Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled and is characterized by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. In OOAD the boundary between analysis and design is blurred. One reason for this blurring is the similarity of basic constructs (i.e., objects and classes) that are used in analysis and design. The fundamental difference between OOA and OOD is that OOA models the problem domain, leading to an understanding and specification of the problem, while OOD models the solution to the problem. That is analysis deals with the problem domain, while design deals with the solution domain. However, in OOAD it is believed that the problem domain representation created by OOA is generally subsumed in the solution domain representation. That is, the solution domain representation, created by OOD, generally contains much of the representation created by OOA.

Dr. Amandeep Kaur is working as an Assistant Professor in the Department of Computer Science at Sri Guru Granth Sahib World University, Fategharh Sahib, Punjab, India. She has more than 5 years of teaching experience in academia. She has published more than 15 research papers in refereed international journals, conferences and workshop proceedings.

Change Proneness of Metrics and Classes in Object-Oriented Systems
Amandeep Kaur
Gaurav Dhiman

Change Proness of Metrics and Classes in Object-Oriented Systems
Change Proness of Metrics and Classes in Object-Oriented Systems
Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius

Printed at: see last page

Zugl. / Approved by: Guru Nanak Dev University, Amritsar

Copyright © Amandeep Kaur, Gaurav Dhiman
Copyright © 2019 International Book Market Service Ltd., member of OmniScriptum Publishing Group
Chapter 1: INTRODUCTION TO OBJECT ORIENTED PARADIGMS

1.1 INTRODUCTION

Object oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. In OOAD the boundary between analysis and design is blurred. One reason for this blurring is the similarity of basic constructs (i.e., objects and classes) that are used in analysis and design [52] [53].

The fundamental difference between OOA and OOD is that OOA models the problem domain, leading to an understanding and specification of the problem, while OOD models the solution to the problem. That is analysis deal with the problem domain, while design deals with the solution domain. However, in OOAD it is believed that the problem domain representation created by OOA is generally subsumed in the solution domain representation. That is, the solution domain representation, created by OOD, generally contains much of the representation created by OOA. There are a number of different notations for representing these models, such as the Unified Modeling Language.

OOAD contributes to the blurring of boundaries between analysis and design. It is often not clear where analysis ends and design begins. The lack of clear separation between analysis and design can also be considered one of the strong points of the object oriented approach—the transition from analysis to design is “seamless.” The main difference between OOA and OOD is in the types of objects that come out of the analysis and design process [52] [53].

The objective of OOA is to develop a series of model that describe computer software as it works to satisfy a set of customer defined requirements. The objects during OOA focus on problem domain and generally give clear understanding of problem. These objects are sometime called semantic objects as they have a meaning in the problem domain. OOA applies object-modeling techniques to analyze the functional requirements for a system. OOA models the problem domain, leading to more clear understanding of the problem. OOA focuses on what the system does. Object-oriented analysis (OOA) is the process of analyzing a task (problem domain), to develop a conceptual model that can then be used to complete the task. A typical OOA model would describe computer software that could be used to satisfy a set of customer-defined requirements. During the analysis phase of problem-solving, a programmer might consider a written requirements statement, a formal
vision document, or interviews with stakeholders or other interested parties. The task to be addressed might be divided into several subtasks (or domains), each representing a different business, technological, or other areas of interest. Each subtask would be analyzed separately. Implementation constraints, (e.g., concurrency, distribution, persistence, or how the system is to be built) are not considered during the analysis phase; rather, they are addressed during object-oriented design (OOD).

![Diagram](image)

Fig1.1: Relationship between OOA and OOD

Object oriented design (OOD) elaborates the analysis models to produce implementation specifications. In OOD, the focus is to identify the classes that must exist in software and the relationship between classes. OOD focuses on how the system does it. The objects during OOD focus on finding and defining solution. Solution domain consists of semantic object as well as other objects. The solution domain objects include interface, application, and utility objects. The interface objects deal with the user interface, which is not directly a part of a of the problem domain but represent some aspect of the solution desired by user. The application objects specify the control mechanism for the proposed solution. Utility objects are those needed to support the services of the semantic object [52] [53].
The basic goal of the analysis and design activities is to identify the classes in the system, their relationships, and frequently represented by class diagrams. Some of the basic concepts of object oriented are:

- **CLASS:**

A class is an object oriented concept that encapsulates the data and procedural abstraction that are required to describe the behavior of some real world entity. In other words a class is a generalized description that describes a collection of similar objects. A super-class is a collection of classes and a subclass is an instance of class.

![Diagram of a class](image)

**Fig 1.2:** An alternative representation of an object oriented class

A class defines a possible set of objects. Classes have:

1. An interface that defines which parts of an object of a class can be accessed from outside and how
2. A class body that implements the operations in the interface.
3. Instance variables that contain the state of an object of that class.
OBJECT:

An object is a basic building block of object oriented programming. Programmers developing a system order, create object classes to represent each component of a system. An object encapsulates data and the algorithms that process the data. These algorithms are called operations. Each of the operations that are encapsulated by an object provides a representation of one of the behaviors of the object. Objects of a class have some attributes, whose values constitute much of the state of an object. When objects of a class are created, memory for the objects is allocated. Each object, when it is created, gets a private copy of the instance variables, and when an operation defined on the class is performed on the object, it is performed on the state of the particular object.

The relationship between a class and objects of that class is similar to the relationship between a type and elements of that type. A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class. The class specifies the operation that can be performed on the objects of that class and the interface of each of the operations.

1.2 ADVANTAGES OF OBJECT ORIENTED ANALYSIS

✓ Simplicity:

Software objects model real world objects, so the complexity is reduced and the program structure is very clear.

✓ Modularity:

Each object forms a separate entity whose internal workings are decoupled from other parts of the system.

✓ Modifiability:

It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.
✓ **Extensibility:**
Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

✓ **Real-World Modeling:**
Object-oriented system tends to model the real world in a more complete fashion than do traditional methods. Objects are organized into classes of objects, and objects are associated with behaviors. The model is based on objects, rather than on data and processing.

✓ **Improved Reliability and Flexibility:**
Object-oriented system promise to be far more reliable than traditional systems, primarily because, new behaviors can be "built" from existing objects. Because objects can be dynamically called and accessed, new objects may be created at any time. The new objects may inherit data attributes from one, or many other objects. Behaviors may be inherited from super-classes, and novel behaviors may be added without effecting existing systems functions.

✓ **Maintainability:**
OOP methods make code more maintainable. Identifying the source of errors becomes easier because objects are self-contained. Objects can be maintained separately, making locating and fixing problems easier.

✓ **Re-usability:**
Objects can be reused in different programs. Because objects contain both data and functions that act on data, objects can be thought of as self-contained “boxes” (encapsulation). This feature makes it easy to reuse code in new systems. When a new object is created, it will automatically inherit the data attributes and characteristics of the class from which it was spawned. The new object will also inherit the data and behaviors from all super classes in which it participates [44] [45].
1.3 PROBLEMS WITH OBJECT ORIENTED SYSTEM ANALYSIS AND DESIGN

– Form And Substance

Project managers and programmers may confuse style with substance. Some developers and managers today believe that OOAD simply means defining classes, objects, and methods. It is like what happened when structured programming development became popular. Developers and managers at first thought that structured development simply meant that one had to remove GOTO statements and increase the use of subroutines. While these steps helped, they were just surface features of a deeper group of more significant principles.

– Implications of OOAD

Managers and developers may not recognize all the implications of OOAD. They often assume that using OOAD will eliminate development bottlenecks. This is due to expectations that OOAD will reduce the complexity of programs and will provide architectural modularity. This different approach to architecture may work better with different management and scheduling techniques.

– Abandonment of Traditional Design Processes

Managers may be tempted to ignore or abandon traditional software design and engineering processes. OOAD is often adopted because of the promise of increased productivity coupled with a shortened development schedule. However, a sufficient amount of development time is often not allowed that will make sure the design processes are followed correctly. This may actually result in bigger problems. These include missed deadlines, schedule slippage, and project failures.

– Insufficient OOAD Training

Finally, one of the disadvantages of object oriented analysis and design is that developers may get some education about OOAD, but not enough. A false sense of programming confidence may stem from just reading articles, or even a book, or taking a class in it. Programmers will need time to immerse themselves in the nuances of OOAD [46] [47].
1.4 CHANGES IN A PROGRAM FROM ONE VERSION TO ANOTHER

The program changes from one version to another because of maintenance. Maintenance includes all the activities after installation of software that is performed to keep the system operational. Maintenance is costly because:

i. It is difficult to understand the code written by other.

ii. If there are more errors in coding, more will be pointed in testing and there is more probability to get accumulated in maintenance. Due to which more money and time will be spent.

NEED OF MAINTENANCE

Maintenance is needed because:

i. It is impossible to remove all the errors.

ii. Requirements are primarily dynamic in nature.

iii. Environment in which the software is made operational, software changes.

1.4.1 TYPES OF MAINTENANCE

Four various types of maintenance are:

- **CORRECTIVE MAINTENANCE:**

  Fixing errors refer to corrective maintenance. Before delivery it is impossible to remove all the errors. Six sigma principle state that three errors per million lines of code should be there. It is impossible to reach at this level. Some of the errors always remain left in software. These get encounter whenever the system is installed. Fixing of those uncorrected errors is done in corrective method.

- **PERFECTIVE MAINTENANCE:**

  As requirements are dynamic in nature, they remain or keep on changing with time. In perfective methods these requirements are incorporated in system. So it refers to enhancing the performance or modifying the program in order to meet the changed need of used.
• **ADAPTIVE MAINTENANCE:**
  Maintenance is also required whenever the environment in which software is made operative changes. This change in environment also changes the desired functionality of system. The software is changed in the developer environment and is installed in user environment; this change may involve the changes in input, output or environment. So it is done in adaptive maintenance.

• **PREVENTIVE MAINTENANCE:**
The primary goal of preventive maintenance is to prevent the failure of equipment before it actually occurs. It is designed to preserve and enhance equipment reliability by replacing worn components before they actually fail. Preventive maintenance activities include equipment checks, partial or complete overhauls at specified periods, oil changes, lubrication and so on. In addition, workers can record equipment deterioration so they know to replace or repair worn parts before they cause system failure. Recent technological advances in tools for inspection and diagnosis have enabled even more accurate and effective equipment maintenance. The ideal preventive maintenance program would prevent all equipment failure before it occurs.

Benefits of preventive maintenance include:

- Improved system reliability.
- Decreased cost of replacement.
- Decreased system downtime.
- Better spares inventory management [49] [50].

1.5 **CLASS QUALITY**

Object oriented analysis, design, and implementation are an iterative process. It is usually impossible to fully and correctly design the classes of an OO system at the outset of a project.

Booch proposes five metrics to measure the quality of classes:

➤ **COUPLING**

How closely do classes rely on each other?
Inheritance makes for strong coupling (generally a bad thing) but takes advantage of the re-use of an abstraction (generally a good thing).

➤ **COHESION**

How tied together are the state and behavior of a class? Does the abstraction model a cohesive, related, integrated thing in the problem space?

➤ **SUFFICIENCY**

Does the class capture enough of the details of the thing being modeled to be useful?

➤ **COMPLETEENESS**

Does the class capture all of the useful behavior of the thing being modeled to be re-usable? What about future users (re-users) of the class? How much more time does it take to provide completeness? Is it worth it?

➤ **PRIMITIVE**

Do all the behaviors of the class satisfy the condition that they can only be implemented by accessing the state of the class directly? If a method can be written strictly in terms of other methods in the class, it isn’t primitive. Primitive classes are smaller, easier to understand, with less coupling between methods, and are more likely to be reused. If one tries to do too much in the class, then it will be difficult to use for other designers.

Sometimes issues of efficiency or interface ease-of-use will violate the general recommendation of making a class primitive. For example, one might provide a general method with many arguments to cover all possible uses, and a simplified method without arguments for the common case [51].

### 1.6 SOFTWARE METRICS

Software metrics are quantifiable measures that could be used to measure different characteristics of a software system or the software development process. Metrics ensure that the product is of high quality and the productivity of project stays high.

**Definition 1:** According to S.R. Chidamber software metrics is a function, with input as the software data, and output is a value which could decide on how the given attribute affect the software [32].
**Definition 2:** Software metrics can be defined as the continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

**Definition 3:** Software metrics provide a measurement for the software and the process of software production. It is giving quantitative values to the attributes involving in the products or the process [31].

**Definition 4:** Software metrics is to give the attributes some quantitative descriptions. These attributes are extracting from the software product, software development process and the related resources. They are listed as below:

**Product:** the documents and programs generated during the software development process.

**Process:** various processes related to software development, such as software design, implementation, test, and maintenance, etc.

**Resources:** the supporting resources such as programmers, and cost of the product and processes, etc.

Software metrics are an integral part of the state-of-the-practice in software engineering. More and more customers are specifying software and or quality metrics as part of their contractual requirements. Companies are using metrics to better understand, track, control, and predict software projects, processes and products. The term software metrics means different things to different people.

Software metrics can provide the information needed by engineers for technical decisions as well as information required by management. If a metric is to provide useful information, everyone involved in selecting, designing, implementing, collecting, and utilizing it must understand its definition and purpose.

Another reason that software metrics are very important is the increased importance of software also places more requirements on it. Thus, it is necessary to have precise, predictable, and repeatable control over the software development process and product.
Moreover, software projects are often late, over budget, and fail to perform as expected. Effective resources allocation, reduction of design complexity, and adoption of effective software engineering techniques are thus the key for resolving or reducing such risks. Software metrics, which are quantitative measure of specific attributes of a software development, including software process and software product, have been suggested as useful means to assist in achieving these goals [30].

1.6.1 Characteristics of Software Metrics

Each Metrics must follow some characteristics to be a good metric for the measurement these are:

− **Objective statement**: The objective for each metric can be formally defined in terms of one of the following functions, the attribute of entity being measured and the goal for the measurement.

− **Understand**: Metrics can help us to understand more about our software products, processes and services.

− **Evaluate**: Metrics can be used to evaluate our software products, processes and services against established standard and goals.
- **Control:** Metrics can provide the information that needed to control resources and processes used to produce our software.

- **Predict:** Metrics can be used to predict attributes of software entities in future.

- **Clear Definitions:** The second step in designing a metric is to agree to a standard definition for the entities and their attributes being measured. When terms like defect, problem report, size and even project are used, other people will interpret these words in their own context with meanings that they may differ from our intended definition. These interpretation differences increase when more ambiguous terms like quality, maintainability and user friendliness are used.

- **Define the Model:** The model defines how the metrics are going to be calculated. Some metrics called metric primitives that are measured directly and their model typically consists of a single variable. Other more complex metrics are modeled using mathematical combinations of metrics primitives or other complex metrics.

- **Establish Counting Criteria:** The next step in designing a metric is to break the model down into its lowest level metric primitives and define the counting criteria used to measure each primitive. This defines the mapping system for the measurement of each metric primitive.

- **Decide what is good:** The next in designing a metric is defining what is good. Once they have decided what to measure and how to measure it, then they have to decide what to do with the results. Is 10 too few or 100 too many? What do the metrics say about whether or not the product is ready to ship?

- **Metrics report:** The next step is to decide how to report the metric. This includes defining the report format, data extraction and reporting mechanisms and distribution and availability.

- **Additional Qualifiers:** The final step in designing a metric is determining the additional metric qualifiers. A good metric is a generic metric. That means that the metric is valid for an entire hierarchy of additional extraction qualifiers. The additional qualifiers provide the demographic information needed for various views of the metric. The main reason that the additional qualifiers need to be defined as part of the metrics design is that they determine the second level of data collection requirements.
1.6.2 Classification of software metrics

Software metrics can be defined on the basis of:

- Their use in the software development life cycle
- Their time of evaluation

1.6.2.1 On the basis of their predictions in SDLC

For the software metrics, there are 3 types: procedure metrics, project metrics and product metrics

a) Procedure Metrics

Procedure metrics describe the effectiveness and quality of the processes that produce the software product. Process metrics are used to measure the properties of the process which is used to obtain the software. Process metrics include the cost metrics, efforts metrics, and advancement metrics and reuse metrics. Process metrics help in predicting the size of final system & determining whether a project on running according to the schedule. Procedure metrics lay emphasize on the procedure of the software development. It is mainly focus on how long a procedure last, how about the cost, where the methods used are effective, how is the result of comparing this method with alternative methods, etc. it includes the improvement of the procedure and the prediction for the future procedure. Procedure metrics is fulfilled within whole organization. The main part of this metrics includes maturity, management, life cycle, product ratio, defect ratio, etc. this metrics is used by high level managers to obtain the development status. It is beneficial to the control and management of the whole development procedure. Software process metrics measure the software development process such as number of man hours charged to the development activities in the design and coding phases. Examples are:

- Efforts required in the process
- Time to produce the product
- Effectiveness of defect removal during development
- Number of defects found during testing
- Maturity of the process [34]
b) Project Metrics:
Project metrics describe the project characteristics and execution. Project metrics is to understand and control the project/situation and status. The metrics are normally accomplished for a specific project and it includes scale, cost, workload, status, production power, risk, the degree of satisfaction from clients, etc. Project metrics is mainly to adjust the project to avoid the problems or risk and help to optimize the development plans. And thereafter project metrics is to improve the quality of the product via advances in technique method and management strategies. Examples are:
✓ Numbers of software developers
✓ Staffing pattern over the life cycle of the software
✓ Cost and schedule
✓ Productivity

e) Product metrics:
Product metrics describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability etc. Product metrics are also known as quality metrics and is used to measure the properties of the software. Product metrics includes product non reliability metrics, functionality metrics, performance metrics, usability metrics, cost metrics, size metrics, complexity metrics and style metrics. Products metrics help in improving the quality of different system component & comparisons between existing systems. Product metrics is to understand and control the quality of product. It is used to predict and manipulate the quality of product. Centering on the quality of the product, the metrics mainly include the reliability, maintainability, product scale, software complexity, portability, documents, etc. Product metrics is to measure the medium or the final product from the phase of system requirement to the phase of system maintainability. Software product metrics measure software products such as source code or design documents [34].

1.6.2.2 On the time of their evaluation
With this respect the metrics can be divided into two categories:
✓ Static Metrics
✓ Dynamic Metrics
a) **Static Metrics:**

Static metrics are those metrics whose values can be calculated by reading the code. They do not consider the impact of the environment. In current software development practice, this process relies almost entirely on the expertise of the software engineers, who work directly with the raw material of software—the source code—and perhaps with supplements such as design documents and UML diagrams. Software metrics, for instance, are little used in mainstream software development practice, with some exceptions such as counting Lines of Code (LOC). It might be argued that software metrics and other static analysis measures are not more widely used because they have little to contribute to engineer’s decision-making. While this argument reflects current practice, it is not a convincing limitation. Although most of the decision-making process of a software designer cannot be automated, static analysis tools can aid the designer by providing relevant information that would otherwise have to be gleaned manually from the program, or might even have gone unnoticed. Metrics and visualizations can illuminate software neighborhoods at appropriate levels of detail.

![Classification of software Metrics](image)

Fig 1.4: Classification of software Metrics
b) Dynamic metrics:

Software metrics measure different aspects of software complexity and therefore play an important role in analyzing and improving software quality. Previous studies have indicated that they provide useful information on external quality aspects of software such as its maintainability, reusability and reliability, and provide a means of estimating the effort needed for testing. Traditional metrics for measuring software such as Lines of Code (LOC) have been found to be inadequate for the analysis of object-oriented software. In recent years many researchers and practitioners have proposed a number of code metrics for object-oriented software, for example, the suite of metrics proposed by these code metrics quantify different aspects of the complexity of the source code, based on a static analysis of that code. However, the ability of such static metrics to accurately predict the dynamic behavior of an application is as yet unproven. Static metrics alone may be insufficient in evaluating the dynamic behavior of an application is as yet unproven.

Static metrics alone may be insufficient in evaluating the dynamic behavior of an application at run time, as its behavior will be influenced by the operational environment as well as the complexity of the source code. Research by [33] has indicated that useful information may be obtained from a measure of qualifying the dynamic complexity of software in its operational environment. However, only considered the use of such metrics in the design stage of the software lifecycle. Software quality is an important external software attribute that is difficult to measure objectively. Several studies have identified a clear empirical relationship between static coupling metrics and software quality [34].

1.6.3 ADVANTAGES OF SOFTWARE METRICS

Software metrics have various advantages such as:

- In Comparative study of various design methodology of software systems.
- For analysis, comparison and critical study of various programming language with respect to their characteristics
- In comparing and evaluating capabilities and productivity of people involved in software development.
- In the preparation of software quality specifications.
- In the verification of compliance of software systems requirements and specifications.
In making inference about the effort to be put in the design and development of the software systems.

In getting an idea about the complexity of the code.

In taking decisions regarding further division of complex module is to be done or not.

In providing guidance to resource manager for their proper utilization.

In comparison and making design tradeoffs between software development and maintenance cost.

In providing feedback to software managers about the progress and quality during various phases of software development life cycle.

In allocation of testing resources for testing the code [34].

1.6.4 LIMITATION OF SOFTWARE METRICS:

The application of software metrics is not always easy and in some cases it is difficult and costly.

The verification and justification of software metrics is based on historical/empirical data whose validity is difficult to verify.

These are useful for managing the software products but not for evaluating performance of the technical staff.

The definition and derivation of Software metrics is generally based on assuming which are not standardized and may depend upon tools available and working environment.

Most of the predictive models rely on estimates of certain variables which are often not known exactly [34].

1.7 Object Oriented Metrics

A metric is defined as the measurement of a particular characteristic of a program's performance or efficiency.

Various metrics according to the variation in classes are as follows:
• COHESION METRICS:

In object oriented design cohesion is an intra-module concept. i.e. the degree to which the methods within a class are related to one another. It focuses on why elements of a module are together in same module. The objective is that cohesion must be high and the interaction with the elements of module must be high. This makes the system more understandable and modifiable. Generally higher cohesion lead to lower coupling as many elements that need to interact a lot will reside together in strongly coupled modules, lessening the needs for interaction with other modules. On the other hand, modules that have low cohesion will often need to interact with other modules to perform their tasks. Making a system more understandable and modifiable, it should consist of the modules that are highly cohesive. Various cohesion metrics are:

1. Lack of cohesion in methods (LCOM):

LCOM measures the dissimilarity of methods in a class by instance variable or attributes. Cohesion is the similarity of two things to be the intersection of the sets of properties of the two things. So, the degree of similarity of the methods within the object can be defined to be the intersection of the sets of instance variables that are used by the methods.

If an object class has different methods performing different operations on the same set of instance variables, then the class is said to be cohesive. Cohesion is centered on data that is encapsulated within an object and how methods interact with data. Lack of cohesion, is a measure of how close methods are to the data they access.

The more methods that access each attribute, the lower the value of LCOM and the more cohesive the class

Cohesion is the degree to which methods within a class are related to one another and work together to provide well bounded behavior.

• LCOM uses variable or attributes to measure the degree of similarity between methods.

• The cohesion for each data field in a class can be measured.

• The percentage of methods that use the data field can be calculated.

• Average the percentage, then subtract from 100 percent. Lower percentage indicates greater data and method cohesion within the class.

• High cohesion indicates good class subdivision while a lack of cohesion increases the complexity.
II.  Tight class cohesion (TCC)
III. Loose class cohesion (LCC)
IV.  Information-f low -based cohesion (ICH)
V.  Cohesion among methods of classes (CAMC)
VI. Normalized Hamming distance based cohesion (NHD)
VII. Scaled NHD (SNHD) [35] [48]

• COUPLING:

It is an inter-module concept, which captures the strength of interconnection between modules. If the modules will be more tightly coupled, they will be more dependent on each other, and the more difficult is to understand and modify them. Low coupling is desirable for making system more understandable and modifiable. The degree of coupling between a module and another module depends on how much information is needed about the other module for understanding and modifying this module, and how complex this information is. Low coupling occur when this information is as little as possible, as simple as possible, and is easily identifiable. Various coupling metrics are:

i. Coupling between classes (CBO)
   • Coupling is a measure of strength of association established by a connection from one entity to another.
   • Classes are couple in three ways. One is, when a message is passed between objects, the object are said to be coupled. Second one is, the classes are coupled when methods declared in one class use methods or attributes of the other classes. Third on is, inheritance introduced significant tight coupling between super class and subclass.
   • 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 class hierarchy on which a class depends.
   • Excessive coupling is detrimental to modular design and prevent reuse. If the number of couple is larger in software than the sensitivity to changes in other in other parts of design.
ii. **Response set for classes (RFC)**

RFC is the count of set of all methods that can be invoked in response to a message to an object of the class or by some method in the class. Methods in a class can be used as measures of communication.

\[ \text{RFC} = \vert \text{RS} \vert \] where RS is the response set for the class.

\[ \text{RS} = \{M\} \cup \{R_i\}, \] where

\[ M = \text{set of all methods in the class.} \]

\[ R_i = \text{set of methods called by method } i. \] The cardinality of this set is a measure of communication.

A message is a request that an object makes to another object to perform an operation. The operation executed as a result of receiving a message is called a method.

- The RFC is the total number of all methods within a set that can be invoked in response to message sent to an object. This includes all methods accessible within the class hierarchy.
- This metrics is used to check the class complexity. If the number of method is larger that can be invoked from class through message than the complexity of the class is increase.

iii. **Message passing coupling (MPC)**

iv. **Data abstraction coupling (DAC)**

v. **Information-f low -based coupling (ICP)**

vi. **Coupling based on inheritance (CBI)** [36] [48].

**DISADVANTAGES OF COUPLING**

Various disadvantages of tightly coupled are as follows:

i. A change in one module usually forces a ripple effect of changes in other modules.

ii. Assembly of modules might require more effort and/or time due to the increased inter-module dependency.

iii. A particular module might be harder to reuse and/or test because dependent modules [36].
• **INHERITANCE METRICS**

One of the important characteristic of the OO system is inheritance. Inheritance is the ability of one class to acquire the properties of another class. The basic idea behind inheritance was reusability of code i.e. there is no need to write the same code again and again. Once the method is defined in a super-class, that method is automatically inherited by all subclasses. Thus, once a method is written it can be used by all subclasses. Once a set of properties are defined in a super class, the same set of properties are inherited by all subclasses. A class and its children share common set of properties. A subclass only needs to implement the differences between it and the parent Inheritance is a key feature of the OO paradigm. This mechanism supports the class hierarchy design and captures the IS-A relationship between a super class and its subclass [37].

The use of inheritance is claimed to reduce the amount of software maintenance necessary and ease the burden of testing and the reuse of software through inheritance is claimed to produce more maintainable, understandable and reliable software. Various inheritance metrics are:

i. **Depth of inheritance (DIT):**

Depth of Inheritance (DIT) = depth of the class in the inheritance tree. The depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree.

- Depth of inheritance indicates the extent to which the class is influenced by the properties of its ancestors.

- Inheritance is a type of relationship among classes that enables programmers to reuse previously defined object objects, including variables & operators.

- Inheritance decrease the complexity by reducing the number of operations and operators, but this abstraction of objects can make maintenance and design more difficult.

- Depth of class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes.

- The deeper a class within the hierarchy, the greater the number of methods and is likely to inherit, making it more complex to predict its behavior.

- A support metric for DIT is the number of methods inherited.
ii. **Number of children (NOC):**

Number of children = Number of immediate descendants of the class. Number of children indicates the potential impact on descendants.

- The number of children is the number of immediate subclasses subordinates to class in the hierarchy.
- The greater the number of children, the greater the parent abstraction.
- The greater the number of children, greater the reusability, since the inheritance is a form of reuse.
- If the number of children in class is larger than it require more testing time for testing the methods of that class [48].

iii. **Weighted Methods Per Class (WMC):**

- WMC is a count of methods implemented within a class or the sum of the complexities of the methods. Method complexity is measured by cyclomatic complexity.
- This metric is used to measure the understandability, reusability and maintainability.
- A class is a template from which objects can be created. Classes with large number of methods are likely to more application specific, limiting the possibility of reuse.
- This set of objects shares a common structure and a common behavior manifested by the set of methods.
- The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods. But the second measurement is more difficult to implement because not all methods are accessible within the class hierarchy because of inheritance.
- The larger the number of methods in a class is the greater the impact may be on children, since children inherit all of the methods defined in a class.

iv. Number of parents (NOP)
v. Average inheritance depth of a class (AID)
vi. Class-to-leaf depth (CLD)
vii. Number of descendants (NOD)
viii. Number of ancestors (NOA)
ix. Number of methods overridden (NMO)
x. Number of methods inherited (NMI)
xi. Number of methods added (NMA) [37] [48] [5].

* SIZE METRICS

Size has been used as software metric for a long time. Various size metrics are:

i. The number of attributes in a class, both inherited and no inherited (NA)
ii. The number of attributes in a class, excluding inherited ones (NAIMP)
iii. The number of methods in a class, both inherited and no inherited (NM)
iv. The number of methods implemented in a class (NMIMP)
v. The sum of the number of parameters of the methods implemented in a class (Num Para)
vi. The number of declaration and executable statements in the methods of a class (Stmts)
vii. The non-commentary source lines of code in a class (SLOC) [34] [48].

1.9 PROBLEM DEFINITION

This thesis analyses the behavior of classes with respect to change. It studies the change prone behavior of classes from various aspects. The change in classes may be reflected through the change in their levels of cohesion, coupling, size and inheritance. This research analyzes frequency of change of classes so as to determine particular categories, in which most of the change prone classes lie. Another aspect of study is to understand the pattern of changes in classes across the software releases.

1.8 SIGNIFICANCE OF THE WORK

In OO software systems, identifying and characterizing which classes are change-prone can be very useful and helpful in guiding the maintenance team, distributing resources more efficiently, and thus, enabling project managers and their teams to focus their effort and attention on the change-prone classes during the software evolution process. For example, refactoring can be focused on change-prone classes to improve their quality and make them more flexible to accommodate future changes and to localize the impact of changing them.
Inspection and testing activities can also be focused on such classes, and thus enable efficient allocation of valuable resources.

In an object-oriented (OO) system, change-proneness is an important external quality attribute that denotes the extent of change of a class across the releases of the system. It not only can help software developers to take focused preventive actions to reduce maintenance costs and improve quality, but can also help software managers to allocate resources more effectively.

This thesis attempts to identify the patterns, if any, in the changeproneness of classes. Understanding the patterns of change may help the developers to predict the future change cycles in the software life cycle. Metrics based analysis of changeprone classes can help them to prevent the classes from attaining metric values above the thresholds.

1.9 OBJECTIVE OF STUDY

The objective of thesis is to analyze the impact of languages on change-proneness. i.e. how the classes change from one language to other in different releases of a program. (what to write in it)

1.10 METHODOLOGY

In this analysis tool Analyst4J will be used and various metrics such as cohesion, coupling, size and inheritance are used in order to analyze the change-proness of classes. The various classes are extracted from the different releases of JAVA program. The program is developed in VB.NET in order to compare the different text files of classes and to obtain the result of changed classes and unchanged classes.

**Step 1**: Collect the metrics using a metric collection tool then study change in classes.

**Step 2**: Create the different metrics file according to change in those classes i.e. either according to size, cohesion, coupling and inheritance in different releases of a program.
The output of the analysis tool will be like:

<table>
<thead>
<tr>
<th>Class</th>
<th>Size1</th>
<th>Size2</th>
<th>Cpl1</th>
<th>Cpl2</th>
<th>Coh1</th>
<th>Coh2</th>
<th>Inh1</th>
<th>Inh2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>12</td>
<td>24</td>
<td>56</td>
<td>24</td>
<td>67</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>B</td>
<td>16</td>
<td>18</td>
<td>20</td>
<td>22</td>
<td>26</td>
<td>28</td>
<td>11</td>
<td>22</td>
</tr>
<tr>
<td>C</td>
<td>13</td>
<td>33</td>
<td>23</td>
<td>57</td>
<td>09</td>
<td>04</td>
<td>05</td>
<td>21</td>
</tr>
<tr>
<td>D</td>
<td>97</td>
<td>56</td>
<td>43</td>
<td>41</td>
<td>23</td>
<td>12</td>
<td>11</td>
<td>17</td>
</tr>
</tbody>
</table>

**Fig. 1.6: Output of analysis tool**

**Step 3:** Compare the different metrics files and check their behavior in different versions i.e., how the change or whether they remain same.

**Step 4:** Find the difference between all the classes in different releases with the help of difference checker tool.

**Step 5:** Find the changed, unchanged, newly added, and removed classes in each version.

**Step 6:** Find the average gap in changes across the releases.

**Step 7:** Find correlation between the metrics and changed classes.

**Step 8:** Check the frequency of change. How many times a class change and why it change.
1.13 Flow chart of the work to be done is as:

Fig.1.7: Flow Chart
CHAPTER 2: EXISTING STUDIES

2.1. INTRODUCTION TO CHANGE-PRONENESS

Change Proneness of a single class of an object oriented software or the entire software as a whole is the likeliness that changes would be made to that class or software respectively, either in the same release or across multiple releases. When we say that a class has high change proneness, we actually mean that the class is more likely to get changed than other classes. The code volatility or change proneness will affect change effort, and that change prone components require less effort, because developers are more experienced with changing these components. While, changes to infrequently changed components represent unfamiliarity, indicating more fundamental changes. Higher code volatility could lead to increased change effort, because frequently changed modules may experience code decay [11].

In an object-oriented (OO) system, change-proneness is an important external quality attribute that denotes the extent of change of a class across the releases of the system (Koru and Tian 2005; Arisholm et al., 2004). It not only can help software developers to take focused preventive actions to reduce maintenance costs and improve quality, but can also help software managers to allocate resources more effectively. In the last two decades, many efforts have been made to investigate the ability of OO metrics to predict change-prone classes (Koru and Tian 2005; Arisholm et al. 2004; Lindvall, 1998; Koru and Liu 2007; Li and Henry 1993; Lindvall, 1999; Bieman et al. 2003a, b, 2001; Di Penta et al., 2008). Li et al. (1993) found that there was a strong correlation between OO metrics and change-proneness. In particular, they reported that the change-proneness prediction model building from OO metrics had a good performance. Lindvall, 1998, has reported that large classes were more change-prone.

Arisholm et al., 2004, has reported that dynamic coupling metrics were significant indicators of change-proneness. Koru and Tian, 2005, has investigated whether the classes with the highest measurement values were the most change-prone classes. They first classified structural metrics into size, coupling, cohesion, and inheritance metrics. Then, they analyzed whether high-change classes were large-size, highly coupled, low-cohesion, or high-inheritance classes. Consequently, they found that top-change classes did not have the highest measurement values but they immediately followed the classes with the highest measurement values in the measurement rankings. Overall, previous studies conclude that OO metrics are able to predict change-proneness. However, there is a need to re-examine the relationships between OO metrics and change-
proneness for two reasons. First, most previous studies only analyze a small number of OO metrics, although a large number of OO metrics have been proposed in the last decade. Therefore, it is not clear whether the previous conclusion is applicable to most, if not all, OO metrics. Second, most studies only uses one to eight systems to investigate the relationships between OO metrics and change-proneness. Consequently, it is not clear whether previous conclusion could be generalized to other systems. In this paper analysis tool analyst4j will be used and various metrics such as cohesion, coupling, size and inheritance are used in order to analyze the change-proness of classes. The various classes are extracted from the different releases of c++ program. The program is developed in VB.NET in order to compare the different text files of classes and to obtain the result of changed classes and unchanged classes.

2.2. REASONS OF CHANGE PRONNESS

During software evolution, adaptive, and corrective maintenance are common reasons for changes. Changes can be due to a variety of reasons such as enhancements, adaptation, perfective maintenance or fixing defects. Some parts of the software may be more prone to changes than others. Knowing which classes are change-prone can be very helpful.

Change proneness of a given class depends on various factors like, if the same class in previous versions underwent change, and then the class will have higher change proneness. Similarly change proneness of a given class also depends on the fault proneness, i.e. if a given class gives rise to faults (higher fault proneness), then it will have to be changed. In other words change proneness is directly in relation with fault proneness.

Change-proneness may indicate specific underlying quality issues. If a maintenance process can identify what parts of the software are change-prone then specific remedial actions can be taken. Thus, knowing where most changes are made over time can identify key change-prone classes, key change-prone interactions, and the evolution process can focus attention on them.

Change-proneness also depends upon the extent of coupling among the classes, i.e. if a system consists of classes that are highly coupled (dependant on each other by using each other’s member functions) the change in one class leads to change on another class which is dependent on the class under consideration.

Change-prone class can vary widely due to following reasons:
There may be specific quality problems due to code decay (if changes are due to defect repair).

The underlying architecture may have problems (again, if changes are due to defect repair)

Encourage changes in certain classes (to add functionality or adaptations).

Regardless, it is useful to know what these key classes and interactions are so that they get proper attention.Actions may range from stepped up quality assurance efforts to refactoring depending on the underlying cause for change-proneness [11].

2.3. IMPACT OF CHANGE-PRONNESS ON SOFTWARE QUALITY

Knowing Changed classes is beneficial. Change-proneness may indicate specific underlying quality issues. If a maintenance process can identify what parts of the software are change-prone then specific remedial actions can be taken. If changes are due to defect repair, the underlying architecture may suffer problems. Actions may range from stepped up quality assurance efforts to re-factoring depending on the underlying cause for change-proneness. Changes in response to a single defect report or change request may be local to a class and involve changes in only that single class, or may involve a whole collection of classes. We thus are not only interested in identifying and making visible classes that see the most lines of code changed, but, identifying and visualizing classes that experience frequent changes together. That is, showing changes in these classes that are related. This is termed as change-coupling between classes.
2.4. Existing Studies:

**Study 1: Li and Offutt (1996)** proposed a set of algorithms that determine what classes are affected by a given change. The methodology represents systems as a set of data dependency graphs, which is a reasonable and effective approach for object-oriented designs. However, as in any change impact model, reports about the potential impact of a given change can be generated only after the user explicitly specifies the changes. They analyse the possible changes that could happen in object-oriented software, how these changes affect other classes in the system, and describes a set of algorithms that can find out all the possibly affected classes if these changes happened. This technique allows software developers to perform “what if” analysis on the impact of proposed process changes in the object-oriented system, choose the proposed change that causes the minimum impact on the system. Once the change is committed, it allows software testers to know what the areas are in the software system that are possibly affected by the change and retest only those classes instead of whole system. Algorithms are presented that calculate change propagation within classes, between client and server classes, and between parent and children classes. They concluded that it can be used by software testers to find what areas are affected by the changes, so they can test only the affected areas and still feel confident about the quality of the software [1].

**Study 2: Lindvall (1999)** has used commercial c++ system and two releases R3 and R4 of industrial object oriented system PMR. Data was collected as a post development analysis of source code in two releases R3 and R4 of the industrial object oriented system PMR. Their study was to understand what constructs are stable and remain unchanged and what constructs are change-prone and change frequently. He used inheritance, size and other software metrics and used c++ and SYBASE language in order to design a comparator too analyze which classes are changed and which remains same. He used 114 classes in R3 in transition to R4 none of the classes were deleted. Four new classes were added making 118 classes in R4. He concluded that large classes were more change prone [2].

**Study 3: Emam and Melo (1999)** predict the faulty classes using object oriented design metrics. They used data collected from one version of a commercial Java application for constructing a prediction model. The model was then validated on a subsequent release of the same application. In addition to identifying the faulty classes, the model can be applied to give an overall quality estimate. The model uses only object-oriented
design metrics. They used 24 metrics Chidamber and Kemerer. They consider two versions of this java application: versions 0.5 and 0.6. Version 0.5 had a total of 69 classes. Version 0.6 had 42 classes. They concluded that most field faults in software applications are found in a small Percentage of the software’s components. This means that if these faulty software components can be detected early in the development project’s life cycle, mitigating actions can be taken, such as a redesign. They also indicate that the prediction model has a high accuracy. Furthermore, they found that an export coupling metric had the strongest association with fault proneness.

**Hypothesis:** They hypothesise a relationship between the object-oriented metrics and fault-proneness due to the effect on cognitive complexity. It is hypothesized that the structural properties of a software component have an impact on its cognitive complexity. Cognitive complexity is defined as the mental burden of the individuals who have to deal with the component, for example, the developers, testers, inspectors, and maintainers. High cognitive complexity leads to a component exhibiting undesirable external qualities, such as increased fault-proneness and reduced maintainability. Therefore, their general hypothesis is that the metrics that they validate are positively associated with the fault-proneness of classes. This means that higher values on these metrics represent structural properties that increase the probability that a class will have a fault that causes a field failure [3].

**Study 4: Girba T et al. (2000)** present an approach, named Yesterday Weather (YW), for predicting change-prone classes. Their approach is based on summarizing the changes in the evolution of OO software system by defining historical measurements. Their basic idea is that classes that changed the most in the recent past will also suffer important changes in the near future. To measure the changes in the history of class, they measure the differences in its number of methods in different releases, i.e. the change is defined only in terms of the addition or removal of the class methods. They showed that YW approach can summarize the changes in the history of the system.

Their aim was to measure how relevant it is to start reverse engineering from the parts of the system which changed the most in the recent past. They used 40 releases of a software system’s code, each version consisting on average of ca. 400 classes. They analyze ca. 16000 classes, i.e., class releases. They concluded that big changes can occur in those classes that are not big in terms of size (i.e. number of methods) [4].
Study 5: Beyer et al (2001) had shown the impact of inheritance on metrics for size, coupling, and cohesion in OO systems. They use crocodile tool and 200 object oriented metrics including size, cohesion, coupling and inheritance. They concluded that the values didn’t change for 39 classes in case of size because they have not any super-class. The main result of these measurement changes is that attribute-based cohesion decreases heavily for classes having at least one superclass. The opposite is in case increase of cohesion. In case of coupling, only the measurement value of one class that has a super-class did not change [5] [37].

Study 6: Arisholm(2001) presents the Empirical assessment of changeability in object oriented software. He used Genova tool and proposed two alternative measurement approaches, termed change profile (CP) and structural attribute (SA) measures. They conclude that that some of the CP measures are statistically significant explanatory variables of change effort, whereas none of the SA measures are [6].

Study 7: Ohlsson et al (2001) defined a case study on Modelling fault-proneness statistically over a sequence of releases. They reported that historical data available primarily on corrective maintenance, they apply the method to four releases of a system consisting of 130 components. In each release, components are classified as fault-prone if the number of defect reports written against them is above a certain threshold. The outcome from the case study shows stabilizing principal components over the releases, and classification trees with lower thresholds in their decision nodes. Also, the variables used in the classification trees’ decision nodes are related to changes in the same files. The discriminant functions use more variables than the classification trees and are more difficult to interpret. They used PCA classification trees, discriminant analysis and box plots as parts of models. A classification tree is a graphical representation of an algorithm to classify components. Discriminant analysis is another multivariate statistical method for classification. The main purpose is to predict group membership, or to identify boundaries between objects, based on a linear combination of interval variables. Box plots are good for depicting the dispersion and skewedness of samples, i.e. the distribution of a variable

PCA involves a mathematical procedure that transforms a number of correlated variables into some number of groups called principal components or factors. Factors are ranked based on the amount of variability in the data for which each factor accounts. Their goal was to combine PCA, the use of classification trees, discriminant analysis and box plots to track software evolution. Overall the results show that: a) the number of
changes to unique files increases; b) more components needed to be changed to fix problems, thus the degree of interaction and relationships between components play a major part in decay for this system; and c) the size of a change slightly decreases, thus it is not any large obvious omission that is the problem, but subtle problems that involve multiple components. The results show that for a context of corrective maintenance, principal components analysis together with classification trees are good descriptors for tracking software evolution [7].

**Study 8: Bieman et al (2001)** presents an industrial case study on design patterns, design structure and program changes. They analyze 39 releases of an evolving industrial OO software system implemented in c++ to see if there is a relationship between patterns, other design attributes, and the number of changes. Their focus has been on the transformations between two specific releases of the system: version A, which is the first stable version of the system, and version B, which is the final version in our data set. Version A consists of 199 classes and approximately 24,000 lines of source code. Version B has 227 classes with approximately 32,000 lines of code. Of the 199 classes in Version A, 191 also appear in Version B. The 191 classes that appear in both Version A and Version B are the focus of their study. They found a strong relationship between class size and the number of changes, and resulted that larger classes were changed more frequently.

They also give three hypotheses:

**H1**: classes that participate in design patterns are not less change prone-- these pattern classes are among the most change prone in the system, and

**H2**: classes that are reused the most through inheritance tend to be more change prone.

**H3**: Larger classes will be more change prone [8].

**Study 9: Emden and Moonen (2002)** present the java quality assurance by detecting code smells. Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices. Originally, code smells are used to find the places in software that could benefit from refactoring. They investigate how the quality of code can be automatically assessed by checking for the presence of code smells and how this approach can contribute to automatic code inspection. They present an approach for the automatic detection and visualization of code smells and discuss how this approach can be used in the design of a software inspection tool. They illustrate the feasibility of their approach with the development of
jCOSMO, a prototype code smell browser that detects and visualizes code smells in JAVA source code. They used C analyzer LINT tool that support automatic code inspection tend to focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. They focused on a different aspect of code quality: Inspired by the metaphor of code smells introduced in the refactoring book. They reviewed the code for problems that are generally associated with bad program design and bad programming practices. Their main focus was on java programming language. They distinguished three classes of users for the detection results:

a. Programmers that use detected smells during development or maintenance of a system to improve the code.

b. Code inspectors (or reviewers) that use detected smells to assess the quality of the code.

c. Tools that use the detected smells to perform further analysis or transformations on the code, for example, software refactoring tools. Generally, these tools do not need specific presentations; they just use the repository content for further processing. They concluded that there were clusters of identical typecasts in particular areas of the system which suggests that a small amount of refactoring could remove a large number of these smells [9].

**Study 10: Chaumun et al (2002)** proposed a change impact model for changeability assessment with primary goal to investigate the relationship between existing design metrics (e.g., Weighted Methods per Class) and the impact of change. However, even if it is useful to know which classes would be impacted from a given change, one has to know the actual changes that occurred in a system, in order to assess the probability of change for a certain class. They assume that high-level design has an influence on maintainability is carried over to changeability and investigated for that quality characteristic. The approach taken to assess the changeability of an object-oriented (OO) system is to compute the impact of changes made to classes of the system. A change impact model is defined at the conceptual level and mapped on the C++ language. In order to experiment the model as a changeability indicator on large industrial software systems, an experiment involving the impact of one change is carried out on a telecommunications system. Their results showed that the method signature change is a change that the system under test can absorb quite easily since on the average, at most only one class was impacted per method signature change. Another outcome of the experiment led to the establishment of a relation between WMC, a design metric, and the mean change impact of a class. The higher the WMC
value was the higher was the mean change impact. Since the resultant correlation is somewhat weak, they plan to investigate the relation on further systems [10].

**Study 11: Bieman et al (2003)** use the OO software to understand the change-proneness through visualization. They collect data on the transformations between 39 releases.

Their focus has been the transformations between two specific releases of the system: version A, which is the first stable version of the system, and version B, which is the final version in their data set. Version A consists of 199 classes and approximately 24,000 lines of source code. Version B has 227 classes with approximately 32,000 lines of code. Of the 199 classes in Version A, 191 also appear in Version B. The 191 classes that appear in both Version A and Version B and all of the transformations between the releases are the focus of their study.

They use size metrics and use commercial OO system implemented in C++. They use COBOL language in order to design a comparator to see which classes are more change-prone and their relationship with the help of UML diagrams. They conclude that class is considered to be change-prone if each of the change-prone measures falls above the threshold [11].

**Study 12: Alshayeb and Li (2003)** present an empirical validation of object-oriented metrics in two different iterative software processes. They investigated the effectiveness of a set of OO product metrics in predicting design efforts and source lines of code added, changed, and deleted in the short-cycled agile process and in the long-cycled framework process.

They used three sets of data for this research: one for the framework evolution process and two for an XP-like process. In the first data set, they collected four major JDK releases ranging from JDK 1.0 through JDK 1.4. In the other two data sets, they collected system evolution data from two client-server systems over a 3-year period. The two systems were developed primarily using Java, Java Servlets, and XML. The two systems were delivered as commercial products to customers. Both systems were developed using an agile process similar to XP with several short iteration cycles and customer-supplied stories. Data from the two systems collected from research logs. The research logs of the two systems recorded daily activities. A log file was created for each working day during system development. The information recorded in each log file includes:
The tasks planned for the current iteration cycle. The tasks come from the user stories, which are supplied by the customer.

A description of the progress or failure of progress towards completing the tasks.

The time spent on each task measured by hours.

A description of the problems encountered during implementing a task and the attempted solutions.

A description of the changes made on the system, the reasons for the changes, and the affected classes.

They set up the following hypotheses to focus our study and to facilitate statistical analyses:

**Hypothesis 1:** Using OO metrics, they can predict added, changed, and deleted source lines of code in classes from one iteration (release) to the next in the long-cycled framework iterative process.

**Hypothesis 2:** Using OO metrics, they can predict added, changed, and deleted source lines of code in classes from one iteration to the next in the short-cycled XP iterative process.

**Hypothesis 3:** Using OO metrics, they can predict maintenance effort (measured in man-hours) in classes from one iteration to the next in the short-cycled XP iterative process.

**Hypothesis 4:** Using OO metrics, they can predict refactoring effort (measured in man-hours) in classes from one iteration to the next in the short-cycled XP iterative process.

To test the hypotheses, they used the MLR statistical technique, which investigates the relationship between a dependent variable and multiple independent variables. They conclude that OO metrics are effective in predicting design efforts and source lines of code added, changed, and deleted in the short-cycled agile process and ineffective in predicting the same aspects in the long-cycled framework process [12].

**Study 13: Bieman et al (2003)** analysed four small and one large system to study pattern change proneness. They studied five systems, three proprietary systems and two open source systems, to identify the observable effects of the use of design patterns in early versions on changes that occur as the systems evolve. Design patterns promote reuse of solutions to recurring design problems by naming and cataloging these solutions. Design structure is characterized by class-size, and class participation in inheritance relationships and design patterns. Changes are measured in terms of a count of the number of times that a class is modified over a period of time. Their specific objective is to see how design patterns are applied to real software development project. They describe 23 patterns that solve common software design problems. The initial study was
conducted on a commercial OO system implemented in C++. The system was developed with the support of a version control system over a period of several years. They studied a total of 39 versions of the system including versions A and B, and all of the intermediate versions. Both the systems are implemented in java. In system A, Version 2, the base version for design analysis contains 384 classes and around 23,000 lines of code. They examined 17 versions of System A. In system B, Version 2, the base system for analysis, contains 101 classes and around 7,500 lines of code. They examined 17 versions of System B. They used JRefactory to restructure java programs. They resulted that in four of the five systems, pattern classes are more rather than less change prone. Pattern classes in one of the systems were less change prone. These results held up after normalizing for the effect of class size. Larger classes are more change prone in two of the five systems. These results provide insight into how design patterns are actually used, and should help them to learn to develop software designs that are more easily adapted.

They tested following five hypothesis on five case studies:

**H1:** Larger classes will be more change prone. A larger class has more functionality, thus there is a greater likelihood that some functionality in the class will need to be corrected or enhanced.

**H2:** Classes participating in design patterns are less change prone. Patterns are designed so that changes are made via subclasses or by adding new participant classes rather than modifying already present classes. Patterns promote ease of change; hence the classes participating in patterns should require fewer changes [13].

**Study 14: Koru and Tian (2005)** identified and compared high change modules and modules with the highest measurement values in two large-scale open-source products. They collected 67 class-level metrics which contain 16 size, 28 coupling, 11 cohesion and 12 inheritance metrics. They use UNIX operating system and Columbus tool to extracts structural measures from Source code. They use PERL language to develop a program to calculate the change count for each count. They conclude that the classes with the highest measurement values were more change-prone classes [14].

**Study 15: Gyimothy et al (2005)** compared the capability of sets of Chidamber and Kemerer metrics to predict fault-prone classes within Mozilla, using logistic regression and other machine learning techniques, for ex. artificial neural networks. They calculated the object-oriented metrics given by Chidamber and Kemerer to illustrate how fault-proneness detection of the source code of the open source Web and e-mail suite called
Mozilla can be carried out. They checked the values obtained against the number of bugs found in its bug database—called Bugzilla—using regression and machine learning methods to validate the usefulness of these metrics for fault-proneness prediction. They also compared the metrics of several versions of Mozilla to see how the predicted faultproneness of the software system changed during its development cycle. They investigated eight metrics. They collected the number of bugs found and corrected in each class of the system from the Bugzilla database, which contains all the bugs that arose during the development of Mozilla. They analysed the source code of Mozilla with the help of reverse engineering framework called Columbus, which they also used to calculate the required metrics. The Columbus framework has been further improved recently with a so-called compiler wrapping method. This allows automatically analysing and extracting information from practically any software system that compiles with GCC on the GNU/Linux platform. They concluded that CBO is the most discriminating metric. They also found LOC to discriminate fault-prone classes well [15].

**Study 16: Tsantalis et al (2005)** tried to quantify the change probability of each class in future releases to estimate the change-process of an OO design by assessing the probability of each class will change in future generation.

They use 58 classes, 13 subsequent releases, and 169 classes, 9 subsequent releases for Jflex and jmol tool respectively. Their goal of study was to check the validity of extracted probabilities of change and identify class with high probability of change. They use internal tool to extract internal probability of change for each class. They use java programming language, a Byte-code parser, an XML generator and probability calculator approaches. They proposed a metric that analyzes the change history to extract the internal probability of change for each class. For example, if a class has changed in one of four releases, the internal probability of change at the fifth release would be 25%. They use joint probability to estimate internal and external changes [16].

**Study 17: Gunes Koru and Liu (2007)** identified and compared high change modules and modules with the highest measurement values in two large-scale open-source products. They use two open-source projects, KOffice and Mozilla. They collected 67 class-level metrics which contain 16 size, 28 coupling, 11 cohesion and 12 inheritance metrics. They use UNIX operating system and Columbus tool to extracts structural measures from source code. The static metrics for Mozilla were obtained using the Columbus tool. They use
PERL language to develop a program to calculate the change count for each count. They conclude that the classes with the highest measurement values were more change-prone classes [17].

**Study 18: Sharafat and Tahvildari (2007 and 2008)** use probabilistic approach to predict changes in object-oriented software systems. They use the dependencies obtained from the UML diagrams, as well as other data extracted from source code of several releases of a software system using reverse engineering techniques. The proposed systematic approach has been evaluated on a multi-version medium size open source project namely JFlex, The Fast Scanner Generator for Java. Their goal was to predict the probability that each class will change in a future generation. They use 394 class metrics including coupling, size and cohesion metrics. They use JAVA, C++ and MATLAB to design a program to extract the metrics from source code. They find dependencies between the classes and then identify classes which are change prone. It improves the maintenance of object oriented systems and reduces the risk to deal with expensive and unpredictable changes. They concluded that large classes were more change prone [18].

**Study 19: Aversano et al (2007)** dealt with the changeability and resilience to change of design patterns and of specific pattern roles. The study has been performed on three Java software systems, JHotDraw, ArgoUML and Eclipse-JDT. First, they detected design patterns on different subsequent releases of the three systems by using JRefactory 2.6.24, which is a refactoring tool for the Java programming language. Then, they mined co-changes from Concurrent Versioning System (CVS) repositories to identify when a pattern changed, what kind of change was performed, which classes co-changed with the pattern, whether these classes had a dependency to or from the pattern, and what was the relationship between the type of change made and the resulting co-change. Specifically, the study analyses how frequently patterns are modified, to what changes they undergo and what classes co-change with the patterns. Results show how patterns more suited to support the application purpose tend to change more frequently and that different kind of changes have a different impact on co-changed classes and a different capability of making the system resilient to changes [19].

**Study 20: Penta and Cerulo (2008)** developed an exploratory study to analyse the change-proneness of design patterns and the kinds of changes occurring to classes involved in design patterns. They represent that
analysing the change-proneness of design patterns and the kinds of changes occurring to classes playing role in some design pattern during software evolution poses the basis for guidelines to help developers who have to choose, apply or maintain design patterns. They represented an empirical study to understand whether there are roles that are more change-prone than others and whether there are changes that are more likely to occur to certain roles. The study relies on data extracted from the source code repositories of three different systems: JHotDraw, Xerces, and Eclipse-JDT, and from 12 design patterns. JHotDraw is a Java framework for drawing 2D graphics. Xerces-j is a Java XML parser. Eclipse-JDT is a set of plug-ins that provides the capabilities of a full-featured Java IDE to the Eclipse platform. This set of plug-ins is entirely in Java and its CVS repository is divided into three types of releases: stable, integration, and nightly builds. The design motif identification was performed using the DeMIMA approach and tool. DeMIMA not only identifies the classes, but also the different roles played by classes in occurrences of motifs. They implemented their study on Adapter motif, Abstract Factory and Command and concluded that in the Adapter motif, classes playing the role of Adapter are more change prone. In a similar way, intuition is confirmed for the Abstract Factory and Command. For the former, Abstract Factories and Products change less frequently than concrete ones; for the latter, Receivers change more frequently than commands. Overall, the obtained results suggest that roles more subjects to changes, since their change proneness can make other parts of the system less robust to changes [20].

Study 21: Abdi et al (2009) used probabilistic approach for change impact prediction in object-oriented systems. They propose a probabilistic approach using Bayesian networks to analyze and predict change impact in OO systems. The experimentation was made on BOAP system. It contains 394 classes. The results of their empirical studies were useful for the graph structure construction (Bayesian Network). The built probabilistic model is tested on data extracted from a real system. The main motivation was to improve the maintenance of object oriented systems. By identifying the potential impact of a modification, they reduce the risk to deal with expensive and unpredictable changes [21].

Study 22: Foutse Khomh, et al (2009) presents Exploratory Study of the Impact of Code Smells on Software Change-Proneness. They use the code smells and investigate if classes with code smells are more change-prone than classes without smells. Specifically, they test the general hypothesis: classes with code smells are not more change prone than other classes.
They detect 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse, and study the relation between classes with these code smells and class change-proneness. They show that, in almost all releases of Azureus and Eclipse, classes with code smells are more change-prone than others.

They investigate whether classes with smells are more change-prone than others by testing the null hypothesis:

**H1**: The proportion of classes undergoing at least one change between two releases does not significantly differ between classes with code smells and other classes.

They also found the relation between the number of smells in a class and its change-proneness. They are also interested to evaluate whether classes with a higher number of smells are more change-prone than others by testing the null hypothesis:

**H2**: The number of smells in change-prone classes is not significantly higher than the number of smells in classes that do not change.

They represent the relation between particular kinds of smells and change proneness. Also, they analyze whether particular kinds of smells contribute more than others to changes by testing the null hypothesis:

**H3**: Classes with particular kinds of code smells are not significantly more change-prone than other classes [22].

**Study 23: Zhou et al (2009)** examined the confounding effect of class size on the associations between object-oriented metrics and change-proneness. They used three size metrics, two of which are available during the high-level design phase, to examine the potentially confounding effect of class size on the associations between OO metrics and change-proneness. The OO metrics that are investigated include including 18 cohesion metrics, 20 coupling metrics, and 17 inheritance metrics.

They used two releases Eclipse: Eclipse 2.0 and Eclipse 2.1 which was developed using the Java programming language. Eclipse 2.0 consists of 6,751 Java files, among which 5,686 are class definition files and 1,065 are interface definition files. Eclipse 2.1 consists of 7,909 Java files, among which 6,702 are class definition files and 1,207 are interface definition files. They collected 4,938 data points from Eclipse. Each data point corresponds to one Java class appearing in both Eclipse 2.0 and Eclipse 2.1 and consists of: 1) a set of class size metrics and OO metrics for that class in Eclipse 2.0 and 2) the number of SLOC changes of that class from version 2.0 to version 2.1.
Their results based on Eclipse indicate that: 1) The confounding effect of class size on the associations between OO metrics and change-proneness, in general, exists, regardless of whichever size metric is used; 2) the confounding effect of class size generally leads to an overestimate of the associations between OO metrics and change-proneness; and 3) for many OO metrics, the confounding effect of class size completely accounts for their associations with change-proneness or results in a change of the direction of the associations [23].

**Study 24: Moha et al (2010)** use DECOR (Defecte Detection for Correction) to specify and detect code smells. DECOR is based on a thorough domain analysis of code smells and anti-patterns defined, and provide a domain-specific language to specify code smells and anti-patterns and methods to detect their occurrences automatically. They propose three contributions to the research field related to code and design smells: (1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells; (2) DETEX, a detection technique that instantiates this method; and (3) an empirical validation in terms of precision and recall of DETEX. Using DETEX, they specify four well-known design smells: the anti-patterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and they automatically generate their detection algorithms. They apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open source systems. Code and design smells include low-level or local problems such as code smells which are usually symptoms of more global design smells such as anti-patterns. Code smells are indicators or symptoms of the possible presence of design smells. The number of false positives appears quite high; however, they obtained many false positives because their objective was 100% recall for all systems. Using DETEX and its DSL, the rules can be refined systematically [24].

**Study 25: G. Canfora et al (2010)** use the multivariate time series analysis and forecasting to determine whether a change occurred on a software artefact was consequentially related to changes on other artifacts. Change impact analysis tasks mainly relied on static analysis or dynamic analysis. While these techniques identify actual (direct or indirect) dependencies between artefacts, they have some limitations. They presents an empirical comparison of change couplings identified by means of the two techniques performed on four open source systems namely: Mylyn, FreeBSD (i386), Rhino, and Squid showing that, while association rules provides more precise results. They also propose the use of a hybrid change coupling recommender obtained by combining the two techniques. FreeBSD is a UNIX operating system kernel written in c/c++. They limited
the analysis to the i386 subsystem. Rhino an ECMA/JavaScript interpreter developed for the Mozilla/Firefox browser. Squid is a web caching proxy, written in ANSI c, supporting HTTP, HTTPS, AND FTP. Results of an empirical study performed on four Java and C open source systems show that Granger causality test is able to provide a set of change couplings complementary to association rules, and a hybrid recommender built combining recommendations from association rules and Granger causality is able to achieve a higher recall than the two single techniques [25].

**Study 26: Romano and Pinzger (2011)** used Source Code Metrics to Predict Change-Prone Java Interfaces. They used 10 java open source systems, IUC and C&K metrics and eight eclipses plug in called creole. They used JAVA and C++ language to develop a program to extract the metrics from source code. They takes into account (1) The set of metrics defined by Chidamber and Kemerer (2) a set of metrics to measure the complexity and the usage of interfaces; and (3) two metrics to measure the external cohesion of Java interfaces. The IUC metric shows a stronger correlation with #SCC than the C&K metrics when applied to interfaces. This metric also improves the performance of prediction models to classify Java interfaces into change-prone and not change-prone [26].

**Study 27: Gaikwad (2011)** used class hierarchy method to find change-proness. They used class, subclasses, class hierarchy, inherited classes and used matrix and list method and conclude that classes with internal change are not change prone whereas classes with external change(If the change made in class is affecting to other class then it is external change to that class) are more change prone [27].

**Study 28: Lu et al (2011)** presents a meta-analysis to predict change-proneness in object-oriented metrics. They used 102 java systems, 62 OO metrics including 7 size metrics, 18 cohesion metrics, 20 coupling metrics, and 17 inheritance metrics. They use AUC (the area under a relative operating characteristic, ROC) to evaluate the predictive effectiveness of OO metrics. They concluded that a) Size metrics exhibit moderate or almost moderate ability in discriminating between change-prone and not change-prone classes.

b) Coupling and cohesion metrics generally have a lower predictive ability compared to size metrics.

c) Inheritance metrics have a poor ability to predict change-proneness.
d) Of the investigated metrics, Stmts and SLOC are respectively the best and the second best predictors for change-proneness [28].

Study 29: Mahmoud O.E. et al (2011) presents a suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. Their objective was to derive and validate a set of evolution based metrics as potential indicator of the change prone classes of an OO system when moving from one release to the next. As they evolve over the time from a release to next they become more complex and larger.

They compare the metrics against 1) c&K, which are popular and widely used product metrics measuring different class structural properties and 2) the internal class probability of changes metrics. They derived 16 evolution based metrics using GOM approach (goal-question-metric). They concluded that class is considered to be changed if at least one of its lines of source code was changed or deleted, or at least one new line of code was added to it. The internal class probability of change metric is computed by dividing the number of releases at which a class was changed by the total number of releases at which the class exists. They tested following hypothesis, for each hypothesis, $H_0$ represents the null hypothesis and $H_1$ represents the alternative hypothesis of the null hypothesis.

Hypothesis 1

$H_0$: Evolution-based metrics do not measure different dimensions than do C&K metrics.

$H_1$: Evolution-based metrics measure different dimensions than do C&K metrics.

Hypothesis 2

$H_0$: There is no statistically significant correlation between the evolution-based metrics and change-proneness of classes.

$H_1$: There is statistically significant correlation between the evolution-based metrics and change-proneness of classes.
Hypothesis 3

H₀: There is no difference in accuracy between a prediction model based on the evolution based metrics and a model based on the C&K metrics in identifying change-prone classes.

H₁: There is a difference in accuracy between a prediction model based on the evolution-based metrics and a model based on the C&K metrics in identifying change-prone classes.

Hypothesis 4

H₀: There is no difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the C&K metrics in identifying change-prone classes.

H₁: There is a difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the C&K metrics in identifying change-prone classes.

Hypothesis 5

H₀: There is no difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the evolution-based metrics in identifying change-prone classes.

H₁: There is a difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the evolution-based metrics in identifying change-prone classes.

Hypothesis 6

H₀: There is no difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the internal class probability of changes metric in identifying change-prone classes.
H1: There is a difference in accuracy between a prediction model based on both C&K and evolution-based metrics and a model based only on the internal class probability of changes metric in identifying change-prone classes [29].

Li and Offutt(1996) proposed a set of algorithms that determine what classes are affected by a given change and concluded that it can be used by software testers to find what areas are affected by the changes, so they can test only the affected areas and still feel confident about the quality of the software. Lindvall (1999) found that classes were more change prone. Emam and Melo (1999) found that an export coupling metric had the strongest association with fault proneness. Girba T (2000) that big changes can occur in those classes that are not big in terms of size (i.e. number of methods). Dirk Beyer (2001) found that attribute-based cohesion decreases heavily for classes having at least one superclass and in case of coupling, only the measurement value of one class that has a super-class did not change. Arisholm (2001) that some of the CP measures are statistically significant explanatory variables of change effort, whereas none of the SA measures are. Bieman (2001) found a strong relationship between class size and the number of changes, and resulted that larger classes were changed more frequently. Emden and Moonen (2002) present the java quality assurance by detecting code smells. Bieman (2003) found that class is considered to be change-prone if each of the change-prone measures falls above the threshold. Korus and Tian (2005) found that classes with the highest measurement values were more change-prone classes. Gyimothy et al (2005) compared the capability of sets of Chidamber and Kemerer metrics to predict fault-prone classes within Mozilla, using logistic regression and other machine learning techniques and concluded that CBO is the most discriminating metric. They also found LOC to discriminate fault-prone classes well. Korus and Liu (2007) identified and compared high change modules and modules with the highest measurement values in two large-scale open-source products. They found that the classes with the highest measurement values were more change-prone classes. Sharafat Tahvildari (2008) use probabilistic approach to predict changes in object-oriented software systems and found that large classes were more change prone. Khomh (2009) presents Exploratory Study of the Impact of Code Smells on Software Change-Proneness and found that classes with code smells are not more change prone than other classes. Moha (2010) use DECOR (Defect Detection for Correction) to specify and detect code smells. Romano and Pinzger (2011) used Source Code Metrics to Predict Change-Prone Java Interfaces. Gaikwad (2011) used class hierarchy method to find change-proneness and conclude that classes with internal change are not change prone whereas classes with external change are more change-prone. Hongmin Lu (2011) presents a meta-analysis to predict change-
proneness in object-oriented metrics. They concluded that a) Size metrics exhibit moderate or almost moderate ability in discriminating between change-prone and not change-prone classes. b) Coupling and cohesion metrics generally have a lower predictive ability compared to size metrics. c) Inheritance metrics have a poor ability to predict change-proneness. d) Of the investigated metrics, Stmts and SLOC are respectively the best and the second best predictors for change-proneness.

CHAPTER 3: EXPERIMENTAL WORK

3.1. INTRODUCTION

In this research, in order to analyze the changes in classes from one version to another jfreechart program is used with 21 releases and 168 classes of jfreechart are used. Difference between classes from one version to another version of jfreechart is analyzed using diffchecker online tool. Various metrics are analyzed using Analyst4j to [38].

3.2. Tool Used

In this research analyst4j tool is used. The purpose of analyst4j is to evaluate the use of software metrics in ensuring quality and maintainability in Java systems. It is used to analyze the change proneness of metrics in object oriented systems. Various metrics supported by analyst4j are:

- Weighted Methods Complexity (WMC)
- Response For Class (RFC)
- Lack Of Cohesive Methods (LCOM)
- Coupling Between Objects (CBO)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)

3.3. DIFFCHECKER TOOL:

In order to find the differences online diffchecker.com is used. In this as shown in figure there are two parts of a pages and in each page there is an option of choose file, classes in which the difference is to find are browsed in each pages.
Thus after clicking on find difference, difference between two classes in different releases of program is obtained. The difference between the classes is highlighted; from this differences can be calculated. These differences are as shown in figure:

Fig.3.2: difference in classes in different releases of a program
As there are 3 changes in class range.java, these 3 changes are highlighted. Hence all the changes in classes from one version to another version of a program can be calculated in the same way. The difference between all the 166 classes is as shown in table.

![Table 3.1: changes in classes in different releases of .NET chart.](image)

### 3.4. INTRODUCTION TO VB.NET

VB.net stands for visual basic network embedded technology. It is an Object Oriented computer programming language that can be viewed as an evolution of the classic Visual Basic (VB), which is implemented on the .NET Framework. Microsoft currently supplies two main editions of IDE’s for developing in Visual Basic Microsoft Visual Studio 2010, which is Commercial Software and Visual Basic Express Edition 2010, which is free of charge.

Visual Basic 2010 was released in April 2010, Microsoft released Visual Basic 2010. It is a computer programming system developed and owned by Microsoft. Visual Basic was originally created to make it easier to write programs for the Windows computer operating system. The basis of Visual Basic is an earlier programming language called BASIC that was invented by Dartmouth College professors John Kemeny and Thomas Kurtz. Visual Basic is often referred to using just the initials; VB. Visual Basic is easily the most widely used computer programming system in the history of software [40] [41] [42].
3.4.1. IMPORTANCE OF VB.NET

All of .NET is truly revolutionary and gives programmers a much more capable, efficient and flexible way to write computer software. Visual Basic .NET is a key part of this revolution. Visual Basic was originally created to make it easier to write programs for the Windows computer operating system. It is more than a programming language. Visual Basic was one of the first systems that made it practical to write programs for the Windows operating system. This was possible because VB included software tools to automatically create the detailed programming required by Windows. These software tools not only create Windows programs, they also take full advantage of the graphical way that Windows works by letting programmers "draw" their systems with a mouse on the computer. This is why it's called "Visual" Basic.

Visual Basic also provides unique and complete software architecture. "Architecture" is the way computer programs, such as Windows and VB programs, work together. One of the major reasons why Visual Basic has been so successful is that it includes everything that is necessary to write programs for Windows [39] [40] [41].

3.4.2. FEATURES OF VB.NET

Inheritance: it allows the developer to extend an existing code base to add new functionality.

Type-safe: VB.NET is a type safe language. In this the code can access only memory location that it is allowed to access. Structured exception handling is an integral part of VB.NET.

Method overloading: VB.NET fully supports method overloading. Overloading allow different signatures to be applied to the methods with the same name.

Method overriding: it allows the child objects to override behaviors found in parent.

Parameterized constructors: VB.NET object constructors can have multiple parameters passed in allowing for greater flexibility while creating objects like: default constructor.

Events: VB.NET has a robust architecture in place for handling events. As a result, understanding how events work is crucial to developing successful VB.NET applications. Events are defined by using event keyword. Events are raised using the raised event keyword.

Delegates: it provides a type safe pointer to a function and can be used to hook an event to an event handler.

Attributes: These are descriptive tag that can be added into VB.NET code to provide additional information about what the code does.
**Multithreading**: VB.NET developers can create multithreaded applications. Threading allow different processes to run separately without typing up the user interface or program. Threading functionality is located in the system [39].

### 3.4.3. ADVANTAGES OF VB.NET

- **VB.NET** provides managed code execution that runs under the Common Language Runtime (CLR), resulting in robust, stable and secure applications. All features of the .NET framework are readily available in VB.NET.
- **VB.NET** is totally object oriented. This is a major addition that VB6 and other earlier releases didn't have.
- The .NET framework comes with ADO.NET, which follows the disconnected paradigm, i.e. once the required records are fetched the connection no longer exists. It also retrieves the records that are expected to be accessed in the immediate future. This enhances Scalability of the application to a great extent.
- **VB.NET** uses XML to transfer data between the various layers in the DNA Architecture i.e. data are passed as simple text strings.
- Error handling has changed in VB.NET. A new Try-Catch-Finally block has been introduced to handle errors and exceptions as a unit, allowing appropriate action to be taken at the place the error occurred thus discouraging the use of ON ERROR GOTO statement. This again credits to the maintainability of the code.
- Another great feature added to VB.NET is free threading against the VB single-threaded apartment feature. In many situations developers need spawning of a new thread to run as a background process and increase the usability of the application. VB.NET allows developers to spawn threads wherever they feel like, hence giving freedom and better control on the application.
- Security has become more robust in VB.NET. In addition to the role-based security in VB6, VB.NET comes with a new security model, Code Access security. This security controls on what the code can access. For example you can set the security to a component such that the component cannot access the database. This type of security is important because it allows building components that can be trusted to various degrees.
- The CLR takes care of garbage collection i.e. the CLR releases resources as soon as an object is no more in use. This relieves the developer from thinking of ways to manage memory. CLR does this for them [43].
3.4.4. DISADVANTAGES OF VB.NET

– It is mainly only used for software development.
– It is not a free source.
– Unit tests are not easy to perform.
– View state can affect performance. View state can sometimes have a negative effect on performance especially with the more complex server controls [42].
CHAPTER 4: DIFFERENCE BETWEEN CLASSES

Difference between various classes of jfreechart program is analyzed using online diffchecker tool. In this research 166 classes of jfreechart are used with 21 releases. Difference between the classes from one version to another is calculated in excel sheet i.e. how many number of lines are added and how many number of lines are deleted from one version of a program to another version of a program. Some of the classes are same in all releases but some of the classes are deleted in newer version. With this new classes are also added in other releases. All the newly added and deleted classes are calculated.

4.1. INTRODUCTION TO JFREE CHART

JFreeChart is a free chart library for the Java platform. It runs on the Java 2 Platform (JDK 1.3 or later) and uses the Java 2D API for drawing. JFreeChart is licensed under the terms of the GNU Lesser General Public License (LGPL).

4.1.1. DEPENDENCIES OF JFREECHART

JFreechart has the following dependencies:

JDK 1.3.1 or higher - JFreeChart requires the Java2D and Collections APIs, so it definitely won't work with JDK 1.1 (with a small amount of effort one can probably get it to compile and run with JDK 1.2). If one is using JFreeChart to create applets, this means that one cannot rely on the JVM integrated with Microsoft's Internet Explorer - their users will need to have the Java-2 plugin installed. Most other browsers (e.g. Firefox, Safari and Chrome) offer good support for modern JREs.

JCommon - version 1.0.0 or later. The runtime jar file (version 1.0.17) is included in the JFreeChart distribution. JCommon is licensed under the terms of the GNU Lesser General Public License.

GNU JAXP (JDK 1.3 only) - a free implementation of the standard XML processing APIs for Java. Classes in the org.jfree.data.xml package require this library, or another JAXP v1.1 compliant parser. The gnujaxp.jar file (from the gnujaxp-1.0beta1.zip distribution) is included with JFreeChart. GNU JAXP is licensed under the
terms of the GNU General Public License with an additional library exception. This library is only required by jfreechart if one is using jdk 1.3.

servlet.jar - classes in the org.jfree.chart.servlet package require this file. The JFreeChart distribution includes the servlet.jar file distributed with Tomcat 4.1.31

JUnit - a unit testing framework. JUnit is licensed under the terms of the IBM Common Public License. The JUnit tests included with JFreeChart have been created using JUnit 4.3.1 [48].

4.2. FEATURES OF JFREECHART

JFreeChart is a free 100% Java chart library that makes it easy for developers to display professional quality charts in their applications. JFreeChart's extensive feature set includes:

A consistent and well-documented API, supports a wide range of chart types.

A flexible design that is easy to extend, and targets both server-side and client-side applications. Support for many output types, including Swing components, image files (including PNG and JPEG), and vector graphics file formats (including PDF, EPS and SVG).

JFreeChart is "open source" or, more specifically, free software. It is distributed under the terms of the GNU LESSER GENERAL PUBLIC LICENCE (LGPL), which permits use in proprietary applications.

4.3. LIMITATIONS OF JFREECHART

JFreeChart has some known limitations that will hopefully be addressed in the future.

Some renderers do not respect the series visibility flags yet; - the chart property editors (accessible by right-clicking on the chart panel) are horribly out of date and probably shouldn't be used. Item labels (if displayed) are not taken into account for the automatically calculated axis range. As a workaround, one can increase the axis margins.

Tick labels on a DateAxis that uses a SegmentedTimeline can be problematic.
There is no support for writing charts to JPEG format on JDK 1.3. JPEG is not a good format for charts in any case, and it is usually better to use PNG format.

Copying charts to the clipboard (supported by the ChartPanel class) will not work if one is running on a very old version of the Java runtime version 1.3.1 [48].

4.4. CATEGORIZATION OF RELEASES

These releases are categorized into three phases as:

Production state

Stable state

Development state

The various releases of jfree chart falls into bug fix, major changes, minor bug fix, maintenance and stable categories. Table representation of these releases is as:

<table>
<thead>
<tr>
<th>Releases</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.1</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.2</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.3</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.5</td>
<td>Major changes have been made in this release</td>
</tr>
<tr>
<td>0.9.6</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.7</td>
<td>fix minor bug</td>
</tr>
<tr>
<td>0.9.8</td>
<td>fix minor bug</td>
</tr>
<tr>
<td>Version</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>0.9.9</td>
<td>Major changes have been made in this release</td>
</tr>
<tr>
<td>0.9.10</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.11</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.12</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.13</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.14</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.15</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.16</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.17</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.18</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.19</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.20</td>
<td>Bug fix</td>
</tr>
<tr>
<td>0.9.21</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.0</td>
<td>first stable release</td>
</tr>
<tr>
<td>1.0.1</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.2</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.3</td>
<td>Bug fix</td>
</tr>
<tr>
<td>Version</td>
<td>Type</td>
</tr>
<tr>
<td>---------</td>
<td>----------------</td>
</tr>
<tr>
<td>1.0.4</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.5</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.6</td>
<td>Promotional</td>
</tr>
<tr>
<td>1.0.7</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.8</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.9</td>
<td>Bug fix</td>
</tr>
<tr>
<td>1.0.10</td>
<td>Maintenance release</td>
</tr>
</tbody>
</table>

Table 4.1: types of releases in jfree chart.

4.5. WORKING OF A SOFTWARE

All the classes of different releases are stored in an excel file and then this excel file is given as an input to the software to produce the results. Thus after clicking on show file excel file is browsed.
Fig. 4.1: Result of software.

In this page after clicking on browse we can browse the file in which we want to find the difference. We have browsed the table in which we calculated the number of changes, i.e. number of times one class is changed. After browsing the file the results are displayed as shown:
4.6. CO-RELATION BETWEEN CLASSES

The next step is to find the correlation between classes. For this the average rate of changed, unchanged, newly added and removed classes is calculated.

For this total number of classes are calculated. Rate of changed classes is calculated.

Average Rate of changed classes = changed classes/number of classes.

Similarly rate of unchanged classes is calculated.

Average Rate of unchanged classes = unchanged classes/number of classes.

Average Rate of removed are calculated by the formula:

Average Rate of removed classes = removed classes/number of classes.
And thus newly added classes are calculated by the same formula.

Average Rate of newly added classes = newly added classes/number of classes.

The result of these classes is as:

<table>
<thead>
<tr>
<th></th>
<th>versions</th>
<th>changed classes</th>
<th>unchanged classes</th>
<th>newly added</th>
<th>removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.01</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2.02</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3.03</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4.04</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5.05</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6.06</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7.07</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8.08</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9.09</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10.10</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>11.11</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12.12</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>13.13</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>14.14</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>15.15</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>16.16</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>17.17</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>18.18</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>19.19</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20.20</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>21.21</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>22.22</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>23.23</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>24.24</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>25.25</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>26.26</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>27.27</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>28.28</td>
<td>0.13</td>
<td>0.01</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: Average of changed, unchanged, newly added and removed classes

The rate of newly added classes is shown with the help of graph:

Graph 1: Average of newly added classes
Rate of total number of removed classes is shown with the help of graph

Graph 2: Average of newly added classes

The correlation between changed, unchanged, removed and newly added classes is represented with the help of graph as:

Graph 3: co-relation between changed, newly added, unchanged and removed classes.
After this the gap between the changes in calculated. Gap means after how many versions one class is changed.

This average gap is calculated by

Average gap = sum of gaps/total number of gaps.

The result of these gaps is stored in excel sheet as shown in figure:

![Excel Sheet](image)

Table 4.5: Average gap in changes across releases.

In this table the gap between the different classes according to different releases is calculated. For example in first row the class is changed after one release and that gap is represented by 1. Then it is changed in the next release and the gap is represented by 0. If it is changed after 3 releases the gap will be represented by 3 and so on.

Next we analyze the change in classes with respect to the type of release, i.e. which class is more changed whether stable release is more change prone, maintenance release or bug fix release is more prone to change. We have shown the differences in the form of patches.
The changes between these classes are as shown in figure 4.5.

Fig: Types of changes with releases

From this we analyze that the classes in bug fix releases are more prone to change as compare to the other classes. Classes in stable release are more change-prone as compare to maintenance release.
CHAPTER 5

RESULTS

In this research we found the change proneness in classes and metrics of different object oriented systems. In classes we use 165 classes of 36 releases of jfree chart program. We used total of near about 4000 classes in all the 36 release. We calculate the difference between classes in difference between classes using online diffchecker tool. After calculating the differences, we calculated the changed classes, unchanged classes, newly added and removed classes in all the 35 releases. We also calculated the gap across the classes in different releases.

In metrics we used six class level metrics that includes size, coupling, cohesion and size metrics. We calculate those metrics using Analst4j tool. After calculating these metrics we found the correlation between changed classes, and class level metrics.

**Case 1**: In this research we find the correlation between the class level metrics and classes in different releases of j-free chart program. We find the correlation of WMC, RFC, NOC, LCOM, CBO, DIT with changed classes. The result is shown in figure:

![Image](image_url)

The above result shows the values of various class level metrics. The value of WMC is -0.0257, RFC is -0.0444, LCOM is 0.07109, CBO is -0.02732, NOC is 0.00414 and DIT is 0.084388. From these values we can analyze that the coupling metrics are more prone to changes as compare to other metrics.

**Case 2**: we find the differences in classes across the different releases of a program. The differences are as shown in figure:
From this we analyze that the classes in bug fix release are more prone to change as compare to stable release and maintenance release.
CHAPTER 6

CONCLUSION AND FUTURE WORK

The goal of thesis was to find the change-proneness in class level metrics and different classes in object oriented systems. We used 31 releases of jfree chart program and 168 classes in each release of jfree chart program. We used near about 4000 classes in all the releases. After finding the results we conclude that classes with bug fix release are more change prone classes and in case of metrics the coupling metrics are more prone to change as compare to other metrics.

In future, we will replicate this study using more OO metrics on more systems. In particular, we will validate our findings in other OO programming languages such as C++, VB. we will replicate this study using more OO metrics on more systems. Consequently, more general conclusions would be obtained. As the ability of OO metrics to predict change-proneness may depend on the programming language used. Consequently, more general conclusions would be obtained.
References


citeseer.ist.psu.edu/vanemden02java.html


http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1199188&tag=1


ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1245308


www.cs.colostate.edu/~rta/publications/DPCmetrics03.pdf


http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1498769

flosshub.org/system/files/Gymothy.pdf


http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4145022
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5328703


http://www.dsba.unisannio.it/Members/ceccarelli/bibliography...

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6080797

http://www.springerlink.com/content/714655556807596/

onlinelibrary.wiley.com/doi/10.1002/smr.1549/pdf


http://139.141.170.133/drjehad/A%20design%20based%20cohesion.pdf

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4021375

Web Links
[38] http://sourceforge.net/projects/jfreechart/?source=directory
[39] http://www.slideshare.net/ddutch/vbnet02
[42] http://www.homeandlearn.co.uk/net/vbnet.html

Books
## ANNEXURE

<table>
<thead>
<tr>
<th>Releases</th>
<th>classes</th>
<th>WMC</th>
<th>RFC</th>
<th>LCOM</th>
<th>CBO</th>
<th>NOC</th>
<th>DIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0</td>
<td>AbstractCategoryItemRenderer.java</td>
<td>9</td>
<td>20</td>
<td>0.88</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>AbstractTitle.java</td>
<td>23</td>
<td>35</td>
<td>0.99</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>AbstractXYItemRenderer.java</td>
<td>8</td>
<td>11</td>
<td>0.76</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>AreaCategoryItemRenderer.java</td>
<td>2</td>
<td>32</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>AreaXYItemRenderer.java</td>
<td>10</td>
<td>58</td>
<td>0.91</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Axis.java</td>
<td>36</td>
<td>67</td>
<td>0.95</td>
<td>13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>AxisConstants.java</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>AxisNotCompatibleException.java</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BarRenderer.java</td>
<td>6</td>
<td>20</td>
<td>1.03</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CandlestickRenderer.java</td>
<td>12</td>
<td>67</td>
<td>0.7</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CategoryAxis.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CategoryItemRenderer.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CategoryPlot.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CategoryPlotConstants.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartFactory.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartFrame.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartMouseEvent.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartMouseListener.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartPanel.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartPanelConstants.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartRenderingInfo.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ChartUtilities.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CombinedXYPlot.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CrosshairInfo.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DateAxis.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DateTitle.java</td>
<td>5</td>
<td>11</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>DateTickUnit.java</td>
<td>2</td>
<td>3</td>
<td>1.7</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>DefaultShapeFactory.java</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DateUnit.java</td>
<td>4</td>
<td>8</td>
<td>0.33</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HighLow.java</td>
<td>14</td>
<td>16</td>
<td>0.9</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HighLowRenderer.java</td>
<td>3</td>
<td>33</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalAxis.java</td>
<td>2.5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalBarRenderer.java</td>
<td>7</td>
<td>41</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalBarRenderer3D.java</td>
<td>9</td>
<td>96</td>
<td>0.75</td>
<td>14</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>HorizontalCategoryAxis.java</td>
<td>14</td>
<td>132</td>
<td>0.73</td>
<td>25</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalCategoryPlot.java</td>
<td>15</td>
<td>92</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalDateAxis.java</td>
<td>19</td>
<td>139</td>
<td>0.61</td>
<td>26</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalIntervalBarRenderer.java</td>
<td>7</td>
<td>40</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalLogarithmicAxis.java</td>
<td>14</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HorizontalNumberAxis.java</td>
<td>17</td>
<td>145</td>
<td>0.56</td>
<td>24</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>HorizontalNumberAxis3D.java</td>
<td>10</td>
<td>75</td>
<td>0.85</td>
<td>18</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Class</td>
<td>Count</td>
<td>Size</td>
<td>Complexity</td>
<td>Measures</td>
<td>Loc</td>
<td>Methods</td>
<td>Classes</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------</td>
<td>------</td>
<td>------------</td>
<td>----------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>HorizontalSymbolicAxis.java</td>
<td>15</td>
<td>55</td>
<td>0.88</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HorizontalValuePlot.java</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ImageTitle.java</td>
<td>10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>IntervalMarker.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JFreeChart.java</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>JFreeChartConstants.java</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JFreeChartInfo.java</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JThermometer.java</td>
<td>19</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Legend.java</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LegendItem.java</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LegendItemCollection.java</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LegendItemLayout.java</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LegendTitle.java</td>
<td>4</td>
<td>5</td>
<td>1.2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LineAndShapeRenderer.java</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Marker.java</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MeterLegend.java</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>MeterPlot.java</td>
<td>35</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NumberAxis.java</td>
<td>21</td>
<td>35</td>
<td>0.96</td>
<td>11</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NumberTickUnit.java</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>OverlaidVerticalCategoryPlot.java</td>
<td>11</td>
<td>59</td>
<td>0.6</td>
<td>14</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>OverlaidXYPlot.java</td>
<td>10</td>
<td>48</td>
<td>0.61</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PeriodMarkerPlot.java</td>
<td>15</td>
<td>69</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Plot.java</td>
<td>46</td>
<td>83</td>
<td>0.96</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PiePlot.java</td>
<td>33</td>
<td>146</td>
<td>0.95</td>
<td>26</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PlotException.java</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PlotNotCompatibleException.java</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SeriesShapeFactory.java</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ShapeFactory.java</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SignalRenderer.java</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Spacer.java</td>
<td>7</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>StackedHorizontalBarRenderer.java</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>StackedVerticalBarRenderer.java</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>StandardLegend.java</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>StandardLegendItemLayout.java</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>StandardXYItemRenderer.java</td>
<td>12</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TextTitle.java</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ThermometerPlot.java</td>
<td>46</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Tick.java</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>TickUnit.java</td>
<td>5</td>
<td>7.5</td>
<td>0.57</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ValueAxis.java</td>
<td>41</td>
<td>51</td>
<td>0.98</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>VerticalAxis.java</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>VerticalBarRenderer.java</td>
<td>7</td>
<td>41</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VerticalCategoryAxis.java</td>
<td>10</td>
<td>72</td>
<td>0.78</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VerticalCategoryPlot.java</td>
<td>17</td>
<td>100</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VerticalIntervalBarRenderer.java</td>
<td>8</td>
<td>71</td>
<td>0.93</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VerticalLogarithmicAxis.java</td>
<td>11</td>
<td>62</td>
<td>0.85</td>
<td>15</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Class</td>
<td>Start</td>
<td>End</td>
<td>Mean</td>
<td>Min</td>
<td>Max</td>
<td>Frequency</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>VerticalNumberAxis.java</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>VerticalSymbolicAxis.java</td>
<td>5</td>
<td>55</td>
<td>0.88</td>
<td>21</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>VerticalValuePlot.java</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>VerticalXYBarRenderer.java</td>
<td>6</td>
<td>34</td>
<td>0.7</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>WindAxis.java</td>
<td>14</td>
<td>14</td>
<td>0.82</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>WindItemRenderer.java</td>
<td>2</td>
<td>51</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>XYItemRenderer.java</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>XYPlot.java</td>
<td>40</td>
<td>187</td>
<td>0.94</td>
<td>23</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>XYStepRenderer.java</td>
<td>2</td>
<td>34</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>dataLinearPlotFitAlgorithm.java</td>
<td>3</td>
<td>17</td>
<td>0.75</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MovingAveragePlotFitAlgorithm.java</td>
<td>3</td>
<td>11</td>
<td>0.39</td>
<td>3.5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PlotFit.java</td>
<td>5</td>
<td>17</td>
<td>0.25</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PlotFitAlgorithm.java</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Classes used in different releases.
Chapter 1: INTRODUCTION TO OBJECT ORIENTED PARADIGMS  01

CHAPTER 2: EXISTING STUDIES  27

CHAPTER 3: EXPERIMENTAL WORK  47

CHAPTER 4: DIFFERNECE BETWEEN CLASSES  53

Chapter 5: RESULTS  64

Chapter 6: CONCLUSION AND FUTURE WORK  66

References  67
Buy your books fast and straightforward online - at one of world’s fastest growing online book stores! Environmentally sound due to Print-on-Demand technologies.

Buy your books online at
www.morebooks.shop

Kaufen Sie Ihre Bücher schnell und unkompliziert online – auf einer der am schnellsten wachsenden Buchhandelsplattformen weltweit! Dank Print-On-Demand umwelt- und ressourcenschonend produziert.

Bücher schneller online kaufen
www.morebooks.shop