AUTOMATED SOURCE CODE MEASUREMENT ENVIRONMENT FOR SOFTWARE QUALITY

Except where reference is made to the work of others, the work described in this dissertation in my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Young Lee

Certificate of Approval:

James H. Cross
Professor
Computer Science and Software Engineering

Kai H. Chang, Chair
Professor
Computer Science and Software Engineering

Dean Hendrix
Associate Professor
Computer Science and Software Engineering

David Umphress
Associate Professor
Computer Science and Software Engineering

George T. Flowers
Interim Dean
Graduate School

AUTOMATED SOURCE CODE MEASUREMENT ENVIRONMENT FOR SOFTWARE QUALITY

Young Lee

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
December 17, 2007

AUTOMATED SOURCE CODE MEASUREMENT ENVIRONMENT FOR

SOFTWARE QUALITY

Young Lee

Signature of Author

Date of Graduation

DISSERTATION ABSTRACT

AUTOMATED SOURCE CODE MEASUREMENT ENVIRONMENT FOR SOFTWARE QUALITY

Young Lee

Doctor of Philosophy, December 17, 2007
(M.S., Hallym University, 1991)
(B.S., Hallym University, 1989)

155 Typed Pages

Directed by Kai H. Chang

Measuring how well software component can be reused and maintained helps programmers not only write reusable and maintainable software, but also identifies reusable or maintainable components. We develop an automated measurement tool, *JamTool*, for object-oriented software system and describe how this tool can guide a programmer through measuring internal characteristics of a program for software reuse and maintenance.

In this research, primitive but comprehensive metrics for object-oriented language are extensively studied and statistically analyzed to show internal characteristics from classes selected from various applications. Also, the automatically identified connected unit, reusable unit, and maintainable unit are discussed.

We demonstrate *JamTool*'s ability through case studies. The first case study investigates whether *JamTool* can be used to assess the reusability on the evolution of an open software system. The second case study investigates whether *JamTool* can be used to capture the difference between two consecutive versions on the evolution of the open software system. The third case study investigates whether the metrics defined and implemented in *JamTool* are related to each other.

Style manual or journal used: IEEE Standard

Computer software used: Microsoft Word 2003

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 The general area of the research

While application development has become a huge and complex task, software productivity has improved slowly over the past years. One of the many goals of software developers (e.g., project managers and programmers) is to have control of software production and its quality. According to Moser and Henderson-Sellers [42], the following three steps are important to achieving this goal.

1. Knowing where one stands
2. Aiming where we wish to go
3. Going there (and reapplying the problem solving steps periodically while going)

Steps 1 and 2 are related to measurement, i.e., we should measure what we want to control.

Fenton and Pfleeger describe that a quality software product is characterized by many sound software attributes that may provide useful indications of the maintainability and reusability of a software product [17]. Without an accompanying assessment of product quality, progress of product is meaningless. Thus, it is important to recognize and measure certain desirable software quality attributes.

Fisher and Light define software quality as “The composite of all attributes which describe the degree of excellence of the computer system.”[19] Fenton et al. focus on the *purpose* of software quality by defining, “The totality of features and characteristics of a software product that bear on its ability to satisfy the stated or implied needs.”[16]

Despite several attempts to quantify the elusive concept of software quality like McCall’s Factor Criteria Metric (FCM) model [40] and Basili’s Goal Question Metric (GQM) approach [2], measurement of software quality is still not empirically validated.

Object-oriented technologies have claimed to improve software quality to support reuse and to reduce the effort of maintaining the software product. However, many object-oriented methods, tools, or notations are being used without evaluation.

Kitchenham et al. have observed that code and design metrics can be used in a way that is analogous to statistical quality control [27]. According to them, object-oriented code can be accepted or rejected based on a range of metric values. Rejected object-oriented code can then be revised until the metric values fall within a specified acceptable range.

It is argued that existing traditional software metrics are not suitable for object-oriented systems. Therefore, many new metrics are being proposed for object-oriented systems, but only a few have been validated. We need a metric set for object-oriented software construction to measure how the reusability and maintainability of the software could be improved. But metric research of the object-oriented paradigm is still in its infancy.

The primary motivation to reuse software components is efficiency. It is achieved by reducing the time, effort and/or cost required to build software systems. The quality and reliability of software systems are enhanced by reusing software components, which also means reducing the time, effort and cost required to maintain software systems. Researchers agree that although maintenance may turn out to be easier for object-oriented systems, it is unlikely that the maintenance burden will completely disappear. One approach to controlling software maintenance costs is the utilization of software metrics during the development phase to help identify potential problem areas.

Measuring how well the software component can be reused and maintained helps programmers not only write reusable and maintainable software, but also identify reusable or fault-prone components. Since there are hundreds of software complexity measures that reveal internal characteristics of an object-oriented program, it is important to have the right criterion to select a good subset of these measures.

This research will develop an automated metric tool that attempts to guide programmers to reuse and maintain object-oriented programs based on software measurement.

The following research activities will be accomplished in this study.

Quality Measurement Model Development

We will first identify essential software properties that have been suggested as having an impact on software quality. The properties that can be directly or indirectly derived from the source code will then be selected for this study.

We will divide measurement factors (i.e., reusability and maintainability) into five subfactors (i.e., identification, separation, modification, validation, and adaptation) in a top-down fashion. We believe that these subfactors are more tangible and useful to connect to software product metrics and cover most of the reuse and maintenance activities. We will also apply bottom-up approach to develop quality measurement models for reusability and maintainability based on available measurement types that are related to reuse and/or maintenance properties. Using these top-down and bottom-up approaches, we will construct a concise quality measurement model for reusability and maintainability.

Widespread adoption of object-oriented metrics can only take place if the metrics have been empirically validated, i.e., they accurately measure the attributes of software and can be applied easily.

Automated Measurement Tool

Users can get an instant measurement feedback while developing object-oriented software using an automated tool implemented in this research. The collection, derivation, and display of metrics would take place interactively to provide practical and non-intrusive feedback. Effort is also devoted to present the metric results along with the connected classes, to locate reusable or maintainable classes.

The research in this dissertation describes how an automated measurement tool [29] can guide a programmer through measuring internal characteristics of a program for software reuse and maintenance purposes.

1.2 Statement of the Problems

It is worthwhile to note that Zuse claims that the results of measurement are difficult to interpret if too many properties of a program are combined into one number [51], and Schneidewind argues that a standard set of quality measurement may be available in the future [47]. Information is lost if a single-valued measure is used. A vector of measures can provide complete information on individual properties of a program.

GQM is useful to identify objectives for measurement, but the available set of metrics may not be applicable to the desired objectives. Thus, identifying a set of "potentially useful" metrics in some systematic manner could improve the object-oriented metrics research effort [22]. After the identification of a set of quality factors and a set of metrics, the relations between them should be identified by empirical test. While GQM++ attempts to resolve some weaknesses of GQM through additional stages, Dromey argues that single-level is better than multiple intermediate levels between quality factors and software metrics as a means of linking them [14]. Empirical tests are needed to back both of these approaches. These previous approaches do not address problems of measurement such as appropriate data scales, alarm threshold, and representation of measurement result.

Consequently, previous efforts have been hampered by the following difficulties that have discouraged or delayed the application of object-oriented metrics. In particular:

- There is no clear relationship between the external quality factors and the metrics of the software.
- Most of existing metrics are not intuitive. It requires education on the users' side to have numerical thinking about the quality of software and how to

apply them.

- In case of measurements that are intrusive and interruptive, measuring software quality intimidates programmers. Therefore, it is difficult to apply them in the industry.
- Some metric sets have not been validated theoretically and empirically.

Furthermore, there are other reasons that software metrics are not used widely in the industrial world:

- Due to the lack of the availability of a standard metric set, it is difficult to choose an appropriate metric set for a user's purpose.
- It is difficult to interpret the measurement results.

This research addresses these issues by focusing on a framework for a customized quality model and interactive automated metric tool. The key features of this research are:

- To define simple and computable object-oriented metrics that quantify potential reuse and maintenance. The metrics should be easy to comprehend and use, and require only simple and well-formed formulas.
- To implement an automated metric tool that collects, analyzes, interprets, and presents the metric data automatically.
- To guide a programmer to software reuse and maintenance through measuring internal characteristics of a program.
- To empirically verify the validity of the metrics.

2 LITERATURE REVIEW

There have been several attempts to quantify the fuzzy concept of software quality by developing a set of metrics for various attributes related to the concept. These metrics all involve some degree of software measurement with the ultimate objective of improving software quality. As described later in this chapter, constructing a quality measuring model could be guided by several approaches. Unfortunately, measuring software quality is still an unsolved problem.

2.1 The GQM Approach and its extension

2.1.1 The GQM approach

If we can measure the development progress towards a quality product, the management of production process is simplified. In manufacturing, individual components are compared to tolerance limits (or goals) in order to reject poor quality products. As a result, desirable manufacturing processes can be found. Likewise, in order to develop a software measurement approach, the goals that we want to attain should be clearly defined, e.g., to increase programmer productivity; to decrease the number of defects reported per unit of time or to improve the efficiency of the overall development process [42].

Basili and Weiss propose a Goal Question Metric (GQM) framework with the aim of providing a systematic approach to translate measurement needs into metrics [2]. The measurement goals that can be refined into questions in measurable terms, should be answered in terms of enumerated metrics. The GQM approach has enabled managers to find objectives for measurement and metrics for their software products. But this also calls for thorough knowledge on their part of the organization as well as the developed software product.

Gray and MacDonell say that GQM is usually applied with its:

- Particular purpose (e.g. to evaluate, predict, classify)
- Certain perspective (e.g. the manager, user, programmer)
- Given object (e.g. specification, code)
- Environment (e.g. the people, tools, methodologies).

Thus, GQM is useful to ensure the proper metrics are allocated to assess the conceptual goal [20].

Gray and MacDonell also argue that organizations are faced with a wide range of software metrics that can lead to difficulties in the selection of appropriate metrics for a particular goal. Given the goals of a specific organization, the generic set of metrics needs to be tailored. As a result, the tailored metrics set has the greatest predicting power for the desired purpose and the least cost of data collection. Grey and MacDonell also recommend a task within a framework that will assist in decomposing goals in order to develop a set of software metrics [20].

2.1.2 Metrics Multidimensional Framework

The problem with the application of the GQM approach to object-oriented metrics is that metrics may not exist. Thus, a complementary activity would be to identify the ‘M’ component of ‘GQM’ independently of the specific goals and questions. The first major step would be to identify a set of "potentially useful" metrics in some systematic manner. Moser and Henderson-Sellers have presented such a method in the form of a so-called Metrics Multidimensional Framework (MMDF) [42].

The MMDF approach is composed of three dimensions. The first dimension is the external characteristic. It is divided into Quality (e.g., Understandability, Maintainability, Reusability, etc.) and Size (e.g., External and Internal). The second dimension is the granularity (e.g., System, Part, Class, and Method) at which the metric is applicable. Finally, the third dimension is the lifecycle phase (e.g., Analysis, Design, and System use). While the reasons and motivation for any individual using metrics in an object-oriented development environment may vary, Moser and Henderson-Sellers argue that the most popular metric usage purposes can be identified and described as combinations of these three dimensions. This approach can lead to a minimal set of metrics that is desirable for practical managerial purposes. Because MMDF permits the identification of useful metrics, it should improve the current, more ad hoc and uncoordinated object-oriented metrics research effort [22].

After defining a set of quality goals and a set of metrics, the correlation between them should be identified by empirical test using regression.

2.1.3 GQM++

MacDonell extends GQM into a hierarchy of goals, subgoals, domains, subdomains, questions, subquestions, and characteristic measures. He also argues that by breaking a goal into separate subgoals, the essential differences between metrics needed for each subgoal can be identified [36].

However, such extensions do not seem to go far enough. Fenton criticized that GQM is useful to identify objectives for measurement, but it does not address the actual problems of measurement such as appropriate data scales [17]. Gray and MacDonell interpret this criticism as the absence of any feasibility, economic, or correctness checks in GQM and that its simple and intuitive nature leads to these problems.

Further, Gray and MacDonell argue that other precedent conditions should be studied in addition to GQM. These considerations include the costs and benefits of data collection, the detailed plan of modeling and analysis methods, and the agreement of how the measurement results could be applied for their software product. The proposed framework, Goal/Question/Metric/Collection/Analysis/Implementation (GQMCAI, simply GQM++), attempts to resolve some weaknesses of GQM through additional stages [36], including data collection, modeling, and implementation. Their extension also includes cost/benefit information and assesses the program in terms of economic justification and feasibility [20].

While GQM++ is suggested to be a more comprehensive and pragmatic data collection and analysis process, empirical tests are needed to back up this claim.

2.2 Software Quality Models

The relationship between software characteristics and software quality has been investigated and proposed by many researchers [17, 40, 5, 24]. There have been attempts to quantify software quality resulting in omnibus models which have fixed relationship between quality and metrics. Assessing quality by measuring internal properties is also attractive because it offers an objective and context independent view of quality [28].

2.2.1 Omnibus Software Quality Metrics

Both McCall et al. and Boehm et al. describe product quality using a hierarchical approach [40, 5]. In McCall's Factors-Criteria-Metrics (FCM) model, high-level product quality like "reusability" and maintainability" are called *factors* that can be decomposed into several lower-level attributes i.e., *criteria* (See Figure 2-1). The Manager, who has responsibility for the software development, or the potential user who will use the to-be-developed software, should be interested in the final product quality, especially its performance, usability, reliability, etc. These views of software product are described in terms of software *factors* and *criteria*. But these factors and criteria are too elusive to be applied to software development. Thus, the criteria should be related directly to measurable attributes of the software process or product. FCM model has three views (uses) of software product quality, eleven factors, and twenty-five criteria. For example, factor "maintainability" relates to several criteria such as consistency, simplicity, conciseness, self-descriptiveness, and modularity. Factor "reusability" is decomposed into generality, self-descriptiveness, modularity, machine independence, and software system independence.

Boehm's model, which has a hierarchical structure similar to the FCM model, has two primary uses, 'maintainability' and 'utility'. Maintainability is further divided into intermediate constructs: understandability, modifiability, and testability.

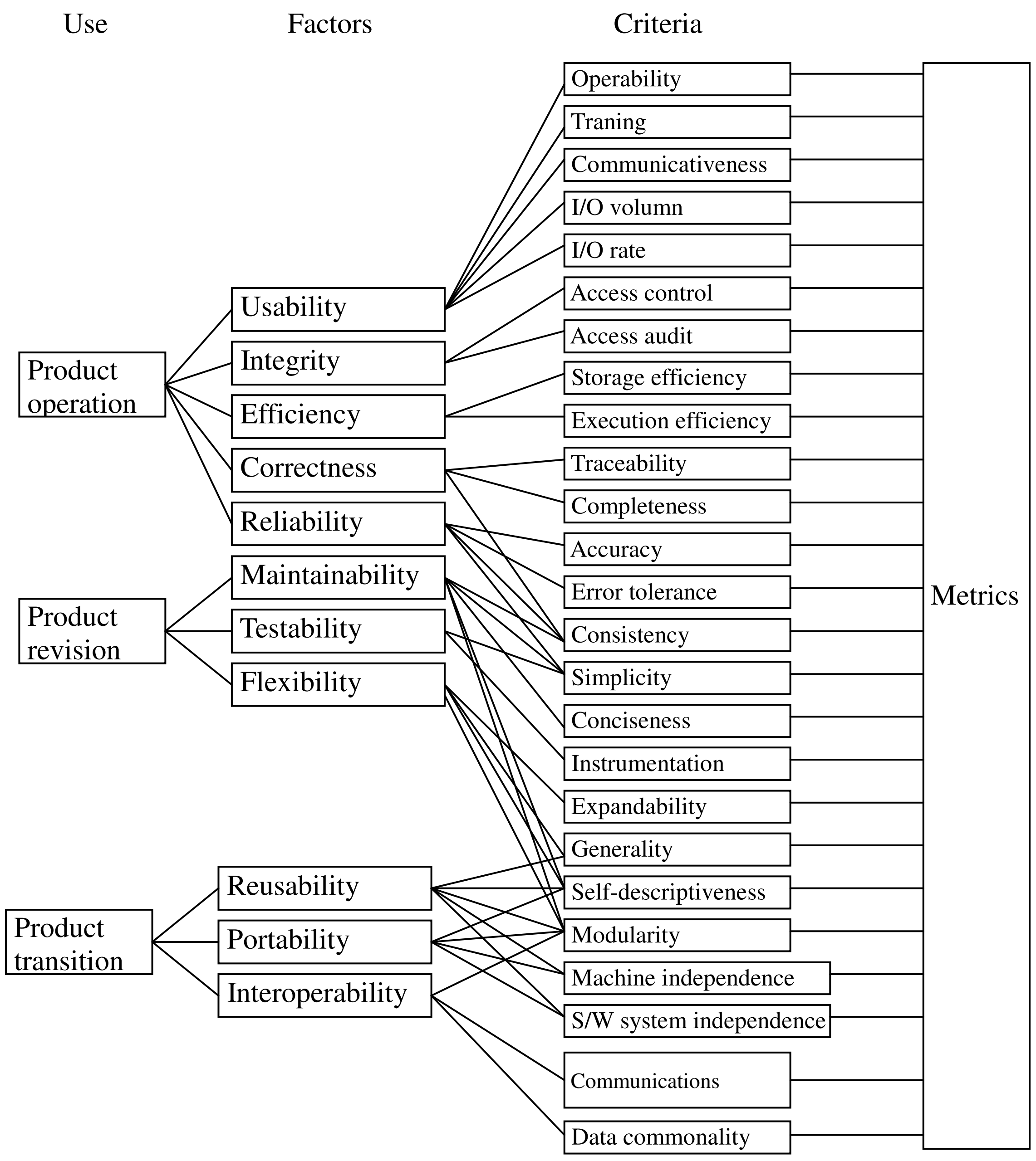


Figure 2-1: McCall Software Quality Model

2.2.2 ISO 9126

According to Fenton and Pfleeger, global software quality model is required for comparing quality among software systems. Because of this requirement, the ISO 9126 model is proposed with six factors - functionality, reliability, efficiency, usability, maintainability, and portability [24]. Despite its incompleteness and conflict with other standards, ISO 9126 is used by many companies to support their quality evaluation as Fenton and Pfleeger described in [16].

2.2.3 Dromey's Quality Model Framework

Recently, Dromey also defines a model for software product quality [14]. In this model, seven high-level quality attributes (six factors of ISO-9126 and reusability) are linked in a structural form of software elements (variable and expression) that influence software quality [14]. The emphasis is on defining and describing the quality-carrying properties, which are classified further into correctness, structure, modularity, and descriptive properties. Because Dromy's model is designed to be refined by empirical use to build a useful model, it is a framework to construct a quality models rather than a fixed model (e.g., McCall's FCM model). He argues that placing a single level (a set of quality-carrying properties) is better than placing several vaguely decomposed intermediate levels between the high-level quality and the components of product as a means of linking them. In linking the desirable quality-carrying properties and the high-level quality, quality mode can be constructed in bottom-up or top-down fashion. Each established link should be verified empirically.

In Dromey's approach, identifying and associating a set of quality-carrying properties in a structural form is the first task in constructing a quality model. With successively defined, evaluated, and refined models, we can build a software quality model that ensures quality and detects quality defects in software.

2.2.4 Fenton's Approach to Software Quality

Fenton defines external product attributes as those that can be measured in terms of how the product relates to its environment [17]. For example, if the product is a software code, then its reliability (defined in terms of the probability of a failure-free operation) is an external attribute. It is dependent on both the machine environment and the user. Whenever we think of software code as a product, we have to investigate the external attributes that the user depends on. Inevitably, we are then dealing with attributes synonymous with software quality.

Fenton uses several general software quality models [5, 40], each of which proposes a specific set of external and internal quality attributes and their interrelationships. For example, maintainability is not restricted to code; it is an attribute of a number of different software products, including specification and design documents, and even test plan documents. There are two broad approaches to measuring maintainability; reflecting the external and internal views of the attribute. The external and more direct approach to measuring maintainability is to measure the maintenance process. If the process is effective, then we assume that the product is maintainable. The alternative internal approach is to identify internal product attributes (e.g., those relating to the structure of the product) and establish that they are predictions of the maintenance process.

All maintenance activities are concerned with making specific changes to a product. Once the need for a change is identified, the required efforts of implementing that change becomes the key characteristic of maintainability. Many measures of maintainability are expressed in terms of mean time to repair (MTTR). A number of the *complexity* measures have been correlated significantly with the level of maintenance effort. There is a clear intuitive connection among poor programming structure, poor documented products, and poor maintainability of a software product. We cannot say that a poorly structured module will inevitably be difficult to maintain. Rather, past experience tells us that such kinds of modules have had poor maintainability, so we should investigate the courses for a module's poor structure and perhaps restructure it.

Fenton and Neil indicate that the most significant benefit of software metrics is to provide information to support managerial decision-making during software development and testing [18]. Simple metrics are accepted by industrialists because they are easy to understand and simple to collect. Thus, Fenton and Neil try to use these simple metrics to build management decision support tools to handle the uncertainty as well as combine different evidences. They use Bayesian Belief nets as a means of handling decision-making under uncertain circumstances.

2.2.5 Karlsson’s Approach to Software Quality

Karlsson proposes a general reusability and maintainability models for C++ code [26]. In addition, he suggested that all measurements should be normalized so that they yield a value between zero and one, where a value close to zero indicates that the measured characteristic may cause problems, and a value close to one indicates that the corresponding characteristic is kept inside its limit. He chose to use the Kiviat diagram for metric presentation. This type of diagram represents parameters as vectors plotted on a circle. It provides an easy-to-grasp representation of assessment results and can be used for factors, criteria and metrics. Kalsson's models for reusability and maintainability are shown in Figure 2-2 [26].

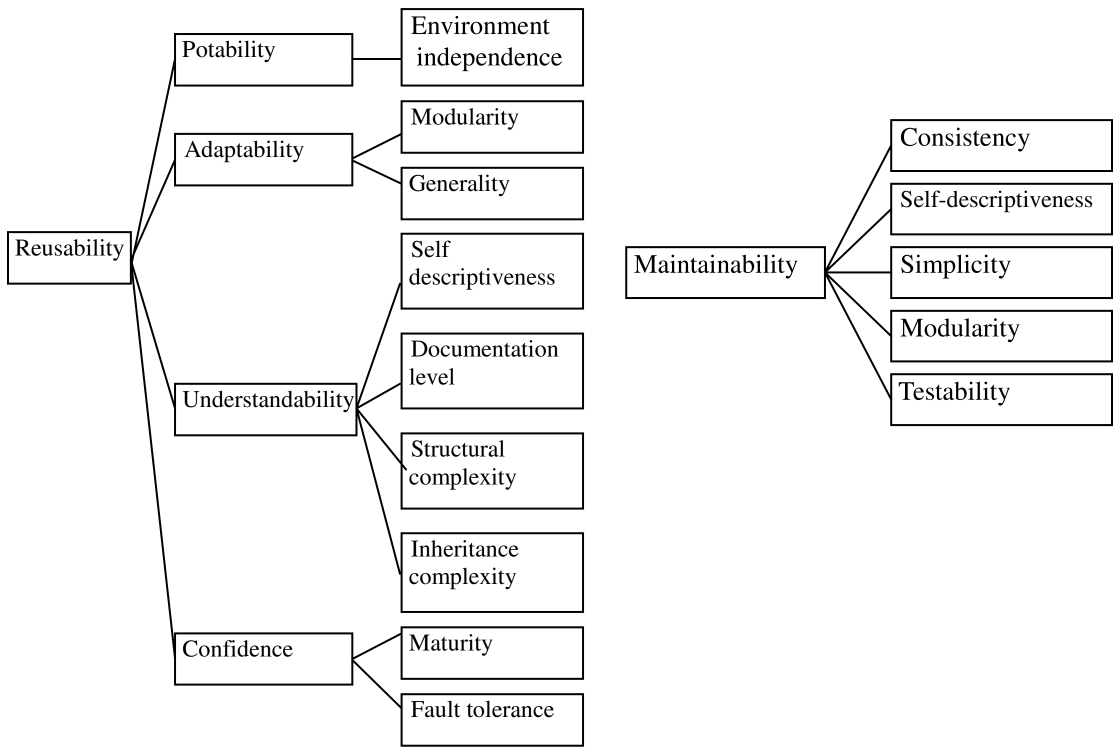


Figure 2-2: Karlsson's Reusability and Maintainability models

2.3 Software Metrics

2.3.1 Object-Oriented Metrics by Chidamber and Kemerer

A set of object-oriented metrics for measurement was proposed by Chidamber and Kemerer [12]. Since this metrics set is very popular, it has become the focus of discussion among many researchers. The resulting six metrics directly relate to design and implementation of object-oriented software.

Because previous metrics were criticized for their lack of theoretical basis, lack of desirable measurement properties, and for being too labor-intensive to collect, Chidamber and Kemerer developed six object-oriented metrics, and evaluated them analytically. They also developed an automated data collection tool to collect an empirical sample of these metrics. They then suggested ways in which the managers may use these metrics for process improvement using empirical data collected from two field sites[12]. We can also interpret these six metrics from the view point of quality.

Weighted Methods per Class (WMC): The WMC metric can be calculated from the sum of the complexities of the methods in a class where method complexity can be measured using cyclomatic complexity or assumed unity weights for all methods. WMC can be used as a predictor of how much time and effort is required to develop and maintain the class. A large value of WMC will have a great impact on the children of the class. Classes with large WMC value limit the possibility of reuse. This metric can be used as a measure of usability and reusability

Depth of Inheritance Tree (DIT): The DIT is the length of the longest path from a class node to the root of the tree. The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit. Thus its behavior could be predicted to be more complex. This metric can be used not only to evaluate reuse, but also to relate understandability and testability.

Number of Children (NOC): The number of children is the number of immediate subclasses subordinate to a class in a hierarchy. The measure is an indication of the potential influence a class can have on other classes in the design. The greater the number of children, the greater the likelihood of improper abstraction of the parent, and the potential misuse of subclassing. This also means greater reuse since inheritance is a form of reuse. If a class has a large number of children, it may require more testing for the methods of that class, thus increasing the testing time.

Coupling Between Object Classes (CBO): Coupling is a measure of the strength of association from one entity to another. CBO is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related classes on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application. The larger the number of couplings, the higher the sensitivity of changes would have to other parts of the design, and therefore maintenance is more difficult. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Lack of Cohesion of Methods (LCOM): Assume P is the number of null intersections and Q is the number of nonempty intersections between two methods. If P is greater than Q then LCOM is the differences between P and Q, else LCOM is zero. Two methods are considered related if both methods use the same instance variable(s). LCOM is based on method interconnection through instance variable reference. Effective object-oriented designs maximize cohesion in order to promote encapsulation. A large number of LCOM implies that the class is attempting to model more than a single concept and thus may need to be decomposed into several classes.

Response for a Class (RFC): The RFC is the cardinality of the set of all methods that could potentially be executed in response to a message to an object of the class. The larger the number, the more complex the testing of the class would be.

2.3.2 Class Cohesion and Coupling Measurement in object-oriented Systems

In addition to the Chidamber and Kemerer's metrics set, other metrics have been proposed to measure the coupling and cohesion of classes. Cohesion and coupling are two structural attributes whose importance is well-recognized in the software engineering community. Cohesion refers to the relatedness of module components within a class while coupling refers to how classes affect each other.

The higher the cohesion of a module, the easier the module is to develop, maintain, and reuse. Further the module becomes less fault prone. Some empirical evidence exists to support this theory for systems developed by object-based techniques [8].

Eder and colleagues [15] propose a framework aiming at providing comprehensive, qualitative criteria for cohesion and coupling in object-oriented systems.

They distinguish between three types of cohesion in an object-oriented software system: method, class and inheritance cohesion. Briand et al. also suggest the framework for coupling and cohesion measurement in object-oriented systems [9, 10] For each type, various degrees of cohesion exist. Within this framework, an analysis of the semantics of a given method or class is required to determine the degree of cohesion. Bieman and Kang define *class cohesion* measure based on dependencies between methods through their references to instance variables [4].

Briand, Morasco, and Basili design a measure to indicate cohesion for software developed in Objected-Oriented programming languages such as Ada [7]. Their primary cohesion measure, *Ratio of Cohesion Interactions* (RCI), is based on the number of interactions between subroutines, variable declarations, and type declarations. A method cannot affect another method through a type reference but it affects the effort required to understand the method.

During the analysis and design phase, and in any code evaluation at the module level, inter-module coupling is measured by the number of relationships between classes or between subsystems [35]. Class coupling should be minimized, in the sense of constructing autonomous modules [6]. Booch also notes that coupling occurs on a peer-to-peer basis and within a generalization/specialization hierarchy. The former should exhibit low coupling, i.e., closely coupled classes should be generalized in a hierarchy.

Berard differentiates between necessary and unnecessary coupling [3]. The rationale is that without any coupling, a system is useless. Consequently, for any given software solution there is a baseline or necessary coupling level. It is the elimination of extraneous coupling that should be the developer's goal. Such unnecessary coupling needlessly decreases the reusability of the classes [43].

Li and Henry offer the *Message Passing Coupling* (MPC) metric as "the number of send statements defined in a class" [34]. A similar approach is taken by Rajaraman and Lyu where they define coupling at the method level [45]. They define *Method Coupling* (MC) as the number of nonlocal references, and then gross these values up to the class totals and class averages. Chidamber and Kemerer define coupling between objects as "the number of other classes to which it is coupled and two classes are coupled when methods declared in one class use methods or instance variables defined by the other class" [12]. Their *Response For a Class* (RFC) metric counts the number of internal and external methods available to a class.

Fan-in and fan-out are the number of references made from outside a class to entities defined within the class and the number of references made from within a class to entities defined outside the class, respectively. A low fan-out is desirable since a high fan-out is a characteristic of a large number of classes needed by the particular class in question [3]. A high fan-out also represents a class coupling to other classes and thus an "excessively complex dependence" on other classes [23]. A high fan-in normally represents a good object design and a high level of reuse. Since summations of these two numbers are the same for a system, it is not likely to maintain a high fan-in and a low fan-out across the whole system.

2.3.3 Profile Software Complexity

Thomas McCabe proposes a measure of software called cyclomatic complexity

[39]. Making use of graph theory, McCabe postulates that software with a large number of possible control paths would be more difficult to understand, more difficult to maintain, and more difficult to test. One of the problems of using cyclomatic numbers as a measure of software complexity is that it produces just a single value to describe a module's complexity.

An alternative approach proposed by McQuaid is a fine-grained approach to computing and visualizing complexity [41]. Unlike cyclomatic complexity, the profile metric is computed and shown on a statement-by-statement basis. It defines complexity in terms of a program statement's content, much like Halstead's effort measurement, and context, which is the environment in which the statement occurs. The context complexity can be further refined into three measures: inherency, reachability, and breadth complexity.

The content complexity tries to measure the information quantity and not quality within a measurable unit. The context complexity tries to measure the location of a measurable unit within the source code. The profile complexity is designed such that the context complexity is the baseline complexity, with the content complexity riding on this baseline. The rationale of this design is to provide easy identification of complex clusters. When a cluster is identified, the content complexity can be used to isolate the heavy segment in the cluster.

2.4 Framework for Coupling and Cohesion Measurement

Briand et al. propose a unified framework for coupling and cohesion measurement for object-oriented programs [9, 10]. The objective of the framework is to support the comparison and selection of existing coupling and cohesion measures with respect to a particular measurement goal. In addition, the framework provides guidelines to support the definition of new measures with respect to a particular measurement goal when no measures exist. The framework, if used as intended, will:

- Ensure that measure definitions are based on explicit decisions and well understood properties,
- Ensure that all relevant alternatives have been considered for each decision made,
- Highlight dimensions of coupling for which there are few or no measures defined.

The framework for coupling consists of six criteria, each criterion determining one basic aspect of the resulting measure. The six criteria of the framework are:

1. *Type of connection*: Choosing the type of connection implies choosing the mechanism that constitutes coupling between two classes. Table 2-1 summarizes the possible types of connections.
2. *Direction of connection*: Fan-in refers to connection to the module being studied. Fan-out measures the connection to other modules from the module.
3. *Granularity of the measure*: Domain of the measure and how to count coupling connections.
4. *Stability of server*: Stable classes are not subject to change in a new project but unstable classes are subject to modification in a new project.
5. *Direct or indirect coupling*: Counting direct connections only or also indirect

connections.

6. *Inheritance*: Inheritance-based vs. noninheritance-based coupling, and how to account for polymorphism, and how to assign attributes and methods to classes.

Table 2-1: Connection Types of Coupling

#	Class 1	Class 2	Description
1	Attribute *a* of class *c*	Class *d*, $d \neq c$	Type of attribute: Class *d* is the type of *a*
2	Method *m* of class *c*	Class *d*, $d \neq c$	Type of parameter: Class *d* is the type of a parameter of *m*, or the return type of *m*
3	Method *m* of class *c*	Class *d*, $d \neq c$	Type of local variable: Class *d* is the type of a local variable of *m*
4	Method *m* of class *c*	Class *d*, $d \neq c$	Type of invoked method: Class *d* is the type of a parameter of a method invoked by *m*
5	Method *m* of class *c*	Attribute *a* of class *d*, $d \neq c$	Attribute reference: m references *a*
6	Method *m* of class *c*	Method *m'* of class *d*, $d \neq c$	Method invocation: *m* invokes *m'*
7	Class *c*	Class d, $d \neq c$	Inheritance: Class *d* the child class of class *c*

Table 2-2: Connection Types of Cohesion

#	Element 1	Element 2	Description
1	Method *m* of class *c*	attribute *a* of class *c*	Attribute reference: *m* references *a*
2	Method *m* of class *c*	Method *m'* of class *c*	Method invocation: *m* invokes *m'*
3	Method *m* of class *c*	Method *m'* of class *c*, $m \neq m'$	Attribute sharing: *m and m'* reference an attribute *a*

When specifying a cohesion measure, the following criteria of the cohesion framework must be considered.

1. *Type of connection*: What makes a class cohesive. Table 2-2 summarizes the possible types of connections.
2. *Domain of measure*: Objects to be measured (methods, classes, and system)
3. *Direct or indirect connections.*
4. *Inheritance:* How to assign attributes and methods to classes, how to account for polymorphism.
5. *How to account for access methods and constructors.*

3 SELECTING THE SOFTWARE QUALITY METRICS

This research focuses on the design, development, and evaluation of an automated measurement tool for object-oriented programs. More specifically, the measurement tool is targeted for software quality measurement in terms of reusability and maintainability.

3.1 Quality Factors to be Measured

For the development of practical and automated metric model, we suggest top-down and bottom-up metrics framework for source code of object-oriented software. In this framework, we develop quality measurement model of object-oriented software in terms of quality factors during implementation or maintenance phase.

In most of the stated software metric models, each software quality aspect (e.g., maintainability and reusability) is expressed in terms of a hierarchy of factors and criteria. The higher-level factors in the hierarchy typically represent the management's point of view; while the lower level criteria represent the code-related measurement, i.e., each criterion is normally a function of the raw attributes of the software. The structure of the hierarchy is largely dependent upon the nature of the software and the desire of the project team.

In a quality hierarchy, code-related criteria are the foundation by which quality is defined, judged, and measured. The measurement represented by a quality metric can be obtained during all phases of the software development to provide an indication of progress towards the desired product quality. In this research, *reusability* and *maintainability* are the two focused factors that can be applied to the source code to provide good quality indication.

3.1.1 Maintainability

Software maintenance includes all post implementation changes made to a software entity. Maintainability refers to the easiness or toughness of the required efforts to do the changes. Before any changes can be made to a software entity, the software must be fully understood. After the changes have been completed, the revised entity must be thoroughly tested as well. For this reason, maintainability can be thought of as three attributes: understandability, modifiability, and testability. Harrison sees software complexity as the primary factor affecting these three attributes [21], while modularity, information hiding, coupling, and cohesion are closely related to the complexity (See Figure 3-1).

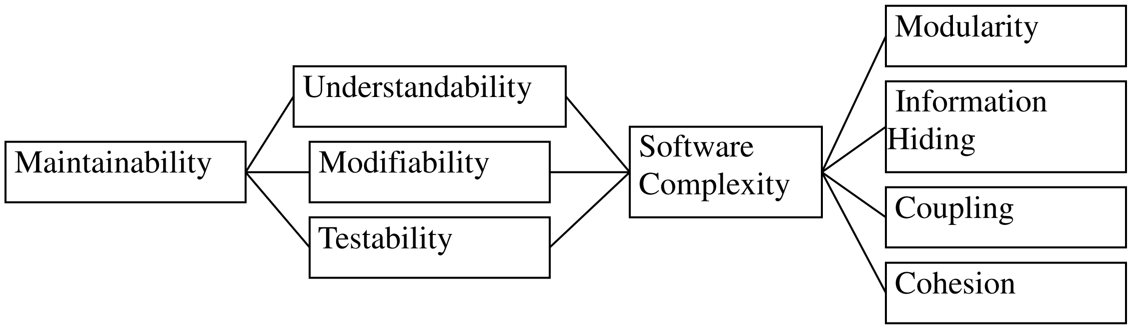


Figure 3-1: Harrison's Maintainability Model

Since maintenance accounts for a large portion of a software product's cost, if properly improved, it has a great potential to reduce the total software cost. However, without meaningful measure of maintainability, there would be no substantial way of verifying improvement, even though certain actions may seem beneficial [21]. Historically, maintainability can only be measured after actual maintenance has been performed. In the same application, the time required per module to determine the changes indicates understandability; the time to change indicates modifiability; the time to test indicates testability.

Instead of collecting the measurement after the product is completed, our approach is to *forecast* the maintainability based on the source code and display the measurement at any time the programmer wishes. The source code can be at any stage of the development, and the measurement will be computed automatically. This will provide a real time *grade* of the software in the dimension of maintainability.

3.1.2 Reusability

When a reusable code is written, the intended users should be somewhat identified. If a code is to include the functionality that every user would want, the resulting code would be too expensive to produce and too difficult to use. Code reuse has been common in practice. But, many difficulties are associated with code reuse:

1. Code identification: It is difficult to identify a piece of reusable code. Many times, programmers reuse only a small fraction of their own or their colleagues' code.

2. Code validation and verification: There is usually little assurance that the reused code is correct.

3. Code dependency: It is a nontrivial task to separate a desired piece of code from an entangled chunk of software with complex dependency.

4. Code modification: In addition to the necessary changes, the reused code may implicitly conflict with the new context.

5. Execution environment: The reused code might assume things that are not true in the new environment. This may result in degraded performance.

With careful planning and implementation, many of these difficulties can be avoided. This requires a reusable code to possess certain properties that our proposed measurement will quantify. A static analysis of a source code in any stage of development can provide instant feedback to the programmer, the quality of the code in the sense of reusability. This would encourage programmers to ensure that the completed code provides good reusability quality before it is discovered too late. The measurement can also allow the manager of a software project to evaluate the quality and reward the programmers accordingly.

3.2 Quality Measurement Model

In this research, a quality measurement model is proposed and its metric set is developed. The overall steps to construct the model and metrics are in Figure 3-2. We obtain subfactors from the software quality factors and measurement types from the essential properties of reusable and maintainable code, then match the subfactors and the measurement types, and create a quality model for reusability and maintainability. Several metrics are defined for each measurement type. Based on the created quality model and the defined metrics, an automated metric tool is implemented, and the measured metrics from the tool are validated through empirical study.

Subfactors and measurement types are discussed in detail in the following sub-sections.

We use a top-down and bottom-up approach to develop this quality model. Its methodology is shown in Figures 3-3. From the top down, we first divide the quality factors (i.e., reusability and maintainability) into subfactors in accordance with procedures of performing reuse and maintenance. The divided subfactors are identification, separation, modification, validation, and adaptation of a module. By dividing the factors into five subfactors, the vague concepts of the factors become clearer.

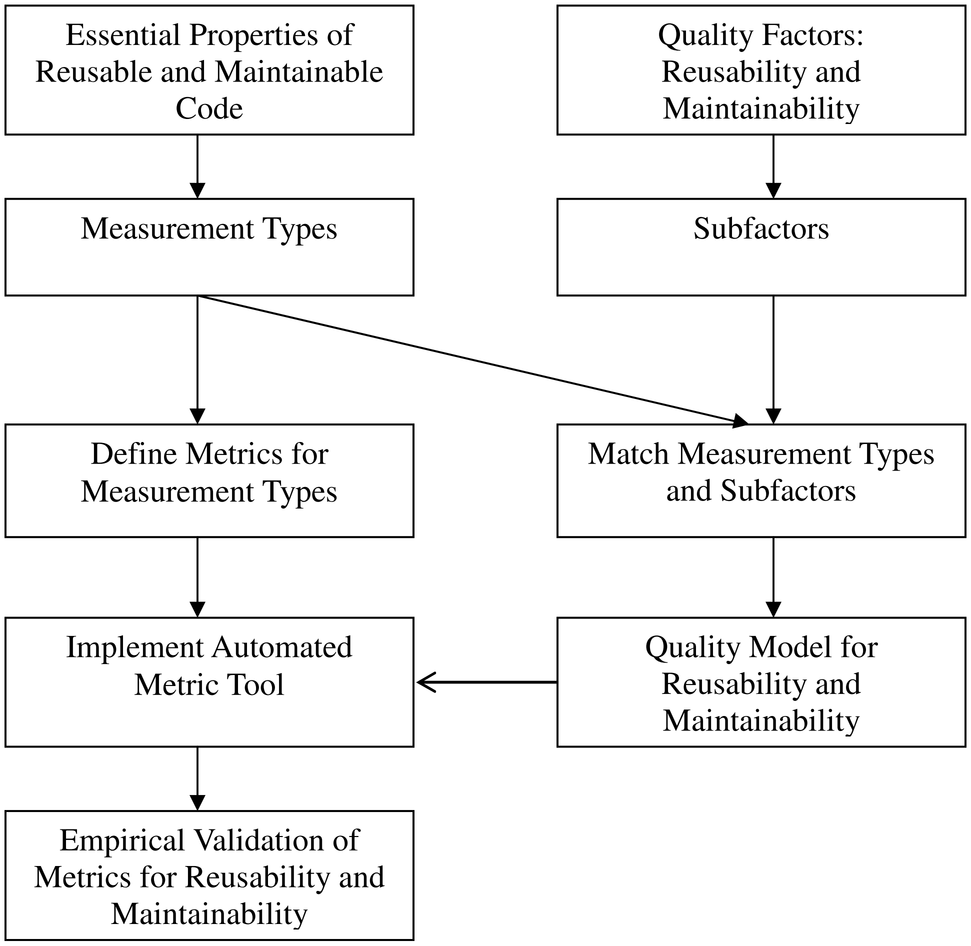

Essential Properties of Reusable and Maintainable Code
Quality Factors: Reusability and Maintainability
Measurement Types
Subfactors
Define Metrics for Measurement Types
Match Measurement Types and Subfactors
Implement Automated Metric Tool
Quality Model for Reusability and Maintainability
Empirical Validation of Metrics for Reusability and Maintainability

Figure 3-2: Steps for constructing the quality model and metric set

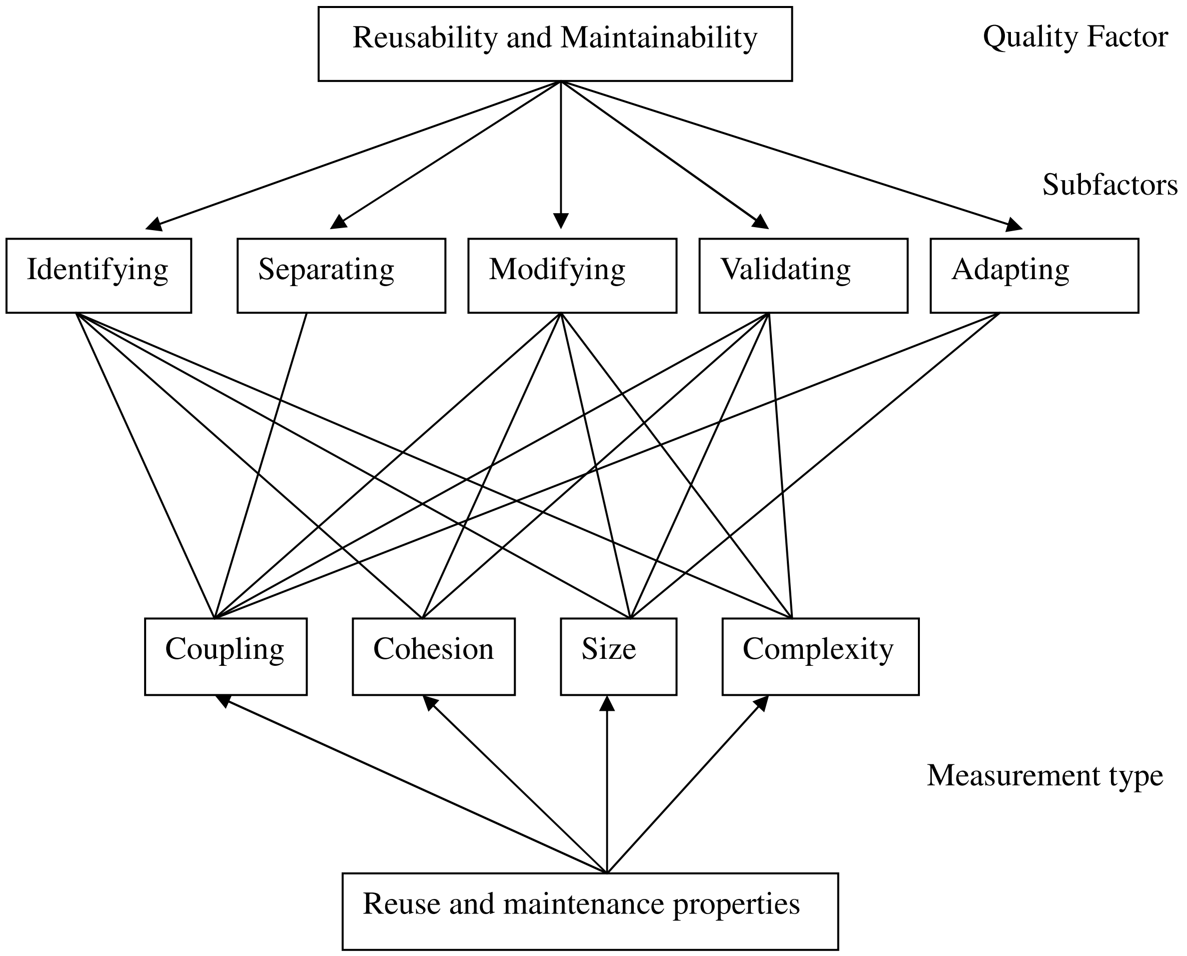


Figure 3-3: Flow of how the subfactors are connected to the metrics

From the bottom up, we propose the desirable features for reusability and maintainability, and apply these features to understand software for reuse and maintenance purposes. These properties can be derived from the source code. The selected measuring properties include External dependency, Cohesion, Information hiding, Size, Complexity, Easy understanding, Proven reliability, Reuse frequency, and Standardization as shown in Table 3-1. Each measuring property has its own measurement type. In this research, we mainly focus on four measurement types

(Coupling from External dependency, Cohesion, Size, and Complexity) from the reuse and maintenance properties. Information hiding, Easy understanding, Proven reliability, Reuse frequency, and Standardization will not be considered due to the difficulties of collecting measurement data.

The definitions of the selected measurement types are summarized in Table 3-2. The Coupling from the external dependency defines the interdependency of a class to other classes in a source code. The Cohesion assesses the relationship of methods and attributes in a class. The Size measures the number of methods and attributes, and lines of code in a class. The Complexity measures the degree of difficulty in understanding the structure of classes.

The important issue in this model construction is to establish links between the subfactors and the measurement types. We want to map the subfactors into the measurement types since each measurement type can be defined as a metric and computed through a simple expression and each metric plays an important role as a key factor in measuring the quality of a software system. Therefore, measuring the measurement types as metrics becomes equal to measuring the factors of software quality. The links between subfactors and measurement types are established in Figure 3-3, and their relationship is presented in Section 3.2.2.

Table 3-1: Essential properties of reusable and maintainable code

- External dependency
    - Requires no separation from any containing code.
    - Requires no changes to be used in a new program
    - Components do not interface with its environment.
    - Low fan-in and fan-out
    - Has more calls to low-level system and utility functions.
- Cohesion
    - Component exhibits high cohesion
- Information hiding
    - Has few input-output parameters.
    - Interface is both syntactically and semantically clear.
    - Interface is written at appropriate (abstraction) level.
- Size
    - Small
- Complexity
    - The lower the values of complexity metrics, the higher the programmer's productivity.
    - Low module complexity
- Easy understanding
    - Component and interface are readable by person other than the author
    - Component is accompanied by documentation to make it traceable
    - Easy to find and understand
    - In-line comments
- Proven reliability
    - Thorough testing and low error rates
    - Reasonable assurance that it is correct
- Reuse frequency
- Standardization
    - Component is standardized in the areas of invoking, controlling, terminating its function, error-handling, communication, and structure.

Table 3-2: Measurement type

Measurement Type	Definition based on class	Measuring	Measuring Properties
Coupling	The interdependency of a class to other classes in a system. Measure of the number of other classes that would have to be accessed by a class in order for that class to function correctly and the number of other classes that use the methods or attributes in this class	Reference methods and attributes among classes	External dependency
Cohesion	The relatedness of methods and attributes in a class.	Reference methods and attributes in a class	Cohesion
Size	The numbers of methods, attributes, and lines in a class	The numbers of methods, attributes, and lines in a class	Size
Complexity	The degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships	Cyclometic complexity of methods in a class	Complexity

3.2.1 Definition of subfactors and measurement type

We choose reusability and maintainability as our measurement factors. Dividing these factors into subfactors helps to find appropriate measurement types. In case of reusing and/or maintaining an existing code, several procedures should be accomplished:

- Identification: When a programmer tries to reuse or maintain an existing source code, he/she needs to locate and understand the code to match the desired purposes.

- Separation: After a programmer locates and understands the identified code, he needs to take apart the code from its containing program.
- Modification: Before a programmer reuses the separated code unit he may need to change the unit to meet the required function or to make the unit fit to the new environment.
- Validation: Error checking will be an important step to make the unit reliable, so a programmer needs to check for errors.
- Adaptation: He/She has to carefully adapt the modified code into the new application to prevent any conflicts.

To derive measurement types for reusability and maintainability, we collect *properties* of reusable and maintainable software from previous research [44] [46]. These essential properties are listed and explained in Table 3-1. Based on these properties, the following measurement types are derived. Each measurement type came from each measuring property in Table 3-2.

- Coupling: The interdependency of a class to other classes in a system. It is a measure of the number of other classes that would have to be accessed by a class in order for that class to function correctly and the number of other classes that use the methods or attributes in this class.
- Cohesion: The relatedness of methods and attributes in a class. Components of a class should be designed for a single purpose. Thus, the class that has low cohesion needs to be decomposed.

- Size: This includes counting lines of code with several options (e.g., ignore blank and comments lines), number of methods and attributes in a class.
- Complexity: The degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships. The structure of a method that has high complexity metric value should be inspected and simplified. Statement level complexity is also considered to locate complex area of source code.

3.2.2 Relationship between subfactors and measurement types

Table 3-3 summarizes the relationship between measurement types and subfactors. A plus symbol (+) in the table indicates that the measurement type has a positive influence on a subfactor, and a minus symbol (-) indicates negative influence.

Table 3-3: Relationships between subfactors and measurement types

Measurement Type / Subfactor	Coupling	Cohesion	Size	Complexity
Identification	-	+	-	-
Separation	-			
Modification	-	+	-	-
Validation	-	+	-	-
Adaptation	-		-	

<u>Relationship from coupling to subfactors</u>

High import coupling of a class indicates strong dependency on other classes, their methods, and attributes. Import coupling may be relevant to the following subfactors:

- *Identification*: To understand a method or class, we must know about the services the method or class uses.
- *Separation*: High import coupling obstructs separating the code from its containing program.
- *Adaptation*: If a class depends on a large amount of external services, it will be more difficult to reuse it in other systems.

High export coupling of a class means that the class is used by many other classes, their methods, and attributes. Export coupling may be relevant to the following subfactors:

- *Modification*: If a method may be invoked by many other methods, any change to the method affects the invoking methods. Any defect in a class with high export coupling is more likely to propagate to other parts of the system. Such defects are more difficult to isolate. In that respect, classes with high export coupling are particularly critical. An export coupling measure could therefore be used to select classes that should undergo special (effective and may be costly) verification or validation processes.
- *Validation*: A class with high export coupling can be difficult to test. If defects propagate to other parts of the system to cause failure there, they may not be detected when testing the class in isolation.

Relationship from cohesion to subfactors

Stevens et al. define cohesion as a measure of the degree to which the elements of a module are together [48]. Some empirical evidence supports that the higher the cohesion of a module, the easier the module is to develop, maintain, and reuse [11, 8].

If elements of a module are not related to each other, the design of the module most likely is not appropriate. Thus, we define cohesion to have positive impact on identifying, modifying, and validating.

Relationship from size to subfactors

Usually a large size module has more attributes and methods, thus it will take more time to understand, modify, validate, and adapt it. Size measurement type probably can be included in other measurement types like complexity.

Relationship from complexity to subfactors

High complexity is an obstacle to understand and modify a module. Validating a module is also difficult as its complexity increases.

3.3 Metrics for measurement types

In forming the quality model, a framework is designed to find the most influential metrics for individual reuse and maintain properties. In the framework, we identified a few sets of metrics to characterize software written in Java. They are listed in Tables 3-4 through 3-6. Each of the metrics was carefully evaluated and experimented for its capability to accurately measure a reusability and/or maintainability property in this dissertation. The rationale used in this experimental test as follows.

These metrics were chosen because they are representatives of metrics based on the measurement types described in Section 3.2. They are also computable using the automated measurement tool implemented for this research and are potential indicators whether or not a class is reusable and maintainable. In each of the metric definitions, *C* represents a class, M represents a method, *S* represents a system composed of classes, *D* represents a domain (i.e., method, class, or system).

We used very primitive forms of coupling and cohesion metrics because these metrics are used to measure subfactors rather than quality factors of a system. All coupling and cohesion metrics assume direct and non-inherited based relationship. Each coupling and cohesion metric is classified by the type of connection and then divided, in detail, by direction of connection and domain level (i.e., class or system).

In the following sub-sections, we describe those metrics in detail, including size, complexity, coupling, and cohesion metrics. They will be investigated throughout the remainder of this dissertation.

3.3.1 Size Metrics

Size metrics measure the number of methods and attributes in a class and the lines of code of a class. Those are defined in Table 3-4. We have three domains for the metrics (method, class, and system domain) and each domain has its own metrics. For a method M, LOC(M) measures the lines of code for the method, and for a class C, LOC(C) measures the lines of code for the class. NOM(C) counts the number of methods in a class and NOA(C) counts the number of attributes in a class.

The size metrics defined for a system domain are LOC(S), aLOCC(S), aLOCM(S), aNOM(S), aNOA(S), and NOC(S). LOC(S) is the lines of code for a system. aLOC(S) and aLOC(S) calculate averaged LOC for classes and methods respectively for a system. aNOM(S) and aNOA(S) compute average number of methods and attributes in a class, and NOC(S) is the number of classes in a system.

From the past experience, we believe that large classes may suffer from poor design. Large size metrics and more functions in a class normally make it more difficult to understand the class. In an iterative development process, more and more functionality is added to a class over time. The danger is that, eventually, many unrelated responsibilities are assigned to a class. As a result, it has low functional cohesion. This in turn negatively impacts the reusability, and maintainability of the class. Therefore, large classes should be reviewed for functional cohesion. If there is no justification for the large size, the class should be considered for refactoring, for instance, and extracting parts of the functionality to make separate and more cohesive classes.

Table 3-4: Size metrics

(a) Size metrics for a method

Symbol	**Description**	**domain**
LOC(M)	LOC for a method	method

(b) Size metrics for a class

Symbol	**Description**	**domain**
LOC(C)	LOC for a class	class
NOM(C)	number of methods in a class	class
NOA(C)	number of attributes in a class	class

(c) Size metrics for a system

Symbol	**Description**	**domain**
LOC(S)	LOC for a system	system
aLOCC(S)	average LOC for classes in a system	system
aLOCM(S)	average LOC for methods in a system	system
aNOC(S)	average NOM for classes in a system	system
aNOA(S)	average NOA for a class in a system	system

3.3.2 Complexity Metrics

Complexity metrics measure the degree of difficulty in understanding internal and external structure of classes and their relationships. In this research, Cyclometic complexity of methods in a class is used, and based on this, we define three complexity metrics for method (Cx(M)), class (aCx(C)), and system (aCx(S)) domains in Table 3-5.

High method complexity in a class can lead to decreased understandability and therefore decreased reusability and maintainability. Also, testing such a class is more difficult.

Table 3-5: Complexity metrics

(a) Complexity metrics for a method

Symbol	**Description**	**domain**
Cx(M)	McCabe complexity of a method	method

(b) Complexity metrics for a class

Symbol	**Description**	**domain**
aCx(C)	average Cx(M) in a class	class

(c) Complexity metrics for a system

Symbol	**Description**	**domain**
aCx(S)	average Cx(C) in a system	system

3.3.3 Coupling Metrics

As we mentioned in section 2.4, the unified framework for coupling provides a guideline to select coupling metrics for a particular measurement goal (Reusability and Maintainability for this research).

Based on the first criterion of the unified framework for coupling (i.e., type of connection), we study seven types of possible connection between two classes. Therefore we define seven coupling metrics for measuring different connection types as in Table 3-6. The seven metric symbols defined in the table are used to define actual coupling metrics based on the criteria of the framework for measuring coupling described in 2.4. The defined metrics for the seven connection types whose domain is class are cplTA(C), cplTP(C), cplTL(C), cplTPM(C), cplIAR(C), cplMI(C), and cplPC(C).

Metrics *cplTA(C), cplTP(C)*, and *cplTL(C)* measure the number of *Type of attribute* connection, the number of *Type of parameter* connection, and the number of *Type of local variable* connection of a class, respectively.

Table 3-6: Connection type for coupling

Symbol	Connection Type	Class C	Class D	Description
cplTA(C)	Type of Attribute	Attribute *a* of class *c*	Class *d*, $d \neq c$	Class *d* is a type of *a*
cplTP(C)	Type of Parameter	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a parameter of *m*, or the return type of *m*
cplTL(C)	Type of local variable	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a local variable of *m*
cplTPM(C)	Invoked method type	Method *m* of class *c*	Class *d*, $d \neq c$	Class *d* is the type of a parameter of a method invoked by *m*
cplAR(C)	Attribute reference	Method *m* of class *c*	Attribute *a* of class *d*, $d \neq c$	*m* references *a*
cplMI(C)	Method invocation	Method *m* of class *c*	Method *m'* of class *d*, $d \neq c$	*m* invokes *m'*
cplPC(C)	Parent-Child	Class *c*	Class d, $d \neq c$	Class *d* is a child class of class *c*

Metrics *cplTPM(C), cplAR(C)*, *cplMI(C)*, and *cplPC(C)* measure the number of *Invoked method type* connection, the number of *Attribute reference* connection, the number of *Method Invocation* connection, and the number of *Parent-Child* connection of a class, respectively. We have chosen the first criterion, *Type of Connection*, to create these basic metric symbols for all the connection types. For the second criterion of the unified framework for coupling (i.e., *Direction of connection*), each coupling metric in Table 3-6 consists of fan-out coupling("using") and fan-in coupling("used") components, which we discuss in the following.

For example, the metric *cplTA(c)* is decomposed into *cplTAout(c)* and *cplTAin(c)* according to the direction of the connection. *cplTAin(c)* measures the connections to the target class *c* from other classes and *cplTAout(c)* measures the connections to other classes from the target class *c*. We measure the *cplTA(c)* as the sum of *cplTAin(c)* and *cplTAout(c)*.

Fan-out coupling measures the degree to which a class has knowledge of, uses, or depends on other classes. To reuse a class with high fan-out coupling in a new context, all the required services must also be understood and reused together. Therefore, high fan-out coupling can decrease the reusability of a class.

Fan-in coupling measures the degree to which a class is used by, depended upon, by other elements. Changing a class with high fan-in coupling may affect other classes which depend on the class. Therefore high fan-in coupling can decrease the maintainability of the class.

Coupling connections cause dependencies among classes, which, in turn, have an impact on maintainability (a modification of a class may require modifications to its connected classes) or reusability (to reuse a class may require reuse connected classes together). Thus, we could say that a principle to improve reusability and maintainability is to minimize coupling, and coupling metrics also greatly help identify problematic classes to be reused or maintained.

We can apply these coupling metrics to a system domain. For example, *aCplTA(s)* is defined as the averaged *CplTA(c)* of classes in system ***s*** and measures the averaged *type of attribute* coupling metrics of classes in system ***s***.

For the third criterion (i.e., *Granularity of the measure*) of the unified framework for coupling, we define a class as the domain for coupling metrics.

For the fourth criterion (i.e., *Stability of server*) of the unified framework for coupling, we didn't define anything because we don't measure the stability of server.

For the fifth criterion (i.e., *Direct or indirect coupling*) of the unified framework for coupling, we only choose and measure direct coupling.

For the sixth criterion (i.e., *inheritance*) of the unified framework for coupling, we choose non-inheritance based coupling. We assign attributes and methods to the class which the attributes and methods are defined, not to their parent classes.

We have a sample code (Figure 3-4) showing couplings between classes and coupling metric values obtained by the system implemented in this research. Each class is counted either a fan-out coupling or a fan-in coupling to other classes by extending or declaring a class.

For instant, class *A* is counted a coupling with class *F* by extending it. In this case, classes *A* and *F* establish a parent-child relationship (one of the seven connection types), which *A* is a child and *F* is a parent. Therefore, we count cplPC fan-out meric value for class *A* (cplPCout(A) = 1) and cplPC fan-in metric value for class *F* (cplPCin(F) = 1). In a similar way, a coupling occurs between class *A* and class *B* by declaring *B* in class *A*. In this case, the type of attribute connection is established, which attribute *b* in class *A* is declared by class *B* as its type, and class *A* makes cplTA fan-out metric value counted 1 and class *B* makes cplTA fan-in metric value counted 1. Symbols ---> and <--- indicate a fan-out coupling and a fan-in coupling occurred in a class, respectively.

Fan-In/Fan-Out Coupling between Classes			Metric Values
public class A extends F{	--->	cplPC	cplPCout(A) = 1
B b;	--->	cplTA	cplTAout(A) = 1
public void ma(D c){	--->	cplTP	cplTPout(A) = 1
E e;	--->	cplTL	cplTLout(A) = 1
e.me(D d);	--->	cplMI, cplTIM	cplMIout(A) = 1
e.i++;	--->	cplAR	cplTIMout(A) = 1
}			cplARout(A) = 1
}			
public class B{	<---	cplTA	cplTAin(B) = 1
}			
public class D{	<---	cplTP, cplTIM	cplTPin(D) = 1
}			cplTIMin(D) = 1
public class E{	<---	cplTL	cplTLin(E) = 1
int i;	<---	cplAR	cplARin(E) = 1
public void me(D d){	<---	cplMI	cplMIin(E) = 1
}			
}			
public class F{	<---	cplPC	cplPCin(F) = 1
}			

Figure 3.4: Fan-in/Fan-out coupling between classes

Table 3-7: Cohesion metrics

Symbol	Connection Type	Element 1	Element 2	Description
cohAR(C)	Attribute reference	Method *m* of class *c*	Attribute *a* of class *c*	Attribute reference: *m* references *a*
cohMI(C)	Method invocation	Method *m* of class *c*	Method *m'* of class *c*	Method invocation: *m* invokes *m'*
cohAS(C)	Attribute sharing	Method *m* of class *c*	Method *m'* of class *c*, $m \neq m'$	Attribute sharing: *m and m'* reference an attribute *a*

3.3.4 Cohesion Metrics

We also defined cohesion metrics based on the framework for cohesion measurement (See Section 2.4).

For the first criterion (i.e., *type of connection*) of the unified framework for cohesion, we define three cohesion metrics with different connection types among the components (i.e., methods and attributes) in a class. *cohAR(c)* measures the number of *attribute reference* connections of a class *c*, *cohMI(c)* measures the number of *method invocation* connections of a class, and *cohAS(c)* measures the number of *attribute sharing* connections of a class. Table 3-7 shows the three cohesion metrics based on the connection type.

For the second criterion of the unified framework for cohesion (i.e., *Domain of measure*), we can apply these cohesion metrics to class and system domains. For example, *aCohAR(s)* is defined as the averaged *CohAR(c)* of classes in system ***s*** and measures the averaged *attribute reference* cohesion metrics of classes in the system.

For the third criterion (i.e., *Direct or indirect connections*) of the unified framework for cohesion, we only choose direct connection and measure the direct connection.

For the fourth criterion (i.e., *inheritance*) of the unified framework for cohesion, we choose non-inheritance based cohesion. We assign attributes and methods to the class which the attributes and methods are defined, not to its parent class. For the fifth criterion (i.e., *access methods and constructors*) of the unified framework for cohesion, we measure the cohesion for the access methods and constructors.

Cohesion is the degree to which the methods and attributes in a class are related. The higher connectivity between methods and attributes means the higher cohesion, and a low cohesive class has been assigned many unrelated responsibilities. Consequently, the low cohesive class is more difficult to understand and harder to maintain and reuse. Therefore classes with low cohesion should be considered for refactoring, for instance, by extracting parts of the functionality to separate classes with clearly defined responsibilities.

We have a sample code (Figure 3-5) showing cohesion in a class and cohesion metric values obtained by the system. Class A has two methods ma and mb, and method ma makes a method invocation connection by invoking method mb, thus the system calculates a choMI metric value of one (cohMI(A) = 1). For the cohAS metric, methods ma and mb establish an attribute sharing connection by sharing an attribute i, thus cohAS cohesion metric value of the class is calculated (cohAS(A) =1).

Cohesion in a Class	Metric Values
`public class A {`	
`  int i;    int j;`	
`  public void ma(){`	
`    mb();                 --->    cohMI`	
`    i++;                  --->    cohAS`	
`    j++;                  --->    cohAR`	cohMI(A) = 1
`  }`	cohAS(A) = 1
`  public void mb(){`	cohAR(A) = 1
`    i++;                  --->    cohAS`	
`  }`	
`}`	

Figure 3-5: Cohesion in a class and metric values

4 AN AUTOMATED MEASUREMENT TOOL

4.1 Automated Measurement Tool Architecture

Java Measurement Tool (JamTool) is a software measurement environment to analyze program source code for software reuse and maintenance. It is especially designed for object-oriented software. This tool measures attributes from Java source code, collects the measured data, computes various object-oriented software metrics, and presents the measurement results in a tabular form. The tabular interface of the tool provides software developers the capabilities of inspecting software systems, and makes it easy for the developers to collect the metric data and to use them for improving software quality. By browsing *reusable units* and *maintainable units*, a developer can learn how to reuse certain software entity and how to locate problematic parts. The application of this easy-to-use tool significantly improves a developer's ability to identify and analyze quality characteristics of an object-oriented software system.

The intended application domain for JamTool is small-to-middle sized software developed in Java. The acceptance of Java as the programming language of choice for industrial and academic software development is clearly evident. The overall system architecture of the JamTool is shown in Figure 4-1, in which solid arrows indicate information flow. The key components of the architecture are: 1) User Interface, 2) Java

Code analyzer, 3) Internal Measurement Tree, 4) Measurement Data Generator, and 5) Measurement Table Generator.

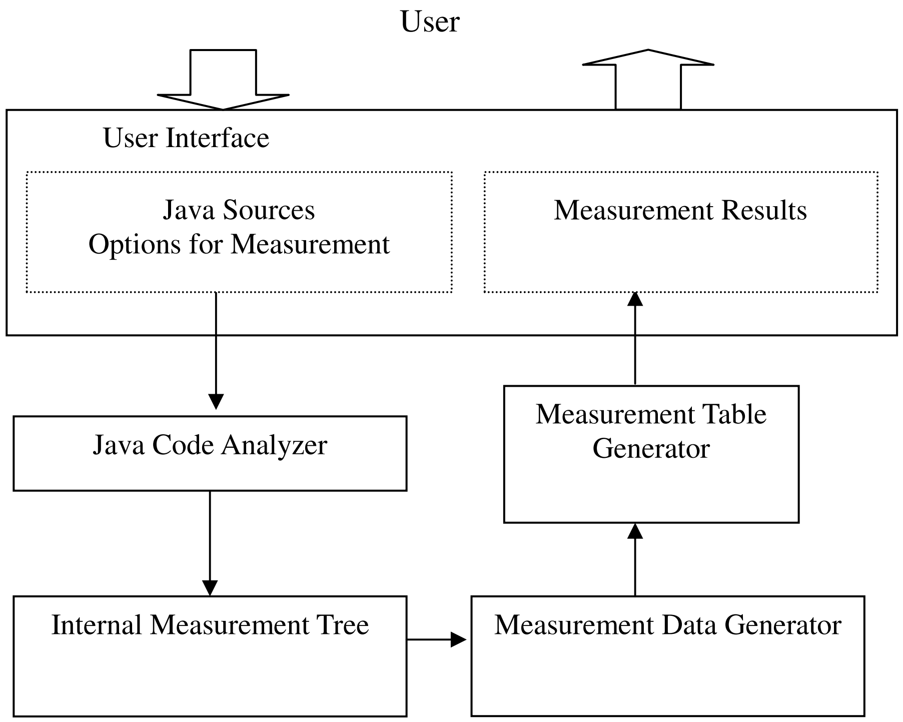


Figure 4-1: Architecture of JaMTool

Each key component works as a subsystem of overall system. The Java Code analyzer syntactically analyzes source code and builds an Internal Measurement Tree (IMT) which is a low level representation of classes, attributes, methods, and relationships of the source code. Then the Measurement Data Generator takes the IMT as an input, collects the measurement data, and generates the size, complexity, coupling and cohesion metrics of classes in the original source code. Those measurement results as well as the other metrics are displayed in a tabular representation through the Measurement Table Generator subsystem. With this interface of tabular form, software developers can easily analyze the characteristics of their own program.

4.1.1 Java Code Analyzer

Java Code analyzer is built by using a Sun Microsystem's popular JavaCC parser generator. It syntactically analyzes Java source code to build an internal measurement tree (IMT) that contains all the information needed to produce measurement results. It performs complete analysis on the source code thus identifies all syntactic errors during the building of the IMT.

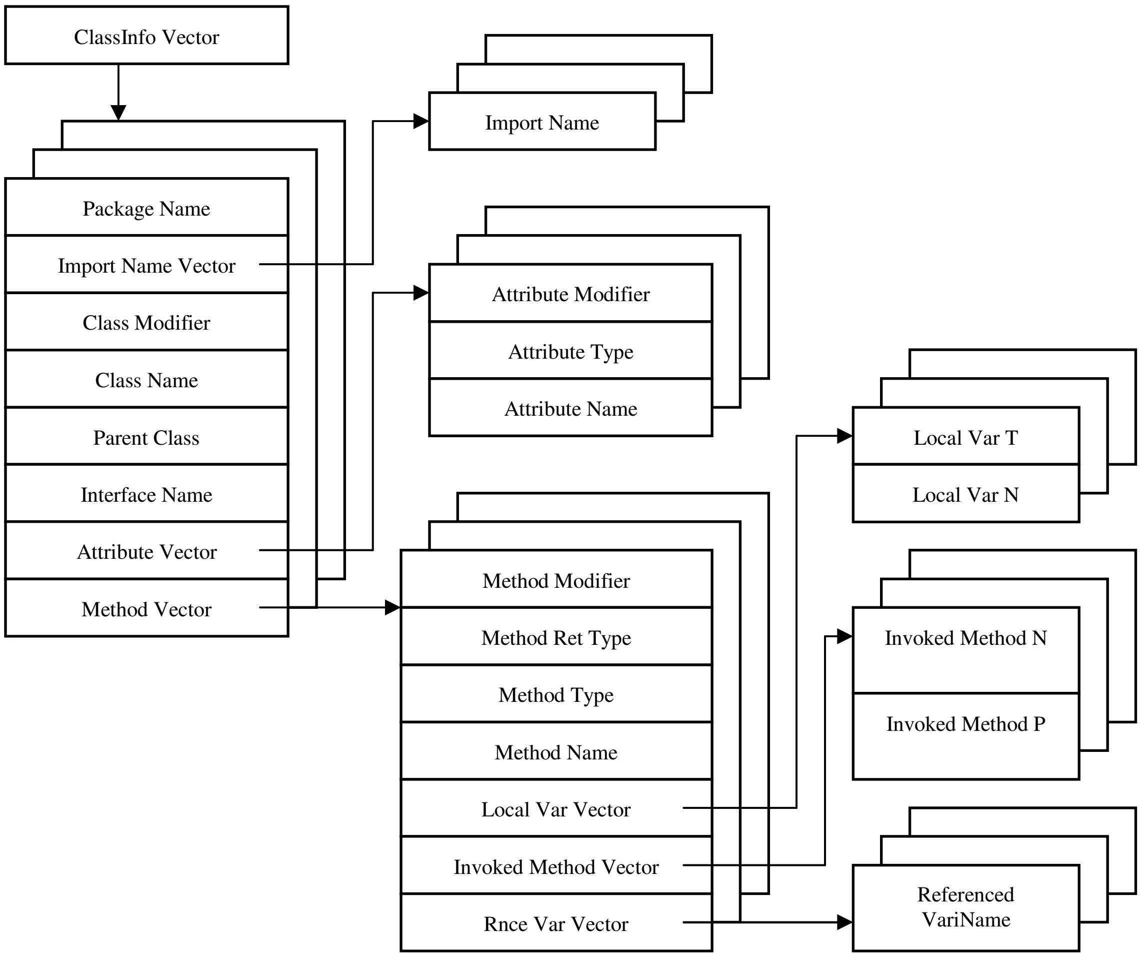


Figure 4-2: Internal Measurement Tree

4.1.2 The Internal Measurement Tree

The Internal Measurement Tree (IMT) is a low level representation of classes, attributes, methods and relationships of the program source code that is being analyzed. The IMT, after it has been completely resolved, contains all relevant information from the source code. It is a representation of the source for measurement. A complete IMT hierarchy is shown in Figure 4-2. The root of an IMT is *classInfoVector* and the *classInfoVector* has a link to *ClassInfo* node. Each *ClassInfo* node contains information about a class including Attribute Vector, Method Vector etc. The Attribute Vector and the Method Vector also have their own links which have detail information about them and so on.

Algorithm 1. *Type of attribute* Coupling.
Traverse IMT and find *Type of attribute* couplings among the classes in a project .

Input: Internal Measurement Tree;
Output: Coupling measurement result for *Type of attribute* metrics;

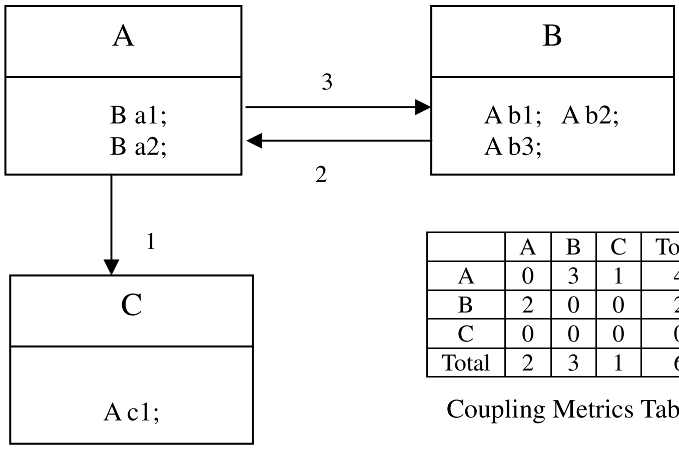
```
Let classNames = all class names in a project;
foreach class in classNames do
  Let targetClass = a class in classNames that has not been measured;
  if targetClass  is empty then
    return couplingResult;
  Traverse class node in IMT and
  let attributeTypes = all attribute types in the targetClass;
    foreach attribute type in attributeTypes do
      Compare to class names in classNames;
      Update couplingResult according to the comparison result;
  endfor
enfdor
```

Figure 4-3: Algorithm 1- Type of attribute coupling

4.1.3 Measurement Data Generator

The *Measurement Data Generator* subsystem takes an IMT as an input, collects the measurement data from the IMT, and builds measurement results such as size, complexity, coupling and cohesion metrics for a class.

Algorithm 1 in Figure 4-3 describes the coupling measurement algorithm for the *type of attribute* metric. The algorithm processes each class node in the IMT and computes coupling strength for the *type of attribute* metric to be displayed in the measurement tables like fan-in, fan-out, and class-to-class tables. For instance, we have three classes A, B, and C to show the *type of attribute* coupling metrics in Figure 4-4. Reading columns, we see that Class A is used by class B three times and used by class C once, which means that the fan-out of A for B and C are 3 and 1, respectively. Class B is used by class A twice, which means that the fan-in of A for B is 2. In this way, the coupling relationship between classes is measured as a coupling metric and the measured metric values are presented in the coupling metrics table form as shown in Figure 4-4.



	A	B	C	Total
A	0	3	1	4
B	2	0	0	2
C	0	0	0	0
Total	2	3	1	6

Coupling Metrics Table

Figure 4-4: Example of *Type of attribute couplings*

From the coupling metric table, we can easily find that the *type of attribute* couplings of classes A, B, and C are 2, 3 and 1, respectively. The *type of attribute* coupling was explained in sections 3.3 and 3.4. It is also clear to see that these three classes are connected together with attribute coupling. Therefore we group the three classes as a set of related classes and identify them a Connected Unit. The detailed discussion of Connected Unit will be done in Section 4.3.

Cohesion measurement data is also generated in this subsystem. Algorithm 2 in Figure 4-5 describes a measurement algorithm for Method Invocation Cohesion (see section 3.2). The algorithm takes each method node from the IMT and computes cohesion strength for the method invocation metric to be displayed in the measurement tables.

Algorithm 2 . *Method Invocation* Cohesion.
Traverse IMT and find *Method Invocation* cohesion from the target class in a project .

Input: Internal Measurement Tree;
Output: Cohesion measurement result for *Method Invocation* cohesion metrics;

```
Let targetClass = the target class names in a project;
Let methodNames = all method names of targetClass;
foreach methods in targetClass do
  Let targetMethod = a method in targetClass that has not been measured;
  if targetMethod  is empty then
    return cohesionResult;
  Traverse method node of targetClass in IMT and
  let invokedMethods = all invoked methods from the targetMethod;
    foreach invoked method in invokedMethods do
      Compare to method names in targetClass;
      Update cohesionResult according to the comparison result;
  endfor
enfdor
```

Figure 4-5: Algorithm 2 - Method invocation cohesion

```
public class A
{

    public void ma(){
          int r = mb() + mc();
    }

    public int mb(){
        return mc() + 1;
    }

    public int mc(){
        int c = 0;
        return c;
    }
}
```

	ma	mb	mc	Fan-Out Total
ma	0	1	1	2
mb	0	0	1	1
mc	0	0	0	0
Fan-In Total	0	1	2	

Cohesion metrics table

Figure 4-6: Cohesion of three methods in a class

Figure 4-6 shows an example of three methods to measure cohesion. We have three methods, ma, mb, and mc, in class A. Method ma invokes two methods mb and mc, and method mb invokes mc. With these invocations, the relationship of methods is measured as cohesion metrics, and the measured metric values are presented in the cohesion metrics table.

The *Measurement Data Generator* also measures all other coupling metrics and cohesion metrics mentioned in Chapter 3. The measured information about coupling for each class is then neatly presented in the coupling measurement tables constructed by the Measurement Table Generator, which will be discussed in detail in the following section.

4.1.4 Measurement Table Generator

The *Measurement Table Generator* generates display tables showing various metrics obtained. For instance, a *class-to-class* coupling measurement table showing the coupling structure among classes is given in window W2 of Figure 4-7. Windows W3 and

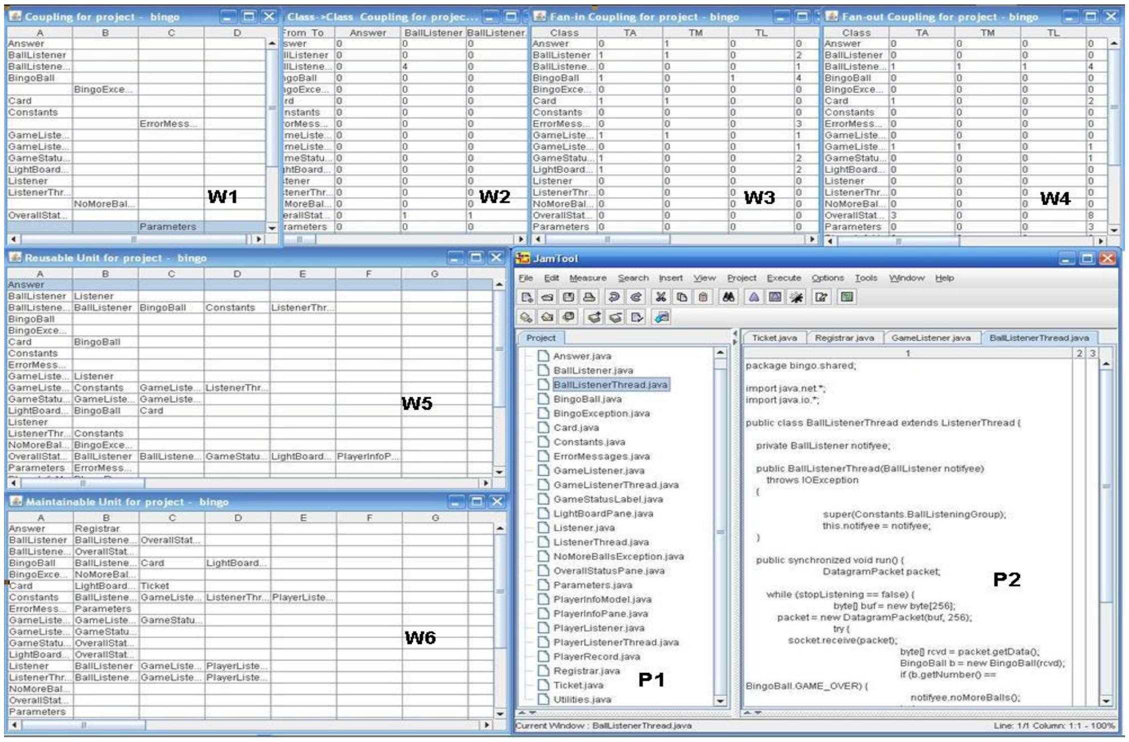
W4 of Figures 4-7 show *fan-in/fan-out* coupling measures in a tabular form for the seven coupling metrics defined in Table 3-4. Fan-in/fan-out and various coupling types can be interpreted differently as we describe fan-in/fan-out coupling measurement tables and how we can find the *connected unit* from these measurement tables in the next section.

Other important tables are *reusable unit* and *maintainable unit* tables shown in windows W5 and W6 of Figure 4-7. In a reusable unit table, each class in the first column depends on classes in other columns since the class uses the others, and in a maintainable unit table, each class in the first column is used by classes in other columns. Thus the classes in the same row make a special reusable unit and maintainable unit. In this way of representation, we could easily recognize which classes need more/less effort when they are needed for reuse, modify, update or fix. This could definitely help programmer in developing and maintaining a program. Detailed discussion for each table and unit will be provided in the following Section 4.2.

4.1.5 User Interface

JamTool provides a graphical user interface that is developed based on the Java Swing library. The measurement results are displayed in a tabular representation and in several windows with various levels of detail as shown in Figure 4-7.

Inputs to the JamTool are Java source files. Users need to provide the name of the group of the Java files (*i.e.*, project) and the location of each file when building a new project or opening an existing project in JamTool. A hierarchical list box is created within a project pane to display classes that form the project (See *P1* in Figure 4-7).



Coupling for project - bingo

A	B	C	D
Answer			
BallListener			
BallListene...			
BingoBall			
	BingoExce...		
Card			
Constants			
		ErrorMess...	
GameListe...			
GameListe...			
GameStatu...			
LightBoard...			
Listener			
ListenerThr...			
	NoMoreBal...		
OverallStat...			
		Parameters	

Class->Class Coupling for projec...

From To	Answer	BallListener	BallListener.
swer	0	0	0
llListener	0	0	0
llListene...	0	4	0
goBall	0	0	0
ıgoExce...	0	0	0
rd	0	0	0
nstants	0	0	0
orMess...	0	0	0
meListe...	0	0	0
meListe...	0	0	0
meStatu...	0	0	0
ıhtBoard...	0	0	0
tener	0	0	0
tenerThr...	0	0	0
MoreBal...	0	0	0
erallStat...	0	1	1
rameters	0	0	0

Fan-in Coupling for project - bingo

Class	TA	TM	TL	
Answer	0	1	0	0
BallListener	1	1	0	2
BallListene...	0	0	0	1
BingoBall	1	0	1	4
BingoExce...	0	0	0	0
Card	1	1	0	0
Constants	0	0	0	0
ErrorMess...	0	0	0	3
GameListe...	1	1	0	1
GameListe...	0	0	0	1
GameStatu...	1	0	0	2
LightBoard...	1	0	0	2
Listener	0	0	0	0
ListenerThr...	0	0	0	0
NoMoreBal...	0	0	0	0
OverallStat...	0	0	0	0
Parameters	0	0	0	0

Fan-out Coupling for project - bingo

Class	TA	TM	TL	
Answer	0	0	0	0
BallListener	0	0	0	0
BallListene...	1	1	1	4
BingoBall	0	0	0	0
BingoExce...	0	0	0	0
Card	1	0	0	2
Constants	0	0	0	0
ErrorMess...	0	0	0	0
GameListe...	0	0	0	0
GameListe...	1	1	0	1
GameStatu...	0	0	0	1
LightBoard...	0	0	0	0
Listener	0	0	0	0
ListenerThr...	0	0	0	0
NoMoreBal...	0	0	0	0
OverallStat...	3	0	0	8
Parameters	0	0	0	3

Reusable Unit for project - bingo

A	B	C	D	E	F	G
Answer						
BallListener	Listener					
BallListene...	BallListener	BingoBall	Constants	ListenerThr...		
BingoBall						
BingoExce...						
Card	BingoBall					
Constants						
ErrorMess...						
GameListe...	Listener					
GameListe...	Constants	GameListe...	ListenerThr...			
GameStatu...	GameListe...	GameListe...				
LightBoard...	BingoBall	Card				
Listener						
ListenerThr...	Constants					
NoMoreBal...	BingoExce...					
OverallStat...	BallListener	BallListene...	GameStatu...	LightBoard...	PlayerInfoP...	
Parameters	ErrorMess...					

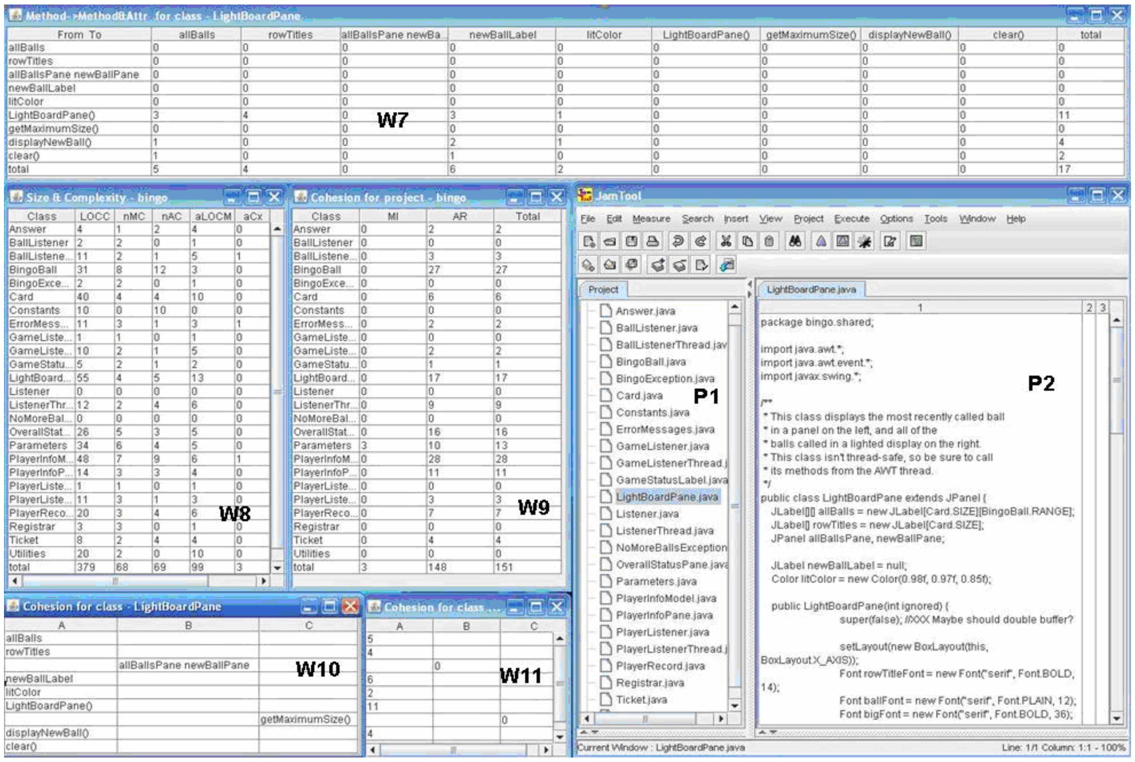
Maintainable Unit for project - bingo

A	B	C	D	E	F	G
Answer	Registrar					
BallListener	BallListene...	OverallStat...				
BallListene...	OverallStat...					
BingoBall	BallListene...	Card	LightBoard...			
BingoExce...	NoMoreBal...					
Card	LightBoard...	Ticket				
Constants	BallListene...	GameListe...	ListenerThr...	PlayerListe...		
ErrorMess...	Parameters					
GameListe...	GameListe...	GameStatu...				
GameListe...	GameStatu...					
GameStatu...	OverallStat...					
LightBoard...	OverallStat...					
Listener	BallListener	GameListe...	PlayerListe...			
ListenerThr...	BallListene...	GameListe...	PlayerListe...			
NoMoreBal...						
OverallStat...						
Parameters						

Answer.java
BallListener.java
BallListenerThread.java
BingoBall.java
BingoException.java
Card.java
Constants.java
ErrorMessages.java
GameListener.java
GameListenerThread.java
GameStatusLabel.java
LightBoardPane.java
Listener.java
ListenerThread.java
NoMoreBallsException.java
OverallStatusPane.java
Parameters.java
PlayerInfoModel.java
PlayerInfoPane.java
PlayerListener.java
PlayerListenerThread.java
PlayerRecord.java
Registrar.java
Ticket.java
Utilities.java

```
package bingo.shared;

import java.net.*;
import java.io.*;

public class BallListenerThread extends ListenerThread {

  private BallListener notifyee;

  public BallListenerThread(BallListener notifyee)
    throws IOException
  {

                super(Constants.BallListeningGroup);
                this.notifyee = notifyee;
  }

  public synchronized void run() {
                DatagramPacket packet;

    while (stopListening == false) {
                  byte[] buf = new byte[256];
      packet = new DatagramPacket(buf, 256);
                  try {
        socket.receive(packet);
                              byte[] rcvd = packet.getData();
                              BingoBall b = new BingoBall(rcvd);
                              if (b.getNumber() ==
BingoBall.GAME_OVER) {
                                notifyee.noMoreBalls();
```

Current Window : BallListenerThread.java   Line: 1/1 Column: 1:1 - 100%


W7
W8
W9
W10
W11
P1
P2
JamTool
File Edit Measure Search Insert View Project Execute Options Tools Window Help
Project
LightBoardPane.java

Method->Method&Attr for class - LightBoardPane

From To	allBalls	rowTitles	allBallsPane newBa...	newBallLabel	litColor	LightBoardPane()	getMaximumSize()	displayNewBall()	clear()	total
allBalls	0	0	0	0	0	0	0	0	0	0
rowTitles	0	0	0	0	0	0	0	0	0	0
allBallsPane newBallPane	0	0	0	0	0	0	0	0	0	0
newBallLabel	0	0	0	0	0	0	0	0	0	0
litColor	0	0	0	0	0	0	0	0	0	0
LightBoardPane()	3	4	0	3	1	0	0	0	0	11
getMaximumSize()	0	0	0	0	0	0	0	0	0	0
displayNewBall()	1	0	0	2	1	0	0	0	0	4
clear()	1	0	0	1	0	0	0	0	0	2
total	5	4	0	6	2	0	0	0	0	17

Size & Complexity - bingo

Class	LOCC	nMC	nAC	aLOCM	aCx
Answer	4	1	2	4	0
BallListener	2	2	0	1	0
BallListene...	11	2	1	5	1
BingoBall	31	8	12	3	0
BingoExce...	2	2	0	1	0
Card	40	4	4	10	0
Constants	10	0	10	0	0
ErrorMess...	11	3	1	3	1
GameListe...	1	1	0	1	0
GameListe...	10	2	1	5	0
GameStatu...	5	2	1	2	0
LightBoard...	55	4	5	13	0
Listener	0	0	0	0	0
ListenerThr...	12	2	4	6	0
NoMoreBal...	0	0	0	0	0
OverallStat...	26	5	3	5	0
Parameters	34	6	4	5	0
PlayerInfoM...	48	7	9	6	1
PlayerInfoP...	14	3	3	4	0
PlayerListe...	1	1	0	1	0
PlayerListe...	11	3	1	3	0
PlayerReco...	20	3	4	6	0
Registrar	3	3	0	1	0
Ticket	8	2	4	4	0
Utilities	20	2	0	10	0
total	379	68	69	99	3

Cohesion for project - bingo

Class	MI	AR	Total
Answer	0	2	2
BallListener	0	0	0
BallListene...	0	3	3
BingoBall	0	27	27
BingoExce...	0	0	0
Card	0	6	6
Constants	0	0	0
ErrorMess...	0	2	2
GameListe...	0	0	0
GameListe...	0	2	2
GameStatu...	0	1	1
LightBoard...	0	17	17
Listener	0	0	0
ListenerThr...	0	9	9
NoMoreBal...	0	0	0
OverallStat...	0	16	16
Parameters	3	10	13
PlayerInfoM...	0	28	28
PlayerInfoP...	0	11	11
PlayerListe...	0	0	0
PlayerListe...	0	3	3
PlayerReco...	0	7	7
Registrar	0	0	0
Ticket	0	4	4
Utilities	0	0	0
total	3	148	151

Cohesion for class - LightBoardPane

A	B	C
allBalls		
rowTitles		
	allBallsPane newBallPane	
newBallLabel		
litColor		
LightBoardPane()		
		getMaximumSize()
displayNewBall()		
clear()		

Cohesion for class ...

A	B	C
5		
4		
	0	
6		
2		
11		
		0
4		

Answer.java
BallListener.java
BallListenerThread.jav
BingoBall.java
BingoException.java
Card.java
Constants.java
ErrorMessages.java
GameListener.java
GameListenerThread.j
GameStatusLabel.java
LightBoardPane.java
Listener.java
ListenerThread.java
NoMoreBallsException
OverallStatusPane.java
Parameters.java
PlayerInfoModel.java
PlayerInfoPane.java
PlayerListener.java
PlayerListenerThread.j
PlayerRecord.java
Registrar.java
Ticket.java

```
package bingo.shared;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * This class displays the most recently called ball
 * in a panel on the left, and all of the
 * balls called in a lighted display on the right.
 * This class isn't thread-safe, so be sure to call
 * its methods from the AWT thread.
 */
public class LightBoardPane extends JPanel {
  JLabel[][] allBalls = new JLabel[Card.SIZE][BingoBall.RANGE];
  JLabel[] rowTitles = new JLabel[Card.SIZE];
  JPanel allBallsPane, newBallPane;

  JLabel newBallLabel = null;
  Color litColor = new Color(0.98f, 0.97f, 0.85f);

  public LightBoardPane(int ignored) {
                super(false); //XXX Maybe should double buffer?

                setLayout(new BoxLayout(this,
BoxLayout.X_AXIS));
                Font rowTitleFont = new Font("serif", Font.BOLD,
14);
                Font ballFont = new Font("serif", Font.PLAIN, 12);
                Font bigFont = new Font("serif", Font.BOLD, 36);
```

Current Window : LightBoardPane.java   Line: 1/1 Column: 1:1 - 100%

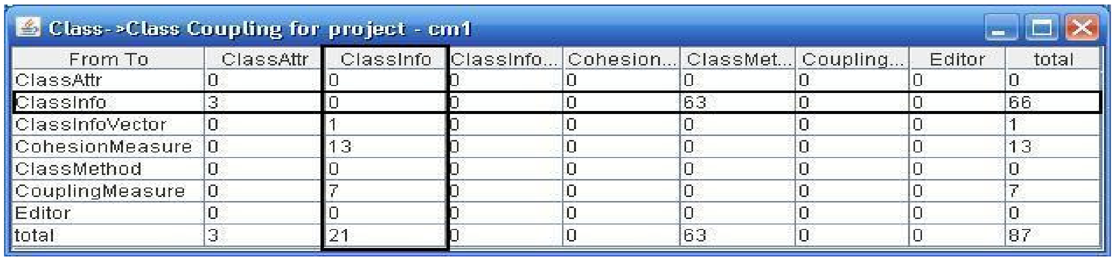
Figure 4-7: Screen shot of JamTool for coupling, cohesion, size, and complexity

Pane *P1* shows that the project is composed of multiple Java programs. Pane *P2* shows the source code of the selected Java program. For the project named *'Bingo',* six windows (W1-W6) display the coupling measurement results: connected unit (W1), class-to-class coupling (W2), fan-in coupling (W3), fan-out coupling (W4), reusable unit (W5), and maintainable unit (W6), and another five windows (W7-W11) display the cohesion, size and complexity measurement results: cohesion in a class (W7), size & complexity (W8), cohesion for each class (W9), and connected unit (W10) and its strength (W11) for cohesion.

4.2 Measurement Result Tables

4.2.1 Class to Class Table

Class-to-class coupling measurement table in Figure 4-8 is to show coupling relationship among classes. All class names in a project are displayed. Regarding a class, *ClassInfo,* in the second row, we see that there is a coupling strength of '3' to *ClassAttr*, '63' to *ClassMethod*, and a '66' for total. These mean *ClassInfo* uses *ClassAttr* 3 times and *ClassMethod* 63 times, thus 66 times for the total. On the other hand, regarding *ClassInfo* in the second column, we find that this class is used by *ClassInfoVector*(1), *CohesionMeasure*(13), and *CouplingMeasure*(7), for a total of 21 times.


Class->Class Coupling for project - cm1

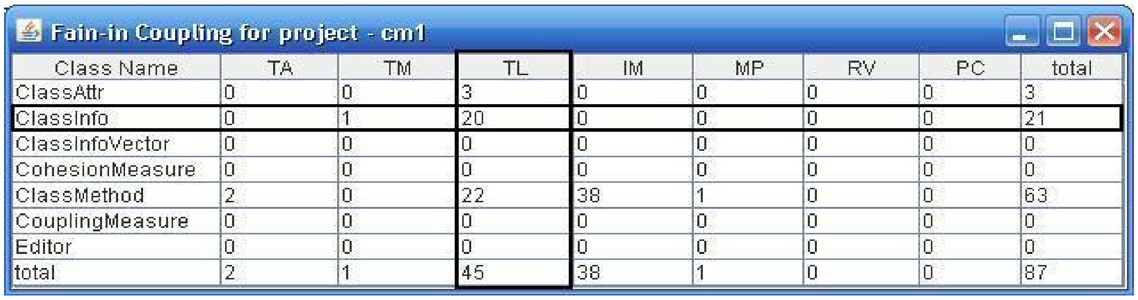
From To	ClassAttr	ClassInfo	ClassInfo...	Cohesion...	ClassMet...	Coupling...	Editor	total
ClassAttr	0	0	0	0	0	0	0	0
ClassInfo	3	0	0	0	63	0	0	66
ClassInfoVector	0	1	0	0	0	0	0	1
CohesionMeasure	0	13	0	0	0	0	0	13
ClassMethod	0	0	0	0	0	0	0	0
CouplingMeasure	0	7	0	0	0	0	0	7
Editor	0	0	0	0	0	0	0	0
total	3	21	0	0	63	0	0	87

Figure 4-8: Class to class coupling measurement table

4.2.2 Fan-in Coupling Table

The 7 coupling metrics defined in Table 3-6 are displayed in a similar tabular form of the Fan-in coupling in Figure 4-9. TA, TM, TL, IM, MP, RV, and PC stand for *Type of Attribute, Type of Method Invocation, Type of Local Variable, Invoked Method Type, Referenced Variable, and Parent-Child,* respectively. All classes in a project are displayed in the first column. We interpret *fan-in* as *used-by, invoked by, or referenced by*, thus we can find how other classes *use* this class through examining each fan-in coupling instance. For instance, *ClassInfo* has fan-in coupling strength of '1' for TM and '20' for TL, which means that other classes in the project (*i.e.*, *cm1*) use *ClassInfo* once as invoked method and twenty times as their local variables. In this figure it is clear that *ClassMethod* is used extensively by other classes (sixty three fan-in coupling at total column). Special attention must be given to such a class when it is examined or modified because it influences many other coupled classes.

If we inspect column TL, it has '3' to ClassAttr, '20' to ClassInfo, '22' to ClassMethod and '45'in total. This means forty five times of fan-in coupling as *Type of Local Variable* have occurred in this project while there are thirty eight times for IM, twice for TA, respectively and only once for TM and MP.

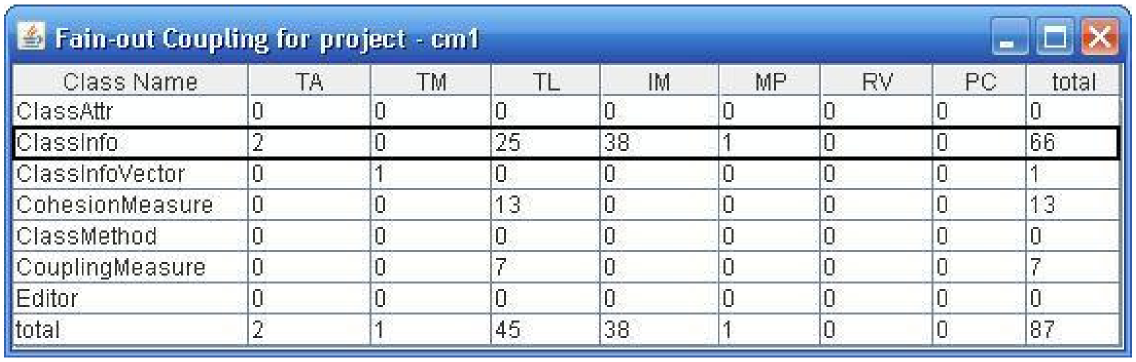

Fain-in Coupling for project - cm1

Class Name	TA	TM	TL	IM	MP	RV	PC	total
ClassAttr	0	0	3	0	0	0	0	3
ClassInfo	0	1	20	0	0	0	0	21
ClassInfoVector	0	0	0	0	0	0	0	0
CohesionMeasure	0	0	0	0	0	0	0	0
ClassMethod	2	0	22	38	1	0	0	63
CouplingMeasure	0	0	0	0	0	0	0	0
Editor	0	0	0	0	0	0	0	0
total	2	1	45	38	1	0	0	87

Figure 4-9: Fan-in coupling measurement table

4.2.3 Fan-out Coupling Table

Fan-out couplings for the same seven coupling metrics are in Figure 4-10. We interpret *fan-out* as *use, invoke or reference*, thus we can find how a class *uses* other classes through examining each fan-out coupling instance. For instance, f*an-out* coupling of *ClassInfo* shows that this class invokes or uses other classes sixty six times in total ('2' for TA, '25' for TL, '38' for IM, and '1' for MP). It mainly uses local variables and invokes methods, and is identified as a highly fan-out coupled class. We believe that such a class is difficult to be reused alone because it needs other classes' services to perform its function. Therefore, it is wise to inspect its fan-out coupled classes from this table for a new application when we reuse a class.



Class Name	TA	TM	TL	IM	MP	RV	PC	total
ClassAttr	0	0	0	0	0	0	0	0
ClassInfo	2	0	25	38	1	0	0	66
ClassInfoVector	0	1	0	0	0	0	0	1
CohesionMeasure	0	0	13	0	0	0	0	13
ClassMethod	0	0	0	0	0	0	0	0
CouplingMeasure	0	0	7	0	0	0	0	7
Editor	0	0	0	0	0	0	0	0
total	2	1	45	38	1	0	0	87

Figure 4-10: Fan-out coupling measurement table

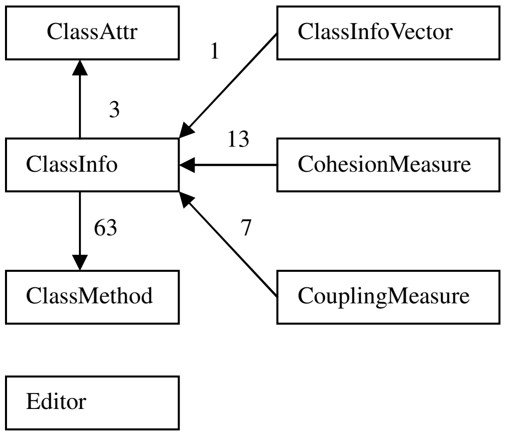
4.2.4 Connected Unit Table for Coupling

We define a connected unit as the classes that are coupled together. In a connected unit table, all classes coupled together are displayed in the same column. A connected unit is likely to be of interest to the user in finding software units that can be reused. We build a connected unit by identifying coupled classes in the coupling metrics and the connected attributes and methods in the cohesion metrics. A user should consider reusing the connected classes together in a new application. In that sense, the connected classes are a reusable unit. The connected unit search algorithm, shown in Figure 4-11, computes a set of coupled classes (i.e., connected unit) and their position in a connected unit table based on a class-to-class coupling table. Figure 4-12 shows the retrieved connected unit and the result of applying Algorithm 3 to the class-to-class coupling table in Figure 4-8. Each class is displayed in a connected unit table according to its position and its coupling strength is displayed in the connected unit strength table in Figure 4-12 (b).

Algorithm 3. *Connected Unit Search.*
Compute *connected units* from a class-to-class table.

```
Input: Class-to-class coupling measurement table;
Output: Connected units and their positions in a connected unit
table;

Let classNames = all class names from a class-to-class table;
foreach class in classNames do
   Let targetClass = a class in classNames
   that has not been searched yet;
   if targetClass is empty then
      return connectUnitsWithPosition;
   Search class-to-class table and let
   connectUnit = coupled classes to targetClass;
   Update connectedUnitsWithPosition with the connectUnit;
end for
```

Figure 4-11: *Connected Unit Search* Algorithm

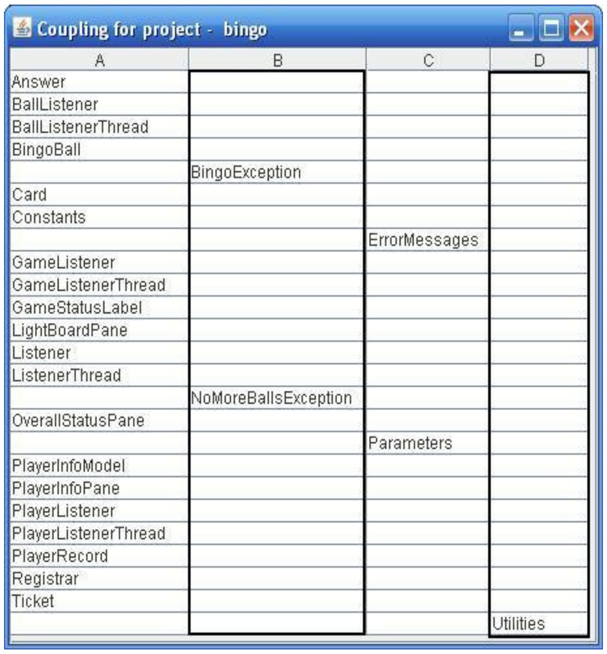


(a)

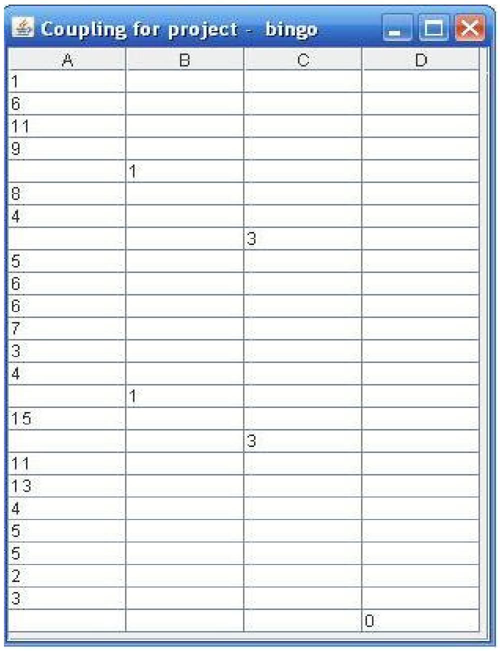
class	position	strength
ClassAttr	1	3
ClassInfo	1	87
ClassInfoVector	1	1
CohesionMeasure	1	13
ClassMethod	1	63
CouplingMeasure	1	7
Editor	2	0

(b)

Figure 4-12: Example of *Connected Unit Search* algorithm



A	B	C	D
Answer			
BallListener			
BallListenerThread			
BingoBall			
	BingoException		
Card			
Constants			
		ErrorMessages	
GameListener			
GameListenerThread			
GameStatusLabel			
LightBoardPane			
Listener			
ListenerThread			
	NoMoreBallsException		
OverallStatusPane			
		Parameters	
PlayerInfoModel			
PlayerInfoPane			
PlayerListener			
PlayerListenerThread			
PlayerRecord			
Registrar			
Ticket			
			Utilities



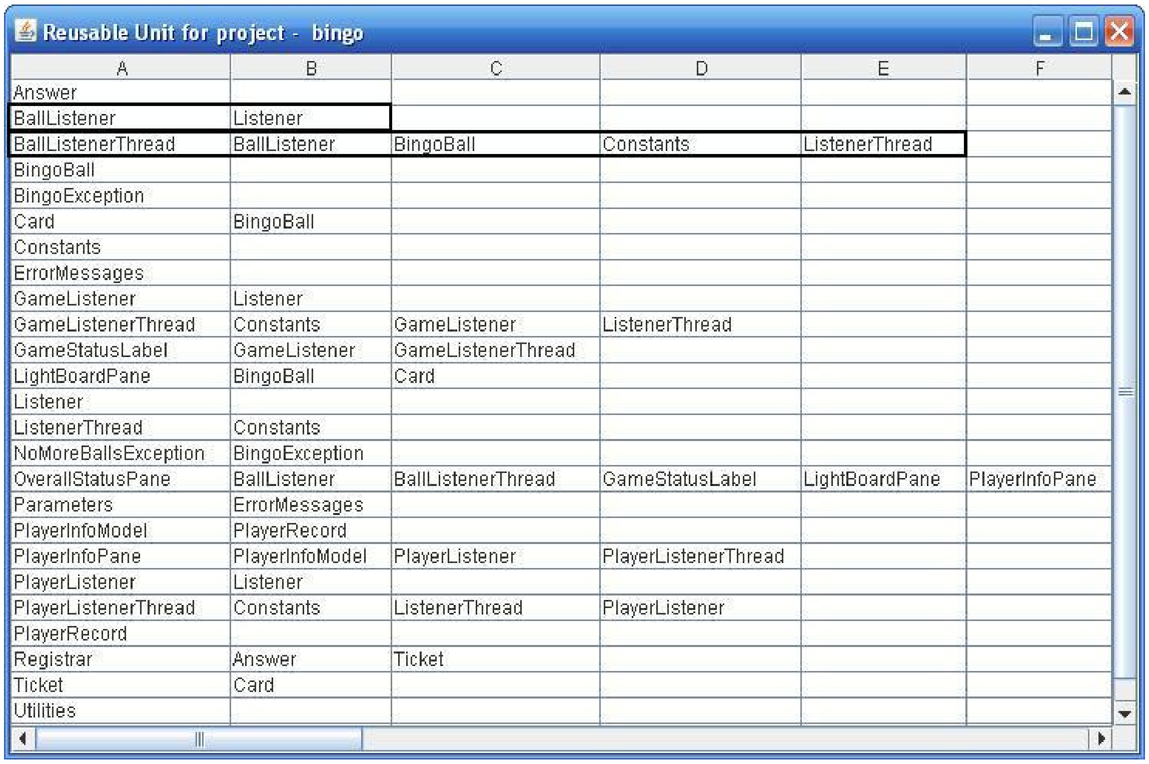
A	B	C	D
1			
6			
11			
9			
	1		
8			
4			
		3	
5			
6			
6			
7			
3			
4			
	1		
15			
		3	
11			
13			
4			
5			
5			
2			
3			
			0

Figure 4-13: Connected unit and its strength

The connected unit and its strength of the ‘bingo’ project are shown in Figure 4-13. In this tables, all classes in the same column are coupled together. For instance, only two classes, *BingoException* and *NoMoreBallaException*, in column B are coupled to each other. *Utilities* in column D could be a dead code because there is no relation to other classes in the project. By observing connected units, we may also discover connection patterns. For example, if a project is composed of an application program and libraries, an investigation of the connected unit will tell how the application program uses a library function. In that sense, this type of connection pattern is a use pattern.

4.2.5 Reusable Unit Table

Other important tables are reusable unit and maintainable unit tables. Reusable unit table is to present how much a class depends on other classes. In Figure 4-14, the first column, A, displays all classes in the selected project. A class in column A uses the classes in columns to its right. The classes in the same row make a special reusable unit. For instant, in the second and third rows, we see that class *BallListener* depends on class *Linstener*, and class *BallListenerThread* depends on classes *BallListene, BingoBall, Constants, and ListenerThread.* This dependency means that, for example, if programmer wants to use a certain class (*BallListenerThread)*, then he/she must use the other classes in the reusable unit (*BallListener, BingoBall, Constants, and ListenerThread)* since they are used by the certain class (*BallListenerThread)*. Therefore, if a class depends on too many other classes, it is obvious that such a class is difficult to be reused.

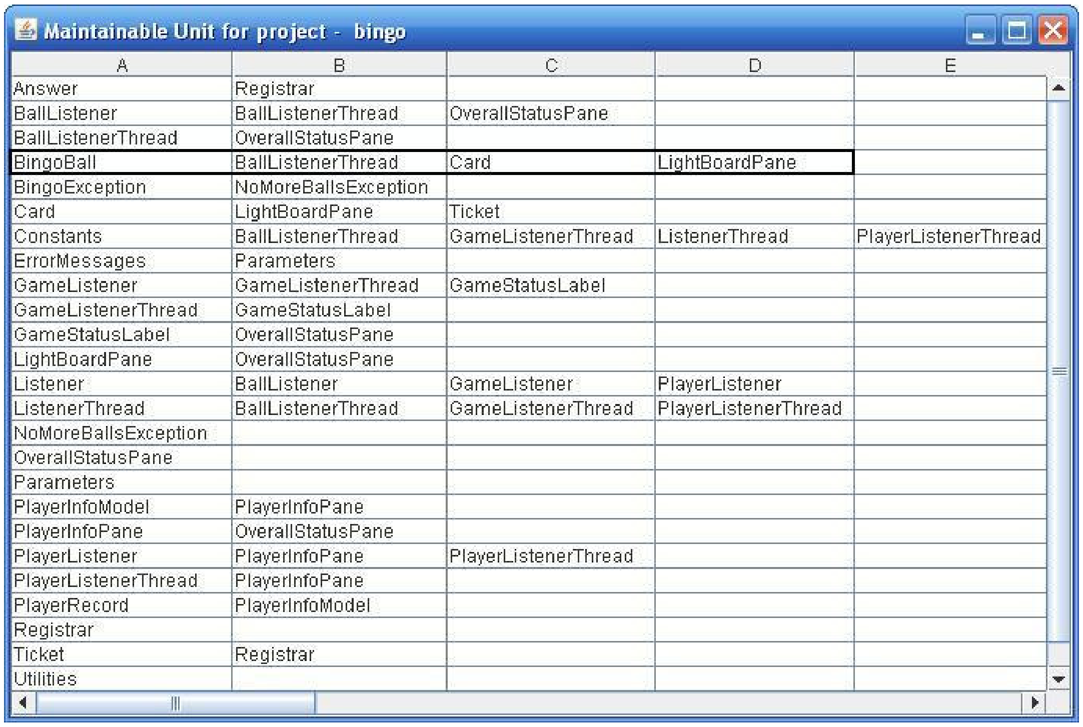

Reusable Unit for project - bingo

A	B	C	D	E	F
Answer					
BallListener	Listener				
BallListenerThread	BallListener	BingoBall	Constants	ListenerThread	
BingoBall					
BingoException					
Card	BingoBall				
Constants					
ErrorMessages					
GameListener	Listener				
GameListenerThread	Constants	GameListener	ListenerThread		
GameStatusLabel	GameListener	GameListenerThread			
LightBoardPane	BingoBall	Card			
Listener					
ListenerThread	Constants				
NoMoreBallsException	BingoException				
OverallStatusPane	BallListener	BallListenerThread	GameStatusLabel	LightBoardPane	PlayerInfoPane
Parameters	ErrorMessages				
PlayerInfoModel	PlayerRecord				
PlayerInfoPane	PlayerInfoModel	PlayerListener	PlayerListenerThread		
PlayerListener	Listener				
PlayerListenerThread	Constants	ListenerThread	PlayerListener		
PlayerRecord					
Registrar	Answer	Ticket			
Ticket	Card				
Utilities					

Figure 4-14: Reusable unit table

4.2.6 Maintainable Unit Table

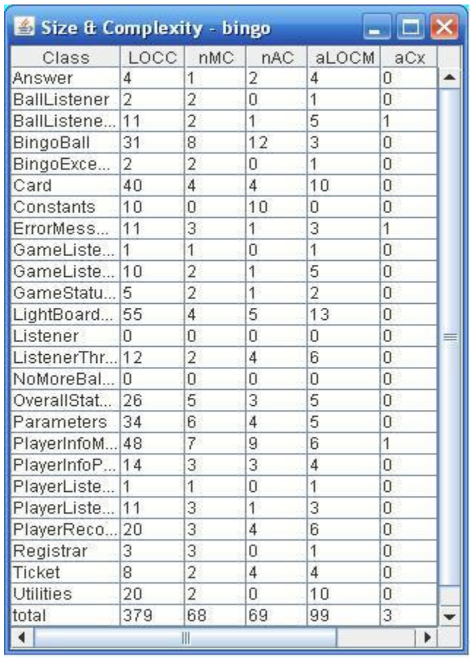
Figure 4-15 shows a maintainable unit table. Maintainable unit is to present how many classes depend on a specific class. All classes in the selected project are displayed in the first column, A, and each class in that column is used by the classes in other columns, thus the classes in the same row are identified as a maintainable unit. For instant, three classes *BallListenerThread*, *Card*, and *LightBoardPane* in the third row use *BingoBall*, thus if you want to modify or update *BingoBall*, you must test *BallListenerThread*, *Card*, and *LightBoardPane* as well. Therefore if there are too many classes in a maintainable unit, it is very hard to maintain that specific class.


Maintainable Unit for project - bingo

A	B	C	D	E
Answer	Registrar			
BallListener	BallListenerThread	OverallStatusPane		
BallListenerThread	OverallStatusPane			
BingoBall	BallListenerThread	Card	LightBoardPane	
BingoException	NoMoreBallsException			
Card	LightBoardPane	Ticket		
Constants	BallListenerThread	GameListenerThread	ListenerThread	PlayerListenerThread
ErrorMessages	Parameters			
GameListener	GameListenerThread	GameStatusLabel		
GameListenerThread	GameStatusLabel			
GameStatusLabel	OverallStatusPane			
LightBoardPane	OverallStatusPane			
Listener	BallListener	GameListener	PlayerListener	
ListenerThread	BallListenerThread	GameListenerThread	PlayerListenerThread	
NoMoreBallsException				
OverallStatusPane				
Parameters				
PlayerInfoModel	PlayerInfoPane			
PlayerInfoPane	OverallStatusPane			
PlayerListener	PlayerInfoPane	PlayerListenerThread		
PlayerListenerThread	PlayerInfoPane			
PlayerRecord	PlayerInfoModel			
Registrar				
Ticket	Registrar			
Utilities				

Figure 4-15: Maintainable unit table

4.2.7 Size and Complexity Table

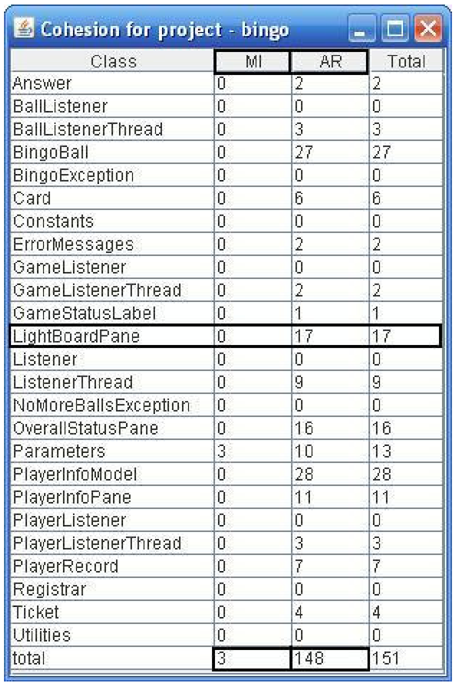

Size & Complexity - bingo

Class	LOCC	nMC	nAC	aLOCM	aCx
Answer	4	1	2	4	0
BallListener	2	2	0	1	0
BallListene...	11	2	1	5	1
BingoBall	31	8	12	3	0
BingoExce...	2	2	0	1	0
Card	40	4	4	10	0
Constants	10	0	10	0	0
ErrorMess...	11	3	1	3	1
GameListe...	1	1	0	1	0
GameListe...	10	2	1	5	0
GameStatu...	5	2	1	2	0
LightBoard...	55	4	5	13	0
Listener	0	0	0	0	0
ListenerThr...	12	2	4	6	0
NoMoreBal...	0	0	0	0	0
OverallStat...	26	5	3	5	0
Parameters	34	6	4	5	0
PlayerInfoM...	48	7	9	6	1
PlayerInfoP...	14	3	3	4	0
PlayerListe...	1	1	0	1	0
PlayerListe...	11	3	1	3	0
PlayerReco...	20	3	4	6	0
Registrar	3	3	0	1	0
Ticket	8	2	4	4	0
Utilities	20	2	0	10	0
total	379	68	69	99	3

Figure 4-16: Size and complexity table

Five size and complexity metrics for each class in a project are given in Figure 4-16. They are based on the definitions given in Chapter 3 (*LOCC: Lines of Code in a Class*, *nMC: number of Metohds in a Class*, *nAC: number of Attributes in a Class* , *aLOCM: average LOC for Methods*, and *aCx*: *average McCabe complexity*) .

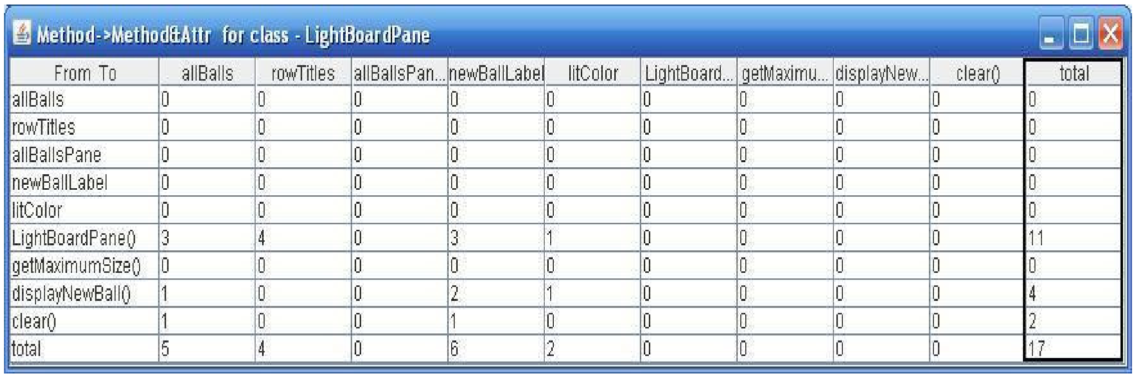
4.2.8 Cohesion Table



Class	MI	AR	Total
Answer	0	2	2
BallListener	0	0	0
BallListenerThread	0	3	3
BingoBall	0	27	27
BingoException	0	0	0
Card	0	6	6
Constants	0	0	0
ErrorMessages	0	2	2
GameListener	0	0	0
GameListenerThread	0	2	2
GameStatusLabel	0	1	1
LightBoardPane	0	17	17
Listener	0	0	0
ListenerThread	0	9	9
NoMoreBallsException	0	0	0
OverallStatusPane	0	16	16
Parameters	3	10	13
PlayerInfoModel	0	28	28
PlayerInfoPane	0	11	11
PlayerListener	0	0	0
PlayerListenerThread	0	3	3
PlayerRecord	0	7	7
Registrar	0	0	0
Ticket	0	4	4
Utilities	0	0	0
total	3	148	151

Figure 4-17: Cohesion table

Cohesion metrics for each class in a project are given in Figure 4-17. *MI* is *Method Invocation* cohesion and *AR* is *Attribute Reference* cohesion; both are discussed in Chapter 3. From this table, we can easily see that in this particular program, most of the classes use/reference attributes (148 times) within a class rather than invoke methods (3 times).



From To	allBalls	rowTitles	allBallsPan...	newBallLabel	litColor	LightBoard...	getMaximu...	displayNew...	clear()	total
allBalls	0	0	0	0	0	0	0	0	0	0
rowTitles	0	0	0	0	0	0	0	0	0	0
allBallsPane	0	0	0	0	0	0	0	0	0	0
newBallLabel	0	0	0	0	0	0	0	0	0	0
litColor	0	0	0	0	0	0	0	0	0	0
LightBoardPane()	3	4	0	3	1	0	0	0	0	11
getMaximumSize()	0	0	0	0	0	0	0	0	0	0
displayNewBall()	1	0	0	2	1	0	0	0	0	4
clear()	1	0	0	1	0	0	0	0	0	2
total	5	4	0	6	2	0	0	0	0	17

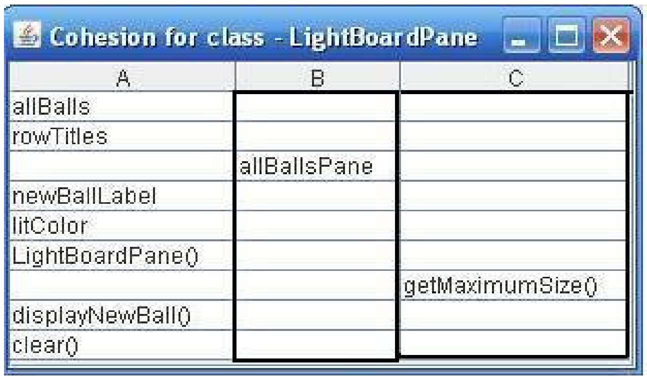
Figure 4-18: Cohesion among methods and attributes

We look inside a class to examine how the methods and attributes in the class are related to each other. In Figure 4-18, the first column and the header row represent all attributes and methods, respectively, in the target class (*LightBoardPane*). If we see *LightBoardPane* class in Figure 4-17, this class has ‘17’ for AR cohesion measure, and in

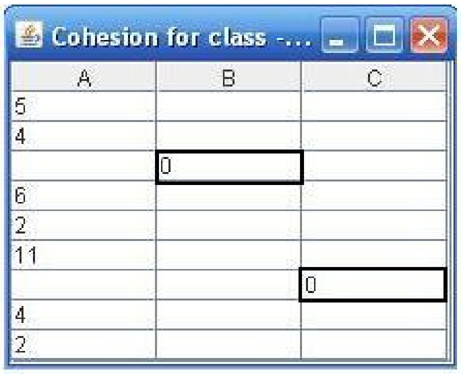
Figure 4-18, we can see each occurrence of AR relation for the class between attributes and methods, making 17 relations in total. For example, methods *lightBoardPane(), displayNewBall()*, and *clear()* reference attributes 11 times (allBalls(3), rowTitles(4), newBallLable(3), and litColor(1)), 4 times, and 2 times, respectively, for a total of 17 times (17 AR cohesion).

4.2.9 Connected Unit Table for Cohesion

We also define a connected unit for cohesion metrics and compute the cohesion strength of a class as shown in Figure 4-19 (a) and (b). All attributes and methods in the same column make a unique connected unit because they are related to each other. In this case, it is clearly indicated that attribute *allBallsPane* and method *getMaximunSize()* have no relation to other elements in the class, thus their cohesion strengths are both zero.



A	B	C
allBalls		
rowTitles		
	allBallsPane	
newBallLabel		
litColor		
LightBoardPane()		
		getMaximumSize()
displayNewBall()		
clear()		



A	B	C
5		
4		
	0	
6		
2		
11		
		0
4		
2		

(a) Connected unit name table (b) Connected unit strength table

Figure 4-19: Connected unit and its strength for cohesion

4.3 Connected Unit

Display techniques and tabular representations have been studied as to how to best depict various metric findings. To represent the coupling and cohesion measurements, we develop a measurement result table and a connected unit table. They can display not only the connection strength (count) among the software components, but also the architectural nature of an object-oriented system.

A connected unit table is composed of a pair of corresponding tables: connected unit name table and connected unit strength table. In the connected unit name table, only coupled classes can be located in a same column, thus a set of classes in the same column is a *connected unit*.

In the connected unit strength table, each number represents the coupling strength of the corresponding classes in the connected unit name table. Figure 4-20 (a) shows that classes *A*, *B*, *D*, *F*, and *I* are in the same column because they are coupled to each other.

Corresponding numbers in Figure 4-20 (b) represent the coupling strength of each of these classes. For example, number seven in the Figure 4-20 (b) indicates that class *A* has a total count of seven for fan-out and fan-in to other classes in this column (*i.e.*, *B*, *D*,

Class A			
Class B			
	Class C		
Class D			
		Class E	
Class F			
	Class G		
			Class H
Class I			

(a) Connected unit name table

7			
3			
	4		
2			
		0	
43			
	4		
			0
11			

(b) Connected unit strength table

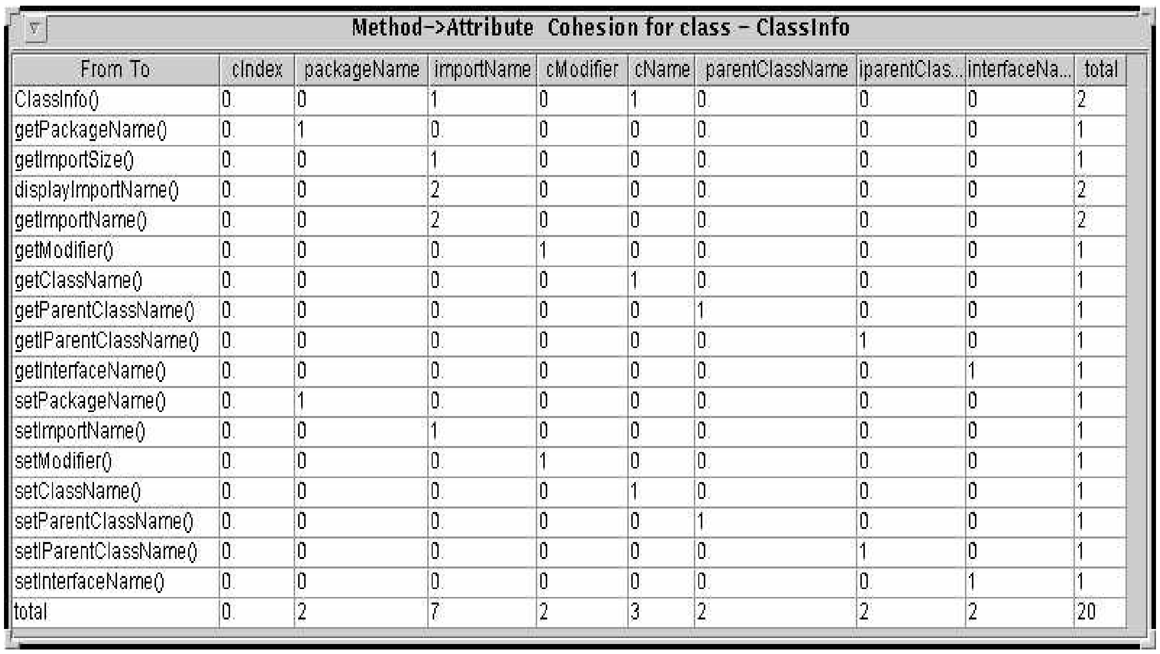
Figure 4-20: Connected unit table

F, and *I*). Classes *E* and *H* are not related to others in this project thus their coupling strengths are both zero. There are several possibilities for those classes that have strength zero:

- They are no longer used in the project therefore they should be deleted from the project.
- They have independent functions that are ready to be used in other applications.

Therefore, we need to inspect their corresponding source codes to determine their usefulness.

Classes *C* and *G* are related to each other but not to others as they appear in the same column. We may classify these two classes as a *reusable or maintainable unit* after inspecting the measurement results and the source code. We also need to inspect class F



Method->Attribute Cohesion for class - ClassInfo

From To	cIndex	packageName	importName	cModifier	cName	parentClassName	iparentClas...	interfaceNa...	total
ClassInfo()	0	0	1	0	1	0	0	0	2
getPackageName()	0	1	0	0	0	0	0	0	1
getImportSize()	0	0	1	0	0	0	0	0	1
displayImportName()	0	0	2	0	0	0	0	0	2
getImportName()	0	0	2	0	0	0	0	0	2
getModifier()	0	0	0	1	0	0	0	0	1
getClassName()	0	0	0	0	1	0	0	0	1
getParentClassName()	0	0	0	0	0	1	0	0	1
getIParentClassName()	0	0	0	0	0	0	1	0	1
getInterfaceName()	0	0	0	0	0	0	0	1	1
setPackageName()	0	1	0	0	0	0	0	0	1
setImportName()	0	0	1	0	0	0	0	0	1
setModifier()	0	0	0	1	0	0	0	0	1
setClassName()	0	0	0	0	1	0	0	0	1
setParentClassName()	0	0	0	0	0	1	0	0	1
setIParentClassName()	0	0	0	0	0	0	1	0	1
setInterfaceName()	0	0	0	0	0	0	0	1	1
total	0	2	7	2	3	2	2	2	20

Figure 4-21: Attribute reference cohesion measurement table

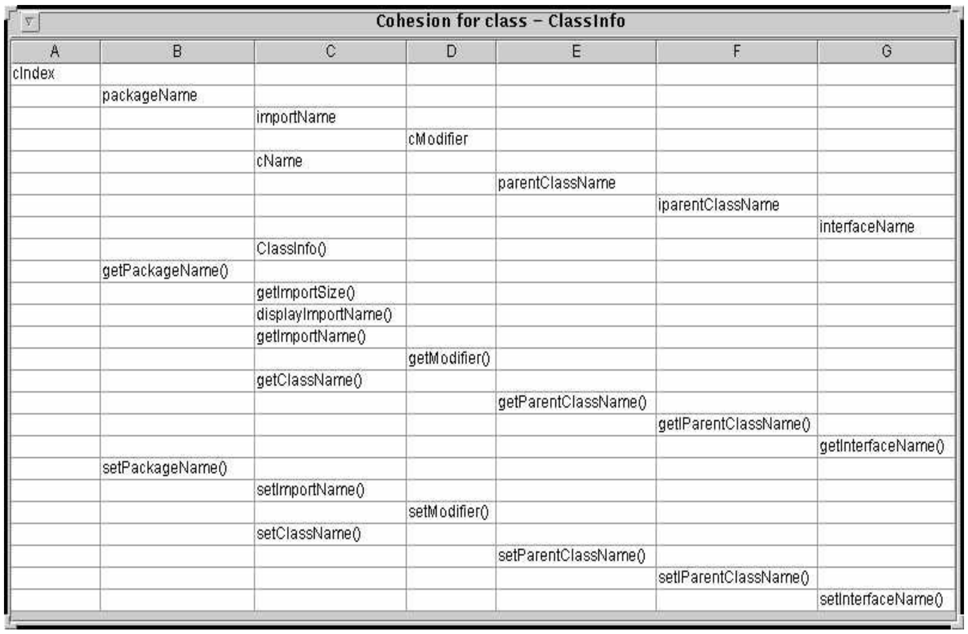
to find out why it has such a high coupling count. Using the connected unit table, a user can inspect a target class and its coupled classes. Connected unit table may be used in a priori or a posteriori manner. A developer may decide in priori to slice the class or remove the dead code in an application after browsing the connected unit table or in posteriori to inspect the software for reuse purpose.

We can apply the same approach to the cohesion metrics. As an example of *connected unit* in a cohesion connected unit table, we may find a class designed for multiple functions. If indeed the class has different functions, the user may slice the class into several small classes and reuse a portion of them. This approach can reduce test and maintenance costs.

Figure 4-21 shows *attribute reference cohesion measurement* for *ClassInfo*. All methods are listed in the first column and all attributes are listed in the first row. A number in this table indicates how many times the method in the row references the corresponding attribute in the column. For example, method *getPackageName()* references attribute *packageName* once.

A cohesion connected unit table can be built based on this cohesion measurement table. Figures 4-22 (a) and (b) show the connected unit tables of Cohesion Measurement for *ClassInfo*. Like the coupling connected unit table, only related attributes or methods can be located in the same column.

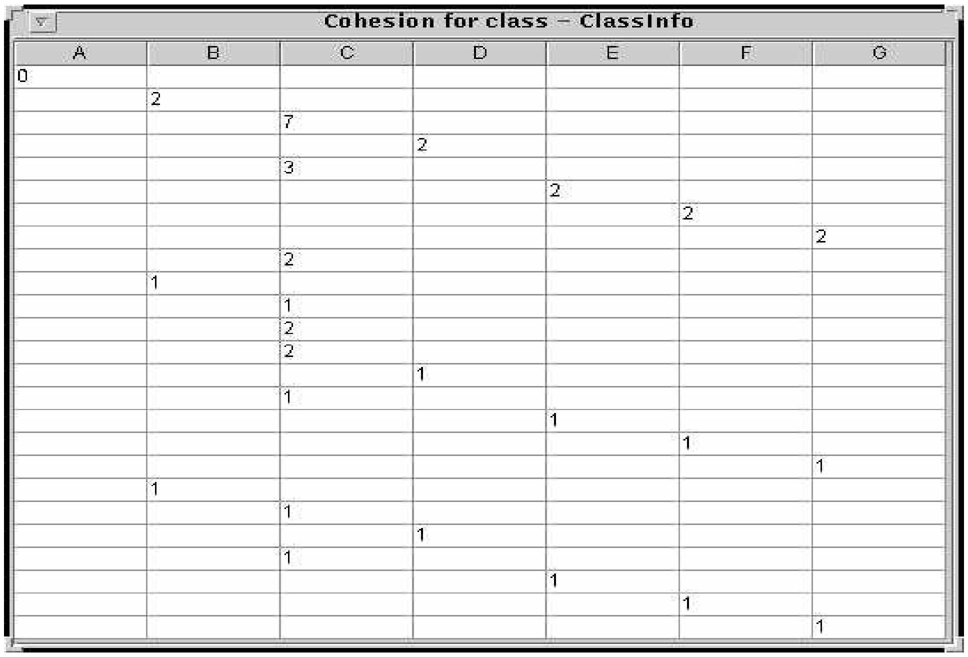
In this example, we can find the *use pattern* in columns *B*, *D*, *E*, *F*, and *G* (getters and setters for attributes).



Cohesion for class – ClassInfo

A	B	C	D	E	F	G
cIndex						
	packageName					
		importName				
			cModifier			
		cName				
				parentClassName		
					iparentClassName	
						interfaceName
		ClassInfo()				
	getPackageName()					
		getImportSize()				
		displayImportName()				
		getImportName()				
			getModifier()			
		getClassName()				
				getParentClassName()		
					getIParentClassName()	
						getInterfaceName()
	setPackageName()					
		setImportName()				
			setModifier()			
		setClassName()				
				setParentClassName()		
					setIParentClassName()	
						setInterfaceName()

(a) Cohesion connected unit name table



Cohesion for class – ClassInfo

A	B	C	D	E	F	G
0						
	2					
		7				
			2			
		3				
				2		
					2	
						2
		2				
	1					
		1				
		2				
		2				
			1			
		1				
				1		
					1	
						1
	1					
		1				
			1			
		1				
				1		
					1	
						1

(b) Cohesion connected unit strength table

Figure 4-22: Cohesion connected unit table for class *ClassInfo*

For example, there are two methods in column B, *getPackageName()* and *setPackageName(),* for an attribute *PackageName,* and two methods in column E, *getParentClassName()* and *setParentClassName(),* for an attribute *ParentClassName.*

Since the attributes/methods in the same column are related to each other, if a user wants to reuse method *getModifier()* in column *D*, he/she would need to reuse attribute *cModifier* and method *setModifier()*. In that case, these three software components in column *D* can be identified as a *reusable unit*. Since attribute *cIndex* in column *A* has no relation to other classes and other parts in this class, this attribute can be classified as a *dead code* thus should be deleted. A user would first browses the connected unit table to identify the reusable units and then inspect their connection patterns to see how such software components are connected and/or used in the software package. By examining the measurement tables, a user can also decide whether he/she can reuse the whole or part of the *reusable unit*. Locating related components and inspecting their *use pattern* can guide a user to reuse them.

4.4 Measurement Result Export for Spreadsheet

When the *Measurement Table Generator* in JamTool creates tables, it generates the measurement results in the CSV (Comma Separated Values) file format. The CSV file format is a file type that stores tabula data which uses a comma to separate values and is supported by almost all spreadsheets. Therefore, JamTool exports measurement results directly into a spreadsheet application such as Microsoft Excel.

Exporting to spreadsheet expands the power of JamTool by enabling further analysis and graphing. Spreadsheet application provides some of the statistical analysis or presentation capabilities required to investigate the measurement results. Therefore it does provide a great advantage to help the JamTool users to derive meanings from the measurement data.

With Export to spreadsheet we can:

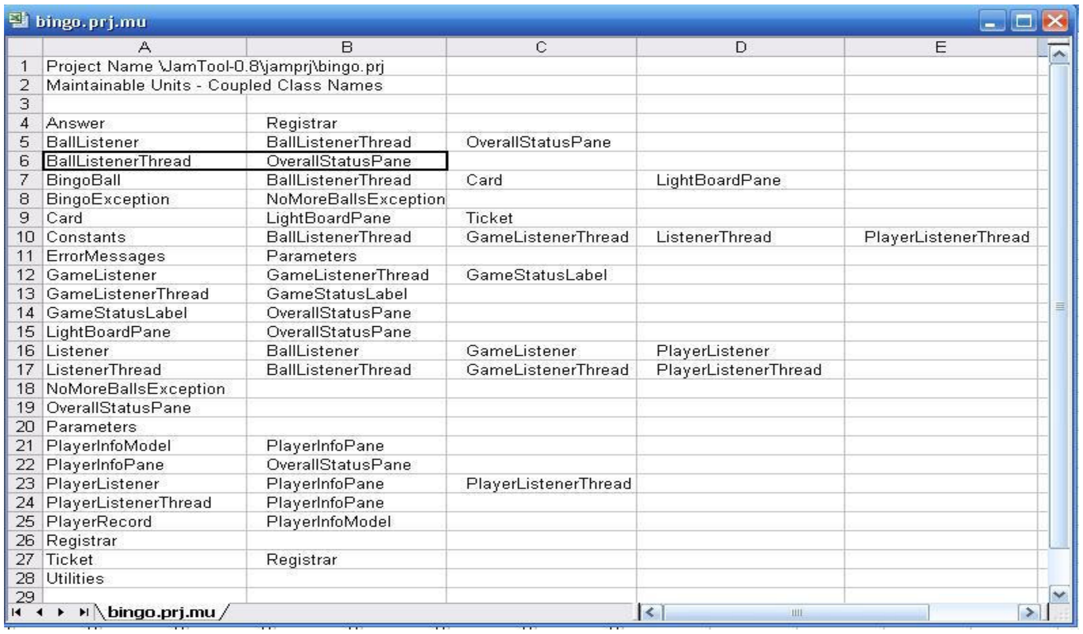
- Display measurement results in spreadsheet instead of JamTool
- Analyze measurement data with a spreadsheet application
- Configure and format reports to represent the measurement data in an easy to understand style such as graph

Spreadsheet application offers the ability to perform calculations and complex mathematical, statistical, and data analysis functions on numbers and text. JamTool's tabular data is suitable to take these advantages.

Figures 4-23 and 4-24 show an example of Measurement Result Export for Spreadsheet. Figures 4-23 (a) and (b) are maintainable/reusable units for coupled classes in a project, and they are the same as Figures 4-14 and 4-15, but exported to Excel for analysis report.

Figure 4-24 displays fan-in/out couplings and their visual graphs. Column A in Figure 4-24 (a) displays all classes in the selected project. For the corresponding class, Columns B and C in Figure 4-24 (a) show the number of classes fan-in coupled and the number of classes fan-out coupled, respectively, and they are obtained from the reusable/maintainable units in Figure 4-23. For instance, if we look at class *BallListenerThread*, this class is used/invoked by only one class (*OverallStatusPane)* as shown in the maintainable units of Figure 4-23 (a), having -1 for fan-in of column B in Figure 4-24 (a), and uses/invokes four classes (*BallListener*, *BingoBall*, *Constants,* and *ListenerThread)* as shown in the reusable units Figure 4-23 (b), having 4 for fan-out of column C in Figure 4-24 (a). The negative sign (-) of column B is to graph fan-in couplings under the x-axis to visually compare them to fan-out couplings above the x-axis. Their actual strengths of coupling (Number of times they are coupled in the coupled classes) are shown and graphed in Figure 4-24 (b). For example, *BallListenerThread* class has -1 for fan-in and 10 for fan-out, which means that this class is used/invoked by *OverallStatusPane* class only once, but uses/invokes four classes (*BallListener*, *BingoBall*, *Constants,* and *ListenerThread*) 10 times for total. The negative sign (-) is for the same purpose as Figure 4-24 (a).

These tabular data and comparative graphic representation will clearly assist and aid JamTool users in a better understanding of software reuse and maintenance. For instance, class *OverallStatusPane*, which has the highest fan-out coupling, will decrease the reusability of the class, and class *BingoBall*, which has the highest fan-in coupling, will decrease the maintainability of the class.

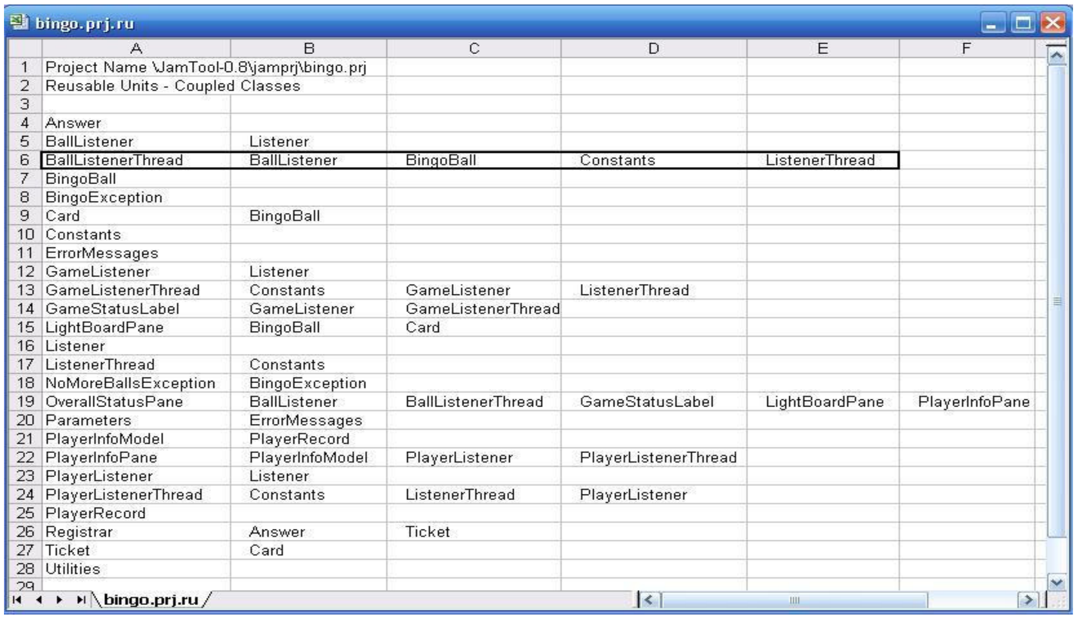


bingo.prj.mu

	A	B	C	D	E
1	Project Name \JamTool-0.8\jamprj\bingo.prj				
2	Maintainable Units - Coupled Class Names				
3					
4	Answer	Registrar			
5	BallListener	BallListenerThread	OverallStatusPane		
6	BallListenerThread	OverallStatusPane			
7	BingoBall	BallListenerThread	Card	LightBoardPane	
8	BingoException	NoMoreBallsException			
9	Card	LightBoardPane	Ticket		
10	Constants	BallListenerThread	GameListenerThread	ListenerThread	PlayerListenerThread
11	ErrorMessages	Parameters			
12	GameListener	GameListenerThread	GameStatusLabel		
13	GameListenerThread	GameStatusLabel			
14	GameStatusLabel	OverallStatusPane			
15	LightBoardPane	OverallStatusPane			
16	Listener	BallListener	GameListener	PlayerListener	
17	ListenerThread	BallListenerThread	GameListenerThread	PlayerListenerThread	
18	NoMoreBallsException				
19	OverallStatusPane				
20	Parameters				
21	PlayerInfoModel	PlayerInfoPane			
22	PlayerInfoPane	OverallStatusPane			
23	PlayerListener	PlayerInfoPane	PlayerListenerThread		
24	PlayerListenerThread	PlayerInfoPane			
25	PlayerRecord	PlayerInfoModel			
26	Registrar				
27	Ticket	Registrar			
28	Utilities				
29					

bingo.prj.mu

(a) Maintainable units exported to Excel



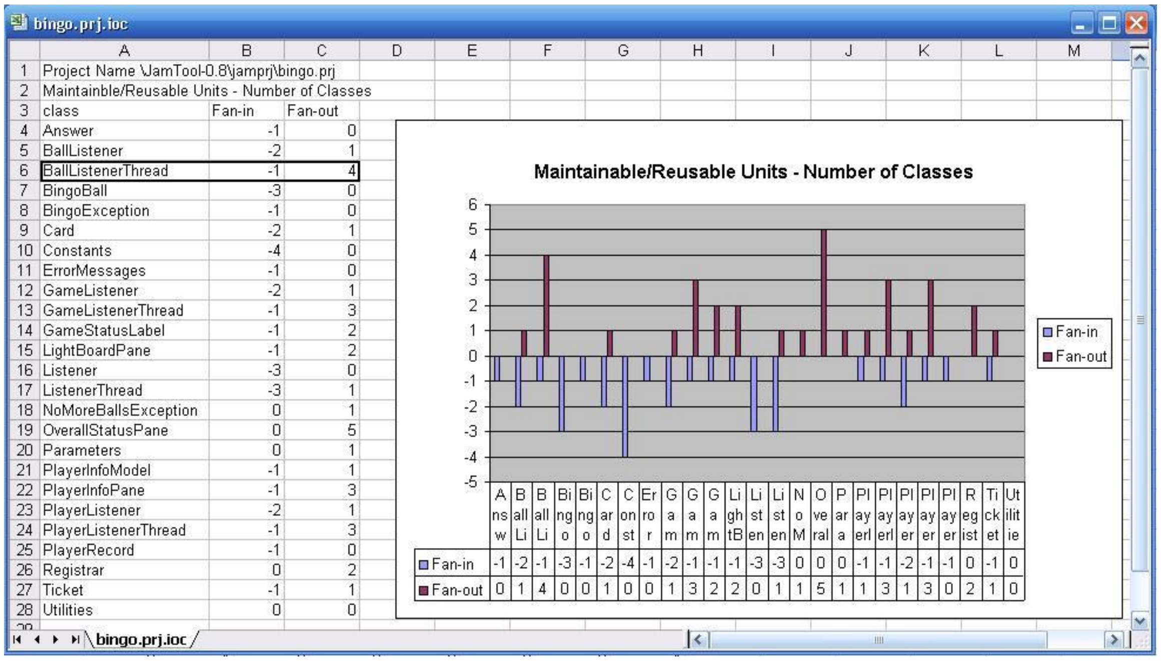
bingo.prj.ru

	A	B	C	D	E	F
1	Project Name \JamTool-0.8\jamprj\bingo.prj					
2	Reusable Units - Coupled Classes					
3						
4	Answer					
5	BallListener	Listener				
6	BallListenerThread	BallListener	BingoBall	Constants	ListenerThread	
7	BingoBall					
8	BingoException					
9	Card	BingoBall				
10	Constants					
11	ErrorMessages					
12	GameListener	Listener				
13	GameListenerThread	Constants	GameListener	ListenerThread		
14	GameStatusLabel	GameListener	GameListenerThread			
15	LightBoardPane	BingoBall	Card			
16	Listener					
17	ListenerThread	Constants				
18	NoMoreBallsException	BingoException				
19	OverallStatusPane	BallListener	BallListenerThread	GameStatusLabel	LightBoardPane	PlayerInfoPane
20	Parameters	ErrorMessages				
21	PlayerInfoModel	PlayerRecord				
22	PlayerInfoPane	PlayerInfoModel	PlayerListener	PlayerListenerThread		
23	PlayerListener	Listener				
24	PlayerListenerThread	Constants	ListenerThread	PlayerListener		
25	PlayerRecord					
26	Registrar	Answer	Ticket			
27	Ticket	Card				
28	Utilities					
29						

bingo.prj.ru

(b) Reusable units exported to Excel

Figure 4-23: Maintainable/Reusable units exported to Excel

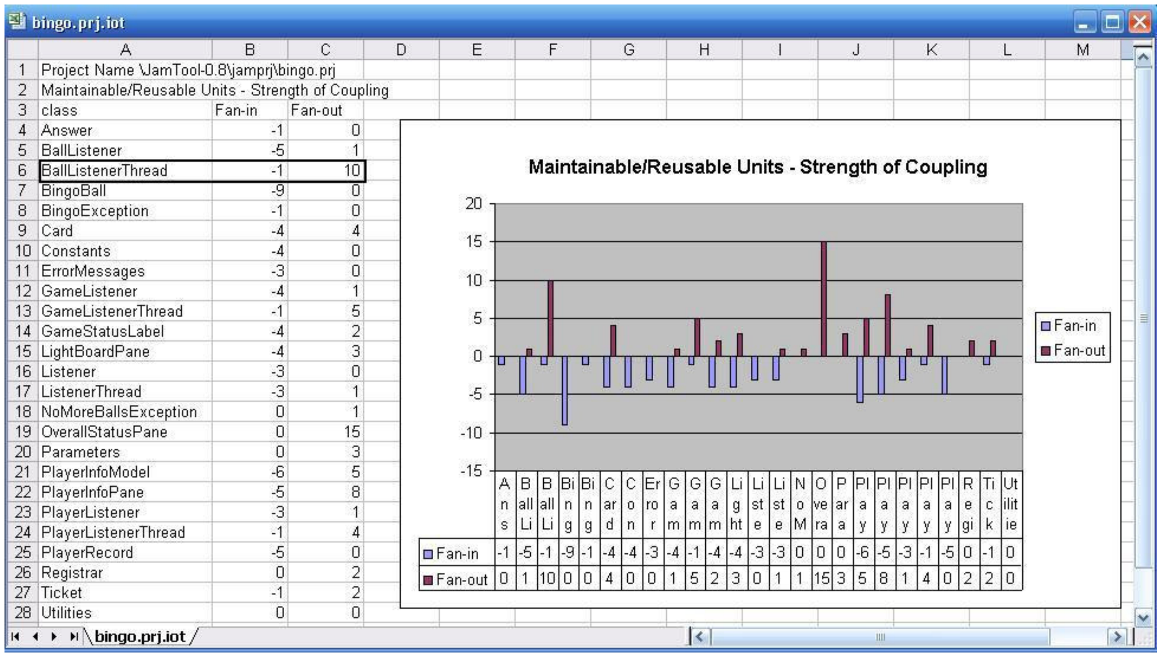


Project Name \JamTool-0.8\jamprj\bingo.prj
Maintainble/Reusable Units - Number of Classes

class	Fan-in	Fan-out
Answer	-1	0
BallListener	-2	1
BallListenerThread	-1	4
BingoBall	-3	0
BingoException	-1	0
Card	-2	1
Constants	-4	0
ErrorMessages	-1	0
GameListener	-2	1
GameListenerThread	-1	3
GameStatusLabel	-1	2
LightBoardPane	-1	2
Listener	-3	0
ListenerThread	-3	1
NoMoreBallsException	0	1
OverallStatusPane	0	5
Parameters	0	1
PlayerInfoModel	-1	1
PlayerInfoPane	-1	3
PlayerListener	-2	1
PlayerListenerThread	-1	3
PlayerRecord	-1	0
Registrar	0	2
Ticket	-1	1
Utilities	0	0

	Answ	BallLi	BallLi	Bingo	Bingo	Card	Const	Error	Gam	Gam	Gam	LightB	Listen	Listen	NoM	Overal	Para	PlayerI	PlayerI	Player	Player	Player	Regist	Ticket	Utilitie
Fan-in	-1	-2	-1	-3	-1	-2	-4	-1	-2	-1	-1	-1	-3	-3	0	0	0	-1	-1	-2	-1	-1	0	-1	0
Fan-out	0	1	4	0	0	1	0	0	1	3	2	2	0	1	1	5	1	1	3	1	3	0	2	1	0

(a) Maintainable/Reusable units – number of classes



Project Name \JamTool-0.8\jamprj\bingo.prj
Maintainable/Reusable Units - Strength of Coupling

class	Fan-in	Fan-out
Answer	-1	0
BallListener	-5	1
BallListenerThread	-1	10
BingoBall	-9	0
BingoException	-1	0
Card	-4	4
Constants	-4	0
ErrorMessages	-3	0
GameListener	-4	1
GameListenerThread	-1	5
GameStatusLabel	-4	2
LightBoardPane	-4	3
Listener	-3	0
ListenerThread	-3	1
NoMoreBallsException	0	1
OverallStatusPane	0	15
Parameters	0	3
PlayerInfoModel	-6	5
PlayerInfoPane	-5	8
PlayerListener	-3	1
PlayerListenerThread	-1	4
PlayerRecord	-5	0
Registrar	0	2
Ticket	-1	2
Utilities	0	0

	Ans	BallLi	BallLi	Bing	Bing	Card	Con	Error	Gam	Gam	Gam	Light	Liste	Liste	NoM	Overa	Para	Play	Play	Play	Play	Play	Regi	Tick	Utilie
Fan-in	-1	-5	-1	-9	-1	-4	-4	-3	-4	-1	-4	-4	-3	-3	0	0	0	-6	-5	-3	-1	-5	0	-1	0
Fan-out	0	1	10	0	0	4	0	0	1	5	2	3	0	1	1	15	3	5	8	1	4	0	2	2	0

(b) Maintainable/Reusable units – strength of coupling

Figure 4-24: Maintainable/Reusable units graphed in Excel

5 UNDERSTANDING SOFTWARE EVOLUTION USING METRICS AND VISUALIZATION

This chapter presents an empirical study to investigate if the metrics defined and implemented by JamTool can be used to assess the quality of software evolution. The empirical study is an analysis of reusability and maintainability during the evolution of an open source software system, *JFreeChart*, which is a charting library [25]. We observe the quality change along the evolution of the twenty-two released versions of *JFreeChart* and discuss its quality change based on the Lehman's laws of evolution. We derive software metrics from the twenty-two releases of the target system and determine whether software quality has significantly changed over this period. More specifically, we compare the *fan-in* and *fan-out* couplings of the *removed* and the *added* classes from one version of the software to the next in order to find out if the quality of each release has improved or declined.

A separate, but related case study to analyze how a software system has evolved was conducted. The case study is to present the global visualization of the evolution of a software system and provide effective ways to analyze the evolution of the system. Since the study does not utilize the developed metrics, the results of study are included in Appendix A.

5.1 Empirical Study: Measuring Quality on Software Evolution

Fan-in is the number of references made from outside a class to entities defined within the class, and *fan-out* is the number of references made from within a class to entities defined outside the class. While *fan-in* coupling is very useful when assessing the impact of a change, *fan-out* is very useful when partitioning programming elements and figuring out what other classes a given class needs in order to run. Therefore a low *fan-out* is desirable since a high *fan-out* is a characteristic of the large number of classes needed by the particular class and makes the class difficult to reuse [3, 6, 23, 37, and 38]. A high *fan-in* normally represents a good object design and a high level of reuse.

Although a system is useless without any coupling, for any given software solution there is a baseline or necessary coupling level and that developer's goal should be the elimination of extraneous coupling. Such unnecessary coupling needlessly decreases the reusability of the classes [43].

For library software like *JFreeChart*, high *fan-out* coupling decreases its reusability. Because it is an open source library and it has been used by other applications for a long time, we expect to find the quality of *JFreeChart* to improve along with its evolution in terms of reusability.

On the other hand, as summarized in [33], the laws of software evolution have been proposed and formalized in [30, 31, and 32] since 1974. The statement of Lehman's laws refers to *E-type* software, which cannot be completely specified and once the system is operational, the development with new requirements of the software is essential.

Evolution is intrinsic and inevitable for this type of software. Eight Lehman's laws are given in Table 5-1.

In this empirical study, we explore the evolution of the *JFreeChart* in terms of size, coupling and cohesion, which are measurable from software source code, and discuss its quality change based on the Lehman's laws of evolution. The study indicates that our experimental results follow three laws (I: Continuing change, II: Increasing complexity, VI: Continuing growth) out of eight. But this indicates more research is still needed for one law (VII: Declining quality). Each of the laws is explained here:

Continuing change: An *E-type* system must be continually adapted otherwise it becomes progressively less satisfactory in use

Increasing complexity: As an *E-type* system is evolved its complexity increases unless work is done to maintain or reduce the complexity

Continuing growth: The functional capability of *E-type* systems must be continually enhanced to maintain user satisfaction over the system lifetime

Declining quality: Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an *E-type* system will decline

Table 5-1: Latest formulation of Lehman's laws of software evolution

No./year of first formulation	Name	No./year of first formulation	Name
I 1974	Continuing change	V 1991	Conservation of familiarity
II 1974	Increasing complexity	VI 1991	Continuing growth
III 1974	Self regulation	VII 1996	Declining quality
IV 1978	Conservation of organizational stability	VIII 1971/96	Feedback system

5.1.1 Objective

The objectives of this empirical study are two fold. First, we investigate if there is any relationship between the class growth of the target software and the metric values (coupling and cohesion) measured by *JamTool*. We normally assume that if the number of classes increases, then the coupling between classes will increase as well since the coupling measures the degree to which a program module (i.e., class) relies on other modules. However, the class growth should not affect the cohesion metric values, since the cohesion metric measures the strength of relationship among internal components within a single class.

Secondly, we observe the quality change along the software evolution by comparing the *fan-in/out* couplings and the cohesion metrics of the *removed* and *added* classes of each version of the software. We expect quality software to have low coupling and high cohesion. When a software system requires updates, i.e., changes to the software to correct bugs or to install new functionalities, some classes in the software are removed and the classes with new functionalities are *added* to the software. At this point we assume the *removed* classes have poorer quality and the *added* classes should have better quality. Therefore, in terms of coupling/cohesion metrics, the newly *added* classes should have lower coupling and higher cohesion than the *removed* classes.

We investigate *fan-in* and *fan-out* couplings separately since a high *fan-in* coupling and a low *fan-out* coupling are desirable for a class. A high *fan-in* coupling indicates the class that is called upon by many other classes. Thus, it is reused. A low *fan-out* coupling means independence and encapsulation, and this kind of class/module is easier to reuse.

5.1.2 Methodology

The software used in the experiment was *JFreeChart* which is a powerful and flexible open source charting library. We choose *JFreeChart* as the target software system because it is a long-term open source library with many releases. To obtain information about the version differences, we used an evolution track table to compare two versions of a program and report all the differences. A very detailed explanation of an evolution track table is provided in section 5.2.

First, we extract information of classes in terms of size, coupling and cohesion metrics from all twenty-two versions of *JFreeChart* and analyze the relationship between the classes and the metrics. According to [13], the size of a system is defined as the number of program units it contains, thus it should be based on the number of "modules' rather than source code size. This is the main reason we use the number of classes as size metrics.

Second, we focus on the *removed* and *added* classes of the target software. We extract metrics of the *removed* classes in each version and the newly *added* classes, divide them into two groups, and compare them to investigate any differences between the groups. Then we perform an analysis by examining the coupling and cohesion metrics of the *removed* and the *added* classes over releases.

Third, we present empirical studies of the relationships between the number of classes and the derived coupling/cohesion metrics, and the relationships between the *removed* and the *added* classes throughout a software evolution.

Metric extraction can be a difficult task due to the size of the system and the number of versions. We use an evolution track table to extract the number of classes and the *removed* and *added* classes, and *JamTool* collects coupling and cohesion metrics from *JFreeChart.*

5.1.3 Hypotheses

Based on the assumption and expectation above, we set up five hypotheses: Two to observe if any relationship exists between class growth and metric values measured by *JamTool*, and three for the *added* classes (i.e., *group A*) and the *removed* classes (i.e., *group R*).

• **Hypothesis 1:** Class growth throughout all versions will be positively reflected in the *fan-in/fan-out* coupling metric values.

• **Hypothesis 2:** Class growth throughout all versions of the program will not be positively reflected in the cohesion metric values.

These two are actually to confirm the findings of the previous studies and our expectation about class growth and metrics [32].

• **Hypothesis 3:** The average *fan-in* coupling of *group A* will be higher than the average *fan-in* of *group R*.

• **Hypothesis 4:** The average *fan-out* coupling of *group A* will be lower than the average *fan-out* of *group R*.

• **Hypothesis 5:** The average cohesion of *group A* will be higher than the average cohesion of *group R*.

We believe that *group A* and *group R* can be categorized in a certain way based on the metric values of coupling and cohesion measured by *JamTool*. In other words, the *added* class group should have better software quality than the *removed* class group does.

5.1.4 Results

We applied an evolution track table and *JarJarDiff* (File comparison tool) to find the differences between two subsequent versions starting with *JFreeChart-0.9.0* and ending with *JFreeChart-0.9.21*. According to the results obtained by evolution track table and *JarJarDiff*, we found that whenever a version is newly evolved, the software had a many changes. It modified interfaces and/or classes, removed interfaces and/or classes, and/or added new packages, interfaces, and/or classes.

Normally, the number of classes gradually increases as a new version is released. Also, there are some huge changes in the middle of releases. With these reasons, in this experimental study, we investigate if the class growth shows any observable phenomenon on the coupling and cohesion metric values, and if the newly *added* classes show any observable trend in comparison with the *removed* classes.

Class Growth, Coupling, and Cohesion

Table 5-2 gives an overview of the version differences of the software and coupling/cohesion metric values obtained by *JamTool.* It shows the number of classes (*Removed*, *Added*, total), average fan-in/-out coupling metrics, and average cohesion metrics in each version of the *JFreeChart.*

Figure 5-1 shows the class growth across all versions of the program and Figure 5-2 reveals an increasing trend for the average *fan-in/fan-out* coupling. We can easily recognize that the number of class increases gradually as new versions of the program evolve and the significant class growth occurred between versions 0.9.3 and 0.9.5, otherwise the number of classes increases consistently.

There was a more than 300% class growth in the number of classes from the beginning of the program (i.e., 139) to the final version of the program (i.e., 460). This is a confirmation of the study by [6] and Lehman's 6th law of software evolution [32] that the evolution of an object-oriented system reveals an increasing trend of the number of classes.

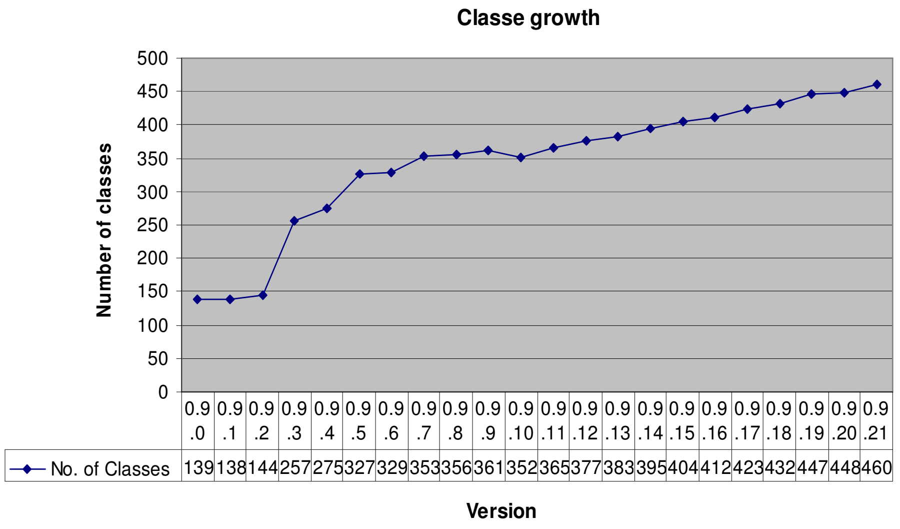
For the *Fan-in/fan-out* coupling, a noticeable change appeared between versions 0.9.3 and 0.9.4. We could say that this is because 113 classes were newly *added* to version 0.9.4 and it affects the average metric values. After that the growth trend is consistent while the average cohesion seems not to grow as the class does.

Table 5-2: Version differences and Coupling/Cohesion metrics

Version of JFreeChart	No. of Removed classes	No. of Added classes	Total no. of classes	Avg. fan-in coupling	Avg. fan-out coupling	Average cohesion
0.9.0			139	11.9	12.1	12.7
0.9.1	1	0	138	12.0	12.2	12.9
0.9.2	0	6	144	11.8	12	12.9
0.9.3	0	113	257	11.0	11	11.6
0.9.4	3	21	275	13.1	14.1	12.8
0.9.5	22	74	327	12.8	13.8	12.6
0.9.6	0	2	329	12.8	13.8	12.6
0.9.7	1	25	353	12.7	13.5	12.4
0.9.8	0	3	356	12.8	13.6	12.5
0.9.9	43	48	361	13.0	14.2	12.7
0.9.10	11	2	352	13.2	14.1	13.2
0.9.11	0	13	365	13.4	14.4	13.4
0.9.12	5	17	377	13.6	14.4	13.5
0.9.13	0	6	383	14.0	14.8	13.8
0.9.14	3	15	395	15.3	15.4	14.1
0.9.15	0	9	404	15.2	15.3	14.0
0.9.16	2	10	412	15.1	15.2	13.8
0.9.17	19	30	423	15.0	15.2	9.8
0.9.18	1	10	432	14.7	14.7	9.9
0.9.19	9	24	447	14.2	14.3	9.8
0.9.20	0	1	448	14.3	14.3	9.8
0.9.21	3	15	460	14.5	14.6	9.8
Total	123	444				

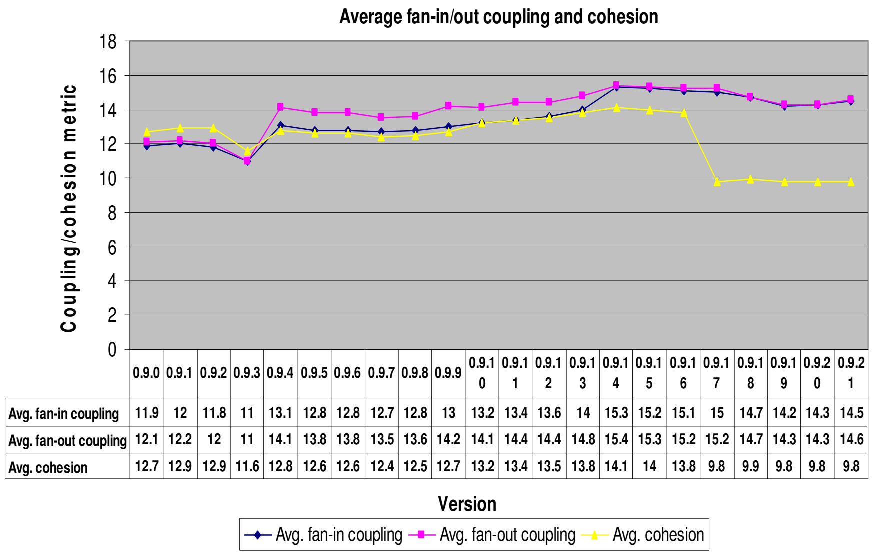
The cohesion metric between versions 0.9.16 and 0.9.17 suddenly drops and this becomes a key reason to affect the average. This can be explained by the fact that 115 classes were modified not included in this research as well as 19 *removed* and 30 *added* at version 0.9.17.

To test the hypotheses if the growth trend of classes is actually related to the metric values, we calculated correlations between the number of classes and one of the average *fan-in* coupling, *fan-out* coupling and cohesion.



	0.9.0	0.9.1	0.9.2	0.9.3	0.9.4	0.9.5	0.9.6	0.9.7	0.9.8	0.9.9	0.9.10	0.9.11	0.9.12	0.9.13	0.9.14	0.9.15	0.9.16	0.9.17	0.9.18	0.9.19	0.9.20	0.9.21
No. of Classes	139	138	144	257	275	327	329	353	356	361	352	365	377	383	395	404	412	423	432	447	448	460

Figure 5-1: Number of class growth



	0.9.0	0.9.1	0.9.2	0.9.3	0.9.4	0.9.5	0.9.6	0.9.7	0.9.8	0.9.9	0.9.10	0.9.11	0.9.12	0.9.13	0.9.14	0.9.15	0.9.16	0.9.17	0.9.18	0.9.19	0.9.20	0.9.21
Avg. fan-in coupling	11.9	12	11.8	11	13.1	12.8	12.8	12.7	12.8	13	13.2	13.4	13.6	14	15.3	15.2	15.1	15	14.7	14.2	14.3	14.5
Avg. fan-out coupling	12.1	12.2	12	11	14.1	13.8	13.8	13.5	13.6	14.2	14.1	14.4	14.4	14.8	15.4	15.3	15.2	15.2	14.7	14.3	14.3	14.6
Avg. cohesion	12.7	12.9	12.9	11.6	12.8	12.6	12.6	12.4	12.5	12.7	13.2	13.4	13.5	13.8	14.1	14	13.8	9.8	9.9	9.8	9.8	9.8

Figure 5-2: Average *fan-in/out* coupling and cohesion

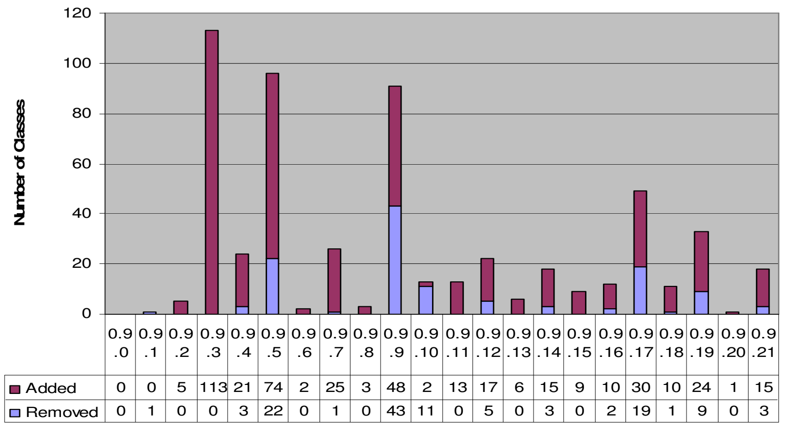
The average *fan-in/out* coupling is the average of the *fan-in/out* coupling metric values of all classes in each version of the program. The average cohesion is the average of the cohesion metric values of all classes in each version. Underlying assumptions are that the number of classes is positively related to the average *fan-in/out* coupling, but is not positively related to the average cohesion.

As we expected, there are strong correlations between the number of classes and the average *fan-in/fan-out* couplings with 0.813 and 0.826, respectively, in the pearson correlation, and at the significant level of p-value= 0.000 (Table 5-3). This statistical analysis strongly supports the first two hypotheses we made, and agrees with the previous research statements about the relationships between the number of classes and the coupling metrics, which stated that if the number of classes increases then coupling metrics increase.

Moreover, Lehman's 2nd law (Increasing complexity) of software evolution states that as a system evolves the complexity of the system increases unless work is done to maintain it. Since *JFreeChart* is an object oriented system written in Java, it is known that the complexity of a Java program depends largely on the coupling metrics among the classes.

Table 5-3: Correlation between the number of classes and coupling/cohesion

At each version	**Number of classes**	
	Pearson correlation	P-value
Average *fan-in* coupling	0.813	0.000
Average *fan-Out* coupling	0.826	0.000
Average cohesion	-0.356	0.104



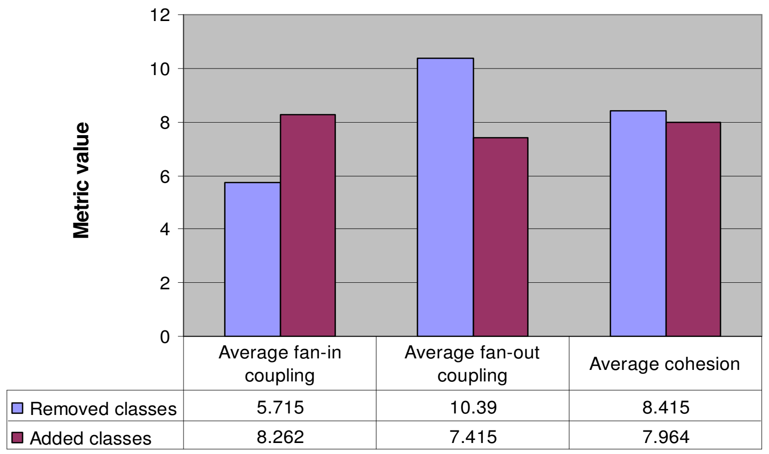
	0.9.0	0.9.1	0.9.2	0.9.3	0.9.4	0.9.5	0.9.6	0.9.7	0.9.8	0.9.9	0.9.10	0.9.11	0.9.12	0.9.13	0.9.14	0.9.15	0.9.16	0.9.17	0.9.18	0.9.19	0.9.20	0.9.21
🟥 Added	0	0	5	113	21	74	2	25	3	48	2	13	17	6	15	9	10	30	10	24	1	15
🟦 Removed	0	1	0	0	3	22	0	1	0	43	11	0	5	0	3	0	2	19	1	9	0	3

Figure 5-3: Number of classes *removed* and *added*

Figure 5-2 shows that as *JFreeChart* evolved, the coupling of the system increased, thus complexity increases as well, following Lehman's 2nd law of evolution with some minor exceptions.

***Removed* and *Added* Classes**

Figure 5-3 shows the numbers of classes *removed* (*group R*) and *added* (*group A*) across all versions. We noticed that the software is constantly changed between versions and, in most cases, many more classes are *added* (total of 444) than *removed* (total of 123). This changing nature of *JFreeChart* follows Lehman's 1*st* law (Continuing change). Almost 50% of *group A* were added around the beginning of the evolution (213 out of 444), prior to version 0.9.5. According to [1], this is a common phenomenon. About 65% of group A were added before version 0.9.9 (291 out of 444). In addition, there seems to be important changes at version 0.9.9 by adding 48 classes and removing 43 (36% of the *removed*).



	Average fan-in coupling	Average fan-out coupling	Average cohesion
▨ Removed classes	5.715	10.39	8.415
■ Added classes	8.262	7.415	7.964

Figure 5- 4: Average coupling/cohesion of the classes *removed* and *added*

To test the last three hypotheses, we calculated average *Fan-in*/out coupling and cohesion metrics for both *group A* and *group R*. It is the average of metric values of all classes *removed*/*added* in each version. Figure 5-4 shows the metric values and compares them in bar graphs. We were expecting to see higher *Fan-in* and cohesion and lower *fan-out* in *group A* than in *group R*.

The results reveal higher *fan-in* coupling and lower *fan-out* coupling for the *added* class group than those for the *removed* class group thus, support Hypotheses 3 and 4. It implies directly that the *added* classes have better software quality than the *removed* classes in terms of coupling. This result is very interesting because the 7th law (Declining quality) of Lehman's software evolution states that E-type programs will be perceived as of declining quality unless adapted to a changing operation environment. We defined reusability as a quality factor for *JFreeChart* since it is a library which is intended to be reused by other applications. We measured *fan-out* and *fan-in* coupling metrics over time to see the trend of the quality in terms of reusability. As we mentioned earlier, low *fan-out* and high *fan-in* coupling are desirable for the classes to be reused. Therefore we can say that with few exceptions, the evolution of the *JFreeChart* does not follow Lehman's 7th law of evolution.

Based on the average cohesion metric values as shown in Figure 5-4, we found no big difference between the two groups and therefore reject Hypothesis 5.

For the averages of the metrics, we looked into each version as shown in Figures 5-5, 5-6, and 5-7. Since we have different numbers of classes across all versions, we normalized the average metric values by dividing the number of classes at each version.

For the *fan-in* coupling in Figure 5-5, we observe two spikes at versions 0.9.3 and 0.9.9. The first is for the *added* and the second is for the *removed*. Although the overall average seems to be influenced by them, the metrics for the *added* are higher and stronger than the removed, which is desirable and expected because it is reusable. More importantly the average at version 0.9.3 is the one with 113 *added* classes.

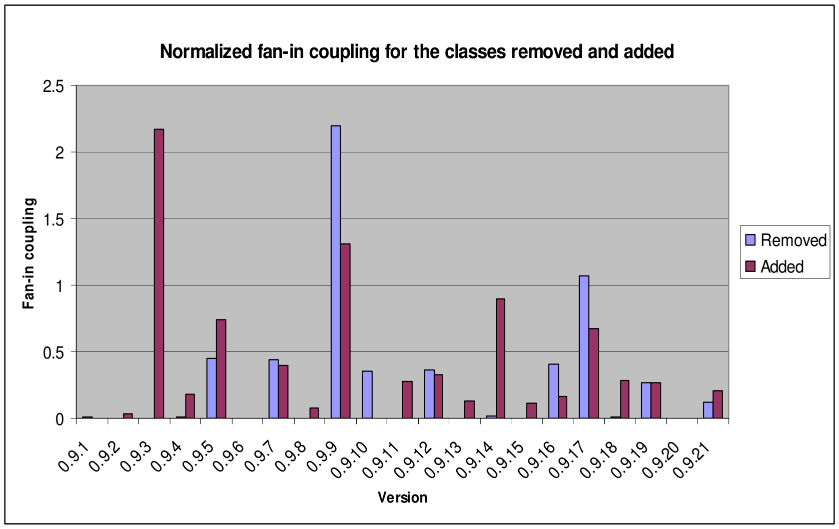


Figure 5-5: Normalized *fan-in* coupling

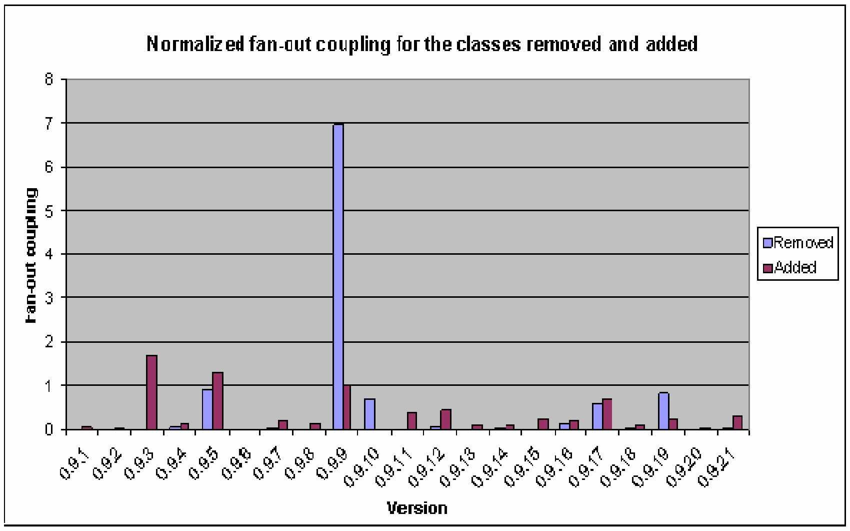


Figure 5-6: Normalized *fan-out* coupling

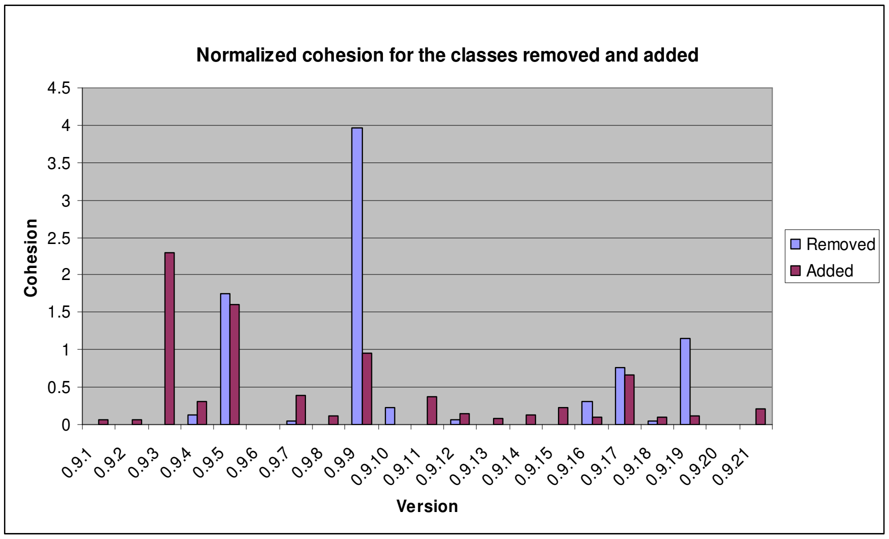


Figure 5-7: Normalized cohesion

For the *fan-out* coupling in Figure 5-6, we again notice a spike at version 0.9.9. This one is especially important because it is the average of 43 *removed* classes out of 123. It is almost 40% of all the *removed* classes at a single version, and this plays an important role toward the undesirable quality of software because it is hard to maintain, thus removed.

For the cohesion value per version in Figure 5-7, the removed classes at version 0.9.9 have high cohesion, even though this could be explained that they were removed because of the high *fan-out* coupling. However, the high cohesion for the *added* at version 0.9.3 is meaningful because it is the average of 113 classes while we can't say that the *added* class group has better quality in terms of cohesion because of the data in Figure 5-4**.**

Obvious common phenomena from these three Figures (5-5 – 5-7) is that the 113 classes added at version 0.9.3 represent high fan-in coupling and cohesion, which is ideal, and the 43 classes removed at version 0.9.9 represent high fan-in/out and cohesion. The high fan-out coupling resulted in having these 43 classes removed from the software.

5.1.5 Summary

In this empirical study, we have mainly focused on tracking the reusability of an open software system*, JFreeChart,* over its evolution with *fan-in* and *fan-out* couplings for *added* and *removed* classes. We found that the number of classes increases gradually over most releases, and they have strong correlations with coupling metrics but not positively related to the cohesion. These confirm the expectations about the relationship between them. We also found that the *added* classes have higher *fan-in* coupling and lower *fan-out* coupling comparing to the *removed* classes. Low *fan-out* and high *fan-in* are desirable in term of reusability since a high fan-out means difficulty to reuse a class and a high fan-in represents a high level of reuse. It also has been found that evolution of this software system is consistent with Lehman's 1*st*, 2*nd*, and 6*th* laws of software evolution.

While more research would be required to make any firm conclusions, this observation leads us to believe that the reusability of *JFreeChart* has improved along with its evolution. In this way, applying metrics from JamTool over the evolution of the software can aid a software engineer to understand how a system has evolved over time.

6 ANALYZING SOFTWARE FOR REUSE AND MAINTENANCE

We applied software metrics and visualization approach to understand the software evolution in Chapter 5. According to the empirical study, there was a big change of coupling metric values from 0.9.3 to 0.9.4 as reported in Table 5-2 and Figure 5-2. This chapter presents a case study to investigate if the metrics defined and implemented by *JamTool* can be used to capture the difference between two consecutive versions on the evolution of *JFreeChart*.

6.1 Added and Removed Classes

When JFreeChart evolves from version 0.9.3 to version 0.9.4, twenty-one new classes were added and three classes were removed. Tables 6-1 and 6-2 summarize fan-in and fan-out couplings for the added and removed classes. The Class Counting Coupling (CCC) fan-out of a class, C, is the number of other classes that are referenced in C. A reference to another class, A, is a reference to a method or a data member of class A. In the CCC fan-out of a class, multiple accesses to the same method or data element are counted as one access. The CCC fan-in of a class, C, is the number of other classes that reference class C. In the CCC fan-in of a class, multiple accesses are also counted as one access.

High CCC fan-out of a class represents couplings to many other classes and thus the class is hard to be reused because this class depends on many other classes. High CCC fan-in of a class represents good object design and high level of reuse but it may be risky to change this class because many classes depend on it.

Strength Counting Coupling (SCC) fan-in and fan-out coupling counts all references between classes. As shown in Table 6-1, added classes have higher (CCC average 2.5) fan-out coupling than fan-in coupling (CCC average 1.3).

Table 6-1: Added classes into 0.9.4

Class Name	CCC		SCC	
	Fan-out	Fan-in	Fan-out	Fan-in
ArrowNeedle	1	2	1	3
CompassPlot	14	0	27	0
DatasetGroup	0	4	0	8
DrawableLegendItem	1	3	3	48
FastScatterPlot	7	0	21	0
Function2D	0	2	0	2
IntervalCategoryToolTipGenerator	2	1	4	1
JThermometer	5	0	35	0
LineFunction2D	1	0	1	0
LineNeedle	1	1	1	1
LongNeedle	1	1	1	1
MeterNeedle	0	8	0	11
PinNeedle	1	1	1	1
PlumNeedle	1	1	1	1
PointerNeedle	1	1	1	1
PowerFunction2D	1	0	1	0
Regression	0	0	0	0
ShipNeedle	1	1	1	1
XYDotRenderer	2	0	2	0
WindNeedle	1	1	1	1
ThermometerPlot	13	1	62	15
Average	2.5	1.3	7.8	4.5

Six classes (*XYDotRenderer, FastScatterPlot, JThermometer, LineFunction2D, PowerFunction2D, and CompassPlot*) have only fan-out couplings and three classes (*DatasetGroup*, *Function2D*, and *MeterNeedle*) have only fan-in couplings. Class *Regression* is added without any relation to other classes. This class may be ready to provide independent service to other software application.

Class *ThermometerPlot* depends on 13 classes with 62 fan-out couplings and 1 class depends on this class with 15 fan-in couplings. Nine added classes have both fan-out and fan-in couplings. Class *DrawableLegendItem* has 1 fan-out class and 3 fan-in classes with 3, and 48 couplings, respectively. Therefore, we need to pay more attention to this class than other classes among the added classes.

In Table 6-2, Class *WindAxis* is removed, but it does not affect the rest of the system because no other classes depended on this class. Classe *ToolTipsCollection* is removed and one class depends on this class with one coupling. Class *ToolTip* is removed and one class depends on this class with six couplings. Even if only one class depends on the removed classes, we still need to test the effect of the removed classes because this one class may trigger riffle effects to other classes in the system.

Table 6-2: Removed classes from 0.9.3

	CCC		SCC	
Class	Fan-out	Fan-in	Fan-out	Fan-in
WindAxis	2	0	6	0
ToolTipsCollection	0	1	0	1
ToolTip	0	1	0	6

6.2 Modified Classes

We compare CCC fan-in and fan-out couplings between 0.9.3 and 0.9.4 to see if there are changes in terms of the number of coupled classes. Table 6-3 shows the changed classes that have big differences in terms of the number of coupled classes. As shown in Table 6-3 (a), class *ChartFactory* depends on 15 new classes; there are only 2 classes depend on more than 3 new classes, but 4 classes decrease the number of coupled classes in version 0.9.4.

Table 6-3: Changed classes with at least 3 differences.

CCC(Fan-out)	0.9.3	0.9.4	Change
ChartFactory	38	53	15
StandardLegendItemLayout	2	6	4
AbstractXYItemRenderer	6	9	3
AreaCategoryItemRenderer	2	5	3
DateAxis	5	8	3
HorizontalDateAxis	9	12	3
StandardCategoryToolTipGenerator	1	4	3
ChartUtilities	6	3	-3
StandardLegend	5	2	-3
JThermometer	4	0	-4
StackedHorizontalBarRenderer	5	1	-4

(a) Changed classes with big difference of fan-out (CCC)

CCC(Fan-in)	0.9.3	0.9.4	Change
LegendItemCollection	1	13	12
CategoryURLGenerator	2	13	11
LegendItem	3	10	7
EntityCollection	20	24	4
StandardCategoryToolTipGenerator	0	4	4
TickUnits	3	7	4
CategoryPlot	7	10	3
DateTickUnit	0	3	3
Plot	17	20	3
StackedVerticalBarRenderer3D	10	1	-9

(b) Changed classes with big difference of fan-in (CCC)

In Figure 6-3 (b), classes *LegendItemCollection*, *CategoryURLGenerator*, and *LegendItem* in version 0.9.4 depend on more than 7 new classes and 9 classes stop depending on class *StackedVerticalBarRenderer3D*. Table 6-4 and Figure 6-1 summarize fan-in/out differences in these two versions. CCC represents the number of coupled classes and SCC represents the coupling strength.

Table 6-4: Fan-in/out differences in two versions

		Average		Min		Median		Max	
		0.9.3	0.9.4	0.9.3	0.9.4	0.9.3	0.9.4	0.9.3	0.9.4
CCC	Fan-in	2.9	3.1	0	0	1	1	36	38
	Fan-out	2.9	3.1	0	0	2	2	38	53
SCC	Fan-in	12.8	16.2	0	0	3	3	255	398
	Fan-out	12.8	16.2	0	0	3	2	331	447

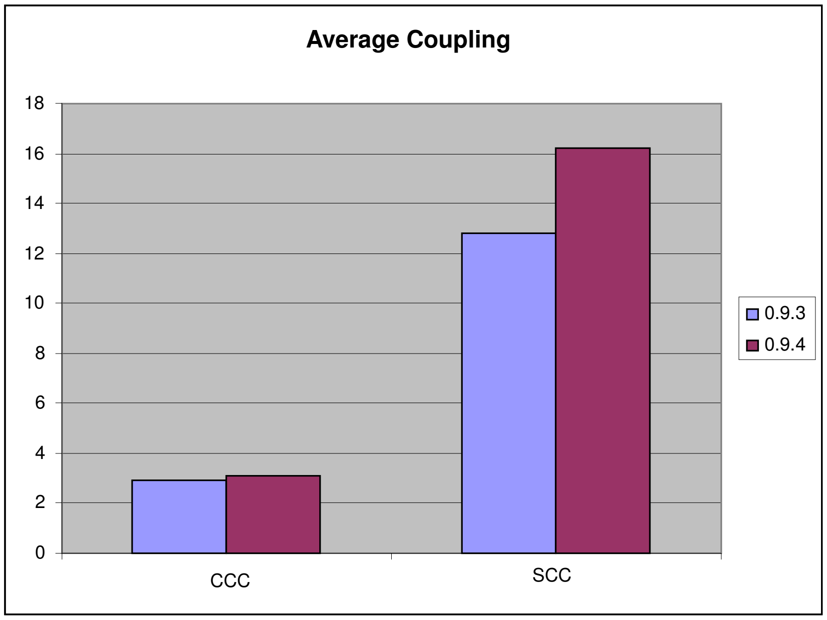
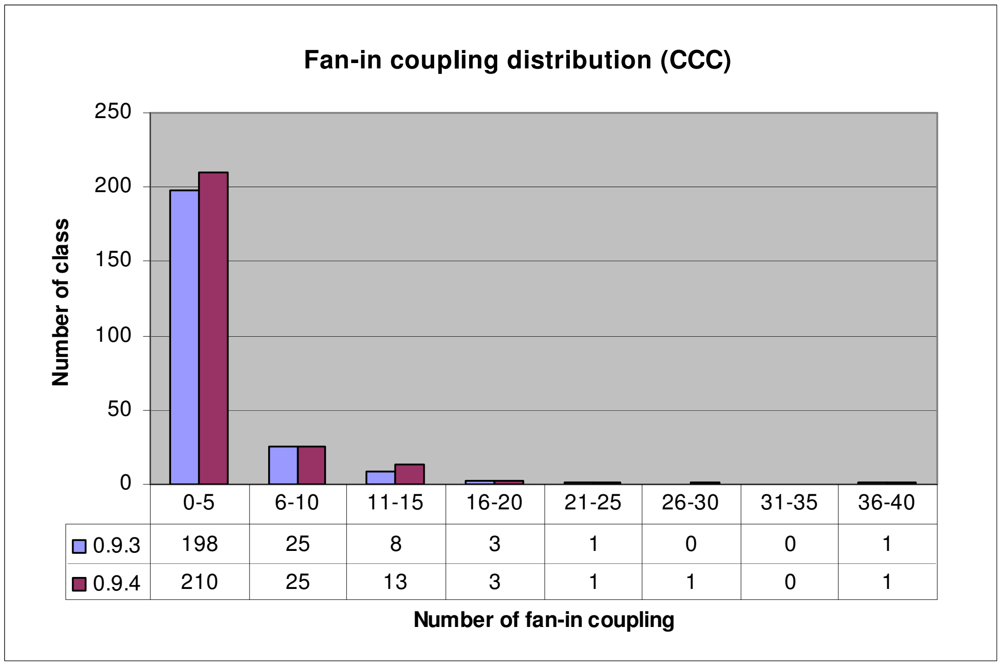
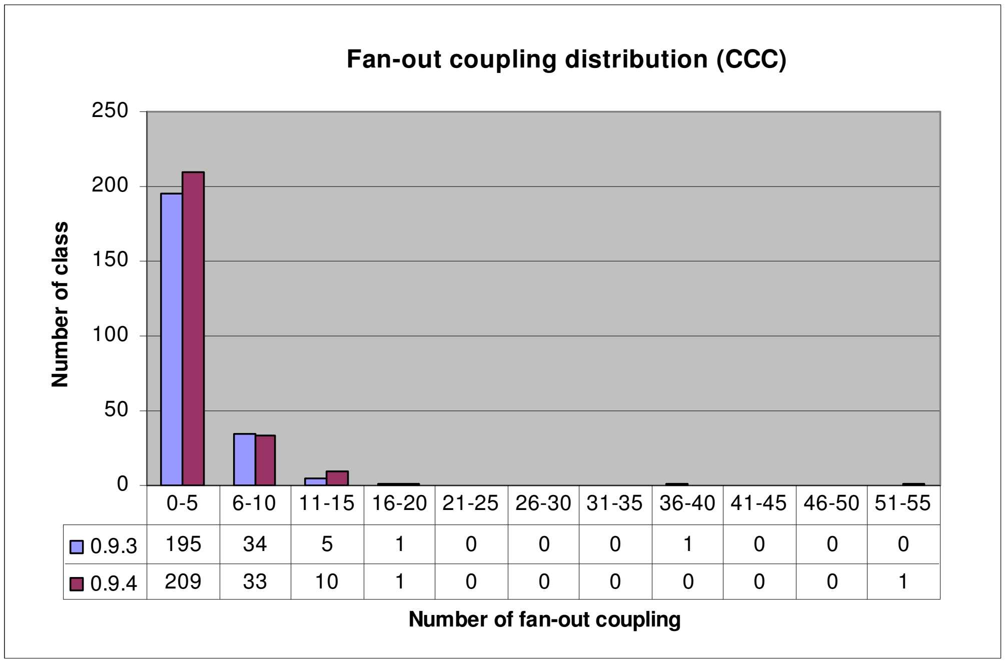


Figure 6-1: Average coupling comparison of changed classes



	0-5	6-10	11-15	16-20	21-25	26-30	31-35	36-40
🟦 0.9.3	198	25	8	3	1	0	0	1
🟥 0.9.4	210	25	13	3	1	1	0	1

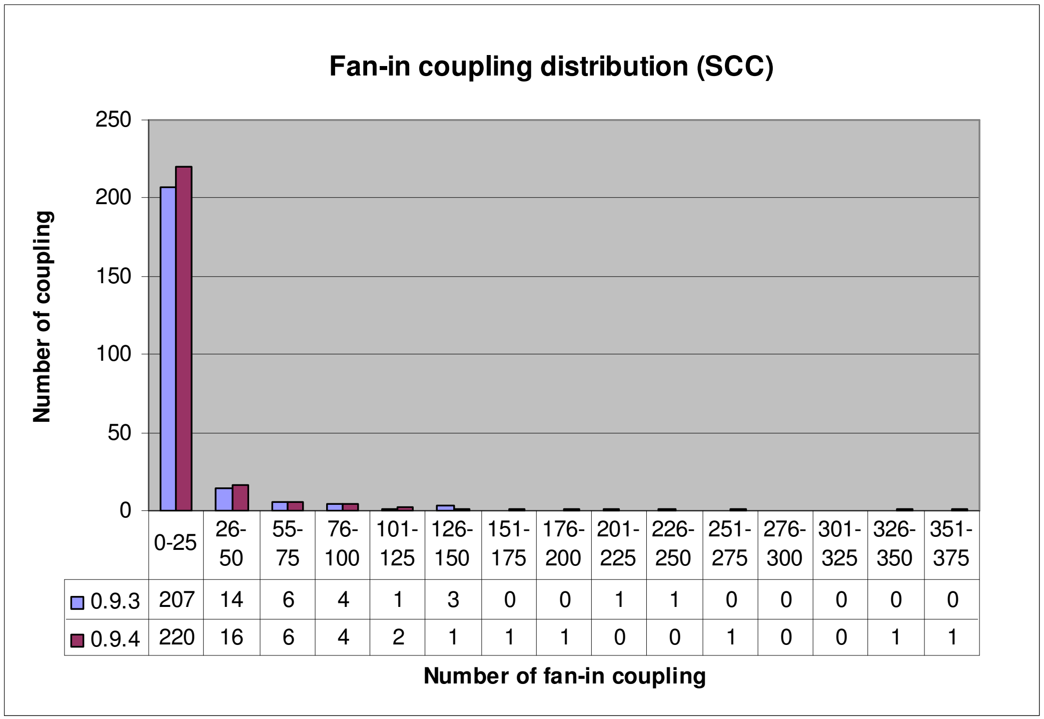
(a) Fan-in coupling distribution



	0-5	6-10	11-15	16-20	21-25	26-30	31-35	36-40	41-45	46-50	51-55
🟦 0.9.3	195	34	5	1	0	0	0	1	0	0	0
🟥 0.9.4	209	33	10	1	0	0	0	0	0	0	1

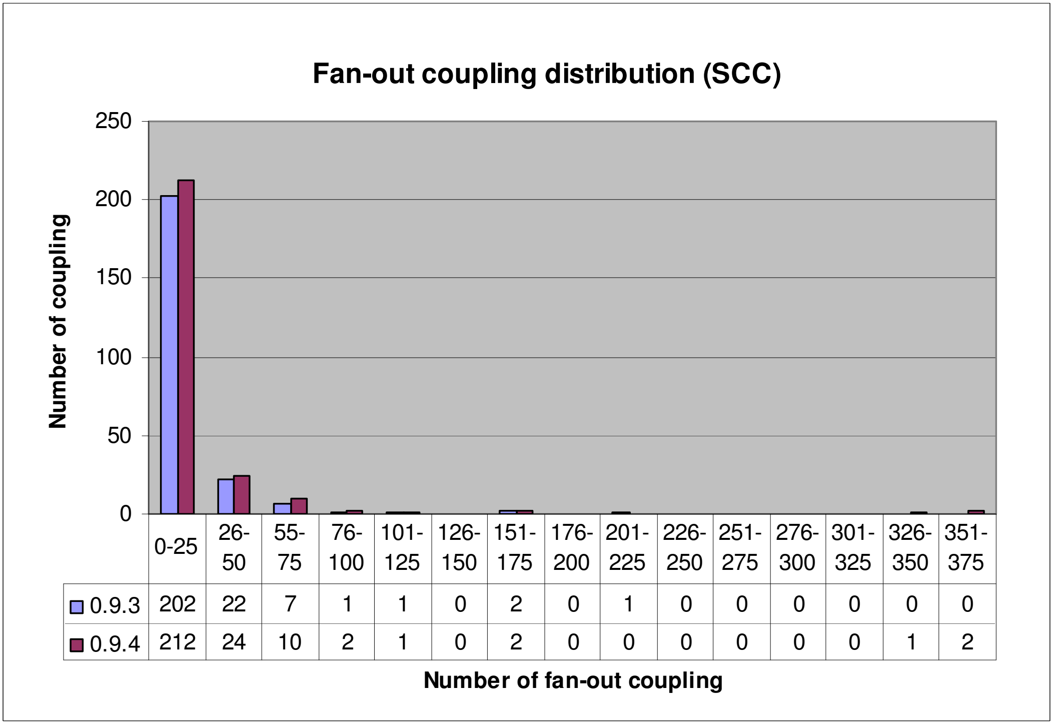
(b) Fan-out coupling distribution

Figure 6-2: Coupling (CCC) distribution in two versions



	0-25	26-50	55-75	76-100	101-125	126-150	151-175	176-200	201-225	226-250	251-275	276-300	301-325	326-350	351-375
▨ 0.9.3	207	14	6	4	1	3	0	0	1	1	0	0	0	0	0
■ 0.9.4	220	16	6	4	2	1	1	1	0	0	1	0	0	1	1

(a) Fan-in coupling distribution



	0-25	26-50	55-75	76-100	101-125	126-150	151-175	176-200	201-225	226-250	251-275	276-300	301-325	326-350	351-375
▨ 0.9.3	202	22	7	1	1	0	2	0	1	0	0	0	0	0	0
■ 0.9.4	212	24	10	2	1	0	2	0	0	0	0	0	0	1	2

(b) Fan-out coupling distribution

Figure 6-3: Coupling (SCC) distribution in two versions

When *JFreeChart* evolves from 0.9.3 to 0.9.4, the average number of coupled classes is increased from 2.9 to 3.1 and the average coupling strength is increased from 12.8 to 16.2. This result means that, in version 0.9.3, each class depends on 2.9 classes on average and references/uses other classes about 12.8 times. Each class, in version 0.9.4, depends on 3.1 classes on average and references/uses other classes about 16.2 times. Therefore, we can say that version 0.9.4 is more difficult to reuse and maintain than 0.9.3.

Figures 6-2 and 6-3 summarize fan-in/out coupling distributions for these two versions. There are classes with high coupling metrics which we need to pay more attention and monitor their changes. Figure 6-2 shows fan-in/out coupling distributions in terms of the number of classes. Most classes have coupled to fewer than 5 classes.

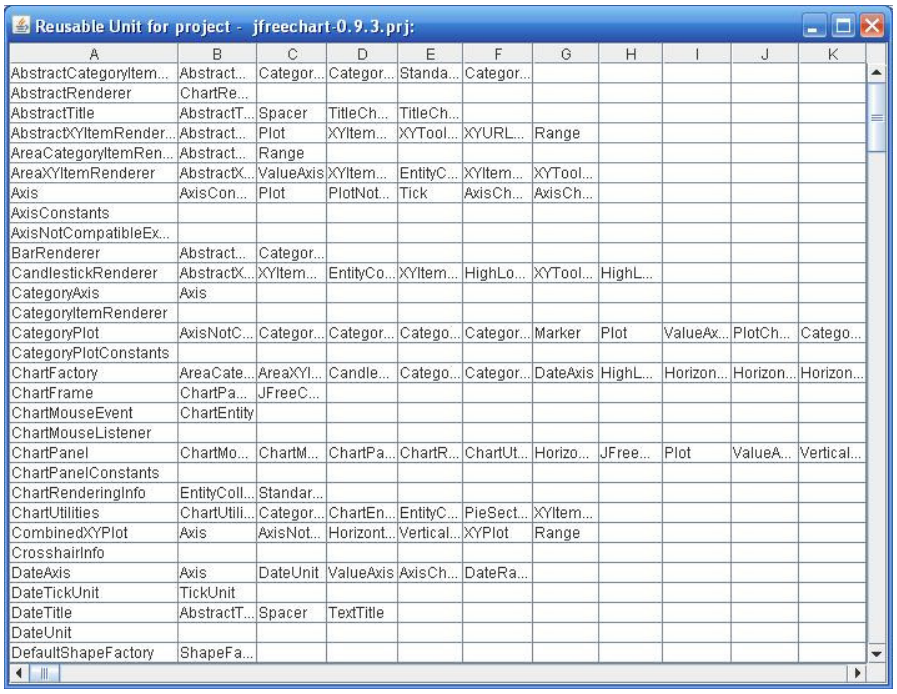
Figure 6-3 shows fan-in/out coupling distribution in terms of the number of actual couplings. It is a distribution of the SCC metrics. Most classes have fewer than 25 couplings and only very few classes have high couplings.

6.3 Reusable Unit and Maintainable Unit

Reusable unit is a collection of a target class and its related classes we should reuse together. Identifying a reusable unit means that each class has its own reusable unit with other classes which the class depends on. The identification of a reusable unit of classes requires an understanding of the relation of classes in a software system. A maintainable unit contains a target class and its related classes we should test together.

Reusable unit and maintainable unit are necessary to understand software structure and, more importantly, to serve as a source of information for reuse and maintenance.

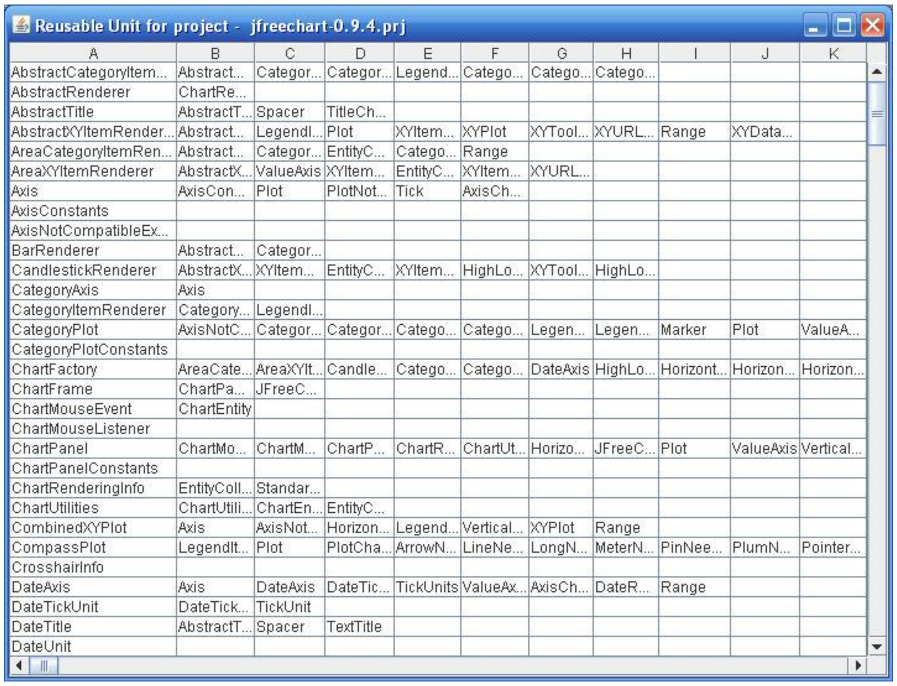
Figure 6-4 shows the reusable units in versions 0.9.3 and 0.9.4. From these reusable units, progression of the reusable units are captured. For example, class *AbstractCategoryItemRender* depends on 5 classes (*StandardCategoryToolTipGenerator*, *CategoryRender*, *CategoryToolTipGenerator*, *AbstractRender*, *CategoryURLGenerator*) in version 0.9.3, which make a unique reusable unit, but 2 new classes (*CategoryDataset*, *LegendItem*) are added into the reusable unit in version 0.9.4.



Reusable Unit for project - jfreechart-0.9.3.prj:

A	B	C	D	E	F	G	H	I	J	K
AbstractCategoryItem...	Abstract...	Categor...	Categor...	Standa...	Categor...					
AbstractRenderer	ChartRe...									
AbstractTitle	AbstractT...	Spacer	TitleCh...	TitleCh...						
AbstractXYItemRender...	Abstract...	Plot	XYItem...	XYTool...	XYURL...	Range				
AreaCategoryItemRen...	Abstract...	Range								
AreaXYItemRenderer	AbstractX...	ValueAxis	XYItem...	EntityC...	XYItem...	XYTool...				
Axis	AxisCon...	Plot	PlotNot...	Tick	AxisCh...	AxisCh...				
AxisConstants										
AxisNotCompatibleEx...										
BarRenderer	Abstract...	Categor...								
CandlestickRenderer	AbstractX...	XYItem...	EntityCo...	XYItem...	HighLo...	XYTool...	HighL...			
CategoryAxis	Axis									
CategoryItemRenderer										
CategoryPlot	AxisNotC...	Categor...	Categor...	Catego...	Categor...	Marker	Plot	ValueAx...	PlotCh...	Catego...
CategoryPlotConstants										
ChartFactory	AreaCate...	AreaXYI...	Candle...	Catego...	Categor...	DateAxis	HighL...	Horizon...	Horizon...	Horizon...
ChartFrame	ChartPa...	JFreeC...								
ChartMouseEvent	ChartEntity									
ChartMouseListener										
ChartPanel	ChartMo...	ChartM...	ChartPa...	ChartR...	ChartUt...	Horizo...	JFree...	Plot	ValueA...	Vertical...
ChartPanelConstants										
ChartRenderingInfo	EntityColl...	Standar...								
ChartUtilities	ChartUtili...	Categor...	ChartEn...	EntityC...	PieSect...	XYItem...				
CombinedXYPlot	Axis	AxisNot...	Horizont...	Vertical...	XYPlot	Range				
CrosshairInfo										
DateAxis	Axis	DateUnit	ValueAxis	AxisCh...	DateRa...					
DateTickUnit	TickUnit									
DateTitle	AbstractT...	Spacer	TextTitle							
DateUnit										
DefaultShapeFactory	ShapeFa...									

(a) Reusable unit in version 0.9.3

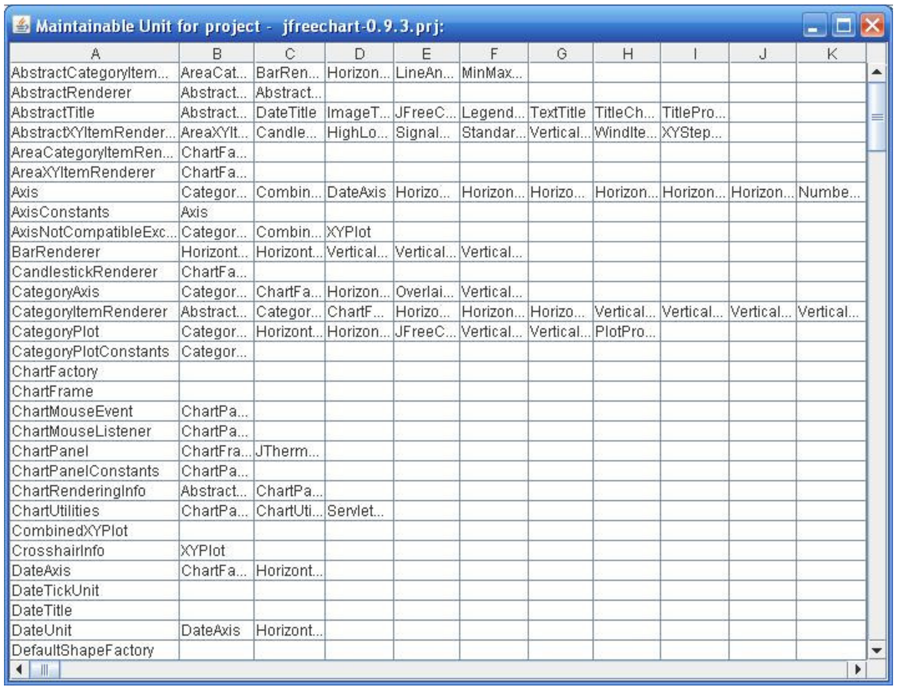


Reusable Unit for project - jfreechart-0.9.4.prj

A	B	C	D	E	F	G	H	I	J	K
AbstractCategoryItem...	Abstract...	Categor...	Categor...	Legend...	Catego...	Catego...	Catego...			
AbstractRenderer	ChartRe...									
AbstractTitle	AbstractT...	Spacer	TitleCh...							
AbstractXYItemRender...	Abstract...	LegendI...	Plot	XYItem...	XYPlot	XYTool...	XYURL...	Range	XYData...	
AreaCategoryItemRen...	Abstract...	Categor...	EntityC...	Catego...	Range					
AreaXYItemRenderer	AbstractX...	ValueAxis	XYItem...	EntityC...	XYItem...	XYURL...				
Axis	AxisCon...	Plot	PlotNot...	Tick	AxisCh...					
AxisConstants										
AxisNotCompatibleEx...										
BarRenderer	Abstract...	Categor...								
CandlestickRenderer	AbstractX...	XYItem...	EntityC...	XYItem...	HighLo...	XYTool...	HighLo...			
CategoryAxis	Axis									
CategoryItemRenderer	Category...	LegendI...								
CategoryPlot	AxisNotC...	Categor...	Categor...	Catego...	Catego...	Legen...	Legen...	Marker	Plot	ValueA...
CategoryPlotConstants										
ChartFactory	AreaCate...	AreaXYIt...	Candle...	Catego...	Catego...	DateAxis	HighLo...	Horizont...	Horizon...	Horizon...
ChartFrame	ChartPa...	JFreeC...								
ChartMouseEvent	ChartEntity									
ChartMouseListener										
ChartPanel	ChartMo...	ChartM...	ChartP...	ChartR...	ChartUt...	Horizo...	JFreeC...	Plot	ValueAxis	Vertical...
ChartPanelConstants										
ChartRenderingInfo	EntityColl...	Standar...								
ChartUtilities	ChartUtili...	ChartEn...	EntityC...							
CombinedXYPlot	Axis	AxisNot...	Horizon...	Legend...	Vertical...	XYPlot	Range			
CompassPlot	LegendIt...	Plot	PlotCha...	ArrowN...	LineNe...	LongN...	MeterN...	PinNee...	PlumN...	Pointer...
CrosshairInfo										
DateAxis	Axis	DateAxis	DateTic...	TickUnits	ValueAx...	AxisCh...	DateR...	Range		
DateTickUnit	DateTick...	TickUnit								
DateTitle	AbstractT...	Spacer	TextTitle							
DateUnit										

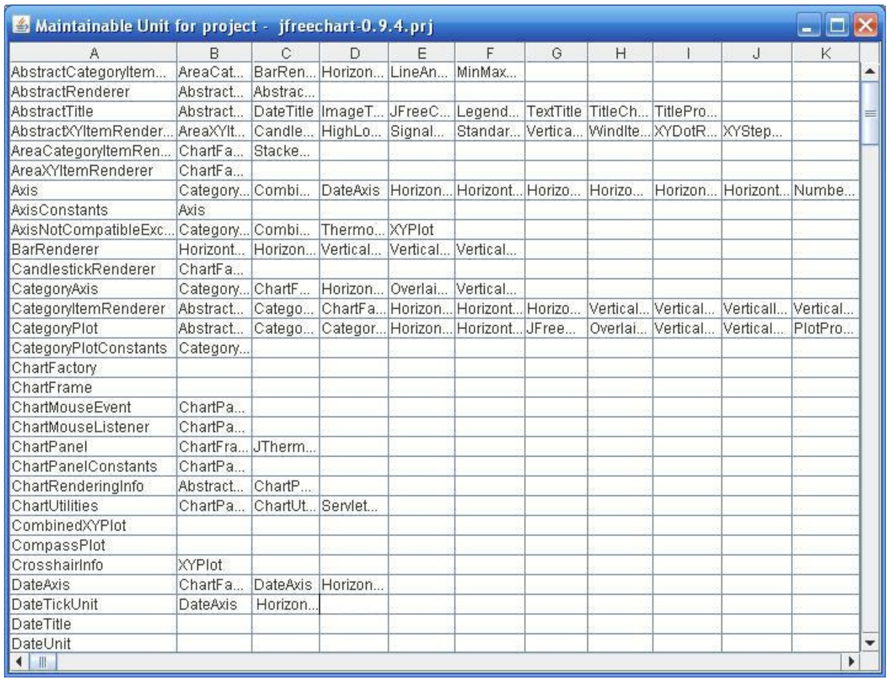
(b) Reusable unit in version 0.9.4

Figure 6-4: Reusable unit


Maintainable Unit for project - jfreechart-0.9.3.prj:

A	B	C	D	E	F	G	H	I	J	K
AbstractCategoryItem...	AreaCat...	BarRen...	Horizon...	LineAn...	MinMax...					
AbstractRenderer	Abstract...	Abstract...								
AbstractTitle	Abstract...	DateTitle	ImageT...	JFreeC...	Legend...	TextTitle	TitleCh...	TitlePro...		
AbstractXYItemRender...	AreaXYIt...	Candle...	HighLo...	Signal...	Standar...	Vertical...	WindIte...	XYStep...		
AreaCategoryItemRen...	ChartFa...									
AreaXYItemRenderer	ChartFa...									
Axis	Categor...	Combin...	DateAxis	Horizo...	Horizon...	Horizo...	Horizon...	Horizon...	Horizon...	Numbe...
AxisConstants	Axis									
AxisNotCompatibleExc...	Categor...	Combin...	XYPlot							
BarRenderer	Horizont...	Horizont...	Vertical...	Vertical...	Vertical...					
CandlestickRenderer	ChartFa...									
CategoryAxis	Categor...	ChartFa...	Horizon...	Overlai...	Vertical...					
CategoryItemRenderer	Abstract...	Categor...	ChartF...	Horizo...	Horizon...	Horizo...	Vertical...	Vertical...	Vertical...	Vertical...
CategoryPlot	Categor...	Horizont...	Horizon...	JFreeC...	Vertical...	Vertical...	PlotPro...			
CategoryPlotConstants	Categor...									
ChartFactory										
ChartFrame										
ChartMouseEvent	ChartPa...									
ChartMouseListener	ChartPa...									
ChartPanel	ChartFra...	JTherm...								
ChartPanelConstants	ChartPa...									
ChartRenderingInfo	Abstract...	ChartPa...								
ChartUtilities	ChartPa...	ChartUti...	Servlet...							
CombinedXYPlot										
CrosshairInfo	XYPlot									
DateAxis	ChartFa...	Horizont...								
DateTickUnit										
DateTitle										
DateUnit	DateAxis	Horizont...								
DefaultShapeFactory										

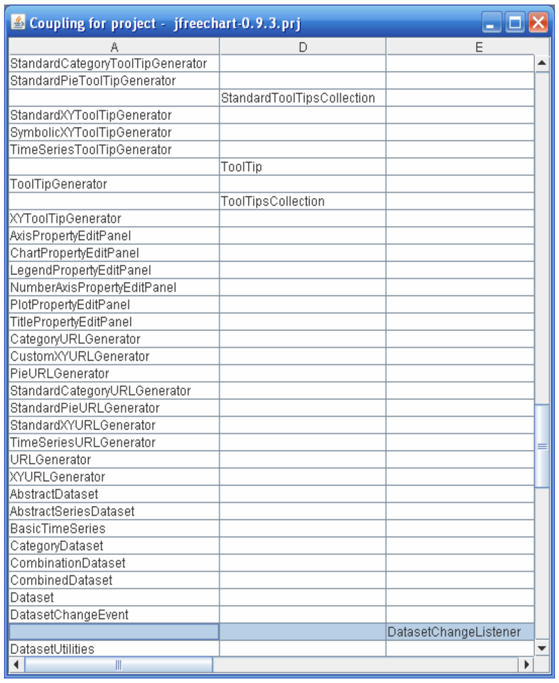
(a) Maintainable unit in version 0.9.3


Maintainable Unit for project - jfreechart-0.9.4.prj

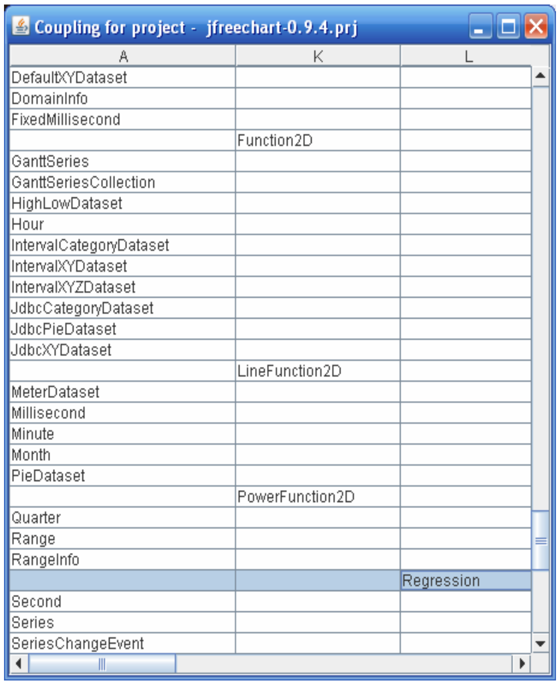
A	B	C	D	E	F	G	H	I	J	K
AbstractCategoryItem...	AreaCat...	BarRen...	Horizon...	LineAn...	MinMax...					
AbstractRenderer	Abstract...	Abstrac...								
AbstractTitle	Abstract...	DateTitle	ImageT...	JFreeC...	Legend...	TextTitle	TitleCh...	TitlePro...		
AbstractXYItemRender...	AreaXYIt...	Candle...	HighLo...	Signal...	Standar...	Vertica...	WindIte...	XYDotR...	XYStep...	
AreaCategoryItemRen...	ChartFa...	Stacke...								
AreaXYItemRenderer	ChartFa...									
Axis	Category...	Combi...	DateAxis	Horizon...	Horizont...	Horizo...	Horizo...	Horizon...	Horizont...	Numbe...
AxisConstants	Axis									
AxisNotCompatibleExc...	Category...	Combi...	Thermo...	XYPlot						
BarRenderer	Horizont...	Horizon...	Vertical...	Vertical...	Vertical...					
CandlestickRenderer	ChartFa...									
CategoryAxis	Category...	ChartF...	Horizon...	Overlai...	Vertical...					
CategoryItemRenderer	Abstract...	Catego...	ChartFa...	Horizon...	Horizont...	Horizo...	Vertical...	Vertical...	VerticalI...	Vertical...
CategoryPlot	Abstract...	Catego...	Categor...	Horizon...	Horizont...	JFree...	Overlai...	Vertical...	Vertical...	PlotPro...
CategoryPlotConstants	Category...									
ChartFactory										
ChartFrame										
ChartMouseEvent	ChartPa...									
ChartMouseListener	ChartPa...									
ChartPanel	ChartFra...	JTherm...								
ChartPanelConstants	ChartPa...									
ChartRenderingInfo	Abstract...	ChartP...								
ChartUtilities	ChartPa...	ChartUt...	Servlet...							
CombinedXYPlot										
CompassPlot										
CrosshairInfo	XYPlot									
DateAxis	ChartFa...	DateAxis	Horizon...							
DateTickUnit	DateAxis	Horizon...								
DateTitle										
DateUnit										

(a) Maintainable unit in version 0.9.4

Figure 6-5: Maintainable unit


Coupling for project - jfreechart-0.9.3.prj

A	D	E
StandardCategoryToolTipGenerator		
StandardPieToolTipGenerator		
	StandardToolTipsCollection	
StandardXYToolTipGenerator		
SymbolicXYToolTipGenerator		
TimeSeriesToolTipGenerator		
	ToolTip	
ToolTipGenerator		
	ToolTipsCollection	
XYToolTipGenerator		
AxisPropertyEditPanel		
ChartPropertyEditPanel		
LegendPropertyEditPanel		
NumberAxisPropertyEditPanel		
PlotPropertyEditPanel		
TitlePropertyEditPanel		
CategoryURLGenerator		
CustomXYURLGenerator		
PieURLGenerator		
StandardCategoryURLGenerator		
StandardPieURLGenerator		
StandardXYURLGenerator		
TimeSeriesURLGenerator		
URLGenerator		
XYURLGenerator		
AbstractDataset		
AbstractSeriesDataset		
BasicTimeSeries		
CategoryDataset		
CombinationDataset		
CombinedDataset		
Dataset		
DatasetChangeEvent		
		DatasetChangeListener
DatasetUtilities		


Coupling for project - jfreechart-0.9.4.prj

A	K	L
DefaultXYDataset		
DomainInfo		
FixedMillisecond		
	Function2D	
GanttSeries		
GanttSeriesCollection		
HighLowDataset		
Hour		
IntervalCategoryDataset		
IntervalXYDataset		
IntervalXYZDataset		
JdbcCategoryDataset		
JdbcPieDataset		
JdbcXYDataset		
	LineFunction2D	
MeterDataset		
Millisecond		
Minute		
Month		
PieDataset		
	PowerFunction2D	
Quarter		
Range		
RangeInfo		
		Regression
Second		
Series		
SeriesChangeEvent		

(a) Connected unit in 0.9.3 (b) Connected unit in 0.9.4

Figure 6-6 : Connected units in two versions

Figure 6-5 shows maintainable units in two versions. From these maintainable units, we can capture the progression how classes depend on a particular class. For example, class *DateTickUnit* has no classes that depend on it in version 0.9.3, but 2 classes (*DateAxis*. *HorizonDateAxis*) depend on it in version 0.9.4

6.4 Connected Unit

In a connected unit table, directly and indirectly coupled classes are located in the same column, thus a set of classes in the same column is a *connected unit*. Figure 6-6 shows part of connected units of *JFreechart* in two versions. From these connected units, we find that version 0.9.3 establishes a main connected unit which has 224 classes out of a total of 257 classes as shown in column A in Figure 6-6 (a), and a minor connected unit with 3 classes in column D of Figure 6-6(a). The three classes (*StandardToolTipsCollection*, *ToolTip*, and *ToolTipsCollection*) belong to the same package named "*com.jrefinery.chart.tooltips*". There are also 11 independent classes, e.g., *DatasetChangeListener* in column E, which have no relation to other classes in Figure 6-6(a). The independent classes are listed in Table 6-5.

We also find that version 0.9.4 has a main connected unit with 254 classes out of a total of 275 as shown column A in Figure 6-6 (b), and a minor connected unit with 3 classes in column K of Figure 6-6 (b). These three classes (*Function2D*, *LineFunction2D*, *PowerFunction2D*) belong to the same package named "*com.jrefinery.data* ". There are 18 independent classes which have no relation to other classes in Figure-6(b). The independent classes are listed in Table 6-5.

6.5 Comparing of Coupling Type

We compare the types of fan-in and fan-out couplings to see which type of the coupling is most affected by the evolution from version 0.9.3 to version 0.9.4. Seven types of couplings for these two versions are partially shown in Figure 6-7 and their actual metrics are shown in Table 6-6. Something very noticeable here is that 48% (1413 out of 3024 in version 0.9.3) and 53% (2192 out of 4111 in version 0.9.4) of the couplings are IM (*Invoked Method Type)* while none of them is RV (*Referenced Variable).*

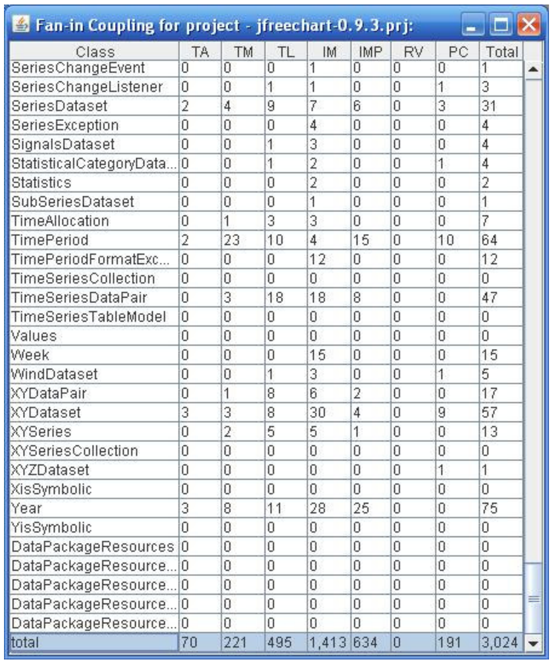
Table 6-5: Independent classes in two versions

0.9.3 (11 classes)	0.9.4 (18 classes)
JFreeChartInfo, PlotException, DatasetChangeListener, Values, XisSymbolic,YisSymbolic, DataPackageResources, DataPackageResources_de, DataPackageResources-es, DataPackageResources_fr, DataPackageResources_pl	DataUnit, JFreeChartInfo, PlotException, ChartChangeListener, LegendChangeListener, lotChangeListener, TitleChangeListener, JFreeChartResource, DatasetChangeListener, Regression, Values, XisSymbolic, YisSymbolic, DataPackageResources, DataPackageResources_de, DataPackageResources-es, DataPackageResources_fr, DataPackageResources_pl

Every coupling metric type has increased from version 0.9.3 to version 0.9.4, and the average increasing rate of the coupling is 35.95%. In particular, Method Invocation type (IM) has increased 55.48%, which is 72.13% (784 out of 1087) of the total number of the increased. This implies that the most significant difference is Method Invocation coupling between these two versions.

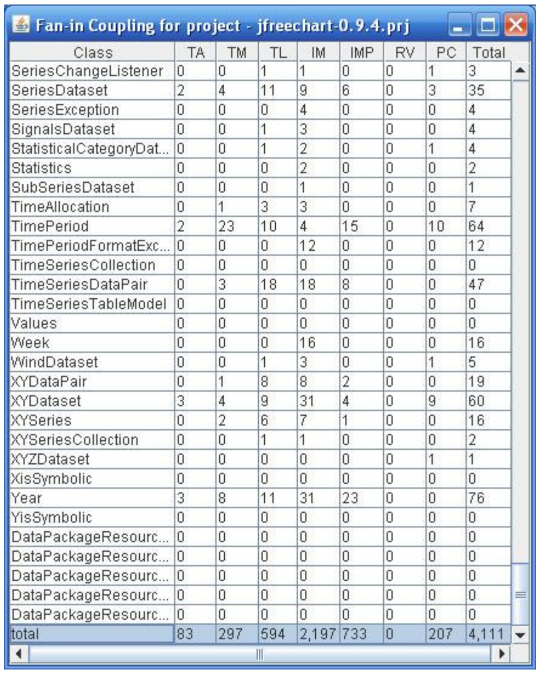
Table 6-6: Comparison of Fan-in and fan-out Coupling Types

Fan-in/out coupling	Version		Increased	Increasing rate
	0.9.3	0.9.4		
TA	70	83	13	18.57%
TM	221	297	76	34.39%
TL	495	594	99	20%
IM	1,413	2,197	784	55.48%
MP	634	733	99	15.62%
RV	0	0	0	0%
PC	191	207	16	8.38%
Total	3024	4,111	1,087	35.95%



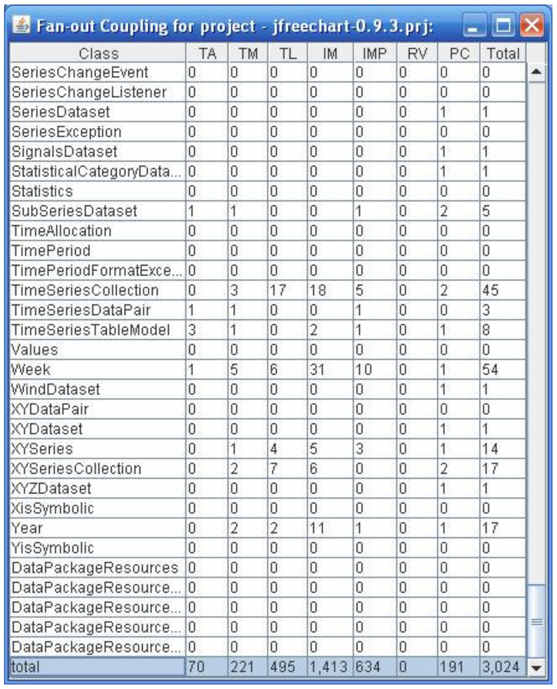
Class	TA	TM	TL	IM	IMP	RV	PC	Total
SeriesChangeEvent	0	0	0	1	0	0	0	1
SeriesChangeListener	0	0	1	1	0	0	1	3
SeriesDataset	2	4	9	7	6	0	3	31
SeriesException	0	0	0	4	0	0	0	4
SignalsDataset	0	0	1	3	0	0	0	4
StatisticalCategoryData...	0	0	1	2	0	0	1	4
Statistics	0	0	0	2	0	0	0	2
SubSeriesDataset	0	0	0	1	0	0	0	1
TimeAllocation	0	1	3	3	0	0	0	7
TimePeriod	2	23	10	4	15	0	10	64
TimePeriodFormatExc...	0	0	0	12	0	0	0	12
TimeSeriesCollection	0	0	0	0	0	0	0	0
TimeSeriesDataPair	0	3	18	18	8	0	0	47
TimeSeriesTableModel	0	0	0	0	0	0	0	0
Values	0	0	0	0	0	0	0	0
Week	0	0	0	15	0	0	0	15
WindDataset	0	0	1	3	0	0	1	5
XYDataPair	0	1	8	6	2	0	0	17
XYDataset	3	3	8	30	4	0	9	57
XYSeries	0	2	5	5	1	0	0	13
XYSeriesCollection	0	0	0	0	0	0	0	0
XYZDataset	0	0	0	0	0	0	1	1
XisSymbolic	0	0	0	0	0	0	0	0
Year	3	8	11	28	25	0	0	75
YisSymbolic	0	0	0	0	0	0	0	0
DataPackageResources	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
total	70	221	495	1,413	634	0	191	3,024

(a) Fan-in coupling in version 0.9.3



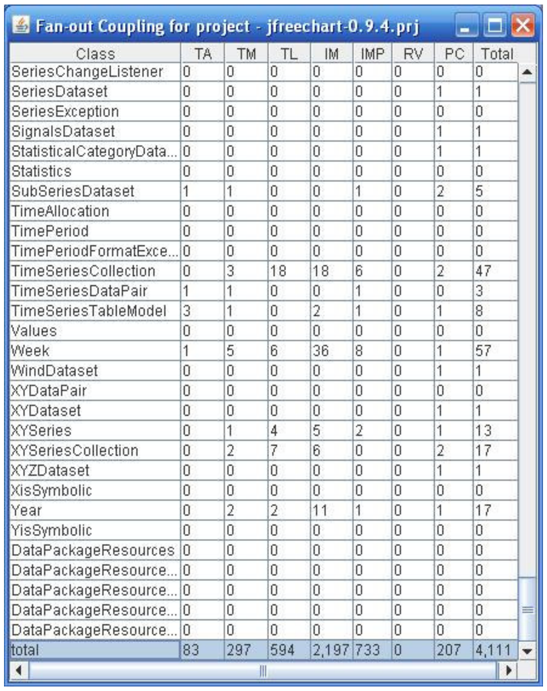
Class	TA	TM	TL	IM	IMP	RV	PC	Total
SeriesChangeListener	0	0	1	1	0	0	1	3
SeriesDataset	2	4	11	9	6	0	3	35
SeriesException	0	0	0	4	0	0	0	4
SignalsDataset	0	0	1	3	0	0	0	4
StatisticalCategoryDat...	0	0	1	2	0	0	1	4
Statistics	0	0	0	2	0	0	0	2
SubSeriesDataset	0	0	0	1	0	0	0	1
TimeAllocation	0	1	3	3	0	0	0	7
TimePeriod	2	23	10	4	15	0	10	64
TimePeriodFormatExc...	0	0	0	12	0	0	0	12
TimeSeriesCollection	0	0	0	0	0	0	0	0
TimeSeriesDataPair	0	3	18	18	8	0	0	47
TimeSeriesTableModel	0	0	0	0	0	0	0	0
Values	0	0	0	0	0	0	0	0
Week	0	0	0	16	0	0	0	16
WindDataset	0	0	1	3	0	0	1	5
XYDataPair	0	1	8	8	2	0	0	19
XYDataset	3	4	9	31	4	0	9	60
XYSeries	0	2	6	7	1	0	0	16
XYSeriesCollection	0	0	1	1	0	0	0	2
XYZDataset	0	0	0	0	0	0	1	1
XisSymbolic	0	0	0	0	0	0	0	0
Year	3	8	11	31	23	0	0	76
YisSymbolic	0	0	0	0	0	0	0	0
DataPackageResourc...	0	0	0	0	0	0	0	0
DataPackageResourc...	0	0	0	0	0	0	0	0
DataPackageResourc...	0	0	0	0	0	0	0	0
DataPackageResourc...	0	0	0	0	0	0	0	0
DataPackageResourc...	0	0	0	0	0	0	0	0
total	83	297	594	2,197	733	0	207	4,111

(a) Fan-in coupling in version 0.9.4



Class	TA	TM	TL	IM	IMP	RV	PC	Total
SeriesChangeEvent	0	0	0	0	0	0	0	0
SeriesChangeListener	0	0	0	0	0	0	0	0
SeriesDataset	0	0	0	0	0	0	1	1
SeriesException	0	0	0	0	0	0	0	0
SignalsDataset	0	0	0	0	0	0	1	1
StatisticalCategoryData...	0	0	0	0	0	0	1	1
Statistics	0	0	0	0	0	0	0	0
SubSeriesDataset	1	1	0	0	1	0	2	5
TimeAllocation	0	0	0	0	0	0	0	0
TimePeriod	0	0	0	0	0	0	0	0
TimePeriodFormatExce...	0	0	0	0	0	0	0	0
TimeSeriesCollection	0	3	17	18	5	0	2	45
TimeSeriesDataPair	1	1	0	0	1	0	0	3
TimeSeriesTableModel	3	1	0	2	1	0	1	8
Values	0	0	0	0	0	0	0	0
Week	1	5	6	31	10	0	1	54
WindDataset	0	0	0	0	0	0	1	1
XYDataPair	0	0	0	0	0	0	0	0
XYDataset	0	0	0	0	0	0	1	1
XYSeries	0	1	4	5	3	0	1	14
XYSeriesCollection	0	2	7	6	0	0	2	17
XYZDataset	0	0	0	0	0	0	1	1
XisSymbolic	0	0	0	0	0	0	0	0
Year	0	2	2	11	1	0	1	17
YisSymbolic	0	0	0	0	0	0	0	0
DataPackageResources	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
total	70	221	495	1,413	634	0	191	3,024

(c) Fan-out coupling in version 0.9.3



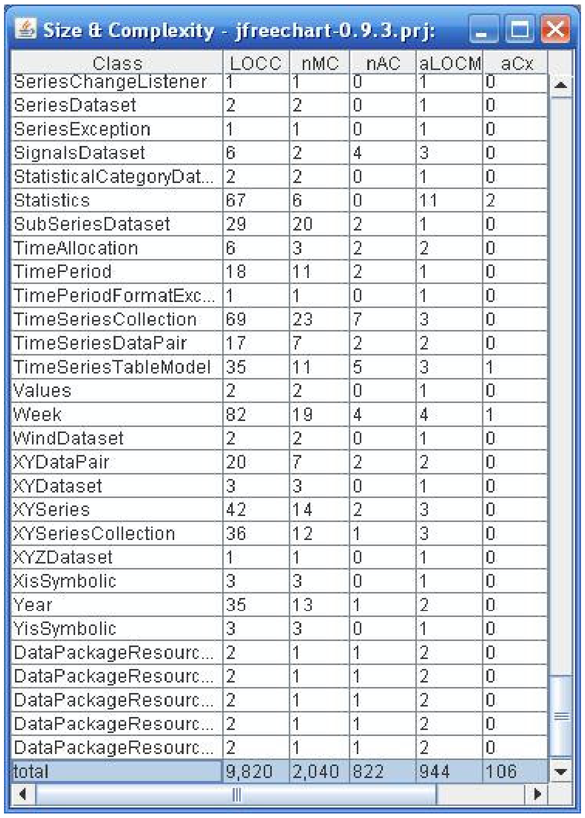
Class	TA	TM	TL	IM	IMP	RV	PC	Total
SeriesChangeListener	0	0	0	0	0	0	0	0
SeriesDataset	0	0	0	0	0	0	1	1
SeriesException	0	0	0	0	0	0	0	0
SignalsDataset	0	0	0	0	0	0	1	1
StatisticalCategoryData...	0	0	0	0	0	0	1	1
Statistics	0	0	0	0	0	0	0	0
SubSeriesDataset	1	1	0	0	1	0	2	5
TimeAllocation	0	0	0	0	0	0	0	0
TimePeriod	0	0	0	0	0	0	0	0
TimePeriodFormatExce...	0	0	0	0	0	0	0	0
TimeSeriesCollection	0	3	18	18	6	0	2	47
TimeSeriesDataPair	1	1	0	0	1	0	0	3
TimeSeriesTableModel	3	1	0	2	1	0	1	8
Values	0	0	0	0	0	0	0	0
Week	1	5	6	36	8	0	1	57
WindDataset	0	0	0	0	0	0	1	1
XYDataPair	0	0	0	0	0	0	0	0
XYDataset	0	0	0	0	0	0	1	1
XYSeries	0	1	4	5	2	0	1	13
XYSeriesCollection	0	2	7	6	0	0	2	17
XYZDataset	0	0	0	0	0	0	1	1
XisSymbolic	0	0	0	0	0	0	0	0
Year	0	2	2	11	1	0	1	17
YisSymbolic	0	0	0	0	0	0	0	0
DataPackageResources	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
DataPackageResource...	0	0	0	0	0	0	0	0
total	83	297	594	2,197	733	0	207	4,111

(d) Fan-out coupling in version 0.9.4

Figure 6-7: Fan-out/Fan-out coupling

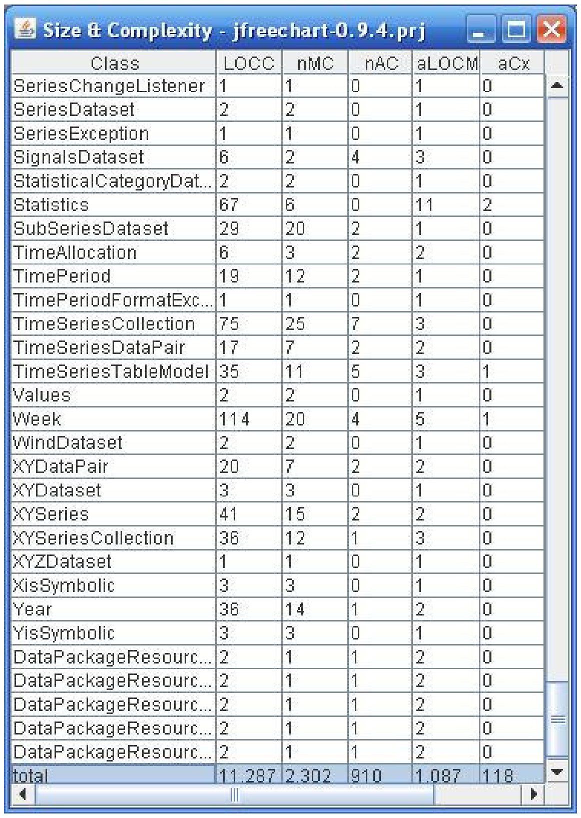
6.6 Size and Complexity

Four size and one complexity metrics for these two versions are partially shown in Figure 6-8 and their differences are in Table 6-7. The metrics have all increased from version 0.9.3 to version 0.9.4. Figure 6-9 graphs the LOCC distributions these two versions, and it is clear that most classes have fewer than 50 lines.



Class	LOCC	nMC	nAC	aLOCM	aCx
SeriesChangeListener	1	1	0	1	0
SeriesDataset	2	2	0	1	0
SeriesException	1	1	0	1	0
SignalsDataset	6	2	4	3	0
StatisticalCategoryDat...	2	2	0	1	0
Statistics	67	6	0	11	2
SubSeriesDataset	29	20	2	1	0
TimeAllocation	6	3	2	2	0
TimePeriod	18	11	2	1	0
TimePeriodFormatExc...	1	1	0	1	0
TimeSeriesCollection	69	23	7	3	0
TimeSeriesDataPair	17	7	2	2	0
TimeSeriesTableModel	35	11	5	3	1
Values	2	2	0	1	0
Week	82	19	4	4	1
WindDataset	2	2	0	1	0
XYDataPair	20	7	2	2	0
XYDataset	3	3	0	1	0
XYSeries	42	14	2	3	0
XYSeriesCollection	36	12	1	3	0
XYZDataset	1	1	0	1	0
XisSymbolic	3	3	0	1	0
Year	35	13	1	2	0
YisSymbolic	3	3	0	1	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
total	9,820	2,040	822	944	106

(a) Size & complexity in version 0.9.3



Class	LOCC	nMC	nAC	aLOCM	aCx
SeriesChangeListener	1	1	0	1	0
SeriesDataset	2	2	0	1	0
SeriesException	1	1	0	1	0
SignalsDataset	6	2	4	3	0
StatisticalCategoryDat...	2	2	0	1	0
Statistics	67	6	0	11	2
SubSeriesDataset	29	20	2	1	0
TimeAllocation	6	3	2	2	0
TimePeriod	19	12	2	1	0
TimePeriodFormatExc...	1	1	0	1	0
TimeSeriesCollection	75	25	7	3	0
TimeSeriesDataPair	17	7	2	2	0
TimeSeriesTableModel	35	11	5	3	1
Values	2	2	0	1	0
Week	114	20	4	5	1
WindDataset	2	2	0	1	0
XYDataPair	20	7	2	2	0
XYDataset	3	3	0	1	0
XYSeries	41	15	2	2	0
XYSeriesCollection	36	12	1	3	0
XYZDataset	1	1	0	1	0
XisSymbolic	3	3	0	1	0
Year	36	14	1	2	0
YisSymbolic	3	3	0	1	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
DataPackageResourc...	2	1	1	2	0
total	11,287	2,302	910	1,087	118

(a) Size & complexity in version 0.9.4

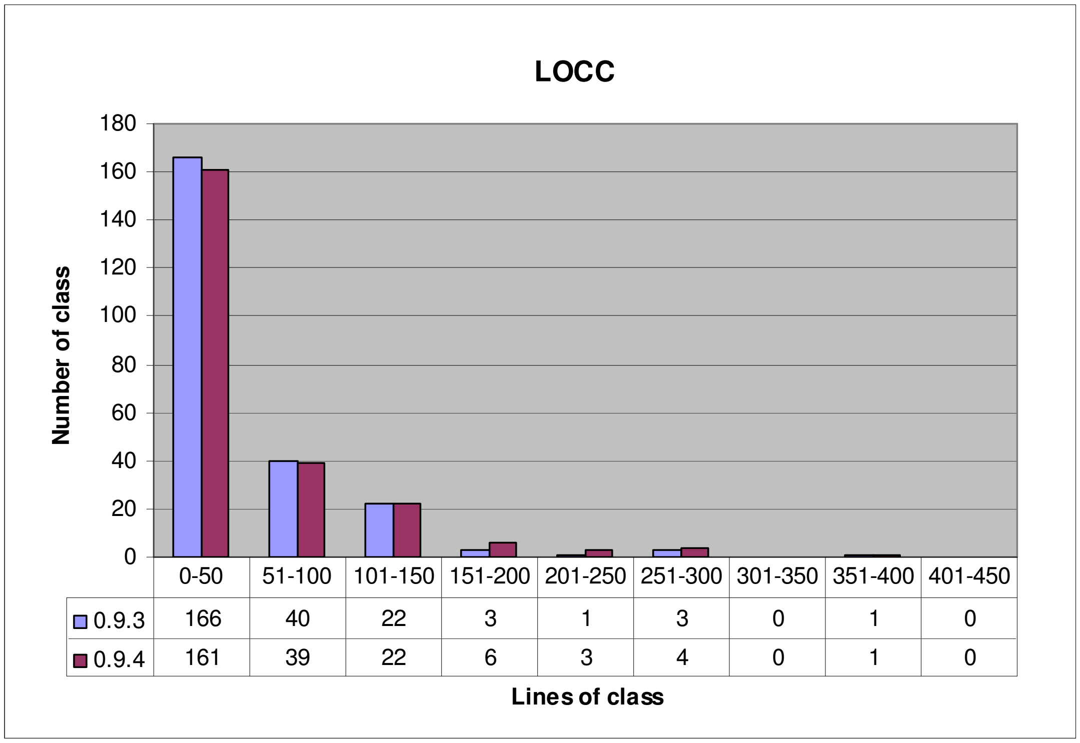
Figure 6-8: Size & complexity

Table 6-7: Size & complexity differences

Size & complexity	Version		Increased	Increasing rate
	0.9.3	0.9.4		
LOCC	9,820	11,287	1,467	14.94%
nMC	2,040	2,302	262	12.84%
nAC	822	910	88	10.71%
aLOCM	944	1,087	143	15.15%
aCX	106	118	12	11.32%

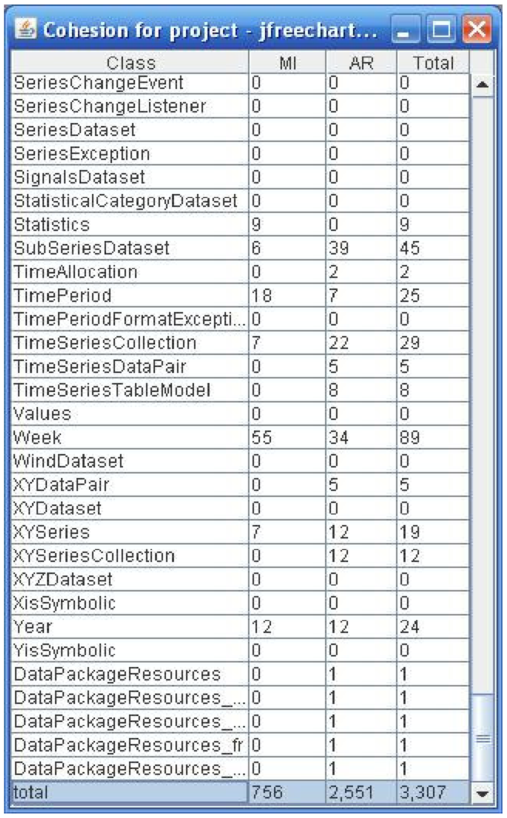
Table 6-8: Cohesion differences

Cohesion	Version		Increased	Increasing rate
	0.9.3	0.9.4		
MI	756	1,028	272	35.98%
AR	2,551	2,867	316	12.39%
Total	3,307	3,895	588	17.78%



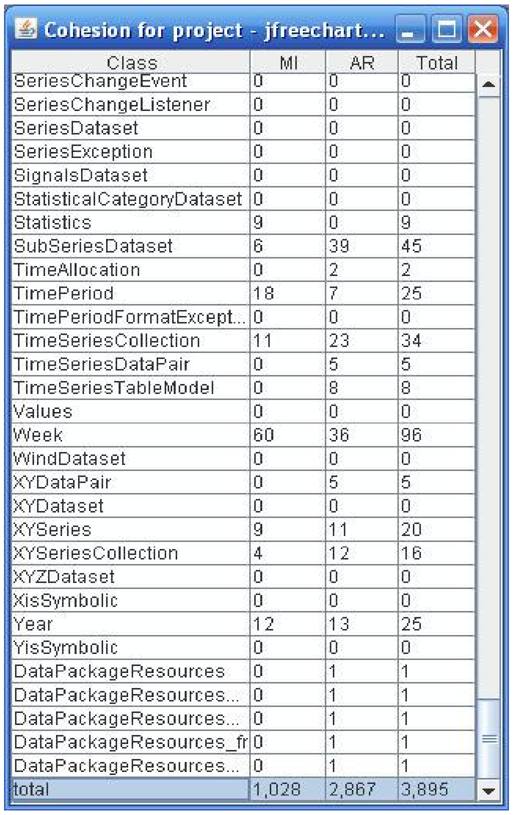
	0-50	51-100	101-150	151-200	201-250	251-300	301-350	351-400	401-450
0.9.3	166	40	22	3	1	3	0	1	0
0.9.4	161	39	22	6	3	4	0	1	0

Figure 6-9: LOCC distribution


Cohesion for project - jfreechart...

Class	MI	AR	Total
SeriesChangeEvent	0	0	0
SeriesChangeListener	0	0	0
SeriesDataset	0	0	0
SeriesException	0	0	0
SignalsDataset	0	0	0
StatisticalCategoryDataset	0	0	0
Statistics	9	0	9
SubSeriesDataset	6	39	45
TimeAllocation	0	2	2
TimePeriod	18	7	25
TimePeriodFormatExcepti...	0	0	0
TimeSeriesCollection	7	22	29
TimeSeriesDataPair	0	5	5
TimeSeriesTableModel	0	8	8
Values	0	0	0
Week	55	34	89
WindDataset	0	0	0
XYDataPair	0	5	5
XYDataset	0	0	0
XYSeries	7	12	19
XYSeriesCollection	0	12	12
XYZDataset	0	0	0
XisSymbolic	0	0	0
Year	12	12	24
YisSymbolic	0	0	0
DataPackageResources	0	1	1
DataPackageResources_...	0	1	1
DataPackageResources_...	0	1	1
DataPackageResources_fr	0	1	1
DataPackageResources_...	0	1	1
total	756	2,551	3,307

(a) Cohesion in version 0.9.3


Cohesion for project - jfreechart...

Class	MI	AR	Total
SeriesChangeEvent	0	0	0
SeriesChangeListener	0	0	0
SeriesDataset	0	0	0
SeriesException	0	0	0
SignalsDataset	0	0	0
StatisticalCategoryDataset	0	0	0
Statistics	9	0	9
SubSeriesDataset	6	39	45
TimeAllocation	0	2	2
TimePeriod	18	7	25
TimePeriodFormatExcept...	0	0	0
TimeSeriesCollection	11	23	34
TimeSeriesDataPair	0	5	5
TimeSeriesTableModel	0	8	8
Values	0	0	0
Week	60	36	96
WindDataset	0	0	0
XYDataPair	0	5	5
XYDataset	0	0	0
XYSeries	9	11	20
XYSeriesCollection	4	12	16
XYZDataset	0	0	0
XisSymbolic	0	0	0
Year	12	13	25
YisSymbolic	0	0	0
DataPackageResources	0	1	1
DataPackageResources...	0	1	1
DataPackageResources...	0	1	1
DataPackageResources_fr	0	1	1
DataPackageResources...	0	1	1
total	1,028	2,867	3,895

(a) Cohesion in version 0.9.4

Figure 6-10: Cohesion

6.7 Cohesion

Cohesion metrics for the two studied versions are partially shown in Figure 6-10 and their differences are in Table 6-8. Something noticeable is that 77% (2,551 out of 3,307 in version 0.9.3) and 74% (2,867 out of 3,895 in version 0.9.4) of the cohesion are AR (*Attribute Reference)* while MI (*Method Invocation*) has increased 35.98% in version 0.9.4.

6.8 Summary

The goal of this case study is to compare and analyze two versions of *JFreeChart* at class level. Specifically, it aims to answer the following questions:

- How does the architecture of *JFreeChart* change between two consecutive versions?
- How can the differences between them be compared and detected?
- How can the huge information of source code be filtered and compared?

In this case study, we analyzed the differences between the metrics of two versions using *JamTool* and found overall trend of metrics of *JFreeChart* in versions 0.9.3 and 0.9.4.

From the comparison and analysis of two versions of *JFreeChart,* we summarize the following findings:

- 21 classes were added to version 0.9.4
- 3 classes were removed from version 0.9.3
- 44 classes have new fan-out couplings and 60 classes have new fan-in couplings in version 0.9.4.
- Most classes have low fan-in or fan-out couplings but few classes have high coupling.
- By comparing reusable units and maintainable units in version 0.9.3 and version 0.9.4, we found newly added classes to the reusable unit and maintainable unit.

- By analyzing connected unit, we found that most classes are directly or indirectly related to each other and they form one main connected unit. But we also found minor connected units with 3 classes, and 11 and 18 independent classes which have no relations to other classes in versions 0.9.3 and 0.9.4, respectively.
- More than half of the newly added couplings were Method invocation.
- Size and complexity metrics are also increased in 0.9.4.

Based on the findings above, we conclude that the metrics tables produced by *JamTool* can be used in the following tasks:

- To monitor new coupling through evolution of the software system.
- To identify outlier classes based on the metrics

7 IDENTIFYING CORRELATION AMONG METRICS

This chapter presents empirical studies that investigate if the metrics defined and implemented in *JamTool* are related to each other. The data sets used for the study are also presented. Finally, the statistical correlation coefficients are described.

Statistical analyses were performed to investigate the following questions:

- Are there correlations in the metrics?

The test programs used in this research are Java classes in the GUI library (i.e., Swing in JFC) and GUI applications (i.e., Bingo and Netbean). Metrics for these classes were automatically collected using *JamTool*. These applications were written by developers in the Sun Microsystems. The test programs used in this experiment are grouped as follows:

SwingLib = {classes in Swing package in JFC},

BingoAppl = {classes in Bingo application},

NetbeanAppl = {classes in Netbean application},

SwingLib contains 502 classes; *BingoAppl* has 48 classes; NetbeanAppl has 52 classes.

<u>JFC/Swing</u>

The Java Foundation Classes (JFC) is a comprehensive set of GUI components and services which simplify the development and deployment of desktop and

Internet/Intranet applications. JFC extends the original Abstract Window Toolkit (AWT) by adding a comprehensive set of graphical user interface class libraries.

These components are written in Java, without window-system-specific code. They facilitate a customizable look-and-feel without relying on the native windowing system, and simplify the deployment of applications.

Swing is a GUI component kit and is part of JFC integrated into Java 2 platform-Standard Edition (J2SE). Swing simplifies deployment of applications by providing a complete set of user-interface elements written entirely in Java. Swing components also permits a customizable look and feel without relying on any window specific components. We shall demonstrate our approach by considering code using the JFC/Swing library.

Bingo

Bingo is a client/server application that implements the game of BINGO and a comprehensive example of JFC provided by the Sun Microsystems. This application broadcasts information via a multicast socket, builds its GUI with Swing components, uses multiple synchronous threads, and communicates with RMI.

NetBean

The NetBean IDE is a development environment - a tool for programmers to write, compile, debug and deploy programs. It is a development tool written in Java for writing programs in Java and other programming languages.

7.1 Methodology

7.1.1 Experiment 1: Correlation Coefficients among the metrics

The goal of this statistical analysis is to answer the question:

- Are any of the metrics in a group (i.e., *SwingLib, BingoAppl*, and *NetbeanAppl*) correlated?

The Pearson product moment correlation coefficient, r, is a dimensionless index that ranges from –1.0 to 1.0 inclusive and reflects the extent of a linear relationship between two data sets. For example, if the r value associated with Metric1 and Metric2 is close to zero, then the metric values of Metric1 and Metric2 are not linearly related. On the other hand, if r is close to 1, then large values of Metric1 are associated with large values of Metric2. Finally, if r is close to –1, then large values of Metric1 are linearly associated with small values of Metric2. The sign of the correlation coefficient indicates whether two variables are positively or inversely related. A negative value means that as Metric1 becomes larger, Metric2 tends to become smaller. A positive correlation means that both Metric1 and Metric2 go in the same direction.

7.1.2 Experiment 2: Correlation Coefficients among the coupling metrics in a group

The goal of this statistical analysis is to answer the question:

- Are any of the coupling metrics in a group (i.e., *SwingLib, BingoAppl*, and *NetbeanAppl*) correlated?

In this experiment, we analyze the correlation among the fan-in and fan-out coupling metrics from *SwingLib, BingoAppl,* and *NetbeanAppl* to find internal features of each system.

In this chapter, we apply metrics defined in Section 3.3. For example, LOC measures number of lines in a class.

7.2 Results

Sections 7.2.1 and 7.2.2 provide the results for each of the statistical analyses described in section 7.1. First, section 7.2.1 discusses the correlation among the metrics in a group. Section 7.2.2 discusses the analysis results for the correlation among coupling metrics in a group.

7.2.1 Result 1: Correlation among the metrics in a group

This section answers the following question:

- Are any of the metrics in *SwingLib, NetBeanAppl,* and *BingoAppl* correlated?

Tables 7-1, 7-2, and 7-3 show the correlation coefficients for the metrics and Table 7-4 shows metrics pairs with *r* values greater than 0.6. In the traditional procedural programming paradigm, studies show that defects correlate with *LOC* and Cyclomatic complexity [49, 50].

From the correlation results of *SwingLib* and *NetbeanAppl* (See Table 7-1 and 7-2), we found common correlation patterns.

Except for *aLOCM*, size metrics (i.e., *LOC*, *NOM*, *NOA*), complexity metrics (i.e., *aCx*), and cohesion metrics (i.e., *cohMI*, *cohAR*) are positively correlated to each other. Coupling metrics are positively correlated to each other except *cplPC* and *cplMI.*

LOC, NOM, and *NOA* are representatives of the size of a class; however a*LOCM* represents the averaged method size in a class.

Size, complexity and cohesion metrics are correlated to each other. But coupling metrics are not correlated to other metrics like size, complexity and cohesion. Size, complexity and cohesion metrics represent the volume within a class, but coupling metrics represent the structure among classes in a system. These two aspects of software system, obviously, are not correlated.

From the correlation results of *BingoAppl* (See Table 7-3), Size metrics and complexity metrics are correlated to each other but cohesion metrics are independent from other metrics. Some coupling metrics (*cplTA*, *cplMI*, and *cplTPM*) are correlated to size and complexity metrics. *aLOCM* and *cplPC* are independent from other metrics.

Table 7-1: Correlation Coefficients of metrics in SwingLib

	LOC	*NOM*	*NOA*	*aLOCM*	*aCx*	*cohMI*	*cohAR*	*cplTA*	*cplTP*	*cplTL*	*cplMI*	*cplTPM*	*cplPC*
LOC	1.00												
NOM	0.94	1.00											
NOA	0.70	0.66	1.00										
aLOCM	0.31	0.12	0.15	1.00									
aCx	0.95	0.87	0.64	0.23	1.00								
cohMI	0.85	0.81	0.72	0.15	0.83	1.00							
cohAR	0.82	0.77	0.73	0.17	0.79	0.76	1.00						
cplTA	0.34	0.33	0.27	0.07	0.33	0.25	0.27	1.00					
cplTP	0.35	0.39	0.20	0.04	0.28	0.29	0.20	0.50	1.00				
cplTL	0.38	0.33	0.24	0.08	0.42	0.31	0.26	0.50	0.52	1.00			
cplMI	0.52	0.47	0.34	0.16	0.51	0.42	0.47	0.52	0.50	0.89	1.00		
cplTPM	0.40	0.33	0.33	0.07	0.45	0.38	0.34	0.43	0.44	0.85	0.77	1.00	
cplPC	0.15	0.15	0.17	0.07	0.17	0.15	0.13	0.18	0.21	0.24	0.20	0.20	1.00

Table 7-2: Correlation Coefficients of metrics in NetbeanAppl

	LOC	*NOM*	*NOA*	*aLOCM*	*aCx*	*cohMI*	*cohAR*	*cplTA*	*cplTP*	*cplTL*	*cplMI*	*cplTPM*	*cplPC*
LOC	1.00												
NOM	0.95	1.00											
NOA	0.67	0.61	1.00										
aLOCM	0.28	0.05	0.20	1.00									
aCx	0.91	0.86	0.49	0.29	1.00								
cohMI	0.84	0.89	0.49	0.09	0.86	1.00							
cohAR	0.81	0.79	0.81	0.16	0.69	0.65	1.00						
cplTA	-0.11	-0.05	-0.14	-0.27	-0.08	-0.06	-0.11	1.00					
cplTP	-0.03	0.05	-0.19	-0.31	0.02	0.08	-0.12	0.90	1.00				
cplTL	0.06	0.12	-0.15	-0.17	0.12	0.16	-0.02	0.87	0.88	1.00			
cplMI	0.32	0.32	0.16	-0.02	0.36	0.35	0.20	0.19	0.25	0.28	1.00		
cplTPM	0.06	0.14	0.07	-0.25	0.03	0.19	0.06	0.86	0.85	0.83	0.21	1.00	
cplPC	0.35	0.40	0.06	-0.02	0.47	0.39	0.37	0.17	0.22	0.33	0.18	0.17	1.00

Table 7-3: Correlation Coefficients of metrics in BingoAppl

	LOC	*NOM*	*NOA*	*aLOCM*	*aCx*	*cohMI*	*cohAR*	*cplTA*	*cplTP*	*cplTL*	*cplMI*	*cplTPM*	*cplPC*
LOC	1.00												
NOM	0.84	1.00											
NOA	0.82	0.64	1.00										
aLOCM	0.46	0.12	0.32	1.00									
aCx	0.88	0.68	0.58	0.34	1.00								
cohMI	-0.06	-0.10	-0.16	0.13	-0.03	1.00							
cohAR	0.03	-0.04	-0.08	0.20	0.04	0.92	1.00						
cplTA	0.87	0.73	0.67	0.25	0.80	-0.11	-0.05	1.00					
cplTP	0.52	0.53	0.36	0.09	0.51	-0.12	-0.11	0.71	1.00				
cplTL	0.12	0.06	0.09	0.14	0.11	0.04	-0.04	0.14	0.37	1.00			
cplMI	0.78	0.60	0.74	0.40	0.61	0.15	0.21	0.75	0.52	0.04	1.00		
cplTPM	0.75	0.42	0.87	0.37	0.57	-0.08	-0.02	0.71	0.39	0.06	0.84	1.00	
cplPC	-0.22	-0.06	-0.16	-0.42	-0.19	-0.16	-0.14	-0.19	-0.22	-0.30	-0.21	-0.21	1.00

Table 7-4: Pairs in SwingLib, NetbeanAppl, and BingoAppl with r-value > 0.6

Pair	*r value*			Pair	*r value*		
	Swing Lib	*Netbean Appl*	*Bingo Appl*		*Swing Lib*	*Netbean Appl*	*Bingo Appl*
LOC and NOM	0.94	0.94	0.84	NOA and cplTA			0.67
LOC and NOA	0.70	0.70	0.82	NOA and cplMI			0.74
LOC and aCx	0.95	0.95	0.88	NOA and cplTPM			0.87
LOC and cohMI	0.85	0.85		aCx and cohMI	0.83	0.83	
LOC and cplTA			0.87	aCx and cplTA			0.80
LOC and cohAR	0.82	0.82		aCx and cplMI			0.61
LOC and cplMI			0.78	aCx and cohAR	0.79	0.79	
LOC and cplTPM			0.75	cohMI and	0.76	0.76	
NOM and NOA	0.66	0.66		cplTL and cplMI	0.89		
NOM and aCx	0.87	0.87		cplTA and cplTP		0.89	0.71
NOM and NOA			0.64	cplTA and cplMI			0.75
NOM and aCx			0.68	cplTA and cplTL		0.86	
NOM and cplTA			0.73	cplTA and cplTPM		0.95	0.71
NOM and cohMI	0.81	0.81		cplMI and cplTPM			0.84
NOM and cohAR	0.77	0.77		cplTP and cplTL		0.88	
NOA and aCx	0.64			cplTP and cplTPM		0.85	
NOA and cohMI	0.72			cplTL and cplTPM	0.86	0.77	
NOA and cohAR	0.73	0.73		cplMI and cplTPM	0.77		

7.2.2 Result 2: Correlation among the coupling metrics in a group

We have found the followings from the previous experiments:

- Size, complexity and cohesion metrics are correlated to each other with some exceptions.
- Coupling metrics are relatively independent from other metrics (i.e., size, complexity, and cohesion)
- Some coupling metrics are correlated to each other.

In this experiment, we also measure fan-in and fan-out coupling metrics for each software system and analyze the measurement results. We collect and analyze the measurement results from *SwingLib, NetbeanApp, and BingoAppl.* In this section, we add *in* and *out* to the end of the metrics name to indicate fan-in and fan-out coupling instead prefix *cpl*. For example, *TAin* represents fan-in coupling with *cplTA* type.

In Tables 7-5, 7-6 and 7-7*,* some fan-in coupling metrics are positively correlated to each other and some fan-out coupling metrics are positively correlated to each other as well. However, fan-in coupling metrics are **not** correlated to fan-out coupling metrics.

The following are our interpretation of the measurement results in this experiment.

- All fan-in coupling metrics are correlated with each other and all fan-out coupling metrics are correlated with each other except *PCin*, *PCout*, and *TMout*.
    - There are two types of classes in *SwingLib*: fan-in coupled classes and fan-out coupled classes. Fan-in coupled classes are used by (i.e.,

export) other classes, but do not use (i.e., import) other classes. Fan-out coupled classes use other classes, but are not used by other classes.

  - Fan-in coupled classes in *SwingLib* are used by other classes with diverse connection types.
  - Fan-out coupled classes in *SwingLib* use other classes with diverse connection types.
  - Classes in *SwingLib* are designed with a specific role – import or export.

- In *BingoAppl* and *NetbeanAppl* not all fan-in coupling metrics are correlated to each other and not all fan-out coupling metrics are correlated to each other either.
- There is no correlation between fan-in and fan-out coupling metrics in *SwingLib*, *NetbeanAppl,* and *BingoAppl.*

Table 7-5: Correlation of coupling metrics in *SwingLib*

	TAin	*TMin*	*TLin*	*IMin*	*IMPin*	*PCin*	*TAout*	*TMout*	*TLout*	*IMout*	*IMPout*	*PCout*
TAin	1.00											
TMin	0.72	1.00										
TLin	0.84	0.72	1.00									
IMin	0.76	0.59	0.89	1.00								
IMPin	0.78	0.60	0.75	0.68	1.00							
PCin	0.08	0.38	0.12	0.08	0.09	1.00						
TAout	0.12	0.06	0.13	0.18	0.09	0.08	1.00					
TMout	0.08	0.05	0.13	0.27	0.07	0.05	0.57	1.00				
TLout	0.02	0.00	0.05	0.09	0.03	0.08	0.58	0.52	1.00			
IMout	0.01	-0.02	0.03	0.09	0.02	0.08	0.60	0.50	0.89	1.00		
IMPout	0.01	-0.02	0.03	0.05	0.08	0.02	0.49	0.44	0.85	0.77	1.00	
PCout	0.21	0.16	0.25	0.22	0.26	-0.02	0.19	0.21	0.24	0.20	0.20	1.00

Table 7-6: Correlation of coupling metrics in *NetbeanAppl*

	TAin	*TMin*	*TLin*	*IMin*	*IMPin*	*PCin*	*TAout*	*TMout*	*TLout*	*IMout*	*IMPout*	*PCout*
TAin	1.00											
TMin	0.97	1.00										
TLin	0.96	0.96	1.00									
IMin	0.28	0.28	0.28	1.00								
IMPin	0.92	0.89	0.88	0.23	1.00							
PCin	0.28	0.29	0.30	0.07	0.23	1.00						
TAout	-0.10	-0.12	-0.05	-0.09	-0.04	0.00	1.00					
TMout	-0.08	-0.09	-0.12	-0.05	-0.09	-0.01	0.36	1.00				
TLout	-0.07	-0.08	-0.10	-0.03	-0.11	0.33	0.24	0.51	1.00			
IMout	-0.11	-0.10	-0.15	0.01	-0.11	0.30	0.05	0.43	0.61	1.00		
IMPout	-0.10	-0.11	-0.15	-0.07	-0.12	0.04	0.24	0.59	0.72	0.48	1.00	
PCout	-0.07	-0.09	-0.10	-0.09	-0.08	-0.03	0.00	0.23	0.13	0.35	0.12	1.00

Table 7-7: Correlation of coupling metrics in *BingoAppl*

	TAin	*TMin*	*TLin*	*IMin*	*IMPin*	*PCin*	*TAout*	*TMout*	*TLout*	*IMout*	*IMPout*	*PCout*
TAin	1.00											
TMin	0.67	1.00										
TLin	-0.11	0.29	1.00									
IMin	0.57	0.47	0.03	1.00								
IMPin	0.33	0.53	0.39	0.19	1.00							
PCin	-0.23	-0.16	-0.14	-0.32	0.00	1.00						
TAout	0.46	0.49	-0.14	0.34	0.49	-0.22	1.00					
TMout	0.11	0.06	-0.16	0.11	-0.19	-0.18	0.33	1.00				
TLout	0.24	0.13	-0.13	-0.09	-0.02	-0.17	0.34	0.51	1.00			
IMout	0.36	0.36	-0.15	0.25	0.37	-0.18	0.90	0.43	0.57	1.00		
IMPout	0.33	0.14	-0.16	0.29	0.20	-0.19	0.61	0.26	0.37	0.64	1.00	
PCout	0.16	-0.01	-0.24	0.14	-0.23	-0.16	0.08	0.04	0.03	0.11	0.04	1.00

8 CONCLUSIONS

The primary objective of this research is to provide an automated measurement tool (i.e., *JamTool*) to guide a programmer for software reuse and maintenance. Measuring how well software components can be reused and maintained helps programmers not only write reusable and maintainable software, but also identifies reusable or fault-prone components.

The following research contributions have been achieved in this study.

Quality Measurement Model Development

We developed a quality model that leads to a metric set implemented in *JamTool*. We first identified essential software properties that have been suggested as having an impact on software reusability and maintainability. Then we divided these quality factors into five subfactors (i.e., identification, separation, modification, validation, and adaptation) in a top-down fashion. We also applied bottom-up approach to develop quality measurement models for reusability and maintainability based on available measurement types that are related to reuse and maintenance properties. Using these top-down and bottom-up approaches, we constructed a concise quality measurement model for reusability and maintainability.

Automated Measurement Tool

An automated measurement tool, *JamTool*, for object-oriented software system was developed in this work. This research describes how this tool can guide a programmer through measuring internal characteristics of a program for software reuse and maintenance.

In this work, primitive but comprehensive metrics for object-oriented language have been extensively studied and statistically analyzed to show internal characteristics from the classes selected from various applications. The automatically identified connected units, reusable units, and maintainable units have been discussed.

JamTool's capabilities have been demonstrated through case studies.

1. Measuring Quality on Software Evolution: It shows that the metrics defined and implemented by *JamTool* can be used to assess the quality on the evolution of a software system.
2. Visualizing Software Evolution: The evolution track-table visualizes the evolution of a software system.
3. Analyzing Software for Reuse and Maintenance: It shows how the architecture of a software system changes between two consecutive versions. It also shows the usage of connect unit, reusable unit, and maintainable unit.
4. Identifying Correlation among Metrics: It shows the correlation among the metrics defined and implemented by *JamTool*.

The first case study investigated whether *JamTool* can be used to assess the reusability of an open software system, *JFreeChart,* over its evolution with *fan-in* and *fan-out* couplings for *added* and *removed* classes. We found that the number of classes increases gradually over most releases, and they have positive improvement with respect to the coupling metrics but not positively related to the cohesion. It has also been found that evolution of this software system is consistent with Lehman's 1*st*, 2*nd*, and 6*th* laws of software evolution. We found that the *added* classes have higher *fan-in* coupling and lower *fan-out* coupling comparing to the *removed* classes, which is desirable in term of reusability. This observation leads us to believe that the reusability of *JFreeChart* has improved along with its evolution and reject Lehman's 7*th* laws of software evolution. In this way, applying metrics from *JamTool* over the evolution of software can aid a software engineer to understand how a system has evolved over time.

The second case study investigated whether *JamTool* can be used to capture the difference between two consecutive versions on the evolution of *JFreeChart*. Based on the findings in this case study, we conclude that the metrics tables produced by *JamTool* can be used in the following tasks:

- To monitor the new coupling through evolution of the software system.
- To identify outlier classes based on the metrics

The third case study investigated whether the metrics defined and implemented in *JamTool* are related to each other. We have found the followings from this case study:

- Size, complexity and cohesion metrics are correlated to each other with some exceptions.
- Coupling metrics are relatively independent from other metrics (i.e., size, complexity, and cohesion)
- All fan-in coupling metrics are correlated with each other and all fan-out coupling metrics are correlated with each other except *PCin*, *PCout*, and

TMout.

- There is no correlation between fan-in and fan-out coupling metrics.

Consequently, having achieved our goal of providing an automated source code measurement environment, we demonstrated that our tool, *JamTool*, is a valuable tool to help software engineers understand and explore software reuse and maintenance.

There are several aspects to the work presented in this dissertation that offer potential for future research. Some of these areas are listed below.

1. Object-oriented metrics and connected units can be used to automate the recognition of design patterns in existing software components. A specific area for future research is to characterize the structure of design patterns and use design metrics and clusters to recognize pattern structures in existing object-oriented software libraries and systems.
2. To analyze features of application domains: After the analysis of the measurement results of various application domains, common features of each domain may be derived.

BIBLIOGRAPHY

[1] Alshyeb, M., Li, W., “An empirical study of system design instability metric and design evolution in an agile software process”, *The Journal of Systems and Software* 74, 2005, pp 269-274

[2] Basili, V.R., and Weiss, D.M. “A Methodology for Collecting Valid Software Engineering Data,” *IEEE Trans. on Software Eng*., 10(6), pp. 728-738, 1984.

[3] Berard, E.V., "Essays on object-oriented Software Engineering," *Prentice Hall*, Englewood Cliffs, NJ, 1992, 392 pp

[4] Bieman, J.M., and Kang, B-K. “Cohesion and Reuse in an object-oriented System,” *Proc. ACM Symposium on Software Reusability (SSR’95),* pp 259-262, Apr. 1995.

[5] Boehm, B.W. et al., "Characteristic of Software Quality," *TRW Series of Software Technology*, Amsterdam, North Holland, 1978.

[6] Booch, G. “Object Oriented Design with Applications,” Benjamin/Cummings, Menlo Park, CA, 1991, 580 pp

[7] Briand, L., Morasca, S., and Basili, V. “Assessing Software Maintainability at the end of high-level design,” *Proc. IEEE Conf. on Software Maintenance (CSM’93),* Sep. 1993.

[8] Briand, L., Morasca, S., and Basili, V. “Defining and Validating High-Level Design Metrics,” Technical Report, *University of Maryland*, CS-TR 3301, 1994.

[9] Briand, L., Daly, J., and Wust, J., “A Unified Framework for Cohesion Measurement in object-oriented Systems,” *Empirical Software Eng*.: An Int’l J., vol. 3, no. 1, pp. 65-117, 1998.

[10] Briand, L., Daly, J., and Wust, J., “A Unified Framework for Coupling Measurement in object-oriented Systems,” *IEEE Trans. on software eng*., vol. 25, no. 1, 1999.

[11] Card, D.N., Church, V.E., and Agresti, W.W., “An Empirical Study of Software Design Practices,” *IEEE Transactions on Software Engineering* 12(2), 264-271, 1986.

[12] Chidamber, S.R, and Kemerer, C.F., "A Metrics Suite for object-oriented Design," *IEEE Trans on Software Eng.*, vol. 20, no. 6, pp. 476-493, Jun. 1994.

[13] David, G., and Scott, L. “The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications,” *International Conference on Software Maintenance*, 1996, pp 134-141

[14] Dromey, G.R. "A Model for Software Product Quality," *IEEE Trans. on Software Eng.*, vol.21, no.2, pp. 146-162, 1995.

[15] Eder, J., Kappel, G, and Schrefl, M. “Coupling and Cohesion in object-oriented Systems,” Technical Report, *University of Klagenfurt*, 1994.

[16] Fenton, N.E., Iizuka, Y., and Whitty, R.W. “Software Quality Assurance and Measurement: A Worldwide Perspective,” *ITP*, London, 1995.

[17] Fenton, N.E., and Pfleeger, S.L. "Software Metrics - A Rigorous & Practical Approach," *ITP*, London, 1997

[18] Fenton, N.E., and Neil, M. “Software metrics: successes, failures and new directions,” *The Journal of Systems and Software*, vol. 47, pp.149-157, 1999.

[19] Fisher, M.J. and Light Jr, W.R. “Sofwtare Quality Management,” *Petrocelli Books*, New York, 1979.

[20] Gray, A., and MacDonell, S.G. “GQM++ A Full Life Cycle Framework for the Development and Implementation of Software Metric Programs,” In *Proceedings of ACOSM ’97 Fourth Austrailian Conference on Software Metrics*, Canberra, Austrailia, ASMA, pp. 22-35, 1997.

[21] Harrison, W., Magel, K., Kluczny, R., and DeDock, A. "Applying Software Complexity Metrics to Program Maintenance," *IEEE Computer*, vol. 15, pp. 65-79, 1982.

[22] Henderson-Sellers, B., Moser, S., Seehusen, S., and Weinelt, B. “A proposed multidimensional framework for object-oriented metrics, measurement – for improved IT management,” *Proc. First Australian Conf. Software Metrics, ACOSM ’93*, J.M. Verner, ed. 24-30, 1993.

[23] Henderson-Sellers, B "Object-Oriented Metrics, Measures of Complexity," *Prentice Hall*, New Jersey, 1996.

[24] ISO 9126 Information Technology - Software Product Evaluation - Quality

Characteristics and Guidelines for Their Use, *International Organization for Standardization*, Geneva, 1992.

[25] http://www.jfree.org/jfreechart/

[26] Karlsson, E. "Software Reuse - A Holistic Approach," *JohnWiley & Sons*, England, 1995.

[27] Kitchenham, B.A., Fenton, N.E., and Pfleeger, S.L. “Towards a Framework for Software Measurement Validation,” *IEEE Trans. Software Eng.,* vol. 21, no. 12, pp. 929-944, 1995.

[28] Kitchenham, B.A., and Pfleeger, S.L. "Software Quality: The Elusive Target," *IEEE Software,* vol. 13, no.1, pp. 12-21, 1996.

[29] Lee, Young, Chang, Kai H., Umphress, D., Hendrix, Dean, and Cross, James H., “Automated Tool for Software Quality Measurement”, *The 13th international conference on software engineering and knowledge engineering (SEKE)*, Buenos Aires, Argentina, June 2001,

[30] Lehman, M., “Programs, Cities, Students, Limits to Growth?” Inaugural Lecture, May 1974, *Publ. in Imp. Col of Sc. Tech.* Inaug. Lect. Ser., vol 9, 1970, 1974, pp 211-229

[31] Lehman, M., “On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle”, *Journal of Sys. and Software*, v. 1, n. 3, 1980, pp 213-221

[32] Lehman, M, “Laws of Software Evolution Revisited”, Position Paper, *EWSPT96*, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp 108-124

[33] Lehman, M., Ramil, J., Wernick, P., Perry, D., “Metrics and laws of software evolution - the nineties view”, *Proceedings of the Fourth International Software Metrics Symposium (1997)*, Portland, Oregon., page 20

[34] Li, W., and Henry, S. "Object-Oriented Metrics that Predict Maintainability," *J. Sys. Software*, 23, pp 111-122, 1993.

[35] Lorenz, M. "Object-Oriented Software Development: A Practical Guide,” *Prentice Hall*, NJ, 227 pp, 1993.

[36] MacDonell, S.G. “Deriving Relevant Functional Measures for Automated Development Project”, *Information and Software Technology 35(9),* pp. 499-512, 1993.

[37] Martin, Robert C. Engineering Notebook, C++ Report, Nov-Dec, 1996

[38] Martin, Robert C., Agile Software Development Principles, Patterns, and Practices, 2002, 255 pp

[39] McCabe, T. J. "A Complexity Measure," *IEEE Trans. on Software Eng*., SE-2(4), pp 308-320, Dec. 1976.

[40] McCall, J.A., Richards, P.K., and Walters, G.F. "Factors in Software Quality," vol. 1, 2, and 3, AD/A-049-014/015/055, *National Tech. Information Service*, Springfield, Va., 1977.

[41] McQuaid, P.A. "Profiling Software Complexity," Ph.D. Dissertation, Computer Science and Engineering Dept., *Auburn University*, 1996.

[42] Moser, S., and Henderson-Sellers, B. ”Object-Oriented Metrics,” *Handbook of object technology, edited-in –chief*, Saba Zamir, Boca Raton, FL, CRC Press, 1999.

[43] Page-Jones, M. "Comparing Techniques by means of Encapsulation and Connascence," *Comm. ACM*, 35(10), pp 147-151,1992, pp147-151

[44] Poulin, J. S. “Measuring Software Reusability,” *Third International Conference on Software Reuse*, Nov. 1994.

[45] Rajaraman, C., and Lyu, M.R. "Some Coupling Measure for C++ programs," In *Procs. TOOLS USA '92*, Prentice Hall, Englewood Cliffs, NJ, pp. 225-234, 1992.

[46] Reyes, M.L. “Assessing the reuse potential of objects,” Ph.D. Dissertation, Computer Science Dept., *Louisiana state university*, 1998.

[47] Schneidewind, N.F. “Report on IEEE Standard Software Quality Metrics Methodology,” *Software Engineering Notes*. Vol.18,no.3,pp.A-95 – A-98, July 1, 1993.

[48] Stevens, W.P., Myers, G.J., and Constantine, L.L. “Structured Design,” *IBM Syst. J.*, 13(2), pp115-139, 1974.

[49] Walsh, T.J. “A Software Reliability Study Using a Complexity Measure,” *Procs. of the 1979 Computer Conference*, Montville, NJ: AFIPS Press, pp. 761-768, 1979.

[50] Withrow, C. “Error Density and Size in Ada Software”, *IEEE Software*, pp26-30, Jan. 1990.

[51] Zuse, H. “Software Complexity: Measures and Methods,” Walter de Gruyter, Berlin, 1991.

APPENDIX A. Visualization of Software Evolution

Research on how a software system evolves over time is difficult and time consuming. The enormous amount of work required by analyzing software evolution makes it difficult without the dedicated tools such as *JamTool*. Automated environments could be key factors in conducting a successful empirical study on software evolution.

Moreover, there are two major challenges that must be overcome in software evolution research. These challenges limit our ability to understand the history of software systems, thus prevent us from generalizing our observations into software evolution theory. The first challenge is how to organize the enormous amount of historical data in a way that allows us to access them quickly and easily. The second challenge is how to analyze the structural changes of software systems.

To overcome these challenges, we use visualization technique in a form of table to provide the overview of the evolution history. We observe the evolution history of real world software system*, JFreeChart.* This system is investigated to demonstrate the effectiveness of our approach as an example to demonstrate the use of various functionalities of *JamTool*. We also introduce several ways to track and analyze the software structural changes from past releases.

A.1   Evolution Track Table

In this section we present the global visualization of software evolution using an evolution track table, which is created to visualize the evolution of a software system.

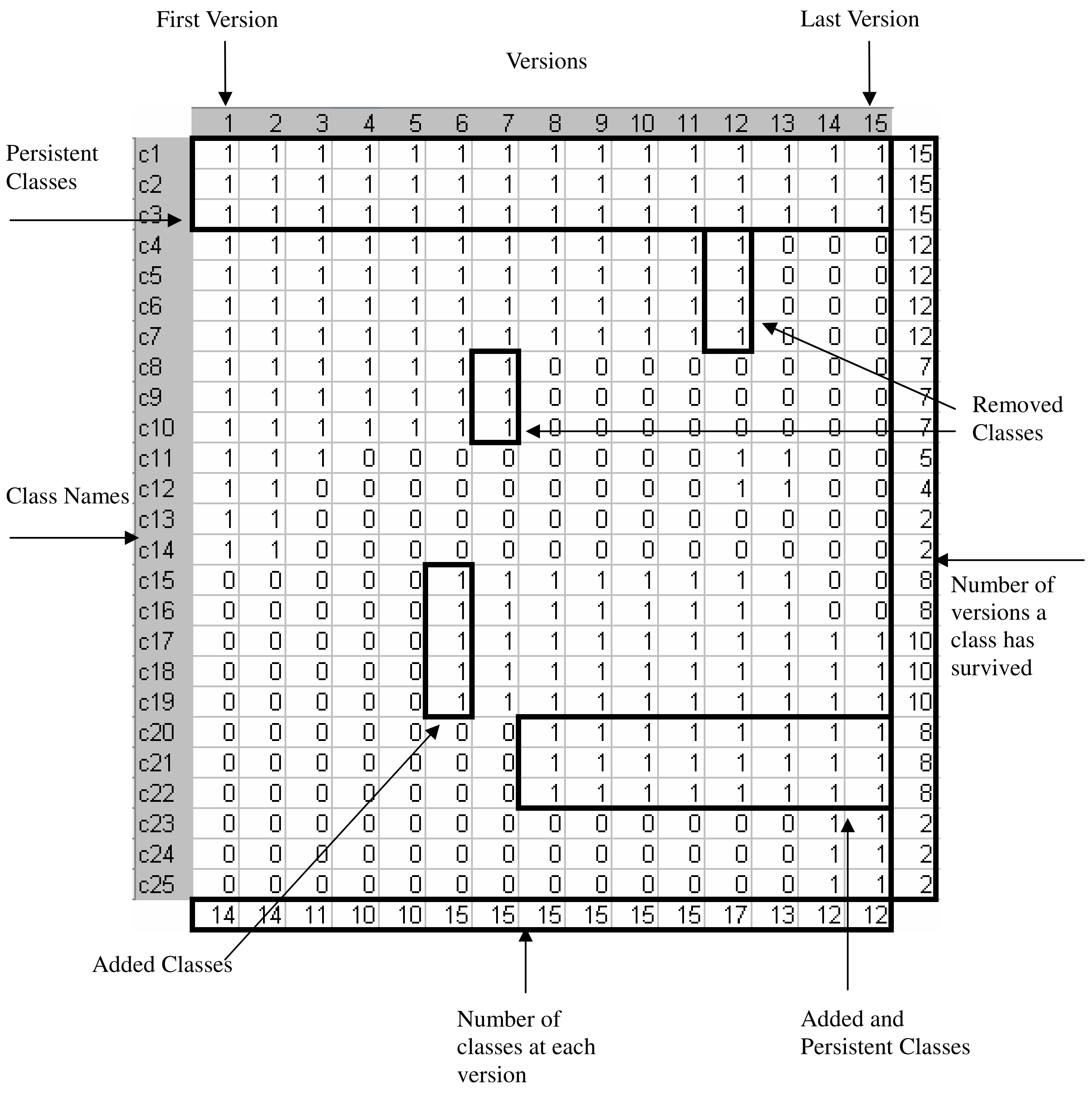
The evolution of classes of a software system can be visualized in an evolution track table as shown in Figure A-1. This table visualizes each class's lifecycle for a software system in Microsoft Excel to achieve various data analysis, and it provides effective ways to analyze the evolution of the system. Each column of the table represents a version of the software, while each row represents a class name in each version. To create the table, we collect and list all class names which are the member of the system at least once, and display '1' or '0' depending on whether or not a class is a member of a version of the system. In this way, the class name which lasts the longest in the evolution appears first.

<u>**Characteristics of Evolution Track Table**</u>

From an evolution track table, we are able to obtain the following information regarding the evolution of a system.

- Size of the system

We can find out how many classes are involved in system evolution. The summation of '1's in each column is the number of classes existed in that particular version of the system. For instance, there are 14 classes in versions 1 and 2 and a total of 25 classes are involved in the evolution in Figure A-1.


First Version
Last Version
Versions
Persistent Classes
Class Names
Removed Classes
Number of versions a class has survived
Added Classes
Number of classes at each version
Added and Persistent Classes

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
c1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
c2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
c3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
c4	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	12
c5	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	12
c6	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	12
c7	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	12
c8	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	7
c9	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	7
c10	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	7
c11	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0	5
c12	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	4
c13	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
c14	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
c15	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	8
c16	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	8
c17	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
c18	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
c19	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	10
c20	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	8
c21	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	8
c22	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	8
c23	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2
c24	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2
c25	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	2
	14	14	11	10	10	15	15	15	15	15	15	17	13	12	12	

Figure A-1: Software evolution in an evolution track table

- Removed and added classes

The classes which have been removed or added in a certain version can be easily detected. The difference between two subsequent versions tells us that if a class is removed or added. If the number is changed from ‘1’ to ‘0’ between two consecutive versions, a class is removed, and if the number is changed from ‘0’ to ‘1’, a new class is added. For example, in Figure A-1, classes c4, c5, c6, and c7 are removed in version 12 and their absence will leave ‘0’ on the table from that version on. Classes, c15, c16, c17, c18, and c19 are newly added to version 6. Therefore, in this example, a total of 13 classes are removed and a total of 11 new classes are added. By detecting the removed and added classes, we see very easily when/how much the system is changed.

- Persistent classes

Persistent classes have the same lifetime as the whole system. They have stayed from the beginning to the end. Those classes should be examined since they may be important in performing key functions of the system as being a part of the original system design. In Figure 5-8, three classes, c1, c2, and c3 are persistent classes.

- Added persistent classes

Some important added classes have stayed until the last version. They might be created to upgrade or improve system as being a part of redesign of the system with some problematic classes removed. In Figure A-1, six classes, c17, c18, c19, c20, c21, and c22 are added persistent classes.

A.2 Tracking Class Evolution

Understanding the evolution of an object-oriented system based on various versions of source code requires analyzing a vast amount of data since an object-oriented system has a complex structure consisting of classes, methods, attributes and different kinds of relationships between them rather than simply a set of classes. Using an evolution track table designed for this study, we provide an approach to understand such an evolution by detecting and visualizing the evolution pattern that characterizes classes. Evolution track table helps us understand an overall evolution of a system, discover problematic parts with unusual measurement values, and visually get a quick understanding of the analyzed history. Thus, in this case study we present the visualization of the evolution track table, and explain how this table can be read, thus how an object-oriented system has evolved into its current state based on the source code.

We use 22 versions of *JFreeChart* as a target system for this study since *JFreeChart* is a long-term open source charting library with many releases.

Size of the System

From the evolution track table along with 22 versions of *JFreeChart,* we collect the number of classes, the removed, and the added classes in each version as shown in Table A-1.

Based on this information, we are able to find out how big the system is and how many classes are involved in the system evolution. This system started with 139 classes at version 0.9.0 and ended with 460 classes at version 0.9.21, which means a 333% class growth.

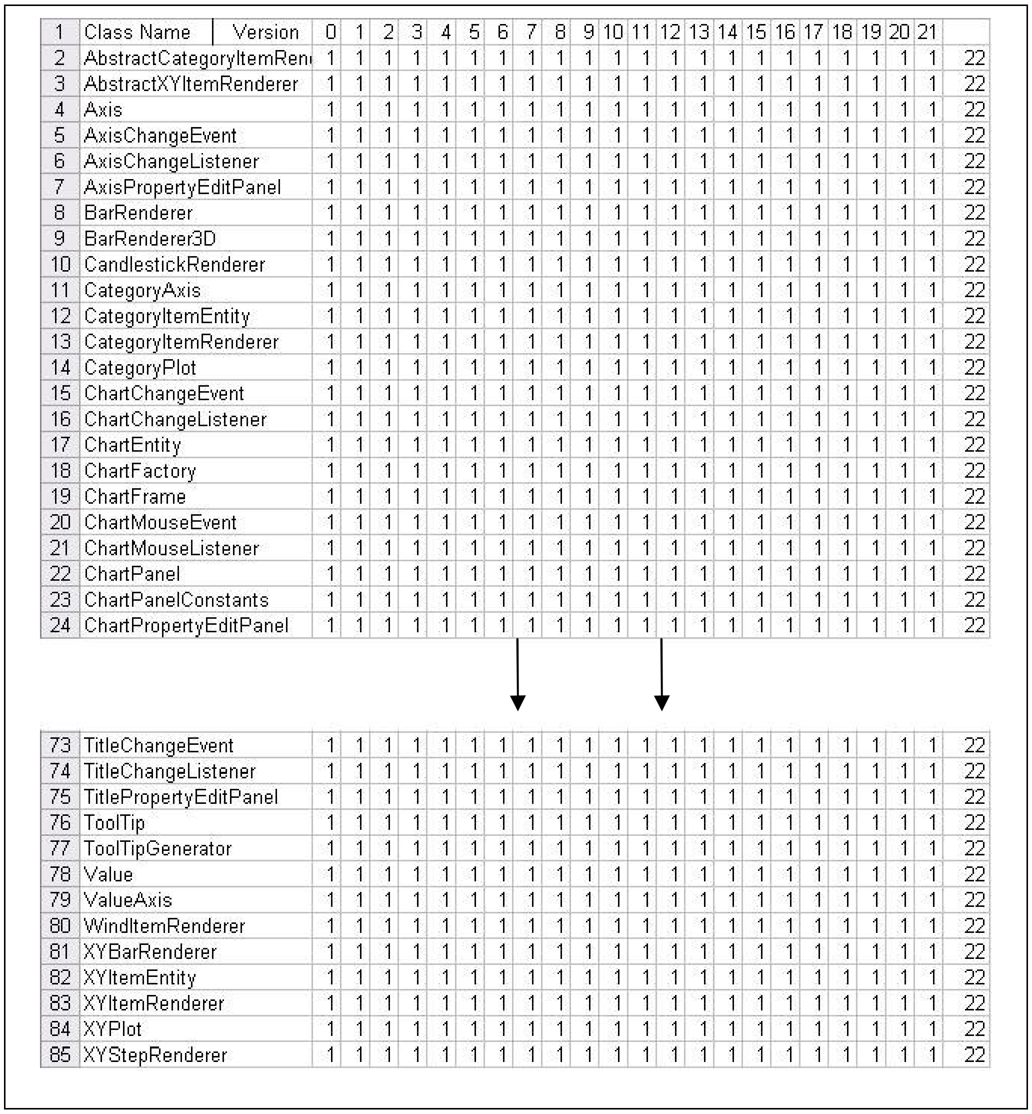
The number of classes increases gradually and consistently as new versions evolve. A total of 569 classes are involved in the evolution. During the evolution, 123 classes are removed while 444 classes are added, which is 3.6 times more than the removed. In most versions more classes are added than removed. Special attention can be given to versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17 since 68% (84 out of 123) of the removed were removed and 60% (265 out of 444) of the added were added in those particular versions.

Table A-1: Number of classes, removed and added

Version of JFreeChart	No. of Removed classes	No. of Added classes	Total no. of classes
0.9.0			139
0.9.1	1	0	138
0.9.2	0	6	144
0.9.3	0	113	257
0.9.4	3	21	275
0.9.5	22	74	327
0.9.6	0	2	329
0.9.7	1	25	353
0.9.8	0	3	356
0.9.9	43	48	361
0.9.10	11	2	352
0.9.11	0	13	365
0.9.12	5	17	377
0.9.13	0	6	383
0.9.14	3	15	395
0.9.15	0	9	404
0.9.16	2	10	412
0.9.17	19	30	423
0.9.18	1	10	432
0.9.19	9	24	447
0.9.20	0	1	448
0.9.21	3	15	460
Total	123	444	

Persistent Classes

Persistent classes have survived through the entire life of a software system. They can be easily detected by looking at ‘1’ at all versions and the total number of versions in the last column. As shown in Figure A-2, they have ‘1’s for all versions and ‘22’ in the last column, which is the number of versions of the target system from 0.9.0 to 0.9.21. We found out that 84 out of the 138 classes in the first version have survived through the entire life of the target system, which is about 61 % of the original design classes. From this result, we see that 54 classes of the original were removed during the evolution.

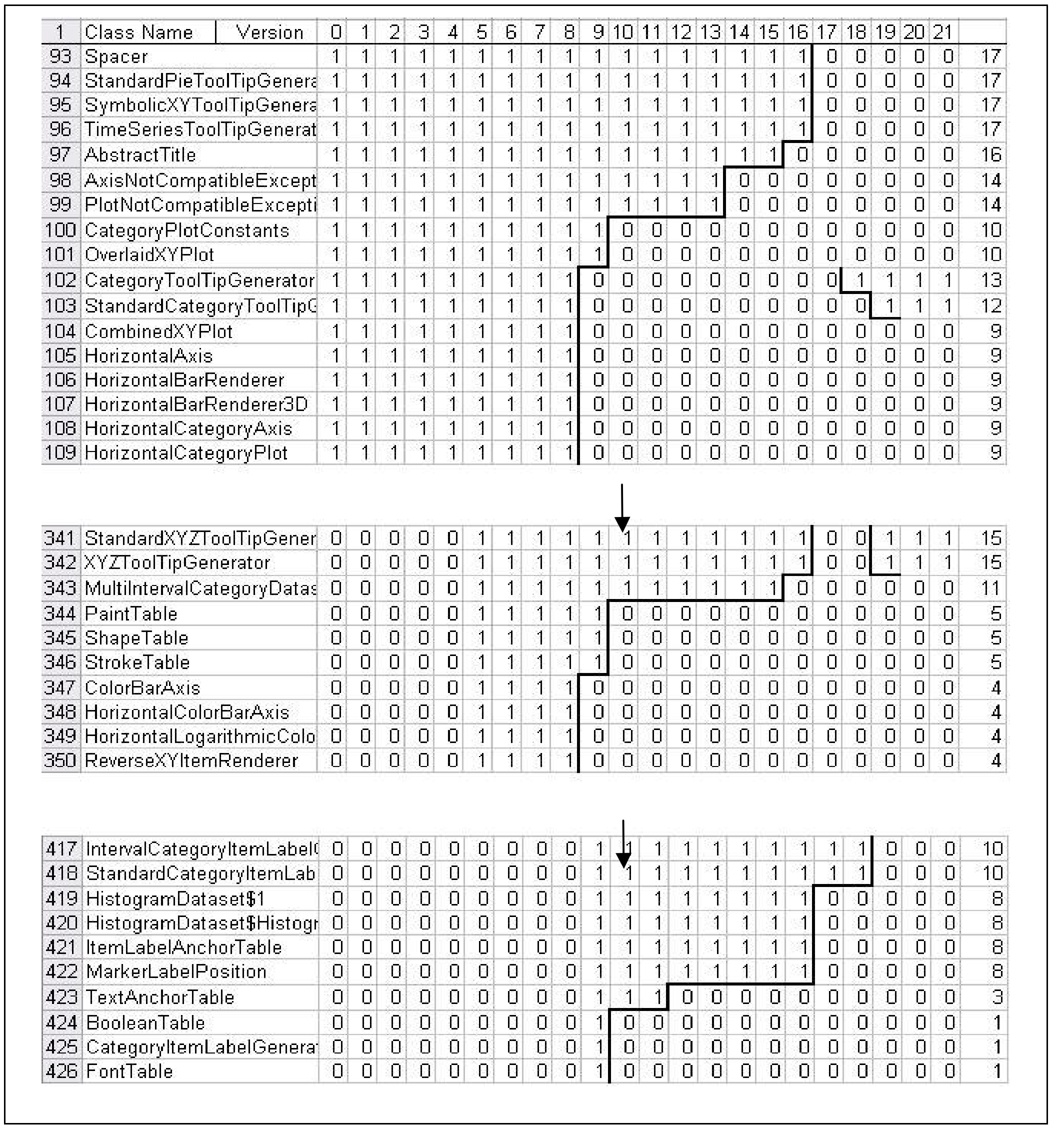


1	Class Name	Version	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
2	AbstractCategoryItemRen		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
3	AbstractXYItemRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
4	Axis		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
5	AxisChangeEvent		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
6	AxisChangeListener		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
7	AxisPropertyEditPanel		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
8	BarRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
9	BarRenderer3D		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
10	CandlestickRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
11	CategoryAxis		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
12	CategoryItemEntity		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
13	CategoryItemRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
14	CategoryPlot		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
15	ChartChangeEvent		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
16	ChartChangeListener		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
17	ChartEntity		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
18	ChartFactory		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
19	ChartFrame		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
20	ChartMouseEvent		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
21	ChartMouseListener		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
22	ChartPanel		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
23	ChartPanelConstants		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
24	ChartPropertyEditPanel		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
73	TitleChangeEvent		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
74	TitleChangeListener		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
75	TitlePropertyEditPanel		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
76	ToolTip		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
77	ToolTipGenerator		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
78	Value		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
79	ValueAxis		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
80	WindItemRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
81	XYBarRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
82	XYItemEntity		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
83	XYItemRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
84	XYPlot		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22
85	XYStepRenderer		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	22

Figure A-2: Persistent classes

Removed Classes

From an evolution track table, we can find what classes are removed from which version of the system. The removed classes can be detected by finding the differences between two subsequent versions from ‘1’ to ‘0’ as shown in Figure A-3. In this way, we found that many classes are removed during the evolution (See Table A-1). In particular, 22, 43, and 19 classes were removed in versions 0.9.5, 0.9.9, and 0.9.17, respectively. These data might imply that in those versions the system was aggressively changed.

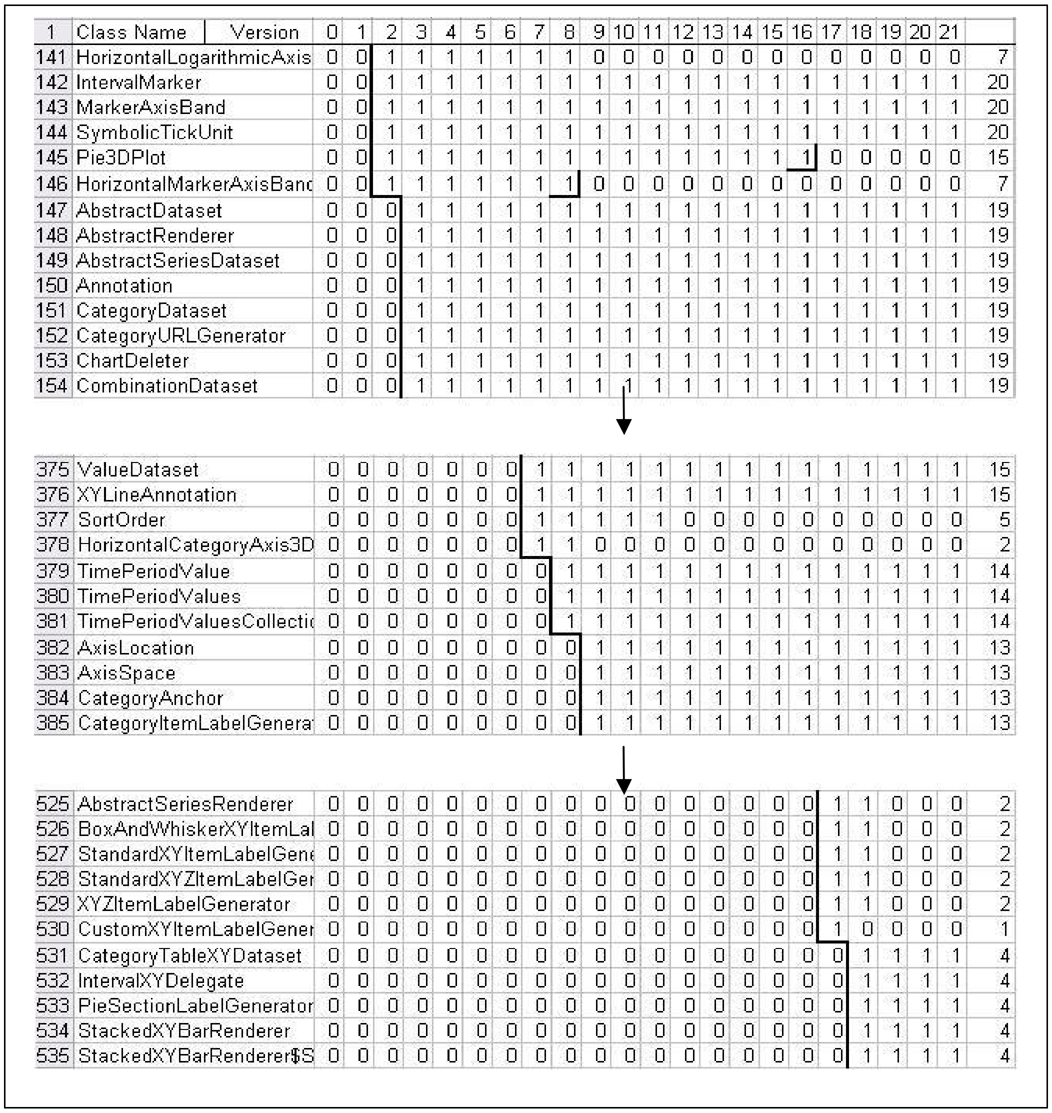


1	Class Name Version	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
93	Spacer	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	17
94	StandardPieToolTipGenera	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	17
95	SymbolicXYToolTipGenera	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	17
96	TimeSeriesToolTipGenerat	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	17
97	AbstractTitle	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	16
98	AxisNotCompatibleExcept	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	14
99	PlotNotCompatibleExcepti	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	14
100	CategoryPlotConstants	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	10
101	OverlaidXYPlot	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	10
102	CategoryToolTipGenerator	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	13
103	StandardCategoryToolTipG	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	12
104	CombinedXYPlot	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
105	HorizontalAxis	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
106	HorizontalBarRenderer	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
107	HorizontalBarRenderer3D	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
108	HorizontalCategoryAxis	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
109	HorizontalCategoryPlot	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
341	StandardXYZToolTipGener	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	15
342	XYZToolTipGenerator	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	15
343	MultiIntervalCategoryDatas	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	11
344	PaintTable	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	5
345	ShapeTable	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	5
346	StrokeTable	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	5
347	ColorBarAxis	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4
348	HorizontalColorBarAxis	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4
349	HorizontalLogarithmicColo	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4
350	ReverseXYItemRenderer	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	4
417	IntervalCategoryItemLabel	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	10
418	StandardCategoryItemLab	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	10
419	HistogramDataset$1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	8
420	HistogramDataset$Histogr	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	8
421	ItemLabelAnchorTable	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	8
422	MarkerLabelPosition	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	8
423	TextAnchorTable	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	3
424	BooleanTable	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
425	CategoryItemLabelGenera	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
426	FontTable	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure A-3: Removed classes

Some classes, which were removed in previous version, reappear later, like classes *CategoryToolTipGenerator* and *StandardCategoryToolGenerator*. They are removed from the system in version 0.9.8, but came back in versions 0.9.18 and 0.9.19, respectively. Classes *StandardXYZToolTipGenerator* and *XYZToolTipGenerator* were removed in 0.9.16, came back in 0.9.19, and stayed until the last version of the system. These kinds of interesting changes can be detected by the evolution track table

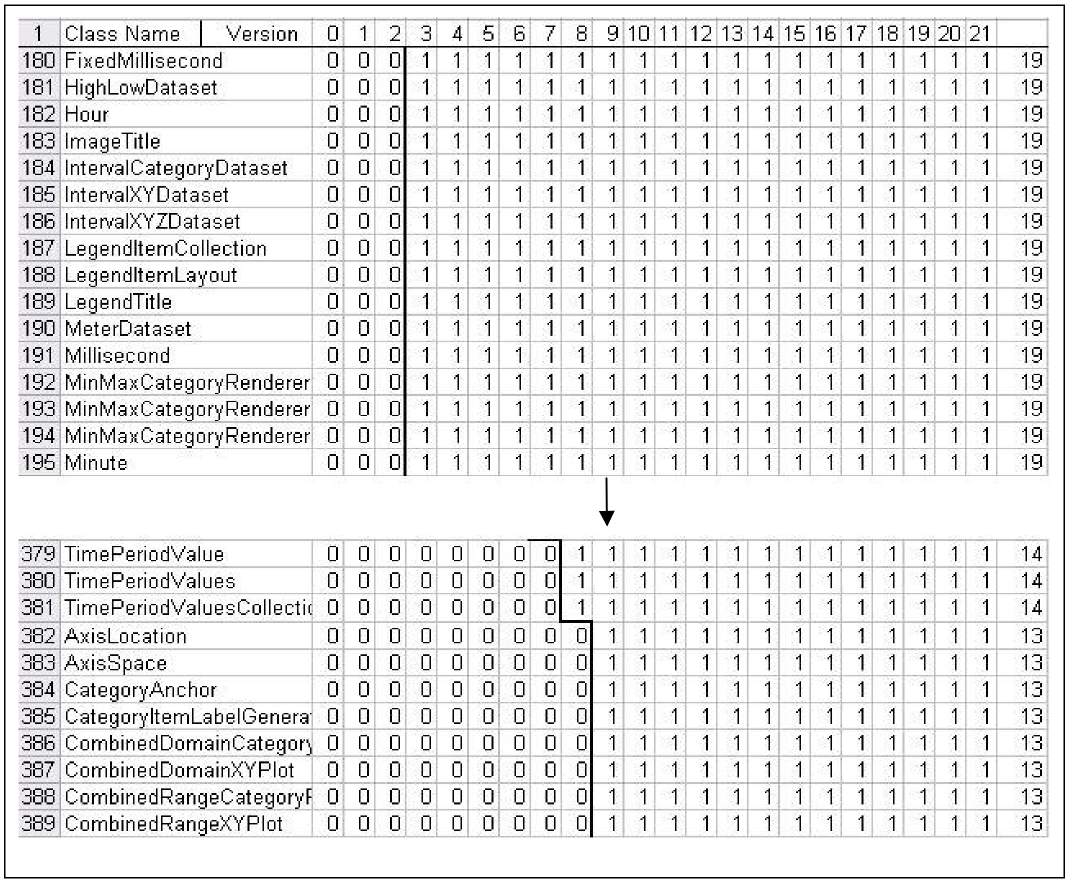
Added Classes



1	Class Name Version	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
141	HorizontalLogarithmicAxis	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	7
142	IntervalMarker	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20
143	MarkerAxisBand	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20
144	SymbolicTickUnit	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20
145	Pie3DPlot	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	15
146	HorizontalMarkerAxisBanc	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	7
147	AbstractDataset	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
148	AbstractRenderer	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
149	AbstractSeriesDataset	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
150	Annotation	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
151	CategoryDataset	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
152	CategoryURLGenerator	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
153	ChartDeleter	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
154	CombinationDataset	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
375	ValueDataset	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
376	XYLineAnnotation	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	15
377	SortOrder	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	5
378	HorizontalCategoryAxis3D	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
379	TimePeriodValue	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
380	TimePeriodValues	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
381	TimePeriodValuesCollectic	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
382	AxisLocation	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
383	AxisSpace	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
384	CategoryAnchor	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
385	CategoryItemLabelGenera	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
525	AbstractSeriesRenderer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2
526	BoxAndWhiskerXYItemLal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2
527	StandardXYItemLabelGene	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2
528	StandardXYZItemLabelGer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2
529	XYZItemLabelGenerator	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	2
530	CustomXYItemLabelGener	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
531	CategoryTableXYDataset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
532	IntervalXYDelegate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
533	PieSectionLabelGenerator	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
534	StackedXYBarRenderer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
535	StackedXYBarRenderer$S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4

Figure A-4: Added classes

The added classes can be detected by finding the differences between two subsequent versions from ‘0’ to ‘1’ as shown in Figure A-4. In this way, we find how many classes were newly added into which version of the system during the evolution (See Table 5-4). In the case of the target system, many classes were added at almost every version. In particular, there were 113, 74, 48, and 30 classes added to 0.9.3, 0.9.5, 0.9.9, and 0.9.17, respectively. Some classes like *Pie3DPlot* and *HorizontalMarkerAxisBand* were removed after staying for several versions. From the results of the removed and added classes, we found that this system had made huge changes in versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17. These versions may need to be specifically investigated



1	Class Name	Version	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
180	FixedMillisecond		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
181	HighLowDataset		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
182	Hour		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
183	ImageTitle		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
184	IntervalCategoryDataset		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
185	IntervalXYDataset		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
186	IntervalXYZDataset		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
187	LegendItemCollection		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
188	LegendItemLayout		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
189	LegendTitle		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
190	MeterDataset		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
191	Millisecond		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
192	MinMaxCategoryRenderer		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
193	MinMaxCategoryRenderer		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
194	MinMaxCategoryRenderer		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
195	Minute		0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	19
379	TimePeriodValue		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
380	TimePeriodValues		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
381	TimePeriodValuesCollecti		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	14
382	AxisLocation		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
383	AxisSpace		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
384	CategoryAnchor		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
385	CategoryItemLabelGenera		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
386	CombinedDomainCategory		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
387	CombinedDomainXYPlot		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
388	CombinedRangeCategoryF		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13
389	CombinedRangeXYPlot		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	13

Figure A-5: Added and persistent classes

<u>Added Persistent Classes</u>

Many classes added in the middle of the evolution have stayed until the last version of the system. We call them ‘added persistent classes’. Figure A-5 shows examples of added persistent classes, and they were added in different versions when the system was changed from one state to another. Table A-2 displays the number of added persistent classes and their survival rate in each version. If we compare these with the number of added classes, we find that a total of 444 classes were added to the system and 349 classes (81.35%: 349 out of 429) have survived till the last.

Table A-2: Number of added persistent classes

Version of JFreeChart	No. of added classes	No. of added persistent classes	Survival rate
0.9.1	0	0	
0.9.2	6	3	50%
0.9.3	113	89	78.76%
0.9.4	21	19	90.48%
0.9.5	74	61	82.43%
0.9.6	2	0	0%
0.9.7	25	23	92%
0.9.8	3	3	100%
0.9.9	48	35	72.92%
0.9.10	2	2	100%
0.9.11	13	10	76.92%
0.9.12	17	17	100%
0.9.13	6	6	100%
0.9.14	15	15	100%
0.9.15	9	9	100%
0.9.16	10	9	90%
0.9.17	30	24	80%
0.9.18	10	5	50%
0.9.19	24	19	79.17%
0.9.20	1	1	100%
0.9.21	15	-	
Total	429 = 444-15	349	81.35%

This is certainly comparable to the persistent classes (61% survival rate of the original design classes).

A.3 Summary

From the evolution track table of *JFreeChart,* we summarize the following findings:

- o Started with 139 classes in version 0.9.0
- o Ended with 460 classes (333% growth) in version 0.9.21
- o 84 (60%) out of the 139 original classes have stayed until the last version
- o 569 classes were involved in whole system evolution
- o 123 classes were removed during the evolution
- o 444 classes were added during the evolution
- o 349 (81%) out of the 429 (444 added classes – number of classes in the last version 0.9.21) added classes have stayed until the last version
- o Big changes occurred in versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17 in terms of removed and added classes.

Based on the findings above, we conclude that the evolution track table can be used in the following tasks:

- o To categorize the evolution of classes

  We found the groups of persistent, removed, added, and added persistent classes from the evolution track table of *JFreeChart.* They characterize the evolution pattern of the system

- To identify unusual evolution pattern of classes

  We found that some classes had stayed unusually for only one, two, or several versions. These dynamic classes need to be analyzed to understand the architecture of the system.