5 Mining Data from Software Repositories

One of the problems faced by the software engineering community is scarcity of data for conducting empirical studies. However, the software repositories can be mined to collect and gather the data that can be used for providing empirical results by validating various techniques or methods. The empirical evidence gathered through analyzing the data collected from the software repositories is considered to be the most important support for software engineering community these days. These evidences can allow software researchers to establish well-formed and generalized theories. The data obtained from software repositories can be used to answer a number of questions. Is design A better than design B? Is process/method A better than process/method B? What is the probability of occurrence of a defect or change in a module? Is the effort estimation process accurate? What is the time taken to correct a bug? Is testing technique A better than testing technique B? Hence, the field of extracting data from software repositories is gaining importance in organizations across the globe and has a central and essential role in aiding and improving the software engineering research and development practice.

As already mentioned in Chapter 1 and 4 the data can either be collected from proprietary software, open source software (OSS), or university software. However, obtaining data from proprietary software is extremely difficult as the companies are not usually willing to share the source code and information related to the evolution of the software. Another source for collecting empirical data is academic software developed by universities. However, collecting data from software developed by student programmers is not recommended, as the accuracy and applicability of this data cannot be determined. In addition, the university software is developed by inexperienced, small number of programmers and thus does not have applicability in the real-life scenarios.

The rise in the popularity of the use of OSS has made vast amount of data available for use in empirical research in the area of software engineering. The information from open source repositories can be easily extracted in a well-structured manner. Hence, now researchers have access to vast repositories containing large-sized software maintained over a period of time.

In this chapter, the basic techniques and procedures for extracting data from software repositories is provided. A detailed discussion on how change logs and bug reports are organized and structured is presented. An overview of existing software engineering repositories is also given. In this chapter, we present defect collection and reporting system that can be used for collecting changes and defects from maintenance phase.

5.1 Configuration Management Systems

Configuration management systems are central to almost all software projects developed by the organizations. The aim of a configuration management system is to control and manage changes that occur in all the artifacts produced during the software development life cycle. The artifacts (also known as deliverables) produced during the software development life cycle include software requirement specification, software design document, source code listings, user manuals, and so on (Bersoff et al. 1980; Babich 1986).

A configuration management system also controls any changes incurred in these artifacts. Typically, configuration management consists of three activities: configuration identification, configuration control, and configuration accounting (IEEE/ANSI Std. 1042–1987, IEEE 1987).

5.1.1 Configuration Identification

Each and every software project artifact produced during the software development life cycle is uniquely named. The following terminologies are related to configuration identification:

- Release: The first issue of a software artifact is called a release. This usually provides most of the functionalities of a product, but may contain a large number of bugs and thus is prone to issue fixing and enhancements.
- Versions: Significant changes incurred in the software project's artifacts are called versions. Each version tends to enhance the functionalities of a product, or fix some critical bugs reported in the previous version. New functionalities may or may not be added.
- Editions: Minor changes or revisions incurred in the software artifacts are termed as editions. As opposed to a version, an edition may not introduce significant enhancements or fix some critical issues reported in the previous version. Rather, small fixes and patches are introduced.

5.1.2 Configuration Control

Configuration control is a critical process of versioning or configuration management activities. This process incorporates the approval, control, and implementation of changes to the software project artifact(s), or to the software project itself. Its primary purpose is to ensure that each and every change incurred to any software artifact is carried out with the knowledge and approval of the software project management team. A typical change request procedure is presented in Figure 5.1.

Figure 5.2 presents the general format of a change request form. The request consists of some important fields such as severity (impact of failure on software operation) and priority (speed with which the defect must be addressed).

The change control board (CCB) is responsible for the approval and tracking of changes. The CCB carefully and closely reviews each and every change before approval. After the changes are successfully implemented and documented, they must be notified so that they are tracked and recorded in the software repository hosted at version control systems (VCS). Sometimes, it is also known as the software library, archive, or repository, wherein the entire official artifacts (documents and source code) are maintained during the software development life cycle.

The changes are notified through a software change notice. The general format of a change notice is presented in Figure 5.3.

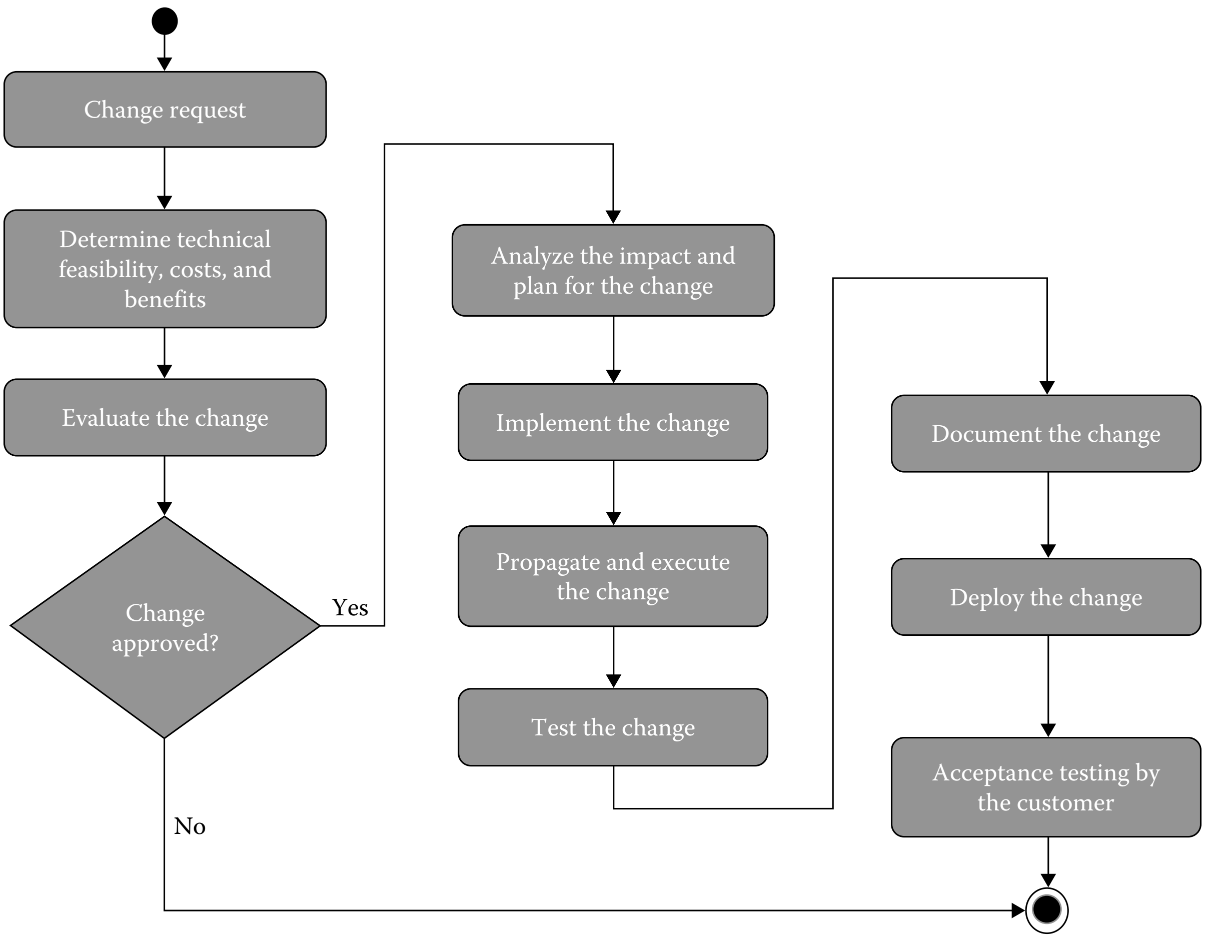
Change request
Determine technical feasibility, costs, and benefits
Evaluate the change
Change approved?
Yes
No
Analyze the impact and plan for the change
Implement the change
Propagate and execute the change
Test the change
Document the change
Deploy the change
Acceptance testing by the customer

FIGURE 5.1
Change cycle.

Change Request Form				
Change Request ID				
Type of Change Request	☐ Enhancement	☐ Defect Fixing	☐ Other (Specify)	
Project				
Requested By	*Project team member name*			
Brief Description of the Change Request	*Description of the change being requested*			
Date Submitted				
Date Required				
Priority	☐ Low	☐ Medium	☐ High	☐ Mandatory
Severity	☐ Trivial	☐ Moderate	☐ Serious	☐ Critical
Reason for Change	*Description of why the change is being requested*			
Estimated Cost of Change	*Estimates for the cost of incurring the change*			
Other Artifacts Impacted	*List other artifacts affected by this change*			
Signature				

FIGURE 5.2
Change request form.

Change Notice Form				
Change Request ID				
Type of Change Request	☐ Enhancement	☐ Defect Fixing	☐ Other (Specify)	
Project				
Module in which change is made				
Change Implemented by	*Project team member name*			
Date and time of change implementation				
Change Approved By	*CCB member who approved the change*			
Brief Description of the Change Request	*Description of the change incurred*			
Decision	☐ Approved	☐ Approved with Conditions	☐ Rejected	☐ Other
Decision Date				
Conditions	*Conditions imposed by the CCB*			
Approval Signature				

FIGURE 5.3
Software change notice.

5.1.3 Configuration Accounting

Configuration accounting is the process that is responsible for keeping track of each and every activity, including changes, and any action that affects the configuration of a software product artifact, or the software product itself. Generally, the entire data corresponding to each and every change is maintained in the VCS. Configuration accounting also incorporates recording and reporting of all the information required for versioning or configuration management of a software project. This information includes the status of software artifacts under versioning control, metadata, and other related information for the proposed changes, and the implementation status of the changes that were approved in the configuration control process.

A typical configuration status report includes

- A list of software artifacts under versioning. These comprise a baseline.
- Version-wise date as to when the baseline of a version was established.
- Specifications that describe each artifact under versioning.
- History of changes incurred in the baseline.
- Open change requests for a given artifact.
- Deficiencies discovered by reviews and audits.
- The status of approved changes.

In the next section, we present the importance of mining information from software repositories, that is, information gathered from historical data such as defect and change logs.

5.2 Importance of Mining Software Repositories

Software repositories usually provide a vast array of varied and valuable information regarding software projects. By applying the information mined from these repositories, software engineering researchers and practitioners do not need to depend primarily on their intuition and experience, but more on field and historical data (Kagdi et al. 2007).

However, past experiences, dominant methodologies, and patterns still remain the driving force for significant decision-making processes in software organizations (Hassan 2008). For instance, software engineering practitioners mostly rely on their experience and gut feeling while making essential decisions. Even the managers tend to allocate their organization's development and testing resources on the grounds of their experience in previous software projects, and their intuition regarding the complexity and criticality of the new project when compared with the previous projects. Developers generally employ their experience while adding new features or issue fixing. Testers tend to prioritize the testing of modules, classes, and other artifacts that are discovered to be error prone based on historical data and bug reports.

A major reason behind the ignorance of how valuable is the information provided in software engineering repositories, is perhaps the lack of effective mining techniques that can extract the right kind of information from these repositories in the right form.

Recognizing the need for effective mining techniques, the mining software repositories (MSR) field has been developed by software engineering practitioners. The MSR field analyzes and cross-links the rich and valuable data stored in the software repositories to discover interesting and applicable information about various software systems as well as projects. However, software repositories are generally employed in practice as mere record-keeping data stores and are rarely used to facilitate decision-making processes (Hassan 2008). Therefore, MSR researchers also aim at carrying out a significant transformation of these repositories from static record-keeping repositories into active ones for guiding the decision-making process of modern software projects.

Figure 5.4 depicts that after mining the relevant information from software repositories, data mining techniques can be applied and useful results can be obtained, analyzed, and interpreted. These results will guide the practitioners in decision making. Hence, mining data from software repositories will exhibit the following potential benefits:

- Enhance maintenance of the software system
- Empirical validation of techniques and methods
- Supporting software reuse
- Proper allocation of testing and maintenance resources

5.3 Common Types of Software Repositories

This section describes different types of software repositories and various artifacts provided by them that may be used to extract useful information (Hassan 2008). Figure 5.5 presents the different types of software repositories commonly employed.

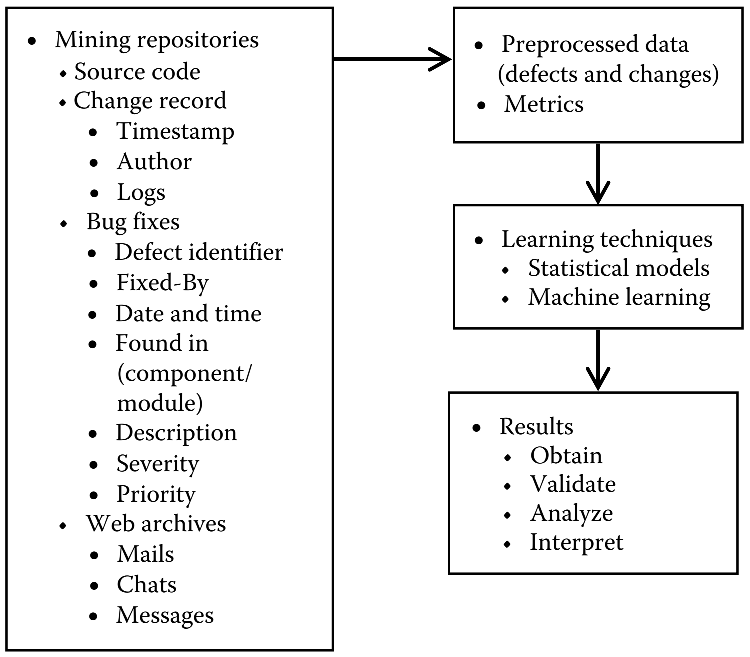

FIGURE 5.4
Data analysis procedure after mining software repositories.

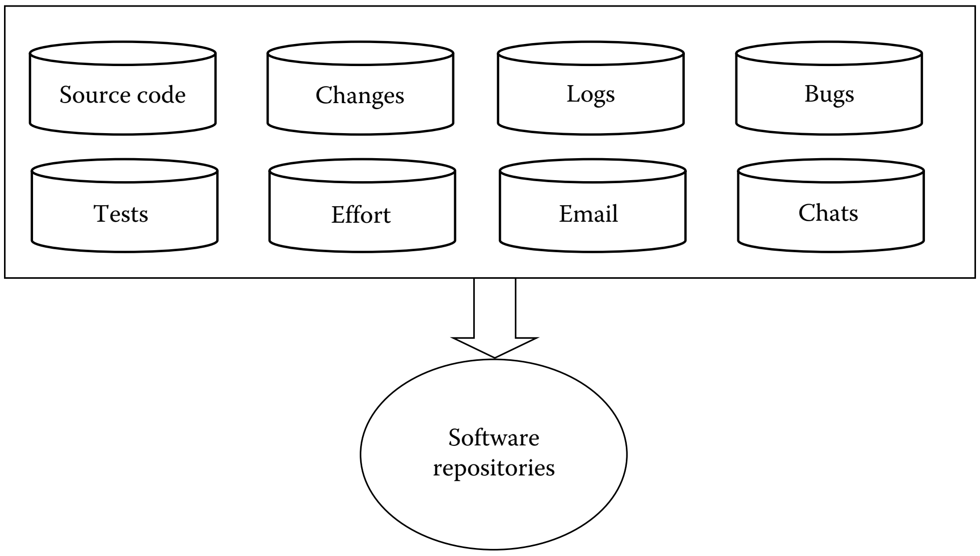

FIGURE 5.5
Commonly used software repositories.

5.3.1 Historical Repositories

Historical repositories record varied information regarding the evolution and progress of a software project. They also capture significant historical dependencies prevalent between various artifacts of a project, such as functions (in the source code), documentation files, or configuration files (Gall et al. 1998). Developers can possibly employ the information extracted from these historical repositories for various purposes. A major area of application is propagating the changes to related artifacts, instead of analyzing only static and/or dynamic code dependencies that may not be able to capture significant dependencies.

For example, consider an application that consists of a module (say module 1) that takes in some input and writes it to a data store, and another module (module 2) that reads the data from that data store. If there is a modification in the source code of the module that saves data to the data store, we may be required to perform changes to module 2 that retrieves data from that data store, although there are no traditional dependencies (such as control flow dependency) between the two modules. Such dependencies can be determined if and only if we analyze the historical data available for the software project. For this example, data extracted from historical repositories will reveal that the two modules, for saving the data to the data store and reading the data from that data store, are co-changing, that is, a change in module 1 has resulted in a change in module 2.

Historical repositories include source control repositories, bug repositories, and archived communications.

- Source control repositories

  Source control repositories record and maintain the development trail of a project. They track each and every change incurred in any of the artifacts of a software system, such as the source code, documentation manuals, and so on. Additionally, they also maintain the metadata regarding each change, for instance, the developer or project member who carried out the change, the timestamp when the change was performed, and a short description of the change. These are the most readily available repositories, and also the most employed in software projects (Ambriola et al. 1990). Git, CVS, subversion (SVN), Perforce, and ClearCase are some of the popular source control repositories that are used in practice. Source control repositories, also known as VCS, are discussed in detail later in Section 5.5.

- Bug repositories

  These repositories track and maintain the resolution history of defect/bug reports, which provide valuable information regarding the bugs that were reported by the users of a large software project, as well as the developers of that project. Bugzilla and Jira are the commonly used bug repositories.

- Archived communications

  Discussions regarding the various aspects of a software project during its life cycle, such as mailing lists, emails, instant messages, and internet relay chats (IRCs) are recorded in the archived communications.

5.3.2 Run-Time Repositories or Deployment Logs

Run-time repositories, also known as deployment logs, record information regarding the execution of a single deployment, or different deployments of a software system. For example, run-time repositories may record the error messages reported by a software application at varied deployment sites. Deployment logs are now being made available at a rapidly increasing rate, owing to their use for remote defect fixing and issue resolution and because of some legal acts. For example, the Sarbanes-Oxley Act of 2002 states that it is mandatory to log the execution of every commercial, financial, and telecommunication application in these repositories.

Run-time repositories can possibly be employed to determine the execution anomalies by discovering dominant execution or usage patterns across various deployments, and recording the deviations observed from such patterns.

5.3.3 Source Code Repositories

Source code repositories maintain the source code for a large number of OSS projects. Sourceforge.net and Google code are among the most commonly employed code repositories, and host the source code for a large number of OSS systems, such as Android OS, Apache Foundation Projects, and many more. Source code is arguably one of the most important artifacts of any software project, and its application is discussed in detail later in Section 5.8.

5.4 Understanding Systems

Understanding large software systems still remains a challenging process for most of the software organizations. This is probably because of various reasons. Most importantly, documentation manuals and files pertaining to large systems rarely exist and even if such data exists, they are often not updated. In addition, system experts are usually too preoccupied to guide novice developers, or may no longer be a part of the organization (Hassan 2008). Evaluating the system characteristics and tracing its evolution history thus have become important techniques to gain an understanding about the system.

5.4.1 System Characteristics

A software system may be analyzed by the following general characteristics, which may prove helpful in decision-making process on whether data should be collected from a software system and used in research-centric applications or not.

1. Programming language(s): The computer language(s) in which a software system has been written and developed. Java remains the most popular programming language for many OSS systems, such as Apache projects, Android OS, and many more. C, C++, Perl, and Python are also other popular programming languages.
2. Number of source files: This attribute gives the total number of source code files contained in a software system. In some cases, this measure may be used to depict the complexity of a software system. A system with greater number of source files tends to be more complex than those with lesser number of source files.
3. Number of lines of code (LOC): It is an important size metric of any software system that indicates the total number of LOC of the system. Many software systems are classified on the basis of their LOC as small-, medium-, and large scale systems. This attribute also gives an indication of the complexity of a software system. Generally, systems with larger size, that is, LOC, tend to be more complex than those with smaller size.
4. Platform: This attribute indicates the hardware and software environment (predominantly software environment) that is required for a particular software system to function. For example, some software systems are meant to work only on Windows OS.
5. Company: This attribute provides information about the organization that has developed, or contributed to the development of a software system.
6. Versions and editions: A software system is typically released in versions, with each version being rolled out to incorporate some significant changes in the

previous version of that software system. Even for a given version, several editions may be released to incorporate some minor changes in the software system.

7. Application/domain: A software system usually serves a fundamental purpose or application, along with some optional or secondary features. Open source systems typically belong to one of these domains: graphics/media/3D, IDE, SDK, database, diagram/visualization, games, middleware, parsers/generators, programming language, testing, and general purpose tools that combine multiple such domains.

5.4.2 System Evolution

Software evolution primarily aims to incorporate and revalidate the probable significant modifications or changes to a software system without being able to predict in advance how the customer or user requirements will eventually evolve (Gall et al. 1997). The existing, large software system can never be entirely complete and hence continuously evolves. As the software system continues to evolve, its complexity will tend to increase until and unless we turn up with a better solution to solve or mitigate these issues.

Software system evolution also aims to ensure the reliability and flexibility of the system. However, to adapt to the ever-changing real-world environment, a system should evolve once in every few months. Faster evolution is achievable and necessary too, owing to the rapidly increasing resources over the Internet, which makes it easier for the users to extract useful information.

The concept of software evolution has led to the phenomenon of OSS development. Any user can easily obtain the project artifacts and modify them according to his/her requirements. The most significant advantage of this open source movement is that it promotes the evolution of new ideas and methodologies that aim to improve the overall software process and product life cycle. This is the basic principal and agenda of software engineering. However, a negative impact is that it is difficult to keep a close and continuous check on the development and modification of a software project, if it has been published as open source (Livshits and Zimmermann et al. 2005).

It can be stated that the software development is an ongoing process, and it is truly a never-ending cycle. After going through various methodologies and enhancements, evolutionary metrics were consequently proposed in the literature to cater to the matter of efficiency and effectiveness of the programs. A software system may be analyzed by various evolutionary and change metrics (suggested by Moser et al. 2008), which may prove helpful in understanding the evolution and release history of a software system (Moser et al. 2008). The details of the evolution metrics are given in Chapter 3.

5.5 Version Control Systems

VCS, also known as source control systems or simply versioning systems, are systems that track and record changes incurred to a single artifact or a set of artifacts of a software system.

5.5.1 Introduction

In this section we provide classification of VCS. Each and every change, no matter how big or small, is recorded over time so that we may recall specific revisions or versions of the system artifacts later.

The following general terms are associated with a VCS (Ball et al. 1997):

- Revision numbers: VCS typically tend to distinguish between different version numbers of the software artifacts. These version numbers are usually called revision numbers and indicate various versions of an artifact.
- Release numbers: With respect to software products, revision numbers are termed as release numbers and these indicate different releases of the software product.
- Baseline or trunk: A baseline is the approved version or revision of a software artifact from which changes can be made subsequently. It is also called trunk or master.
- Tag: Whenever a new version of a software product is released, a symbolic name, called the tag, is assigned to the revision numbers of current software artifacts. The tag indicates the release number. In the header section of every tagged artifact, the relation tag (symbolic name)—revision number is stored.
- Branch: They are very common in a VCS and a single branch indicates a self-maintained line of development. In other words, a developer may create a copy of some project artifacts for his own use, and give an appropriate identification to the new line of development. This new line of development created from the originally stored software artifacts is referred to as a branch. Hence, multiple copies of a file may be created independent of each other. Each branch is characterized by its branch number or identification.
- Head: It (sometimes also called "tip") refers to the commit that has been made most recently, either to a branch or to the trunk. The trunk and every branch have their individual heads. Head is also sometimes referred to the trunk.

Figure 5.6 depicts the branches that come out of a baseline or trunk.

The major functionalities provided by a VCS include the following:

- Revert project artifacts back to a previously recorded and maintained state
- Revert the entire software project back to a previously recorded state
- Review any change made over time to any of the project artifacts
- Retrieve metadata about any change, such as the developer or project member who last modified any artifact that might be causing a problem, and more

Employing a VCS also means that if we accidentally modify, damage, or even lose some project artifacts, we can generally recover them easily by simply cloning or downloading those artifacts from the VCS. Generally, this can be achieved with insignificant costs and overheads.

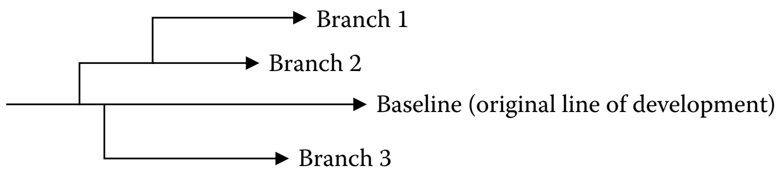

FIGURE 5.6
Trunk and branches.

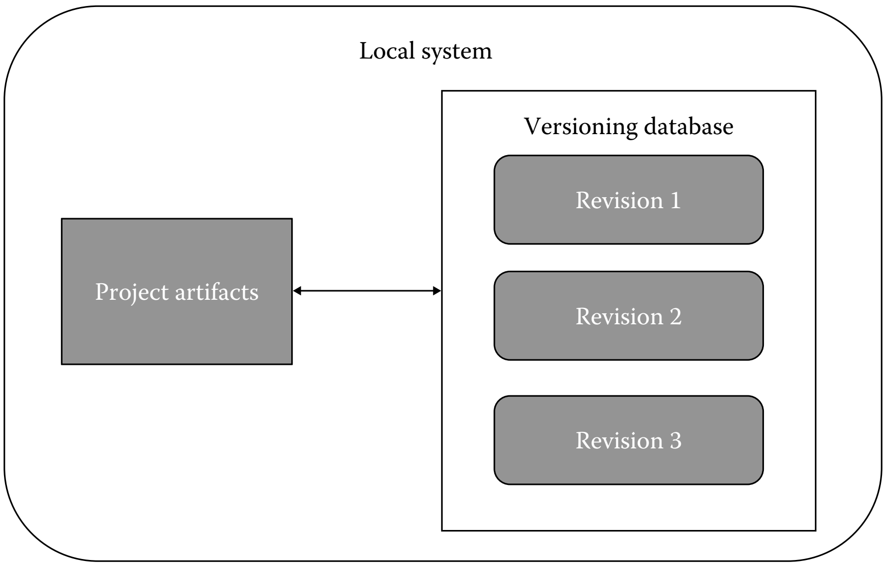

FIGURE 5.7
Local version control.

5.5.2 Classification of VCS

VCS may be categorized as follows (http://git-scm.org).

5.5.2.1 Local VCS

Local VCS employ a simple database that records and maintains all the changes to artifacts of the software project under revision control. Figure 5.7 presents the concept of a local VCS.

A system named revision control system (RCS) was a very popular local versioning system, which is still being used by many organizations as well as the end users. This tool operates by simply recording the patch sets (i.e., the differences between two artifacts) while moving from one revision to the other in a specific format on the user's system. It can then easily recreate the image of a project artifact at any point of time by summing up all the maintained patches.

However, the user cannot collaborate with other users on other systems, as the database is local and not maintained centrally. Each user has his/her own copy of the different revisions of project artifacts, and thus there are consistency and data sharing problems. Moreover, if one user loses the versioning data, recovering it is impossible until and unless a backup is maintained from time to time.

5.5.2.2 Centralized VCS

Owing to the drawbacks of local versioning systems, centralized VCS (CVCS) were developed. The main aim of CVCS is to allow the user to easily collaborate with different users on other systems. These systems, such as CVS, Perforce, and SVN, employ a single centralized server that records and maintains all the versioned artifacts of a software project under revision control, and there are a number of clients or users that check out (obtain) the project artifacts from that central server. For several years, this has been the standard methodology followed in various organizations for version control.

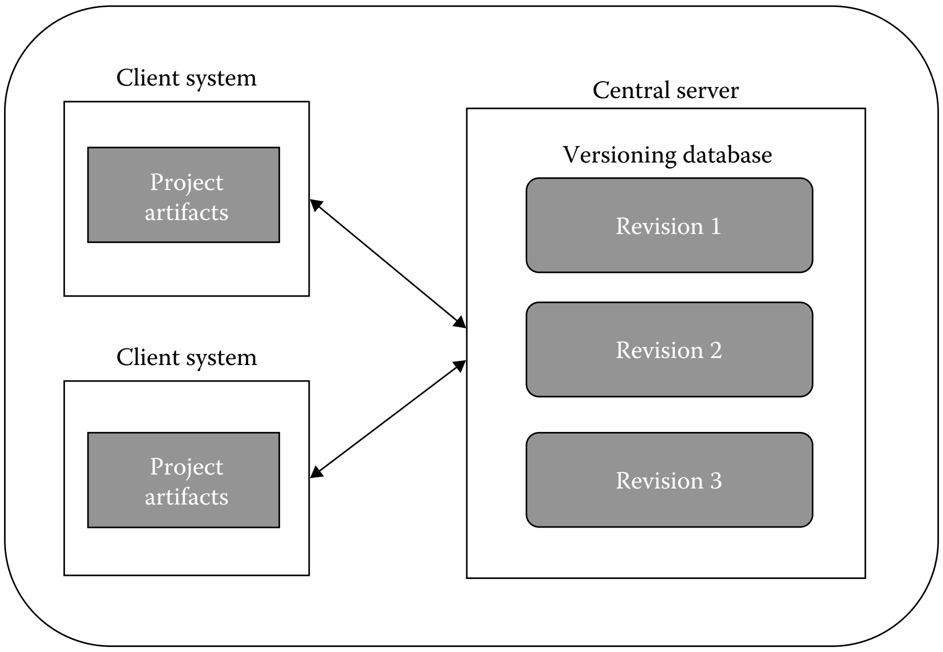

FIGURE 5.8
Centralized version control.

However, if the central server fails or the data stored at central server is corrupted or lost, there are no chances of recovery unless we maintain periodic backups. Figure 5.8 presents the concept of a CVCS.

5.5.2.3 Distributed VCS

To overcome the limitations of CVCS, distributed VCS (DVCS) were introduced. As opposed to CVCS, a DVCS (such as Bazaar, Darcs, Git, and Mercurial) ensures that the clients or users do not just obtain or check out the latest revision or snapshot of the project artifacts, but clone, mirror, or download the entire software project repository to obtain the artifacts.

Thus, if any server of the DVCS fails or its data is corrupted or lost, any of the software project repositories stored at the client machine can be uploaded as back up to the server to restore it. Therefore, every checkout carried out by a client is essentially a complete backup of the entire software project data.

Nowadays, DVCS have earned the attention of various organizations across the globe, and these organizations are relying on them for maintaining their software project repositories. Git is the most popular DVCS employed in practice and hosts a large number of software project repositories. Google and Apache Software Foundation also employ Git to maintain the source code and change control data for their various projects, including Android OS (https://android.googlesource.com), Chromium OS, Chrome browser (https://chromium.googlesource.com), Open Office, log4J, PDFBox, and Apache-Ant, respectively (https://apache.googlesource.com). The concept of a DVCS is presented in Figure 5.9. The figure shows that a copy of entire software project repository is maintained at each client system.

The next section discusses the information maintained by the bug tracking system.

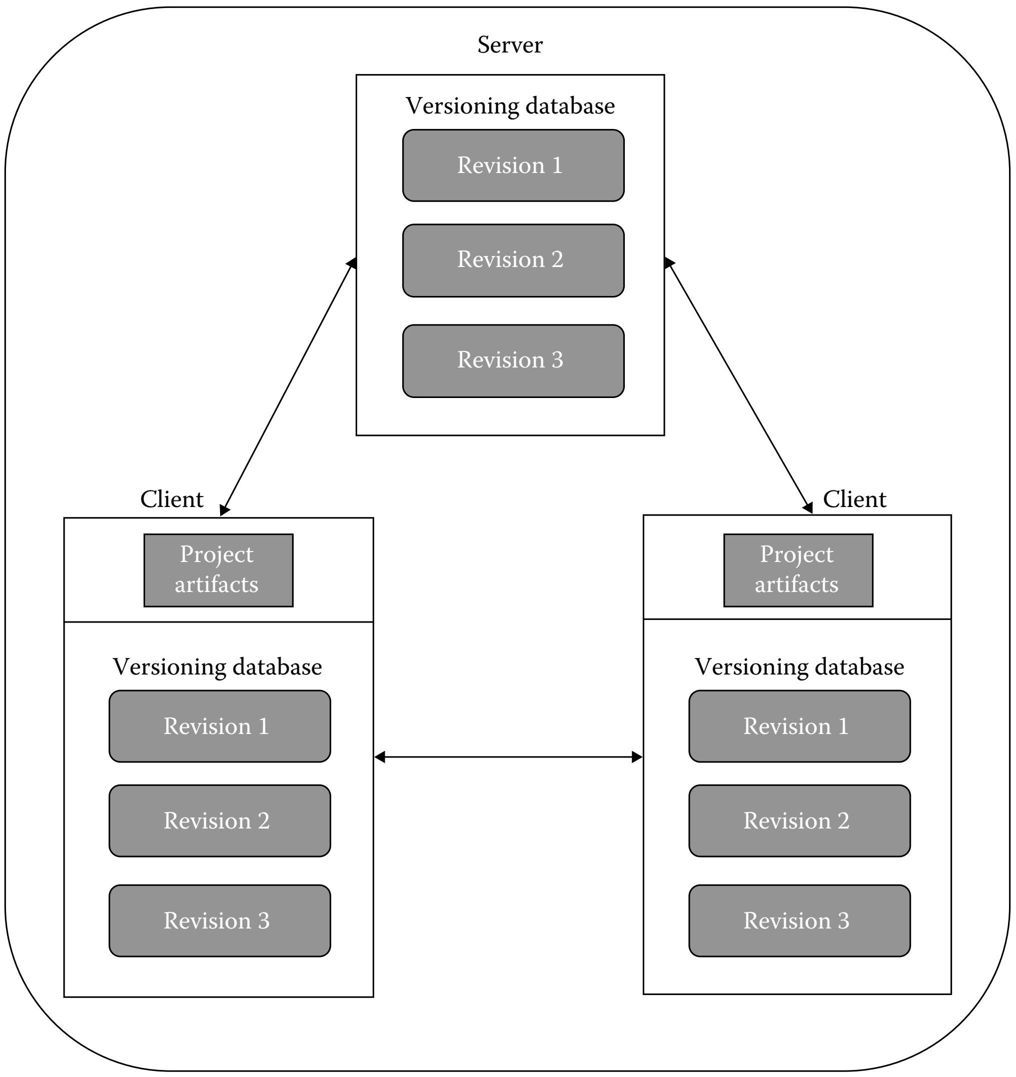

FIGURE 5.9
Distributed version control systems.

5.6 Bug Tracking Systems

A bug tracking system (also known as defect tracking system) is a software system/application that is built with the intent of keeping a track record of various defects, bugs, or issues in software development life cycle. It is a type of issue tracking system. Bug tracking systems are commonly employed by a large number of OSS systems and most of these tracking systems allow the users to generate various types of defect reports directly. Typical bug tracking systems are integrated with other software project management tools and methodologies. Some systems are also used internally by some organizations (http://www.dmoz.org).

A database is a crucial component of a bug tracking system, which stores and maintains information regarding the bugs reported by the users and/or developers. These bugs are generally referred to as known bugs. The information about a bug typically includes the following:

- The time when the bug was reported in the software system
- Severity of the reported bug
- Behavior of the source program/module in which the bug was encountered
- Details on how to reproduce that bug
- Information about the person who reported that bug
- Developers who are possibly working to fix that bug, or will be assigned the job to do so

Many bug tracking systems also support tracking through the status of a bug to determine what is known as the concept of bug life cycle. Ideally, the administrators of a bug tracking system are allowed to manipulate the bug information, such as determining the possible values of bug status, and hence the bug life cycle states, configuring the permissions based on bug status, changing the status of a bug, or even remove the bug information from the database. Many systems also update the administrators and developers associated with a bug through emails or other means, whenever new information is added in the database corresponding to the bug, or when the status of the bug changes.

The primary advantage of a bug tracking system is that it provides a clear, concise, and centralized overview of the bugs reported in any phase of the software development life cycle, and their state. The information provided is valuable for defining the product road map and plan of action, or even planning the next release of a software system (Spolsky 2000).

Bugzilla is one of the most widely used bug tracking systems. Several open source projects, including Mozilla, employ the Bugzilla repository.

5.7 Extracting Data from Software Repositories

The procedure for extracting data from software repositories is depicted in Figure 5.10. The example shows the data-collection process of extracting defect/change reports. The first step in the data-collection procedure is to extract metrics using metrics-collection tools such as understand and chidamber and kemerer java metrics (CKJM). The second step involves collection of bug information to the desired level of detail (file, method, or class) from the defect report and source control repositories. Finally, the report containing the software metrics and the defects extracted from the repositories is generated and can be used by the researchers for further analysis. The data is kept in software repositories in various types such as CVS, Git, SVN, ClearCase, Perforce, Mercurial, Veracity, and Fossil. These repositories are used for management of software content and changes, including documents, programs, user documentation, and other related information. In the next subsections, we discuss the most popular VCS, namely CVS, SVN, and Git. We also describe Bugzilla, the most popular bug tracking system.