environments. In Proceedings of Requirements Engineering Conference, 1995.
89. Robinson, W.N. Implementing rule-based monitors within a framework for continuous requirements monitoring. In Proceedings of HICSS'05, 2005.
90. Yu, Y.; Wang, Y.; Mylopoulos, J.; Liaskos, S.; Lapouchnian, A. doPrado Leite, J.C.S. Reverse engineering goal models from legacy code. In Proceedings of IEEE RE'05, 2005.
91. Moskewicz, M.W.; Madigan, C.F.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: engineering an efficient sat solver. In Design automation. ACM Press: New York, USA, 2001.
92. Wang, Y.; McIlraith, S.; Yu, Y.; Mylopoulos, J. An automated approach to monitoring and diagnosing requirements. In Proceedings of IEEE/ACM International Conference on Automated Software Engineering, 2007.
93. Razavi, A.; Kontogiannis, K. Pattern and policy driven log analysis for software monitoring. In Proceedings of IEEE Conference on Computer Software and Applications, 2008.
94. Chaim, M.L.; Maldonado, J.C.; Jino, M. A debugging strategy based on requirements of testing. In Proceedings of IEEE Conference on Software Maintenance and Reengineering, 2003.
95. Brown, W.J.; Malveau, R.C.; Mowbray, T.J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*; Wiley, 1998.

96. Ying, A.T.T.; Murphy, G.C.; Ng, R.T.; Chu-Carroll, M. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. **2004**, *30* (9), 574–586.
97. Mockus, A. Votta, L.G. Identifying reasons for software changes using historic databases. In Proceedings of the IEEE International Conference on Software Maintenance, San Jose, CA, 120–130, 2000.
98. Larson, E. Assessing work for static software bug detection. In Proceedings of the 1st ACM international Workshop on Empirical Assessment of Software Engineering Languages and Technologies, 2007.
99. Ayewah, N.; Pugh, W.; Morgenthaler, J.D.; Penix, J.; Zhou, Y. Using FindBugs on production software. In Proceedings of the ACM OOPSLA 2007 Companion, 2007.
100. Koenig, A. Patterns and antipatterns. J. Object-Oriented Program. **1995**, *8* (1), 46–48.
101. Tsantalis, N.; Chaikalis; T.; Chatzigeorgiou, A. JDeodorant: identification and removal of type-checking bad smells. In Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR ’08), Athens, Greece, 2008; 329–331.
102. Sneed, H.M. Integrating legacy software into a service oriented architecture. In Proceedings of IEEE Conference on Software Maintenance and Reengineering, 2006.
103. Schmidt, D.; Stal, M.; Rohnert, H.; Buschmann, F. *“Pattern-Oriented Software Architecture” Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.
104. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley, 2002.
105. Hohpe, G.; Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*; Addison-Wesley, 2003.
106. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.M. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley, 1994.
107. Clements, P.; Northrop, L. *Software Product Lines: Practices and Patterns*, 3rd Ed.; Addison-Wesley Professional, 2001.
109. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley, 1999.
110. Gschwind, T.; Koehler, J.; Wong, J. Applying patterns during business process modeling. In Proceedings of the 6th International Conference on Business Process Management (BPM), vol. 5240 of Lecture Notes in Computer Science, 2008.
111. Yau, S.S.; Collofello, J.S.; MacGregor, T. Ripple effect analysis of software maintenance. In Proceedings of IEEE Conference on Computer Software and Applications Conference, COMPSAC, 1978.
113. Binkley, D. An empirical study of the effect of semantic differences on programmer comprehension. In Proceedings of the IEEE International Conference on Program Comprehension, 2002.
114. Mehra, A.; Grundy, J.; Hosking, J. A generic approach to supporting diagram differencing and merging for collaborative design. In ASE ’05: Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering, ACM Press, 2005.
115. Ivkovic, I.; Kontogiannis, K. Tracing evolution changes of software artifacts through model synchronization. In Proceedings of the 20th International Conference on Software Maintenance ICSM, 2004.
116. Hung, M.; Zou, Y. Recovering workflows from multi tiered e-commerce systems. In Proceedings of the IEEE International Conference on Program Comprehension, 2007.
117. Ernst, M.D. Static and dynamic analysis: synergy and duality. In Proceedings of International Conference on Software Engineeroing (ICSE) Workshop on Dynamic Analysis (WODA), 2003.
118. Ivkovic, I.; Kontogiannis, K. Towards automatic establishment of model dependencies using formal concept analysis. Int. J. Softw. Eng. Knowl. Eng. **2006**, *16* (4), 499–522.
119. Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Application*; Addison-Wesley, 2000.
120. El-Ramly, M.; Iglinski, P.; Stroulia, E.; Sorenson, P.; Matichuk, B. Modelling the system-user dialog using interaction traces. Eighth Working Conference on Reverse Engineering (WCRE), 2001.
121. Canfora, G.; Cimitile, A.; DeLucia, A.; DiLucca, G.A. Decomposing legacy programs: a first step towards migrating to client-server platforms. J. Syst. Softw. **2000**, *54* (2), 99–110.
122. Aversano, L.; Canfora, G.; Cimitile, A.; DeLucia, A. Migrating legacy systems to the web: an experience report. In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), 2001.
123. Canfora, G.; Fasolino, A.R.; Frattolillo, G.; Tramontana, P. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. J. Syst. Softw. **2008**, *81* (4), 463–480.
124. Sneed, H.M. Encapsulation of legacy software: a technique for reusing legacy software components. J. Ann. Softw. Eng. **2000**, *9*, 293–313.
125. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D. *Documenting Software Architectures: Views and Beyond* (SEI Series in Software Engineering); Addison-Wesley, 2002.
126. Feiler, F. et al. *Ultra-Large-Scale Systems: The Software Challenge of the Future*; Software Engineering Institute, 2006.
128. Czarnecki, K.; Helsen, S. Feature-based survey of model transformation approaches. IBM Syst. J. **2006**, *45* (3).
129. Kontogiannis, K.; Martin, J.; Wong, K.; Gregory, R.; Muller, H.; Mylopoulos, J. Code migration through transformations: an experience report. In Proceedings of CASCON, 1998.
130. Martin, J.; Müller, H.A. C to Java migration experiences. In Proceedings of IEEE Conference on Software Maintenance and Reengineering, 2002.
131. Penny, D. *The Software Landscape: A Visual Formalism for Programming-in-the-Large*; University of Toronto, 1993.
132. Devanbu, P.T. GENOA – a customizable front-end retargetable source code analysis framework. ACM TOSEM **1999**, *8* (2), 177–212.
133. Cordy, J.R.; Dean, T.R.; Malton, A.J.; Schneider, K.A. Source transformation in software engineering using the TXL transformation system. J. Inf. Softw. Technol. **2002**, *44* (13), 827–837.
134. Ferenc, R.; Beszedes, A.; Tarkiainen, M.; Gyimothy, T. Columbus reverse engineering tool and schema for C++.

In Proceedings of the International Conference on Software Maintenance, 2002.

135. Raza, A.; Vogel, G.; Plödereder, E. Bauhaus—a tool suite for program analysis and reverse engineering. In Proceedings of Reliable Software Technologies, Ada-Europe 2006, LNCS(4006), 2006.

136. IEEE Technical Committee on Software Engineering, Committee on Reverse Engineering and Reengineering. http://reengineer.org/tcse/revengr/.

138. Abrahamsson, P. et al. Agile processes in software engineering and extreme programming. In Proceedings of the 9th International Conference, XP, Springer, 2008.

139. M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. “Stratego/XT0.16: components for transformation systems”. In Proc. Of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics Manipulation, 2006.

140. Barros, S.; Bodhuin, T.; Escudie, A.; Queille, J.; Voidrot, J. Supporting impact analysis: a semi-automated technique and associated tool. In Proceedings of the 11th ICSM, IEEE CS Press, 1995.

142. Rilling, J.; Meng, W.J.; Chen, F.; Charland, P. Software visualization—a process perspective. In Proceedings of VISSOFT, 2007.

144. Storey, M.; Best, C.; Michaud, J.; Rayside, D.; Litoiu, M.; Musen, M. SHriMP views: an interactive environment for information visualization and navigation. In Proceedings of Conference on Human Factors in Computing Systems. ACM Press: New York, USA, 2002.

145. Greevy, O.; Lanza, M.; Wysseier, C. Visualizing live software systems in 3D. In Proceedings of the 2006 ACM symposium on Software Visualization, ACM Press: New York, USA, 2006.

146. German, D.; Hindle, A. Visualizing the evolution of software using softchange. Int. J. Softw. Eng. Knowl. Eng. **2006**, *16* (1).

147. Storey, M.A.; Bennett, C.; Bull, I.; German, D. Remixing visualization to support collaboration in software maintenance. In Proceedings of IEEE Conference on Software Maintenance, FOSM Track, 2008.

148. Ogawa, M.; Ma, K.; Bird, C.; Devanbu, P.; Gourley, A. Visualizing social interaction in open source software projects. In Proceedings of 6th International Asia-Pacific Symposium on Visualization (APVIS).

149. Xie, T.; Pei, J. MAPO: mining API usages from open source repositories. In Proceedings of International Workshop on Mining Software Repositories (MSR), 2006.

150. Mockus, A.; Fielding, R.T.; Herbsleb, J.D. A case study of open source software development: the apache server. In Proceedings of the 22nd International Conference on Software Engineering, 2000.

151. M. Godfrey, D. German. “The Past, Present, and Future of Software Evolution”. In Proc. of IEEE International Conference on Software Maintenance, FOSM Track, 2008.

152. Hassan, A.E.; Mockus, A.; Holt, R.C.; Johnson, P.M. Guest Editor’s Special Issue on Mining Software Repositories. IEEE Trans. Softw. Eng. **2005**, *31* (6), 426–428.

153. Zimmermann, T.; Weißgerber, P.; Diehl, S.; Zeller, A. Mining version histories to guide software changes. IEEE Trans. Softw. Eng. **2005**, *31* (6), 429–445.

154. Canfora, G.; Cerulo, L. Impact analysis by mining software and change request repositories. In Proceedings of the 11th International Symposium on Software Metrics, IEEE CS Press, 2005.

155. Hassan, A. The road ahead: mining software repositories. In Proceedings of IEEE Conference on Software Maintenance, FOSM Track, 2008.

156. Bassil, S.; Keller, R. Software visualization tools: survey and analysis. In Proceedings of International Workshop on Program Comprehension (IWPC ’01), 2001. **Q16**

157. Beyer, D.; Hassan, A.E. Animated visualization of software history using evolution story boards. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE ’06), 2006.

158. Collard, M.L.; Kagdi, H.H.; Maletic, J.I. An XML-based lightweight C++ fact extractor. In Proceedings of the 11th International Workshop on Program Comprehension (IWCP), 2003.

159. Gotel, O.C.Z.; Finkelstein, A.C.W. An analysis of the requirements traceability problem. In Proceedings of 1st International Conference on Requirements Engineering, 1994.

160. B.A. Kitchenham, G.H. Travassos, A. von Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. “Towards an ontology of software maintenance“. Journal of Software Maintenance and Evolution: Research and Practice, 11(6), 1999.

161. Lehman, M.M.; Perry, D.E.; Ramil, J.F. Implications of evolution metrics on software maintenance. In Proceedings of the IEEE International Conference on Software Maintenance, 1998.

162. Margolis, B. *SOA for the Business Developer: Concepts, BPEL, and SCA*; Mc Press, 2007.

163. Williams, C.C.; Hollingsworth, J.K. Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Softw. Eng. **2005**, *31* (6), 466–480.