http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 7 (1): 1-15, 2008
ISSN 1812-5638
© 2008 Asian Network for Scientific Information

Artifacts Recovery at Different Levels of Abstractions

Nadim Asif
Department of Computer Science and IT, The University of Lahore, Riwind Road, Lahore, Pakistan

Abstract: The software systems evolve and new modules and dependencies are added to support new features, while obsolete functionality is removed. Consequently, the design gradually diverges from its original design. Different design artifacts become inconsistent with the current implementations, making software evolution and servicing tasks difficult and error prone. This study describes a Reverse Engineering Abstraction Methodology (REAM) used to recover the design artifacts from the source code and available documentation. The methodology consists of (five models) high level, functional, architectural, source code and mapping models and these models represent the information of the subject system artifacts at different levels of abstractions for maintenance task at hand.

Key words: Maintenance, re-engineering, reverse engineering, design recovery, program understanding and architecture recovery

INTRODUCTION

The changes initiate the system's evolution due to a variety of reasons; by adding the new functionality in the system on the users request, adapting the new hardware and software technology and business decisions to improve the maintainability, reusability and quality of source code. In evolution stage, development effort focus on extending system capabilities to meet the user needs and the design gradually diverges from its original design. Different design artifacts become inconsistent with the current implementation, making change tasks difficult and error prone. Software evolution and servicing phase (maintenance) depends on several factors including the existence of accurate documentation of the system design (Rajlich and Bennett, 2000). In some cases, software and documentation fail to be consistent and subsequently the design-is rarely updated to reflect the modifications made to the system. In other cases, the original system design does not have any type of existing documentation and as such, any rationale behind the design decisions made during the implementation of the system is lost. In either case, lack of a consistent design has many impacts on the effectiveness of any efforts to maintain and modify existing systems.

Reverse engineering involves in extracting high-level information from the existing source code and available documentation (Chikofsky and Cross, 1990). Source code does not contain much of the original design information, which must be reconstructed from the available sources. Thus, additional information sources both human and automated are required from a combination of code, existing design documentation (if available), personal experience and general knowledge about problem and application domain (Biggerstaff, 1989). There are many recovery approaches exist and it become difficult for software engineers to choose a recovery technique suitable to recover the artifact for maintenance task at hand. Many existing approaches to recover the artifacts for software maintenance tasks can be categories into seven categories (Asif, 2006, 2007b); Program Comprehension approaches (Mayrhauser and Vans, 1995; Kothari *et al*., 2006, 2007), design Patterns based approaches (Kramer and Prechelt, 1996; Antoniol *et al*., 1998; Philippow *et al*., 2005), knowledge based approaches (Wills, 1993; Abd-El-Hafiz and Basili, 1996; Witte *et al*., 2006), Domain based approaches (Neighbors, 1980; Prieto-Diaz, 1989; Ornburn and Rugaber, 1992; Batory and Malley, 1992; Batory *et al*., 1993; Frakes and Kang, 2005), program slicing approaches (Weiser, 1984; Gallagher and Lyle, 1991; Tip, 1995; David and Lyle, 1998; Mund *et al*., 2002), clustering approaches (Lakhotia, 1997; Mancoridis *et al*., 1998; Sartipi and Kontogiannis, 2003; Mitchell and Spiros, 2006; Romero and Ventura, 2007) and Concept based approaches (Wille and Lattice, 1982; Snelting, 1996; Giuliano *et al*., 2006; Igor and Kostas, 2006; Dubois and Saint-Cyr, 2007; Cho and Richards, 2007). These approaches provide the benefit for recovering the design artifacts but resist the user in many ways to recover the design artifacts for maintenance tasks at different levels of abstractions and have limited support when the source code is not compilable, incomplete, have errors or have different programming languages dialects. For example, to recover the architectural artifacts for a maintenance task may require recovering many artifacts at the implementation, structural, functional and domain level in varying details. Many approaches also use recovery process, which is not suitable for the user to recover the design artifacts for maintenance task at different levels of abstractions. This paper describes the Reverse Engineering Abstraction Methodology (REAM) as a result developed to overcome these problems. REAM allows the user to use the available source code, documents, domain knowledge and experience to recover the artifacts at different levels of abstractions. The user can tailor the recovery process according to the available source code (written in multiple languages and have different dialects, contain errors, incomplete and not possible to compile), documents, knowledge, experience, resources and time. REAM has five models which help the software engineer to recover the artifacts at the implementation, structural, functional and domain level required for the maintenance task at hand in varying level of desired details. Many case studies (Asif *et al*., 2002b; Asif, 2002a, 2003; Asif, 2007a; Asif and Ramachandran, 2005, 2007b) on different types of software are conducted to recover the artifacts in varying desired details for maintenance tasks at different levels of abstractions using the REAM is presented in section two of this study. The results of these studies are also presented in this study to elaborate the models of the methodology. The section three describes the REAM five models to recover the artifacts for maintenance task at hand. A REAM process sketch is presented in section four to elaborate the steps to recover the desired artifact. The last section presents a case study and its results.

ARTIFACTS

In Archaeology artifact is an object made or modified by human and later recovered by some archaeological endeavor e.g., stone tools, pottery and jewelry. Hoard is a collection of artifacts purposely buried in ground. In software systems, the artifacts are buried in different layers of code and documents. The term artifact is adapted to define the basic unit for abstraction and to serve a purpose. Software system artifacts can be classified on the bases of the abstractions levels; implementation, structural, functional and domain level. Implementation level is a lowest level abstraction and at this level the abstraction of the knowledge of the language in which the system is written, the syntax and semantics of language and the hierarchy of system components (program or module tree) and system outputs rather then data structures and algorithms are represented. Structural level is a further abstraction of system components (program or modules) to extract the program structures, how the components are related and control to each other and at this level the data design and program design is extracted. The artifacts at this level are the data flow, control flow diagrams, processes and architectures. Functional abstraction level is a further higher abstraction level, it usually achieve by further abstraction of components or sub-components (programs or modules or class) to reveal the relations and logic, which perform certain tasks e.g., use cases and scenarios. Domain abstraction further abstracts the functions by replacing its algorithmic nature with concepts and specific to the application domain. The domain and external knowledge is also used to abstract the artifacts in the maintenance tasks. When the code and available documentation exist for maintenance activities, the artifacts are required to abstract it at different levels of abstractions. The relationships and interaction among different types of artifacts become complex and the artifacts need to abstract it for the required maintenance tasks at hand.

In reverse engineering process, aiding software engineer to understand the domain, functional, structural and implementation of software system in a particular problem help to recover the design artifacts for software evolution. The methods that provide a static set of techniques for the reverse engineering operations of recovering the artifacts is not suitable for all users in different domains. The users should be able to choose the way to recover the design artifacts according to the task at hand. An approach to recover the design artifacts and the tools supporting the approach must be flexible so that it can be applied to diverse domains at different levels of abstraction in varying details. Consider the variety of design information in the artifacts of the two example software systems-Mozilla (M8) and Apache (2.0.43) system. These two systems are chosen as examples for three reasons. First, the artifacts comprising the system are publicly available. Second, the systems are implemented in different programming languages; Mozilla is implemented primarily in C and C++ (use also HTML, XML and Java scripts) and Apache in C. Third, the systems are of moderate size; Mozilla comprising about three million lines of code and Apache consist of half million lines of code.

Each system is comprised of a variety of artifacts. Some artifacts like files data items exist in both systems and other artifacts like classes, functions, structures depend on the programming languages used to implement the system. A design artifact can be a logical view (Kruchten, 1995) of a Mozilla HTML parser, which is an object model when an object-oriented method is used. The design artifacts defined for the system is not limited to identifiable pieces of the static system artifacts but may extend to the system's dynamic state during execution. In

Mozilla, for instance, which is designed as several intercommunication of C, C++, Java and scripts processes, a process may be considered as a design artifact. Similarly, a variety of interactions or relations may occur between the artifacts. The relations are not limited to static properties of the system artifacts but extend to dynamic relations as well. For instance, in Mozilla interactions and events related to the user interface flow through Java scripts and are handled either in source code or in a script-more options normally specify command handlers, which flow through Java scripts to C++ and from C++ the handlers may drop directly to C.

The diversity of artifacts and relations makes it difficult for an engineer to gain an understanding and recovery of a system's design artifacts when performing a maintenance task at hand. A simple analysis of Mozilla source code for instance shows that there are about 1990 C++ files, 814 C files, 3038 header files and 18 java files. The core of the Apache includes 461 C files and 225 header files. The provision of design artifact information in a manageable form to the software engineer becomes even more difficult as system grows larger than Mozilla or Apache.

The software engineers also make use of programming knowledge, domain knowledge and comprehension strategies when attempting to understand the programs. They usually extract syntactic knowledge from the source code and rely on programming knowledge to form semantic abstraction for maintenance tasks. As the software evolves, the design artifacts drift farther and farther from the original designer's intent. Mostly, the software engineers make the project related decisions on their understanding of the architecture of the software systems and they rely on design artifacts and maintenance histories and source code. The most obvious way to support the design recovery is to produce and maintain adequate design artifacts. The methodology contributes to recover the design artifacts exist at different levels of abstraction in varying details for maintenance tasks at hand using the available source code, documentation, knowledge and experience. The methodology supports the integration of artifacts from sources other than the source code incrementally. The methodology helps to adapt top-down (starting from a high level goal and expectations), bottom-up (starting at code) or even opportunistic approach (combination between the two) for the design artifacts recovery for maintenance. The software engineers have specific goals for maintenance tasks at hand and they require different types of design artifacts at different levels of abstraction for the task at hand. The engineer can quickly form hypothesis from a variety of sources and then concentrate the effort on recovering the artifacts of the system that are relevant to the task at hand using the REAM. It also enables the engineer to tailor the recovery process to leverage their domain knowledge and experience at different levels of abstraction to recover the design artifacts of varying levels of details. The following case studies are conducted to recover the relevant artifacts of varying levels of details at different levels of abstraction for maintenance tasks at hand.

Software systems	Artifacts recovered for maintenance
Zip and unravel codes	Artifacts recovery (Functions, function calls, structnres, enumerations and abstract artifacts are extracted in this study. The abstract patterns are also designed to extract these artifacts) (Asif, 2002a; Asif *et al.*, 2002b).
Mozilla HTML	Design artifacts to access the parser feasibility to reuse it (Asif, 2006).
Design Recovery Tool (DRT)	The functional artifacts (e.g., use cases, scenarios) are recovered for maintenance (Asif and Ramachandran, 2005).
Email System (BridgeMail)	The high level and functional artifacts are recovered to understand and perform the maintenance tasks (Asif, 2006).
Mozilla	Architecture (Asif, 2003)
Apache	Conceptual architectnre (Asif, 2007a).

REVERSE ENGINEERING ABSTRACTION METHODOLOGY

The methodology consists of five models; high level model, functional model, architectural model, source code model and mapping model (Fig. 1). High level model help to gain an abstract understanding of the system at a higher level-as hints to recover the artifacts. Functional model represents the functional elements and relationships among the system artifacts to understand the mechanical details of the functionality of the system artifacts. Source code model is extracted from the source code using the high-level model (and functional model) to develop an understanding and model it at an abstract level. The architecture model is developed with the understanding gained out of developing the high level, functional and source code models and by understanding the dependencies between the various artifacts. The user defines a mapping model between the entities in the source code model (i.e., functions, classes) and the entities in the high-level (i.e., concepts), functional (i.e., relation and logic among programs or concepts) and architectural (i.e., modules or components) models. These models abstract the system artifacts at different levels of abstraction to recover the artifacts.

High level model: The first phase of the methodology involves the collection of artifacts (source code,

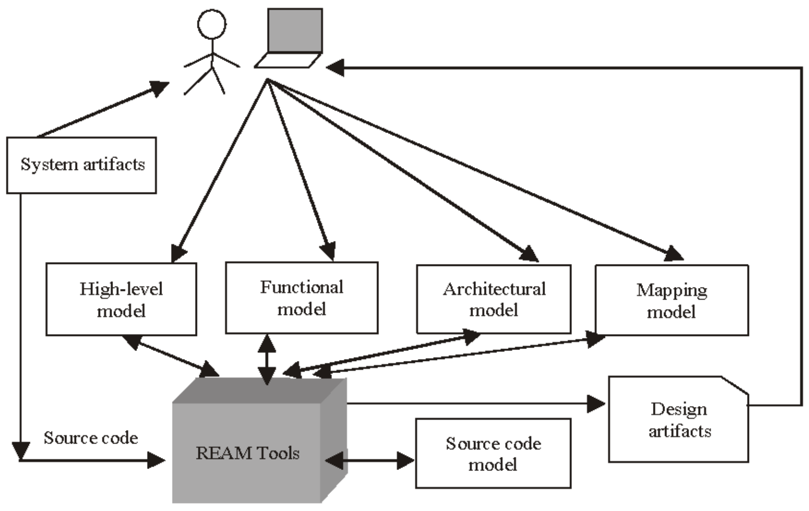


Fig. 1: REAM system (user iteratively develop the desired models for a maintenance task at hand using the available source code, documents, knowledge and experience)

documents and interviewing experts), reviewing the existing artifacts that give information about the system, building system knowledge and defining the goals for reverse engineering process and selecting the visual model to represent the reverse engineered artifacts. The information embedded in a system source code often leads software engineers to rely on high-level models that are separate from the source. Sometimes these high level models are part of the system documentation. The documentation of a system might contain the design information describing the major artifacts of the system. The engineers also develop mental models through interactions and experience with systems and these models are not accurate representation of systems source code but these high level models are useful for summarizing the information extracted or collected from the software system artifacts and reason about the change tasks (Murphy *et al*., 2001).

Instead of updating the high-level models continuously, the REAM reduces the risk associated with these high-level models by recovering the artifacts at different levels of abstraction. In essence, the engineer uses a task-specific high-level model as hints on which to recover the design information from the software system's artifacts. Using the high level model as hints, the engineer may then selectively investigate aspects of the system functions, architecture, source code and mappings pertinent to the task to recover the design artifacts for maintenance task at hand.

Artifact collection: Artifact collection of the subject system is an essential step of REAM. The higher-level abstractions cannot be constructed and explored without the raw data because it is used to identify the system's artifacts and relationships. There are several existing artifacts in any software system like the source code, specification documents and design documents that are vital important for the reverse engineering effort. The other available documentation consists of the program maintenance manual and the user manuals are also important source. These are gathered together in an effort to build the knowledge for the software system. The engineer identify the require artifacts that he want to collect from the subject system, how (and when) want this data to collect and how wish to represent it. These highlight the important artifacts and relations in the collected artifacts and de-emphasize or filter out immaterial ones for the task at hand.

Existing documents review: The existing documentation may be the only starting point from which the application can be appreciated. This step of the methodology involves a review of the existing documentation. The output of this is a functional description of the system with out mentioning the implementation details or programming language. It begins with a short summary of the overall system behavior. The description is top-down and it proceeds from a discussion of the overall system behavior to a discussion of those sub-components that are visible to the user. In the absence of accurate documentation, the reverse engineers are required to construct a description of what a system does given only a description of how it does it.

System knowledge: For successful recovery of design artifacts, the data must be in a form that facilitates efficient storage and retrieval, permits analysis of artifacts and relationships and reflect the user's perception of the systems characteristics. By adding narrative information, describing the system functionality and purpose help to produces more appropriate documentation under the constraints imposed by the computing environment. The generated reports, the input and output files and the user interfaces also improve the system knowledge. The descriptive information can be obtained from existing documentation and from knowledgeable system maintenance personnel (if available).

Identification of goals: It is important to identify the goals and limitations of the effort before beginning the reverse engineering activity. The reverse engineering of huge and complex could be limited to the extraction of the architectural design from the source code. For example, a re-engineering effort might entail the adoption of a process to define the feature level abstraction of the system functionality.

Reverse engineering is a time bound activity and a clear definition of how far to go, as a trade off against the cost involved is necessary. The effort is also expected to be iterative and incremental and could potentially lead to a bigger and more complex artifact than the source code. It is therefore important to keep the big picture in mind and focus on predetermined goals. The documentation available for the software system, the nature and size of the source code, suggestions and ideas from the system experts or developers would be the inputs to develop such milestones.

Visual model: To communicate the understanding achieved during or at the end of the reverse engineering effort a visual modeling medium is required. A suitable modeling tool can be chosen that support the software system that is being reverse engineered. For example, the Unified Modeling Language (UML) can be used as a visual modeling medium (Kollmann *et al*., 2002). The UML has become the de-facto standard adopted by the software industry to visualize and communicate a software system design.

In a BridgeMail (is a online e-communication solution for enterprise-direct marketing, epublishing, sales and call centers, senior management, product management, IT management and other customize activities for different types of clients) case study (Asif, 2006), a high level model of BridgeMail system developed using the REAM by the software engineer from the available documentation and with the help of users to perform the maintenance tasks. The Fig. 2 shows the BridgeMail high level model. The system is composed of five servers; apache server, web server (Tomcat), application server (JBoss), database server (MySQL) and mail server (Linux). The apache is used as proxy server and web server contains the java programs, beans, utility and client components. The application server has EJBS (Enterprise Java Beans) and server side components (All Jar files). The database server store and retrieve all data of the system. The mail server receives and sends all the messages.

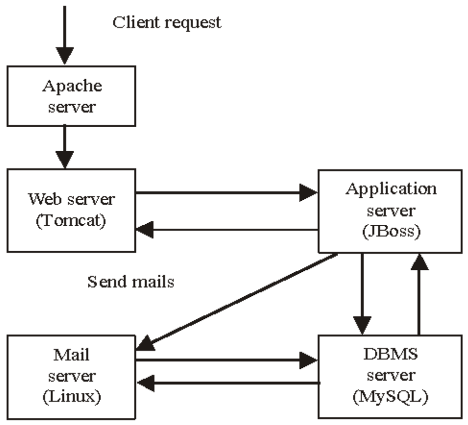
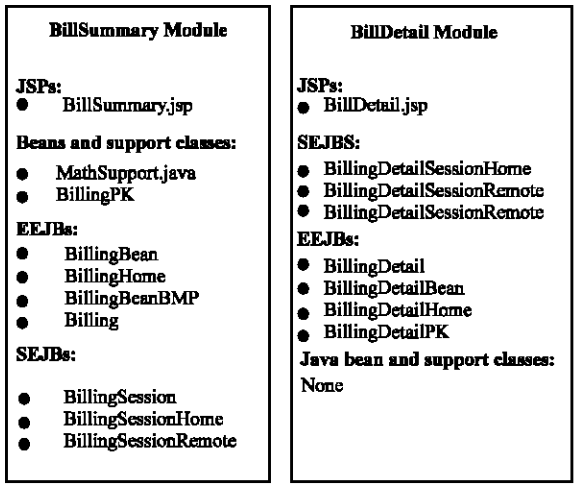


Fig. 2: BridgeMail high level model developed by collecting the existing artifacts, reviewing the existing documents, making relationships between artifacts and mapping the artifacts to the available source code for validation

Functional model: Understanding the functionality of the software system aids in the reverse engineering process (Mayrhause and Vans, 1995; Kothari *et al*., 2007). An abstract understanding of the functions that the system performs is the first step towards understanding the works inside. It consists of an analysis of the systems input/output behavior expressed in terms of nested data flow diagrams. At the highest level, a Context diagram can represent the system behavior. A Context diagram is a data flow diagram with one activity. The Context diagram generated from the system description is verified by examining the source code. This serves as an internal check on the consistency of the recovered design artifact. It may be a Use Case diagram in Unified Modeling Language (UML)-documents the functional features of the system. The history facts (comments etc.), which are recorded in the source code during the development and the maintenance of the software system is extracted at this phase to understand the mechanical details of the functionality.



BillSummary Module

JSPs:
- BillSummary.jsp

Beans and support classes:
- MathSupport.java
- BillingPK

EEJBs:
- BillingBean
- BillingHome
- BillingBeanBMP
- Billing

SEJBs:
- BillingSession
- BillingSessionHome
- BillingSessionRemote

BillDetail Module

JSPs:
- BillDetail.jsp

SEJBS:
- BillingDetailSessionHome
- BillingDetailSessionRemote
- BillingDetailSessionRemote

EEJBs:
- BillingDetail
- BillingDetailBean
- BillingDetailHome
- BillingDetailPK

Java bean and support classes:
None

Fig. 3: Abstracted functional details developed for maintenance tasks at hand and each contains the java script files, beans, support classes and other details, which are relevant to the module to elaborate the functionality

In the BridgeMail case study (Asif, 2006), the engineer was very much interested to understand how the BridgeMail source code was divided into different modules and how these modules interact to perform the particular tasks. The functional model was developed starting with a short summary of the overall system and the engineer found that high-level model of the system developed in the previous phase was very useful and natural to start the process. The source code models were extracted iteratively and mapping models mapped the entities according to the maintenance tasks to develop relations between different entities with the help of REAM tools. The engineer preferred to develop the abstract functional description as shown in Fig. 3, which contain the different files and details of the modules only. This helped to understand and access the relevant files of source code to perform the required maintenance task within target time. The developers comments were also extracted from the source code with the help of REAM tool and summarized to further elaborate the details of the functions the software perform.

Source code model: The source code of a large system or application suite is recognized as a collection of independent and interrelated texts (Malton *et al*., 2001). Source code is text and has two purposes, both essential: to represent artifacts, which can be realized mechanically (i.e., by compilation) and to record communication between human beings. It is text by virtue of its recording and communication role. Source texts are independent and some source texts are maintained as themselves, rather than being dependent on some prior data. They are input to the mechanical realization of the software. Some source texts are interrelated because of all the links between them that arise as a result of abstraction, including lexical links (e.g., include directives in C language) and semantic links (subroutine calls, global data, class instantiations etc.). The source code is a flat view of the system as extracted from the source files. This phase of the methodology extract the developers documentation, reference formats (menus, screens, reports), design decisions and debug source code to gain an understanding of the source code and to model it at an abstract level from the source code which exist in many forms; implemented in different languages or have different dialects and scripts, incomplete and can not be compiled or have errors.

Developers documentation: Software developers normally document important and complex portions of the source code with comments and with other notes. The extraction of these comments provides clues about the functionality, structure and behavior of the system. This also provides the description of the essential decisions that have been taken in the design of a system.

Debug source code: Stepping through the source code using a development environment provide vital clues about the flow of control in complex software projects. This could prove especially useful for object-oriented systems since they are more complex and non-intuitive than sequential programs. For example, object interactions in an object-oriented system can be identified and modeled by running through the code and by closely watching the objects of interest during their transactions.

Reference formats: The details of menus, screens and reports, as well as a complete list of textual requirements, external and internal interfaces and data elements are necessary to easily understand the design because all of these are appropriately integrated into the design. These reference formats provide vital clues about the functionality of the components or modules.

Design decisions: Design decisions are structural decisions made by the original designer or programmer. Typical design decisions include the decomposition of a function into its sub functions, the handling of special cases and the use of one data structure to represent another that is not directly provided by the programming language. Complicated programs are designed as layers of abstraction and the detection of a one decision usually leads the detection of the other decisions in other layers.

The specification language: The specification language is used to define the construct of require artifacts to extract it from the source code. The language for specifying the source code model extraction has three parts: patterns, actions and analysis. The patterns of interest describe the constructs to search in system artifacts, actions to execute after the pattern is matched to a portion of system artifacts and analysis operations that extract a source code model from an intermediate representation produced during scanning.

Patterns: The engineer specifies the information to extract from the system artifacts as patterns. Each pattern uses regular expressions to describe the artifact construct that is required to find within the system artifacts. The abstract patterns are composed of different patterns to extract the complex artifacts from the source code.

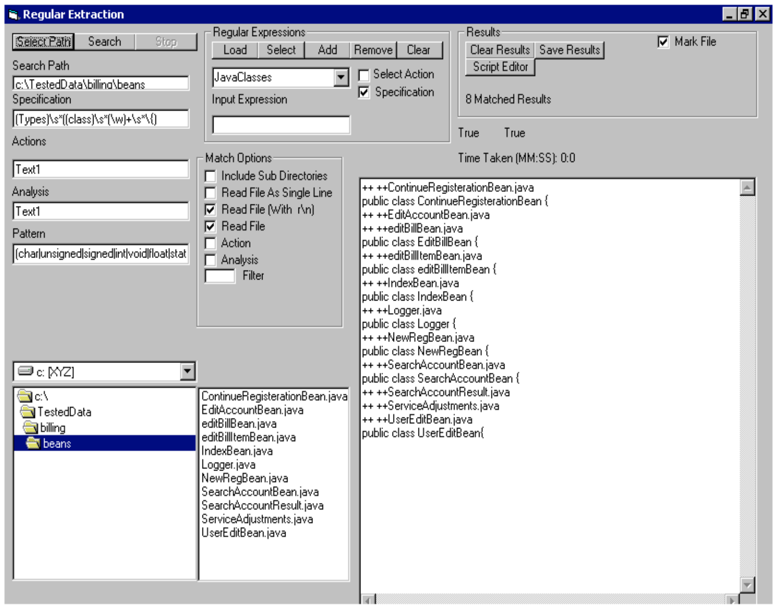
Actions: An engineer may attach the action to the pattern to be executed when a pattern is matched in the source code. The action code performs operations such as controlling the matching of the constructs in the source code to particular patterns. Specifically, an engineer may reject matches to a particular pattern by invoking the regular expression within the action. This control is often used to reject matches when patterns are too general.

Analysis operations: In certain cases, the desired source code model cannot be built directly during the scanning of the source code. The source code model can be extracted at the conclusion of scanning from multiple types of information extracted from the system artifacts. An engineer defines the desired extraction pattern in an analysis section of the specification. The extraction is performed on the intermediate results produced from scanning.

In the BridgeMail case study, the source code model was developed to extract the required artifacts is presented here to elaborate the source code model. First, the initial specifications were written by considering the task, an engineer examines the required system artifacts for the task and wrote a lexical specification, which describe the information to extract the source code model. The initial specification defined by the engineer was <Types> class <ClassName> to extract the java class names of the BridgeMail system. The words in angle brackets means the abstracted types, i.e., types represent the type of the class that can be public, private or protected. By using the initial specification then the abstract pattern (JavaClasses) of regular expression (Types)\s*((class)\s*(\w)+\s*\{) designed to extract the classes Fig. 4, further each class functions and relationships with the other classes are also extracted iteratively.

Mapping model: One way to create the associations of source code model entities to high level, functional and architectural models entities are to enumerate the associations explicitly. However, this approach may not be suitable if the source code model of the system has the functions in thousands. This seeming difficulty is mitigated using three techniques to collapse the size of the mapping specification. First, the map may be partial so that the user need only provide the associations for those portions of the system of interest for the task at hand. Second, the user may use the physical (e.g., directory and file) and logical (e.g., functions and classes) artifacts of source to name many source code model entities in a single map entry. Finally, the user may use the regular expressions to take advantage of naming conventions in the system artifacts. The user iteratively computes and investigate successive mapping model until acquires enough information for the task being performed. A source code model entity editBillItemBean is presented in Fig. 4 is selected for more details. The editBillItem is further mapped to the source code to extract all the relevant editBillItem details, which are shown in Fig. 5.

Architectural model: The importance of high-quality documentation in design recovery is widely recognized. Without it, only the source of reliable information is the source code itself. While the design recovery may not be a problem for a single developer or even for a small team while they are together, it is problem for a long-term large-system evolution. The only reliable up-to-date and applicable documentation is the program source code. It is left to maintenance personnel to explore the low-level source code and piece together disparate information to form high-level structural models. Manually creating just one such architectural documentation is always arduous; creating the necessary design documents that describe the architecture from multiple points of view is often impossible. It is exactly this sort of in-the-large design documentation that is needed to expose the structure of large software systems. Recovering the design of such systems involves uncovering the system-level structure. Software structure is the collection of artifacts used by software engineers when forming mental models of software systems. These artifacts include software components such as procedures, modules, classes and interfaces; dependencies among components such as client-supplier, inheritance and control flow; attributes such as component type, interface size and interconnection strength. A software system's design is



Search Path: c:\TestedData\billing\beans

Specification: (Types)\s*((class)\s*(\w)+\s*\{()

Actions: Text1

Analysis: Text1

Pattern: (char|unsigned|signed|int|void|float|stat

Regular Expressions: JavaClasses

Input Expression:

[ ] Select Action

[✓] Specification

[✓] Mark File

8 Matched Results

True True

Time Taken (MM:SS): 0:0

Match Options:

[ ] Include Sub Directories

[ ] Read File As Single Line

[✓] Read File (With r\n)

[✓] Read File

[ ] Action

[ ] Analysis

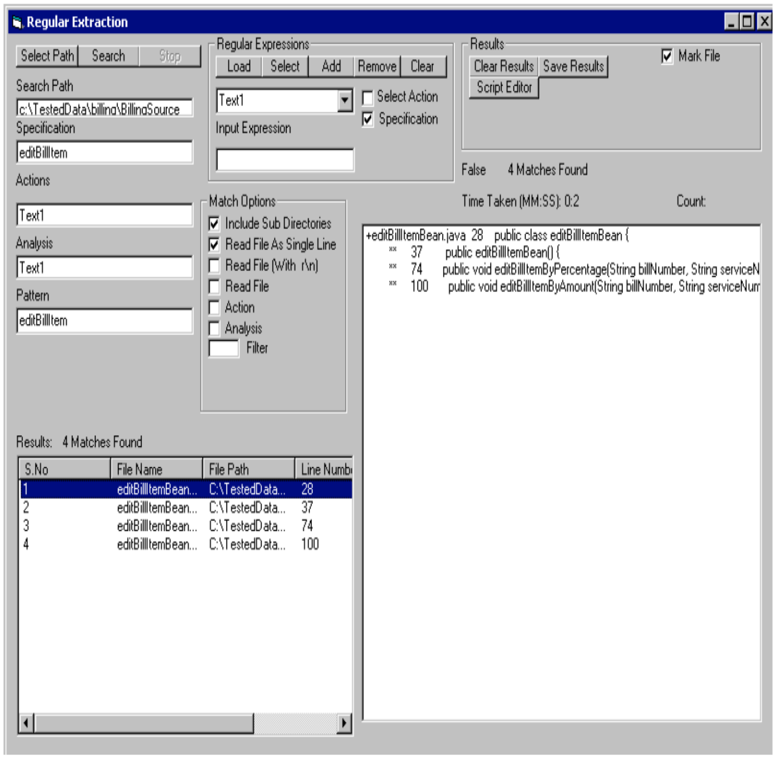
Filter

```
c: [XYZ]
c:\
 TestedData
  billing
   beans
```

```
ContinueRegisterationBean.java
EditAccountBean.java
editBillBean.java
editBillItemBean.java
IndexBean.java
Logger.java
NewRegBean.java
SearchAccountBean.java
SearchAccountResult.java
ServiceAdjustments.java
UserEditBean.java
```

```
++ ++ContinueRegisterationBean.java
public class ContinueRegisterationBean {
++ ++EditAccountBean.java
++ ++editBillBean.java
public class EditBillBean {
++ ++editBillItemBean.java
public class editBillItemBean {
++ ++IndexBean.java
public class IndexBean {
++ ++Logger.java
public class Logger {
++ ++NewRegBean.java
public class NewRegBean {
++ ++SearchAccountBean.java
public class SearchAccountBean {
++ ++SearchAccountResult.java
++ ++ServiceAdjustments.java
++ ++UserEditBean.java
public class UserEditBean{
```

Fig. 4: Abstract pattern used to extract the classes from the BridgeMail system



Search Path: c:\TestedData\billing\BillingSource

Specification: editBillItem

Actions: Text1

Analysis: Text1

Pattern: editBillItem

Regular Expressions: Text1

Input Expression:

[ ] Select Action

[✓] Specification

[✓] Mark File

False 4 Matches Found

Time Taken (MM:SS): 0:2 Count:

Match Options:

[✓] Include Sub Directories

[✓] Read File As Single Line

[ ] Read File (With r\n)

[ ] Read File

[ ] Action

[ ] Analysis

Filter

Results: 4 Matches Found

S.No	File Name	File Path	Line Numb
1	editBillItemBean...	C:\TestedData...	28
2	editBillItemBean...	C:\TestedData...	37
3	editBillItemBean...	C:\TestedData...	74
4	editBillItemBean...	C:\TestedData...	100

```
+editBillItemBean.java  28    public class editBillItemBean {
     **    37       public editBillItemBean() {
     **    74      public void editBillItemByPercentage(String billNumber, String serviceN
     **    100      public void editBillItemByAmount(String billNumber, String serviceNum
```

Fig. 5: editBillItem class is mapped to the source code to get the more relevant details

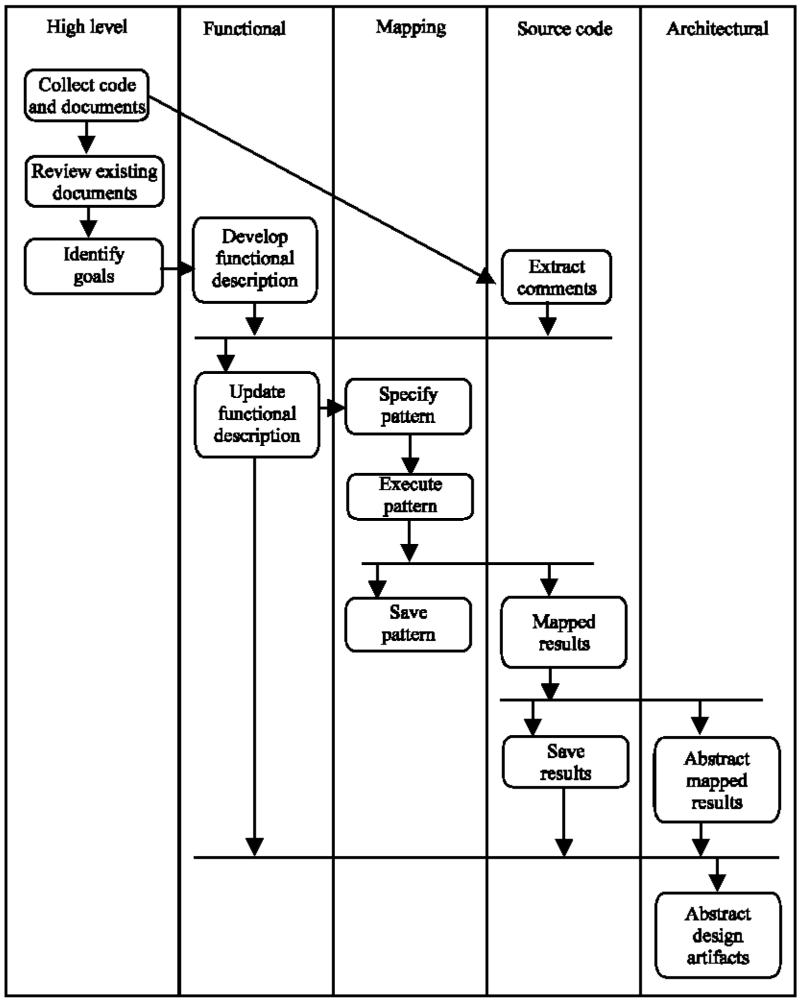


Fig. 6: REAM process depicts the recovery of artifact for task at hand

the organization and interaction of these artifacts (Perry and Wolf, 1992; Andersson and Johnson, 2001).

The architectural model is extracted with the understanding gained out of developing the high level and functional models. The high level model and functional model provide the functionality of extracting the architectural information from the source code and by understanding some of the dependencies between the various design artifacts (REAM view in Fig. 7).

Co-relate: Once the models are developed at the different levels of abstraction described above, it is important to correlate them to verify and glean away any discrepancies. Re-documentation of the models will increase comprehension about the system and also offer scope for improving the models before they are released. The result of this phase of the process is the reverse engineered documentation, which can then be utilized.

REAM PROCESS

A sketch of Reverse Engineering Abstraction Methodology (REAM) process is shown in Fig. 6. The REAM process is based on a combined top-down and bottom-up approach to recover the design artifacts. First, high-level model for the system is developed using the available documentation, system knowledge and experience and refined based on empirical investigations involving the existing system. Second, source code models are constructed by using the REAM tools developed during this research or can be extracted by using third party tools.

In the next step, high-level and source code models are used to develop the functional model. The mapping model is defined to explore and build the functional model (relationship between high-level and source code model) to recover the design artifacts. At this stage an abstract understanding of the functions that the system performs is developed. It can consist of an analysis of the system's input/output behavior expressed in terms of nested data flow diagrams or it may be a Use Case diagram in Unified Modeling Language-documents the functional features of the system. This help to understand some of the reasons driving the design decisions made by the developers of the software.

The architectural model is extracted from the understanding and the artifacts developed by high-level, functional, source code and mapping models. The architectural description is extracted through out the process and this provides a detail view of the system. The component and package diagram of Unified Modeling Language UML) or other can be used to convey the information about the architecture of the system. Once the models are developed at the different levels of abstraction described above, it is important to correlate them to verify and glean away any discrepancies. Re-documentation of the models increases the comprehension about the system and also offers scope for improving the models before they are released. The result of this process is the recovered design artifacts, which can then be utilized for the task at hand.

The Methodology permits an engineer to recover the design artifacts from the available documentation and the source code within the framework of a source code, high level, functional, architectural and mapping models. The user selects artifacts rather than those artifacts being created through the process of using the techniques. By selecting the artifacts, the user is assured the recovered artifacts are those useful for the design recovery and reasoning about the task being performed. The methodology is:

- **Lightweight:** In that an engineer can quickly and easily develop different models of interest, keeping in view of the task at hand.
- **Iterative:** In that the engineer may selectively refine the inputs to develop the REAM models until the desired information is obtained.
- **Partial:** In that the mapping entities in a given source code, functional and high-level models may not be included for the recovery of all design artifacts of a system.
- **Approximate:** In that the mapping used to associate the models may coarsely associate source code model entities with high-level, functional and architectural model entities.
- **Scalable:** means that it works well on multilingual system, ranging from a few thousands to over a million lines-of-code within the context of an iterative approach.

A CASE STUDY

During this research, a case study conducted is presented here to describe the key features of the REAM. First, present a sketch of its use to aid an engineer to recover the design artifacts. The task is to develop an HTML parser, which is a part of current software development project. Two options are considered regarding the HTML parser, one is to design and implement the parser from the start and another is to reuse the existing HTML parser. But it is decided to reuse the Mozilla HTML parser by performing the changes according to the requirement because the design and implementation is required for new development and the development team has no experience of such an implementation. The task facing the engineer is to recover the design artifacts to gain an understanding about the design and functionality and to assess the feasibility of reusing the Mozilla HTML parser with an existing development in a specific time. The engineer must first extract the design artifacts comprising the HTML parser from the source code and the available documentation to reuse the parser in the application. The Mozilla system is comprised of about three million lines of code; it is difficult for an engineer to recover the design of the system directly from source code. REAM process depicts in Fig. 6 is applied to aid this task.

First, the engineer developed a high level model suitable for recovering the design artifacts and to reason about the task. For instance, a high level model may be an object diagram or it may be an informal sketch of the calls between system modules. High level model was formed by collecting the available system artifacts from several available sources like source code, design documents, specification documents, the developer/user knowledge and using experience. In the absence of accurate documentation, the engineer is required to construct a description of what a system does given only a description of how it does it. The output of this may be a functional description of the system, without mentioning the implementation details. It begins with a short summary of the overall system. The description is top down and it proceeds from the discussion of those components or sub-components that are visible to the user or specific concepts related to application domain and resulting to develop a high level model. The functional model developed on the bases of high level model, started with an abstract understanding of the functions that the system performs regarding the mechanical details of functionality. The history facts (comments etc.) can be viewed or extracted in terms of the history of its development as recorded in the source code at this phase. The history facts were abstracted and found useful to

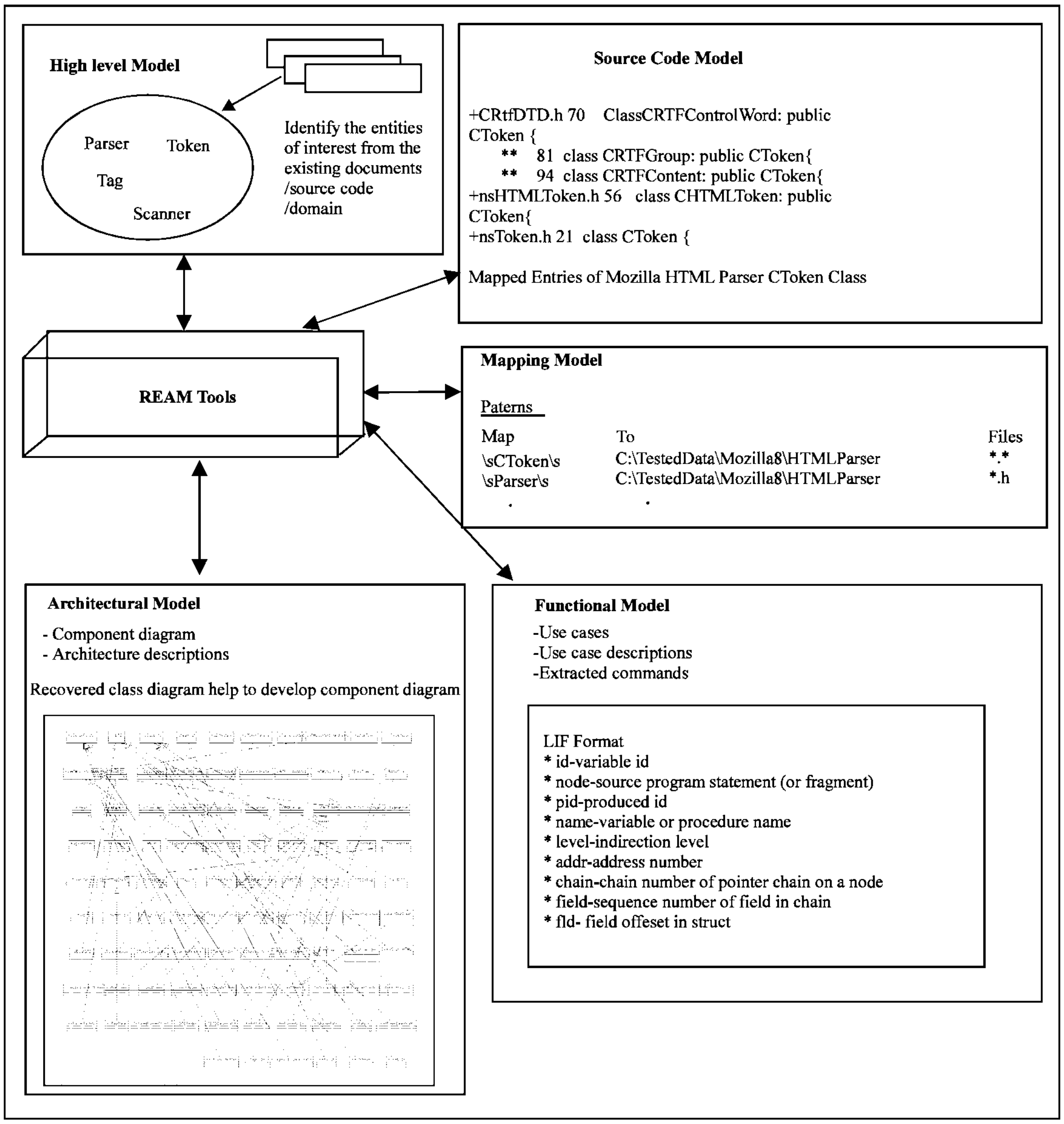

High level Model
Parser
Token
Tag
Scanner
Identify the entities of interest from the existing documents /source code /domain
Source Code Model
+CRtfDTD.h 70 ClassCRTFControlWord: public CToken {
** 81 class CRTFGroup: public CToken{
** 94 class CRTFContent: public CToken{
+nsHTMLToken.h 56 class CHTMLToken: public CToken{
+nsToken.h 21 class CToken {
Mapped Entries of Mozilla HTML Parser CToken Class
REAM Tools
Mapping Model
Paterns
Map
To
Files
\sCToken\s
C:\TestedData\Mozilla8\HTMLParser
.
\sParser\s
C:\TestedData\Mozilla8\HTMLParser
*.h
Architectural Model
- Component diagram
- Architecture descriptions
Recovered class diagram help to develop component diagram
Functional Model
-Use cases
-Use case descriptions
-Extracted commands
LIF Format
* id-variable id
* node-source program statement (or fragment)
* pid-produced id
* name-variable or procedure name
* level-indirection level
* addr-address number
* chain-chain number of pointer chain on a node
* field-sequence number of field in chain
* fld- field offeset in struct

Fig. 7: REAM view used for the recovery of desired artifacts for the task at hand

improve and verify the understanding about the functionality. The good understanding about the functional aspects of the application was developed. This helped to understand some of the reasons driving the design decisions made by the developers. The Fig. 7 depicts the REAM view used to recover the desired artifacts.

In the next step, the engineer extracted the design information from the source code. A source code model may be produced either by statically analyzing the system's artifacts or collecting information dynamically (during the system's execution). For instance, a source code model may consist of an inheritance relation between classes in an object-oriented system or a relation describing the message sends between objects or both. In this case, the engineer used the REAM implemented tools of a system to extract a source code model comprising information about classes and calls from the HTML parser. The REAM tool used the regular expressions as shown in the Fig. 8 to collect all the classes and derived classes' information from the Mozilla HTML parser source code. In the regular expression (Class | Deriveclass), the words Class and Deriveclass are the abstract reserve word, which represents the regular expression pattern (b) for the classes and derived classes. The regular expression pattern (b) contains the abstract patterns ClassName and Type, which represent the sub regular expression pattern (c) for the class name and class type in this case (Fig. 8).

In the fourth step of the methodology, the engineer described the mapping models to find the required relations of the particular artifact with the other artifacts using the high level, functional and source code models. The engineer defined the mapping between the entities in the source code model, high level and functional models. For example, the mapping model first entry will map the CToken class to all the HTML parser files (Fig. 9). The HTML parser CToken class relationship with the other artifacts was mapped to the source code by using the regular expression \sCToken\s (Fig. 10).

In Fig. 10, the numbers represent the line number of that particular file where the mapped artifact exists. The

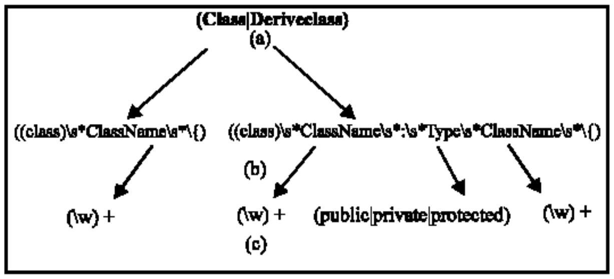
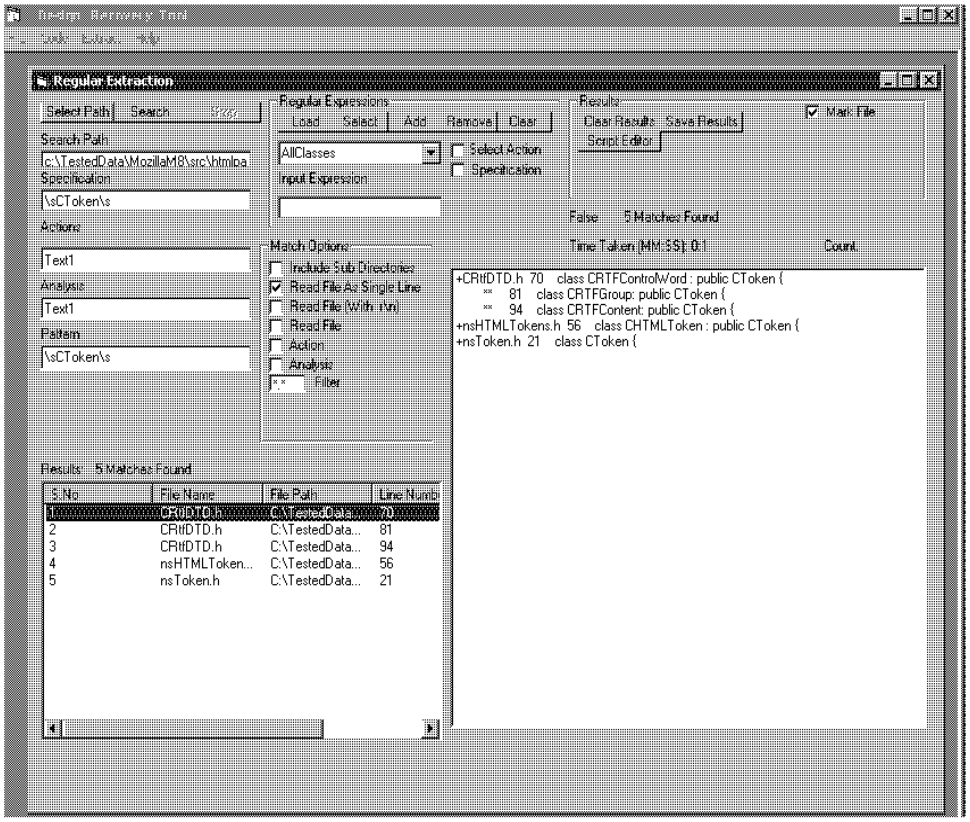

(Class|Deriveclass)
(a)
((class)\s*ClassName\s*\{)
((class)\s*ClassName\s*:\s*Type\s*ClassName\s*\{)
(b)
(\w) +
(\w) +
(c)
(public|private|protected)
(\w) +

Fig. 8: Regular expression pattern to extract the HTML parser classes

Map	To	Files
\sCToken\s	C:\TestedData\Mozilla8\HTMLParser	*.*
\sParser\s	C:\TestedData\Mozilla8\HTMLParser	*.h
. .	. .	. .

Fig. 9: Mapping Model Entries for Mozilla HTML Parser Classes



Search Path: c:\TestedData\MozillaM8\src\htmlpa

Specification: \sCToken\s

Actions: Text1

Analysis: Text1

Pattern: \sCToken\s

AllClasses

☐ Select Action
☐ Specification

✓ Mark File

☐ Include Sub Directories
✓ Read File As Single Line
☐ Read File (With \r\n)
☐ Read File
☐ Action
☐ Analysis
. Filter

False 5 Matches Found

Time Taken (MM:SS): 0:1 Count

```
+CRtfDTD.h  70    class CRTFControlWord : public CToken {
      **    81    class CRTFGroup: public CToken {
      **    94    class CRTFContent: public CToken {
+nsHTMLTokens.h  56    class CHTMLToken : public CToken {
+nsToken.h  21    class CToken {
```

Results: 5 Matches Found

S.No	File Name	File Path	Line Numb
1	CRtfDTD.h	C:\TestedData...	70
2	CRtfDTD.h	C:\TestedData...	81
3	CRtfDTD.h	C:\TestedData...	94
4	nsHTMLToken...	C:\TestedData...	56
5	nsToken.h	C:\TestedData...	21

Fig. 10: REAM tool mapped the CToken class to the Mozilla HTML parser source code

mapping model entries mapped all the classes to the HTML parser source code. This causes the relationship of all the classes in the HTML parser with the other classes. The regular expression language used in the mapping is parameterized so that engineer may tailor the language according to the need to ease the specification of the mapping for a particular system to map the design artifacts. For Instance, an engineer may tailor the regular expression language to refer to functions and calls if working with a system implemented in a procedural language. The engineer iteratively compute and investigate successive mapping models until acquire enough information for the task. The mapping develops relation and help to consolidate all the models.

CONCLUSIONS

Software evolution activities are required to represent the artifacts at higher levels of abstraction than source code. The methodology contributes to recover the design artifacts exist at different levels of abstractions in varying levels of details for the maintenance tasks at hand. A software engineer can adapt a recovery process, which is suitable to recover the desired design artifacts exist at different levels of abstraction varying in details for the maintenance task at hand. It also allows the software engineer to use the existing recovery approaches in its context. The approach uses the available existing documents and source code, which can not be compiled or have errors or have coded in different programming languages and have different dialects. The approach helps the engineer to recover the specific design artifacts from the source code and using the available sources (documents, experience and domain knowledge) at different levels of abstraction in varying details.

The software engineer develop the required models (High Level, Functional, Source Code, Mapping and Architecture) of interest according to the required design artifacts at varying levels of details for the task at hand. The methodology is sufficiently lightweight, iterative, partial, approximate and scaleable so that a user based on the particular needs of the task can recover the design artifacts of the system at varying levels of details.

In the future, more case studies will be conducted to abstract and measure the recovery effectiveness for different types of artifacts of software systems. The REAM tools will be extended in the future to abstract and visualize the recovered artifacts. The abstract regular expressions for C, C++, JAVA, COBOL and PASCAL languages codes have been designed to produce source code models and it will be further improved to cover more other languages like Smalltalk. This makes it also possible to apply REAM tools on large set of source codes of different languages. In near future, more studies will be conducted to extract the conceptual artifacts like architecture rational using the methodology. The methodology will also be applied on large industrial legacy systems and web based systems to recover the artifacts for re-engineering, decision making and future planning.

ACKNOWLEDGMENTS

I would like to thanks to Prof. Dr. Janet Finlay, Dr. Patricia Allen, Dr. Mark Dixon and Dr. George Coxhead for their support and comments on this research. Thanks also to the Dr. Muthu Ramachandran and Dr. Peter Sawyer also for taking the time to comments on this research. I should also thanks to the University of Lahore, staff members and students for providing the technical assistance in this research Final thanks to Dr. Emden R. Gansner (AtandT Research Laboratories) for making Acacia tool available to me for this research and Higher Education Commission (HEC) Pakistan for funds to present this research at international conferences.

REFERENCES

Abd-El-Hafiz, S.K. and V.R. Basili, 1960. A Knowledge-based approach to the analysis of loops. IEEE Trans. Software Eng., 22: 339-360.

Andersson, J. and P. Johnson, 2001. Architectural integration styles for large-scale enterprise software systems. In: Proceedings of 5th IEEE International Enterprise Distributed Object Computing Conference, IEEE Computer Soc. Press, pp: 224-236.

Antoniol, G., R. Fiutem and L. Cristoforetti, 1998. Design Pattern Recovery in Object Oriented Software. In: Proceedings of 6th International Workshop on Program Comprehension, June 24th-26th, IEEE Computer Soc. Press, pp: 153-160.

Asif, N., 2002a. Architecture recovery. In: Proceedings of International Conference of Information and Knowledge Engineering (IKE). June 24th-27th, Las Vegas, Nevada, CSREA Press, pp: 656-662.

Asif, N., M. Dixon, J. Finlay and G. Coxhead, 2002b. Recover the design artifacts. In: Proceedings of International Conference of Information and Knowledge Engineering (IKE). June 24th-27th, Las Vegas, Nevada, CSREA Press, pp: 656-662.

Asif, N., 2003. Reverse engineering methodology to recover the design artifacts: A case study. In: Proceedings of International Conference of Software Engineering Research and Practice (SERP). June 23rd-26th, Las Vegas, Nevada, CSREA, pp: 932-938.

Asif, N. and M. Ramachandran, 2005. Recover the use case models. In: Proceedings of International Conference of Software Engineering Research and Practice (SERP). June 27th-30th, Las Vegas, Nevada, USA., pp: 884-889.

Asif, N., 2006. Software Reverse Engineering. Soft Research Press (ISBN: 969-9062-00-2) http://www.softsole.org/sre.

Asif, N., 2007a. Recovery of architecture artifacts. The International Conference on Software Engineering Theory and Practice (SETP), July 9-12, Orlando, FL, USA., pp: 249-254.

Asif, N., 2007b. Artifacts recovery techniques. Int. J. Software Eng. (JSE), 1: 26-66.

Batory, D. and Sean O'Malley, 1992. The design and implementation of hierarchical software systems wit reusable components. Trans. Software Eng. Methodol., ACM, 1: 355-398.

Batory, D., V. Singhal, M. Sirkin and J. Thomas, 1993. Scalable Software Libraries, Sigsoft. ACM, December, pp: 191-199.

Biggerstaff, T.J., 1989. Design Recovery for Maintenance and Reuse. IEEE Comput., July, pp: 36-49.

Chikofsky, E.J. and J.H. Cross, 1990. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7: 13-17.

Cho, W.C. and Richards, 2007. D. Ontology construction and concept reuse with formal concept analysis for improved web document retrieval. Web Intelligence and Agent Systems, March, 5: 109-126.

David, W.B. and J.R. Lyle, 1998. Application of the pointer state subgraph to static program slicing. J. Syst. Software, pp: 17-27.

Dubois, D. and D. Saint-Cyr, 2007. Florence dupin, prade, henri. A possibility-theoretic view of formal concept Analysis. Fundamenta Informaticae, 75: 195-213.

Frakes, W.B. and K. Kang, 2005. Software Reuse research: Status and future. IEEE Trans. Software Eng., 31: 529-536.

Gallagher, K.B. and J.R. Lyle, 1991. Using program slicing in software maintenance. IEEE Trans. Software Eng., 17: 751-761.

Giuliano, A., Guéhéneuc and Yann-Gaël, 2006. Feature identification: An epidemiological metaphor. IEEE Trans. Software Eng., 32: 627-641.

Igor, I. and K. Kostas, 2006. Towards automatic establishment of model dependencies using formal concept analysis. Int. J. Software Eng. Knowledge Eng., 16: 499-522.

Kollmann, R., P. Selonen, E. Stroulia, T. Systa and A. Zundorf, 2002. A study on the current state of the art in tool-supported UML-based static reverse engineering. In: Proceeding of the 9th Working Conference on Reverse Engineering (WCRE). IEEE Computer Soc. Press, Los Alamitos, pp: 22-34.

Kothari, J., T. Denton, S. Mancoridis and A. Shokoufandeh, 2006. On computing the canonical features of software systems. In Proceedings of the 13th Working Conference on Reverse Engineering, Benevento, October 23-27). IEEE Computer Soc., pp: 93-102.

Kothari, J., T. Denton, A. Shokoufandeh and S. Mancoridis, 2007. Reducing program comprehension effort in evolving software by recognizing feature implementation convergence. In: Proceedings of the 15th International Conference on Program Comprehension, ICPC, pp: 17-26.

Kramer, C. and L. Prechelt, 1996. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Proceedings of the 3rd Working Conference on Reverse Engineering. IEEE Computer Soc. Press, pp: 206-216.

Kruchten, P., 1995. The 4+1 view model of architecture. IEEE Software, 12: 42-50.

Lakhotia, A., 1997. A unified framework for expressing software subsystem classification techniques. J. Syst. Software, 36: 211-231.

Malton, A.J., J.R. Cordy, D. Cousineau, K.A. Schneider, T.R. Dean and J. Reynolds, 2001. Processing Software Source Text in Automated Design Recovery and Transformation. In: Proceedings of 9th International Workshop on Program Comprehension. Toronto, May, IEEE Press, pp: 127-134.

Mancoridis, S., B.S. Mitchell, C. Rorres, Y. Chen and E.R. Gansner, 1998. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: Proceedings of the Sixth International Workshop on Program Comprehension, 24th-26th June, IEEE Computer Soc. Press, pp: 45-52.

Mayrhauser, V.A. and A.M. Vans, 1995. Program comprehension during software maintenance and evolution. IEEE Computer, August, pp: 44-55.

Mitchell, B.S. and M. Spiros, 2006. On the automatic modularization of software systems using the bunch tool. IEEE Trans. Software Eng., 32: 193-208.

Mund, G.B., R. Mall and S. Sarkar, 2002. An efficient dynamic program slicing technique. Inform. Software Technol., 44: 123-132.

Murphy, G., D. Notkin and K. Sullivan, 2001. Software reflexion models: Bridging the gap between design and implementation. IEEE Trans. Software Eng., 27: 364-380.

Neighbors, J.M., 1980. Software construction from components. Ph.D Thesis, TR-160, University of California at Irvine, USA.

Ornburn, S. and S. Rugaber, 1992. Reverse Engineering: Resolving Conflicts Between Expected and Actual Software Design. In: Proceedings of Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA., pp: 32-40.

Perry, D.E. and A.L. Wolf, 1992. Foundations for the study of software architecture. ACM SIG-SOFT Software Eng. Notes, 17: 40-52.

Philippow, I., D. Streitferdt, M. Riebisch and S. Naumann, 2005. An approach for reverse engineering of design patterns. Software Syst. Model, 4: 55-70.

Prieto-Diaz, R., 1989. Classification of Reusable Modules, in Software Reusability/Concepts and Models, Addison Wesley.

Rajlich, V.T. and K.H. Bennett, 2000. A Staged Model for the Software Life Cycle. IEEE Computer, pp: 66-71.

Romero, C. and S. Ventura, 2007. Educational data mining: A survey from 1995 to 2005. Expert Syst. Appl., 33: 135-146.

Sartipi, K. and K. Kontogiannis, 2003. A User-assisted approached to component clustering. J. Software Mainten. Res. Pract., 0: 1-32.

Snelting, G., 1996. Reengineering of configurations based on mathematical concept analysis. ACM Trans. Software Eng. Methodol., 5: 146-189.

Tip, F., 1995. A survey of program slicing techniques. J. Programming Languages, 3: 121-189.

Weiser, M., 1984. Program slicing. IEEE Trans. Software Eng., 10: 352-357.

Wille, R. and R. Lattice, 1982. Theory: An Approach Based on Hierarchies of Concepts. In: Ordered Sets, Rival, I. (Ed.). Reidel, pp: 445-470.

Wills, L.M., 1993. Flexible Control for Program Recognition. In: Proceedings of Working Conference on Reverse Engineering, Baltimore Maryland, IEEE Computer Soc. Press, pp:134-143.

Witte, R., T. Kappler and C. Baker, 2006. Ontology Design for Biomedical Text Mining. Chapter 13 in Semantic Web: Revolutionizing Knowledge Discovery in the Life Sciences, Springer Verlag.