

2

Traditional Techniques for Software Fault Localization

Yihao Li¹, Linghuan Hu², W. Eric Wong³, Vidroha Debroy^{3,4}, and Dongcheng Li³

¹ School of Information and Electrical Engineering, Ludong University, Yantai, China

² Google Inc., Mountain View, CA, USA

³ Department of Computer Science, University of Texas at Dallas, Richardson, TX, USA

⁴ Dottid Inc., Dallas, TX, USA

As mentioned in Chapter 1, software fault localization is one of the most expensive activities in program debugging. The high demand for automatic fault localization techniques that can guide programmers to the locations of faults, with minimal human intervention, has fueled the proposal and development of various techniques over recent decades. Slice-based, program spectrum-based, statistics-based, machine learning-based, model-based, and other such techniques have greatly advanced the effectiveness of fault localization. While a thorough discussion of these advanced techniques is quite essential (starting from Chapter 3), it is necessary to review some traditional and intuitive fault localization techniques that have been commonly adopted in debugging research and applications. In this way, we can obtain a comprehensive understanding of progress, issues, and concerns in software fault localization. Therefore, this chapter will focus on the traditional and intuitive fault localization techniques, including program logging, assertions, breakpoints, and profiling.

2.1 Program Logging

Program logging is both a common and an essential software development practice that is used to record vital information regarding a program's execution [1]. This saved information is later used by developers to determine and analyze the

Table 2.1 Additional program spectra relevant to fault localization.

Level	Description
TRACE	Very detailed information
DEBUG	Detailed information on the flow through the system
INFO	Interesting runtime events (such as startups and shutdowns)
WARNING	Runtime oddities and recoverable errors
ERROR	Other runtime errors or unexpected conditions
FATAL	Severe errors causing premature termination

Source: Adapted from log4j [2].

elements of the source code that were present in successful or failed executions. When abnormal program behavior is detected, developers examine the program log in terms of saved log files or printed run-time information to diagnose the underlying cause of failure. Messages can be logged at various levels, which often indicate different levels of severity of the cause and/or verbosity of the logging. For example, Table 2.1 provides a list of common levels from Apache Commons Logging [3]. The sample code in Table 2.2 illustrates a typical usage pattern of Simple Logging Facade for Java (SLF4J) [4], a simple facade or abstraction for various logging frameworks (such as log4j [5], Logback [6], and Java Logging API [7]), which allows the end user to plug in the desired logging framework at deployment time. In addition, Table 2.3 shows a list of popular logging frameworks with their corresponding log levels and supported programming languages. As shown in the table, C logging framework sclog4c has the most (12) log levels, while frameworks

Table 2.2 SLF4J – a logging façade.

```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    final Logger logger = LoggerFactory.getLogger(HelloWorld.class);
    public void doSomething() {
        // some other code
        logger.debug("This is a DEBUG level message");
        logger.info("This is an INFO level message");
        logger.warning("This is a WARNING level message");
        logger.error ("This is an ERROR level message");
        logger.fatal("This is a FATAL level message");
    }
}

```

Source: Adapted from log4j [2].

Table 2.3 Summary of tools used in the fault localization studies.

Framework	Log levels	Language
log4j [5]	FATAL; ERROR; WARN; INFO; DEBUG; TRACE	Java
Java Logging API [7]	SEVERE; WARNING; INFO; CONFIG; FINE; FINER; FINEST	Java
Apache Common Logging [3]	FATAL; ERROR; WARN; INFO; DEBUG; TRACE	Java
SLF4J [4]	ERROR; WARN; INFO; DEBUG; TRACE	Java
tinylog [8]	ERROR; WARNING; INFO; DEBUG; TRACE	Java
Logback [6]	ERROR; WARN; INFO; DEBUG; TRACE;	Java
Object Guy [9]	DEBUG; INFO STATUS; WARNING; ERROR; CRITICAL; FATAL	.Net
Smart Inspect [10]	DEBUG; VERBOSE; MESSAGE; WARNING; ERROR; FATAL	Java; .Net
NLog [11]	DEBUG; TRACE; INFO; WARN; ERROR; FATA	.Net
log4net [12]	DEBUG; INFO; WARN; ERROR; FATA	.Net
Enterprise Library [13]	VERBOSE; INFORMATION; WARNING; ERROR; CRITICAL	.Net
log4c [14]	VERBOSE; INFORMATION; WARNING; ERROR; CRITICAL	C
sclog4c [15]	ALL; FINEST; FINER; FINE; DEBUG; CONFIG; INFO; WARNING; ERROR; SEVERE; FATAL; OFF	C
Syslog [16]	EMERGENCY; ALERT; CRITICAL; ERROR; WARNING; NOTICE; INFORMATION; DEBUG	C
zlog [17]	NOTICE; INFO; WARNING; ERROR; CRITICAL; FATAL; UNKNOWN	C
Pantheios [18]	EMERGENCY; ALERT; CRITICAL; ERROR; WARNING; NOTICE; INFO; DEBUG	C; C++

SLF4J, tinylog, Logback, Enterprise Library, log4c, Syslog, and zlog have the least (5) log levels. The three most commonly used levels are WARN/WARNING (17), INFO/INFORMATION/INFOSTATUS (16), and ERROR (16).

2.2 Assertions

Assertions [19] are constraints added to a program that are required to be true during the correct operation of a program. Developers specify these assertions in the program code as conditional statements that will terminate execution if

```

static void Main (string[] args)
{
    string inputString;
    int minutes;
    do
    {
        Console.WriteLine("Enter the number of minutes to add.");
        inputString = Console.ReadLine();
    } while (!int.TryParse(inputString, out minutes));

    ShowTimePlusMinutes(minutes);
    Console.ReadLine();
}

static void ShowTimePlusMinutes(int minutes)
{
    Debug.Assert(minutes >= 0);
    DateTime time = DateTime.Now.AddMinutes(minutes);
    Console.WriteLine("In {0} minutes it will be {1}", minutes, time);
}

```

Figure 2.1 Debugging using assertion in a C# program. Source: Carr [20]/BlackWasp.

they evaluate to false. Thus, they can be used to detect erroneous program behavior at runtime. For example, Figure 2.1 shows how to create an assertion in C# using the static Assert method of the Debug class [21]. This class is found within the System.Diagnostics namespace. To demonstrate the use of this method, create a new console application and add the following statement to the top of the class containing the Main method: using System.Diagnostics. The simplest syntax for the Debug.Assert method requires only a single Boolean parameter. This parameter contains the predicate that evaluates to true under normal conditions. As shown in the table, the program requests the additional number of minutes from the user. Once a valid number has been entered, the program outputs the resulting time after that number of minutes. In the ShowTimePlusMinutes method, the first line asserts the precondition that the number of minutes is zero or more.

The program in Figure 2.1 contains a bug. The ShowTimePlusMinutes method assumes that the number of minutes has been pre-validated and will be a non-negative integer. However, the Main method has not correctly validated the user input and will allow the number of minutes to be negative. This means that the assertion can be triggered by entering a negative number. Suppose a user executes the program and enters a negative value. As shown in Figure 2.2, the assertion is triggered and a dialog box is displayed. Specifically, the dialog box shows the stack trace at the point where the assertion fails. It provides the user with three options. If the user selects Abort, the program will be immediately halted and closed. If Retry is selected, the program will enter the debug mode to allow the user to

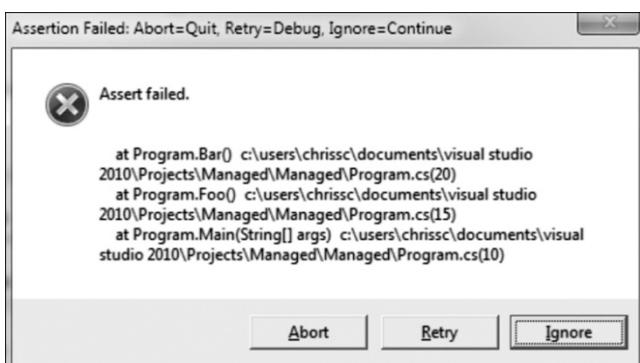


Figure 2.2 A dialog box is displayed when the assertion is triggered. *Source:* Schmich [22].

investigate and go through the code as required. Otherwise, the Ignore option will ignore the assertion and instruct the program to continue execution.

Murillo et al. [6] proposed system-wide assertions, which enable the observation and correlation of Hardware/Software interactions among several cores, devices, and software tasks. A language, SWAT, was developed to create such assertions to facilitate debugging at the system level. Al-Sharif et al. [3] introduced Dynamic Temporal Assertions (DTA) into the conventional source-level debugging session. Each assertion is capable of: (1) validating a sequence of execution states, named the temporal interval, and (2) referencing out-of-scope variables, which may not be live in the execution state at evaluation time. Boos et al. [23] proposed BRACE, a framework that enables developers to correlate cyber and physical properties of the system via assertions. First, BRACE introduces new forms of assertions catering to the unique demands of cyber-physical systems (CPS). For example, CPS assertions can span logical and physical variables and nodes, and specify both spatial and temporal properties. They are checked by an external, omniscient process that can independently observe the physical states. Second, BRACE supports asynchronous checking of CPS assertions to avoid critical timing failures caused by processing latencies. Third, BRACE supports explicit actuation of error handling code. For example, when an assertion is violated, the system can be configured to either halt or execute a user-provided callback function. Fourth, BRACE can be configured to tolerate spatial and temporal discrepancies. Schwartz-Narbonne et al. [15] proposed parallel assertions, a mechanism for expressing correctness criteria in parallel code. These parallel assertions allow users to use intuitive syntax and semantics to express the assumptions in a multi-threaded program involving both the current state and any actions that other parallel executing threads may take (such as accesses to shared memory).

2.3 Breakpoints

Breakpoints are used to pause the program when the execution reaches a specified point, to allow the user to examine the current state. After a breakpoint is triggered, the user can modify the value of variables or continue the execution to observe the progress of the program. Data breakpoints can be configured to trigger when the value changes for a specified expression. Conditional breakpoints pause the execution upon the satisfaction of a predicate specified by the user. These breakpoints can be added to statements in a program or class library [24]. Debugging tools such as GNU GDB [25] and Microsoft Visual Studio Debugger [26] support the use of breakpoints. Take the latter as an example. We will use a simple console application (Figure 2.3) that computes and displays multiplication tables.

A breakpoint can be added by positioning the cursor on the desired line before selecting “Toggle Breakpoint” from the Debug menu. To test this, position the cursor in the first line of code containing a `Console.WriteLine` statement. The line should change color, and you will see a circular icon appear in the gray margin area to the left of the code as shown in Figure 2.4. This icon is called a breakpoint glyph. The filled circle glyph indicates an active, normal breakpoint. When the

```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Multiplication table {0}.\n", i);
    for (int j = 1; j <= 10; j++)
    {
        Console.WriteLine("{0} x {1} = {2}", j, i, i * j);
    }
    Console.WriteLine("\n");
}
```

Figure 2.3 A console application.

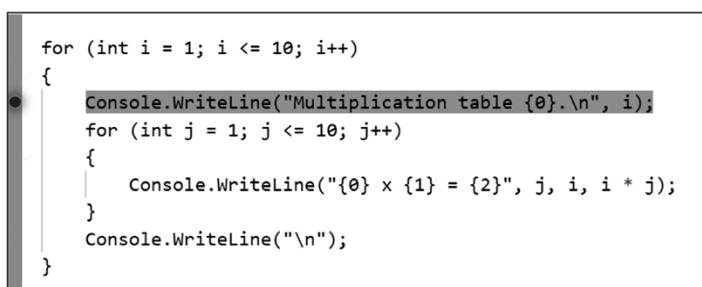
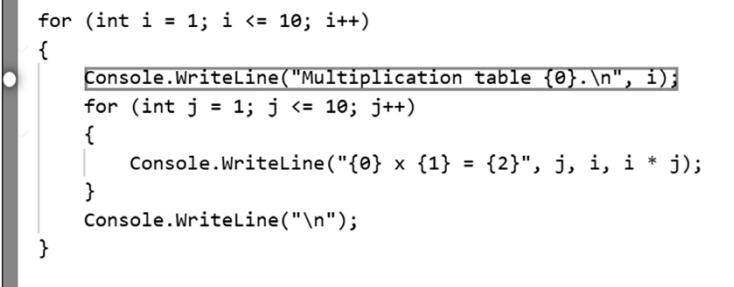


Figure 2.4 Add a breakpoint under Visual Studio.



```
for (int i = 1; i <= 10; i++)
{
    Console.WriteLine("Multiplication table {0}.\n", i);
    for (int j = 1; j <= 10; j++)
    {
        Console.WriteLine("{0} x {1} = {2}", j, i, i * j);
    }
    Console.WriteLine("\n");
}
```

Figure 2.5 Disable a breakpoint under Visual Studio.

above code is executed, the breakpoint location will be hit. The program will enter break mode, allowing users to go through the code using the debugging tools.

If a breakpoint is no longer required, it can be removed using the same actions used to create it. Users can select the marked line of code and choose the “Toggle Breakpoint” command. Alternatively, simply click the breakpoint glyph. When a breakpoint is successfully removed, the glyph disappears as shown in Figure 2.5. When debugging large programs, it is common to set many breakpoints, possibly in several different files. Once the debugging operation is complete, users can remove all of the breakpoints in a project by selecting the “Delete All Breakpoints” option from the Debug menu. If users still want to prevent a breakpoint from halting execution without deleting it completely, they can disable all of the breakpoints in the project by selecting the “Disable All Breakpoints” command from the Debug menu. To disable a single breakpoint, right-click its glyph and choose “Disable Breakpoint” from the context-sensitive menu that appears.

2.4 Profiling

Profiling in software engineering refers to the runtime analysis that measures the performance of a program from different perspectives, such as execution time, CPU/GPU usage, and memory usage, which helps practitioners inspect and optimize the program implementation. Table 2.4 shows a list of profiling tools, such as Eclipse Test & Performance Tools Platform (TPTP) [28], gprof [29], and JVM Monitor [30], which provide comprehensive profiling solutions.

Typically, profiling is used for performance optimization rather than fault localization. However, many fault localization techniques use profiling to collect data for analysis. To reduce the overhead, some advanced profiling techniques are proposed so that the analysis can be conducted in a cost-effective way. One of the most

Table 2.4 A list of profiling tools (some also appear in [27]).

Name	Vendor	Language/Technology
AppDynamics	AppDynamics	Java/.NET/PHP/Node.js/C++/ Python/ Apache Web Server/GoLang
Dynatrace	Dynatrace	Java/.NET/Node.js/AWS/Azure/ Cloud Foundry/OpenShift/Docker/OpenStack/ VMware/OracleDB/MySQL
JProfiler	EJ Technologies	Java
Visual Studio Build-in Profile	Microsoft	C/C++
VisualVM	Oracle	Java
NetBeans Profiler	Oracle	Java
JBoss Profiler	Red Hat	Java
AQTime Pro	SmartBear	Java/.NET/C/C++/Delphi/JScript/JavaScript/ VBScript/Silverlight/ Native 64-bit Application
Prefix	Stackify	Java/.NET
Retrace	Stackify	Java/.NET
YourKit	YourKit	Java/.NET
XRebel	Perforce	Java
GCC	GNU	C
Valgrind	Valgrind	C
Callgrind	Valgrind	C++
KCachegrind	N/A	C
gprof	GNU	C/C++
Shark	Apple	C
gperftools	Google	C
OProfile	N/A	Linux Kernel
VTune	Intel	C/C++/ C#/ Fortran/Java/Python/Go/ASM assembly
TAU	Advanced Computing Laboratory	Fortran/C/C++/UPC/Java/Python
CodeAnalyst	AMD	Java/CLR/Native Code
perf	N/A	Linux System
ANTS	Red Gate	.NET/ASP.NET
Arm Map	Arm Holdings	C/C++/Fortran

Table 2.4 (Continued)

Name	Vendor	Language/Technology
DevPartner Studio	Micro Focus	C/C++/.NET
DTrace	Sun Microsystems	Assembly language /C /C++ /Java/Erlang/JavaScript/Perl/PHP /Python/Ruby/Shell Script/Tcl
FusionReactor	FusionReactor	Java/ColdFusion/MongoDB
GlowCode	Electric Software	C++/ C#/.NET
Intel Advisor	Intel	C/C++/Fortran/C#
LTTng	N/A	C/ C++/Java/Python
Oracle Solaris Studio	Oracle	C/C++/Java/Scala/Fortran
Oracle Developer Studio	Oracle	C/C++/Java/Scala/Fortran
PAPI	Innovative Computing Laboratory	C/C++/Fortran/Java/MATLAB
Chrome DevTools	Google	JavaScript
Firefox Developer Tools	Firefox	JavaScript
Xcode	Apple	Objective-C/Swift
TPTP	IBM	Java
JVM Monitor	N/A	Java

Source: Adapted from List of Performance Analysis Tools [27].

common uses of profiling in fault localization is to provide the program spectrum, which details the execution information of a program from certain perspectives, such as execution information for conditional branches or loop-free intra-procedural paths. For example, Hauswirth and Chilimbi [31] proposed SWAT to identify memory leaks or code that performs unexpectedly poorly. SWAT monitors the subject program's memory allocations for objects and their loads to construct a heap model that can report potential leaks. Based on the assumption that many hidden or longstanding bugs often lurk in some rarely executed locations, an adaptive profiling that samples executions of code segments at a rate inversely proportional to their execution frequency was used to collect the data to improve

detection strength with low overhead. Results indicated that SWAT can identify potential memory leaks as well as their locations in the code. Chilimbi et al. [32] developed a debugging tool, HOLMES, to identify the causes of program failures from failing executions and successful executions. The path profiling was used to collect the execution histories of program paths. To achieve low overhead, an adaptive scheme that only starts profiling after observing a failure was adopted. The data collected were then analyzed to identify the locations that are highly likely to contain bugs. Runciman and Wakeling [33] designed and implemented a tool to profile the content of heap memory such as the memory usage of lazy functional programs. By doing so, it helped users optimize the implementation by reducing the memory consumption as well as identify potential space faults of the lazy functions.

2.5 Discussion

Logging is an important aspect of software development and promotes traceability of actions and operations. Well-written logging code offers quick debugging, easy maintenance, and structured storage of an application's runtime information. However, it can slow down an application if too verbose, which can cause scrolling blindness [5].

The role of assertions is to identify bugs in a program. In practice, the major benefit of assertions is to make testing more effective. An assertion that is never executed does not provide any value. An assertion is only useful if the path containing it is executed [34]. Assuming the code is being properly tested, assertions can aid from the following aspects [34]:

- Detecting subtle errors that might otherwise go undetected.
- Detecting errors soon after they occur.
- Declaratively stating conditions of the code that are guaranteed to be true.

However, there are downsides to assertions. One of the biggest is that, like other code, assertions may themselves contain errors. A bug in an assertion will likely cause one of the following problems [34]:

- Reporting an error where none exists. This kind of bug can lead an unwary programmer down a blind alley. Most programmers will learn to approach assertion failures with the proper skepticism after making such mistakes a few times.
- Failing to report a bug that does exist. This problem is less severe than the first one. If the program uses assertions heavily, it is likely that the error will be caught sooner or later by another assertion. Since an assertion that actually

detects errors is more valuable than one that does not, it is a good idea to simulate failed test cases to identify relevant assertions that actually do fail.

- Not being side-effect free. This problem can be very dangerous. Assertions are not supposed to affect the logic of a program, i.e. the program should run in the same way with or without them. Assertions are usually stripped out of a program before the final program binaries are delivered to customers. However, side effects do sometimes slip in when the assertions are not properly stripped out, or when the assertions are not stripped out because they were thought to be benign but actually change the program behavior. In the worst cases, these will introduce severe bugs to the build released to the customer. One way to avoid these kinds of bugs is to test the release build (with assertions stripped out) before releasing to the customer.

Moreover, assertions can and often do impact the program from nonfunctional perspectives. They can take time to execute and consume extra memory. For example, simple assertions such as the common check for null are relatively inexpensive, but more stringent checks, especially in assertion-heavy code, can measurably slow down the execution, sometimes severely so. Assertions also take up space, and not necessarily just for the code itself. For example, C/C++ assert macros often embed the string representation of the assertion's Boolean expression as well as the filename and line number of the assertion in the source code. This can add up if assertions make up a substantial percentage of the code. Normally, assertions are turned off for builds that are released to actual customers for performance reasons. Sometimes, organizations would rather have a program crash in an unexplained fashion than display a message that an end user will find disturbing (especially if they then have to choose between terminating the application, throwing an exception, or continuing as if nothing happened). When programmers identify an assertion failure, they can either fix the bug or, in extreme cases, disable the assertion. When assertions allow execution to continue (either by default or by user choice), one often runs into exactly the opposite problem, where testing should have been blocked but was not. In this scenario, a tester reports many different symptoms of the original bug as separate bugs. When an exception is thrown to attempt recovery, the situation can be even worse. Sometimes, the throwing of the exception causes more harm than good, resulting in many bugs that would never occur if execution had simply been allowed to continue.

Breakpoints provide a powerful tool that enables users to suspend execution where and when they need to. Rather than stepping through the code line by line or instruction by instruction, the program will run until it hits a breakpoint, and then it starts to debug. This speeds up the debugging process, especially for the case of large programs. Breakpoints can be deleted or changed without having to

change the program's source code. Because breakpoints are not statements, they never produce any extra code when a release version of a program is built [35]. Implementing data breakpoints in software, however, can reduce the performance of the application being debugged, since it is using additional resources on the same processor.

Profilers are great for finding hot paths in code, such as figuring out what uses a large percent of the total CPU usage and determining how to improve that. They help people look for methods that can lead to improvements. They are also useful for finding memory leaks and understanding the performance of dependency calls and transactions. Profilers usually have two different levels: high and low. A high-level profiler tracks performance of key methods in the code. These profilers will perform transaction timing, such as tracking how long a web request takes, while also providing visibility to errors and logs. Low-level profiling can be very slow and has a large amount of overhead. A low-level profiler usually tracks the performance of every single method in the code and potentially every single line of code within each method [36].

2.6 Conclusion

In this chapter, we have reviewed traditional techniques that are commonly used for fault localization, including program logging, assertions, breakpoints, and profiling. Program logging is used to record diagnostic information regarding a program's execution. However, it slows down an application if it is too verbose, which can cause scrolling blindness. Assertions are constraints added to a program that have to be true during the correct operation of a program. They can help detect subtle errors that might otherwise go undetected and speed up error detection after they have occurred. The major challenge for assertions is that they may themselves contain errors, which will cause unexpected problems not in the original code. Using breakpoints allow developers to suspend execution where and when they need to. Using breakpoints may reduce the performance of the application being debugged as they take up additional resources on the same processor. Profiling analyzes the performance of a program from different perspectives and helps practitioners inspect and optimize the program implementation. This process can be very slow and can have a large amount of overhead when using a low-level profiler, which usually tracks the performance of every single method and potentially every single line of code within each method. In general, these techniques focus on manual inspections based on hints and deductions from pre-defined rules or pre-conducted analyses. This leads to the study of a more advanced and automatic process to locate software faults both effectively and efficiently.

References

- 1 Saini, S., Sardana, N., and Lal, S. (2016). Logger4u: predicting debugging statements in the source code. *Proceedings of the 2016 Ninth International Conference on Contemporary Computing (IC3 '16)*, Noida, India (11–13 August 2016), 1–7. IEEE. <https://doi.org/10.1109/IC3.2016.7880255>.
- 2 log4j (2022). Logging levels. https://www.tutorialspoint.com/log4j/log4j_logging_levels (accessed 1 January 2022).
- 3 Al-Sharif, Z.A., Jeffery, C.L., and Said, M.H. (2014). Debugging with dynamic temporal assertions. *Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Naples, Italy (3–6 November 2014), 257–262. IEEE. <https://doi.org/10.1109/ISSREW.2014.60>.
- 4 SLF4J (2022). Simple logging facade for Java (SLF4J). <https://www.slf4j.org> (accessed 1 January 2022).
- 5 log4j (2022). Apache log4j 2. <https://logging.apache.org/log4j/2.x> (accessed 1 January 2022).
- 6 Murillo, L.G., Bücs, R.L., Hincapie, D. et al. (2015). SWAT: assertion-based debugging of concurrency issues at system level. *Proceedings of the 20th Asia and South Pacific Design Automation Conference*, Chiba, Japan (19–22 January 2015), 600–605. IEEE. <https://doi.org/10.1109/ASPDAC.2015.7059074>.
- 7 Java Platform (2022). Package java.util.logging. <https://docs.oracle.com/javase/7/docs/api/java/util/logging/package-summary.html> (accessed 1 January 2022).
- 8 Winandy, M. (2022). tinylog. <https://tinylog.org> (accessed 1 January 2022).
- 9 The Object Guy (2022). Java logging framework. <http://windowsbulletin.com/files/dll/bit-factory-inc/the-object-guy-sLogging-framework> (accessed 1 July 2022).
- 10 Gurock Software (2022). SmartInspect. <https://www.gurock.com/testrail/about> (accessed 1 July 2022).
- 11 NLog (2022). NLog. <https://nlog-project.org> (accessed 1 July 2022).
- 12 log4net (2022). Apache log4net. <http://logging.apache.org/log4net> (accessed 1 January 2022).
- 13 Microsoft (2022). Enterprise library. https://en.wikipedia.org/wiki/Microsoft_Enterprise_Library (accessed 1 July 2022).
- 14 log4c (2022). Log4c: logging for C library. <http://log4c.sourceforge.net> (accessed 1 January 2022).
- 15 Schwartz-Narbonne, D., Liu, F., Pondicherry, T. et al. (2011). Parallel assertions for debugging parallel programs. *Proceedings of the 9th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE '11)*, Cambridge, UK (11–13 July 2011), 181–190. IEEE. <https://doi.org/10.1109/MEMCOD.2011.5970525>.
- 16 Stackify (2022). Syslog tutorial: how it works, examples, best practices, and more. <https://stackify.com/syslog-101> (accessed 1 January 2022).

- 17 Simpson, H. (2022). zlog: a reliable pure C logging library. <http://hardysimpson.github.io/zlog> (accessed 1 January 2022).
- 18 Pantheios (2022). Pantheios. <http://www.pantheios.org/index.html> (accessed 1 January 2022).
- 19 Rosenblum, D.S. (1995). A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* 21 (1): 19–31. ISSN 0098-5589. <https://doi.org/10.1109/32.341844>.
- 20 Carr, R. (2008). Debugging using assertions. <http://www.blackwasp.co.uk/DebugAssert.aspx> (accessed 1 January 2022).
- 21 Coutant, D.S., Meloy, S., and Ruscetta, M. (1988). DOC: a practical approach to source-level debugging of globally optimized code. *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*, Atlanta, GA, USA (22–24 June 1988), 125–134. ACM. <http://doi.acm.org/10.1145/53990.54003>. ISBN 0-89791-269-1.
- 22 Schmich, C. (2010). Insert line number in Debug.Assert statement. <https://stackoverflow.com/questions/3868644/insert-line-number-in-debug-assert-statement> (accessed 1 January 2022).
- 23 Boos, K., Fok, C., Julien, C., and Kim, M. (2012). BRACE: an assertion framework for debugging cyber-physical systems. *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland (2–9 June 2012), 1341–1344. IEEE. <https://doi.org/10.1109/ICSE.2012.6227084>.
- 24 BlackWasp (2022). Breakpoints and tracepoints in visual studio. <http://www.blackwasp.co.uk/VSSBreakpoints.aspx> (accessed 1 January 2022).
- 25 GNU (2022). GDB: the GNU project debugger. <http://www.gnu.org/software/gdb> (accessed 1 January 2022).
- 26 Microsoft (2022). Debugging in visual studio. <https://msdn.microsoft.com/en-us/library/sc65sadd.aspx> (accessed 1 January 2022).
- 27 List of Performance Analysis Tools (2022). https://en.wikipedia.org/wiki/List_of_performance_analysis_tools (accessed 1 July 2022).
- 28 Eclipse-Foundation (2022). Test and performance tools platform. <https://projects.eclipse.org/projects/tptp.platform/developer> (accessed 1 January 2022).
- 29 GNU (2022). GNU gprof. <http://sourceware.org/binutils/docs/gprof> (accessed 1 January 2022).
- 30 JVMMonitor project (2022). JVMMonitor: Java profiler integrated with Eclipse. <http://jvmmonitor.org> (accessed 1 January 2022).
- 31 Hauswirth, M. and Chilimbi, T.M. (2004). Low-overhead memory leak detection using adaptive statistical profiling. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, Boston, MA, USA (7–13 October 2004), 156–164. ACM. <http://doi.acm.org/10.1145/1024393.1024412>. ISBN 1-58113-804-0.

- 32** Chilimbi, T.M., Liblit, B., Mehra, K. et al. (2009). HOLMES: effective statistical debugging via efficient path profiling. *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE '09)*, Vancouver, BC, Canada (16–24 May 2009), 34–44. IEEE. <https://doi.org/10.1109/ICSE.2009.5070506>.
- 33** Runciman, C. and Wakeling, D. (1993). Heap profiling of lazy functional programs. *Journal of Functional Programming* 3 (2): 217–245. <https://doi.org/10.1017/S0956796800000708>.
- 34** Cary, D. (2022). What are assertions. <http://wiki.c2.com/?WhatAreAssertions> (accessed 1 January 2022).
- 35** Microsoft (2022). Debugging basics: breakpoints. <https://docs.microsoft.com/en-us/visualstudio/debugger/using-breakpoints?view=vs-2022> (accessed 1 July 2022).
- 36** Stackify (2022). What is code profiling? Learn the 3 types of code profilers. <https://stackify.com/what-is-code-profiling> (accessed 1 January 2022).

CHAPTER TWENTY-FOUR

The Art of Collecting Bug Reports

*Rahul Premraj
Thomas Zimmermann*

Kids love bugs, and some kids even collect bugs and keep them in precious “kill jars.” Over a period of time, bug collectors can amass a large number of different species of bugs. Some kids study the bugs they collected and label them based on such characteristics as shape, size, color, number of legs, whether it can fly, and so on. The bugs may be valued differently depending upon how rare they are or how difficult they are to catch. The collection may have some duplicate bugs. But duplicates are rarely identical, as characteristics such as appearance and size can differ widely.

But we software developers do not like bugs. We hope to have none in our software, and when they are found, we squash them! Unfortunately, squashing bugs, or more politely, responding to software change requests, is rarely easy. Developers have to study the information about the bug in detail, conduct a thorough investigation on how to resolve the bug, examine its side effects, and eventually decide on and take a course of action. This is a difficult task because, like earthly bugs, software bugs differ widely. Often software bugs that are collected in bug databases of projects are studied in isolation because they are different from the other bugs in their effects on the software system, their cause and location, and their severity. Over time, a project will accumulate duplicate bugs, just as a live-bug collector may have multiple bugs of the same species. And finally, nearly every project knows about more bug reports than it can fix, just as there are too many live bugs to be collected by a single person.

So the study of software bugs is a valuable pastime. The starting point for the study of software bugs is the information filed in bug reports by those who report them. In this chapter, we discuss the characteristics of good bug reports and implications for the practice of collecting bug reports.

Good and Bad Bug Reports

When some users of a software system encounter a bug, they report it to developers with the hope that the bug will be fixed. The information in a bug report gives the developers a detailed description of the failure and occasionally hints at the cause.

But the quality and level of detail of information can vary a lot from one bug report to another. Take for example the following bug report (#31021) from the Eclipse project:

120030205

Run the following example. Double click on a tree item and notice that it does not expand.

Comment out the Selection listener and now double click on any tree item and notice that it expands.

```
public static void main(String[] args){  
    Display display = new Display();  
    Shell shell = new Shell(display);  
    [...] (21 lines of code removed)  
    display.dispose();  
}
```

The reporter provides a code example and concise steps on how to run it in order to reproduce the bug. Once a developer can reproduce and observe the bug, it is likely to make the process of investigating the cause of the bug comparatively easier.

On the other hand, the following bug report (#175222), again from the Eclipse project, is not in fact a bug at all and has been misfiled as one:

I want to create a new plugin in Eclipse using CDT. Shall it possible.
I had made a RD in eclipse documentation. I had get an idea about
create a plugin using Java. But i want to create a new plugin (user
defined plugin) using CDT. After that I want to implement it in my
programme. If it possible?. Any one can help me please...

The quality of information in bug reports can crucially influence the resolution of a bug as well as its resolution time. Reports that contain all the necessary information can make resolving the bug somewhat easier. In contrast, reports with inadequate information may lead to avoidable delays when developers find themselves filling the gaps in information or contacting reporters to request more information.

It's clear that high-quality information in bug reports would be in the interest of everyone involved. Developers might be able to resolve more and more bugs in shorter periods of time, and reporters would have their bugs fixed quickly. But what information in a bug report enhances its quality? We describe our search for the answer in this chapter.

What Makes a Good Bug Report?

Through an online survey, we asked over 150 developers from three large and successful open source projects—Apache, Eclipse, and Mozilla—what information in bug reports they consider valuable and helpful when resolving bugs. We also contacted bug reporters from the same projects to find out what information they provide and what information is most difficult to provide.

Our online survey is presented in [Table 24-1](#). The survey sent out to developers comprised four questions in two categories:

Contents of bug reports

- (D1) Which items have developers previously used when fixing bugs?
- (D2) Which three items helped the most?

Insight into this issue can help develop guides or tools for reporters to provide information in bug reports that focuses on the details that are most important to developers. We provided the developers with 16 items to choose from, some selected from the Mozilla bug-writing guidelines* and others found as standard fields in the Bugzilla database. Developers were free to check as many items as they liked for the first question (D1), but at most three for the second question (D2), thus indicating the importance of the choices in their perspectives.

Problems with bug reports

- (D3) Which problems have developers encountered when fixing bugs?
- (D4) Which three problems caused the most delay in fixing bugs?

Our motivation for these questions was to find the prominent obstacles faced by developers that can be tackled in the future by more cautious, and perhaps even automated, bug reporting. Typical problems include reporters accidentally providing incorrect information, such as the wrong operating system.

* The Mozilla bug-writing guidelines describe principles of effective bug reporting (e.g., be precise and clear, one bug per report) and list information that is essential to every bug report, such as steps to reproduce it, actual results, and expected results [[Goldberg 2010](#)].

Other problems in bug reports include poor use of language (ambiguity), bug duplicates, and incomplete information. Spam recently has become a problem, especially for the TRAC issue tracking system. We decided not to include the problem of incorrect assignments to developers, because bug reporters have little influence on the triaging of bugs. In total, we provided 21 problems that developers could select. Again, they were free to check as many items for the third question (D3), but at most three for the fourth question (D4). For a complete list of items, see [Table 24-1](#).

We asked bug reporters the following three questions, divided into two categories (again, see [Table 24-1](#)):

Contents of bug reports

- (R1) Which items have reporters previously provided?
- (R2) Which three items were most difficult to provide?

We offered the same 16 items to reporters that we offered to developers. This allowed us to check whether the information provided by reporters is in line with what developers frequently use or consider to be important (by comparing the responses for R1 with D1 and D2). The second question helped us to identify items that are difficult to collect and for which better tools might support reporters in this task. Reporters were free to check as many items as they wanted for the first question (R1), but at most three for the second question (R2).

Contents considered to be relevant

- (R3) Which three items do reporters consider to be most relevant for developers?

Again we listed the same items to see how much reporters agree with developers (comparing R3 with D2). For this question (R3), reporters were free to check at most three items, but could choose any item, regardless of whether they selected it for question R1.

Additionally, we asked both developers and reporters about their thoughts and experiences with respect to bug reports (D5 and R4).

TABLE 24-1. The questionnaire presented to Apache, Eclipse, and Mozilla developers (Dx) and reporters (Rx)

Contents of bug reports.	D1: Which of the following items have you previously used when fixing bugs? D2: Which three items helped you the most? R1: Which of the following items have you previously provided when reporting bugs? R2: Which three items were the most difficult to provide? R3: In your opinion, which three items are most relevant for developers when fixing bugs?
	<input type="checkbox"/> product <input type="checkbox"/> hardware <input type="checkbox"/> observed behaviour <input type="checkbox"/> screenshots <input type="checkbox"/> component <input type="checkbox"/> operating system <input type="checkbox"/> expected behaviour <input type="checkbox"/> code examples <input type="checkbox"/> version <input type="checkbox"/> summary <input type="checkbox"/> steps to reproduce <input type="checkbox"/> error reports <input type="checkbox"/> severity <input type="checkbox"/> build information <input type="checkbox"/> stack traces <input type="checkbox"/> testcases

Problems with bug reports.	D3: Which of the following problems have you encountered when fixing bugs? D4: Which three problems caused you most delay in fixing bugs?			
	You were given:	There were errors in:	The reporter used:	Others:
	<input type="checkbox"/> product name	<input type="checkbox"/> code examples	<input type="checkbox"/> bad grammar	<input type="checkbox"/> duplicates
	<input type="checkbox"/> component name	<input type="checkbox"/> steps to reproduce	<input type="checkbox"/> unstructured text	<input type="checkbox"/> spam
	<input type="checkbox"/> version number	<input type="checkbox"/> test cases	<input type="checkbox"/> prose text	<input type="checkbox"/> incomplete information
	<input type="checkbox"/> hardware	<input type="checkbox"/> stack traces	<input type="checkbox"/> too long text	<input type="checkbox"/> viruses/worms
	<input type="checkbox"/> operating system		<input type="checkbox"/> non-technical language	
	<input type="checkbox"/> observed behaviour		<input type="checkbox"/> no spell check	
	<input type="checkbox"/> expected behaviour			
Comments.	D5/R4: Please feel free to share any interesting thoughts or experiences.			

Survey Results

The results of our survey are summarized in [Figure 24-1](#) (for developers) and [Figure 24-2](#) (for reporters).

In the figures, responses for each item are annotated as bars (), which can be interpreted as follows (explained with D1 and D2 as examples):

- The colored part () denotes the count of responses for an item in question D1.
- The black part () of the bar denotes the count of responses for the item in both question D1 and D2.

The larger the black bar is in proportion to the gray bar, the higher is the corresponding item's importance in the developers' perspective. The importance of every item is also listed in parentheses. For example, [Figure 24-1](#) shows that developers consider steps to reproduce (, 83%) to be more important than build information (, 8%).

Contents of Bug Reports (Developers)

[Figure 24-1](#) shows that the most widely used items across projects are steps to reproduce the bug, observed and expected behavior, stack traces, and test cases. Information rarely used by developers include hardware and severity. Eclipse and Mozilla developers liked screenshots, whereas Apache and Eclipse developers more often used code examples and stack traces.

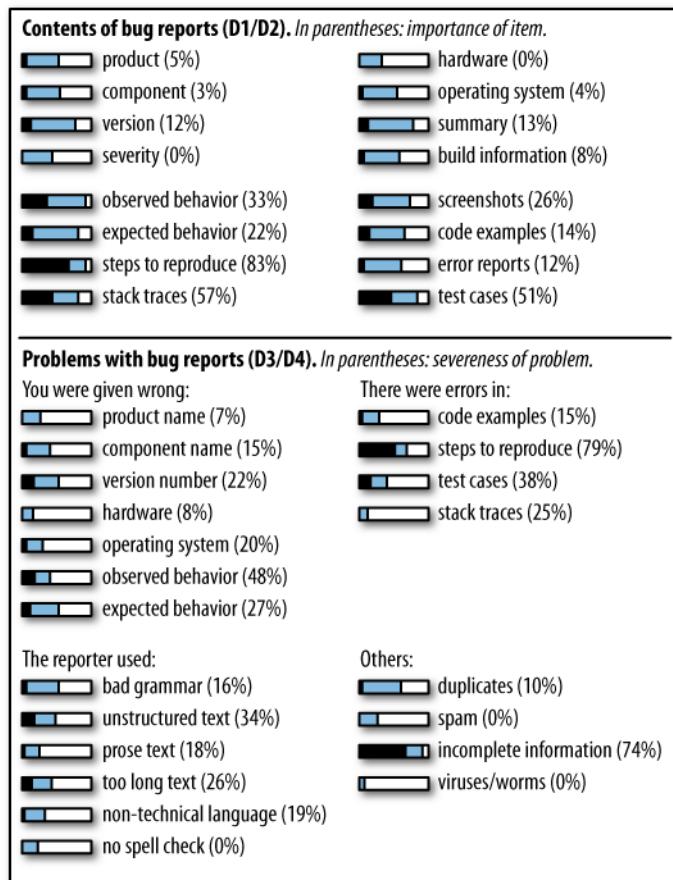


FIGURE 24-1. Results from the survey among developers (130 consistent responses by Apache, Eclipse, and Mozilla developers)

For the importance of items, steps to reproduce the bug stand out clearly as most important. Next in line are stack traces and test cases, both of which help to narrow down the search space for defects. Observed behavior mimics, albeit weakly, steps to reproduce the bug, which is why it may be rated as important. Screenshots were rated as high, but often are helpful only for a subset of bugs, e.g., GUI errors.

Smaller surprises in the results include the relative low importance of items such as expected behavior, code examples, summary, and mandatory fields such as version, operating system, product, and hardware. As pointed out by a Mozilla developer, not all projects need the information that is provided by mandatory fields:

“That’s why product and usually even component information is irrelevant to me and that hardware and to some degree [OS] fields are rarely needed as most our bugs are usually found in all platforms.”

In any case, we advise caution when interpreting these results. Items with low importance in our survey are not totally irrelevant, because they still might be needed to understand, reproduce, or triage bugs.

Contents of Bug Reports (Reporters)

The items provided by most reporters are listed in the first part of [Figure 24-2](#). As expected, observed and expected behavior and steps to reproduce the bug rank highest. Only a few users added stack traces, code examples, and test cases to their bug reports. An explanation might be the difficulty of providing these items, which we measured by the percentage of people who selected an item as difficult to provide in bug reports; the difficulty is listed in parentheses. All three items rank among the more difficult items, with test cases being the most difficult item. Surprisingly, steps to reproduce the bug are considered difficult as well, as is the component. For the latter, reporters revealed in their comments that often it is impossible for them to locate the component in which a bug occurs.

Among the items considered to be most helpful to developers, reporters ranked steps to reproduce the bug and test cases highest. Comparing the results for test cases among all three questions reveals that most reporters consider them to be helpful, but only a few provide them because they are most difficult to provide. This suggests that capture/replay tools that record test cases should be integrated into bug tracking systems. A similar but weaker observation can be made for stack traces, which are often hidden in logfiles and difficult to find. On the other hand, both developers and reporters consider the component to be only marginally important; furthermore, as already mentioned, it is rather difficult to provide.

Evidence for an Information Mismatch

We compared the results from the developer and reporter surveys to find out whether they agree on what is important in bug reports.

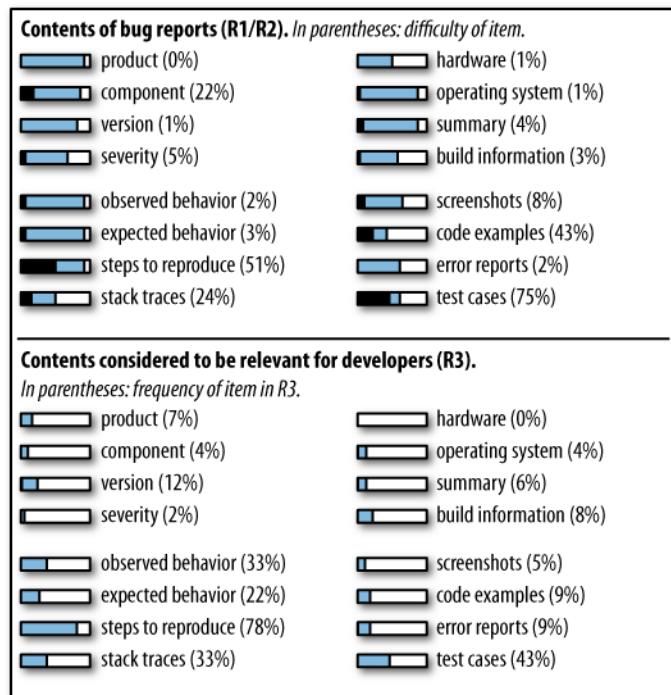


FIGURE 24-2. Results from the survey among reporters (215 consistent responses by Apache, Eclipse, and Mozilla reporters)

First, we compared the information developers use to resolve bugs (question D1) and the information reporters provide (R1). In Figure 24-3, items in the left column are sorted decreasingly by the percentage of developers who have used them, and items in the right column are sorted decreasingly by the percentage of reporters who have provided them. Lines connect the same items across columns and indicate the agreement (or disagreement) between developers and reporters on that particular item. Figure 24-3 shows that the results match only for the top three items and the last one. In between there are many disagreements, and the most notable ones are for stack traces, test cases, code examples, product, and operating system. Overall, the Spearman correlation[†] between what developers use and what reporters provide was 0.321, far from ideal.

† Spearman correlation is a measure of strength of the association between two variables. Its value ranges from -1 to +1. Values closer to -1 or +1 indicate a strong relationship, while 0 suggests there is no relationship between the variables. The sign indicates whether the association is in the same (+) or opposite (-) directions.

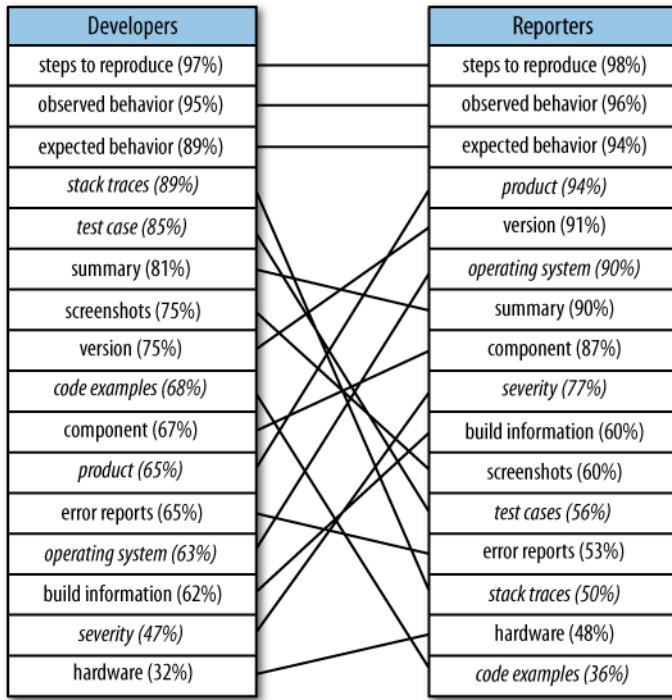


FIGURE 24-3. Used by developers versus provided by reporters

Next, we checked whether reporters provide the information that is most important for developers. In Figure 24-4, the left column corresponds to the importance of an item for developers (measured by questions D2 and D1), and the right column to the percentage of reporters who provided an item (R1). Developers and reporters still agree on the first and last item, but overall the disagreement increased. The Spearman correlation of -0.035 between what developers consider as important and what reporters provide shows a huge gap. In particular, it indicates that reporters do not focus on the information important for developers.

Interestingly, Figure 24-5 shows that most reporters know which information developers need. In other words, ignorance among reporters cannot be blamed for the aforementioned information mismatch. As before, the left column corresponds to the importance of items for developers; the right column now shows what reporters expect to be most relevant (question R3). Overall there is a strong agreement; the only notable disagreement is for screenshots. This is confirmed by the Spearman correlation of 0.839, indicating a very strong relation between what developers and reporters consider as important.

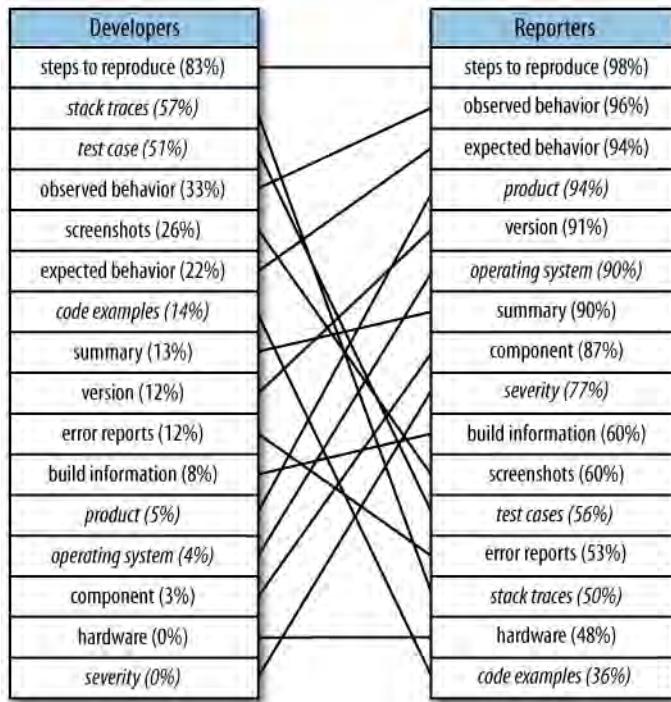


FIGURE 24-4. Most helpful for developers versus most often provided by reporters

To sum up, to improve bug reporting systems, one could tell users while they are reporting a bug what information is important (e.g., screenshots). At the same time, systems should provide better tools to collect important information, because often this information is difficult to obtain for users.

Problems with Bug Reports

Among the problems experienced by developers, incomplete information was, by far, most commonly encountered. Other common problems include errors in steps to reproduce the bug, errors in test cases, bug duplicates, incorrect version numbers, and incorrect observed and expected behavior. Another issue that often challenges developers is the reporter's language fluency. These problems can easily lead developers astray when fixing bugs.

The most severe problems were errors in the steps to reproduce the bug and incomplete information. In fact, in question D5 many developers commented on being plagued by bug reports with incomplete information. As one developer commented:

“The biggest causes of delay are not wrong information, but absent information.”

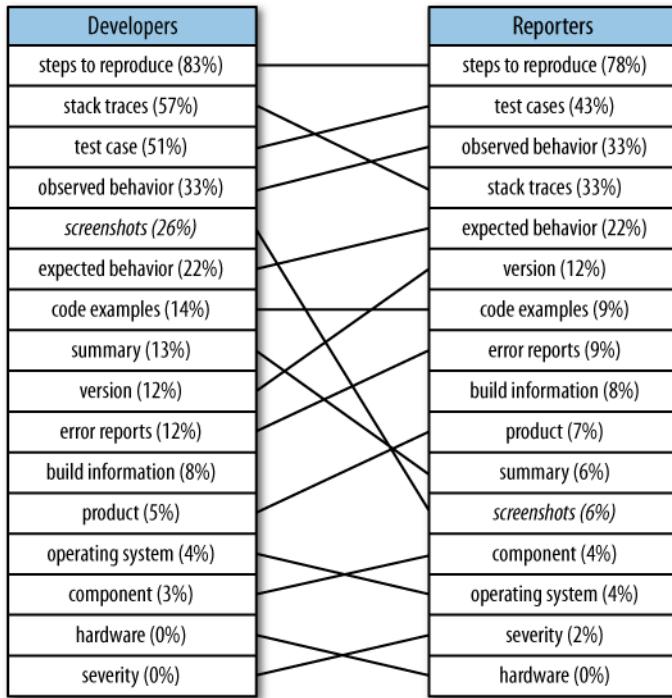


FIGURE 24-5. Most helpful for developers versus expected by reporters to be helpful

The low occurrence of spam is not surprising, because in Bugzilla and Jira, reporters have to register before they can submit bug reports, and this registration successfully prevents spam. Finally, errors in stack traces are highly unlikely because they are copy-pasted into bug reports, but when they do happen they can be a severe problem.

Other major problems included errors in test cases and observed behavior. A very interesting observation is that developers do not suffer too much from bug duplicates, although earlier research considered this to be a problem [Anvik et al. 2005]. Perhaps developers can easily recognize duplicates, and sometimes even benefit from a different bug description. The following section provides an extensive study of the value of duplicate bug reports.

The Value of Duplicate Bug Reports

A common argument against duplicate bug reports is that they strain bug tracking systems and demand more effort from quality assurance teams—effort that could instead be spent to improve the product. In this section, we provide empirical evidence for the contrary: duplicate bug reports actually contain additional information that may be useful to resolve bugs.

When a bug report is identified as a duplicate, a common practice is to simply close the bug and discard the information, which in the long term discourages users from submitting bug reports. They become reluctant to provide additional information, once they see that a bug report has already been filed:

“Typically bugs I have reported are already reported but by much less savvy people who make horrible reports that lack important details. It is frustrating to have spent lots of time making an exceptionally detailed bug report to only have it marked as a duplicate...”

Not everyone agrees that bug duplicates are bad. In our survey, several developers pointed out the value of bug duplicates for resolving bugs:

“Duplicates are not really problems. They often add useful information. That this information were filed under a new report is not ideal though.”

“It would be better to somehow mend the reports instead of just writing off the good report simply because it was posted after the bad report. This would probably help software engineers much more.”

Alan Page, a test architect at Microsoft, makes a similar argument and summarizes three reasons why “worrying about [duplicates] is bad” [Page et al. 2008]:

- Often there are negative consequences for users who enter duplicates. As a result, *they might err on the side of not entering a bug*, even though it is not filed yet.
- *Triagers are more skilled in detecting duplicates* than users, and they also know the system better. Whereas a user will need a considerable amount of time to browse through similar bugs, triagers can often decide within minutes whether a bug report is a duplicate.
- *Bug duplicates can provide valuable information* that helps diagnose the actual problem.

Do bug duplicates really provide extra information, and if so, how much? We found out with an empirical study. First, we built a tool to detect and quantify information items such as patches, screenshots, and stack traces in bug reports. Next, we compared the amount of information found in the original bug report (called the *master report*), and the *extended report*, which combines the original master bug report with its duplicates.

Table 24-2 summarizes our findings for the Eclipse project. The first column, “Information item,” presents all the pieces of information that we commonly found and extracted from bug reports. The items fall in four categories: predefined fields such as product and component, patches, screenshots, and stack traces. The second column, “Master,” lists the average count of each information item in the original bug reports. For example, every master report contains exactly one operating system, as indicated by the 1.000 value in the “Master” column. This count corresponds to a practice that is found in many projects: once duplicates are detected, they are simply closed and all information that they provided is discarded.

The third column, “Extended,” lists the average count of each information item in the extended bug reports. When merged with their duplicates, the average number of unique operating systems in extended bug reports increases to 1.631. This count would correspond to a practice where duplicates are merged with master reports and all information is retained. The fourth column, “Change,” calculates the increase from “Master” to “Extended” and quantifies the amount of information added by bug duplicates per master report. For example, duplicates could add on average 0.631 operating systems to existing bugs as long as duplicates are not just discarded.

TABLE 24-2. Average amount of information added by duplicates per master report in Eclipse

Information item	Average count per master report		
	Master	Extended	Change ^a
Predefined fields			
Product	1.000	1.127	+0.127 (12.7%)
Component	1.000	1.287	+0.287 (28.7%)
Operating system	1.000	1.631	+0.631 (63.1%)
Reported platform	1.000	1.241	+0.241 (24.1%)
Version	0.927	1.413	+0.486 (52.4%)
Reporter	1.000	2.412	+1.412 (41.2%)
Priority	1.000	1.291	+0.291 (29.1%)
Target milestone	0.654	0.794	+0.140 (21.4%)
Patches			
Total	1.828	1.942	+0.113 (6.2%)
Unique: patched files	1.061	1.124	+0.062 (5.9%)
Screenshots			
Total	0.139	0.285	+0.145 (105.0%)
Stacktraces			
Total	0.504	1.422	+0.918 (182.1%)
Unique: exception	0.195	0.314	+0.118 (61.0%)
Unique: exception, top frame	0.223	0.431	+0.207 (93.3%)
Unique: exception, top 2 frames	0.229	0.458	+0.229 (100.0%)
Unique: exception, top 3 frames	0.234	0.483	+0.248 (106.4%)
Unique: exception, top 4 frames	0.239	0.504	+0.265 (110.9%)
Unique: exception, top 5 frames	0.244	0.525	+0.281 (115.2%)

^a For all information items, the increase is significant at p < .001.

Table 24-2 shows that most duplicates are filed by different users from those who filed the original master report, which explains the large increase in the number of unique reporters.

We found that duplicates add on average only 0.113 patches and 0.062 patched files per master report. This is a rather small relative increase of about 6% and suggests that most patches are filed against the master report. However, bug duplicates add on average 0.145 screenshots, which doubles the number of screenshots. Duplicates also provide substantial additional information for version, priority, and component.

We compared stack traces by considering the exception that triggered them and the first five stack frames. For Eclipse on average, 0.918 additional stack traces were found in the duplicate bug reports (an increase of 182%). Within these, we found on average 0.118 occurrences (61%) of additional exceptions in duplicates and 0.281 stack traces (115.2%) that contained code locations (in the top five frames) that had not been reported before.

These findings show that duplicates are likely to provide different perspectives and additional pointers to the origins of bugs and thus can help developers correct them. For example, having more stack traces reveals more active methods during a crash, which helps to narrow down the suspects. These findings make a case for reevaluating the treatment and presentation of duplicates in bug tracking systems.

Not All Bug Reports Get Fixed

Most projects have more bug reports that can be fixed given limited resources and time. To characterize how the quality of bug reports increases the chances of bugs getting fixed, we sampled 150,000 bugs from Apache, Eclipse, and Mozilla (50,000 per project). These bugs had various resolutions, such as FIXED, DUPLICATE, MOVED, WONTFIX, and WORKSFORME. We divided the bug reports into two groups, successful and not successful, and then used statistical tests such as Chi-Square and Kruskal-Wallis ($p < .05$) to check for relationships between the success of bug reports and the presence of information items (code samples, stack traces, patches, screenshots). Our comparisons were:

Resolution of bug reports

We compared bug reports resolved as FIXED against bug reports with other resolutions, such as WONTFIX and WORKSFORME. We treated duplicate bug reports as a separate group because for some the master report is fixed, whereas for others the master report is not fixed.

Lifetime of bug reports

We compared bug reports with a short lifetime against bug reports with a long lifetime. A fast turnaround for bug reports is desirable, especially for users, but also for developers.

We also checked for the influence of readability on the success of bug reports. To measure the readability of bug reports, we used the *Style* tool, which “analyses the surface characteristics of the writing style of a document” [Cherry and Vesterman 1981]. The readability of a text is measured by the number of syllables per word and the length of sentences. Readability measures are used by Amazon.com to inform customers about the difficulty of books and by

the U.S. Navy to ensure readability of technical documents. For our experiments we used the following seven readability measures: Kincaid, Automated Readability Index (ARI), Coleman-Liau, Flesh, Fog, Lix, and SMOG Grade.

Our findings from this experiment are:

- Bug reports with stack traces get fixed sooner. (Apache, Eclipse, and Mozilla)
- Bug reports that are easier to read get fixed sooner. (Apache, Eclipse, and Mozilla)
- Including code samples in a bug report increases the chances of it getting fixed. (Mozilla)

Independently from us, Hooimeijer and Weimer observed for Firefox that bug reports with attachments get fixed later, and bug reports with many comments get fixed sooner [[Hooimeijer and Weimer 2007](#)]. They also confirmed our results that easy-to-read reports are fixed faster. Panjer observed for Eclipse that comment count and activity as well as severity have the most effect on bugs' lifetimes [[Panjer 2007](#)]. Schröter et al. validated our finding that bugs with stack traces get fixed sooner and further emphasized the importance of adding stack traces to bug reports [[Schröter et al. 2010](#)]. A study conducted by Guo et al. at Microsoft for Windows Vista and Windows 7 [[Guo et al. 2010](#)] found that number of reassignments, organizational and geographical distribution, and reputation of bug opener influence the chances of bug reports getting fixed.

The findings in this section show that well-written bug reports help in comprehending the problem better, consequently increasing the likelihood of the bug getting fixed in a shorter time.

Conclusions

Bug reports are central to maintaining the quality of software systems. Their information contents help developers identify the cause of the failure and resolve it. But for this to happen, it is important that the information in the bug reports be reliable and complete.

But what makes a good bug report? In this chapter, we have presented the findings from a survey of developers and bug reporters from three large open source software products to identify which contents contribute to the information quality of bug reports from the perspective of those surveyed. Responses to our survey sent to developers indicated the importance of stack traces, steps to reproduce the bug, and expected and observed behavior in resolving bugs. Reporters indicated a similar set of important items as useful to help resolve bugs, but interestingly their responses also showed that they frequently don't provide such information, i.e., there is a mismatch between what information developers expect and what reporters provide.

Bug tracking systems can be enhanced to mitigate such information mismatch between developers and reporters. Several automations can be built in for data collection (such as stack traces from crashes), and the interfaces can be improved to remind reporters to add certain information. Our extensive study on bug duplicates has shown that duplicate bug reports often contain additional and useful information, so perhaps such new information should be automatically merged with the original bug reports. Offering incentives, such as listing the advantages of including certain types of information (“Bug reports with stack traces get fixed sooner”), may encourage bug reporters to go the extra mile to improve their reports.

All in all, the better the quality of bug reports, the more likely the bug will get fixed soon. Meanwhile, happy bug reporting and fixing!

Acknowledgments

This chapter is based on research work conducted by us jointly with Nicolas Bettenburg, Sascha Just, Sunghun Kim, Adrian Schröter, and Cathrin Weiss.

References

- [Anvik et al. 2005] Anvik, John, Lyndon Hiew, and Gail C. Murphy. 2005. Coping with an open bug repository. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*: 35–39.
- [Bettenburg et al. 2008a] Bettenburg, Nicolas, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*: 308–318.
- [Bettenburg et al. 2008b] Bettenburg, Nicolas, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. 2008. Duplicate Bug Reports Considered Harmful...Really?. *Proceedings of the 24th International Conference on Software Maintenance*.
- [Breu et al. 2010] Breu, Silvia, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*: 301–310.
- [Cherry and Vesterman 1981] Cherry, L.L., and W. Vesterman. 1981. Writing Tools—The STYLE and DICTION Programs. Computer Science Technical Report No. 91, Bell Laboratories, Murray Hill, NJ.
- [Goldberg 2010] Markham, Gervase, and Eli Goldberg. 2010. Bug Writing Guidelines. https://developer.mozilla.org/en/Bug_writing_guidelines.

- [Guo et al. 2010] Guo, Philip J., Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* 1: 496–504.
- [Hooimeijer and Weimer 2007] Hooimeijer, Peter, and Westley Weimer. 2007. Modeling bug report quality. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*: 34–43.
- [Page et al. 2008] Page, Alan, Ken Johnston, and B.J. Rollison. 2008. *How We Test Software at Microsoft*. Bellevue, WA: Microsoft Press.
- [Panjer 2007] Panjer, Lucas D. 2007. Predicting Eclipse Bug Lifetimes. *Proceedings of the Fourth International Workshop on Mining Software Repositories*: 29.
- [Schröter et al. 2010] Schröter, Adrian, Nicolas Bettenburg, and Rahul Premraj. 2010. Do Stack Traces Help Developers Fix Bugs?. *Proceedings of the 7th International Working Conference on Mining Software Repositories*: 118–121.

CHAPTER TWENTY-FIVE

Where Do Most Software Flaws Come From?

Dewayne Perry

The holy grail of software development management is “cheaper, faster, and better.” Unfortunately, a lot of poor management decisions are made in pursuit of this grail. While “cheaper and faster” are often very important, “better” clearly is the most important in a wide variety of software systems where reliability or safety is of paramount importance.

There are a variety of different ways in which a product can be made better, ranging from more clearly understanding customer needs to minimizing faults in the software system. It is the latter that is the focus of this chapter. Only by understanding the mistakes we make can we determine what remedies need to be applied to improve either the products or the processes. Monitoring faults is a relatively simple matter, either as they are found or in project retrospectives (often referred to as “project post-mortems”).

A fundamental aspect in minimizing faults in software systems is the managing of complexity, the most critical of essential characteristics of software systems [Brooks 1995]. One of the most useful techniques in managing that complexity is that of separating the interfaces of components from their implementations. It is because of this critical technique that the difference between interface and implementation faults is an important distinction that is addressed in this chapter.

Studying Software Flaws

On the one hand, it is frustrating that so few studies of software faults have been published to guide researchers in finding ways of detecting, ameliorating, or preventing these faults from happening. On the other hand, it is not at all surprising that projects are reluctant to make such sensitive data public because of internal politics or external competitiveness in software-intensive businesses.

Despite this paucity of studies and the reluctance of companies to make data available, there is a set of landmark studies about software faults that provide useful foundations for product and process improvements. Endres [Endres 1975], Schneidewind and Hoffmann [Schneidewind and Hoffmann 1979], and Glass [Glass 1981] reported on various fault analyses of software development. A weakness in their work is that they do not delineate interface faults as a specific category.

Thayer, Lipow, and Nelson [Thayer et al. 1978] and Bowen [Bowen 1980] provide extensive categorization of faults, but with a relatively narrow view of interface faults. Basili and Perricone [Basili and Perricone 1984] offer the most comprehensive study of problems encountered in the development phase of a medium-scale system, reporting data on the fault, the number of components affected, the type of the fault, and the effort required to correct the fault. Interface faults were the largest class of faults (39% of the faults).

We note, however, that none of these studies address the kinds of problems that arise in very large-scale software developments, nor do they address the evolutionary phase of developments. Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987] were the first to address fault studies in the evolution of a large real-time system. An extremely important factor in this study is the fact that interface faults were by far the overwhelming and dominant faults (68% of the faults). An important question that was left unanswered was whether these interface faults were the easy or the hard ones to find and fix.

The distinction between an evolutionary software system release and an initial development release is a critical one. In the latter case, the design and implementation choices are much less constrained than in evolutionary development. In the former, you have to make changes to an existing system, and so the choices are far more constrained and there are many more difficulties in understanding the implications of changes. As the evolutionary development part of a system's life cycle is far greater than its initial development, so too are studies of the faults in that evolutionary part much more important.

In this study, we take a detailed look at one specific release of one ultra-reliable, ultra-large-scale, real-time system rather than a more superficial look at several more moderately sized systems in several domains. The advantage of this approach is that we gain a deeper understanding of the system and its problems. The disadvantage is that we are less able to generalize our results compared to the latter course. This type of trade-off is often encountered in empirical studies.

As we will see, however, this deeper look provides us with a number of practical and useful insights. For example, it is commonly accepted wisdom that “once a bug is found, it is easy to fix.” Unfortunately, our data contradicts this “common wisdom.” Or, in the case of the unanswered question about interface faults, our data supports our original but unsubstantiated intuition that interface faults are harder to fix than implementation faults.

Context of the Study

The system discussed in this chapter is a very large-scale (that is, a million lines or more), distributed, real-time system written in the C programming language (with additional domain-specific languages as needed) in a Unix-based, multiple machine, multiple location environment.

The organizational structure is typical with respect to projects of this size and for the number of people in each organization. Not surprisingly, different organizations are responsible for various parts of the system development: requirements specification; architecture, design, coding and capability testing; system and system stability testing; and alpha testing.

The process of development is also typical with respect to projects of this size. Systems engineers prepare informal and structured documents defining the requirements for the changes to be made to the system. Designers prepare informal design documents that are subjected to formal reviews by 3 to 15 peers, depending on the size of the unit under consideration. The design is then broken into design units for low-level design and coding. The products of this phase are subjected both to formal code reviews by three to five reviewers and to low-level unit testing. As components are available, integration and system testing is performed until the system is completely integrated.

The release considered here is a “non-initial” release—one that can be viewed as an arbitrary point in the evolution of this class of systems. Because of the size of the system, the system evolution process consists of multiple, concurrent releases—that is, while the release dates are sequential, a number of releases proceed concurrently in differing phases. This concurrency accentuates the inter-release dependencies and their associated problems. The magnitude of the changes (approximately 15–20% new code for each release) and the general make up of the changes (bug fixes, improvements, new functionality, etc.) are generally uniform across releases. It is because of these two facts that we consider this study to provide a representative sample in the life of the project.

Faults discovered during testing phases are reported and monitored by a modification request (MR) tracking system (such as, for example, CMS [Rowland et al. 1983]). Access to source files for modification is possible only through the tracking system. Thus all change activity (whether fixing faults, adding new functionality, or improving existing functionality—that is, whether they are corrective, adaptive, or perfective changes) is automatically tracked by the system. This activity includes not only repairs, but enhancements and new functionality

as well. It should be kept in mind, however, that this fault tracking activity occurs only during the testing and released phases of the project, not during the architecture, design, and coding phases. Problems encountered during these earlier phases are resolved informally without being tracked by the MR system.

The goal of this study was to gain insight into the current process of system evolution by concentrating on one release of a particular system. The approach we used is that of surveying, by means of a prepared questionnaire, those developers who “owned” the fault MR at the time it was closed, surveying first the complete set of faults and then concentrating on the largest set of faults in more depth. This survey was the first of its type, although there have been some smaller studies using random selections. The CMS MR database was used to determine the initial set of fault MRs to survey and the developers who were responsible for closing those fault MRs. The survey identifying the fault MR was then sent to the identified developer to complete.

For a variety of reasons (schedule pressure among them), there were significant constraints placed on the study by project management: first, the study had to be completely non-intrusive; second, it had to be strictly voluntary; and third, it had to be completely anonymous. We will see in the later discussion about validity issues that these mandates were the source of some study weaknesses.

It is with this background that we present our surveys, analyses, and results.

Phase 1: Overall Survey

There were three specific purposes in the original overall survey:

- To determine, generally, what kinds of problems were found (which we report here), as well as, specifically, what kinds of application-specific problems arose during the preparation of this release (which we do not report, because of their lack of generality)
- To determine how the problem was found (that is, in which testing phase)
- To determine when the problem was found

One of the problems encountered in any empirical survey study is ensuring that the survey is in the “language” of those being surveyed. By this “language” we mean both the company- or project-specific jargon that is used but also the process that is being used. You want the developer surveyed to clearly understand what is being asked in terms of his own context. Failing to understand this results in questions about the validity of the survey results. To this end, we used developers to help us design the survey, and we used the project jargon and process to provide a familiar context.

Summary of Questionnaire

The first phase of the survey questionnaire had two main components: the determination of the category of the fault reported in the MR and the testing phase in which the fault was found. In determining the fault, two aspects were of importance: first, the development phase in which the fault was introduced, and second, the particular type of the fault. Since the particular type of fault reported at this stage of the survey tended to be application or methodology specific, we have emphasized the phase-origin nature of the fault categorization. The general fault categories are as follows:

Previous

Residual problems left over from previous releases

Requirements

Problems originating during the requirements specification phase of development

Design

Problems originating during the architectural and design phases of development

Coding

Problems originating during the coding phases of development

Testing environment

Problems originating in the construction or provision of the testing environment (for example, faults in the system configuration, static data, etc.)

Testing

Problems in testing (for example, pilot faults, etc.)

Duplicates

Problems that have already been reported

No problems

Problems due to misunderstandings about interfaces, functionality, etc., on the part of the user

Other

Various problems that do not fit neatly in the preceding categories, such as hardware problems, etc.

The other main component of the survey concerned the phase of testing that uncovered the fault. The following are the different testing phases:

Capability Test (CT)

Testing isolated portions of the system to ensure proper capabilities of that portion

System Test (ST)

Testing the entire system to ensure proper execution of the system as a whole in the laboratory environment

System Stability Test (SS)

Testing with simulated load conditions in the laboratory environment for extended periods of time

Alpha Test (AT)

Live use of the release in a friendly user environment

Released (RE)

Live use. However, in this study, this data refers not to this release, but the previous release. Our expectation is that this provides a projection of the fault results for this release

The time interval during which the faults were found (that is, when the fault MRs were initiated and when they were closed) was retrieved from the MR tracking system database.

Ideally, the testing phases occur sequentially. In practice, however, due to the size and complexity of the system, various phases overlap. The overlap is due to several specific factors. First, various parts of the system are modified in parallel. This means that the various parts of the system are in different states at any one time. Second, the iterative nature of evolution results in recycling back through previous phases for various parts of the system. Third, various testing phases are begun as early as possible, even though it is known that that component may be incomplete. Looked at in one way, testing proceeds in a hierarchical manner: testing is begun with various pieces, then subsystems, and finally integrating those parts into the complete system. It is a judgment call as to when different parts of the system move from one phase to the next, determined primarily by the percentage of capabilities incorporated and the number of tests executed. Looked at in a slightly different way, testing proceeds by increasing the size and complexity of the system, while at the same time increasing its load and stress.

Summary of the Data

[Table 25-1](#) summarizes the fault MRs by fault category. The fault MRs representing the earlier part of the development or evolution process (that is, those representing requirements, design, and coding) are the most significant, accounting for approximately 33.7% of the fault MRs. Given that the distinction between a design fault and a coding fault required a “judgment call” on the part of the respondent, we decided to merge the results of those two categories into one: design/coding faults account for 28.8% of the MRs. However, in the process structure used in the project, the distinction between requirements and design/coding is much clearer. Requirements specifications are produced by systems engineers, whereas the design and coding are done by developers.

TABLE 25-1. Summary of faults

MR category	Proportion
Previous	4.0%
Requirements	4.9%
Design	10.6%
Coding	18.2%
Testing environment	19.1%
Testing	5.7%
Duplicates	13.9%
No problems	15.9%
Other	7.8%

The next most significant subset of MRs were those that concern testing (the testing environment and testing categories)—24.8% of the MRs. On the one hand, it is not surprising that a significant number of problems are encountered in testing a large and complex real-time system where conditions have to be simulated to represent the “real world” in a laboratory environment. First, the testing environment itself is a large and complex system that must be tested. Second, as the real-time system evolves, so must the laboratory test environment evolve. On the other hand, this general problem is clearly one that needs to be addressed by further study.

“Duplicate” and “no problem” MRs account for another significant subset of the data—29.8%. Historically, these have been considered to be part of the overhead. Certainly the “duplicate” MRs are in large part due to the inherent concurrency of activities in a large-scale project and, as such, are difficult to eliminate. The “no problem” MRs, however, are in large part due to the lack of understanding that comes from informal and out-of-date documentation. Obviously, measures taken to reduce these kinds of problems will have beneficial effects on other categories as well. In either case, reduction of administrative overhead will improve the cost effectiveness of the project.

“Previous” MRs indicate the level of difficulty in finding some of the faults in a large, real-time system. These problems may have been impossible to find in the previous releases and have only now been exposed because of changes in the use of the system.

[Figure 25-1](#) charts the requirements, design, and coding MRs by testing phase. We have focused on this early part of the software development process because that is where the most MRs occurred and, accordingly, where closer attention should yield the most results.

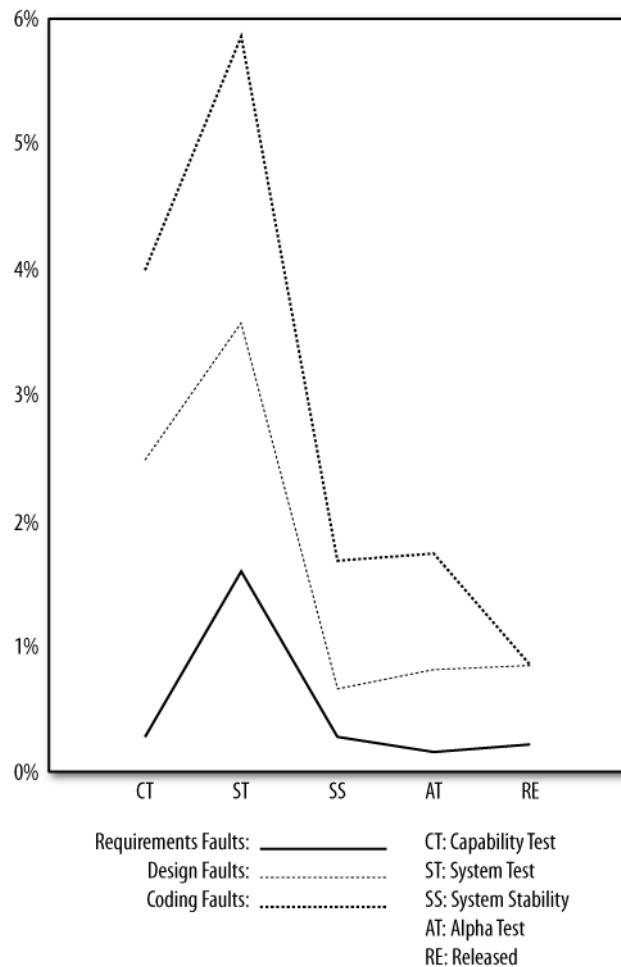


FIGURE 25-1. Fault categories found by phase

Please note that the percentages used in Figures 25-1 and 25-2 are the percentages of those faults relative to all the faults, not the percentages relative to just the charted faults. The phases in Figure 25-1 appear as sequential when in actual fact (as is almost always the case in software systems' development) there is a lot of parallelism, with phases overlapping significantly. With hundreds of software engineers developing hundreds of features concurrently, the actual project life cycle is nothing like the sequential waterfall model, even though software development proceeds through a set of, often iterative, phases. That is the reason for Figure 25-2, which shows the same data relative to a fixed timeline.

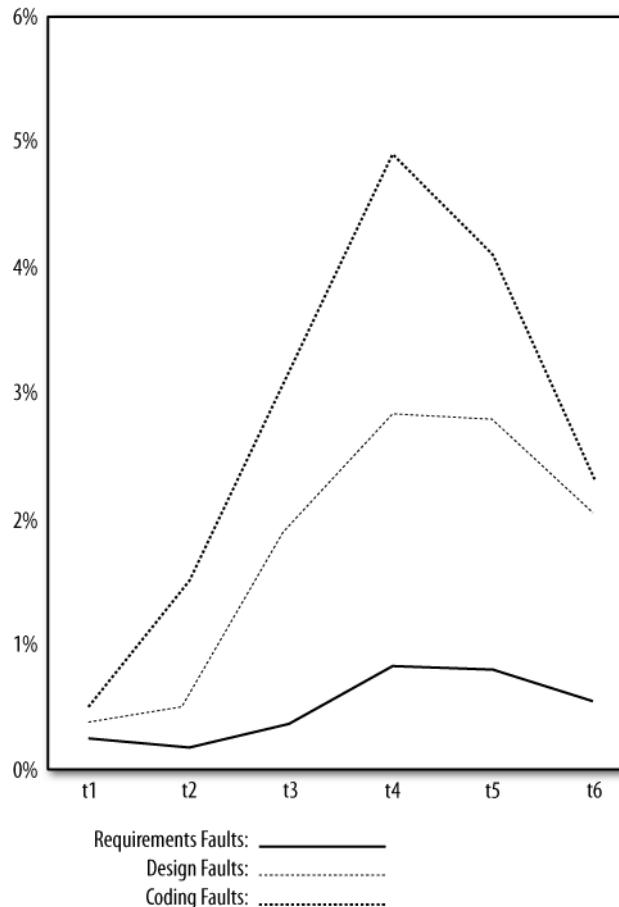


FIGURE 25-2. Fault categories found over time

There are two important observations. First, system test (ST) was the source of most of the MRs in each category; capability testing (CT) was the next largest source. This is not particularly surprising, since that is when we were looking the hardest for, and expecting the most, faults. System test is where the various components are first integrated and is the most likely place to encounter the mismatches and misassumptions among components. So while all testing is looking for faults, system test is where we look the hardest. Capability testing is akin to unit testing in an appropriate context—internal and initial interaction faults are looked for. The capability test context limits the kinds of faults that can be exposed.

Second, all testing phases found MRs of each fault category. It is also not surprising that coding faults are found over the entire set of testing phases. One obvious reason for this phenomena is that changes are continually made to correct the faults that are found in the various earlier

testing phases. Moreover, while it is disturbing to note that both design and requirements faults continue to be found throughout the entire testing process, we feel that this is due to the lack of precision and completeness in requirements and design documentation and is a general problem in the current state-of-practice rather than a project-specific problem.

The time values in [Figure 25-2](#) are fixed intervals. From the shape of the data, it is clear that System Testing overlaps interval t4. It is unfortunate that we have only the calendar data (that is, the time boundaries on the MRs), as a correlation with effort data [[Musa et al. 1987](#)] (that is, the actual amount of time spent in that time period) would be extremely valuable.

For the requirements, design, and coding fault categories over time, [Figure 25-2](#) shows that all fault types peaked at time t4 and held through time t5, except for the coding faults, which decreased.

Summary of the Phase 1 Study

The following general observations may be drawn from this general survey of the problems encountered in evolving a large-scale, real-time system:

- Implementation, testing, and administrative overhead faults occurred in roughly equal proportions
- Requirements problems, while not overwhelmingly numerous, are still significant (especially since the majority were found late in the testing process)
- All types of faults continued to be found in all the testing phases
- The most faults were found when the level of testing effort was the highest (that is, at system test)
- The majority of faults were found late in the testing cycle

These observations are limited by the fact that the tracking of fault MRs is primarily a testing activity. It would be extremely useful to observe the kinds and frequencies of faults that exist in the earlier phases of the project. Moreover, it would be beneficial to incorporate ways of detecting requirements and design faults into the existing development process.

Phase 2: Design/Code Fault Survey

As a result of the general survey, we decided to survey the design and coding MRs in depth. The following were the goals we wanted to achieve in this part of the study:

- Determine the kinds of faults that occurred in design and coding
- Determine the difficulty both in finding or reproducing these faults and in fixing them
- Determine the underlying causes of the faults
- Determine how the faults might have been prevented

- Compare the difficulty in finding and fixing interface and implementation faults

There were two reasons for choosing this part of the general set of MRs. First, it seemed to be exceedingly difficult to separate the two kinds of faults. Second, catching these kinds of faults earlier in the process would provide a significant reduction in overall fault cost; that is, the cost of finding faults before system integration is significantly less than finding them in the laboratory testing environment. Our internal cost data is consistent with Boehm's [Boehm 1981] (see also Chapter 10 by Barry Boehm). Thus, gaining insight into these problems will yield significant (and cost beneficial) results.

In the two subsections that follow, we summarize the survey questionnaire, present the results of our statistical analysis, and summarize our findings with regard to interface and implementation faults.

The Questionnaire

The respondents were asked to indicate the difficulty of finding and fixing the problem, determine the actual and underlying causes, indicate the best means of either preventing or avoiding the problem, and give their level of confidence in their responses. It should be kept in mind that the people surveyed were those who owned the MR at the time it was closed (i.e., completed).

- For each MR, rank it according to how difficult it was to reproduce the failure and locate the fault.
 1. *Easy*—could produce at will.
 2. *Moderate*—happened some of the time (intermittent).
 3. *Difficult*—needed theories to figure out how to reproduce the error.
 4. *Very Difficult*—exceedingly hard to reproduce.
- For each MR, how much time was needed to design and code the fix, and document and test it. (Note that what would be an easy fix in a single-programmer system takes considerably more time in a large, multiperson project with a complex laboratory test environment.)
 1. *Easy*—less than one day
 2. *Moderate*—1 to 5 days
 3. *Difficult*—6 to 30 days
 4. *Very difficult*—greater than 30 days

- For each MR, consider the following 22 possible types and select the one that most closely applies to the immediate cause (that is, the fault type).
 1. *Language pitfalls*—for example, pointer problems, or the use of “=” instead of “==”.
 2. *Protocol*—violated rules about interprocess communication.
 3. *Low-level logic*—for example, loop termination problems, pointer initialization, etc.
 4. *CMS complexity*—for example, due to software change management system complexity.
 5. *Internal functionality*—either inadequate functionality or changes and/or additions were needed to existing functionality within the module or subsystem.
 6. *External functionality*—either inadequate functionality or changes and/or additions were needed to existing functionality outside the module or subsystem.
 7. *Primitives misused*—the design or code depended on primitives that were not *used* correctly.
 8. *Primitives unsupported*—the design or code depended on primitives that were not adequately developed (that is, the primitives do not work correctly).
 9. *Change coordination*—either did not know about previous changes or depended on concurrent changes.
 10. *Interface complexity*—interfaces were badly structured or incomprehensible.
 11. *Design/code complexity*—the implementation was badly structured or incomprehensible.
 12. *Error handling*—incorrect handling of, or recovery from, exceptions.
 13. *Race conditions*—incorrect coordination in the sharing of data.
 14. *Performance*—for example, real-time constraints, resource access, or response-time constraints.
 15. *Resource allocation*—incorrect resource allocation and deallocation.
 16. *Dynamic data design*—incorrect design of dynamic data resources or structures.
 17. *Dynamic data use*—incorrect *use* of dynamic data structures (for example, initialization, maintaining constraints, etc.).
 18. *Static data design*—incorrect design of static data structures (for example, their location, partitioning, redundancy, etc.).
 19. *Unknown interactions*—unknowingly involved other functionality or parts of the system.
 20. *Unexpected dependencies*—unexpected interactions or dependencies on other parts of the system.
 21. *Concurrent work*—unexpected dependencies on concurrent work in other releases.
 22. *Other*—describe the fault.

- Because the fault may be only a symptom, provide what you regard to be the underlying root cause for each problem.
 1. *None given*—no underlying causes given.
 2. *Incomplete/omitted requirements*—the source of the fault stemmed from either incomplete or unstated requirements.
 3. *Ambiguous requirements*—the requirements were (informally) stated, but they were open to more than one interpretation. The interpretation selected was evidently incorrect.
 4. *Incomplete/omitted design*—the source of the fault stemmed from either incomplete or unstated design specifications.
 5. *Ambiguous design*—the design was (informally) given, but was open to more than one interpretation. The interpretation selected was evidently incorrect.
 6. *Earlier incorrect fix*—the fault was induced by an earlier, incorrect fix (that is, the fault was not the result of new development).
 7. *Lack of knowledge*—there was something that I needed to know, but did not know that I needed to know it.
 8. *Incorrect modification*—I suspected that the solution was incorrect, but could not determine how to correctly solve the problem.
 9. *Submitted under duress*—the solution was submitted under duress, knowing that it was incorrect (generally due to schedule pressure, etc.).
 10. *Other*—describe the underlying cause.
- For this fault, consider possible ways to prevent or avoid it, and select the most useful or appropriate choice for preventing, avoiding, or detecting the fault.
 1. *Formal requirements*—use precise, unambiguous requirements (or design) in a formal notation (which may be either graphical or textual).
 2. *Requirements/design templates*—provide more specific requirements (or design) document templates.
 3. *Formal interface specifications*—use a formal notation for describing the module interfaces.
 4. *Training*—provide discussions, training seminars, and formal courses.
 5. *Application walk-throughs*—determine, informally, the interactions among the various application-specific processes and data objects.
 6. *Expert person/documentation*—provide an “expert” person or clear documentation when needed.
 7. *Design/code currency*—keep design documents up to date with code changes.
 8. *Guideline enforcement*—enforce code inspections guidelines and the use of static analysis tools such as lint.

9. *Better test planning*—provide better test planning and/or execution (for example, automatic regression testing).
10. *Others*—describe the means of prevention.

Confidence levels requested of the respondents were: *very high, high, moderate, low, and very low*. We discarded the small number of responses that had a confidence level of either low or very low.

Statistical Analysis

Out of all the questionnaires, 68% were returned. Of those, we dropped the responses that were either low or very low in confidence (6%). The remainder of the questionnaires were then subjected to Chi-Square analysis [Siegel et al. 1988] to test for independence (and for interdependence) of various paired sets of data. In Chi-Square analysis, the lower the total chi-square value, the more independent the two sets of data; the higher the value, the more interdependent the two sets of data. The p-value indicates the significance of the analysis: the lower the number, the less likely the relationships are due to chance. In [Table 25-2](#), the prevention data and the find data are the most independent (the total chi-square is the lowest), and that lack of relationship is significant (the p-value is less than the standard .05 and indicates that the odds are less than 1 in 20 that the relationship happened by chance). The fault-cause, fault-prevention, and cause-prevention pairs are the most interdependent, as their total chi-square values are the largest three of the entire set and the significance of these relationships is very high (the odds are less than 1 in 10,000 of being by chance).

The fact that the relationships between the faults and their underlying causes, faults and means of prevention, and means of prevention and the underlying causes are the most significantly interdependent is a good thing: 1) faults should be strongly related to their underlying causes, and 2) both faults and their underlying causes should be strongly related to their means of prevention. This indicates that the respondents were consistent in their responses and the data aligns with what one would logically expect.

TABLE 25-2. Chi-Square analysis summary

Variables	Degrees of freedom	Total Chi-Square	p
Find, Fix	6	51.489	.0001
Fault, Find	63	174.269	.0001
Fault, Fix	63	204.252	.0001
Cause, Find	27	94.493	.0001
Cause, Fix	27	55.232	.0011
Fault, Cause	189	403.136	.0001
Prevention, Find	27	41.021	.041

Variables	Degrees of freedom	Total Chi-Square	p
Prevention, Fix	27	97.886	.0001
Fault, Prevention	189	492.826	.0001
Cause, Prevention	81	641.417	.0001

Finding and fixing faults

Table 25-3 provides a cross-tabulation of the difficulty in finding and fixing the design and coding faults. Of these faults, 78% took five days or less to fix. In general, the easier-to-find faults were easier to fix; the more difficult-to-find faults were more difficult to fix.

TABLE 25-3. Find versus fix comparison

Find/fix		<1 day	1-5 days	6-30 days	>30 days
		30.1%	48.8%	18.0%	3.6%
easy	67.5%	23.7%	32.1%	10.0%	1.7%
moderate	23.4%	4.2%	12.5%	5.6%	1.1%
difficult	7.7%	1.7%	3.4%	2.1%	.5%
very difficult	1.4%	.5%	.3%	.3%	.3%

One of the interesting things about Chi-Square analysis is that it is based on the difference between expected and observed values of the paired data. The expected value in this case is the product of the observed find value and the observed fix value. If the two sets of data are independent of each other, the expected percentages will match or be very close to the observed percentages; otherwise, the two sets of data are not independent.

The first row of data is the observed percentages of how long it took to fix the MR; the first column is the observed percentages of how hard it was to find/duplicate the problem. The expected value of easy to find and fixable in a day or less is $67.1\% \times 30.1\% = 20.2\%$, whereas the actually observed value of 23.7% is 17% more than that expected value.

There were more faults that were easy to find and took less than one day to fix than were expected by the Chi-Square analysis. Interestingly, there were fewer than expected easy to find faults (expected: 12%) that took 6 to 30 days to fix (observed: 10%).

Although the coordinates of the effort to find and fix the faults are non-comparable, we note that the following relationship is suggestive. Collapsing the previous table yields an interesting insight in Table 25-4 that seems counter to the common wisdom that says “once you have found the problem, it is easy to fix it.” There is a significant number of “easy/moderate to find” faults that require a relatively long time to fix.

TABLE 25-4. Summary of find/fix

Find/fix effort	≤ 5 days	≥ 6 days
easy/moderate	72.5%	18.4%
difficult/very difficult	5.9%	3.2%

Faults

[Table 25-5](#) shows the fault types of the MRs as ordered by their frequency in the survey independent of any other factors. For the sake of brevity in the subsequent tables, we use the fault type number to represent the fault types.

The first five fault types account for 60% of the faults. That “internal functionality” is the leading fault by such a large margin is somewhat surprising; that “interface complexity” is such a significant problem is not surprising at all. However, that the first five fault types are leading faults is consistent with the nature of the evolution of the system. Adding significant amounts of new functionality to a system easily accounts for problems with “internal functionality,” “low-level logic,” and “external functionality.”

The fact that the system is a very large, complicated real-time system easily accounts for the fact that there are problems with “interface complexity,” “unexpected dependencies” and design/code complexity,” “change coordination,” and “concurrent work.”

C has well-known “language pitfalls” that account for the rank of that fault in the middle of the set. Similarly, “race conditions” are a reasonably significant problem because of the lack of suitable language facilities in C.

That “performance” faults are relatively insignificant is probably due to the fact that this is not an early release of the system, and performance was always a significant concern of code inspections.

Fault Frequency Adjusted by Effort

There are two interesting relationships to consider in the ordering of the various faults: the effect that the difficulty in *finding* the faults has on the ordering and the effect that the difficulty of *fixing* the faults has on the ordering. The purpose of weighting is to provide an adjustment to the observed frequency by how easy or hard the faults are to find or to fix. From the standpoint of “getting the most bang for the buck,” the frequency of a fault is a good *prima facie* indicator of the importance of one fault relative to another. [Table 25-5](#) shows the fault types ordered by frequency.

TABLE 25-5. Fault types ordered by frequency

Fault type	Observed %	Fault type description
5	25.0%	internal functionality
10	11.4%	interface complexity
20	8.0%	unexpected dependencies
3	7.9%	low-level logic
11	7.7%	design/code complexity
22	5.8%	other
9	4.9%	change coordination
21	4.4%	concurrent work
13	4.3%	race conditions
6	3.6%	external functionality
1	3.5%	language pitfalls
12	3.3%	error handling
7	2.4%	primitives misused
17	2.1%	dynamic data use
15	1.5%	resource allocation
18	1.0%	static data design
14	.9%	performance
19	.7%	unknown interactions
8	.6%	primitives unsupported
2	.4%	protocol
4	.3%	CMS complexity
16	.3%	dynamic data design

Table 25-6 is an attempt to capture the weighted difficulty of finding the various faults. The weighting is done by multiplying the proportion of observed values for each fault with multiplicative weights of 1, 2, 3, and 4 for each find category, respectively, and summing the results.

Obviously it would have been better to have had some duration assigned to the effort to find faults and then correlated the weighting with those durations, as we do subsequently in weighting by effort to fix faults. The weights used are intended to be suggestive, not definitive.

We experimented with several different weightings, and the results were pretty much the same. Thus we used the simplest approach.

Better yet would have been effort data associated with each MR that could be used to get a more realistic picture of actual difficulty. But this type of data is seldom available, and an approximation is needed instead.

For example, if a fault was easy to find in 66% of the cases, moderate in 23%, difficult in 11%, and very difficult in 0%, the weight is $145 = (66 * 1) + (23 * 2) + (11 * 3) + (0 * 4)$. [Table 25-6](#) shows the fault types weighted by difficulty to find, from easiest to most difficult.

TABLE 25-6. Determining the find weighting

Fault type	Find proportion e/m/d/vd	Weight	Fault type description
4	100/0/0/0	100	CMS complexity
18	100/0/0/0	100	static data design
7	88/8/4/0	120	primitives misused
2	75/25/0/0	125	protocol
20	78/16/5/1	129	unexpected dependencies
21	70/23/2/4	130	concurrent work
3	73/22/5/0	132	low-level logic
22	82/12/2/5	132	other
5	74/19/6/1	134	internal functionality
6	67/31/3/0	139	external functionality
1	68/26/2/2	141	language pitfalls
10	66/23/11/0	145	interface complexity
9	65/20/12/2	149	change coordination
8	67/17/17/0	152	primitives unsupported
19	88/8/4/0	157	unknown interactions
16	67/0/33/0	157	dynamic data design
17	52/38/10/0	158	dynamic data use
15	47/47/7/0	162	resource allocation
12	55/30/12/3	163	error handling
11	55/29/16/1	165	code complexity
14	56/11/11/22	199	performance
13	12/67/21/0	209	race conditions

Typically, performance faults and race conditions are very difficult to isolate and reproduce. We would expect that “code complexity” and “error handling” faults also would be difficult to find and reproduce. Not surprisingly, “language pitfalls” and “interface complexity” are reasonably hard to detect.

In the Chi-Square analysis, “internal functionality,” “unexpected dependencies,” and “other” tended to be easier to find than expected. “Code complexity” and “performance” tended to be harder to find than expected. There tended to be more significant deviations where the population was larger.

If we weight the proportions by multiplying the number of occurrences of each fault by its weight from [Table 25-5](#) and dividing by the total weighted number of occurrences, we get only a slight change in the ordering of the faults, with “internal functionality,” “code complexity,” and “race conditions” (faults 5, 11, and 13) changing slightly more than the rest of the faults.

[Table 25-7](#) represents the results of weighting the difficulty of fixing the various faults by factoring in the actual time needed to fix the faults. The multiplicative scheme uses the values 1, 3, 15, and 30 for the four average times in fixing a fault. The calculations are performed as in the example of weighting the difficulty of finding the faults.

The weighting according to the difficulty in fixing the fault causes some interesting shifts in the ordering. “Language pitfalls,” “low-level logic,” and “internal functionality” (faults 1, 3, and 5) drop significantly in their relative importance. This coincides with one’s intuition about these kinds of faults. “Design/code complexity,” “resource allocation,” and “unexpected dependencies” (faults 11, 15, and 20) rise significantly in their relative importance; “interface complexity,” “race conditions,” and “performance” (faults 10, 13, 14) also rise, but not significantly so.

TABLE 25-7. Determining the fix weighting

Fault type	Proportion e/m/d/vd	Weight	Fault type description
16	67/33/0/0	166	dynamic data design
4	67/33/0/0	166	CMS complexity
8	50/50/0/0	200	primitives unsupported
18	50/50/0/0	200	static data design
1	63/31/6/0	244	language pitfalls
3	59/37/3/1	245	low-level logic
2	25/75/0/0	250	protocol
17	38/48/14/0	392	dynamic data use
9	37/49/14/0	394	change coordination
5	27/59/14/0	414	internal functionality

Fault type	Proportion e/m/d/vd	Weight	Fault type description
22	40/43/12/5	496	other
7	46/37/8/8	497	primitives misused
10	17/57/26/1	608	interface complexity
21	25/43/30/2	661	concurrent work
6	22/50/22/6	682	external functionality
13	16/56/21/7	709	race conditions
12	21/52/18/9	717	error handling
19	29/43/14/14	785	unknown interactions
20	24/39/33/5	786	unexpected dependencies
11	22/39/27/12	904	design/code complexity
14	11/22/44/22	1397	performance
15	0/47/27/27	1356	resource allocation

Table 25-8 shows the top fix-weighted faults. According to our weighting schemes, these four faults account for 55.2% of the effort expended to fix all the faults and 51% of the effort to find them, but represent 52.1% of the faults by frequency count. Collectively, they are somewhat harder to fix than rest of the faults and slightly easier to find. We again note that although the two scales are not strictly comparable, the comparison is an interesting one nonetheless.

TABLE 25-8. Faults weighted by fix difficulty

Fault type	Weighted %	Brief description
5	18.7%	internal functionality
10	12.6%	interface complexity
11	12.6%	code complexity
20	11.3%	unexpected dependencies

In the Chi-Square analysis, “language pitfalls” and “low-level logic” took fewer days to fix than expected. “Interface complexity” and “internal functionality” took 1 to 6 days to fix more often than expected, and “design/code complexity” and “unexpected dependencies” took longer to fix (that is, 6 to over 30 days) than expected. These deviations reinforce our weighted assessment of the effort to fix the faults.

Underlying causes

In [Table 25-9](#), we show the underlying causes of the MRs as ordered by their frequency in the survey, independent of any other factors.

TABLE 25-9. Underlying causes of faults

Underlying causes	Observed %	Brief description
4	25.2%	incomplete/omitted design
1	20.5%	none given
7	17.8%	lack of knowledge
5	9.8%	ambiguous design
6	7.3%	earlier incorrect fix
9	6.8%	submitted under duress
2	5.4%	incomplete/omitted requirements
10	4.1%	other
3	2.0%	ambiguous requirements
8	1.1%	incorrect modification

The high proportion of “none given” as an underlying cause requires some explanation. One of the reasons for this is that faults such as “language pitfalls,” “low-level logic,” “race conditions,” and “change coordination” tend to be both the fault and the underlying cause (7.8%—or 33% of the faults in the “none given” underlying cause category in [Table 25-12](#) below). In addition, one could easily imagine that some of the faults, such as “interface complexity” and “design/code complexity,” could also be considered both the fault and the underlying cause (3.4%—or 16% of the faults in the “none given” underlying cause category in [Table 25-12](#)). On the other hand, we were surprised that no cause was given for a substantial part of the “internal functionality” faults (3.3%—or 16% of the faults in the “none given” category in [Table 25-12](#)). One would expect there to be some underlying cause for that particular fault.

[Table 25-10](#) shows the relative difficulty in finding the faults associated with the underlying causes. The resulting ordering is particularly nonintuitive: the MRs with no underlying cause are the second most difficult to find; those submitted under duress are the most difficult to find.

TABLE 25-10. Weighting of the underlying causes by find effort

Underlying causes	Proportion	Weight	Brief description
8	91/9/0/0	109	incorrect modification
7	74/18/7/1	135	lack of knowledge
3	60/40/0/0	140	ambiguous requirements
5	66/27/7/0	141	ambiguous design
2	70/17/13/0	143	incomplete/omitted requirements
4	68/25/7/1	143	incomplete/omitted design
6	73/12/10/5	147	earlier incorrect fix
10	76/12/0/12	148	other
1	63/25/11/1	150	none given
9	50/46/4/0	158	submitted under duress

In the Chi-Square analysis of finding underlying causes, faults caused by “lack of knowledge” tended to be easier to find than expected, whereas faults caused by “submitted under duress” tended to be moderately hard to find more often than expected. This latter finding is interesting, as we know very little about faults “submitted under duress.”

In [Table 25-11](#), we weight the underlying causes by the effort to fix the faults represented by the underlying causes. This yields a few shifts in the proportion of effort: “incomplete/omitted design” increased significantly; “unclear requirements” and “incomplete/omitted requirements” increased less significantly; “none” decreased significantly; and “unclear design” and “other” decreased less significantly. However, the relative ordering of the various underlying causes is unchanged.

TABLE 25-11. Weighting of the underlying causes by fix effort

Underlying causes	Proportion	Weight	Brief description
10	37/42/12/10	340	other
1	43/43/12/2	412	none given
5	29/55/14/2	464	ambiguous design
7	30/50/17/3	525	lack of knowledge
6	34/45/17/4	544	earlier incorrect fix
9	18/57/25/0	564	submitted under duress
8	18/55/27/0	588	incorrect modification
4	23/50/22/5	653	incomplete/omitted design

Underlying causes	Proportion	Weight	Brief description
2	26/44/24/6	698	incomplete/omitted requirements
3	25/30/24/6	940	ambiguous requirements

The relative weighting of the effort to fix these kinds of underlying causes seems to coincide with one's intuition very nicely.

In the Chi-Square analysis of fixing underlying causes, faults caused by "none given" tended to take less time to fix than expected, whereas faults caused by "incomplete/omitted design" and "submitted under duress" tended to take more time to fix than expected.

In [Table 25-12](#), we present the cross-tabulation of faults and their underlying causes. Faults are represented by the rows, underlying causes by the columns. The numbers in the matrix are the percentages of the total population of faults. Thus, 1.5% of the total faults were fault 1 with the underlying cause 1. The expected number of faults for fault 1 and underlying cause 1 can be computed by multiplying the total faults for each of those categories: $20.5\% * 3.5\% = .7\%$. In this example, the actual number of faults was higher than expected.

TABLE 25-12. Cross-tabulating fault types and underlying causes

		1	2	3	4	5	6	7	8	9	10
		20.5%	5.4%	2.0%	25.2%	9.8%	7.3%	17.8%	1.1%	6.8%	4.1%
1 language pitfalls	3.5%	1.5	.0	.0	.2	.1	.2	.8	.1	.5	.1
2 protocol	.4%	.0	.0	.1	.2	.0	.0	.1	.0	.0	.0
3 low-level logic	7.9%	3.7	.3	.1	.6	.3	1.2	.7	.0	.6	.4
4 CMS complexity	.3%	.1	.0	.0	.0	.0	.1	.1	.0	.0	.0
5 internal functionality	25.0%	3.3	1.3	.6	7.7	2.8	2.0	5.2	.3	1.2	.6
6 external functionality	3.6%	.7	.3	.1	.4	.5	.6	.7	.0	.3	.0
7 primitives misused	2.4%	.4	.0	.0	.5	.0	.1	.8	.0	.0	.6
8 primitives unsupported	.6%	.0	.2	.0	.1	.0	.1	.1	.0	.1	.0
9 change coordination	4.9%	1.1	.0	.0	.8	1.0	.6	.8	.1	.3	.2

		1	2	3	4	5	6	7	8	9	10
		20.5%	5.4%	2.0%	25.2%	9.8%	7.3%	17.8%	1.1%	6.8%	4.1%
10 interface complexity	11.4%	2.1	.6	.2	4.1	1.4	1.1	1.4	.2	.0	.3
11 design/code complexity	7.7%	1.3	.0	.3	3.0	1.6	.2	1.0	.0	.0	.3
12 error handling	3.3%	.9	.3	.0	.8	.0	.1	.7	.0	.4	.1
13 race conditions	4.3%	1.4	.2	.0	1.3	.5	.1	.3	.0	.4	.1
14 performance	.9%	.2	.0	.1	.2	.0	.0	.3	.0	.0	.1
15 resource allocation	1.5%	.5	.0	.0	.3	.1	.0	.4	.1	.0	.1
16 dynamic data design	.3%	.0	.0	.0	.1	.0	.0	.1	.0	.1	.0
17 dynamic data use	2.1%	.7	.1	.0	.2	.1	.0	.6	.0	.4	.0
18 static data design	1.0%	.3	.1	.1	.2	.1	.0	.1	.0	.1	.0
19 unknown interactions	.7%	.0	.1	.1	.0	.2	.0	.2	.0	.1	.0
20 unexpected dependencies	8.0%	.5	.8	.3	2.7	.5	.1	1.4	.0	1.7	.0
21 concurrent work	4.4%	.6	.3	.0	1.2	.2	.4	.9	.2	.4	.2
22 other	5.8%	1.2	.8	.0	.6	.4	.4	1.1	.1	.2	1.0

For the sake of brevity, we consider only the most frequently occurring faults and their major underlying causes. “Incomplete/omitted design” (cause 4) is the primary underlying cause in all of these major faults. “Ambiguous design” (cause 5), “lack of knowledge” (cause 7), and “none given” (cause 1) were also significant contributors to the presence of these faults.

internal functionality (fault 5)

“Incomplete/omitted design” (cause 4) was felt to have been the cause of 31% (that is, 7.7% / 25%) of the occurrences of this fault, a percentage higher than expected; “lack of knowledge” (cause 7) was thought to have caused 21% of the occurrences of this fault, higher than expected; and “none given” was listed as the third underlying cause, representing 13% of the occurrences.

interface complexity (fault 10)

Again, “incomplete/omitted design” was seen to be the primary cause in the occurrence of this fault (36%), higher than expected; “lack of knowledge” and “ambiguous design” were seen as the second and third primary causes of this fault (13% and 12%, respectively).

unexpected dependencies (fault 20)

Not surprisingly, “incomplete/omitted design” was felt to have been the primary cause of this fault (in 34% of the cases); “submitted under duress” (cause 9) contributed to 21% of the occurrences, a percentage higher than expected; and “lack of knowledge” was the tertiary cause of this fault, representing 18% of the occurrences.

design/code complexity (fault 11)

Again, “incomplete/omitted design” was felt to have been the primary cause in 39% of the occurrences of this fault, a percentage higher than expected; “ambiguous design” was the second most frequent underlying cause of this fault, causing 21% of the faults (also a higher percentage than expected); and “none given” was listed as the third underlying cause, representing 17% of the occurrences.

Again, for the sake of brevity, we consider only the most frequently occurring underlying causes and the faults to which they were most applicable.

incomplete/omitted design (cause 4)

As we noted previously, “internal functionality,” “interface complexity,” “code/design complexity,” and “unexpected dependencies” were the major applicable faults (31%, 12%, 12%, and 11%, respectively), with the first three occurring with higher than expected frequency.

none given (cause 1)

“Low-level logic” (fault 3) was the leading fault, representing 18% of the occurrences (a percentage higher than expected); “internal functionality” (fault 5) was the second major fault, representing 16% of the occurrences (a percentage lower than expected); “interface complexity” (fault 10) was the third leading fault, representing 10% of the occurrences; and “language pitfalls” was the fourth leading fault, representing 8% of the occurrences (a percentage higher than expected).

lack of knowledge (cause 7)

“Internal functionality” was the leading fault, representing 29% of the occurrences (a percentage higher than expected); “interface complexity” was next with 8% of the

occurrences (a percentage lower than expected); “unexpected dependencies” was third with 8% of the occurrences; and “other” (fault 22) was the fourth with 6%.

ambiguous design (cause 5)

“Internal functionality” represented 29% of the occurrences; “code/design complexity” (fault 11) was second fault, representing 16% of the occurrences (a percentage higher than expected); “interface complexity” was third with 14%; and “change coordination” (fault 9) was fourth, representing 10% of the occurrences (a percentage higher than expected).

Means of prevention

Table 25-13 shows the means of prevention of the MRs, as ordered by their occurrence independent of any other factors. We note that the means selected may well reflect a particular approach of the responder in selecting one means over another (for example, see the discussion later in this section about formal versus informal means of prevention).

TABLE 25-13. Means of error prevention

Means of prevention	Observed %	Brief description
5	24.5%	application walk-throughs
6	15.7%	expert person/documentation
8	13.3%	guideline enforcement
2	10.0%	requirements/design templates
9	9.9%	better test planning
1	8.8%	formal requirements
3	7.2%	formal interface specifications
10	6.9%	other
4	2.2%	training
7	1.5%	design/code currency

It is interesting to note that the application-specific means of prevention (“application walk-throughs”) is considered the most effective. This selection of application walk-throughs as the most useful means of error prevention appears to confirm the observation of Curtis, Krasner, and Iscoe [Curtis et al. 1988] that a thin spread of application knowledge is the most significant problem in building large systems.

Further, it is worth noting that informal means of prevention rank higher than formal ones. On the one hand, this may reflect the general bias in the United States against formal methods. On the other hand, the informal means are a nontechnical solution to providing the

information that may be supplied by formal representations (and which provide a more technical solution with perhaps higher attendant adoption costs).

The level of effort to find the faults for which these are the means of prevention does not change the order found in [Table 25-13](#), with the exception of “requirements/design templates,” which seems to apply to the easier-to-find faults, and “guideline enforcement,” which seems to apply more to the harder-to-find faults.

In the Chi-Square analysis, the relationship between finding faults and preventing them is the most independent of the relationships, reported here with $p=.041$. “Application walk-throughs” applied to faults that were marginally easier to find than expected, whereas “guideline enforcement” applied to faults that were less easy to find than expected.

In [Table 25-14](#), the means of prevention is weighted by the effort to fix the associated faults.

TABLE 25-14. Means of prevention weighted by fix effort

Prevention	Proportion	Weight	Brief description
8	38/52/7/3	389	guideline enforcement
9	35/52/12/1	401	better test planning
7	40/40/20/0	460	design/code currency
5	33/50/17/1	468	application walk-throughs
10	49/36/6/9	517	other
2	10/52/30/1	654	requirements/design templates
3	26/43/26/4	675	formal interface specifications
6	22/48/24/6	706	expert person/documentation
1	20/50/22/8	740	formal requirements
4	23/36/23/18	1016	training

It is interesting to note that the faults considered to be prevented by training are the hardest to fix. The formal methods also apply to classes of faults that take a long time to fix.

Weighting the means of prevention by effort to fix their corresponding faults yields a few shifts in proportion: “application walk-throughs,” “better test planning,” and “guideline enforcement” decreased in proportion; “expert person/documentation” and “formal requirements” increased in proportion; and “formal interface specifications” and “other” less so. As a result, the ordering changes slightly to 5, 6, 2, 1, 8, 10, 3, 9, 4, 7: “expert person/documentation” and “formal requirements” (numbers 6 and 1) are weighted significantly higher; “requirements/design templates,” “formal interface specifications,” “training,” and “other” (numbers 2, 3, 4, and 10) are less significantly higher; and “guideline enforcement” and “better test planning” (numbers 8 and 9) are significantly lower.

In the Chi-Square analysis, faults prevented by “application walk-throughs,” “guideline enforcement,” and “other” tended to take fewer days to fix than expected, whereas faults prevented by “formal requirements,” “requirements/design templates,” and “expert person/documentation” took longer to fix than expected.

In [Table 25-15](#), we present the cross-tabulation of faults and their means of prevention. Again, the faults are represented by the rows, and the means of prevention are represented by the columns. The data is analogous to the preceding cross-tabulation of faults and underlying causes.

For the sake of brevity, we consider only the most frequently occurring faults and their major means of prevention. “Application walk-throughs” were felt to be an effective means of preventing these most significant faults. “Expert person/documentation,” “formal requirements,” and “formal interface specifications” were also significant means of preventing these faults.

internal functionality (fault 5)

“Application walk-throughs” (prevention 5) were thought to be the most effective means of prevention, applicable to 27% of the occurrences of this fault; “expert person/documentation” (prevention 6) was felt to be the second most effective means, applicable to 18% of the fault occurrences; and “requirements/design templates” were thought to be applicable to 14% of the fault occurrences, a percentage higher than expected.

TABLE 25-15. Cross-tabulating faults and means of prevention

		1	2	3	4	5	6	7	8	9	10
		8.8%	10.0%	7.2%	2.2%	24.5%	15.7%	1.5%	13.3%	9.9%	6.9%
1 language pitfalls	3.5%	.0	.1	.1	.0	1.0	.3	.1	1.3	.4	.2
2 protocol	.4%	.1	.2	.0	.0	.1	.0	.0	.0	.0	.0
3 low-level logic	7.9%	.1	.0	.1	.2	2.3	.3	.2	3.2	.8	.7
4 CMS complexity	.3%	.0	.0	.0	.0	.0	.1	.0	.1	.1	.0
5 internal functionality	25.0%	1.9	3.5	1.5	.4	6.6	4.4	.2	3.3	3.1	.1
6 external functionality	3.6%	.6	.3	.4	.0	.1	.7	.0	.5	.9	.1
7 primitives misused	2.4%	.1	.1	.2	.0	.8	.3	.0	.1	.2	.6

		1	2	3	4	5	6	7	8	9	10
		8.8%	10.0%	7.2%	2.2%	24.5%	15.7%	1.5%	13.3%	9.9%	6.9%
8 primitives unsupported	.6%	.1	.0	.0	.0	.3	.0	.0	.0	.1	.1
9 change coordination	4.9%	.4	.9	.3	.4	.8	.3	.3	.3	.7	.5
10 interface complexity	11.4%	2.1	.3	2.1	.0	3.0	1.7	.1	1.2	.7	.2
11 design/code complexity	7.7%	.8	.5	.1	.4	2.2	2.4	.2	.3	.4	.4
12 error handling	3.3%	.2	.2	.3	.1	.6	.6	.0	.4	.5	.4
13 race conditions	4.3%	.8	.0	.4	.0	1.2	.4	.2	.4	.2	.7
14 performance	.9%	.0	.0	.0	.2	.2	.3	.0	.0	.0	.2
15 resource allocation	1.5%	.1	.1	.1	.0	.3	.3	.0	.3	.3	.0
16 dynamic data design	.3%	.0	.0	.0	.0	.1	.0	.0	.1	.0	.1
17 dynamic data use	2.1%	.0	.0	.2	.0	.8	.5	.0	.5	.0	.1
18 static data design	1.0%	.1	.1	.0	.0	.2	.2	.0	.0	.3	.1
19 unknown interactions	.7%	.1	.0	.2	.0	.0	.2	.0	.0	.2	.0
20 unexpected dependencies	8.0%	.6	2.2	1.1	.1	2.3	.6	.0	.4	.6	.1
21 concurrent work	4.4%	.4	.7	.0	.2	1.2	1.1	.1	.3	.0	.4
22 other	5.8%	.3	.8	.1	.2	.4	1.0	.1	.6	.4	1.9

interface complexity (fault 10)

Again, “application walk-throughs” were considered to be the most effective, applicable to 26% of the cases; “formal requirements” (prevention 1) and “formal interface specifications” were felt to be equally effective, with each preventing 18% of the fault occurrences (in both cases, a percentage higher than expected).

unexpected dependencies (fault 20)

“Application walk-throughs” were felt to be the most effective means of preventing this fault, applicable to 29% of the occurrences; “requirements/design templates” were considered the second most effective and applicable to 28% of the fault occurrences (a percentage higher than expected); and “formal interface specifications” were considered applicable to 14% of the fault occurrences, a percentage higher than expected.

design/code complexity (fault 11)

“Expert person/documentation” was felt to be the most effective means of preventing this fault, applicable to 31% of the cases (higher than expected); “application walk-throughs” were the second most effective means, applicable to 29% of the occurrences; and “formal requirements” was third, applicable to 10% of the fault occurrences.

Again, for the sake of brevity, we consider only the most frequently occurring means of prevention and the faults to which they were most applicable. Not surprisingly, these means were most applicable to “internal functionality” and “interface complexity,” the most prevalent faults. Counterintuitively, they are also strongly recommended as applicable to “low-level logic.”

application walk-throughs (prevention 5)

“Internal functionality” (fault 5) was considered as the primary target in 27% of the uses of this means of prevention; “interface complexity” (fault 10) was felt to be the secondary target, representing 12% of the uses of this means; and “low-level logic” (fault 3) and “unexpected dependencies” (fault 20) were next with 9% each.

expert person/documentation (prevention 6)

Again, “internal functionality” is the dominant target for this means, representing 29% of the possible applications; “design/code complexity” is the second most applicable target, representing 15% of the possible applications (a percentage higher than expected); and “interface complexity” represented 11% of the uses (higher than expected).

guideline enforcement (prevention 8)

“Internal functionality” and “low-level logic” were the dominant targets for this means of prevention, representing 25% and 24%, respectively (the latter being higher than expected); “language pitfalls” (fault 1) was seen as the third most relevant fault, representing 10% of the possible applications (higher than expected); and “interface complexity” was the fourth with 9% of the possible applications of this means of prevention.

Underlying causes and means of prevention

In [Table 25-16](#), it is interesting to note that in the Chi-Square analysis there are lots of deviations (that is, there is a wider variance between the actual values and the expected values in correlating underlying causes and means of prevention). This indicates that there are strong dependencies between the underlying causes and their means of prevention. Intuitively, this type of relationship is just what we would expect.

TABLE 25-16. Cross-tabulating means of prevention and underlying causes

		1	2	3	4	5	6	7	8	9	10
		20.5%	5.4%	2.0%	25.2%	9.8%	7.3%	17.8%	1.1%	6.8%	4.1%
1 formal requirements	8.8%	.4	2.3	.9	3.5	.8	.3	.5	.1	.0	.0
2 reqs/design templates	10.0%	.4	1.7	.1	3.7	1.9	.1	.8	.0	1.3	.0
3 formal interface specs	7.2%	.8	.3	.1	2.7	.8	.3	2.0	.0	.2	.0
4 training	2.2%	.4	.0	.1	.7	.1	.3	.6	.0	.0	.0
5 application walk-thrus	24.5%	7.5	.2	.3	7.3	3.1	1.8	3.1	.0	.5	.7
6 expert person/doc	15.7%	1.5	.4	.4	3.5	1.8	1.0	5.8	.6	.3	.4
7 design/code currency	1.5%	.4	.0	.0	.6	.2	.1	.2	.0	.0	.0
8 guideline enforcement	13.3%	4.0	.1	.0	.6	.2	1.6	2.5	.0	3.7	.6
9 better test planning	9.90%	2.8	.2	.0	1.7	.8	1.6	1.9	.3	.2	.4
10 others	6.9%	2.3	.2	.1	.9	.1	.2	.4	.1	.6	2.0

We first summarize the means of prevention associated with the major underlying causes. “Application walk-throughs,” “expert person/documentation,” and “guideline enforcement” were considered important in addressing these major underlying causes.

incomplete/omitted design (cause 4)

“Application walk-throughs” (prevention 5) was thought to be applicable to 28% of the faults with this underlying cause (a percentage higher than expected); “requirements/design templates” (prevention 2) and “expert person/documentation” (prevention 6) were

next in importance with 14% each (the first being higher than expected); and “formal requirements” (prevention 1) was felt to be applicable to 12% of the faults with this underlying cause (a percentage higher than expected).

none given (cause 1)

Again, “application walk-throughs” was thought to be applicable to 37% of the faults with these underlying causes; “guideline enforcement” (prevention 8), “better test planning” (prevention 9), and “other” (prevention 10) were felt to be applicable to 19%, 14%, and 10% of the faults, respectively. In all four of these cases, the percentages were higher than expected.

lack of knowledge (cause 7)

“Expert person/documentation” was thought to be applicable to 32% of the faults with this underlying cause, a percentage higher than expected; “application walk-throughs,” “guideline enforcement,” and “formal interface specifications” were felt to be applicable to 17%, 14%, and 11% of the faults with this underlying cause, respectively, though “application walk-throughs” had a lower percentage than expected, whereas “formal interface specifications” had a higher percentage than expected.

The following summarizes the major underlying causes addressed by the most frequently considered means of prevention. “Lack of knowledge,” “none given,” “incomplete/omitted design,” and “ambiguous design” were the major underlying causes for which these means of prevention were considered important. It is somewhat non-intuitive that the “none given” underlying cause category is so prominent as an appropriate target for these primary means of prevention.

application walk-throughs (prevention 5)

“None given” (cause 1) and “incomplete/omitted design” (cause 4) were thought to be the appropriate for this means of prevention for 31% and 30% of the cases, respectively (higher than expected); “ambiguous design” (cause 5) and “lack of knowledge” (cause 7) both were felt to apply to 13% of the cases (though the first was higher than expected and the second lower).

expert person/documentation (prevention 6)

“Lack of knowledge” was considered the major target for this means of prevention, accounting for 37% of the cases (a higher than expected value); “incomplete/omitted design” and “ambiguous design” were thought to be appropriate in 23% and 11% of the cases, respectively; and “none given” was thought appropriate in 10% of the cases (lower than expected).

guideline enforcement (prevention 8)

“None given” and “incorrect modification” were felt to be the most appropriate for this means of prevention for 30% and 28% of the cases, respectively (both higher than expected); “lack of knowledge” and “incorrect earlier fix” were appropriate in 19% and 12% of the cases, respectively (the latter was higher than expected).

Interface Faults Versus Implementation Faults

The definition of an interface fault that we use here is that of Basili and Perricone [Basili and Perricone 1984] and Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987]: interface faults are “those that are associated with structures existing outside the module’s local environment but which the module used.” Using this definition, we roughly characterize “language pitfalls” (1), “low-level logic” (3), “internal functionality” (5), “design/code complexity” (11), “performance” (14), and “other” (22) as implementation faults. The remainder are considered interface faults. We say “roughly” because there are some cases where the implementation categories may contain some interface problems; remember that some of the “design/code complexity” faults were considered preventable by formal interface specifications. [Table 25-17](#) shows our interface versus implementation fault comparison.

TABLE 25-17. Interface/implementation fault comparison

	Interface	Implementation
Frequency	49%	51%
Find weighted	50%	50%
Fix weighted	56%	44%

Interface faults occur with slightly less frequency than implementation faults, but require about the same effort to find them and more effort to fix them.

[Table 25-18](#) compares interface and implementation faults with respect to their underlying causes. Underlying causes “other,” “ambiguous requirements,” “none given,” “earlier incorrect fix,” and “ambiguous design” tended to be the underlying causes more for implementation faults than for interface faults. Underlying causes “incomplete/omitted requirements,” “incorrect modification,” and “submitted under duress” tended to be the causes more for interface faults than for implementation faults.

Note that underlying causes that involved ambiguity tended to result more in implementation faults than in interface faults, whereas underlying causes involving incompleteness or omission of information tended to result more in interface faults than in implementation faults.

TABLE 25-18. Interface/implementation faults and underlying causes

		Interface	Implementation
		49%	51%
1	none given	45.2%	54.8%
2	incomplete/omitted requirements	79.6%	20.4%
3	ambiguous requirements	44.5%	55.5%
4	incomplete/omitted design	50.8%	49.2%

		Interface	Implementation
		49%	51%
5	ambiguous design	47.0%	53.0%
6	earlier incorrect fix	45.1%	54.9%
7	lack of knowledge	49.2%	50.8%
8	incorrect modification	54.5%	45.5%
9	submitted under duress	63.1%	36.9%
10	other	39.1%	60.1%

Table 25-19 compares interface and implementation faults with respect to the means of prevention. Not surprisingly, means 1 and 3 were more applicable to interface faults than to implementation faults. Means of prevention 8, 4, and 6 were considered more applicable to implementation faults than to interface faults.

TABLE 25-19. Interface/implementation faults and means of prevention

		Interface	Implementation
		49%	51%
1	formal requirements	64.8%	35.2%
2	requirements/design templates	51.5%	48.5%
3	formal interface specifications	73.6%	26.4%
4	training	36.4%	63.6%
5	application walk-throughs	48.0%	52.0%
6	expert person/documentation	44.3%	55.7%
7	design/code currency	46.7%	53.3%
8	guideline enforcement	33.1%	66.9%
9	better test planning	48.0%	52.0%
10	others	49.3%	50.7%

What Should You Believe About These Results?

Designing empirical studies is just like designing software systems: it is impossible to create a bug-free system, and we often make design mistakes and create systems with flaws and weaknesses. The main question in both cases is: do the problems negate the usefulness of the software systems or the empirical studies?

There are three main questions we need to address in order to determine how good our study is and whether you can justifiably use our results: 1) are we measuring the right things, 2) are there other things that might be the explanations for what we see (i.e., did we do it right?), and 3) what do our results apply to (i.e., what can we do with the results)?

Are We Measuring the Right Things?

We believe that we have a very strong argument to support our claim that we have addressed the critical issues in understanding software development faults and their implications. We address the fundamental issues in fault studies: the faults that occur, how hard it is to find and fix them, their underlying causes, and how might we prevent them, detect them, or ameliorate them. In addition, we addressed a question raised in response to the interface fault studies we had done earlier: which are harder to find and/or fix, interface or implementation faults? Strong support for this comes from the consistency and mutual support provided by the Chi-Square analysis, in which there are very strong relationships between the faults detected, their underlying causes, and their means of prevention. These strong relationships indicate a consistent understanding of the various parts of the survey and their interrelationships.

There are, however, several weaknesses that need to be addressed. First, the fault categories are poorly constructed. Second, the find and fix scales are not identical, with the fix scale being much better than the find scale. Third, the line between interface and implementation faults is not cleanly drawn.

The primary strength of the fault type list is that it was drawn up by the developers themselves, not the researchers. The weakness is that the list is basically unstructured and too long. There may be a tendency to pick the first thing that comes close rather than search the list exhaustively to find the best match.

In a subsequent study [Leszak et al. 2000], [Leszak et al. 2002], we corrected the fault type problem by partitioning the fault list into three categories: implementation, interface, and external faults (which also solved the third weakness mentioned above). Under each of these three fault categories were then between six and eight fault subcategories appropriate for each fault category (see page 177 of [Leszak et al. 2002]).

The scales used for finding and fixing faults was again the choice of the software developers, but had more serious consequences than poor structuring. The scale used for fixing faults is intuitively one that can be used easily, as it is easy to remember if something took less than a day, week, or month, which makes this measure much less likely to be misclassified. The scale for finding a fault, on the other hand, was qualitative rather than quantitative and much more likely to be subjective with individual variance (which we were not able to determine, because of management restrictions). Our recommendation is to use the quantitative scale for effort in terms of time for both scales. Indeed we did do that as well in subsequent studies [Leszak et al. 2000], [Leszak et al. 2002].

The separation of faults into interface and implementation faults was not a completely clean one, as some of the fault categories counted as interface may have included some implementation faults as well (and vice versa). So our distinction between the two is approximate at best. As mentioned earlier, we later solved that problem by structuring the separation of faults into implementation, interface, and external [Leszak et al. 2000], [Leszak et al. 2002].

Did We Do It Right?

In both phases, approximately 68% of questionnaires were returned—that is, we have data on about two-thirds of the MRs in both the overall survey and in the design/coding survey. Given the circumstances under which the survey was taken, this level of response exceeded our best expectations. Indeed, this factor of a large number of responses alone provides an argument for having data that can be relied upon.

As we cannot give the hard numbers as part of our report, we have tried to indicate the level of responses via the precision we used in discussing the results. Given the amount of data, we could have easily justified using two decimal places in reporting the data instead of the one decimal place we used for ease of understanding the data.

As with all surveys, there is the unanswerable question of how those who did not respond would have affected the results. Fortunately, we know of no existent factors (such as reporting only the hard or easy problems, receiving reports from only junior or senior programmers, etc.) that would have skewed the results in any way [Basili and Hutchens 1983].

We mentioned earlier that there were significant constraints placed on the study by project management: first, the study had to be completely nonintrusive; second, it had to be strictly voluntary; and third, it had to be completely anonymous. Because of these management mandates, we were unable to validate the results [Basili and Weiss 1984] and are unable to assess the accuracy of the responses. Mitigating the lack of validation are two facts: first, the questionnaire was created by the authors working with a group of developers; second, the questionnaire was reviewed by an independent group of developers. Since the purpose of the post-survey validation is understanding the level at which those surveyed understood the survey properly, we believe that our pre-survey efforts provide a useful and valid alternative because 1) we ensured that the survey was the language used by the developers themselves by their participation in its development, and 2) we pretested the survey successfully with a small group of developers, and no misunderstandings arose in the pretests.

The remaining problem is raised by the fact that there was a lapse time of up to a year between closing the MR and filling out the survey. Thus, there is a possibility of information loss due to the time lapse between solving the problem and describing it. However, having the person who was in charge of the problem at time of closure is still much better than having someone who had no involvement in the problem interpreting the MR for the survey. This lack of

“freshness” could of course be resolved by making the fault survey part of the normal process of closing an MR.

One remaining caveat: the overall proportions of the faults may be affected by the fact that data is kept only during the testing phase of evolution. MRs for the entire process from the receipt of the requirements to the release of the system would, of course, give a much more accurate picture. We note, however, that this approach of keeping track of faults only once testing has begun is pretty much standard for most software developments and therefore only a very minor issue.

On the whole, we believe that the few problems with our empirical design are significantly outweighed by the evidence supporting our claim that our data is valid and that there are no other factors responsible for our results.

What Can You Do with the Results?

The main question for any empirical study is: “what do these results mean for me in the context of my work as a software developer?” Part of that answer depends on how representative the study is, and there are two different ways of answering that question.

The first way is to ask the question: “how representative is this release in the context of all the releases for this system that has been studied?” If it is not representative of the system and its various releases, then its general usefulness is not clear. In this case, we claim that it is representative because the mix of fault fixes, new features, and improvements was the same as for previous releases. For the first few releases after this one, however, there was an increased emphasis on removing faults before the previous mix of corrective, adaptive, and perfective changes was resumed.

Given a positive answer to the first question, then the second way to answer this question is to ask: “how representative is this release of this system of other software systems?” With respect to other large-scale, highly fault-tolerant, ultra-reliable real-time systems, this release would represent this small class of systems in that it is built and evolved in the context of a commonly used Unix Development Environment using a commonly used programming language such as C. One would expect to see similar kinds of problems in such systems.

How relevant is it to the development of other types of software systems? We would claim that it is highly relevant. Look at the top five fault types. There is nothing there that would lead one to believe that the main problems were domain specific. Indeed the entire list of faults, with a few exceptions, would be the kinds of things found in pretty much any software system development, whatever the domain or size of the project. We would further claim that there is nothing in the list of underlying causes that would preclude the vast variety of other types of software developments. We would make a similar claim for the means of prevention. The primary differences would be in the observed frequencies of the various faults, causes, and means of preventions.

The design of the study is certainly applicable, no matter what the size or domain of the software system being developed. The data itself is also applicable if you find you have the same frequently observed problems. There is an internal consistency to the data and their interrelationships that supports this claim.

What Have We Learned?

The results of the two studies are summarized as follows:

- Problems with requirements, design, and coding accounted for 34% of the total MRs. Requirements account for about 5% of the total MRs and, although not extremely numerous, are particularly important because they have been found so late in the development process, a period during which they are particularly expensive to fix.
- Testing large, complex real-time systems often requires elaborate test laboratories that are themselves large, complex real-time systems. In the development of this release, testing-related MRs accounted for 25% of the total MRs.
- The fact that 16% of the total MRs are “no problems” and the presence of a significant set of design and coding faults such as “unexpected dependencies” and “interface and design/code complexity” indicate that *lack of system knowledge* is a significant problem in the development of this release.
- Of the design and coding faults, 78% took five days or less to fix; 22% took six or more days to fix. We note that there is a certain overhead factor that is imposed on the fixing of each fault that includes getting consensus, building the relevant pieces of the system, and using the system test laboratory to validate the repairs. Unfortunately, we do not have data on those overhead factors.
- Five fault categories account for 60% of the design and coding faults: internal functionality, interface complexity, unexpected dependencies, low-level logic, and design/code complexity. With the exception of “low-level logic,” this set of faults is what we expect would be significant in evolving a large, complex real-time system.
- Weighting the fault categories by the effort to find and to fix them yielded results that coincide with our intuition of which faults are easy and hard to find and fix.
- “Incomplete/omitted design,” “lack of knowledge,” and “none given” (which we interpret to mean that sometimes we just make a mistake with no deeper, hidden underlying cause) account for the underlying causes for 64% of design and coding faults. The weighting of the effort to fix these underlying causes coincides very nicely with our intuition: faults caused by requirements problems require the most effort to fix, whereas faults caused by ambiguous design and lack of knowledge were among those that required the least effort to fix.
- “Application walk-throughs,” “expert person/documentation,” “guideline enforcement,” and “requirements/design templates” represent 64% of the suggested means of preventing

design and coding faults. As application walk-throughs accounted for 25% of the suggested means of prevention, we believe that this supports Curtis, Krasner, and Iscoe's claim [Curtis et al. 1988] that lack of application knowledge is a significant problem.

- Although informal means of prevention were preferred over formal means, it was the case that informal means of prevention tended to be suggested for faults that required less effort to fix and formal means tended to be suggested for faults that required more effort to fix.
- In Perry and Evangelist [Perry and Evangelist 1985], [Perry and Evangelist 1987], interface faults were seen to be a significant portion of the entire set of faults (68%). However, there was no weighting of these faults versus implementation faults. We found in this study that interface faults were roughly 49% of the entire set of design and coding faults and that they were harder to fix than the implementation faults (see the previous discussion). Not surprisingly, formal requirements and formal interface specifications were suggested as significant means of preventing interface faults.

The system reported here was developed and evolved using the current "best practice" techniques and tools with well-qualified practitioners. Because of this fact, we feel that the data point is generalizable to other large-scale real-time systems. With this in mind, we offer the following recommendations to improve the current "best practice":

- Obtain fault data throughout the entire development/evolution cycle (not just in the testing cycle), and use it monitor the progress of the process.
- Incorporate the fault survey as an integral part of MR closure and gather the fault-related information while it is fresh in the developer's mind. This data provides the basis for measurement-based process improvement where the current most frequent or most costly faults are remedied.
- Incorporate the informal, people-intensive means of prevention into the current process (such as application walk-throughs, expert person or documentation, guideline enforcement, etc.). As our survey has shown, this will yield benefits for the majority of the faults reported here.
- Introduce techniques and tools to increase the precision and completeness of requirements, architecture, and design documents. This will yield benefits for those faults that were generally harder to fix and will help to detect the requirements, architecture, and design problems earlier in the life cycle.

We close with several lessons learned that may go a long way toward the improvement of future system developments:

- The fastest way to product improvement as measured by reduced faults is to hire people who are knowledgeable about the domain of the product. Remember, lack of knowledge tended to dominate the underlying causes. The fastest way to increase the knowledge needed to reduce faults is to hire knowledgeable people.

- One of the least important ways to improve software developments is to use a “better” programming language. We found relatively few problems that would have been solved by the use of better programming languages.
- Techniques and tools that help to understand the system and the implications of change should be emphasized in improving a development environment. Remember that knowledge-intensive activities tended to dominate the means of prevention.

Acknowledgments

My special thanks to Carol Steig for her earlier work with me on this project. David Rosik contributed significantly to the general MR survey; Steve Bruun produced the cross-tabulated statistical analysis for the design/coding survey and contributed, along with Carolyn Larson, Julie Federico, H. C. Wei, and Tony Lenard, to the analysis of the design/coding survey; and Clive Loader increased our understanding of the Chi-Square analysis. We especially thank Marjory P. Yuhas and Lew G. Anderson for their unflagging support of this work. And finally, we thank all those who participated in the survey.

References

- [Basili and Hutchens 1983] Basili, Victor R., and David H. Hutchens. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering* SE-9(6): 664–672.
- [Basili and Perricone 1984] Basili, Victor R., and Barry T. Perricone. 1984. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM* 27(1): 42–52.
- [Basili and Weiss 1984] Basili, Victor R., and David M. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* SE-10(6): 728–738.
- [Boehm 1981] Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.
- [Bowen 1980] Bowen, John B. 1980. Standard Error Classification to Support Software Reliability Assessment. *Proceedings of the AFIPS Joint Computer Conferences, 1980 National Computer Conference*: 697–705.
- [Brooks 1995] Brooks, Frederick P., Jr. 1995. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition. Addison-Wesley.
- [Curtis et al. 1988] Curtis, Bill, Herb Krasner, and Neil Iscoe. 1988. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM* 31(11): 1268–1287.
- [Endres 1975] Endres, Albert. 1975. An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering* SE-1(2): 140–149.

- [Fenton and Ohlsson 2000] Fenton, N.E., and N. Ohlsson. 2000. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering* SE-26(8): 797–814.
- [Glass 1981] Glass, Robert L. 1981. Persistent Software Errors. *IEEE Transactions on Software Engineering* SE-7(2): 162–168.
- [Graves et al. 2000] Graves, T.L., A.F. Karr, J.S. Marron, and H. Siy. 2000. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering* SE-26(7): 653–661.
- [Lehman and Belady 1985] Lehman, M.M., and L.A. Belady. 1985. *Program Evolution: Processes of Software Change*. London: Academic Press.
- [Leszak et al. 2000] Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2000. A Case Study in Root Cause Defect Analysis. *Proceedings of the 22nd International Conference on Software Engineering*: 428–437.
- [Leszak et al. 2002] Leszak, Marek, Dewayne E. Perry, and Dieter Stoll. 2002. Classification and Evaluation of Defects in a Project Retrospective. *Journal of Systems and Software* 61(3): 173–187.
- [Musa et al. 1987] Musa, J.D., A. Jannino, and K. Okumoto. 1987. *Software Reliability*. New York: McGraw-Hill.
- [Ostrand and Weyuker 1984] Ostrand, Thomas J., and Elaine J. Weyuker. 1984. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software*, 4(4): 289–300.
- [Ostrand and Weyuker 2002] Ostrand, Thomas J., and Elaine J. Weyuker. 2002. The Distribution of Faults in a Large Industrial Software System. *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*: 55–64.
- [Ostrand et al. 2005] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2005. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering* SE-31(4): 340–355.
- [Perry and Evangelist 1985] Perry, Dewayne E., and W. Michael Evangelist. 1985. An Empirical Study of Software Interface Errors. *Proceedings of the International Symposium on New Directions in Computing*: 32–38.
- [Perry and Evangelist 1987] Perry, Dewayne E., and W. Michael Evangelist. 1987. An Empirical Study of Software Interface Faults—An Update. *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences II*: 113–126.
- [Perry and Steig 1993] Perry, Dewayne E., and Carol S. Steig. 1993. Software Faults in Evolving a Large, Real-Time System: A Case Study. In *Lecture Notes in Computer Science, Volume 717: Proceedings of the 4th European Software Engineering Conference*, ed. I. Sommerville and M. Paul, 48–67.

- [Perry and Wolf 1992] Perry, Dewayne E., and Alexander L. Wolf. 1992. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes* 17(4): 40–52.
- [Perry et al. 2001] Perry, Dewayne E., Harvey P. Siy, and Lawrence G. Votta. 2001. Parallel Changes in Large Scale Software Development: An Observational Case Study. *Transactions on Software Engineering and Methodology* 10(3): 308–337.
- [Purushothaman and Perry 2005] Purushothaman, Ranjith, and Dewayne E. Perry. 2005. Toward Understanding the Rhetoric of Small Source Code Changes. *IEEE Transactions on Software Engineering* SE-31(6): 511–526.
- [Rowland et al. 1983] Rowland, B.R., R.E. Anderson, and P. S. McCabe. 1983. The 3B20D Processor & DMERT Operating System: Software Development System. *The Bell System Technical Journal* 62(1/2): 275–290.
- [Schneidewind and Hoffmann 1979] Schneidewind, N.F., and Heinz-Michael Hoffmann. 1979. An Experiment in Software Error Data Collection and Analysis. *IEEE Transactions on Software Engineering* SE-5(3): 276–286.
- [Shao et al. 2007] Shao, D., S. Khurshid, and D. Perry. 2007. Evaluation of Semantic Interference Detection in Parallel Changes: An Exploratory Experiment. *Proceedings of the 23rd IEEE International Conference on Software Maintenance*: 74–83.
- [Siegel et al. 1988] Siegel, Sidney, and N. John Castellan, Jr. 1988. *Nonparametric Statistics for the Behavioral Sciences*, Second Edition. New York: McGraw-Hill.
- [Thayer et al. 1978] Thayer, Thomas A., Myron Lipow, and Eldred C. Nelson. 1978. Software Reliability—A Study of Large Project Reality. In *TRW Series of Software Technology*, Volume 2. Amsterdam: North-Holland.
- [Thione and Perry 2005] Thione, G. Lorenzo, and Dewayne E. Perry. 2005. Parallel Changes: Detecting Semantic Interferences. *Proceedings of the 29th Annual International Computer Software and Applications Conference*: 47–56.

Identifying and Removing Software Clones

Rainer Koschke

Universität Bremen, Germany

Summary. Ad-hoc reuse through copy-and-paste occurs frequently in practice affecting the evolvability of software. Researchers have investigated ways to locate and remove duplicated code. Empirical studies have explored the root causes and effects of duplicated code and the evolution of duplicated code. This chapter summarizes the state of the art in detecting, managing, and removing software redundancy. It describes consequences, pros and cons of copying and pasting code.

2.1 Introduction

A venerable and long-standing goal and ideal in software development is to avoid duplication and redundancy. Yet, in reality code duplication is a common habit. Several authors report on 7–23% code duplication [29, 291, 303]; in one extreme case even 59% [156].

Duplication and redundancy can increase the size of the code, make it hard to understand the many code variants, and cause maintenance headaches. The goal of avoiding redundancy has provided the impetus to investigations on software reuse, software refactoring, modularization, and parameterization. Even in the face of the ethic of avoiding redundancy, in practice software frequently contains many redundancies and duplications. For instance the technique of “code scavenging” is frequently used, and works by copying and then pasting code fragments, thereby creating so-called “clones” of duplicated or highly similar code. Redundancies can also occur in various other ways, including because of missed reuse opportunities, purposeful duplication because of efficiency concerns, and duplication through parallel or forked development threads.

Because redundancies frequently exist in code, methods for detecting and removing them from software are needed in many contexts. Over the past few decades, research on clone detection have contributed towards addressing the issue. Techniques for finding similar code and on removing duplication have been investigated in several specific areas such as software reverse engineering, plagiarism in student programs, copyright infringement investigation, software evolution analysis, code

compaction (e.g., for mobile devices), and design pattern discovery and extraction. Common to all these research areas are the problems involved in understanding the redundancies and finding similar code, either within a software system, between versions of a system, or between different systems.

Although this research has progressed over decades, only recently has the pace of activity in this area picked up such that significant research momentum could be established. This chapter summarizes the state of the art in detecting, managing, and removing software redundancy. It describes consequences, pros and cons of copying and pasting code.

Software clones are important aspects in software evolution. If a system is to be evolved, its clones should be known in order to make consistent changes. Cloning is often a strategic means for evolution. For instance, copies can be made to create a playground for experimental feature evolution, where modifications are made in cloned code of a mature feature reducing the risk to break stable code. Once stable, the clone can replace its original. Often, cloning is the start of a new branch of evolution if the changes in the cloned code are not merged back to the main development branch. Clone detection techniques play an important role in software evolution research where attributes of the same code entity are observed over multiple versions. Here, we need to identify for an entity in one version the corresponding entity in the next version (known as *origin analysis* [568]). If refactoring (as for instance *renaming*) is applied between versions, the relation between entities of different versions is not always obvious. And last but not least, the evolution of clones can be studied to better understand the nature of cloning in practice.

2.2 Software Redundancy, Code Cloning, and Code Duplication

There are different forms of redundancy in software. Software comprises both programs and data. In the data base community, there is a clear notion of redundancy that has led to various levels of normal forms. A similar theory does not yet exist for computer programs.

In computer programs, we can also have different types of redundancy. We should note that not every type of redundancy is harmful. For instance, programming languages use redundant declarations so that a compiler is able to check consistency between declarations and their uses. Also, at the architectural level, n-version programming is a strategy in which redundancy is purposefully and consciously used to implement reliable systems.

Sometimes *redundant* is used also in the sense of *superfluous* in the software engineering literature. For instance, Xie and Engler show that superfluous (they call them redundant) operations such as idempotent operations, assignments that are never read, dead code, conditional branches that are never taken, and redundant NULL-checks can pinpoint potential errors [550, 551].

Redundant code is also often misleadingly called *cloned* code in the literature—although that implies that one piece of code is derived from the other one in the original sense of this word. According to the Merriam-Webster dictionary, a *clone*

is one that appears to be a copy of an original form. It is a synonym to *duplicate*. Although cloning leads to redundant code, not every redundant code is a clone. There may be cases in which two code segments that are no copy of each other just happen to be similar or even identical by accident. Also, there may be redundant code that is semantically equivalent but has a completely different implementation.

There is no agreement in the research community on the exact notion of redundancy and cloning. Ira Baxter's definition of clones expresses this vagueness:

Clones are segments of code that are similar according to some definition of similarity.
—Ira Baxter, 2002

According to this definition, there can be different notions of similarity. They can be based on text, lexical or syntactic structure, or semantics. They can even be similar if they follow the same pattern, that is, the same building plan. Instances of design patterns and idioms are similar in that they follow a similar structure to implement a solution to a similar problem.

Semantic similarity relates to the observable behavior. A piece of code, *A*, is semantically similar to another piece of code, *B*, if *B* subsumes the functionality of *A*, in other words, they have “similar” pre and post conditions.

Unfortunately, detecting such semantic similarity is undecidable in general although it would be worthwhile as you can often estimate the number of developers of a large software system by the number of hash table or list implementations you find.

Another definition of cloning considers the program text: Two code fragments form a clone if their program text is similar. The two code fragments may or may not be equivalent semantically. These pieces are redundant because one fragment may need to be adjusted if the other one is changed. If the code fragments are executable code, their behavior is not necessarily equivalent or subsumed at the concrete level, but only at a more abstract level. For instance, two code pieces may be identical at the textual level including all variable names that occur within but the variable names are bound to different declarations in the different contexts. Then, the execution of the code changes different variables. Figure 2.1 shows two textually identical segments in the line range of 4–6 and 10–12, respectively. The semantic difference is that the first segment sets a global variable whereas the second one a local variable. The common abstract behavior of the two code segments is to iterate over a data structure and to increase a variable in each step.

Program-text similarity is most often the result of *copy&paste*; that is, the programmer selects a code fragment and copies it to another location. Sometimes, these programmers are forced to copy because of limitations of the programming language. In other cases, they intend to reuse code. Sometimes these clones are modified slightly to adapt them to their new environment or purpose.

Clearly, the definition of redundancy, similarity, and cloning in software is still an open issue. There is little consensus in this matter. A study by Walenstein et al. [532], for instance, reports on differences among different human raters for clone candidates. In this study, clones were to be identified that ought to be removed and Walenstein et al. gave guidelines towards clones worthwhile being removed. The

```

1 int sum = 0;
2
3 void foo(Iterator iter){
4     for (item = first(iter); has_more(iter); item = next(iter)){
5         sum = sum + value (item);
6     }
7 }
8 int bar(Iterator iter){
9     int sum = 0;
10    for (item = first(iter); has_more(iter); item = next(iter)){
11        sum = sum + value (item);
12    }
13 }
```

Fig. 2.1. Example of code clones

human raters of the clones proposed by automated tools did rarely agree upon what constitutes a clone worth to be removed. While the sources of inter-rater difference could be the insufficient similarity among clones or the appraisal of the need for removal, the study still highlights that there is no clear consensus yet, even for task-specific definitions of clones.

Another small study was performed at the Dagstuhl seminar 06301 “Duplication, Redundancy, and Similarity in Software” 2007. Cory Kapser elicited judgments and discussions from world experts regarding what characteristics define a code clone. Less than half of the clone candidates he presented to these experts had 80% agreement amongst the judges. Judges appeared to differ primarily in their criteria for judgment rather than their interpretation of the clone candidates.

2.3 Types of Clones

Program-text clones can be compared on the basis of the program text that has been copied. We can distinguish the following types of clones accordingly:

- **Type 1** is an exact copy without modifications (except for whitespace and comments).
- **Type 2** is a syntactically identical copy; only variable, type, or function identifiers have been changed.
- **Type 3** is a copy with further modifications; statements have been changed, added, or removed.

Baker further distinguishes so called parameterized clones [28], which are a subset of type-2 clones. Two code fragments *A* and *B* are a parameterized clone pair if there is a bijective mapping from *A*'s identifiers onto *B*'s identifiers that allows an identifier substitution in *A* resulting in *A'* and *A'* is a type-1 clone to *B* (and vice versa).

While type-1 and type-2 clones are precisely defined and form an equivalence relation, the definition of type-3 clones is inherently vague. Some researchers consider

Table 2.1. Classification by Balazinska et al. [33] ©[1999] IEEE

- | |
|--|
| <ul style="list-style-type: none"> • difference in method attributes (<code>static</code>, <code>private</code>, <code>throws</code>, etc.) • single-token difference in function body <ul style="list-style-type: none"> – further distinction into type of token: <ul style="list-style-type: none"> – called method – parameter type – literal – ... • token-sequence difference in function body <ul style="list-style-type: none"> – one unit (expression or statement) differs in token sequence – two units – more than two units |
|--|

two consecutive type-1 or type-2 clones together forming a type-3 clone if the gap in between is below a certain threshold of lines [29, 328]. Another precise definition could be based on a threshold for the Levenshtein Distance, that is, the number of deletions, insertions, or substitutions required to transform one string into another. There is no consensus on a suitable similarity measure for type-3 clones yet.

The above simple classification is still very rough. Balazinska et al. introduced a more refined classification for function clones [33] as described in Table 2.1. This classification makes sense for selecting a suitable strategy for clone removal. For instance, the design pattern *TemplateMethod* may be used to factor out differences in the types used in different code fragments or the design pattern *Strategy* can be used to factor out algorithmic differences [31, 32]. Furthermore Balazinska et al. argue that each class is associated with a different risk in clone removal.

Kapser et al.'s classification is the most elaborated classification to date [267, 265, 264] (cf. Table 2.2). The first level is a hint about the distance of clones. An argument can be made (although there is no empirical study on this hypothesis) that it is likely that clones between files are more problematic than within the same file as that it is more likely to overlook the former clones when it comes to consistent changes. The second decision distinguishes which syntactic units are copied. The third gives the degree of similarity and the fourth may be used to filter irrelevant or spurious clones.

2.4 The Root Causes for Code Clones

A recent ethnographic study by Kim and Notkin [277] has shed some light on why programmers copy and paste code. By observing programmers in their daily practice they identified the following reasons.

Sometimes programmers are simply forced to duplicate code because of limitations of the programming language being used. Analyzing these root causes in more detail could help to improve the language design.

Furthermore, programmers often delay code restructuring until they have copied and pasted several times. Only then, they are able to identify the variabilities of their

Table 2.2. Classification by Kapser et al. [265, 264] ©[2003] IEEE

<ol style="list-style-type: none"> 1. At first level, distinguish clones within the same or across different files 2. then, according to type of region: <ul style="list-style-type: none"> • functions • declarations • macros • hybrids (in more than one of the above) • otherwise (among typedefs, variable declarations, function signatures) 3. then, degree of overlap or containment 4. then, according to type of code sequence: <ul style="list-style-type: none"> • initialization clones (first five lines) • finalization clones (last five lines) • loop clones (60% overlap of bodies) • switch and if (60% overlap of branches) • multiple conditions: several switch and if statements • partial conditions: branches of switch/if are similar
--

code to be factored out. Creating abstract generic solutions in advance often leads to unnecessarily flexible and hence needlessly complicated solutions. Moreover, the exact variabilities may be difficult to foresee. Hence, programmers tend to follow the idea of extreme programming in the small by not investing too much effort in speculative planning and anticipation.

Systems are modularized based on principles such as information hiding, minimizing coupling, and maximizing cohesion. In the end—at least for systems written in ordinary programming languages—the system is composed of a fixed set of modules. Ideally, if the system needs to be changed, only a very small number of modules must be adjusted. Yet, there are very different change scenarios and it is not unlikely that the chosen modularization forces a change to be repeated for many modules. The triggers for such changes are called *cross-cutting concerns* (see also Chapter 9). For instance, logging is typically a feature that must be implemented by most modules. Another example is parameter checking in defensive programming where every function must check its parameters before it fulfills its purpose [92]. Then copy&paste dependencies reflect important underlying design decisions, namely, cross-cutting concerns.

Another important root cause is that programmers often reuse the copied text as a template and then customize the template in the pasted context.

Kapser et al. have investigated clones in large systems [266]. They found what they call *patterns of cloning* where cloning is consciously used as an implementation strategy. In their case study, they found the following cloning patterns:

Forking is cloning used to bootstrap development of similar solutions, with the expectation that evolution of the code will occur somewhat independently, at least in the short term. The assumption is that the copied code takes a separate evolu-

tion path independent of the original. In such a case, changes in the copy may be made that have no side effect on the original code.

Templating is used as a method to directly copy behavior of existing code but appropriate abstraction mechanisms are unavailable. It was also identified as a main driver for cloning in Kim and Notkin's case study [277]. Templating is often found when a reused library has a relatively fixed protocol (that is, a required order of using its interface items) which manifests as laying out the control flow of the interface items as a fixed pattern. For instance, the code in Fig. 2.1 uses a fixed iteration scheme for variable `iter`.

Customization occurs when currently existing code does not adequately meet a new set of requirements. The existing code is cloned and tailored to solve this new problem.

Very likely other more organizational aspects play a role, too. Time pressure, for instance, does not leave much time to search for the best long-term solution. Unavailable information on the impact of code changes leads programmers to create copies in which they make the required enhancement; such changes then are less likely to affect the original code negatively. Inadequate performance measures of programmers' productivity in the number of lines of code they produce neither invite programmers to avoid duplicates.

2.5 Consequences of Cloning

There are plausible arguments that code cloning increases maintenance effort. Changes must be made consistently multiple times if the code is redundant. Often it is not documented where code has been copied. Manual search for copied code is infeasible for large systems and automated clone detection is not perfect when changes are made to the copies (see Section 2.8). Furthermore during analysis, the same code must be read over and over again, then compared to the other code just to find out that this code has already been analyzed. Only if you make a detailed comparison, which can be difficult if there are subtle differences in the code or its environment, you can be sure that the code is indeed the same. This comparison can be fairly expensive. If the code would have been implemented only once in a function, this effort could have been avoided completely.

For these reasons, code cloning is number one on the stink parade of bad smells by Beck and Fowler [183]. But there are also counter arguments. In Kapser and Godfrey's study [266], code cloning is a purposeful implementation strategy which may make sense under certain circumstances (see Section 2.4).

Cordy makes a similar statement [128]. He argues that in the financial domain, cloning is the way in which designs are reused. Data processing programs and records across an organization often have very similar purposes, and, consequently, the data structures and programs to carry out these tasks are therefore very similar. Cloning becomes then a standard practice when authoring a new program. Opponents would argue that a better means would be to pursue systematic and organized reuse through software product lines.

Cordy also argues that the attempt to avoid cloning may lead to higher risks. Making changes to central data structures bears the risk to break existing applications and requires to run expensive regression tests. Instead programmers tend to copy the data structure if they want to restructure or add a different view and make the necessary changes in the copy. Even the argument that errors must be fixed in every copy does not count, he states. Errors would not necessarily be fixed in the original data structure because the many running applications may already rely on these errors, Cordy argues. On the other hand, repeated work, need for data migration, and risk of inconsistency of data are the price that needs to be paid following this strategy. The Y2K problem has shown how expensive and difficult it is to remedy systems that have suffered from massive decentralized use of data structures and algorithms.

While it is difficult to find arguments for type-1 and type-2 clones, one can more easily argue in favor of type-3 clones. It is not clear when you have type-3 clones whether the unifying solution would be easier to maintain than several copies with small changes. Generic solutions can become overly complicated. Maintainability can only be defined in a certain context with controlled parameters. That is, a less sophisticated programmer may be better off maintaining copied code than a highly parameterized piece of code. Moreover, there is a risk associated with removing code clones [128]. The removal requires deep semantic analyses and it is difficult to make any guarantees that the removal does not introduce errors. There may be even organizational reasons to copy code. Code cloning could, for instance, be used to disentangle development units [128].

The current debate lacks empirical studies on the costs and benefits of code cloning. There are very few empirical studies that explore the interrelationship of code cloning and maintainability. All of them focus on code cloning and errors as one (out of many) maintainability aspect.

Monden et al. [374] analyzed a large system consisting of about 2,000 modules written in 1 MLOC lines of Cobol code over a period of 20 years. They used a token-based clone detector (cf. Section 2.8.2) to find clones that were at least 30 lines long. They searched for correlations of maximal clone length with change frequency and number of errors. They found that most errors were reported for modules with clones of at least 200 lines. They also found many errors—although less than in those with longer clones—in modules with shorter clones up to 50 lines. Yet, interestingly enough, they found the lowest error rate for modules with clones of 50 to 100 lines. Monden et al. have not further analyzed why these maintainability factors correlate in such a way with code cloning.

Chou et al. [113] investigated the hypothesis that if a function, file, or directory has one error, it is more likely that it has others. They found in their analysis of the Linux and OpenBSD kernels that this phenomenon can be observed most often where programmer ignorance of interface or system rules combines with copy-and-paste. They explain the correlation of bugs and copy-and-paste primarily by programmer ignorance, but they also note that—in addition to ignorance—the prevalence of copy-and-paste error clustering among different device drivers and versions suggests that programmers believe that “working” code is correct code. They note that if the copied

code is incorrect, or it is placed into a context it was not intended for, the assumption of goodness is violated.

Li et al. [328] use clone detection to find bugs when programmers copy code but rename identifiers in the pasted code inconsistently. On average, 13% of the clones flagged as copy-and-paste bugs by their technique turned out to be real errors for the systems *Linux kernel*, *FreeBSD*, *Apache*, and *PostgreSQL*. The false positive rate of their technique is 73% on average, where on average 14% of the potential problems are still under analysis by the developers of the analyzed systems.

2.6 Clone Evolution

There are a few empirical studies on the evolution of clones, which describe some interesting observations. Antoniol et al. propose time series derived from clones over several releases of a system to monitor and predict the evolution of clones [14]. Their study for the data base system *mSQL* showed that their prediction of the average number of clones per function is fairly reliable. In another case study for the Linux kernel, they found that the scope of cloning is limited [15]. Only few clones can be found across subsystems; most clones are completely contained within a subsystem. In the subsystem *arch*, constituting the hardware architecture abstraction layer, newer hardware architectures tend to exhibit slightly higher clone rates. The explanation for this phenomenon is that newer modules are often derived from existing similar ones. The relative number of clones seems to be rather stable, that is, cloning does not occur in peaks. This last result was also reported by Godfrey and Tu who noticed that cloning is common and steady practice in the Linux kernel [205]. However, the cloning rate does increase steadily over time. Li et al. [328] observed for the Linux kernel in the period of 1994 to 2004 that the redundancy rate has increased from about 17% to about 22%. They observed a similar behavior for FreeBSD. Most of the growth of redundancy rate comes from a few modules, including *drivers* and *arch* in Linux and *sys* in FreeBSD. The percentage of copy-paste code increases more rapidly in those modules than in the entire software suite. They explain this observation by the fact that Linux supports more and more similar device drivers during this period.

Kim et al. analyzed the clone genealogy for two open-source Java systems using historical data from a version control system [278]. A clone genealogy forms a tree that shows how clones derive in time over multiple versions of a program from common ancestors. Beyond that, the genealogy contains information about the differences among siblings. Their study showed that many code clones exist in the system for only a short time. Kim et al. conclude that extensive refactoring of such short-lived clones may not be worthwhile if they likely diverge from one another very soon. Moreover, many clones, in particular those with a long lifetime that have changed consistently with other elements in the same group cannot easily be avoided because of limitations of the programming language.

One subproblem in clone evolution research is to track clones between versions. Duala-Ekoko and Robillard use a clone region descriptor [155] to discover a clone

of version n in version $n + 1$. A clone region descriptor is an approximate location that is independent from specifications based on lines of source code, annotations, or other similarly fragile markers. Clone region descriptors capture the syntactic block nesting of code fragments. A block therein is characterized by its type (e.g., for or while), a string describing a distinguishing identifier for the block (the anchor), and a corroboration metric. The anchor of a loop, for instance, is the condition as string. If two fragments are syntactic siblings, their nesting and anchor are not sufficient to distinguish them. In such cases, the corroboration metric is used. It measures characteristics of the block such as cyclomatic complexity and fan-out of the block.

2.7 Clone Management

Clone management aims at identifying and organizing existing clones, controlling growth and dispersal of clones, and avoiding clones altogether. Lague et al. [303] and Giesecke [199] distinguish three main lines of clone management:

- *preventive* clone management (also known as *preventive control* [303]) comprises activities to avoid new clones
- *compensative* clone management (also known as *problem mining* [303]) encompasses activities aimed at limiting the negative impact of existing clones that are to be left in the system
- *corrective* clone management covers activities to remove clones from a system

This section describes research in these three areas.

2.7.1 Corrective Clone Management: Clone Removal

If you do want to remove clones, there are several way to do so. There are even commercial tools such as *CloneDr*¹ by *Semantic Designs* to automatically detect and remove clones. Cloning and automatic abstraction and removal could even be a suitable implementation approach as hinted by Ira Baxter:

Cloning can be a good strategy if you have the right tools in place. Let programmers copy and adjust, and then let tools factor out the differences with appropriate mechanisms.
—Ira Baxter, 2002

In simple cases, you can use functional abstraction to replace equivalent copied code by a function call to a newly created function that encapsulates the copied code [166, 287]. In more difficult cases, when the difference is not just in the variable names that occur in the copied code, one may be able to replace by macros if the programming languages comes with a preprocessor. A preprocessor offers textual transformations to handle more complicated replacements. If a preprocessor is available, one can also use conditional compilation. As the excessive use of macros and

¹ Trademark of Semantic Designs, Inc.

conditional compilation may introduce many new problems, the solution to the redundancy problem may be found at the design level. The use of design patterns is an option to avoid clones by better design [31, 32]. Yet, this approach requires much more human expertise and, hence, can be less automated. Last but not least, one can develop code generators for highly repetitive code.

In all approaches, it is a challenge to cut out the right abstractions and to come up with meaningful names of generated functions or macros. Moreover, it is usually difficult to check the preconditions for these proposed transformations—be they manual or automated—in order to assure that the transformation is semantic preserving.

2.7.2 Preventive and Compensative Clone Management

Rather than removing clones after the offense, it may be better to avoid them right from the beginning by integrating clone detection in the normal development process. Lague et al. identify two ways to integrate clone detection in normal development [303].

It can be used as *preventive control* where the code is checked continuously—for instance, at each check-in in the version control system or even on the fly while the code is edited—and the addition of a clone is reported for confirmation.

A complementary integration is *problem mining* where the code currently under modification is searched in the rest of the system. The found segments of code can then be checked whether the change must be repeated in this segment for consistency.

Preventive control aims at avoiding code clones when they occur first whereas problem mining addresses circumstances in which cloning has been used for a while.

Lague et al. assessed the benefits of integrating clone detection in normal development by analyzing the three-year version history of a very large procedural telecommunication system [303]. In total, 89 millions of non-blank lines (including comments) were analyzed, for an average size of 14.83 million lines per version. The average number of functions per release was 187,000.

Problem mining is assessed by the number of functions changed that have clones that were not changed; that is, how often a modification was missed potentially. Preventive control is assessed by the number of functions added that were similar to existing functions; that is, the code that could have been saved.

It is interesting to note, that—contrary to their expectations—they observed a low rate of growth in the number of overall clones in the system, due to the fact that many clones were actually removed from the system.

They conclude from their data that preventive control would help to lower the number of clones. Many clones disappeared only long after the day they came into existence. Early detection of clones could lead to taking this measure earlier.

They also found that problem mining could have provided programmers with a significant number of opportunities for correcting problems before end-user experienced them. The study indicates a potential for improving the software quality and customer satisfaction through an effective clone management strategy.

An alternative to clone removal is to live with clones consciously. Clones can be managed, linked, and changed simultaneously using linked editing as proposed

by Toomim et al. [504]. Linked editing is also used by Duala-Ekoko and Robillard. Linked editing allows one to link two or more code clones persistently. The differences and similarities are then analyzed, visualized, and recorded. If a change needs to be made, linked editing allows programmers to modify all linked elements simultaneously, or particular elements individually. That is, linked editing allows the programmer to edit all instances of a given clone at once, as if they were a single block of code. It overcomes some of the problems of duplicated code, namely, verbosity, tedious editing, lost clones, and unobservable consistency without requiring extra work from the programmer.

2.8 Clone Detection

While there is an ongoing debate as to whether remove clones, there is a consensus about the importance to at least detect them. Clone avoidance during normal development, as described in the previous section, as well as making sure that a change can be made consistently in the presence of clones requires to know where the clones are. Manual clone detection is infeasible for large systems, hence, automatic support is necessary.

Automated software clone detection is an active field of research. This section summarizes the research in this area. The techniques can be distinguished at the first level in the type of information their analysis is based on and at the second level in the used algorithm.

2.8.1 Textual Comparison

The approach by Rieger et al. compares whole lines to each other textually [156]. To increase performance, lines are partitioned using a hash function for strings. Only lines in the same partition are compared. The result is visualized as a dotplot, where each dot indicates a pair of cloned lines. Clones may be found as certain patterns in those dotplots visually. Consecutive lines can be summarized to larger cloned sequences automatically as uninterrupted diagonals or displaced diagonals in the dotplot.

Johnson [258, 259] uses the efficient string matching by Karp and Rabin [268, 269] based on fingerprints, that is, a hash code characterizing a string is used in the search.

Marcus et al. [345] compare only certain pieces of text, namely, identifiers using latent semantic indexing, a technique from information retrieval. Latent semantic analysis is a technique in natural language processing analyzing relationships between a set of documents and the terms they contain by producing a set of concepts related to the documents and terms. The idea here is to identify fragments in which similar names occur as potential clones.

2.8.2 Token Comparison

Baker's technique is also a line-based comparison. Instead of a string comparison, the token sequences of lines are compared efficiently through a suffix tree. A suffix tree for a string S is a tree whose edges are labeled with substrings of S such that each suffix of S corresponds to exactly one path from the tree's root to a leaf.

First, Baker's technique summarizes each token sequence for a whole line by a so called *functor* that abstracts of concrete values of identifiers and literals [29]. The functor characterizes this token sequence uniquely. Assigning functors can be viewed as a perfect hash function. Concrete values of identifiers and literals are captured as parameters to this functor. An encoding of these parameters abstracts from their concrete values but not from their order so that code fragments may be detected that differ only in systematic renaming of parameters. Two lines are clones if they match in their functors and parameter encoding.

The functors and their parameters are summarized in a suffix tree, a tree that represents all suffixes of the program in a compact fashion. A suffix tree can be built in time and space linear to the input length [356, 30]. Every branch in the suffix tree represents program suffixes with common beginnings, hence, cloned sequences.

Kamiya et al. increase recall for superficially different, yet equivalent sequences by normalizing the token sequences [263]. For instance, each single statement after the lexical patterns `if(...)`, `for(...)`, `while(...)`, and `do` and `else` in C++ is transformed to a compound block; e.g., `if (a) b = 2;` is transformed to `if (a) {b = 2;}`. Using this normalization, the `if` statement can be matched with the equivalent (with parameter replacement) code `if (x) {y = 2;}`.

Because syntax is not taken into account, the found clones may overlap different syntactic units, which cannot be replaced through functional abstraction. Either in a preprocessing [485, 127, 204] or post-processing [234] step, clones that completely fall in syntactic blocks can be found if block delimiters are known. Preprocessing and postprocessing both require some syntactic information—gathered either lightweight by counting tokens opening and closing syntactic scopes or island grammars [377] or a full-fledged syntax analysis [204].

2.8.3 Metric Comparison

Merlo et al. gather different metrics for code fragments and compare these metric vectors instead of comparing code directly [303, 291, 353, 289]. An allowable distance (for instance, Euclidean distance) for these metric vectors can be used as a hint for similar code. Specific metric-based techniques were also proposed for clones in web sites [151, 307].

2.8.4 Comparison of Abstract Syntax Trees

Baxter et al. partition subtrees of the abstract syntax tree (AST) of a program based on a hash function and then compare subtrees in the same partition through tree matching (allowing for some divergences) [47]. A similar approach was proposed

earlier by Yang [557] using dynamic programming to find differences between two versions of the same file.

Suffix trees—central to token-based techniques following Baker’s idea—can also be used to detect sequences of identical AST nodes. In the approach by Koschke et al. [294], the AST nodes are serialized in preorder traversal, a suffix tree is created for these serialized AST nodes, and the resulting maximally long AST node sequences are then cut according to their syntactic region so that only syntactically closed sequences remain.

The idea of metrics to characterize code and to use these metrics to decide which code segments to compare can be adopted for ASTs as well. Jian et al. [257] characterize subtrees with numerical vectors in the Euclidean space \Re and an efficient algorithm to cluster these vectors with respect to the Euclidean distance metric. Subtrees with vectors in one cluster are potential clones.

2.8.5 Comparison of Program Dependency Graphs

Textual as well as token-based techniques and syntax-based techniques depend upon the textual order of the program. If the textual order is changed, the copied code will not be found. Programmers modify the order of the statements in copied code, for instance, to camouflage plagiarism. Or they use code cloning as in the templating implementation strategy (see Section 2.4), where the basic skeleton of an algorithm is reused and then certain pieces are adjusted to the new context.

Yet, the order cannot be changed arbitrarily without changing the meaning of the program. All control and data dependencies must be maintained. A program dependency graph [175] is a representation of a program that represents only the control and data dependency among statements. This way program dependency graph abstract from the textual order. Clones may then be identified as isomorphic subgraphs in a program dependency graph [298, 286]. Because this problem is NP hard, the algorithms use approximative solutions.

2.8.6 Other Techniques

Leitao [324] combines syntactic and semantic techniques through a combination of specialized comparison functions that compare various aspects (similar call subgraphs, commutative operators, user-defined equivalences, transformations into canonical syntactic forms). Each comparison function yields an evidence that is summarized in an evidence-factor model yielding a clone likelihood. Walter et al. [531] and Li et al. [327] cast the search for similar fragments as a data mining problem. Statement sequences are summarized to item sets. An adapted data mining algorithm searches for frequent item sets.

2.9 Comparison of Clone Detection Algorithms

The abundance of clone detection techniques calls for a thorough comparison so that we know the strength and weaknesses of these techniques in order to make an

informed decision if we need to select a clone detection technique for a particular purpose.

Clone detectors can be compared in terms of recall and precision of their findings as well as suitability for a particular purpose. There are several evaluations along these lines based on qualitative and quantitative data.

Bailey and Burd compared three clone and two plagiarism detectors [27]. Among the clone detectors were three of the techniques later evaluated by a subsequent study by Bellon and Koschke [55], namely, the techniques by Kamiya [263], Baxter [47], and Merlo [353]. For the latter technique, Bailey used an own re-implementation; the other tools were original. The plagiarism detectors were JPlag [420] and Moss [452].

The clone candidates of the techniques were validated by Bailey, and the accepted clone pairs formed an oracle against which the clone candidates were compared. Several metrics were proposed to measure various aspects of the found clones, such as scope (i.e., within the same file or across file boundaries), and the findings in terms of recall and precision.

The syntax-based technique by Baxter had the highest precision (100%) and the lowest recall (9%) in this experiment. Kamiya's technique had the highest recall and a precision comparable to the other techniques (72%). The re-implementation of Merlo's metric-based technique showed the least precision (63%).

Although the case study by Bailey and Burd showed interesting initial results, it was conducted on only one relatively small system (16 KLOC). However, because the size was limited, Bailey was able to validate all clone candidates.

A subsequent larger study was conducted by Bellon and Koschke [54, 55]. Their likewise quantitative comparison of clone detectors was conducted for 4 Java and 4 C systems in the range of totaling almost 850 KLOC. The participants and their clone detectors evaluated are listed in Table 2.3.

Table 2.4 summarizes the findings of Bellon and Koschke's study. Row *clone type* lists the type of clones the respective clone detector finds (for clone types, see Section 2.3). The next two rows qualify the tools in terms of their time and space consumption. The data is reported at an ordinal scale --, -, +, ++ where -- is worst (the exact measures can be found in the paper to this study [54, 55]). Recall and precision are determined as in Bailey and Burd's study by comparing the clone

Table 2.3. Participating scientists

Participant	Tool	Comparison
Brenda S. Baker [29]	<i>Dup</i>	Token
Ira D. Baxter [47]	<i>CloneDr</i>	AST
Toshihiro Kamiya [263]	<i>CCFinder</i>	Token
Jens Krinke [298]	<i>Duplix</i>	PDG
Ettore Merlo [353]	<i>CLAN</i>	Function Metrics
Matthias Rieger [156]	<i>Duploc</i>	Text

Table 2.4. Results from the Bellon and Koschke study. Adapted from [54, 55] ©[2007] IEEE

	Baker	Baxter	Kamiya	Krinke	Merlo	Rieger
Clone type	1, 2	1, 2	1, 2, 3	3	1, 2, 3	1, 2, 3
Speed	++	—	+	--	++	?
RAM	+	—	+	+	++	?
Recall	+	—	+	—	—	+
Precision	—	+	—	—	+	—

detectors' findings to a human oracle. The same ordinal scale is used to qualify the results; exact data are reported in the paper [54, 55].

Interestingly, Merlo's tool performed much better in this experiment than in the experiment by Bailey and Burd. However, the difference in precision of Merlo's approach in this comparison to the study by Bailey and Burd can be explained by the fact that Merlo compared not only metrics but also the tokens and their textual images to identify type-1 and type-2 clones in the study by Bellon and Koschke.

While the Bailey/Burd and Bellon/Koschke studies focus on quantitative evaluation of clone detectors, other authors have evaluated clone detectors for the fitness for a particular maintenance task. Rysselberghe and Demeyer [525] compared text-based [156, 438], token-based [29], and metric-based [353] clone detectors for refactoring. They compare these techniques in terms of suitability (can a candidate be manipulated by a refactoring tool?), relevance (is there a priority which of the matches should be refactored first?), confidence (can one solely rely on the results of the code cloning tool, or is manual inspection necessary?), and focus (does one have to concentrate on a single class or is it also possible to assess an entire project?). They assess these criteria qualitatively based on the clone candidates produced by the tools. Figure 2.2 summarizes their conclusions.

Bruntink et al. use clone detection to find cross-cutting concerns in C programs with homogeneous implementations [93]. In their case study, they used *CCFinder*—Kamiya's [263] tool evaluated in other case studies, too—one of the Bauhaus² clone detectors, namely *ccdiml*, which is a variation of Baxter's technique [47], and the PDG-based detector *PDG-DUP* by Komondoor [286]. The cross-cutting concerns they looked for were error handling, tracing, pre and post condition checking, and memory error handling. The study showed that the clone classes obtained by Bauhaus' *ccdiml* can provide the best match with the range checking, null-pointer

criterion	most suitable technique
suitability	metric-based
relevance	no difference
confidence	text-based
focus	no difference

Fig. 2.2. Assessment by Rysselberghe and Demeyer. Adapted from [525] ©[2004] IEEE

² <http://www.axivion.com>.

checking, and error handling concerns. Null-pointer checking and error handling can be found by *CCFinder* almost equally well. Tracing and memory error handling can best be found by *PDG-DUP*.

2.10 Clone Presentation

Because there is typically a huge amount of clones in large systems and these clones differ in various attributes (type, degree of similarity, length, etc.), presentation issues of clone information are critical. This huge information space must be made accessible to a human analyst. The analyst needs a holistic view that combines source code views and architectural views.

There have been several proposals for clone visualization. Scatter plots—also known as dot plots—are two-dimensional charts where all software units are listed on both axes [114, 156, 512] (cf. Fig. 2.3). There is a dot if two software units are similar. The granularity of software units may differ. It can range from single lines to functions to classes and files to packages and subsystems. Visual patterns of cloning may be observed by a human analyst. A problem with this approach is scalability for many software units and the order of the listed software units as this has an impact on the visual patterns. While there is a “natural” order for lines (i.e., lexical order) within a file, it is not clear how to order more coarse-grained units such as functions, files, and packages. Lexical order of their names is in most cases as arbitrary as random order.

Johnson [260] proposes Hasse diagrams for clone representation between sets of files so that one can better see whether code has been copied between files, which is possibly more critical than cloning within a file (cf. Fig. 2.4). A Hasse diagram (named after a German mathematician) is used to draw a partial order among sets as an acyclic graph. Directed arcs connect nodes that are related by the order relation and for which no other directed path exists.

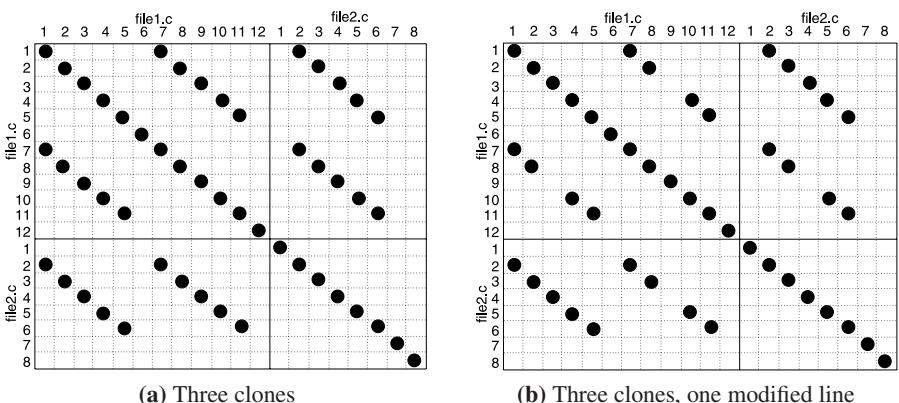


Fig. 2.3. Dot plots [114, 156, 512]

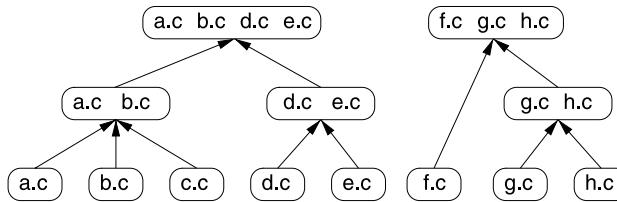


Fig. 2.4. Hasse diagram adapted from [260]

In Johnson's context, each match of a block of text identifies a range of characters (or lines) from two (or more) files. For each subset of files, one can total the number of characters that the matching process has discovered to match between the given set of files. A subset of files forms a node, if the files have non-zero matches. The inclusion between subsets of files yields the edges.

Rieger et al. [437] propose to use Michele Lanza's polymetric views[309] to visualize various aspects of clones in one view (cf. Fig. 2.5). A polymetric view is again based on the graph metaphor and representation where a node represents a software unit and an edge a cloning relation. Visually, additional information can be attached to the graph by the degrees of freedom for the position (X/Y in the two-dimensional space), color of nodes and edges, thickness of edges, breadth and width of nodes. Rieger et al. propose a fixed set of metric combinations to be mapped onto graphical aspects to present the clone information from different perspective for different tasks.

Beyond polymetric views, Rieger et al. [437] propose a variation of tree maps to show the degree of cloning along with the system decomposition (cf. Fig. 2.6). Tree maps display information about entities with a hierarchical relationship in a fixed space (for instance, the whole system on one screen) where the leaves of the hierarchy contain a metric to be visualized. Each inner node aggregates the metric values of its descendants. Each node is represented through a piece of the available space. The space of a descendent node is completely contained in the space of its ancestor.

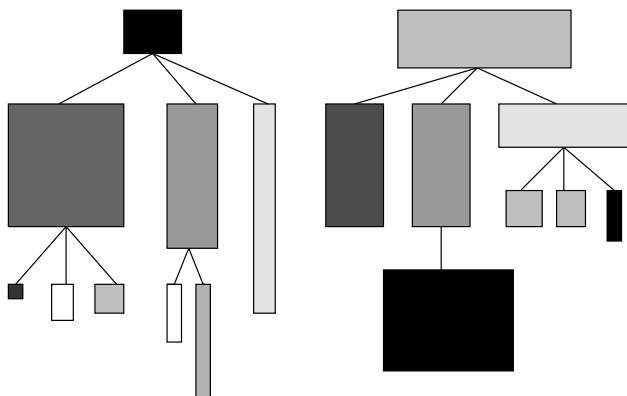


Fig. 2.5. Polymetric view adapted from [437]

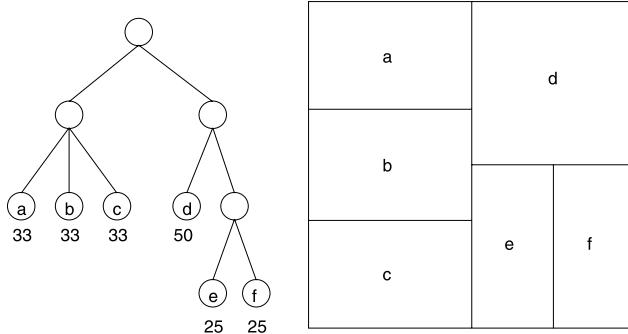


Fig. 2.6. A system decomposition whose leaves are annotated with the number of cloned lines of code and its corresponding tree map

There is no overlap in space for nodes that are not in an ancestor/descendant relation. This is how the hierarchy is presented. Essentially the hierarchy is projected onto the two dimensional space seen from the root of the hierarchy. In order to show the hierarchy clearly, the space of each node appears as rectangle where the direction of subdivision of nested nodes is alternated horizontally and vertically at each level. The space of each rectangle is proportional to the metric.

This visualization was originally proposed by Ben Shneiderman in the early 1990s to show space consumption of a hard disk with a hierarchical file system. While space is used very efficiently, problems arise when the hierarchy is deeply nested.

Another visualization was proposed by Wattenberg to highlight similar substrings in a string. The arc diagram has an arc connecting two equal substrings in a string where the breadth of the arc line covers all characters of the identical substrings. The diagram shows the overlapping of strings but becomes quickly unreadable if many arcs exist. Another disadvantage is that it shows only pairs but not classes of equal strings.

Tairas et al. [489] have created an Eclipse plugin to present clone information. One of their visualizations is the clone visualizer view, a window showing the distribution of clone classes among files (cf. Fig. 2.8). A bar in this view represents a source file, a stripe within a bar a cloned code segment, and its colors the set of clone classes the segment is member of.

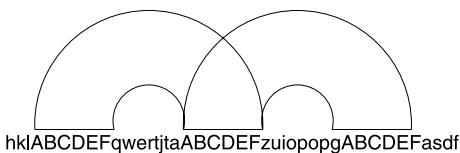


Fig. 2.7. Arc diagram adapted from [537]

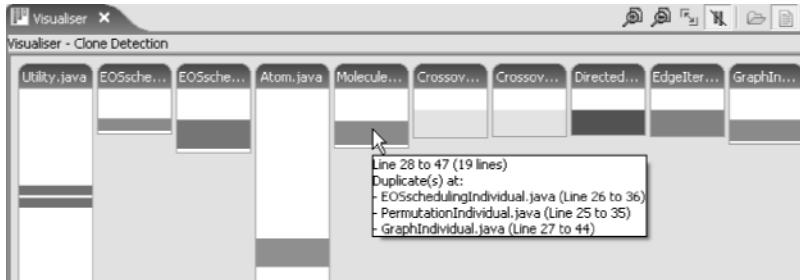


Fig. 2.8. Clones visualizer view in Eclipse adapted from [489]

2.11 Related Fields

Clone detection has applications in other fields and—vice versa—ideas from related fields can be reused for clone detection.

Bruntink et al. for instance, use clone detectors to search for code that could be factored out as aspects using an aspect-oriented language [92, 93]. They identify error handling, tracing, pre and post condition checking, and memory error handling. Although they used classic clone detectors that were not designed for this particular purpose, the clone detectors appeared to be helpful. Classic clone detectors try to find similar code—similar in terms of their program text. The implementations of an aspect, on the other hand, are often very heterogeneous and are similar only at a more semantic level. For instance, precondition checking tests each parameter of a function for certain criteria. At the implementation level, functions differ in the order and type of parameters so that checks are generally different in the program text.

The code compaction community tries to minimize the memory footprint of programs for small devices. They use very similar algorithms to identify redundant code that could be compressed [124].

The detection of plagiarism faces similar but even worse problems as clone detection [452, 420, 180, 343, 251, 337, 210]. In plagiarism cases, people try to camouflage their copy in order to make it more difficult to detect the plagiarism. In order to reuse classic clone detectors for plagiarism, we would need to reduce programs to a normal form for comparison. This normalization, on the other hand, could lead to false positives. Also in virus detection, code patterns significant for a particular hostile code need to be quickly identified in large code bases, where virus programmers try to vary the patterns.

Another application of clone detection is the comparison of versions or variants of software systems. While versions derive from each other, variants have a common ancestor. In both cases, they are very similar. In software evolution research, where information on software units is observed over time or versions, respectively, it is necessary to map the software entities of one version to those of the other version in order to carry over the information. This problem is called the *origin analysis* [508]. The same problem needs to be solved when two software variants are to be compared

or merged [237]. Relaying solely on names of these units for this analysis may be misleading if a refactoring like *renaming* has taken place [183]. Also, the refactoring *extract method* moves statements from one function to create a new function. Clone detection can help to establish a mapping between two versions or variants of a program. Several authors have used clone detection techniques or at least a code similarity measure to determine this mapping [509, 568, 206, 205, 524, 552, 553].

The difference of comparing versions or variants to detecting clones is that the task here is to map a code entity onto only one or at least a small set of candidates in the other system, the comparison is only between systems (clones within the same version or variant are irrelevant), cloning is the rule rather than the exception as the two versions or variants overlap to a very high degree, the focus is on the differences rather than the similarities, and the comparison should tolerate renaming and all refactorings that move entities around such as pull-up field, move method, etc.

2.12 Conclusions

This section summarizes the open issues of the subareas in software cloning presented in this chapter.

One fundamental issue is that there is no clear consensus on what is a software clone. We should develop a general notion of redundancy, similarity, and cloning, and then identify more task-oriented categorizations of clones. Other research areas have similar difficulties in defining their fundamental terms. For instance, the architecture community debates the notion of *architecture* and the community of object-oriented programming the notion of *object*. To some extent, these fundamental terms define the field. So it is important to clarify them. It is difficult, for instance, to create benchmarks to evaluate automatic clone detectors if it is unclear what we consider a clone. It were very helpful if we could establish a theory of redundancy similar to normal forms in databases.

Concerning types of clones, we should look at alternative categorizations of clones that make sense (e.g., semantics, origins, risks, etc.). On the empirical side of clone categorizations, we should gather the statistical distribution of clone types in practice and investigate whether there are correlations among apparently orthogonal categories. Studying which strategies of removal and avoidance, risks of removal, potential damages, root causes, and other factors are associated with these categories would be worthwhile, too.

Although the two empirical studies by Kim and Notkin as well as Kapser and Godfrey on the root causes and main drivers for code cloning are important first contributions, this area certainly requires more similar studies. Other potential reasons should be investigated, such as insufficient information on global change impact, badly organized reuse and development processes, questionable productivity measures (e.g., LOCs per day), time pressure, educational deficiencies, ignorance, or shortsightedness, intellectual challenges (e.g., generics), lack of professionalism/end-user programming by non experts, and organizational issues, e.g., distributed development and organizations.

Identifying the root causes would help us to fight the reasons, not just the symptoms, for instance, by giving feedback for programming language design.

Clearly, more empirical studies are required. These studies should take industrial systems into account, too, as it is unclear to which extent these current observations can be attributed to the nature of open-source development. It would also be interesting to investigate what the degree of cloning tells about the organization or development process. For instance, a study by Nickell and Smith reports that the extreme programming projects in their organization produce significantly fewer clones [394, 533].

In particular, empirical investigations of costs and benefits of clone removal are needed so that informed refactoring decisions can be made. We currently do not have a clear picture of the relation of clone types to quality attributes. Most of what we report on the consequences of cloning is folklore rather than fact. We should expect that there is a relevance ranking of clone types for removal, that is, some clones should be removed, others are better left in certain circumstances. Moreover, we can expect that different types of clones are associated with different removal techniques in turn associated with different benefits, costs, and risks.

Unwanted clones should be avoided right from the start. But it is not yet clear what is the best integration of clone detection in the normal development process. In particular, what are the benefits and costs of such possible integrations and what are reliable cloning indicators to trigger refactoring actions?

If it is too late to avoid cloning and if existing clones cannot be removed, we should come up with methods and tools to manage these clones. This clone management must stop further spread of clones and help to make changes consistently.

The most elaborated field in software cloning is the automatic detection of clones. Yet, there is still room for improvement as identified in the quantitative and qualitative comparisons. Most helpful would be a ranking function that allows to present clone candidates in an order of relevance. This ranking function can be based on measures such as type, frequency, and length of clones but should also take into account the task driving the clone detection.

Although various types of visualization to present clones have been proposed we have not fully explored all opportunities. There is a large body of research on information visualization in general and software visualization in particular that we have not yet explored for clone visualization. In order to understand which visualization works best for which purpose, we need more systematic empirical research. Clone representation is difficult due to the large and complex information space. We have various aspects that we need to master: the large amount of data, clone class membership, overlap and inclusion of clones, commonalities and differences among clones in the same class, degree of similarity, and other attributes such as length, type, frequency, and severity.

Clone detection overlaps with related fields, such as code compression or virus detection. The interesting questions here are “What can clone detection learn from other fields?” and “What can other fields learn from clone detection?”

CHAPTER TWENTY-THREE

Evidence-Based Failure Prediction

*Nachiappan Nagappan
Thomas Ball*

Empirical software engineering (SE) studies collect and analyze data from software artifacts and the associated processes and variables to quantify, characterize, and explore the relationship between different variables to deliver high-quality, secure software on time and within budget. In this chapter we discuss empirical studies related to failure prediction on the Windows operating system family. Windows is a large commercial software system implemented predominantly in C/C++/C# and currently used by several hundreds of millions of users across the world. It contains 40+ million lines of code and is worked on by several hundreds of engineers.

Failure prediction forms a crucial part of empirical SE, as it can be used to understand the maintenance effort required for testing and resource allocation. For example:

Resource allocation

Software quality assurance consumes a considerable effort in any large-scale software development. To raise the effectiveness of this effort, it is necessary to plan in advance for fixing issues in the components that are more likely to fail and thereby need quality assurance most.

Decision making

Predictions on the number of failures can support other decisions, such as choosing the correct requirements or design, the ability to select the best possible fix for a problem, etc. The associated risk of a change (or likelihood to fail) can help in making decisions about

the risk introduced by the change. Failure predictions also help in assessing the overall stability of the system and help make decisions about the ability to release on time.

We explore the step-by-step progression of applying various metrics to predict failures. For each set for metrics we discuss the rationale behind metric selection, a description of the metric, and the results of applying the metrics for actual failure prediction in a large commercial software system.

Introduction

Software organizations can benefit greatly from an early estimation regarding the quality of their product. Because product quality information is available late in the process, corrective actions tend to be expensive [Boehm 1981]. The IEEE standard [IEEE 1990] for software engineering terminology defines the waterfall software development cycle as “a model of the software development process in which the constituent activities, typically a concept phase, requirements phase, design phase, implementation phase, test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration.” During the development cycle, different metrics can be collected that can be related to product quality. The goal is to use such metrics to make estimates of post-release failures early in the software development cycle, during the implementation and testing phases. For example, such estimates can help focus testing and code and design reviews and affordably guide corrective actions.

The selection of metrics is dependent on using empirical techniques such as the G-Q-M principle [Basili et al. 1994] to objectively assess the importance of selecting the metrics for analysis. An *internal metric* (measure), such as cyclomatic complexity [McCabe 1976], is a measure derived from the product itself. An *external metric* is a measure of a product derived from the external assessment of the behavior of the system. For example, the number of failures found in the field is an external metric. The ISO/IEC standard [ISO/IEC 1996] states that an internal metric is of little value unless there is evidence that it is related to an externally visible attribute. Internal metrics have been shown to be useful as early indicators of externally visible product quality when they are related in a statistically significant and stable way to the field quality/reliability of the product. The validation of internal metrics requires a convincing demonstration that (1) the metric measures what it purports to measure and (2) the metric is associated with important external metrics such as field reliability, maintainability, or fault-proneness [El Emam 2000].

In this chapter we discuss six different sets of metrics for failure prediction. Each set of metrics is described, and its importance to failure prediction is illustrated. Results of published industrial case studies at Microsoft are provided with references for future reading. We then provide a summary of failure prediction and discuss future areas of importance. We discuss six different sets of internal metrics to predict failures:

1. Code coverage
2. Code churn
3. Code complexity
4. Code dependencies
5. People and organizational metrics
6. Integrated/combined approach

Executable binaries are the level of measurement we use in reference to the Windows case studies. Binaries are the result of compiling source files to form an *.exe*, *.dll*, or *.sys* file. Our choice of binaries was governed by the facts that: (1) binaries are the lowest level at which field failures are mapped back to; (2) fixes usually involve changes to several files, most of which are compiled into one binary; (3) binaries are the lowest level at which code ownership is maintained, thereby making our results more actionable. Historically, Microsoft has used binaries as the unit of measurement due to the ability to map back customer failures accurately.

For each of the six sets of metrics, we provide evidence from a case study at Microsoft predicting failures in Windows Vista and/or Windows Server 2003. For each of the predictions, we provide the related precision and recall values or accuracy values. *Precision* measures the false-negative rate, which is the ratio of failure-prone binaries that were classified as not-failure-prone. *Recall* measures the false-positive rate, which denotes the ratio of not-failure-prone binaries that were classified as failure-prone. We focus here on the metrics for failure prediction rather than on the statistical or machine-learning techniques for which standard techniques such as logistic regression, decision trees, support vector machines, etc., can be used [Han and Kamber 2006].

As with all research on empirical evidence on software engineering, drawing general conclusions from empirical studies in software engineering is difficult because any process depends to a large degree on a potentially large number of relevant context variables [Basili et al. 1999]. For this reason, we cannot assume a priori that the results of a study generalize beyond the specific environment in which it was conducted [Basili et al. 1999]. Researchers become more confident in a theory when similar findings emerge in different contexts [Basili et al. 1999]. We encourage readers to investigate and, if possible, replicate the studies in their specific environments to draw conclusions. We hope this chapter serves as a documentation of the metrics and feature sets that have been shown to be successful (or not) at predicting failures at Microsoft.

Code Coverage

Code coverage is an important metric by which the extent of testing is often quantified. The main assumption behind coverage is that if a branch or a statement contains a flaw, it cannot be detected unless a test at least executes that statement or branch (and obtains an output contrary to the normal expectation). It can be argued that higher coverage should lead to detection of more flaws in the code and, if they are fixed, to better release quality. Although

this assumption is widely believed, there is little evidence to show that higher code coverage results in fewer failures.

There are a number of potential flaws with the argument that higher code coverage leads to better quality. First, the coverage measure reflects the percent of statements covered but does not consider whether these statements are likely to contain a flaw. Therefore, it is possible to create two test sets with the same coverage measure but a markedly distinct ability to detect post-release defects. Second, the fact that a statement or a branch was executed does not imply that all possible data values have been exercised by the test. The customer/usage scenario could be totally different from what the tester perceives it to be. This is of particular concern for systems with simple control flow that can be easily covered with a few test cases. In our study on code coverage, based on qualitative interviews with engineers, the reasons for observing such a trend was attributed to several factors [Mockus et al. 2009]:

- Code covered is not correct code.
- Developers inherently know that certain binaries are complex and will tend to get very high coverage in those binaries. But these binaries may be the ones most used in the system, which can lead to more failures being found in them, i.e., code coverage and usage profiles might not match.
- Complexity should tie into code coverage. For example, obtaining a code coverage of 80% on a binary with cyclomatic complexity 1000 is more difficult than getting a coverage of 80% on a binary with cyclomatic complexity 10. The complexity values tied into the code coverage helps exploit the efficacy of the code coverage measures.

Despite these flaws, coverage appears to be a promising measure due to its wide use in practice. However, there has been little published about the relationship between code coverage and quality. Most studies performed to date on this relationship suffer from issues of internal validity (studies of 100-line programs). In order to find evidence of the relationship between code coverage and quality, we mapped the branch and block coverage values for Windows Vista (40+ million lines of code and several thousand binaries and engineers) and field failures for Windows Vista six months after release. We observe a weak positive correlation between coverage and quality and a low prediction precision and recall (precision 83.8% and recall 54.8%; for further reading, please refer to [Mockus et al. 2009]). Our investigation of this straightforward and simple hypothesis leads us to believe that code coverage is not best used in isolation, but instead needs to be combined with other metrics, such as code churn, complexity, etc.

Code Churn

Software systems evolve over time due to changes in requirements, optimization of code, fixes for security and reliability bugs, etc. Code churn measures the changes made to a component over a period of time and quantifies the extent of this change. It is easily extracted from a

system's change history, as recorded automatically by a version control system. Most version control systems use a file comparison utility (such as *diff*) to automatically estimate how many lines were added, deleted, and changed by a programmer to create a new version of a file from an old version. These differences are the basis of churn measures.

Relative churn measures are normalized values of the various churn measures. Some of the normalization parameters are total lines of code, file churn, file count, etc. In an evolving system it is highly beneficial to use a relative approach to quantify the change in a system. As we show, these relative measures can be devised to cross-check each other so that the metrics do not provide conflicting information. Our basic hypothesis is that code that changes many times pre-release will likely have more post-release defects than code that changes less over the same period of time.

In our analysis, we used the code churn between the release of Windows Server 2003 (W2k3) and the release of the W2k3 Service Pack 1 (W2k3-SP1) to predict the defect density in W2k3-SP1. Using the directly collected churn metrics such as added, modified, and deleted lines of code (LOC), we also collected the time and file count measures to compute a set of relative churn measures [Nagappan and Ball 2005]:

METRIC1: Churned LOC / Total LOC

We expect the larger the proportion of churned (added + changed) code to the LOC of the new binary, the larger the magnitude of the defect density for that binary.

METRIC2: Deleted LOC / Total LOC

We expect the larger the proportion of deleted code to the LOC of the new binary, the larger the magnitude of the defect density for that binary.

METRIC3: Files churned / File count

We expect the greater the proportion of files in a binary that get churned, the greater the probability of these files introducing defects. For example, suppose binaries A and B contain 20 files each. If binary A has five churned files and binary B has two churned files, we expect binary A to have a higher defect density.

METRIC4: Churn count / Files churned

Suppose binaries A and B have twenty files each and also have five churned files each. If the five files in binary A are churned 20 times and the five files in binary B are churned 10 times, then we expect binary A to have a higher defect density. METRIC4 acts as a cross check on METRIC3.

METRIC5: Weeks of churn / File count

METRIC5 is used to account for the temporal extent of churn. A higher value of METRIC5 indicates that it took a longer time to fix a smaller number of files. This may indicate that the binary contains complex files that may be hard to modify correctly. Thus, we expect that an increase in METRIC5 would be accompanied by an increase in the defect density of the related binary.

METRIC6: Lines worked on / Weeks of churn

The measure “Lines worked on” is the sum of the churned LOC and the deleted LOC. METRIC6 measures the extent of code churn over time in order to cross check on METRIC5. Weeks of churn does not necessarily indicate the amount of churn. METRIC6 reflects our expectation that the more lines are worked on, the longer the weeks of churn should be. A high value of METRIC6 cross checks on METRIC5 and should predict a higher defect density.

METRIC7: Churned LOC / Deleted LOC

METRIC7 is used in order to quantify new development. All churn is not due to bug fixes. In feature development the lines churned is much greater than the lines deleted, so a high value of METRIC7 indicates new feature development. METRIC7 acts as a cross check on METRIC1 and METRIC2, neither of which accurately predicts new feature development.

METRIC8: Lines worked on / Churn count

We expect that the larger a change (lines worked on) relative to the number of changes (churn count), the greater the defect density will be. METRIC8 acts as a cross check on METRIC3 and METRIC4, as well as METRIC5 and METRIC6. With respect to METRIC3 and METRIC4, METRIC8 measures the amount of actual change that took place. METRIC8 cross checks to account for the fact that files are not getting churned repeatedly for small fixes. METRIC8 also cross checks on METRIC5 and METRIC6 to account for the fact that the higher the value of METRIC8 (more lines per churn), the higher the time (METRIC5) and lines worked on per week (METRIC6). If this is not so, then a large amount of churn might have been performed in a small amount of time, which can cause an increased defect density.

Using the relative code churn metrics identified about a prediction model identifies with random splitting an accuracy of 89% the failure-prone and non-failure-prone binaries in W2K3-SP1. Further, the defect density of W2k3-SP1 is also predicted with a high degree of accuracy (shown in [Figure 23-1](#)). (The axes are normalized to protect confidential data). The spikes above the solid line indicate overestimations and below the solid line indicate underestimations. The correlation between actual and estimated defect density is strong, positive, and statistically significant—indicating that with the increase in actual defect density there is an increase in estimated defect density.

These results, along with results from prior published research studies (for example, [\[Ostrand et al. 2004\]](#) and [\[Nagappan and Ball 2005\]](#) have a more detailed list), indicate that code churn and code quality are highly and positively correlated. That is, the higher the code churn, the more the failures in a system. Because code churn is an essential part of new software development (say, new product features), we introduced the relative code churn measures to alleviate this problem by cross-checking the metrics. However, our results show that it is clear that code churn measures need to be used in conjunction with other internal metrics.

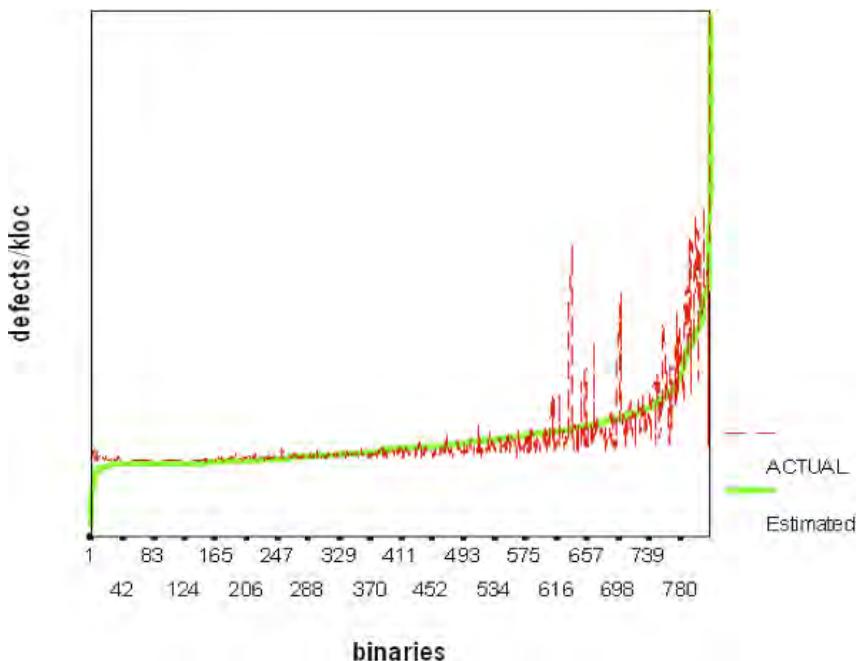


FIGURE 23-1. Actual versus estimated system defect density [Na^gappan and Ball 2005]

Code Complexity

Following code churn, we analyzed code complexity. That is, how does the complexity of a piece of code influence the ability to predict failures? Complexity can be quantified using several metrics, more suited toward object-oriented and non-object-oriented metrics ranging from the more traditional metrics such as fan-in and fan-out to the more recent CK metrics [Chidamber and Kemerer 1994]. The CK metric suite consists of six metrics (designed primarily as object-oriented design measures): weighted methods per class (WMC), coupling between objects (CBO), depth of inheritance (DIT), number of children (NOC), response for a class (RFC), and lack of cohesion among methods (LCOM). The typical object-oriented and non-object-oriented complexity metrics used at Microsoft based on prior published metrics (for example, [Chidamber and Kemerer 1994]) are:

Lines

Executable noncommented lines of code.

Cyclomatic complexity

The Cyclomatic complexity metric [McCabe 1976] measures the number of linearly independent paths through a program unit.

Fan-in

Fan-in is the number of other functions calling a given function in a module.

Fan-out

Fan-out is the number of other functions being called from a given function in a module.

Methods

Number of methods in a class, including public, private, and protected methods.

Inheritance depth

Inheritance depth is the maximum depth of inheritance for a given class.

Coupling

This signifies coupling to other classes through (a) class member variables, (b) function parameters, and (c) classes defined locally in class member function bodies, as well as (d) coupling through immediate base classes and (e) coupling through return type.

Subclasses

This metric indicates the number of classes directly inheriting from a given parent class in a module.

The overall efficacy of using these metrics for predicting failures is with a precision of 79.3% and a recall of 66% in Windows Vista. The measures indicate that though complexity measures are fair predictors of code quality, they are not as good as code churn. In related work, Khoshgoftaar et al. studied two consecutive releases of a large legacy system (containing over 38,000 procedures in 171 modules) for telecommunications and identified fault-prone modules based on 16 static software product metrics [Khoshgoftaar et al. 1996]. Their model, when used on the second release, showed an overall misclassification rate of 21.0%. El Emam et al. studied the effect of class size on fault-proneness by using a large telecommunications application [El Emam et al. 2001]. Class size was found to confound the effect of all the metrics on fault-proneness. The CK metrics have also been investigated in the context of fault-proneness. Basili et al. studied the fault-proneness in software programs using eight student projects [Basili et al. 1996]. They observed that the WMC, CBO, DIT, NOC, and RFC were correlated with defects, whereas the LCOM was not correlated with defects. Briand et al. performed an industrial case study and observed the CBO, RFC, and LCOM to be associated with the fault-proneness of a class [Briand et al. 1999]. Tang et al. studied three real-time systems for testing and maintenance defects [Tang et al. 1999]. Higher WMC and RFC were found to be associated with fault-proneness. For further reading, refer to [Nagappan et al. 2006a], [Briand et al. 1999], and [Bhat and Nagappan 2006].

Code Dependencies

We have seen that code churn and complexity are good predictors of failures. This led us to look at tying together these two pieces of information to effectively quantify the relationship between different pieces of code. In general, in any large-scale software development effort, a good software architecture enables teams to work independently on different components in

the architecture. A software dependency is a relationship between two pieces of code, such as a data dependency (component A uses a variable defined by component B) or call dependency (component A calls a function defined by component B). Suppose that component A has many dependencies on component B. If the code of component B changes (churns) a lot between versions, we may expect that component A will need to undergo a certain amount of churn in order to keep it in sync with component B. That is, churn often will propagate across dependencies. Together, a high degree of dependence plus churn can cause errors that will propagate through a system, reducing its reliability.

In order to study the effectiveness of dependencies to predict failures, we studied the dependencies between binaries in Windows Vista. The dependencies are computed at the function level, including caller-callee dependencies, imports, exports, RPC, COM, Registry access, etc. The system-wide dependency graph can be viewed as the low-level architecture of Windows Vista. Dependencies are classified into incoming and outgoing dependencies with the binaries. Further classifications include the total number of dependencies between binaries and the span of binaries across different components. The prediction models built using the dependency metrics to predict failures in Windows Vista have a precision and recall of 74.4% and 69.9%, respectively. This is as good as the precision and recall results obtained from code complexity metrics but does not compare to the results obtained from code churn. Further reading on the ability of dependencies and code churn to predict failures on the Microsoft case study can be found in [\[Nagappan and Ball 2007\]](#).

In related work, Schröter et al. showed that import dependencies can predict defects [\[Schröter et al. 2006\]](#). They proposed an alternate way of predicting failures for Java classes. Rather than looking at the complexity of a class, they looked exclusively at the components that a class uses. For Eclipse, an open source IDE, they found that using compiler packages results in a significantly higher failure-proneness (71%) than using GUI packages (14%). Von Mayrhauser et al. investigated the relationship of the decay of software architectures with faults using a bottom-up approach of constructing a fault-architecture model [\[von Mayrhauser et al. 1999\]](#). A fault-architecture model was constructed incorporating the degree of fault-coupling between components and how often these two components are involved in a defect fix. Their results indicated the most fault-prone relationships for each release and showed that the same relationships between components are repeatedly fault-prone, indicating underlying architectural problems.

People and Organizational Measures

Conway's Law states that "organizations that design systems are constrained to produce systems which are copies of the communication structures of these organizations" [\[Conway 1968\]](#). Similarly, Fred Brooks argues in *The Mythical Man-Month* that the product quality is strongly affected by organizational structure [\[Brooks 1995\]](#). Software engineering is a complex engineering activity. It involves interactions between people, processes, and tools to develop

a complete product. In practice, commercial software development is performed by teams consisting of a number of individuals, ranging from the tens to the thousands. Often these people work via an organizational structure reporting to a manager or set of managers. Often failure predictions are obtained from measures such as code churn, code complexity, code coverage, code dependencies, etc. But these studies often ignore one of the most influential factors in software development: specifically, “people and organizational structure.”

With the advent of global software development where teams are distributed across the world, the impact of organization structure on Conway’s Law and its implications on quality are significant. In this section, we investigate the relationship between organizational structure and software quality via a set of eight measures that quantify organizational complexity [Nagappan et al. 2008]. For the organizational metrics, the metrics capture issues such as organizational distance of the developers; the number of developers working on a component; the amount of multitasking developers are doing across organizations; and the amount of change to a component within the context of that organization. Using these measures, we predict failure-prone binaries in Windows Vista. Some of the organizational measures are the following (for a more detailed list, refer to [Nagappan et al. 2008]):

Number of Engineers (NOE)

This is the absolute number of unique engineers who have touched a binary and are still employed by the company.

Number of Ex-Engineers (NOEE)

This is the total number of unique engineers who have touched a binary and have left the organizations as of the release date of the software system.

Edit Frequency (EF)

This is the total number times the source code that makes up the binary was edited. An edit is when an engineer checks code out of the version control system, alters it, and checks it back in again. This is independent of the number of lines of code altered during the edit.

Depth of Master Ownership (DMO)

This metric determines the level of ownership of the binary depending on the number of edits done. The organizational level of the person whose reporting engineers perform more than X% of the rolled-up edits is deemed as the DMO. The DMO metric determines the binary owner based on activity on that binary. We used 75% as the X%, based on prior historical information on Windows to quantify ownership.

Percentage of Org contributing to development (PO)

The ratio of the number of people reporting to the DMO-level owner relative to the DMO overall org size.

Level of Organizational Code Ownership (OCO)

The percent of edits from the organization that contains the binary owner, or if there is no owner, then the organization that made the majority of the edits to that binary.

Overall Organization Ownership (OOW)

This is the ratio of the percentage of people at the DMO level making edits to a binary relative to total engineers editing the binary. A high value is desired.

Organization Intersection Factor (OIF)

A measure of the number of different organizations that contribute greater than 10% of edits, as measured at the level of the overall org owners.

The measures proposed here attempt to balance the various hypotheses about how organizational structure can influence the quality of the binary, some of which seem to represent opposing positions. A high-level summary of the hypotheses and the measures that purport to quantify these hypotheses is presented in [Table 23-1](#).

TABLE 23-1. Summary of organizational measures [Nagappan et al. 2008]

Hypothesis	Metric
The more people who touch the code, the lower the quality.	NOE
A large loss of team members affects the knowledge retention and thus quality.	NOEE
The more edits to components, the higher the instability and lower the quality.	EF
The lower the level of ownership, the better the quality.	DMO
The more cohesive are the contributors (organizationally), the higher the quality.	PO
The more cohesive the contributions (edits), the higher the quality.	OCO
The more diffused the contribution to a binary, the lower the quality.	OOW
The more diffused the different organizations contributing code, the lower the quality.	OIF

Using random splitting on the entire Vista code base for 50 random splits, we obtained the precision and recall values as shown in [Figure 23-2](#). The figure indicates the uniformity and consistency of the predictions of precision and recall. The values show little variance, explaining the overall efficacy of the models built and the ability of organizational metrics to effectively act as predictors of software quality. The precision and recall values were 86.2% and 84%, respectively. This is, as from our prior studies, the highest precision and recall in terms of predicting failures. This provided evidence that understanding the team structure in the organization is a crucial factor in predicting and understanding software quality. The interpretation lies in the fact that the organizational metrics were much better indicators of code quality, more than actual code metrics. This stresses the importance of getting the organization and team right for doing software development. These results also illustrate the impact of organizational structure on quality and the importance of planning for reorganization of teams to minimize the impact on software quality.

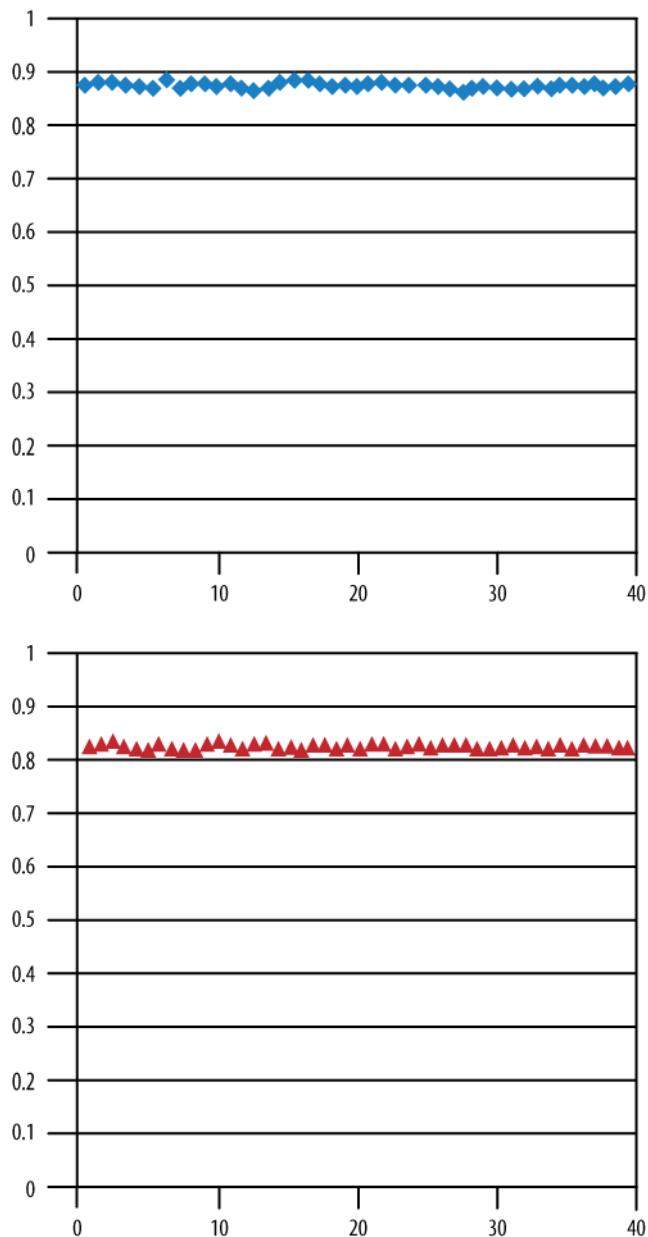


FIGURE 23-2. Precision and recall values for predictions using organization structure [Nagappan et al. 2008]

Integrated Approach for Prediction of Failures

So far we have analyzed sets of metrics—complexity metrics, code churn, code coverage, etc.—in isolation. In this section we address how the metrics may be combined to form stronger predictors of failures.

In [Figure 23-3](#) a simple network of engineering working together in different binaries is shown. Similarly, [Figure 23-3](#) shows the code dependencies between various networks. [Figure 23-3](#) shows combining both pieces of information to integrate people, churn (in terms of edits/contributions), and dependencies together into one network.

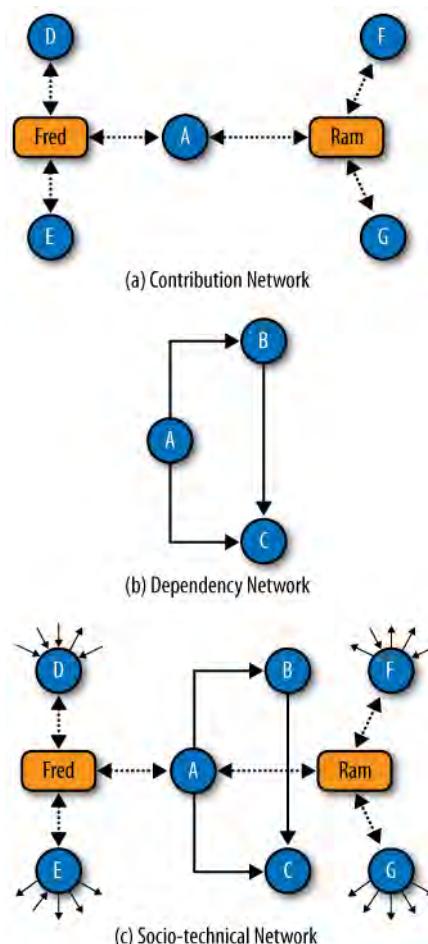


FIGURE 23-3. Socio-technical networks [[Bird et al. 2009](#)]

For Windows Vista we generate such a network integrating the people, churn contribution, and dependency information. Several social-network measures [Bird et al. 2009], detailed next, are computed for the Windows Vista social network (similar to the network in Figure 23-3).

Ego network measures [Borgatti et al. 2002] are based on the neighborhood for any particular node. The node being evaluated is denoted ego, and the neighborhood includes ego, the set of nodes connected to an ego, and the complete set of edges between this set of nodes.

Size

The number of nodes in the ego network

Ties

Number of edges in the ego network

Pairs

Number of possible directed edges in the ego network

Density

Proportion of possible ties that actually are present (Ties/Pairs)

Weak Components

Number of weakly connected components

Normalized Weak Components

Number of weakly connected components normalized by size, i.e., (Weak Components/Size)

Two Step Reach

The proportion of nodes that are within two hops of ego

Reach Efficiency

Two Step Reach normalized by size of the network (higher reach efficiency indicates that the ego's primary contacts are influential in the network)

Brokerage

Number of pairs of nodes that are connected only by ego (thus ego acts as the sole broker for the pair)

Normalized Brokerage

Brokerage normalized by number of pairs

Ego Betweenness

Betweenness of ego within its ego network

The preceding social network measures are computed for the complete Vista network (which includes the developers, contributions, and dependencies). Using these social network measures as input, prediction models are built. We observe that precision and recall of the built models are much higher when also using the dependency network for prediction. Similar results were also observed for multiple versions of IBM Eclipse (the open source IDE from IBM) [Bird et al. 2009]. Table 23-2 shows the precision and recall values with the model fit F-scores,

which also indicate the increased ability of the socio-technical approach to predict failures. The “combined” model in [Table 23-2](#) denotes a model purely built by just adding both the “Contribution” network (people working together) and the “Dependency” network (between various pieces of code), which provides a contrast to the socio-technical network. Further readings in which code complexity, churn, and coverage metrics are combined to predict failures can be found in [\[Nagappan et al. 2006b\]](#).

TABLE 23-2. Overall socio-technical network model efficacy using different release of Eclipse [Bird et al. 2009]

Release	Network	Precision	Recall	F-score	Nagel.
2.0	Dependency	0.667	0.779	0.705	0.532
	Contribution	0.808	0.854	0.824	0.702
	Combined	0.826	0.814	0.813	0.909
	Socio-technical	0.755	0.859	0.800	0.747
2.1	Dependency	0.693	0.753	0.710	0.626
	Contribution	0.675	0.780	0.719	0.607
	Combined	0.755	0.777	0.758	0.805
	Socio-technical	0.747	0.809	0.770	0.689
3.0	Dependency	0.631	0.737	0.673	0.494
	Contribution	0.681	0.683	0.673	0.353
	Combined	0.745	0.756	0.743	0.616
	Socio-technical	0.767	0.777	0.769	0.600
3.1	Dependency	0.579	0.718	0.634	0.391
	Contribution	0.639	0.646	0.629	0.295
	Combined	0.693	0.796	0.735	0.689
	Socio-technical	0.820	0.800	0.806	0.668
3.2	Dependency	0.698	0.780	0.731	0.495
	Contribution	0.614	0.720	0.654	0.371
	Combined	0.835	0.866	0.846	0.816
	Socio-technical	0.793	0.784	0.785	0.572
3.3	Dependency	0.693	0.743	0.711	0.433
	Contribution	0.725	0.669	0.688	0.356
	Combined	0.742	0.780	0.754	0.686
	Socio-technical	0.820	0.831	0.823	0.727

Summary

In the preceding sections we have described various metrics that have been used to predict failures from the Windows experience. To summarize, [Table 23-3](#) summarizes the precision and recall of using these sets of metrics to predict failures in Windows Vista.

TABLE 23-3. Overall model accuracy using different software measures

Model	Precision	Recall
Organizational structure	86.2%	84.0%
Code churn	78.6%	79.9%
Code complexity	79.3%	66.0%
Social network/combination measures	76.9%	70.5%
Dependencies	74.4%	69.9%
Code coverage	83.8%	54.4%

In addition, social network measures developed by integrating several of the measures also provides accurate predictions of failures. As stated earlier, as with all empirical studies, the evidence provided here is based only upon empirical results obtained at Microsoft, and more specifically in Windows. It is possible that these results might not hold for other software products or environments. We become more confident in results when similar studies are done in different software domains to show or disprove generality of the evidence provided for failure prediction. A step-by-step guide to building predictors is illustrated in the sidebar “Building Quality Predictors: A Step-by-Step Guide” [Nagappan et al. 2006a]. The primary purpose and goal of this chapter is to share experiences of results learned at Microsoft using various metrics and to describe the metrics, to encourage repetition in different domains by researchers and software engineers and thereby determine the efficacy of the applied metrics for other projects on a case-by-case basis.

BUILDING QUALITY PREDICTORS: A STEP-BY-STEP GUIDE

1. Determine a software E from which to learn. E can be an earlier release of the software at hand or a similar project.
2. Decompose E into entities (subsystems, modules, files, classes...) $E = \{e_1, e_2, \dots\}$ for which you can determine the individual quality (for example, binaries/modules).
3. Build a function $quality: E \rightarrow R$, which assigns to each entity $e \in E$ a quality. This typically requires mining version and bug histories.

4. Have a set of *metric functions* $M = \{m_1, m_2, \dots\}$ such that each $m \in M$ is a mapping $m: E \rightarrow \mathbb{R}$, which assigns a metric to an entity $e \in E$. The set of metrics M should be adapted for the project and programming language at hand.
 5. For each metric $m \in M$ and each entity $e \in E$, determine $m(e)$.
 6. Determine the correlations between all $m(e)$ and $\text{defects}(e)$, as well as the inter-correlations between all $m(e)$.
 7. If needed, using principal component analysis, extract a set of principal components $PC = \{pc_1, pc_2, \dots\}$, where each component $pc_i \in PC$ has the form $pc_i = \langle c_1, c_2, \dots, c_{|M|} \rangle$.
 8. You can now use the principal components PC to build a predictor for new entities $E' = \{e'_1, e'_2, \dots\}$ with $E' \cap E = \emptyset$. Be sure to evaluate the explanatory and predictive power.
-

The collection of the metrics discussed in this chapter is fully automated and most users can automatically collect these metrics for their projects using tools available in commercial and open source IDEs. Example screenshots from the Microsoft Visual Studio™ and IBM Eclipse plug-ins are shown in Figures 23-4* and 23-5.†

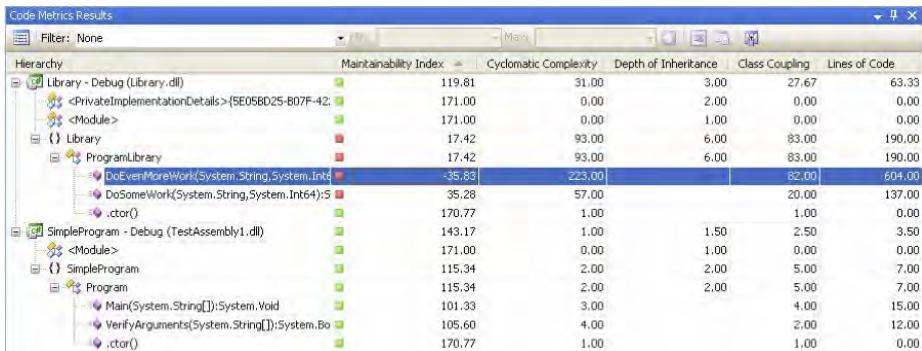


FIGURE 23-4. Metrics report in Microsoft Visual Studio™

* From: <http://blogs.msdn.com/b/codeanalysis/archive/2007/02/28/announcing-visual-studio-code-metrics.aspx>. Retrieved on June 13, 2010.

† From: <http://metrics.sourceforge.net/>. Retrieved on June 13, 2010.

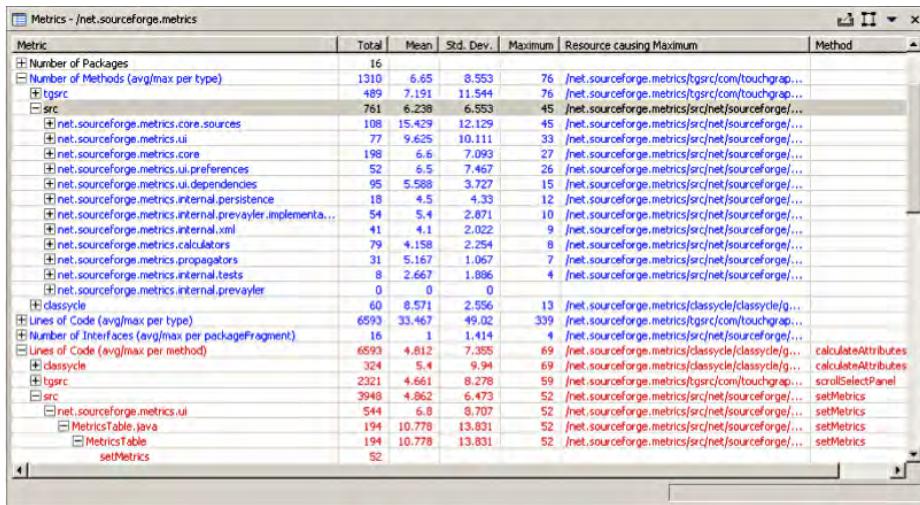


FIGURE 23-5. Metrics plug in for IBM Eclipse

Acknowledgments

We would like to thank all our colleagues at Microsoft Corporation and Windows for their help with these studies. We would also like to thank all our coauthors, more specifically, Thomas Zimmermann, Brendan Murphy, Vic Basili, Andreas Zeller, Audris Mockus, Prem Devanbu, and Chris Bird.

References

- [Basili et al. 1994] Basili, V., G. Caldiera, and D.H. Rombach. 1994. The Goal Question Metric Paradigm. In *Encyclopedia of Software Engineering*, ed. J. Marciniak, 528–532. Hoboken, NJ: John Wiley & Sons, Inc.
- [Basili et al. 1996] Basili, V., L. Briand, and W. Melo. 1996. A Validation of Object Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering* 22(10): 751–761.
- [Basili et al. 1999] Basili, V., F. Shull, and F. Lanubile. 1999. Building Knowledge through Families of Experiments. *IEEE Transactions on Software Engineering* 25(4): 456–473.
- [Bhat and Nagappan 2006] Bhat, T., and N. Nagappan. 2006. Building Scalable Failure-proneness Models Using Complexity Metrics for Large Scale Software Systems. *Proceedings of the XIII Asia Pacific Software Engineering Conference*: 361–366.

- [Bird et al. 2009] Bird, C., et al. 2009. Putting It All Together: Using Socio-technical Networks to Predict Failures. *Proceedings of the 20th IEEE international conference on software reliability engineering*: 109–119.
- [Boehm 1981] Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc.
- [Borgatti et al. 2002] Borgatti, S., M.G. Everett, and L.C. Freeman. 2002. *UCINET 6 for Windows: Software for Social Network Analysis*. Harvard, MA: Analytic Technologies.
- [Briand et al. 1999] Briand, L.C., J. Wuest, S. Ikonomovski, and H. Lounis. 1999. Investigating quality factors in object-oriented designs: an industrial case study. *Proceedings of the 21st international conference on software engineering*: 345–354.
- [Brooks 1995] Brooks, F.P. 1995. *The Mythical Man-Month*, Anniversary Edition. Boston: Addison-Wesley.
- [Chidamber and Kemerer 1994] Chidamber, S.R., and C.F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20(6): 476–493.
- [Conway 1968] Conway, M.E. 1968. How Do Committees Invent? *Datamation* 14(4): 28–31.
- [El Emam 2000] El Emam, K. 2000. *A Methodology for Validating Software Product Metrics*. Ottawa, Ontario, Canada: National Research Council of Canada.
- [El Emam et al. 2001] El Emam, K., S. Benlarbi, N. Goel, and S.N. Rai. 2001. The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering* 27(7): 630–650.
- [Han and Kamber 2006] Han, J., and M. Kamber. 2006. *Data Mining: Concepts and Techniques*, Second Edition. San Francisco: Elsevier.
- [Herbsleb and Grinter 1999] Herbsleb, J.D., and R.E. Grinter. 1999. Architectures, coordination, and distance: Conway’s Law and beyond. *IEEE Software* 16(5): 63–70.
- [IEEE 1990] IEEE. 1990. *IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. New York: The Institute of Electrical and Electronics Engineers.
- [ISO/IEC 1996] ISO/IEC. 1996. *DIS 14598-1 Information Technology—Software Product Evaluation*. International Organization for Standardization.
- [Khoshgoftaar et al. 1996] Khoshgoftaar, T.M., E.B. Allen, N. Goel, A. Nandi, and J. McMullan. 1996. Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System. *Proceedings of the Seventh International Symposium on Software Reliability Engineering*: 364.
- [McCabe 1976] McCabe, T.J. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2(4): 308–320.
- [Mockus et al. 2009] Mockus, A., N. Nagappan, and T.T. Dinh-Trong. 2009. Test Coverage and Post-verification Defects: A Multiple Case Study. *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*: 291–301.

- [Nagappan and Ball 2005] Nagappan, N., and T. Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the 27th international conference on software engineering*: 284–292.
- [Nagappan and Ball 2007] Nagappan, N., and T. Ball. 2007. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*: 364–373.
- [Nagappan et al. 2006a] Nagappan, N., T. Ball, and A. Zeller. 2006. Mining metrics to predict component failures. *Proceedings of the 28th international conference on software engineering*: 452–461.
- [Nagappan et al. 2006b] Nagappan, N., T. Ball, and B. Murphy. 2006. Using Historical In-Process and Product Metrics for Early Estimation of Software Failures. *Proceedings of the 17th International Symposium on Software Reliability Engineering*: 62–74.
- [Nagappan et al. 2008] Nagappan, N., B. Murphy, and V. Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. *Proceedings of the 30th international conference on software engineering*: 521–530.
- [Ostrand et al. 2004] Ostrand, T.J., E.J. Weyuker, and R.M. Bell. 2004. Where the Bugs Are. *Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and analysis*: 86–96.
- [Schröter et al. 2006] Schröter, A., T. Zimmermann, and A. Zeller. 2006. Predicting Component Failures at Design Time. *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering*: 18–27.
- [Tang et al. 1999] Tang, M.-H., M.-H. Kao, and M.-H. Chen. 1999. An empirical study on object-oriented metrics. *Proceedings of the 6th International Symposium on Software Metrics*: 242.
- [von Mayrhauser et al. 1999] von Mayrhauser, A., J. Wang, M.C. Ohlsson, and C. Wohlin. 1999. Deriving a fault architecture from defect history. *Proceedings of the 10th International Symposium on Software Reliability Engineering*: 295.

Chapter 7

Mining Source Code for Snippet Reuse



7.1 Overview

As already mentioned in the previous chapters, modern software development practices rely more and more on reuse. Systems are built using components found in software libraries and integrating them by means of small source code fragments, called *snippets*. As such, a large part of the development effort is spent on finding the proper snippets to perform the different envisioned tasks (e.g., read a CSV file, send a file over ftp, etc.) and integrate them into one's own source code. Using typical tools, such as search engines and programming forums for this task is far from optimal, as it requires leaving one's IDE to navigate through several online pages, in an attempt to comprehend the different ways to solve the problem at hand before selecting and integrating an implementation.

As a result, several methodologies have been proposed to address this challenge, most of which focus on the problems of *API usage mining* and *snippet mining*. API usage mining systems extract and present examples for specific library APIs [1–6]. Though effective, these systems are only focused on finding out how to use an API, without providing solutions in generic cases or in cases when determining which library to use is part of the question. Furthermore, several of them [1–3] return API call sequences instead of ready-to-use snippets.

On the other hand, generic snippet mining systems [7–9] employ code indexing mechanisms and thus include snippets for many different queries. Nevertheless, they also have important limitations. Concerning systems with local indexes [7], the quality and the diversity of their results are usually confined by the size of the index. Moreover, the retrieved snippets for all systems [7–9] are presented in the form of lists that do not allow easily distinguishing among different implementations (e.g., using different libraries to perform file management). The quality and the reusability of the results are also usually not evaluated. Finally, a common limitation in certain systems is that they involve some specialized query language, which may require additional effort by the developer.

In this chapter, we design and develop *CodeCatch*, a system that receives queries in natural language, and employs the Google search engine to extract useful snippets from multiple online sources. Our system further evaluates the readability of the retrieved snippets, as well as their preference/acceptance by the developer community using information from online repositories. Moreover, *CodeCatch* performs clustering to group snippets according to their API calls, allowing the developer to first select the desired API implementation and subsequently choose which snippet to use.

7.2 State of the Art on Snippet and API Mining

As already mentioned, in this chapter we focus on systems that receive queries for solving specific programming tasks and recommend source code snippets suitable for reuse in the developer's source code. Some of the first systems of this kind, such as *Prospector* [10] or *PARSEWeb* [11], focused on the problem of finding a path between an input and an output object in source code. For *Prospector* [10], such paths are called jungloids and the resulting program flow is called a jungloid graph. The tool is quite effective for certain reuse scenarios and can also generate code. However, it requires maintaining a local database, which may easily become deprecated. A rather more broad solution was offered by *PARSEWeb* [11], which employed the Google Code Search Engine¹ and thus the resulting snippets were always up-to-date. Both systems, however, consider that the developer knows exactly which API objects to use, and he/she is only concerned with integrating them.

Another category of systems is those that generate API usage examples in the form of call sequences by mining client code (i.e., code using the API under analysis). One such system is *MAPO* [1], which employs *frequent sequence mining* in order to identify common usage patterns. As noted, however, by Wang et al. [2], *MAPO* does not account for the diversity of usage patterns, and thus outputs a large number of API examples, many of which are redundant. To improve on this aspect, the authors propose *UP-Miner* [2], a system that aims to achieve high coverage and succinctness. *UP-Miner* models client source code using graphs and mines frequent closed API call paths/sequences using the *BIDE* algorithm [12]. A screenshot of *UP-Miner* is shown in Fig. 7.1. The query in this case concerned the API call `SQLConnection.Open`, and, as one may see in this figure, the results include different scenarios, e.g., for reading (pattern 252) or for executing transactions (pattern 54), while the pattern is also presented graphically at the right part of the screen.

PAM [3] is another similar system that employs probabilistic machine learning to extract API call sequences, which are proven to be more representative than those of *MAPO* and *UP-Miner*. An interesting novelty of *PAM* [3] is the use of an automated evaluation framework based on handwritten usage examples by the developers of

¹ As already mentioned in the previous chapters, the Google Code Search Engine resided in <http://www.google.com/codesearch>, however, the service was discontinued in 2013.

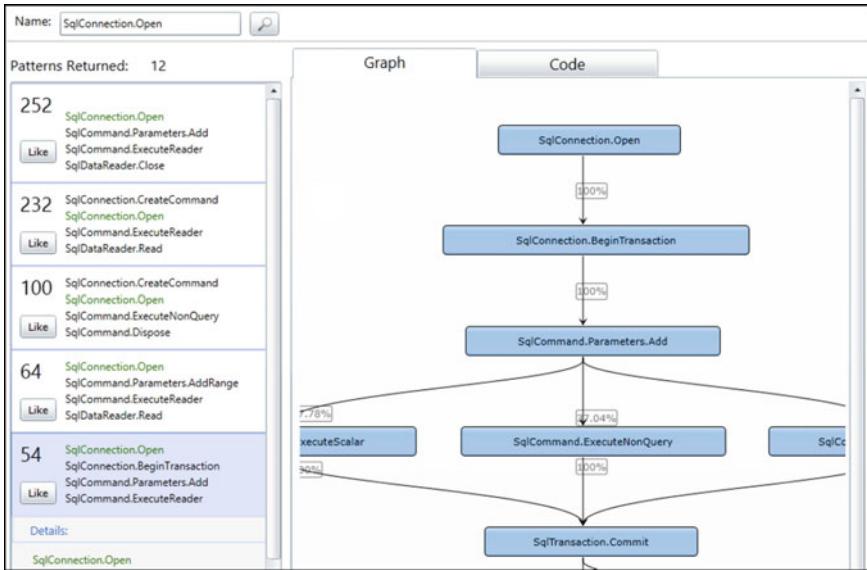


Fig. 7.1 Screenshot of UP-Miner [2]

the API under analysis. CLAMS [13] also employs the same evaluation process to illustrate how effective clustering can lead to patterns that are precise and at the same time diverse. The system also employs summarization techniques to produce snippets that are more readable and, thus, more easily reusable by the developer.

Apart from the aforementioned systems, which extract API call sequences, there are also approaches that recommend ready-to-use snippets. Indicatively, we refer to APIMiner [4], a system that performs code slicing to isolate useful API-relevant statements of snippets. Buse and Weimer [5] further employ path-sensitive data flow analysis and pattern abstraction techniques to provide more abstract snippets. Another important advantage of their implementation is that it employs clustering to group the resulting snippets into categories. A similar system in this aspect is eXoaDocs [6], as it also clusters snippets, however, using a set of semantic features proposed by the DECKARD code clone detection algorithm [14]. The system provides example snippets using the methods of an API and allows also receiving feedback from the developer. An example screenshot of eXoaDocs is shown in Fig. 7.2.

Though interesting, all of the aforementioned approaches provide usage examples for specific API methods, and do not address the challenge of choosing which library to use. Furthermore, several of these approaches output API call sequences, instead of ready-to-use solutions in the form of snippets. Finally, none of the aforementioned systems accept queries in natural language, which are certainly preferable when trying to formulate a programming task without knowing which APIs to use beforehand.

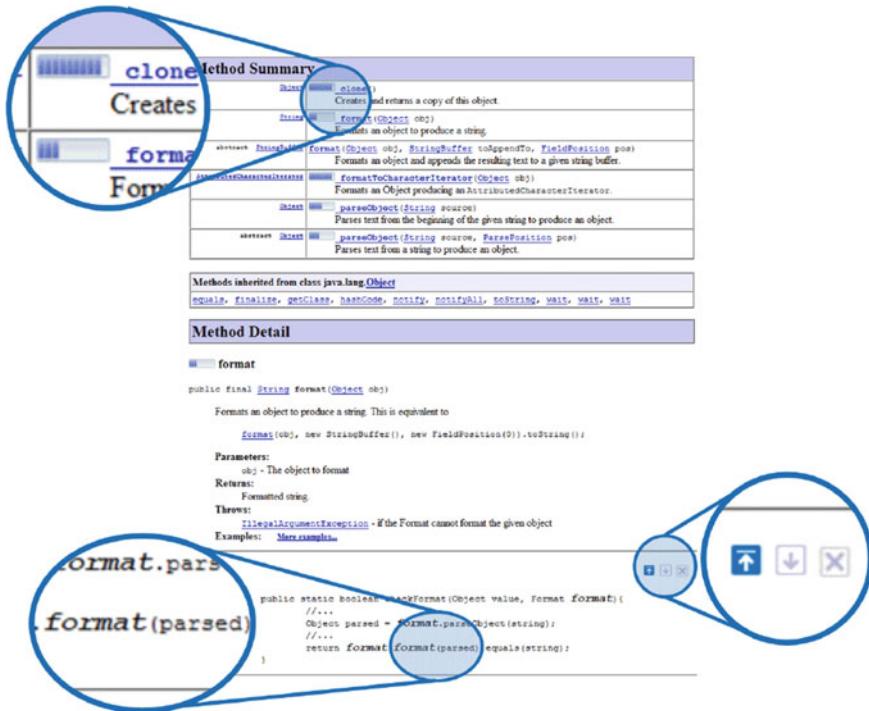


Fig. 7.2 Example screenshot of eXoDocs [6]

To address the above challenges, several recent systems focus on generic snippets and employ some type of processing for natural language queries. Such an example system is SnipMatch [7], the snippet recommender of the Eclipse IDE, which also incorporates several interesting features, including variable renaming for easier snippet integration. However, its index has to be built by the developer who has to provide the snippets and the corresponding textual descriptions. A relevant system also offered as an Eclipse plugin is Blueprint [8]. Blueprint employs the Google search engine to discover and rank snippets, thus ensuring that useful results are retrieved for almost any query. An even more advanced system is Bing Code Search [9], which employs the Bing search engine for finding relevant snippets, and further introduces a multi-parameter ranking system for snippets as well as a set of transformations to adapt the snippet to the source code of the developer.

The aforementioned systems, however, do not provide a choice of implementations to the developer. Furthermore, most of them do not assess the retrieved snippets from a reusability perspective. This is crucial, as libraries that are preferred by developers typically exhibit high quality and good documentation, while they are obviously supported by a larger community [15, 16]. In this chapter, we present CodeCatch, a snippet mining system designed to overcome the above limitations. CodeCatch employs the Google search engine in order to receive queries in natural language

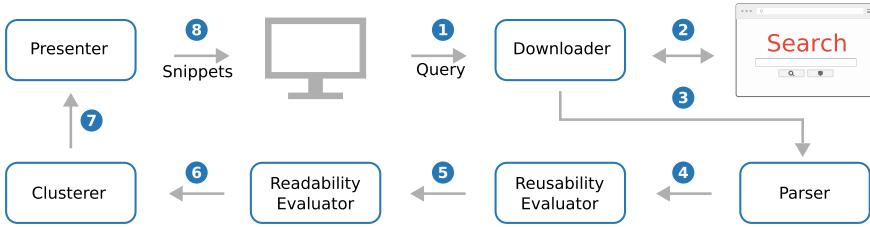


Fig. 7.3 CodeCatch system overview

and at the same time extract snippets from multiple online sources. As opposed to current systems, our tool assesses not only the quality (readability) of the snippets but also their reusability/preference by the developers. Furthermore, CodeCatch employs clustering techniques in order to group the snippets according to their API calls, and thus allows the developer to easily distinguish among different implementations.

7.3 CodeCatch Snippet Recommender

The architecture of CodeCatch is shown in Fig. 7.3. The input of the system is a query given in natural language to the Downloader module, which is then posted to the Google search engine, to extract code snippets from the result pages. Consequently, the Parser extracts the API calls of the snippets, while the Reusability Evaluator scores the snippets according to whether they are widely used/preferred by developers. Additionally, the readability of the snippets is assessed by the Readability Evaluator. Finally, the Clusterer groups the snippets according to their API calls, while the Presenter ranks them and presents them to the developer. These modules are analyzed in the following subsections.

7.3.1 Downloader

The downloader receives as input the query of the developer in natural language and posts it in order to retrieve snippets from multiple sources. An example query that will be used throughout this section is the query “How to read a CSV file”. The downloader receives the query and augments it before issuing it in the Google search engine. Note that our methodology is programming language-agnostic; however, without loss of generality we focus in this work on the Java programming language. In order to ensure that the results returned by the search engine will be targeted to the Java language, the query augmentation is performed using the Java-related keywords `java`, `class`, `interface`, `public`, `protected`, `abstract`, `final`, `static`, `import`,

Fig. 7.4 Example snippet for “How to read a CSV file”

```

String line = "";
BufferedReader br = null;
try {
    br = new BufferedReader(new FileReader("test.csv"));
    while((line = br.readLine()) != null) {
        String[] data = line.split(",");
    }
    br.close();
} catch (Exception e) {
    System.err.println("CSV file cannot be read: " + e);
}

```

if, for, void, int, long, and double. Similar lists of keywords can be constructed for supporting other languages.

The URLs that are returned by Google are scraped using Scrapy.² In specific, upon retrieving the top 40 web pages, we extract text from HTML tags such as `<pre>` and `<code>`. Scraping from those tags allows us to gather the majority (around 92% as measured) of code content from web pages. Apart from the code, we also collect information relevant to the webpage, including the URL of the result, its rank at the Google search results list, and the relative position of each code fragment inside the page.

7.3.2 Parser

The snippets are then parsed using the sequence extractor described in [17], which extracts the AST of each snippet and takes two passes over it, one to extract all (non-generic) type declarations (including fields and variables), and one to extract the method invocations (API calls). Consider, for example, the snippet of Fig. 7.4. During the first pass, the parser extracts the declarations `line: String`, `br: BufferedReader`, `data: String[]`, and `e: Exception`. Then, upon removing the generic declarations (i.e., literals, strings, and exceptions), the parser traverses the AST for a second time and extracts the relevant method invocations, which are highlighted in Fig. 7.4. The caller of each method invocation is replaced by its type (apart from constructors for which types are already known), to finally produce the API calls `FileReader.__init__`, `BufferedReader.__init__`, `BufferedReader.readLine`, and `BufferedReader.close`, where `__init__` signifies a call to a constructor.

Note that the parser is quite robust even in cases when the snippets are not compilable, while it also effectively isolates API calls that are not related to a type (since generic calls, such as `close`, would only add noise to the invocations). Finally, any

²<https://scrapy.org/>.

snippets not referring to Java source code and/or not producing API calls are dropped at this stage.

7.3.3 Reusability Evaluator

Upon gathering the snippets and extracting their API calls, the next step is to determine whether they are expected to be of use to the developer. In this context of *reusability*, we want to direct the developer toward what we call *common practice*, and, to do so, we make the assumption that snippets with API calls *commonly used* by other developers are more probable to be of (re)use. This is a reasonable assumption since answers to common programming questions are prone to appear often in the code of different projects. As a result, we designed the Reusability Evaluator by downloading a set of high-quality projects and determining the amount of reuse for the API calls of each snippet.

For this task, we have used the 1000 most popular Java projects of GitHub, as determined by the number of stars assigned. These were drawn from the index of AGORA (see Chap. 5). The rationale behind this choice of projects is highly intuitive and is also strongly supported by current research; popular projects have been found to exhibit high quality [15], while they contain reusable source code [18] and sufficient documentation [16]. As a result, we expect that these projects use the most effective APIs in a good way.

Upon downloading the projects, we construct a local index where we store their API calls, which are extracted using the Parser.³ After that, we score each API call by dividing the number of projects in which it is present by the total number of projects. For the score of each snippet, we average over the scores of its API calls. Finally, the index also contains all qualified names so that we may easily retrieve them given a caller object (e.g., `BufferedReader: java.io.BufferedReader`).

7.3.4 Readability Evaluator

To construct a model for the readability of snippets, we used the publicly available dataset from [19] that contains 12,000 human judgements by 120 annotators on 100 snippets of code. We build our model as a binary classifier that assesses whether a code snippet is *more readable* or *less readable*. At first, for each snippet, we extract a set of features that are related to readability, including the average line length, the average identifier length, the average number of comments, etc. (see [19] for the full

³As a side note, this local index is only used to assess the reusability of the components; our search, however, is not limited within it (as is the case with other systems) as we employ a search engine and crawl multiple pages. To further ensure that our reusability evaluator is always up-to-date, we could rebuild its index along with the rebuild cycles of the index of AGORA.

Fig. 7.5 Example snippet for “How to read a CSV file” using Scanner

```

Scanner scanner = null;
try{
    scanner = new Scanner(new File("test.csv"));
    scanner.useDelimiter(",");
    while(scanner.hasNext()) {
        System.out.print(scanner.next() + " ");
    }
    scanner.close();
} catch (Exception e) {
    System.err.println("CSV file cannot be read: " + e);
}

```

list of features). After that, we train an AdaBoost classifier on the aforementioned dataset. The classifier was built with decision trees as the base estimator, while the number of estimators and the learning rate were set to 160 and 0.6, respectively. We built our model using 10-fold cross-validation and the average F-measure for all folds was 85%, indicating that it is effective enough for making a binary decision concerning the readability of new snippets.

7.3.5 Clusterer

Upon scoring the snippets for both their reusability and their readability, the next step is to cluster them according to the different implementations. In order to perform clustering, we first have to extract the proper features from the retrieved snippets. A simple approach would be to cluster the snippets by examining them as text documents; however, this approach would fail to distinguish among different implementations. Consider, for example, the snippet of Fig. 7.4 along with that of Fig. 7.5. If we remove any punctuation and compare the two snippets, we may find out that more than 60% of the tokens of the second snippet are also present in the first. The two snippets, however, are quite different; they have different API calls and thus refer to different implementations.

As a result, we decided to cluster code snippets based on their API calls. To do so, we employ a Vector Space Model (VSM) to represent snippets as documents and API calls as vectors (dimensions). Thus, at first, we construct a document for each snippet based on its API calls. For example, the document for the snippet of Fig. 7.4 is “FileReader.__init__ BufferedReader.__init__ BufferedReader.readLine BufferedReader.close”, while the document for that of Fig. 7.5 is “File.__init__ Scanner.__init__ Scanner.hasNext Scanner.next Scanner.close”. After that, we use a *tf-idf* vectorizer to extract the vector representation for each document. The weight (vector value) of each term t in a document d is computed by the following equation:

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D) \quad (7.1)$$

where $tf(t, d)$ is the term frequency of term t in document d and refers to the appearances of the API call in the snippet, while $idf(t, D)$ is the inverse document frequency of term t in the set of all documents D , referring to how common the API call is in all the retrieved snippets. In specific, $idf(t, D)$ is computed as follows:

$$idf(t, D) = 1 + \log \frac{1 + |D|}{1 + d_t} \quad (7.2)$$

where $|d_t|$ is the number of documents containing the term t , i.e., the number of snippets containing the relevant API call. The idf ensures that very common API calls (e.g., `Exception.printStackTrace`) are given low weights, so that they do not outweigh more decisive invocations.

Before clustering, we also need to define a distance metric that shall be used to measure the similarity between two vectors. Our measure of choice is the cosine similarity, which is defined for two document vectors d_1 and d_2 using the following equation:

$$\text{cosine_similarity}(d_1, d_2) = \frac{d_1 \cdot d_2}{|d_1| \cdot |d_2|} = \frac{\sum_1^N w_{t_i, d_1} \cdot w_{t_i, d_2}}{\sum_1^N w_{t_i, d_1}^2 \cdot \sum_1^N w_{t_i, d_2}^2} \quad (7.3)$$

where w_{t_i, d_1} and w_{t_i, d_2} are the tf-idf scores of term t_i in documents d_1 and d_2 , respectively, and N is the total number of terms.

We select K-Means as our clustering algorithm, as it is known to be effective in text clustering problems similar to ours [20]. The algorithm, however, still has an important limitation as it requires as input the number of clusters. Therefore, to automatically determine the best value for the number of clusters, we employ the silhouette metric. The silhouette was selected as it encompasses both the similarity of the snippets within the cluster (cohesion) and their difference with the snippets of other clusters (separation). As a result, we expect that using the value of the silhouette as an optimization parameter should yield clusters that clearly correspond to different implementations. We execute K-Means for 2–8 clusters, and each time we compute the value of the silhouette coefficient for each document (snippet) using the following equation:

$$\text{silhouette}(d) = \frac{b(d) - a(d)}{\max(a(d), b(d))} \quad (7.4)$$

where $a(d)$ is the average distance of document d from all other documents in the same cluster, while $b(d)$ is computed by measuring the average distance of d from the documents of each of the other clusters and keeping the lowest one of these values (each corresponding to a cluster). For both parameters, the distances between documents are measured using Eq. (7.3). Finally, the silhouette coefficient for a cluster is given as the mean of the silhouette values of its snippets, while the

Fig. 7.6 Silhouette score of different number of clusters for the example query “How to read a CSV file”

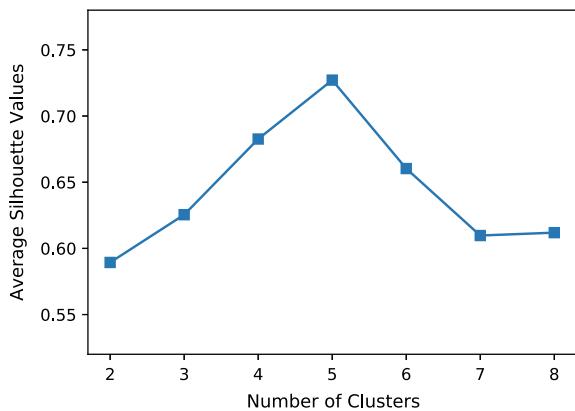
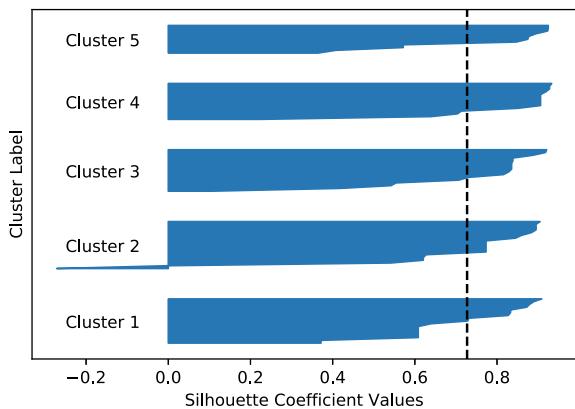


Fig. 7.7 Silhouette score of each cluster when grouping into 5 clusters for the example query “How to read a CSV file”



total silhouette for all clusters is given by averaging over the silhouette values of all snippets.

We present an example silhouette analysis for the query “How to read a CSV file”. Figure 7.6 depicts the silhouette score for 2–8 clusters, where it is clear that the optimal number of clusters is 5.

Furthermore, the individual silhouette values for the documents (snippets) of the five clusters are shown in Fig. 7.7, and they also confirm that the clustering is effective as most samples exhibit high silhouette and only a few have marginally negative values.

7.3.6 Presenter

The presenter is the final module of CodeCatch and handles the ranking and the presentation of the results. We have developed the CodeCatch prototype as a web



Fig. 7.8 Screenshot of CodeCatch for query “How to read a CSV file”, depicting the top three clusters

application. When the developer inserts a query, he/she is first presented with the clusters that correspond to different implementations for the query.

An indicative view of the first three clusters containing CSV file reading implementations is shown in Fig. 7.8. The proposed implementations of the clusters involve the BufferedReader API (e.g., as in Fig. 7.4), the Scanner API (e.g., as in Fig. 7.5), and the Java CSV reader API.⁴ The clusters are ordered according to their API reusability score, which is the average of the score of each of their snippets, as defined in Sect. 7.3.3. For each cluster, CodeCatch provides the five most frequent API imports and the five most frequent API calls, to aid the developer to distinguish among the different implementations. In cases where imports are not present in the snippets, they are extracted using the index created in Sect. 7.3.3.

Upon selecting to explore a cluster, the developer is presented with a list of its included snippets. The snippets within a cluster are ranked according to their API reusability score, and in cases of equal scores according to their distance from the cluster centroid (computed using Eq.(7.3)). This ensures that the most common usages of a specific API implementation are higher on the list. Furthermore, for each snippet, CodeCatch provides useful information, as shown in Fig. 7.9, including its reusability score (*API Score*), its distance from the centroid, its readability (either Low or High), the position of its URL in the results of the Google search engine and its order inside the URL, its number of API calls, and its number of lines of code. Finally, apart from immediately reusing the snippet, the developer has the option to isolate only the code that involves its API calls, while he/she can also check the webpage from which the snippet was retrieved.

7.4 Example Usage Scenario

In this section, we provide an example usage scenario of CodeCatch based on the scenario described in the previous chapter. In specific, we build on top of the duplicate file check application that was developed using our component reuse methodology.

⁴<https://gist.github.com/jaysridhar/d61ea9cbede617606256933378d71751>.

API Score: 0.29 Centroid Distance: 0.10 Readability: Low Position: 6 Order in page: 4 Number of API calls: 4 Lines of Code: 14

Show invocations - Go to snippet webpage

```
public class InsertValuesIntoTestDb {

    public static void main(String[] args) throws Exception {
        String splitBy = ",";
        BufferedReader br = new BufferedReader(new FileReader("test.csv"));
        while((line = br.readLine()) != null) {
            String[] b = line.split(splitBy);
            System.out.println(b[0]);
        }
        br.close();
    }
}
```

Fig. 7.9 Screenshot of CodeCatch for query “How to read a CSV file”, depicting an example snippet

Table 7.1 API clusters of CodeCatch for query “How to upload file to FTP”

ID	Library	API	Score (%)
1	Apache commons	org.apache.commons.net.ftp.*	83.57
2	Jscape	com.jscape.inet.ftp.*	11.39
3	EnterpriseDT edtFTPj	com.enterprisedt.net.ftp.*	4.78
4	Spring framework	org.springframework.integration.ftp.*	0.26

In this case, we assume that the developer has built the relevant application, which scans the files of a given folder and checks their MD5 checksums before adding them to a given path. Let us now assume that the next step involves uploading these files to an FTP server.

FTP file handling is typically a challenge that can be confronted using external libraries/frameworks, therefore the process of writing code to perform this task is well suited for CodeCatch. Thus, the developer, who does not necessarily know about any components and/or APIs, initially issues the simple textual query “How to upload file to FTP”. In this case, CodeCatch returns four result clusters, each linked to a different API (and, thus, a different framework). The returned API clusters are summarized in Table 7.1 (the API column is returned by the service, while the library column is provided here for clarity).

As one may easily observe, there are different equivalent implementations for uploading files to an FTP server. The developer could, of course, choose any of them, depending on whether he/she already is familiar with any of these libraries. However, in most scenarios (and let us assume also in our scenario), the developer would select to use the most highly preferred implementation, which in this case is the implementation offered by Apache Commons. Finally, upon selecting to enter the first cluster, the developer could review the snippets and select one that fits his/her purpose, such as the one shown in Fig. 7.10.

```

String ftpUrl = "ftp://%" + host + "%" + type + "i";
String host = "www.myserver.com";
String user = "tom";
String pass = "secret";
String filePath = "E:/Work/Project.zip";
String uploadPath = "/MyProjects/archive/Project.zip";

ftpUrl = String.format(ftpUrl, user, pass, host, uploadPath);
System.out.println("Upload URL: " + ftpUrl);

try {
    URL url = new URL(ftpUrl);
    URLConnection conn = url.openConnection();
    OutputStream outputStream = conn.getOutputStream();
    FileInputStream inputStream = new FileInputStream(filePath);
    byte[] buffer = new byte[BUFFER_SIZE];
    int bytesRead = -1;
    while ((bytesRead = inputStream.read(buffer)) != -1) {
        outputStream.write(buffer, 0, bytesRead);
    }
    inputStream.close();
    outputStream.close();
    System.out.println("File uploaded");
} catch (IOException ex) {
    ex.printStackTrace();
}

```

Fig. 7.10 Example snippet for “How to upload file to FTP” using Apache Commons

7.5 Evaluation

7.5.1 *Evaluation Framework*

Comparing CodeCatch with similar approaches has not been performed in a straightforward manner, as several of them focus on mining single APIs, while others are not maintained and/or are not publicly available. Our focus is mainly on the merit of reuse for results, and the system that is most similar to ours is Bing Code Search [9], however, it targets the C# programming language. Hence, we have decided to perform a usability-related evaluation against the Google search engine on a dataset of common queries shown in Table 7.2.

The purpose of our evaluation is twofold; we wish not only to assess whether the snippets of our system are relevant, but also to determine whether the developer

Table 7.2 Statistics of the queries used as evaluation dataset

ID	Query	Clusters	Snippets
1	How to read CSV file	5	76
2	How to generate MD5 hash code	5	65
3	How to send packet via UDP	5	34
4	How to split string	4	22
6	How to upload file to FTP	4	31
5	How to send email	5	79
7	How to initialize thread	6	51
8	How to connect to a JDBC database	5	42
9	How to read ZIP archive	6	82
10	How to play audio file	6	45

can indeed more easily find snippets for all different APIs relevant to a query. Thus, at first, we annotate the retrieved snippets for all the queries as relevant and non-relevant. To maintain an objective and systematic outlook, the annotation procedure was performed without any knowledge on the ranking of the snippets. For the same reason, the annotation was kept as simple as possible; snippets were marked as relevant if and only if their code covers the functionality described by the query. That is, for the query, “How to read CSV file”, any snippets used to read a CSV file were considered relevant, regardless of their size or complexity, and of any libraries involved, etc.

As already mentioned, the snippets are assigned to clusters, where each cluster involves different API usages and thus corresponds to a different implementation. As a result, we have to assess the relevance of the results per cluster, hence assuming that the developer would first select the desired implementation and then navigate into the cluster. To do so, we compare the snippets of each cluster (i.e., of each implementation) to the results of the Google search engine. CodeCatch clusters already provide lists of snippets, while for Google, we construct one by assuming that the developer opens the first URL, subsequently examines the snippets of this URL from top to bottom, then he/she opens the second URL, etc.

When assessing the results of each cluster, we wish to find snippets that are relevant not only to the query but also to the corresponding API usages. As a result, for the assessment of each cluster, we further annotate the results of both systems to consider them relevant when they are also part of the corresponding implementation. This, arguably, produces less effective snippet lists for the Google search engine, however, note that our purpose is not to challenge the results of the Google search

engine in terms of relevance to the query, but rather to illustrate how easy or hard it is for the developer to examine the results and isolate the different ways of answering his/her query.

For each query, upon having constructed the lists of snippets for each cluster and for Google, we compare them using the *reciprocal rank* metric. This metric was selected as it is commonly used to assess information retrieval systems in general and also systems similar to ours [9]. Given a list of results, the reciprocal rank for a query is computed as the inverse of the rank of the first relevant result. For example, if the first relevant result is in the first position, then the reciprocal rank is $1/1 = 1$, if the result is in the second position, then the reciprocal rank is $1/2 = 0.5$, etc.

7.5.2 Evaluation Results

Figure 7.11 depicts the reciprocal rank of CodeCatch and Google for the snippets corresponding to the three most popular implementations for each query. At first, interpreting this graph in terms of the relevance of the results indicates that both systems are very effective. In specific, if we consider that the developer would require a relevant snippet regardless of the implementation, then for most queries, both CodeCatch and Google produce a relevant result in the first position (i.e., reciprocal rank equal to 1).

If, however, we focus on all different implementations for each query, we can make certain interesting observations. Consider, for example, the first query (“How to read a CSV file”). In this case, if the developer requires the most popular `BufferedReader` implementation (I1), both CodeCatch and Google output a relevant snippet in the first position. Similarly, if one wished to use the `Scanner` implementation (I2) or the Java CSV reader (I3), our system would return a ready-to-use snippet in the top of the second cluster or in the second position of the third cluster (i.e., reciprocal rank equal to 0.5). On the other hand, using Google would require examining more results (3 and 50 results for I2 and I3, respectively, as the corresponding reciprocal ranks are equal to 0.33 and 0.02, respectively). Similar conclusions can be drawn for most queries.

Another important point of comparison of the two systems is whether they return the most popular implementations at the top of their list. CodeCatch is clearly more effective than Google in this aspect. Consider, for example, the sixth query; in this case, the most popular implementation is found in the third position of Google, while the snippet found in its first position corresponds to a less popular implementation. This is also clear in several other queries (i.e., queries 2, 3, 5, 7). Thus, one could argue that CodeCatch does not only provide the developer with all different API implementations for his/her query but also further aids him/her to select the most popular of these implementations, which is usually the most preferable.

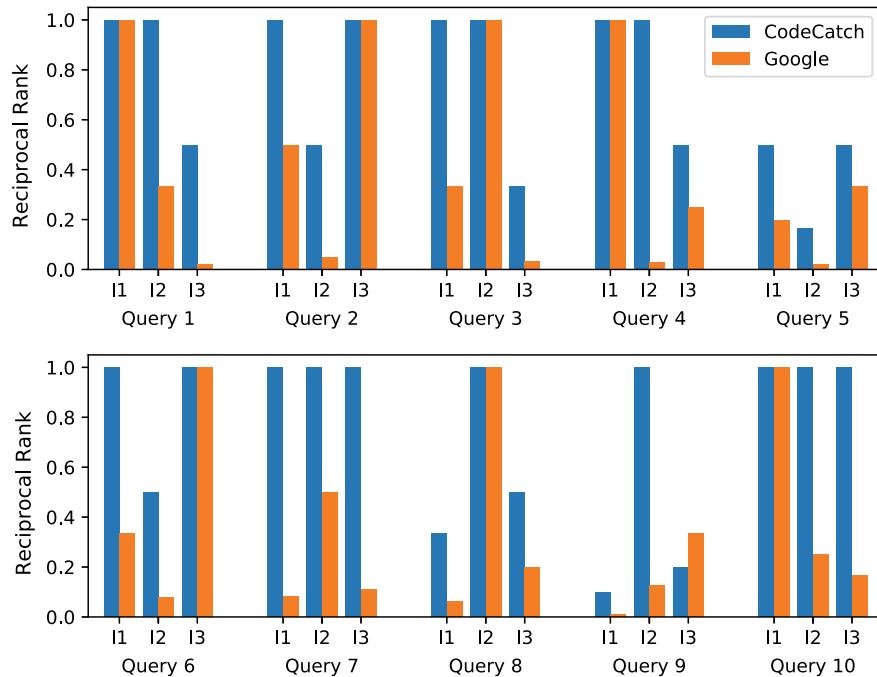


Fig. 7.11 Reciprocal Rank of CodeCatch and Google for the three most popular implementations (I1, I2, I3) of each query

7.6 Conclusion

In this chapter, we proposed a system that extracts snippets from online sources and further assesses their readability as well as their reusability based on the preference of developers. Moreover, our system, CodeCatch, provides a comprehensive view of the retrieved snippets by grouping them into clusters that correspond to different implementations. This way, the developer can select among possible solutions to common programming queries even in cases when he/she does not know which API to use beforehand.

Future work on CodeCatch lies in several directions. At first, we may extend our ranking scheme to include the position of each snippet's URL in the Google results, etc. Furthermore, snippet summarization can be performed using information from the clustering (e.g., removing statements that appear in few snippets within a cluster). Finally, an interesting idea would be to conduct a developer study in order to further assess CodeCatch for its effectiveness in retrieving useful source code snippets.

References

1. Xie T, Pei J (2006) MAPO: mining API usages from open source repositories. In: Proceedings of the 2006 international workshop on mining software repositories, MSR '06, pp 54–57, New York, NY, USA. ACM
2. Wang J, Dang Y, Zhang H, Chen K, Xie T, Zhang D (2013) Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 10th working conference on mining software repositories, pp 319–328, Piscataway, NJ, USA. IEEE Press
3. Fowkes J, Sutton C (2016) Parameter-free probabilistic API mining across GitHub. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 254–265, New York, NY, USA. ACM
4. Montandon JE, Borges H, Felix D, Valente MT (2013) Documenting APIs with examples: lessons learned with the APIMiner platform. In: 2013 20th working conference on reverse engineering (WCRE), pp 401–408, Piscataway, NJ, USA. IEEE Computer Society
5. Buse RPL, Weimer W (2012) Synthesizing API usage examples. In: Proceedings of the 34th international conference on software engineering, ICSE '12, pp 782–792, Piscataway, NJ, USA. IEEE Press
6. Kim J, Lee S, Hwang SW, Kim S (2010) Towards an intelligent code search engine. In: Proceedings of the Twenty-Fourth AAAI conference on artificial intelligence, AAAI'10, pp 1358–1363, Palo Alto, CA, USA. AAAI Press
7. Wightman D, Ye Z, Brandt J, Vertegaal R (2012) SnipMatch: using source code context to enhance snippet retrieval and parameterization. In: Proceedings of the 25th annual ACM symposium on user interface software and technology, UIST '12, pp 219–228, New York, NY, USA. ACM
8. Brandt J, Dontcheva M, Weskamp M, Klemmer SR (2010) Example-centric programming: integrating web search into the development environment. In: Proceedings of the SIGCHI conference on human factors in computing systems, CHI '10, pp 513–522, New York, NY, USA. ACM
9. Wei Y, Chandrasekaran N, Gulwani S, Hamadi Y (2015) Building bing developer assistant. Technical Report MSR-TR-2015-36, Microsoft Research
10. Mandelin D, Lin X, Bodik R, Kimelman D (2005) Jungloid mining: helping to navigate the API jungle. SIGPLAN Not 40(6):48–61
11. Thummalapenta S, Xie T (2007) PARSEWeb: a programmer assistant for reusing open source code on the web. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering, ASE '07, pp 204–213, New York, NY, USA. ACM
12. Wang J, Han J (2004) BIDE: efficient mining of frequent closed sequences. In: Proceedings of the 20th international conference on data engineering, ICDE '04, pp 79–90, Washington, DC, USA. IEEE Computer Society
13. Katirtzis N, Diamantopoulos T, Sutton C (2018) Summarizing software API usage examples using clustering techniques. In: 21th international conference on fundamental approaches to software engineering, pp 189–206, Cham. Springer International Publishing
14. Jiang L, Mishergi G, Su Z, Glouud S (2007) DECKARD: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on software engineering, ICSE '07, pp 96–105, Washington, DC, USA. IEEE Computer Society
15. Papamichail M, Diamantopoulos T, Symeonidis AL (2016) User-perceived source code quality estimation based on static analysis metrics. In: Proceedings of the 2016 IEEE international conference on software quality, reliability and security, QRS, pp 100–107, Vienna, Austria
16. Aggarwal K, Hindle A, Stroulia E (2014) Co-evolution of project documentation and popularity within Github. In: Proceedings of the 11th working conference on mining software repositories, MSR 2014, pp 360–363, New York, NY, USA. ACM
17. Diamantopoulos T, Symeonidis AL (2015) Employing source code information to improve question-answering in stack overflow. In: Proceedings of the 12th working conference on mining software repositories, MSR '15, pp 454–457, Piscataway, NJ, USA. IEEE Press

18. Dimaridou V, Kyprianidis A-C, Papamichail M, Diamantopoulos T, Symeonidis A (2017) Towards modeling the user-perceived quality of source code using static analysis metrics. In: Proceedings of the 12th international conference on software technologies - Volume 1, ICSOFT, pp 73–84, Setubal, Portugal. INSTICC, SciTePress
19. Buse RPL, Weimer WR (2010) Learning a metric for code readability. IEEE Trans Softw Eng 36(4):546–558
20. Aggarwal CC, Zhai C (2012) A survey of text clustering algorithms, pp 77–128. Springer, Boston

The Road Ahead for Mining Software Repositories

Ahmed E. Hassan

Software Analysis and Intelligence Lab (SAIL)

School of Computing, Queen's University, Canada

ahmed@cs.queensu.ca

Abstract

Source control repositories, bug repositories, archived communications, deployment logs, and code repositories are examples of software repositories that are commonly available for most software projects. The Mining Software Repositories (MSR) field analyzes and cross-links the rich data available in these repositories to uncover interesting and actionable information about software systems. By transforming these repositories from static record-keeping ones into active repositories, we can guide decision processes in modern software projects. For example, data in source control repositories, traditionally used to archive code, could be linked with data in bug repositories to help practitioners propagate complex changes and to warn them about risky code based on prior changes and bugs. In this paper, we present a brief history of the MSR field and discuss several recent achievements and results of using MSR techniques to support software research and practice. We then discuss the various opportunities and challenges that lie in the road ahead for this important and emerging field.

1 Introduction

Prior experiences and dominant patterns are the driving force for many decision-processes in modern software organizations. Practitioners often rely on their experience, intuition and gut feeling in making important decisions. Managers allocate development and testing resources based on their experience in previous projects and their intuition about the complexity of the new project relative to prior projects. Developers commonly use their experience when adding a new feature or fixing a bug. Testers usually prioritize the testing of features that are known to be error prone based on field and bug reports. Software repositories contain a wealth of valuable information about software projects. Using the information stored in these repositories, practitioners can depend less on their intuition and experience, and depend more on historical and field data.

The Mining Software Repositories (MSR) field analyzes and cross-links the rich data available in software repositories to uncover interesting and actionable information about software systems and projects. Examples of software repositories are:

Historical repositories such as source control repositories, bug repositories, and archived communications record several information about the evolution and progress of a project.

Run-time repositories such as deployment logs contain information about the execution and the usage of an application at a single or multiple deployment sites.

Code repositories such as Sourceforge.net and Google code contain the source code of various applications developed by several developers.

Software repositories are commonly used in practice as record-keeping repositories and are rarely used to support decision making processes. For example, historical repositories are used to track the history of a bug or a feature, but are not commonly used to determine the expected resolution time of an open bug based on the resolution time of previously-closed bugs.

MSR researchers aim to transform these repositories from static record-keeping ones into active repositories which can guide decision processes in modern software projects. Mining these historical, run-time and code repositories, we can uncover useful and important patterns and information. Historical repositories capture important historical dependencies [20] between project artifacts, such as functions, documentation files, or configuration files. Developers can use this information to propagate changes to related artifacts, instead of only using static or dynamic code dependencies which may fail to capture important dependencies. For example, a change to the code which writes data to a file may require changes to the code which reads data from the file, although there exists no traditional dependencies (e.g., data and control flow) between both pieces of

code. As for run-time repositories, they could be used to pinpoint execution anomaly by identifying dominant execution or usage patterns across deployments, and flagging deviations from these patterns. Code repositories could be used to identify dominant and correct library usage patterns by mining the usage of a library across many projects.

The MSR field is rapidly taking a central and important role in supporting software development practice and software engineering research. In this paper, we present a brief history of the MSR field and discuss several recent achievements and results of using MSR techniques to support software research and practice. We then discuss the various opportunities and challenges that lie in the road ahead for this important and emerging field.

2 A Brief History of the Mining Software Repositories (MSR) Field

The Mining Software Repositories (MSR) field is maturing thanks to the rich, extensive, and readily available software repositories. Table 1 lists several examples of software repositories which could be mined. Although these repositories are readily available for most large software projects, the data stored in these repositories has not been the focus of software engineering research until recently. This is owing primarily to the following two reasons:

Limited Access to Repositories. Companies in many cases are not willing to give researchers access to such detailed information about their software systems. Another possible source for software repositories is academic projects and systems. Unfortunately, the repositories of such projects and systems are not as rich nor as interesting as the those of long-lived widely-deployed commercial software systems. The earliest research work in the MSR field were based on the repositories of commercial software systems and were done in cooperation with a few commercial organizations such as NASA [7], AT&T [55, 14], Nortel [30], Nokia [20], Avaya [50], and Mitel [59].

Complexity of Data Extraction. Most repositories are not designed with automated bulk data-extraction and mining in mind, so they provide limited support for automated extraction. The complexity of automated data extraction hindered the adoption and integration of software repositories in other software engineering research. In many cases, software engineering researchers do not have the expertise required nor do they have the interest to extract data from software repositories. Extracting such data requires a great deal of effort and time from researchers, instead they are more interested in gaining convenient access to the extracted data in an easy to process format.

With the advent of open source systems, easy access to repositories of large software systems became a reality. Researchers now have access to rich repositories for large projects used by thousands of users and developed by hundreds of developers over extended periods of time. Early research in mining software repositories, e.g.[10], made use of open source repositories, thanks to the wide spread and growth of open source projects.

In an effort to bring together the practitioners and researchers working in this important and emerging field, the first International Workshop on Mining Software Repositories (MSR) was held at the International Conference on Software Engineering (ICSE), the flagship conference for Software Engineering. After four successful years as ICSE's largest workshop, MSR became a Working Conference in 2008. As a Working Conference, MSR recognizes the maturity and breadth of the work in the MSR field, while still encouraging free-form open discussions about the MSR field. An MSR challenge is also held on a yearly basis so researchers can compare their techniques towards a common problem or project. The MSR field continues to attract a large amount of interest within software engineering. A 2005 issue of the IEEE Transactions on Software Engineering (TSE) on the MSR topic received over 15% of all the submissions to the TSE in 2004 [27]. And MSR-related publications in top research venues continue to grow in size and quality with many MSR-related papers winning awards in top venues.

3 An Overview of MSR Achievements

MSR research focuses primarily on two aspects:

1. The creation of techniques to automate and improve the extraction of information from repositories.
2. The discovery and validation of novel techniques and approaches to mine important information from these repositories.

The first line of work is essential to ease the adoption of MSR techniques by others, and the second line of work demonstrates the importance and value of information stored in software repositories and encourages others to adopt MSR techniques. In this section we present a few interesting research results along a number of dimensions to demonstrate the benefits of using MSR techniques in solving several challenging and important software engineering problems. For a more extensive review of the work in the MSR field, we encourage the reader to refer to the “Mining Software Engineering Data Bibliography” maintained by Tao Xie at <http://ase.csc.ncsu.edu/dmse/> at North Carolina State University, and a survey paper by Kagdi *et al.* [32] on mining historical repositories.

Repository	Description
Source control repositories	These repositories record the development history of a project. They track all the changes to the source code along with meta-data about each change, e.g., the name of the developer who performed the change, the time the change was performed and a short message describing the change. Source control repositories are the most commonly available and used repository in software projects. CVS, subversion, Perforce, ClearCase are examples of source control repositories which are used in practice.
Bug repositories	These repositories track the resolution history of bug reports or feature requests that are reported by users and developers of large software projects. Bugzilla and Jira are examples of bug repositories.
Archived communications	These repositories track discussions about various aspects of a software project throughout its lifetime. Mailing lists, emails, IRC chats, and instant messages are examples of archived communications about a project.
Deployment logs	These repositories record information about the execution of a single deployment of a software application or different deployments of the same applications. For example, the deployment logs may record the error messages reported by an application at various deployment sites. The availability of deployment logs continues to increase at a rapid rate due to their use for remote issue resolution and due to recent legal acts. For instance, the Sarbanes-Oxley Act of 2002 [1] stipulates that the execution of telecommunication and financial applications must be logged.
Code repositories	These repositories archive the source code for a large number of projects. Sourceforge.net and Google code are examples of large code repositories.

Table 1. Examples of Software Repositories

3.1 Understanding Software Systems

Understanding large software systems remains a challenge for most software organizations. Documentations for large systems rarely exist and if they exist they are often not up-to-date. Moreover system experts are usually too busy to help novice developers, or may no longer be part of an organization. Information stored in historical software repositories, such as mailing lists and bug repositories, represent a group memory for a project. Such information is very valuable for current members of a project.

Cubranic *et al.* propose a tool called Hipikat which silently indexes historical repositories, and displays on-demand relevant historical information within the development environment [12]. While working on a particular change, Hipikat can display pertinent artifacts such as old emails and bug reports, discussing the code being viewed in the code editor.

Dependency graphs and the source code documentation offer a static view of a system and fail to reveal details about the history of a system or the rationale for its current state or design. For example, traditional dependency graphs cannot give the rationale behind an *Optimizer* function unexpectedly depending on a *Parser* function in a compiler. Such rationale is stored in the group's memory which lives in the historical repositories for a project. Hassan and Holt propose mining source control repositories and attaching Historical Sticky Notes to each code dependency in a software system [26]. These notes record various properties concerning a dependency such as the time it was introduced, the name of the developer who introduced it, and the rationale for adding it. Using the historical sticky notes on the NetBSD system, a large open source operating system, many unexpected dependencies could be easily explained and rationalized.

3.2 Propagating Changes

Change propagation is the process of propagating code changes to other entities of a software system to ensure the consistency of assumptions in the system after changing an entity. For example, a change to an interface may require the change to propagate to all the components which use that interface. The propagation of this change would ensure that both the interface and entities using it have a consistent set of assumptions. Change propagation plays a central role in software development. However current practices for propagating software changes rely heavily on human communication, and the knowledge of experienced developers. Many hard to find bugs are introduced by developers who did not notice dependencies between entities, and failed to propagate changes correctly.

Instead of using traditional dependency graphs to propagate changes, we could make use of the historical co-changes. The intuition is that entities co-changing frequently in the past are very likely to co-change in the future. This is similar in spirit to how retailers suggest other products for customers – Customers, who bought milk, bought cereal 99% of the time. Zimmermann *et al.* [68], Ying *et al.* [66], Shirabad [59], Hassan and Holt [25] show that suggestions based on historical co-changes are on average accurate 30% of the time (i.e., precision), and can correctly propose 44% of the entities which must co-change (i.e., recall). Recent results by Malik and Hassan show a precision of 64% and a recall of 78% could be achieved using more advanced data mining techniques [45]. Hassan and Holt also demonstrate that historical dependencies outperform traditional dependency information when propagating changes for several open source projects [25]. Kagdi *et al.* demonstrate the applicability of historical information in propagating changes from code to documentation entities where there are no structural code dependencies [33].

3.3 Predicting and Identifying Bugs

Predicting the occurrence of bugs remains one of the most active areas in software engineering research. Using this knowledge, managers can allocate testing resource appropriately, developers can review risky code more closely, and testers can prioritize their testing efforts. A substantial number of complexity metrics have been proposed over the years. The intuition being that complex code is likely to be buggy. However, results by Graves *et al.* [22] on a commercial system and by Herraiz *et al.* on a large sample of open source projects [28] show that most complexity metrics correlate well with LOC. Instead Graves *et al.* indicate that the two of the best predictors of bugs are prior bugs and prior changes, i.e., code that has bugs in the past is likely to have bugs in the future and bugs are not likely to appear in unchanged code. Moreover, Graves *et al.* [22] show that recent bugs and changes have a higher effect on the bug potential of a code over older changes. The importance of process metrics from historical repositories in predicting bugs over traditional complexity metrics has as well been recently shown by Moser *et al.* for the Eclipse open source project [52].

Instead of predicting the number of future bugs, practitioners desire tools that would flag the location of bugs in their current code base so they can fix these bugs. Unfortunately the specifications of large software applications rarely exist, so comparing an application against its expected and specified behavior using traditional static analysis techniques is usually not feasible in practice. Instead researchers have adopted novel techniques which analyze large amount of data about a software application to uncover the dominant behavior or patterns and to flag variations from that behavior as possible bugs. By analyzing the source code of an application, tools such as Coverity [16], can uncover undocumented correctness rules such as “acquire lock L before modifying x” or “do not block when interrupts are disabled”. The tools then flag any deviations of these rules in the source code. These deviations are likely bugs. Several types of rules could be mined from the source code: Coverity and PR-Miner [41] mine function-pairing rules; Daikon [17] and DIDUCE [23] mine variable-value invariants by analyzing execution traces; MUFI mines variable-pairing rules [44]; and AutoISES mines function and variable access rules to detect security violations [61]. Recent work by Jiang *et al.* applies many of the aforementioned techniques to uncover load testing problems for large enterprise applications [31]. A load test involves the repetitive execution of similar requests by an application; the proposed approach mines the execution logs to uncover dominant rules and deviation from these rules.

Other tools such as CP-Miner [40] can flag bugs by recognizing deviations in mined patterns for renaming variables when cloning (i.e., copy-and-paste) code.

Using historical code changes, DynaMine [43] uncovers function-pairing rules. For example, if addListener() and removeListener() are always added together to the code, then a change where addListener() is added without removeListener() is likely a buggy change. Williams and Hollingsworth use historical changes to refine the order of warnings produced by static code checkers [63]. For example, warnings for missing to check the return-value for a called function are ordered based on examining prior code changes – If a developer in the past added a check for the return-value for a particular function, then the importance of checking the return of that function is asserted, and warnings related to missing check are moved higher in the warning list.

3.4 Understanding Team Dynamics

Many large projects communicate through mailing lists, IRC channels, or instant messaging. These discussions cover many important topics such as future plans, design decisions, project policies, and code or patch reviews. These discussions represent a rich source of historical information about the inner workings of large projects. These discussions could be mined to better understand the dynamics of large software development teams.

In many open source projects, project outsiders can submit code patches to the mailing list. However outsiders cannot commit code directly to the source control repository until they are invited to be part of the core group of developers of a project. Inviting an outsider too early may lead to inviting individuals that are not well-qualified or may not fit well with the group. Inviting an outsider too late may lead to the outsider losing interest in the project and the project losing a valuable core developer. Bird *et al.* study the usual timelines for inviting developers to the core group in open source projects by mining information from source code repositories and mailing lists [9].

In addition to uncovering the process for inviting developers, mailing lists discussions could uncover the overall morale of a development team with developers using more optimistic words when they feel positive about the progress of the project and using negative words when they are concerned about the state of the project. Rigby and Hassan used a psychometric text analysis tool to analyze the mailing lists discussions to capture the overall morale of a development team before and after release time for the Apache web server project [56].

Users and developers are continuously logging bugs in the bug repository of large software projects. Each report must be triaged to determine if the report should be addressed, and to which developer it should be assigned. Bug triage is a time-consuming effort requiring extensive knowledge about a project and the expertise of its developers. Anvik and Murphy speed up the bug triage efforts by using

prior bug reports to determine the most suitable developers to which a bug should be assigned [5].

3.5 Improving the User Experience

Michail and Xie propose a Stabilizer tool which mines reported bugs and execution logs to prevent an application from crashing [47]. When a user attempts to perform an action which has been reported by others to be buggy, the Stabilizer tool presents a warning to the user who is given the opportunity to abort the action. This approach permits users to use buggy applications while they wait for developers to fix bugs.

Mockus *et al.* study the quality of a software application as perceived by its users [51]. Instead of studying the quality of the source code, they mine data captured by project monitoring and tracking infrastructures as well as customer support records to determine the expected quality of a software application. They find that the deployment schedule, hardware configurations, and software platform have a significant effect on the perceived quality of an application, increasing the probability of observing a software failure by more than 20 times.

3.6 Reusing Code

Code reuse is an important activity in modern software development. Unfortunately, large code libraries are usually not well-documented, and have flexible and complex APIs which are hard to use by non-experts. Several researchers have proposed tools and techniques to mine code repositories to help developers reuse code. The techniques locate uses of code such as library APIs, and attempt to match these uses to the needs of a developer working on a new piece of code. For example, Mandelin *et al.* develop a technique which helps a developer write the code needed to get access to a particular object [46], while Xie and Pei [65] propose a technique to help a developer write the setup and tear down code needed to use a library method.

3.7 Automating Empirical Studies

A major contribution of the MSR field, is the automation of many of the activities associated with empirical studies in software engineering. The automation permits the repetition of studies on a large number of subject and the ability to verify the generality of many findings in these studies. For example, recent work by Robles *et al.* [57] showed that the growth of 18 large open software follows a usually linear or superlinear trend. This work contradicts the fourth law of software evolution which was proposed by Lehman and colleagues [39] based on a small sample of industrial system. The MSR automation enables the verification of the generality of prior findings.

Common wisdom and literature about code cloning indicate that cloning is harmful and has a negative impact on the

maintainability of software systems. Kapser and Godfrey examine several large open source systems and show that cloning is often used as a principled engineering tool [34]. For example, cloning is often used to experiment with new features while ensuring the stability of the old code. New features are added in cloned code then re-integrated into the original code once the features are stabilized.

4 Opportunities in the Road Ahead

In the last section, we gave a brief survey of the impact of MSR on many important dimensions in software research and practice. The MSR field is very fortunate in that the cost of experimenting with MSR techniques is usually low – the data needed to perform MSR research and to demonstrate the value of adopting many of the MSR findings is readily available as it is collected by projects for other purposes. In the road ahead there exists many opportunities for the MSR field to demonstrate the strategic importance of software repositories and the benefit of transforming static software repositories into active ones which could support and automate many daily decision-processes in modern software development organizations. We present below several areas opportunities and discuss the challenges associated with these opportunities and work done within the MSR community to tackle some of these challenges.

4.1 Taming the Complexity of MSR

MSR techniques remain as advanced techniques with a large barrier of entry due to the complexities associated with data extraction and analysis. Although the data needed for MSR is readily available, the data is not collected with mining in mind. For example, commonly used source control repositories, such as CVS and subversion, track changes at the file level instead of tracking changes at the code entity level (e.g., functions and classes). And CVS, the most used source control repository, does not track the fact that several files have co-changed together or the purpose of a change, e.g., to fix a bug or to merge a code branch.

In the next few years, MSR researchers should focus on lowering the barrier of entry into the MSR field. Lowering the barrier of entry will bring in researchers from several other domains and wider experience, and will raise the diversity of the important problems that are being investigated using software repositories. MSR researchers should work on documenting best practices for mining repository data, and on providing access to mining tools and already mined data in well-defined exchange formats. Early work in exchange formats, such as [36], is still not well-adopted Nevertheless there are currently a few toolsets (e.g., [42]) and datasets (e.g., [2, 19]) that are available for others to use. The following are a few of the challenges that researchers must address by providing toolkits and advice for others to overcome these challenges.

Simplifying the Extraction of High-Quality Data. Many heuristics are used to extract data from code repositories (e.g., [24, 67, 18, 49]), from email repositories (e.g., [8, 56]), and from run-time repositories (e.g., [31]). Heuristics are used to deal with un-compilable code in a code repository. Heuristics are used to map a user’s email to a single individual since users do not have unique emails over the years or even within the same day (sending emails from home and work or school email addresses). Researchers should closely examine, document, and study the correctness of the used heuristics. Toolkits to help others extract data with limited knowledge about these heuristics are needed. However building tools and extracting data from repositories are time consuming and challenging tasks. There is currently a lack of proper ways to acknowledge the contributions of researchers who build tools and provide or share extracted data. The contributions of these researchers should be acknowledged as an important and essential contribution to the MSR field for these researchers and others to tackle this challenge.

Dealing with Skew in Repository Data. Often the data available in software repositories exhibits a large amount of noise and skew in it. For example, the count of changes and bugs to files tends to have high skew in it with a small portion of files having most of the bugs or changes in them. This large skew requires special attention when using traditional data mining algorithms. For example, decision tree learners have a high tendency to simply return the most common category in the data as a prediction when most-common-category occurs at a very high rate. More robust algorithms and data re-sampling techniques should be adopted [45].

Another example of skew exists in source control repositories with a small number of changes involving most of the files in a project. For example, each year all files of a project may be changed together in one large change to update the copyright year at the top of each file. The use of such data may lead to incorrect results and conclusions. MSR researchers should closely study the noise and skew in the data and better understand their effect on the analysis. Guidelines, techniques and tools are needed to detect noise and skew, and to accommodate them in the analysis. Visualization techniques, such as [15, 38, 64] are essential in helping spot noise and are important tools when mining software repositories. Detailed statistical sampling or manual (e.g., [29]) analysis may be needed to better understand the characteristics of the noise and whether it should be included in the analysis.

Scaling MSR Techniques to Very Large Repositories.

Most MSR techniques have been demonstrated on large scale software systems. However, the size of data

available for mining continues to grow at a very rapid rate. More intelligent techniques are needed to handle such large amount of data. These techniques must tackle the size and the age of the data. For example, when analyzing historical repositories, techniques should explore assigning more weight to recent data over older data [22].

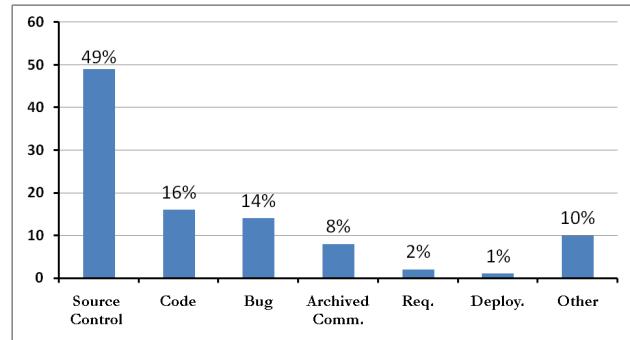


Figure 1. Repositories in MSR publications

Improving the Quality of Repository Data. Researchers should provide guidelines and tools to practitioners who are interested in supporting future mining efforts of their repositories by improving the data entered in their repository. Lexical heuristics used to determine the rationale for a change [49] would not be needed, if the User Interface of source control repository tool would permit a developer to select the purpose of a change from a drop-down menu. Similarly, heuristics to determine the actual change which fixed a particular bug [37] would no longer be required if developers would simply enter the bug-id when committing their bug-fix change.

4.2 Going Beyond Code and Bugs

An analysis of the publications at the MSR venue over the last five years along various types of repositories (*see* Figure 1) shows that a large amount (~80%) of the published work focuses on source code and bug related repositories. We believe that this is primarily due to the fact that the used repositories (e.g., bug repositories or source control repositories) are commonly available and the structured nature of source code and bug reports which eases the analysis. The Figure also shows that documentation repositories (e.g., requirements) are rarely studied primarily due to the limited availability of such repositories. The Figure is derived by examining the full and short papers published in the proceedings of MSR from 2004 to 2008. The Figure counts all types of repositories used by a particular paper. For example, a paper may use data from a source control repository and link it with archived project communication

(e.g., emails), so we mark the paper as using both types of repositories.

MSR research should expand its analysis beyond structured data and beyond data in single repository, while tackling the following challenges.

Exploring Non-Structured Data. Although program data is usually structured, repositories contain unstructured data such as archived communication repositories which contain natural language text [56]. Similarly deployment logs in practice usually do not follow a strict structure [31]. Although Figure 1 shows that there has been a large focus on using source control and bug repositories in papers, their use has shifted in recent years to study other non-code aspects of these repositories such as analyzing the social network of software developers who committed changes or fixed bugs to these repositories. The study of non-structured data in source control repositories and non-structured repositories (e.g., archived communications) continues to rise in popularity with many papers examining the social and technical aspects of software projects. As software engineering researchers, our knowledge and experience with techniques to analyze non-structured data is limited so we should collaborate closely with experienced researchers, such as as social scientists.

Linking Data Between Repositories. A large amount of research uses data in a single repository. However the use and linking of data across different repositories can help in improving the quality of the data [37] and in providing practitioners with a more complete view of the project [12]. Techniques which could accurately link a bug report, to the email discussions about it, to the logs for deployments exhibiting it, to test cases which verify it, and to the actual change which introduced the bug and the change which fixed it would be very valuable in improving the type of analysis done in the field. Research should explore the benefits and challenges of linking data between repositories.

Seeking Non-Traditional Repositories. Figure 1 shows that around 10% of all publications make use of non-traditional repositories. Examples of such repositories are: repositories of build warnings or test results, repositories of programs in large software distributions (e.g., Linux distributions), and IDE interaction history. Some of these repositories, such as IDE interaction history, may not be currently available; while other repositories,e.g., build warnings or test results, may not be commonly available. The work presented at MSR provides a critical analysis of the risks and benefits of building or using such repositories in supporting software research and practice. As research progresses in the MSR field, researchers and practitioners should explore other non-traditional (i.e.,

not-commonly available) repositories and demonstrate the importance of these repositories so others can better understand the value of creating and maintaining such repositories for their projects.

Understanding the Limitations of Repository Data.

Repository data cannot be used to conclude causation instead it can only show correlation. MSR findings must be investigated more closely within the context of the studied project or system. Project and system context are very important to reveal the true cause for particular findings. For example, although an analysis of historical repositories may show that particular developers are more likely to perform buggy changes. This may be due to the fact that they are usually assigned more complex changes and not due to the skill level of these developers.

Moreover, findings may not generalize across projects [53] and the use of repositories varies between different projects. So researchers should closely examine the project culture to better understand the use of the repositories before reaching conclusions. For example, in open source projects a limited number of developers are given commit rights and they commit code for other contributors. Therefore an analysis of the source control repository may incorrectly indicate that there are a small number of contributors to open source projects whereas in reality the number of contributors is much larger [21, 48]. In short, the limitation of repository data should be closely examined and communicated when presenting the results of mining research to avoid the misinterpretation of findings.

4.3 Showing the Value of Repositories

MSR researchers should continue to show and demonstrate the value of data in software repositories and the benefits of MSR techniques in helping practitioners in their daily activities. With practitioners seeing the value of the MSR field, they are more likely to consider adopting MSR techniques in practice. They are also more willing to work on improving the quality of the data stored in software repositories to ease future data extraction efforts and to improve the quality of results of mining efforts. The following are a few of the challenges that MSR researchers must tackle to demonstrate the value of software repositories.

Understanding the Needs of Practitioners. The effectiveness of MSR techniques should be explored relative to the needs of practitioners. We should aim to address problems that are relevant and important to them. For example, although there exists many techniques to predict buggy files in large software projects, the value of such predictions may be low for developers who are usually well-aware of the most buggy parts of their system.

Similarly bug prediction techniques which predict the incidence of bugs at the file level, may be too coarse of a level for practitioners to adopt as they may not add much to their current knowledge. Metrics which could be mapped to actual time and money savings, if possible, are highly desired.

Studying The Performance of Techniques in Practice.

The effectiveness of MSR techniques is often demonstrated using historical repository data. Once these techniques are adopted, it is not clear how these proposed techniques would affect the daily activities of practitioners which in turn would affect the data stored in the repository. For example, research has shown that techniques, which use historical code co-changes to predict other entities which must be co-changed, perform well. However if such techniques are adopted in practice, developers may start relying too much on these techniques to guide them. This high dependence on the technique will affect the historical co-change data which the techniques themselves depend on to perform well.

Showing the Practical Benefit of MSR Techniques.

Researchers must not only demonstrate the statistical benefit and improvement of MSR techniques over traditional techniques. They must also discuss the practical benefit and cost of their techniques. For example, techniques that require a great deal of manual work and maintenance may be hard to adopt in practice, even if they outperform other techniques unless the manual work is a one-time effort.

The performance of most MSR techniques is measured by demonstrating the performance of the techniques at a particular point of time. The maintenance efforts needed to ensure that these techniques keep on performing well is not studied. A technique which mines historical repositories may perform well when first adopted. But as the amount of historical data increases, the technique performance may suffer and the technique may require some calibration. Techniques which can auto-calibrate or which can at least warn users that they need re-calibration are needed. Techniques which require minimal intervention once adopted are highly desirable. Such area of work has not yet to be explored in the MSR field. However, the importance of such work is well-recognized in traditional data mining. For example when mining consumer data, the shift of demographics or consumers perception should be accounted for.

Evaluating Techniques on Non Open Source Systems.

MSR researchers continue to demonstrate their techniques on open source system due to their accessibility and availability. The generalization of findings and techniques to commercial, non-open source systems,

has not been studied. Unfortunately, access to commercial repositories still remains limited.

4.4 Easing the Adoption of MSR

Although the value of MSR techniques may have been demonstrated, there are several challenges preventing the adoption of MSR techniques by practitioners and researchers. Müller and colleagues have over the past few years explored many of the challenges associated with adopting software engineering research in practice (e.g., [6]). We discuss below a few of these challenges from an MSR perspective and we comment on work done in the MSR community to address some of these challenges.

Simplifying Access to Techniques. MSR Researchers should simplify the resources and tools needed for practitioners to experiment and use MSR techniques on their repositories. One option is to integrate MSR techniques into current toolsets instead of establishing MSR-specific toolsets. Fortunately, modern development environments such as Eclipse offer APIs to extend them. HATARI [60], Hipikat [12], Myln [35], and eROSE [68] are examples of MSR research along this direction. These tools integrate within development environments and require minimal effort for practitioners to use. Another option is to consider web service for MSR techniques where repositories can be uploaded for analysis. One possibility is to use a few large open source projects as guinea pigs, to demonstrate the functionality of such services and the benefits of MSR techniques to practitioners.

Helping Practitioners Make Decisions MSR techniques should not aim for full automation instead they should aim to create a synergy between practitioners and MSR techniques. Surprisingly full automation is not always the most desired option for practitioners. Practitioners prefer techniques that support their decision-making process instead of replacing them [11]. Practitioners also prefer simple and easy to understand and rationalize models (e.g., decision tree) over better-performing yet complex models (e.g., genetic algorithms). The simplicity of the models help practitioners in rationalizing the output of MSR techniques, and in gaining buy-in by other shareholders in large projects.

5 Conclusion

Software repositories have traditionally been used for archival purposes. The MSR field has shown that these repositories could be mined to uncover useful patterns and actionable information about software systems and projects. MSR researchers have proposed techniques which augment traditional software engineering data, techniques and tools,

in order to solve important and challenging problems, such as identifying bugs, and reusing code, which practitioners must face and solve on a daily basis.

In this paper, we presented a brief history of the MSR field and discussed several recent achievements and results of using MSR techniques to support software research and practice. We then explored the various opportunities and challenges that lie in the road ahead for this important and emerging field while highlighting work done in the MSR community to address some of these challenges. For up-to-date information about the MSR field, please refer to <http://msrconf.org/>.

Acknowledgements

The author thanks Stephan Diehl, Daniel German, Zhen Ming Jiang, Tom Zimmermann, and Ying Zou for their suggestions on improving the paper and for sharing their thoughts and ideas. The author gratefully acknowledges the contributions of the members of the MSR community for their work and input in helping establish and grow the MSR community as an important part of Software Engineering.

The MSR community as a whole acknowledges the significant contributions from the open source community who assisted our community in understanding and acquiring their valuable software repositories. These repositories were essential in progressing the state of research in the MSR field and Software Engineering.

References

- [1] Summary of Sarbanes-Oxley Act of 2002. <http://www.soxlaw.com/>.
- [2] The PROMISE repository. <http://promisedata.org/>.
- [3] *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society, 2003.
- [4] *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*. IEEE Computer Society, 2007.
- [5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In Osterweil et al. [54], pages 361–370.
- [6] R. Balzer, J. H. Jahnke, M. Litoiu, H. A. Müller, D. B. Smith, M.-A. D. Storey, S. R. Tilley, and K. Wong. 3rd International Workshop on Adoption-centric Software Engineering (ACSE). In ICSE [3], pages 789–790.
- [7] V. R. Basili and B. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984.
- [8] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining Email Social Networks. In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, Shanghai, China, May 2006.
- [9] C. Bird, A. Gourley, P. T. Devanbu, A. Swaminathan, and G. Hsu. Open Borders? Immigration in Open Source Projects. In MSR [4], page 6.
- [10] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code Using CVS Comments. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 364–374, Florence, Italy, 2001.
- [11] J. R. Cordy. Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *IWPC*, pages 196–206. IEEE Computer Society, 2003.
- [12] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Trans. Software Eng.*, 31(6):446–465, 2005.
- [13] S. Diehl, H. Gall, and A. E. Hassan, editors. *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. ACM, 2006.
- [14] S. G. Eick, C. R. Loader, M. D. Long, S. A. V. Wiel, and L. G. Votta. Estimating Software Fault Content Before Coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, Melbourne, Australia, May 1992.
- [15] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft-A Tool For Visualizing Line Oriented Software Statistics. *IEEE Trans. Software Eng.*, 18(11):957–968, 1992.
- [16] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP*, pages 57–72, 2001.
- [17] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
- [18] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *ICSM*, pages 23–. IEEE Computer Society, 2003.
- [19] FLOSSmole. Available online at <http://ossmole.sourceforge.net/>.
- [20] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings of the 14th International Conference on Software Maintenance*, Bethesda, Washington D.C., Nov. 1998.
- [21] D. M. Germán. An Empirical Study Of Fine-Grained Software Modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.
- [22] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.
- [23] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ICSE*, pages 291–301. ACM, 2002.
- [24] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers*. PhD thesis, University of Waterloo, 2004.
- [25] A. E. Hassan and R. C. Holt. Predicting Change Propagation in Software Systems. In *Proceedings of the 20th International Conference on Software Maintenance*, Chicago, USA, Sept. 2004.
- [26] A. E. Hassan and R. C. Holt. Using Development History Sticky Notes to Understand Software Architecture. In *Proceedings of the 12th International Workshop on Program Comprehension*, Bari, Italy, June 2004.
- [27] A. E. Hassan, A. Mockus, R. C. Holt, and P. M. Johnson. Guest Editor’s Introduction: Special Issue on Mining Software Repositories. *IEEE Trans. Software Eng.*, 31(6):426–428, 2005.
- [28] I. Herreraiz, J. M. González-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In *MSR* [4], page 21.
- [29] A. Hindle, D. M. Germán, and R. C. Holt. What Do Large Commits Tell Us?: A Taxonomical Study Of Large Commits. In A. E. Hassan, M. Lanza, and M. W. Godfrey, editors, *MSR*, pages 99–108. ACM, 2008.

- [30] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. Emerald: Software Metrics and Models on the Desktop. *Computer*, 13(5), 1996.
- [31] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann. Automatic Identification of Load Testing Problems. In *Proceedings of the 24th International Conference on Software Maintenance*, Beijing, China, Sept. 2008.
- [32] H. H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.
- [33] H. H. Kagdi, J. I. Maletic, and B. Sharif. Mining software repositories for traceability links. In *ICPC*, pages 145–154. IEEE Computer Society, 2007.
- [34] C. Kapser and M. W. Godfrey. “Cloning Considered Harmful” Considered Harmful. In *WCSE*, pages 19–28. IEEE Computer Society, 2006.
- [35] M. Kersten and G. C. Murphy. Mylar: A Degree-of-interest Model for IDEs. In M. Mezini and P. L. Tarr, editors, *AOSD*, pages 159–168. ACM, 2005.
- [36] S. Kim, T. Zimmermann, M. Kim, A. E. Hassan, A. Mockus, T. Gîrba, M. Pinzger, J. Whitehead, and A. Zeller. TA-RE: An Exchange Language for Mining Software Repositories. In Diehl et al. [13], pages 22–25.
- [37] S. Kim, T. Zimmermann, K. Pan, and J. Whitehead. Automatic Identification of Bug-Introducing Changes. In *ASE*, pages 81–90. IEEE Computer Society, 2006.
- [38] M. Lanza and S. Ducasse. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Trans. Software Eng.*, 29(9):782–795, 2003.
- [39] M. M. Lehman, J. F. Ramil, P. Wernick, D. E. Perry, and W. M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *IEEE METRICS*, pages 20–. IEEE Computer Society, 1997.
- [40] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.
- [41] Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In Wermelinger and Gall [62], pages 306–315.
- [42] LibreSoft tools web site. Available online at <http://tools.libresoft.es/>.
- [43] V. B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns By Mining Software Revision Histories. In Wermelinger and Gall [62], pages 296–305.
- [44] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In T. C. Bressoud and M. F. Kaashoek, editors, *SOSP*, pages 103–116. ACM, 2007.
- [45] H. Malik and A. E. Hassan. Supporting Software Evolution Using Adaptive Change Propagation Heuristics. In *Proceedings of the 24th International Conference on Software Maintenance*, Beijing, China, Sept. 2008.
- [46] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API Jungle. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 48–61. ACM, 2005.
- [47] A. Michail and T. Xie. Helping Users Avoid Bugs in GUI Applications. In Roman et al. [58], pages 107–116.
- [48] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A Case Study of Open Source Software Development: the Apache Server. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 263–272, Limerick, Ireland, June 2000. ACM Press.
- [49] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *ICSM*, pages 120–130, 2000.
- [50] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and Predicting Effort in Software Projects. In *ICSE* [3], pages 274–284.
- [51] A. Mockus, P. Zhang, and P. L. Li. Predictors Of Customer Perceived Software Quality. In Roman et al. [58], pages 225–233.
- [52] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis Of The Efficiency Of Change Metrics And Static Code Attributes For Defect Prediction. In Robby, editor, *ICSE*, pages 181–190. ACM, 2008.
- [53] N. Nagappan, T. Ball, and A. Zeller. Mining Metrics to Predict Component Failures. In Osterweil et al. [54], pages 452–461.
- [54] L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors. *28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, May 20-28, 2006. ACM, 2006.
- [55] D. E. Perry and W. M. Evangelist. An Empirical Study of Software Interface Errors. In *Proceedings of the International Symposium on New Directions in Computing*, pages 32–38, Trondheim, Norway, Aug. 1985.
- [56] P. C. Rigby and A. E. Hassan. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *MSR* [4], page 23.
- [57] G. Robles, J. J. Amor, J. M. González-Barahona, and I. Herreraiz. Evolution and Growth in Large Libre Software Projects. In *IWPSE*, pages 165–174. IEEE Computer Society, 2005.
- [58] G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors. *27th International Conference on Software Engineering (ICSE 2005)*, 15-21 May 2005, St. Louis, Missouri, USA. ACM, 2005.
- [59] J. S. Shirabad. *Supporting Software Maintenance by Mining Software Update Records*. PhD thesis, University of Ottawa, 2003.
- [60] J. Sliwerski, T. Zimmermann, and A. Zeller. HATARI: Raising Risk Awareness. In Wermelinger and Gall [62], pages 107–110.
- [61] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *Proceedings of the 17th USENIX Security Symposium (USENIX Security '08)*, July-August 2008.
- [62] M. Wermelinger and H. Gall, editors. *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, Lisbon, Portugal, September 5-9, 2005. ACM, 2005.
- [63] C. C. Williams and J. K. Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Trans. Software Eng.*, 31(6):466–480, 2005.
- [64] J. Wu, R. C. Holt, and A. E. Hassan. Exploring Software Evolution Using Spectrographs. In *WCSE*, pages 80–89. IEEE Computer Society, 2004.
- [65] T. Xie and J. Pei. MAPO: Mining API Usages From Open Source Repositories. In Diehl et al. [13], pages 54–57.
- [66] A. T. T. Ying, G. C. Murphy, R. T. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.
- [67] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories*, Edinburgh, UK, May 2004.
- [68] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Trans. Software Eng.*, 31(6):429–445, 2005.

Can We Trust Software Repositories?

Andreas Zeller

Abstract

To acquire data for empirical studies, software engineering researchers frequently leverage *software repositories* as data sources. Indeed, version and bug databases contain a wealth of data on how a product came to be. To turn this data into knowledge, though, requires deep insights into the specific process and product; and it requires careful scrutiny of the techniques used to obtain the data. The central challenge of the future will thus be to combine both automatic and manual empirical analysis.

1 Introduction

The idea of engineering is to *use insights to solve problems*—or more precisely, to design and develop solutions by applying scientific, economic, and social knowledge. With software engineering, the goal is the same: We want to acquire and apply established knowledge such that we can craft well-designed solutions to given problems. But what is the “established knowledge” in software engineering? This is a challenge both for researchers, who look out for principles and laws, and for practitioners, who need to apply them to the problem at hand.

The aim of *empirical software engineering* is to systematically acquire laws and theories on software engineering. It works by means of *studies*, whose observations help build hypotheses; and *experiments*, which help to verify, falsify, or establish the validity of such hypotheses. By using scientific methods, we can refine the hypotheses until they become *theories* and *laws* with predictive power for a wide range of software engineering challenges. In their pioneering book from 2003 [3],

A. Zeller (✉)

Software Engineering Chair (Prof. Zeller), Saarland University – Computer Science,
Campus E1 1, 66123 Saarbrücken, Germany, <http://www.st.cs.uni-saarland.de/zeller/>
e-mail: zeller@cs.uni-saarland.de

Endres and Rombach compiled a number of such software engineering laws—rules of wisdom pertaining to any software engineering project. The Compiled laws include *Glass's law*: “Requirement deficiencies are the prime source of project failures”, *Lehman's law*: “A system that is used will be changed”, or *Pareto's law*: “Approximately 80 percent of defects come from 20 percent of modules”.

Of course, these laws do not come out of thin air; each of them is supported by a number of empirical studies. The problem, however, is that conducting such studies can be very costly. Already talking to developers and managers eats into valuable time. Asking them to record data on their doings is worse. And running systematic, repeated experiments to find out how individual factors influence a specific development question is limited by the high hourly wages of the experiment subjects. Finally, there is a goal conflict: Whereas the scientist is interested in general knowledge, his industrial partners want specific answers for their problem at hand. The problem of high cost—combined with little specific gain—seriously hampers the progress of empirical software engineering.

About 10 years ago, researchers discovered a cheap alternative to manually gathering data—namely *mining development data that is already there*. The source of this data is *software repositories*; that is, databases that record events and actions of developers while the software is developed. Such databases had long been used to organize and coordinate the software development process; a version database, for instance, contains every single change to the software ever made—with information on who made the change, when, where, how, and why. Rather than collecting insights from the participants, one could simply collect the data from the repository, by means of simple database queries. However, just having data does not mean having insights; and as much as the field of mining software archives has fueled empirical research all over the place, it is now time to take a critical look at the field—and see how much it constitutes an alternative to classical data gathering.

2 The Sources of Mining

To understand how mining works, let us consider a standard goal: We want to know *where the bugs are*—or to be more precise, which parts of our software had the highest number of bugs in the past. This is a classic setting in mining software repositories, because the defect distribution in a system allows correlating defects with other software properties, and thus making predictions on where future defects will be. To mine such information, the central repositories to be mined are:

Version archives. Version histories track every change ever made to a software module since its creation. The motivation is to track and organize the trail of changes that is being made during development. Each change comes with the text inserted or deleted, the date and author of the change, as well as a *commit message* in which the author motivates the change.

Bug databases. A bug database contains information about every problem ever reported for the software. The motivation is to keep track of pending issues and to

guide development decisions; for instance, a new release could be made when no serious open bugs remain. Each entry comes with a problem description, steps to reproduce, as well as the severity and importance of the bug; as problems are worked on by developers, they also change their status from “open” via “assigned” to “fixed”.

In addition to these main sources, other data sources come in handy. *Automated tests* allow assessing software quality automatically. *Crash databases* are automatically filled with failure data from the field. *Developer databases* allow finding out more about individuals, e.g., relating their experience or team to defect densities. And of course, the *source code* itself is used to determine properties—from simple metrics to full-fledged data-flow or control-flow analysis.

It should be stressed that all of these archives are used on a day-to-day basis for the sole purpose of organizing software development. They are *not* meant to support empirical studies. This is an advantage for studies, as there is no Hawthorne effect; any interaction between researchers and study subjects would take place after the fact. But this is also a disadvantage, as the data is not collected with the aim of being automatically analyzable. This is a challenge—and a risk.

3 The Perils of Mining

For practitioners, just knowing where the bugs have been is already valuable knowledge. The general idea is straightforward: For every module in a system, we analyze its change history from the version database. A *fix* in the change history implies that a bug has been fixed in this very module. The more fixes, the lower the reliability of the module. So far, so good. The central challenge, however, is to tell that *a fix actually is a fix* that addresses a real defect—and not a regular change which adds new features, refactors the code for better maintenance, or adds comments for better readability. Telling fixes from other changes and mapping bugs to changes are central issues in automated mining [4]:

- One might use the *commit message* to tell fixes from changes—for instance, by looking for indicator words such as “bug”, “problem”, or “fix”. This requires that changes come with meaningful commit messages, and that these are organized consistently.
- One might use *bug identifiers* to refer to a bug database. If a commit message refers to a “bug id 38468”, and 38468 is actually a valid entry in the bug database, then one may assume that the change is related to the problem in the bug database—which allows associating it with severity, description, and others. In particular, one may want to differentiate between *pre-production bugs*, i.e., those found during development and in-house testing, and *post-production bugs*, i.e., those found and reported by customers.

The crucial point here is that if the commit message of the change does not allow relating it to a bug, then the change cannot be associated to a particular problem; likewise, there will be problems that are reported as fixed, but cannot be associated

with a particular change that fixed it. In an analysis of the ECLIPSE history, for instance, Bird et al. found that out of 24,119 fixed bugs, only 10,017 could be linked to an entry in the bug database, causing bias in the dataset [1].

But not only can the association of changes and bugs be missing; it can also be overly imprecise. If a developer has a habit of committing all pending local changes before the weekend in one single transaction, then all changes will effectively form one single blob, addressing multiple problems at once, while adding a few changes somewhere else. If this blob of changes is now classified as a fix (because one of the contained changes refers to a bug report, among others), then all contained modules will be marked as having had a defect in the past—even if only one of them was actually defective.

The situation is actually worse because the *data can be wrong to begin with*. In a recent study [5], Kim Herzog and Sascha Just analyzed and reclassified more than 7,000 issue reports into a fixed set of issue report categories clearly distinguishing the kind of maintenance work required to resolve the task. They found that more than 40 % of issue reports are inaccurately classified in the first place, with 33.8 % of all “bug reports” not referring to corrective code maintenance—that is, the bug database misclassified the reports in the first place. Due to such misclassifications, 39 % of files marked as defective actually never had a bug.

Fortunately, there are ways to avoid such issues. First and foremost comes *discipline* in organizing changes and commit messages. If fixes and features end up in separate *branches* of the version control system, they are much less likely to cause confusion; the best situation occurs if each logical fix forms its own set of logical changes (say, its own branch). This is the case in several industrial studies on mining software repositories, in particular those at Microsoft [7] and SAP [6]. Second, one can validate fixes through *tests*: If a test fails on the original version, and passes on the changed version, then the change must be a fix. The *iBugs* repository [2] of fixes and bugs, for instance, is validated this way. All in all, the message is clear:

Automated quantitative analysis should always include human qualitative analysis of the input data.

4 Insights and Correlations

The limitations that noisy source data imposes were always known to the pioneers of mining software archives. As a consequence, their research turned from “findings” to “recommendations”; the general idea was to have tools that first learn patterns and correlations from software archives, and then make recommendations for future decisions—for instance, which modules to focus upon during testing. The term “recommendation” carries a connotation of insecurity, and is just appropriate in light of the noisy data. Furthermore, the given recommendations would always be project-specific, and not necessarily generalize to all software projects. But even so, any such recommendation should be taken with a grain of salt; as the following stories indicate, taking a correlation at face value may be totally misleading:

Fig. 1 The IROP keyboard [8]



- In early work, Thomas Zimmermann and I correlated the average bug density of changes with individual developers, and found that across all ECLIPSE developers, Erich Gamma's contributions had the second-highest bug density. Does this mean that Erich Gamma has low coding skills? On the contrary: As a team lead, he had no one to delegate difficult tasks to; and thus worked on the most failure-prone parts of ECLIPSE. His code was failure-prone, yes—but every other team member would have performed worse.
- In a study with an anonymous client, we correlated bug density and testing coverage, assuming that low testing coverage would result in higher bug density. Interestingly, what we found was the opposite: The higher the coverage, the higher the bug density. The reason was simple: The test managers had a good idea of where to find the bugs, and thus focused their tests on these modules. It is just that despite all the testing, a number of bugs still remained.

In both examples, just relying on the quantitative findings could have led to drastic misinterpretations of the situation at hand. Where human checks are not in place, empirical findings can be trivial to outright misleading; in any case, they will be overblown. To express my disdain of the various “findings” researchers claim to have extracted from open source archives, ignoring the noise that is there in the first place, and ruthlessly overgeneralizing their correlations, Tom Zimmermann, Christian Bird, and I wrote a paper entitled “Failure is a Four-Letter Word” [8], a parody in empirical research, where we report our finding that the letters I, R, O, and P have a stronger correlation to defect density than others in some ECLIPSE data set (which is true), and where our consequences range from rewriting source code to avoid “the letters of failure” to designing a special keyboard where these letters are missing (Fig. 1). Keep this in mind:

Automated quantitative analysis should always include human qualitative analysis of the findings.

5 The Next Big Challenges

So can we trust the data and findings from software repositories? Despite all reservations, my answer is yes. We must be aware of the noise and faults in the originating data, and carefully document our assumptions and processes. When it comes to interpretations, we must be sure to check our findings with project insiders, who will give us insights into what is actually going on. These are standard practices in any empirical research, and the fact that plenty of data is available for automatic processing does not make these practices obsolete. Mining data from software archives is a tool for conducting empirical research. It is a powerful tool, a tool that saves time and other resources, a tool that brings great insights and impact, but it is a tool in the hand of humans.

The biggest challenge in the future will therefore be to further integrate these tools into the portfolio of empirical research techniques, combining both automatic and manual analysis. This requires being aware of the strengths as well as the weaknesses of automatic mining tools; but also to be aware of the many standard practices of empirical research in software engineering. It means being aware of risks due to noisy data and misclassifications; it means being aware of risks such as overfitting and overgeneralization.

The best way to avoid this is to build a vibrant community of empirical researchers who keep on refining and cross-checking data and techniques. The more and better empirical findings we have, the better we can improve the practice of software engineering and make it an engineering discipline worthy of its name; and this is where the legacy of empirical pioneers such as Barry Boehm, Victor Basili, or Dieter Rombach will live on.

References

1. Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P.: Fair and balanced? bias in bug-fix datasets. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE '09, Amsterdam, pp. 121–130. ACM, New York (2009)
2. Dallmeier, V., Zimmermann, T.: Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, Atlanta, pp. 433–436. ACM (2007)
3. Endres, A., Rombach, H.D.: A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories. Addison-Wesley, Harlow (2003)
4. Fischer, M., Pinzger, M., Gall, H.: Populating a release history database from version control and bug tracking systems. In: Proceedings of International Conference on Software Maintenance, ICSM 2003, Amsterdam, pp. 23–32. IEEE (2003)
5. Herzig, K., Just, S., Zeller, A.: It's not a bug, it's a feature: how misclassification impacts bug prediction. In: Proceedings of the International Conference on Software Engineering (ICSE 2013), San Francisco. ACM (2013)
6. Holschuh, T., Pauser, M., Herzig, K., Zimmermann, T., Premraj, R., Zeller, A.: Predicting defects in SAP Java code: an experience report. In: 31st International Conference on Software Engineering-Companion Volume, ICSE-Companion 2009, Vancouver, pp. 172–181. IEEE (2009)

7. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, pp. 452–461. ACM (2006)
8. Zeller, A., Zimmermann, T., Bird, C.: Failure is a four-letter word: a parody in empirical research. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Banff, p. 5. ACM (2011)

USING DATA TO MAKE DECISIONS IN SOFTWARE ENGINEERING: PROVIDING A METHOD TO OUR MADNESS

13

Brendan Murphy*, Jacek Czerwonka†, Laurie Williams‡

*Microsoft Research Cambridge, Cambridge, UK** *Microsoft Corporation, Redmond, WA, USA†* *Department of Computer Science, North Carolina State University, Raleigh, NC, USA‡*

CHAPTER OUTLINE

13.1 Introduction	350
13.2 Short History of Software Engineering Metrics	352
13.3 Establishing Clear Goals	353
13.3.1 Benchmarking	354
13.3.2 Product Goals	355
13.4 Review of Metrics	356
13.4.1 Contextual Metrics	358
13.4.1.1 <i>Product Objective, Audience, and Delivery Method</i>	358
13.4.1.2 <i>Project Scope, History, and Revision Specifics</i>	358
13.4.1.3 <i>Organizational Structure and Expertise</i>	360
13.4.1.4 <i>Development Methodology</i>	360
13.4.2 Constraint Metrics	360
13.4.2.1 <i>Quality</i>	361
13.4.2.2 <i>Performance and scalability</i>	361
13.4.2.3 <i>Compatibility</i>	362
13.4.2.4 <i>Security and Privacy</i>	362
13.4.2.5 <i>Legacy Code</i>	362
13.4.3 Development Metrics	363
13.4.3.1 <i>Code Churn</i>	363
13.4.3.2 <i>Code Velocity</i>	363
13.4.3.3 <i>Complexity</i>	364
13.4.3.4 <i>Dependencies</i>	365
13.4.3.5 <i>Quality</i>	365
13.5 Challenges with Data Analysis on Software Projects	366
13.5.1 Data Collection	366
13.5.1.1 <i>Collection Methods</i>	366

13.5.1.2 Purpose of Used Data Source.....	366
13.5.1.3 Process Semantics	367
13.5.2 Data Interpretation	368
13.5.2.1 Metrics as Indicators.....	368
13.5.2.2 Noisy Data.....	368
13.5.2.3 Gaming	369
13.5.2.4 Outliers.....	369
13.6 Example of Changing Product Development Through the Use of Data	370
13.7 Driving Software Engineering Processes with Data	372
References.....	374

13.1 INTRODUCTION

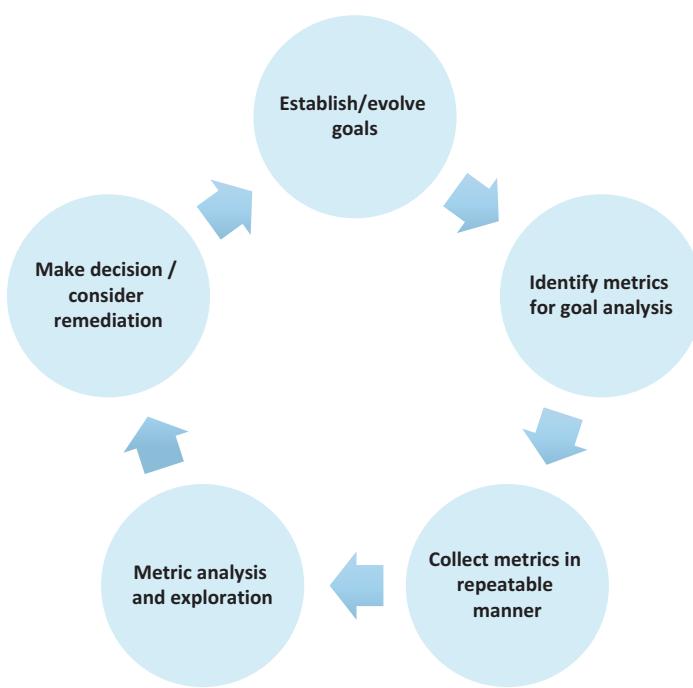
Each day, software development teams make decisions about their development process and the release-readiness of their product. The teams may be facing new challenges, such as the need to release more frequently, or the need to improve product quality. These teams may choose to use a different software process or different software practices to meet these challenges. Have other similar teams developing similar products achieved increased speed and quality by using these practices? Do results to-date suggest objectives will be met? Additionally, teams must decide if the software product is of high enough quality to release. Does the rate of test failures indicate the desired level of reliability has been achieved? And, are the teams actually following the process or actually incorporating the practices that were decided upon?

With the right data, important software engineering decisions can be made through evidence rather than through intuition or by following the latest process or technology trend. This era of “big data” provides a deluge of information. However, data only provides valuable information with carefully planned processing and analysis. Without a plan, the right data may not be collected, the wrong data may be analyzed, the research methods may not provide defensible or insightful results, or undesired comparisons, or inappropriate interpretations may be made.

The goal of this chapter is to aid software engineers and software engineering researchers by providing guidance for establishing a metrics-based decision support program in software engineering. This guidance has three components:

1. accentuating the need for establishing clear goals for the metrics-based program;
2. articulating important components of the metrics-based decision support program, including metrics to record project and product’s context, constraints and development;
3. sharing pitfalls to avoid when collecting and interpreting metrics.

The five major components of an iterative and cyclical metrics-based decision support program are summarized in [Figure 13.1](#). The cyclical process begins with the establishment of goals for the product and development for which the measurement program is being developed. Example software development goals include increased time to product deployment, improved quality, and increased

**FIGURE 13.1**

Metrics-based decision support program.

productivity. Next, the team determines which metrics need to be collected so that analysis can reveal whether the goals have been met or are predicted to be on target. The metrics are then collected. Procedures are documented so that the metrics collection can be repeated on the same product at a different time, or on other projects. The metrics data is then analyzed and explored. Based on the result of the metric analysis the team should determine if the development is on target and if it is not they need to determine if changes are required to the process or if the product goals need adjustment.

In software development, this data analysis and exploration is often called *software analytics* [1]. Through software analytics, insightful and actionable information is folded into the decision-making process. Once decisions are made, the goals of the measurement program can be evolved. The cycle is then repeated. The ultimate goal of such a program is to positively impact the behavior of the team and the success of the project. To illustrate this process, this chapter will provide an example of a Microsoft product group, who re-architected their development process based on data derived from benchmarking previous releases of their product.

We base our guidance on experiences and lessons learned when developing metrics-based decision support programs at Microsoft and when doing research with other companies. The first author previously worked at Digital Corporation monitoring customer systems to characterize hardware and

software reliability [2], and initially continued this work in Microsoft [3, 4]. He then focused his research in understanding the relationship between the way software is developed and its post-release behavior. Both the first and second authors form part of the Empirical Software Engineering [ESE] team in Microsoft Research and have published a substantial body of work in this area. The second author manages CodeMine, a process that is used to collect and analyze the engineering processes used to develop the majority of Microsoft products [5]. Finally, the third author has conducted research on the use of agile process and agile practices at Microsoft, IBM, Telelec, SAS, Sabre Airlines [6–10], and other companies.

This chapter describes the process, as depicted in [Figure 13.1](#), through which practitioners can use data to drive improvements in their software engineering processes. The key area that practitioners should focus on is the establishment of clear goals. Often, teams, at best, have aspirations of what they would like to achieve through their development process. Establishing metrics that can be repeatedly collected, and the careful analysis of the metrics can help teams understand if they have achieved their goals.

The rest of this chapter proceeds as follows: [Section 13.2](#) provides information on a short history in the area of software metrics in software engineering. [Section 13.3](#) discusses the need to provide clear goals for the project. [Section 13.4](#) provides information on the components of a metrics-based program. [Section 13.5](#) identifies the pitfalls to avoid in choosing metrics and information about interpretation challenges, respectively. [Section 13.6](#) provides an example of how the Windows Server product team is re-architecting their development environment based on a metric-based decision support program. [Section 13.7](#) shows how to use the processes described in this chapter to allow product teams to use data to drive their engineering process.

13.2 SHORT HISTORY OF SOFTWARE ENGINEERING METRICS

Software engineering differs from other engineering disciplines by its lack of a standardized set of metrics to manage projects. Other engineering disciplines have standardized processes and practices that have evolved over time. For instance, the railway industry standardized their total process for building and managing trains over the last 150 years [11].

Research into the relationship between metrics and software reliability can be traced back to the 1970s. John Musa took an empirical approach in his data collection works [12], whereas other researchers built reliability models based on the software structure [13]. The computer industry initially focused on hardware failures using reliability metrics, such as Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) [14, 15]. In the 1980s, software was identified as increasingly impacting overall product reliability. Additionally, [16] identified that software reliability is also influenced by human factors.

In the late 1980s, a number of major computer manufacturers started providing real-time monitoring of their computer systems at the customer sites. This monitoring allowed the total behavior of the computers to be analyzed. Digital Equipment Corp identified that software was becoming the dominant factor impacting the overall system reliability, and its reliability could no longer be measured only in terms of system crash rates [2, 17]. Similarly, IBM analysts attempted to characterize software quality

and productivity [18]. Barry Boehm built a model to analyze the economics of software development [19], while Fred Brooks highlighted the difficulty of bringing out-of-control software development projects back under control, based upon his experiences with the IBM System/360 [20]. Similarly Lehman and Belady [21] analyzed the impact of program evolution, from which Manny Lehman derived the “Lehman’s laws of software evolution” [22], which assist in our understanding of how products evolve over time.

In the 1990s, in some areas, the perceived lack of standards was assumed to result in catastrophic failures as the new millennium approached. One solution was to apply formal development methods to control the software development process through the use of guidelines and standards, such as ISO 9000 Standard [23] or those promoted by the Software Engineering Institute [24]. During this same timeframe, Norman Fenton produced a comprehensive book on software metrics [25], and other authors focused on the analysis of software development methods [26].

At the same time, the 1990’s saw the emergence of the Open Source community producing reliable software, such as Linux and Apache, without the use of these formal methods. The turn of the millennium (Y2K) came and went without catastrophic events. A backlash against the overemphasis of processes in software development resulted in a group of software engineering consultants publishing the *Manifesto for Agile Software Development* [27].

In this millennium, researchers in software engineering have performed numerous studies linking software metrics to post-release failures. For example, Li [28] examined factors that led to system and application unreliability. Mockus [29] discovered the effects of organizational volatility on software reliability, and Zimmermann [30] performed similar studies to identify the causes of security defects in released products. Cataldo [31] found that coordination breakdowns negatively impacted development productivity and software failures. Finally, Zimmermann [32] predicted defects using network analysis on dependency graphs. This sampling of studies found relationships between attributes of the development process and post-release failures for a specific set of products. However, researchers often have difficulty in replicating results on different product sets, suggesting the product and project context influence the metrics. Some practitioners have analyzed which metrics are applicable to specific coding practices, such as Lanza and Marinescu [33] providing assistance in developing object-oriented code.

This chapter incorporates a lot of the learning gained from all of this historical work to improve the effectiveness of managing software development through metrics; and provides a practical approach to this subject.

13.3 ESTABLISHING CLEAR GOALS

The development of any engineering project requires a clear set of goals to provide a focus for the engineers and a mechanism to ensure the development is on target. At times, the goals may change during the development period. Other engineering disciplines may have an established set of metrics that can be used to determine if the overall product goals have been met. For instance, engineers building a bridge can decide on materials and the design of the bridge, but their usage must exist within defined bounds that are verified through a set of metrics and mathematical equations.

With a plethora of available data, the Goal-Question-Metric (GQM) [34] methodology can be used to guide the development of a metrics-based decision support program, as shown in [Figure 13.1](#). GQM is focused, whereby only the data needed to determine if a goal has been met is processed and analyzed. The GQM approach forms the basis for any organization to measure its attainment of goals in a purposeful way. The first step is to build a *goal* statement. Then, a set of *questions* is developed that need to be answered to gain an understanding of whether the goal has been met. Finally, the *metrics* that need to be collected to answer the questions are enumerated. A set of data is associated with every question, such that the question can be answered in a quantitative way. In this way, the organization traces the goals to the needed data to define those goals operationally, and finally provides a framework for interpreting the data with respect to the stated goals. Other available data not required to answer the questions is ignored, even though it may intuitively look interesting.

The choice of any metric depends on the relevance of the metric to the attainment of product goals and also its accuracy. The accuracy of metrics, in turn, depends on how the data is collected, which often relies upon the tools and practices of the engineering teams. An effective way to establish the metrics and target values for those metrics is to benchmark an equivalent product, as discussed in [Section 13.3.1](#). The benchmark provides a framework for collecting and interpreting metrics for the new product in the relevant context and provides a point of comparison [35]. Re-using metrics from previous projects will provide the analyst some confidence that the metrics accurately capture their development process. Once the metrics are defined, a project-specific criteria for metrics can be established, which more closely relate to the specific product goals.

This section describes the methods of obtaining benchmarking data and how that benchmarking data can then be used in interpreting the metrics. The section also describes the methods for quantifying the goals of the project.

13.3.1 BENCHMARKING

Many software products are either a new version of an existing system or derivatives of other products. The objective of the benchmarking process is to:

1. collect the metrics generated during the product development;
2. process the metrics to identify missing data and outliers;
3. ensure the metrics can be correctly interpreted.

Outside of major refactoring activities, which are typically evident, a new version of an existing product will rarely make a significant change to the core attributes of the product; especially as they relate to the product's architecture. Ideally, the products that are to be used as a benchmark should have developed using equivalent tools to the target product. If possible, metrics should be collected from more than one single product release. Product teams should not overly focus on the metrics from a problematic release. Rather, the objective should be to identify the normal value for a metric, because bad releases may only provide outlier values for a metric. Additionally, a team can consider the baseline being a product with a good reputation for quality and customer acceptance.

More generally, the same collection and analysis methods that were applied to previous versions of the product should be used for data collection and analysis of the release under study. Therefore, the methods for data collection and analysis should be clearly defined to enable replication. Additionally,

the team interpreting the metrics should have knowledge of the data collection tools and their idiosyncrasies. The metrics collected from the prior release should be analyzed and an initial interpretation of the metrics should be produced. As part of this interpretation, the team should also focus on comparing metrics across the development cycle analyzing differences in trends to identify inaccuracies of the metrics or possible gaming of the system.

The creation of benchmarks for products that are significantly different from prior versions is more complex, and is dependent on the percentage of the new product that has been derived from the prior version. The smaller the percentage of derivation, the less applicable it is to use the prior development as a benchmark for the new product. But even in these circumstances, the benchmarking process can provide bounds for the metrics, whereby when metrics move outside of these bounds the product group should initiate investigations to identify if any potential issues exist.

The analysis of the metrics should be shared with the team that developed the product to clarify the discrepancies between the interpreted metrics and the product team's perception. During these discussions, the software engineers will often identify why the metrics may not be accurate. The team may reveal sources of noise in the data, gaming of the system, sources of inaccuracy, or outliers.

To illustrate the point of the relevance of choosing appropriate products to provide benchmarking data, we can consider how the Microsoft Bing team would choose a relevant benchmark. Bing is a service product that is continually deploying new features onto a single deployment target. Consequently, Bing does not support older versions, and it exposes its functionality through web servers. Alternatively, each release of Windows can be separated by years and it is deployed onto millions of computers whose failure profile is very different from Bing. So while the different product groups will share experiences, they would never attempt to benchmark their developments against each other.

Within a single product group, not all product releases may be used as benchmarks for other releases. For example, while the development tools and process are common for a Windows product and its Service Packs, neither would be a valid benchmark for the other. The focus of a main Windows release is to provide a new user experience, whereas for Service Packs the focus is on maintaining the current experience while correcting security or reliability issues. Therefore Windows 8.1 would use Windows 8 as a benchmark, whereas Windows 8.1 Service Pack 1 would use Windows 8 Service Pack 1 as a benchmark.

The phase of the product's development lifecycle should also be considered. For a product such as Windows, the development cycle is split into milestones that consist of periods of feature development and periods of stabilization. The expected product behavior should change between these milestones. In these circumstances, the development performance during Windows 8.1 feature development milestone would be compared against the equivalent feature development milestone in Windows 8.

13.3.2 PRODUCT GOALS

Once a clear relationship is established between the metrics and the development attributes of the previous release, the metric goals can be defined for the future products. The quantified metric targets are estimated based on the data calculated from benchmarks and the overall goals of the next release. Specifically, if the objective of the next release is to improve performance, then the goal could be a specified performance percentage increase over the previous release.

The goals should be realistic, which often takes considerable experience, judgment and restraint. For a complex product, improving all product attributes simultaneously is difficult. So, if the goal is to improve performance, then the goal for reliability may be to maintain the current level. The goals should also reflect any changes in development methodology.

A common mistake is to define goals at too detailed a level. For instance, if the goal is to improve productivity in terms of rate of code changes or feature added, defining that goal in terms of the increased productivity expected by individual engineers is discouraged. By breaking the goals down to the individual level, ownership of the goal from the teams is reduced, which removes the motivation of the team to innovate (e.g., assigning engineers to improve tools may be far more effective than asking them to code faster) and to work together as a team. The preferred way is to define and monitor the goal at the product level. If the product is not meeting the goals, then the causes of the issue should be investigated. The solution may be improved tools, sharing of best practices between teams or identifying that the goal itself is unachievable and needs to be reset.

13.4 REVIEW OF METRICS

This section provides the details of the metrics that are available to assist teams in the characterization of their software development process. Since the collection and analysis of metrics is a cost overhead for product teams, the objectives of the metrics must be to provide the developers with useful information that is not otherwise available to them. A small development team of five people may not need to collect daily metrics to know the amount of effort and bug fixes that occurred over the last few weeks. Their own experience would provide that knowledge. However, they may like to see trends over a period of time to monitor improvements. Conversely, a large product, such as Windows, benefits from detailed daily and weekly metrics, because no individual would have the knowledge of the amount and type of effort occurring in the entirety of its development. As the objective of the process is to use metrics to determine the status of a development, then the number and type of metrics that are required should be determined by the size and complexity of the product. Smaller teams and simpler products require fewer metrics to understand the product status.

The following should be considered in developing a metric framework for a product's development:

1. *The product's characteristics.* Is the product stand-alone, an application that is deployed on multiple platforms, or a service product? What is the objective of the next release of the product? For example, is the release focused on introducing new features or is the release intended to address customer feedback from prior release? If the objective is to add new features to a product, then the product goals can be complex, especially in regard to quality. The introduction of new features can change the product's user experience for the customers. Whereas, if the objective of a release is to improve reliability, reliability measures should be more precisely defined.
2. *The development process.* The chosen development process controls how the engineers behave, which has an obvious impact on interpretation of the metrics. One team may decide that features must be complete prior to integration into the master version of the product or component. For these teams, integrated features should be complete. Other teams may opt for continuous integration and will allow unfinished features to be submitted. For these teams, burst of changes until the product stabilizes is expected. Within Microsoft, and in most organizations, changes

being submitted into the master version of the product should not break that product in fundamental ways, often referred to as “not breaking the build”. Often, teams opt for verification processes prior to checking into the trunk branch to prevent such build breakage. Quality metrics should reflect the verification processes available to the developers. If the developers are not able to perform a “system test” on a change, a higher failure rate for these changes would be predicted.

3. *The deployment and support characteristics.* Nowadays, product can be deployed via many different mediums. Some types of software are installed to be executed locally. Externally managed channels, such as app stores, may help with deploying such applications, or they can be deployed completely independently (such as via downloaded or shrink-wrapped packages). Other software can be shipped as a component of a larger product. In this case, the developer has much less control over the deployment and the usage scenario of the software. Deployment of the software can occur through a browser reaching a web service. In this case, software engineers have full control of their deployment and servicing of the product. Finally, some software might be deployed by multiple methods simultaneously.

All metrics collected and analyzed should be tied to a specified product goal, as discussed in [Section 13.3](#). We grouped metrics into three classes, each with a different purpose. This grouping is an extension of the work of Williams et al. [20] when establishing the Extreme Programming Evaluation Framework. Product teams should aim at understanding all these aspects of the project and product characteristics to be able to make informed decisions regarding the way they develop software. These three classes are as follows:

1. *Contextual metrics.* Drawing conclusions from empirical studies of teams may be difficult because the results of any process largely depend upon the relevant context variables. One cannot assume *a priori* that a study’s results generalize beyond the specific environment in which it was conducted [36]. Therefore, recording an experiment’s context factors is essential for comparison purposes and for fully understanding the generality and utility of the conclusions and the similarities and differences between the case study and one’s own environment. Factors such as team size, project size, criticality, and staff experience can help explain differences in the results of applying the practices. Product teams should look back at the project history as it creates a framework for assessing future performance. Organizational structure and relevant product expertise permeate the project and heavily influence the product and these also need to be understood.
2. *Constraint metrics.* While the goals of the product under development may be to release a set of new features, each release must meet a series of constraints. The majority of software products have to work under predefined constraints. For instance, applications distributed through an app store have to satisfy requirements of the particular store. Constraints can be related to past releases of products. For example, a new release may be required to maintain the same level of performance or reliability. Products may be required to ensure backward compatibility or security or privacy constraints. For instance, the new version of the product is required to improve the performance of data retrieval by 10%.
3. *Development metrics.* Assessing the state of development, verification, and deployment readiness and tracking these over time is an important part of managing a project. Is the development progressing according to plan? Is the progression rate higher or lower than in the past? How much time is spent in verification as opposed to design and implementation? These types of metrics can

be used to assess the achievement of a goal, such as the code should decrease in complexity by 20%.

This section discusses a set of metrics that individually can address some aspects of the above characteristics. Later parts of the document will discuss methods and challenges often occurring when interpreting these metrics. [Table 13.1](#) provides an overview of the metrics discussed in this chapter, identifying the metric category, providing specific examples of the metrics, their goals.

13.4.1 CONTEXTUAL METRICS

The developments of software projects are varied in terms of scope, resources, and time commitments, and the resulting products have widely differing functionalities, intended audiences and lifecycles. An operating system that is deployed on servers will be, by necessity, differently scoped, produced and verified than a web service or a single-function application on a consumer-class device. The product and project context information explicitly describes some of the key attributes which later assist in identifying relevant products as benchmarks and in the interpretation of the metrics.

Among the most crucial context details are the project objectives, intended audience and the mode of deployment, project history and revision specifics, organizational structure and knowledge, and applied development methodology.

13.4.1.1 *Product objective, audience, and delivery method*

Understanding the project and the product context begins with business goals. These inform decisions around what value the product will provide, who the product is intended for, and how the functionality will be delivered and presented to the user. The differences between traditional software packages that need to be installed versus software exposing the functionality through a browser, are profound, and will require adjustments to the development methodology and the analyst's understanding of the data.

The output of this information will result in the adjustments of other metrics. For example, if a product moves from a traditional software package to a service, then that often results in increases in the frequency of releases.

13.4.1.2 *Project scope, history, and revision specifics*

Project size and scope can be measured in multiple ways, starting from counting source code artifacts, executables or higher-level functionality through a measure, such as function points. Unless the product is brand new, the product history is useful for putting metrics into context. History, moreover, provides the means of allowing for more precise analysis. The scope and timing of a release, put in context by the objectives and timing of the past releases, determine how relevant measurements are to understanding the project today. On the other hand, if past releases were focused on delivering new functionality and the current release is focused on improving reliability, historical comparisons might be less useful, although the past release will define the baseline for the minimum improvement in quality required for the current release.

In addition, previous releases at least partially determine the scope of the current release. The scope might take a form of addressing users' feedback from, or quality issues found in previous releases. Lastly, product history is invaluable in understanding persistent strong and weak points of the product; areas of concern across multiple releases in the product and in the development methods; and the levels

Table 13.1 Selected Product Metrics

Metric	Category	Example Metric	Goals	Challenges
Product objectives	Contextual	Product releases every month	Relate metrics to product goals	Each product may have unique characteristics
Product revision	Contextual	Service-based release	Assists in setting realistic development goals	Each release may have unique characteristics
Organizational structure	Contextual	% of experienced engineers	Ensure development teams have correct balance of experience	Sometimes this is not within the control of the product team
Development methodology	Contextual	Code velocity	Identify the most relevant product metrics	Product teams may use a variety of development processes
Quality	Constraint	MTBF	To improve the end-user experience	Some aspects of quality such as usage and visualization are difficult to measure
Performance	Constraint	Average response time on specific configurations	To understand the end-user performance	Performance is dependent upon a lot of environmental factors, such as network speed, outside product control
Compatibility	Constraint	Product is compatible with applications released since the previous version	Upgrading the software does not break compatibility	Difficult to emulate all the user's environments
Security and privacy	Constraint	Development complies with industry standards	Improve product security	Security threat landscape is constantly evolving making it less amenable to quantification
Legacy code	Constraint	Code age	Minimize technical debt	Can impact compatibility with older software
Code churn	Development	Files changed per week	Verify development activity match plan	Tracking only the code that forms the released product
Code velocity	Development	Time for feature completion	Increasing productivity	Difficult to interpret
Complexity	Development	McCabe's complexity	Minimize product complexity	Unclear what action to take based on McCabe's metrics
Dependency	Development	Count of direct dependencies on a module	Minimize product complexity	Tools do not calculate state dependencies
Quality	Development	Number of open defects	Improve product quality	Metrics are often gamed and not reflective of actual product quality

and concentrations points for accumulated technical debt. Technical debt is a metaphor for incomplete or inadequate artifacts in the software development lifecycle [37].

Data from prior releases can be used to determine the percentage change in the metric in a new release. For instance, if the goal is to improve the quality of the product, then the reliability goals can be set at a specified percentage improvement over a prior release. The previous product release may have focused on providing new functionality, and the current release is focused on addressing technical debt (e.g., a Service Pack release), then you would expect the current release would be more reliable than the previous release, without any changes to the development process.

13.4.1.3 Organizational structure and expertise

The organizational structure and the expertise of the team play a pivotal role in the success or failure of any project and the long-term viability of the product. Further, the architecture of the product is typically related to the organizational structure that created it; often referred to as Conway's law [38].

Understanding the organizational context starts from identifying the team's size, structure, defined and implied roles and responsibilities and, if possible, lines of communication. Distributed teams will work differently than co-located teams and will have different patterns of decision-making and knowledge sharing. Conflicts within teams may indicate goal- and role-definition problems, which often will translate into architectural, design, or quality issues later in the product lifecycle [39].

Finally, mapping out the longevity and churn of team members on the project is an indicator of retained organizational knowledge and expertise. Useful metrics in this space are the average level of experience per team. New people joining can bring new ideas and experiences to the teams but teams also benefit from retaining product expertise and experience. Another useful metric to track is the turnover of all the teams. Excessive turnover may indicate morale problems in the team.

13.4.1.4 Development methodology

The development methodology chosen by the team, and applied to developing the product, has an impact on the goals of the project, especially when changes to the methodology occur during or between product releases. Methodologies most often change in response to a project failure, inability to quickly respond to new requirements, quality issues with past releases, or frustration with the existing development workflow by the engineering team. If a team starts a project using deep branching hierarchy, then they may be focused on metrics, such as code velocity across branches. If the team then moves to a more agile approach during system verification, then code integration velocity may no longer be of prime interest to the team.

13.4.2 CONSTRAINT METRICS

During product development, the focus for engineers is the implementation of new features or the fixing or improvement of existing functionality. Changes to the software are verified through a number of different processes, such as code reviews, unit tests, static analysis, component verification, system verification, deployments, usability studies, or acceptance tests. The objective of the verification process is to ensure the change meets its goals by:

1. satisfying its functional requirements,
2. satisfying any constraints placed on the features by the system and the environment.

While functional requirements are product- and feature-specific and largely self-explanatory, constraint metrics focus on the system constraints. Typical constraints found in today's software relate to reliability, compatibility, security, standards and regulatory compliance, privacy, accessibility, UI guidelines, physical limitations, power consumption, and performance. Project goals should explicitly enumerate all the constraints the final product should meet, but often at least some of the constraints remain implicit. Users only perceive software meeting all its explicit and implicit constraints as high quality.

The constraints that a feature has to satisfy vary based on the product. A feature developed for a mobile application and for the kernel of an operating system would have to comply with completely different sets of constraints.

Lastly, all constraints can be expressed as absolute or relative. The former often have a form of a precise statement, creating a target for the engineers, for example, the start-up time of the software from a user initiating its launch to having the user interface ready to accept inputs should be less than 3 seconds. The latter is typically relative to the previous release or a competing product, such as to meet the value for the constraint in a prior release. For instance, a product's Service Pack may be required to be compatible with all the same applications as the original product.

13.4.2.1 Quality

The quality of the released product is dependent upon the ability of the product to match the needs of the customer. As such, quality is a difficult characteristic to measure. Often, product groups try to translate quality into the reliability attributes that are readily noticeable by end-users, which is translated into reliability constraints. In addition, reliability can also be characterized as the lack of availability of the product. Reliability targets and reliability trend monitoring are typically easy to establish, although care should be given to ensuring the target is appropriate for the scenario for which the product was designed. The traditional primary metric for measuring reliability is the mean time to failure (MTTF), which is still appropriate for most of the software written today. The secondary metric of mean time to recovery (MTTR) is often useful as a complement to the MTTF metric.

Depending on the nature of the product, increased MTTF may be traded off for decreased MTTR. For systems processing non-critical data where a failure will not corrupt any data, speed of recovery (MTTR) is more important to perceived quality than encountering a failure (MTTF). A system processing critical data would prioritize MTTF and the need to guarantee correctness of the data more than the MTTR.

13.4.2.2 Performance and scalability

Performance or scalability constraints relate to resource consumption, both on the part of the software (e.g., ability to hold sufficient number of objects in its data structure) or the user (e.g., time spent waiting for an operation to complete). In certain product areas, such as communications, performance goals may be specified both in normal state and in a degraded state. In other software areas, performance and scalability is dependent on multiple factors, including those that the software cannot directly control, such as environment on which it runs. Performance and scalability metrics are typically expressed as distributions over a representative population of environments, for example, the response rate from the website for a user with a broadband connection should be between 1 and 2 seconds. The consequence of such representation is that the question of whether a product meets the constraints is subject to interpretation. At the very least, the boundary between meeting and not meeting a performance and

scalability constraint is often soft. Performance metrics are often based around the performance of the software on representative equipment with representative user load, although some software, such as communications, often specify allowable performance degradation based on poor environments.

13.4.2.3 Compatibility

Compatibility relates to how the product interacts with its environment. The most prevalent case of having the ability to use data (and sometimes programs, for example, scripts) from its own previous versions is often referred to as providing backwards compatibility. Compatibility constraints come in a wide range depending on how interoperable the product needs to be with its environment and how big an installed-base it needs to support. In principle, the problem is confined to finding a representative set of configurations needing to maintain compatibility. In practice, fully establishing whether this constraint is met depends on how difficult it is to validate various combinations of the product interoperating with other software or its own previous versions. When such a representative validation set is established, the answer to how compatible the product is in relation to its requirements is straightforward.

The compatibility metrics often fall into two categories. One category is the type of equipment that can run the software and the other is the versions of the application that can interface with the software. Examples of equipment metrics are the software runs on computers running Windows 7 or above and have at least 1GB of memory. Examples of application compatibility are that the software must be compatible with browsers running at or above a specific version.

13.4.2.4 Security and privacy

Security of systems is another aspect of the product requirement set that becomes a constraint. Various ways of probing the security boundary of the system have been established over the years. Starting from performing a comprehensive analysis of threats, through analysis of entry points and paths through the software statically or through testing (including security focused fuzzing of inputs), to targeted penetration testing.

The security and privacy requirements of a software system must also comply within a legal system and laws governing the customer's location and, in case of companies, its line of business. The software has obligations for compliance with various policies, such as user data privacy, requirements around data storage and transmission, defined by the laws of the country (e.g., European Union (EU) data privacy laws).

Security metrics are often related to the process that product groups use to minimize the risk of security holes, rather than product goals. Examples of these metrics are whether the product groups follow the security development lifecycle (SDL). Privacy goals often relate to defining what types of data is classified as privacy related and how that data should be stored and accessed.

13.4.2.5 Legacy code

Legacy code is the amount of "old" code in a code base as compared with new code. Legacy metrics provide measurement in terms of age of the code or in terms of in which previous products did the code exist, depending on the definition of age. Engineers may use this to identify areas of accumulated technical debt, ripe for refactoring as old code is often not really well understood by current team members and a lack of expertise in such parts of the codebase may represent future risk.

Examples of legacy metrics are the percentage of the current code base that existed in prior releases of the product.

13.4.3 DEVELOPMENT METRICS

Development metrics capture the status of the product across its development cycle over time. These metrics are used by the development team to identify bottlenecks and problems with the development process. Different product development models will place different emphasis on different areas of its development. Therefore, the development metrics are organized into the categories below for different characteristics of the development process.

13.4.3.1 *Code churn*

The code churn occurring on a single file is defined as the lines added, modified, or deleted between two revisions of a file. These metrics are most often obtained through the use of textual diff'ing tools, which provide a convenient way for engineers to track and understand changes occurring at a level of a source line. Calculating churn is a lot easier when files are managed by a version control system (VCS), as the majority of VCSs have built-in diff tools.

For any type of software, a released product is comprised primarily of files that form the deployable artifacts; in addition, there can exist secondary software to perform the act of deployment or other process management functions. A project team may additionally store files that perform related tasks, such as testing or build process. These files need to be classified into their own category, as the churn of these files should be analyzed separately from the churn of the files that form the final deployed product. Monitoring the churn of these non-product related files and their correlations with testing efforts and product churn can be informative. If a correlation does not exist, the new features may not have been tested at the right level.

Numerous research has studied the relationship between code churn and reliability and through combining code churn with other progressive metrics. Nagappan [40] showed that characteristics of code churn can have a negative impact on overall system reliability.

The churn rate over the product lifecycle should correspond with the product lifecycle phase. For example, during feature development, the churn should consist of large changes while during stabilization, changes should be primarily smaller changes and bug fixes. Tracking churn provides a method to identify if the development process matches the plan. It is also invaluable to correlate where churn is occurring against where churn is planned to occur. A lot of churn happening in unplanned areas may be an indicator of problematic code, that is, code that continually requires patching. In these scenarios, the code should be investigated to identify if it requires to be refactored.

Code churn metrics are based on the number of changes that are occurring within specific time periods or within specific releases. The types of changes that are monitored are lines changed or files changed. For products with long development cycles the amount of churn is often tracked over time periods, such as a week. For short release cycles, such as services where a release could be weekly or monthly, the churn metric is the change occurring continuously within a release.

13.4.3.2 *Code velocity*

Code velocity is a term encompassing measures of efficiency of the development process, characterizing the time from the development of a new feature to its ultimate deployment to the end-user. In all product developments within Microsoft there is an intermediary stage, where the completed feature is merged into the master version of the product or component, this is traditionally managed within a trunk branch in the version control system. The applicability of the code velocity metrics assumes that the development process can be categorized into three phases:

1. feature development, this phase often includes unit testing;
2. system verification and the merging the feature into the trunk branch;
3. deployment of the feature to the end-user.

Based on these development phases the code velocity metrics can be deconstructed into the following:

1. *Implementation velocity*: The time between a team starting to write a feature and the feature being merged into the trunk branch. This does not include the time needed for clarification of requirements, designing a feature and performing design verification, as accurately measuring these time periods is difficult. This metric includes the time for the implementation and verification of the feature.
2. *Integration velocity*: The time from feature completion to the feature being integrated into the trunk branch. This metric is more relevant for teams that develop software within a branch structure and measures the time for the feature to move through intermediary branches, before being integrated into the trunk branch.
3. *Deployment velocity*: The time between a feature being integrated into the trunk branch to deploying that feature into a running service or into the shipped product.

To provide an example for these metrics we consider a feature being developed in its own feature branch and the master version of the product being managed in the trunk branch of a source tree. A team starts working on implementation at time T_1 ; the feature is verified and ready for merging into the trunk branch at time T_2 . After going through an integration and verification process, the feature is merged into the trunk branch at time T_3 , and deployed to the end-user at time T_4 . The metrics characterizing the code movement are:

$$\text{Implementation velocity} = T_2 - T_1$$

$$\text{Integration velocity} = T_3 - T_2$$

$$\text{Deployment velocity} = T_4 - T_3$$

The appropriateness of the code velocity metrics is dependent upon the objective of the development process, for example, for a product such as Windows, the whole product is deployed yearly so measuring the deployment velocity for an individual feature is irrelevant, whereas for service products, features can be continuously deployed, and so the time for feature deployment is important.

If the code velocity of a development is slower than a team's goal (derived from past developments) a possible bottleneck may exist in the process or quality issues may be present necessitating additional verification and bug fixes that will slow the code flow into its trunk and subsequent deployment.

13.4.3.3 Complexity

The original complexity metric was defined by McCabe's and was defined as the number of branches in the flow of code at the level of a procedure. While this is accepted as a good metric for calculating the testability of a product, one complaint against this particular metric is that it does not handle loops very well.

Research results conflict relating to the importance of the complexity metrics, some indicating that the metric is correlated with failures [41] and others that do not find any such relationship. Additionally, measuring the total complexity of a product does not differentiate between legacy code and new code, and therefore is not an actionable metric.

A more relevant measure is the amount of change in complexity that is occurring over the project development. This metric is more actionable as it can be used to identify whether the increase in complexity matches planned changes to the product and helps detect mismatches between the intended design and implementation. This metric can also identify areas that are increasing in complexity or decreasing in their testability.

13.4.3.4 Dependencies

Dependency metrics are measures of architectural complexity. They typically count the number of artifacts that an artifact in question is dependent upon to perform its functionality. Artifacts can be binaries, components, modules, functions. In addition, dependencies can be measured in terms of direct dependencies (i.e., a binary calls a method or function of another binary) or indirect dependencies (a binary makes a call to another through an intermediary).

Engineers often analyze the dependency of the program at the start of the product development to identify areas of dependency complexity, which may require refactoring. Once the product development begins, it is important to track and characterize the changes in dependencies that occur over time. These changes should reflect the product plans. Areas of code that are increasing in complexity where no added functionality is planned may indicate issues.

Examples of dependency metrics are measures of the number of dependencies a component or binary is taking or has on them. Other useful measurements are identifying the total depth of dependencies in the program and the existence of cyclic dependencies.

13.4.3.5 Quality

Various in-process metrics, such as the number of defects found, tests executed and passed, achieved code coverage, or uptime under stress, are used as proxies for quality during development. Ultimately though, the level of quality of any software system is fully manifest in real world usage. Therefore, quality is most often equivalent to counting defects filed and/or fixed after release. Defects filed and fixed are the most common metrics collected by product groups but are among the most difficult metrics to correctly interpret. Issues with noise in defect data notwithstanding, some measures of overall quality collected from defect data are possible and often applied. These include:

1. the number of defects found during a period of time,
2. the number of defects resolved during a period of time,
3. the number of defects resolved through making a change to the product,
4. the number of defects that remain unsolved,
5. product availability under representative user load,
6. percentage of test executions succeeding.

Such metrics are often broken down by severity to give more weight to more urgent and important issues. These defect trends are quite often used as a proxy for determining the product's readiness to release.

A common practice is for engineers to use the product as it is being developed. This practice is often called “Eating your own dog food” or “dogfooding,” for short. In these scenarios, it is possible to measure changes in end-user quality over the product lifecycle, such as system failure rates (e.g., MTTF), system availability (e.g., MTTR).

13.5 CHALLENGES WITH DATA ANALYSIS ON SOFTWARE PROJECTS

The greatest challenge for product groups is not collecting data for metrics but in their interpretation. For product groups to interpret the metrics to determine the current status of the product development, it is important to take into account how the data was collected to create the metric but also to understand and counteract common interpretation issues.

This section addresses both of these issues related to managing a project through metrics.

13.5.1 DATA COLLECTION

As described in [Section 13.3](#), characterizing the software development process through metrics begins with goals, defining the specific questions that will be asked, and defining metrics appropriate for answering the questions. Once the objectives and definitions are established, typically the next step is the collection of data.

13.5.1.1 *Collection methods*

Data can be collected manually or automatically. Manual data collection takes a form of surveys, interviews, or manual inspections of artifacts. The process can be time-consuming but it is often the easiest to perform especially in the absence of any structured data collection available on the project. Automated data collection, on the other hand, typically consults some existing data source, such as a source version control system or a defect repository, and extracts the necessary information from there.

There is a trade-off between the two methods in terms of cost of the collection and quality of the data. Frequently, a combination of both methods is necessary for the collection to reach the desired level of precision and data volume. For example, the initial data extraction is often done automatically but then the result needs to be manually verified and noise removed. The noise at the level of the structure of data is quite easy to identify. This is the case, for example, with missing values or values not conforming to the prescribed schema. Much harder cases involve the data semantics where the knowledge of the product or the team's process is necessary for disambiguation and a successful resolution.

There are situations, however, in which automated data collection creates datasets of higher quality than those created by a manual process. This happens when the artifacts that the data is collected from are generated outside of the critical-path in the engineer workflow. A classic example of data of this kind are the various relationships between semantically-related artifacts, such as a work item and a source code change or a code change and a code review performed on it. In many engineering workflows, explicitly saving such relationships in the repository is unnecessary for successful task completion and is often considered extra effort. Thus, work items, code reviews and code changes often form independent data silos. An engineer might be asked to make a connection between any two artifacts by a team policy but as the engineering system will work without them, the adherence to policy is based on managerial and peer pressure and often not universally applied. In such cases, being able to discover the relationships automatically from metadata surrounding the artifacts (code reviews and code submissions share many characteristics that can be retrieved from their metadata and correlated) often creates a more complete, higher quality dataset without requiring human input, making it less expensive.

13.5.1.2 *Purpose of used data source*

No matter if the collection is manual or automated; the particulars of the collection process can heavily influence the ability to draw conclusions from data. Therefore, it is important to have a full

understanding not only of the data collection process but how the data was introduced into the system in the first place. Often, knowledge of details allows the analyst to work around data deficiencies. To that end it is important to know the objective of the process or tool that generates the metric.

The most accurate data often comes from tools designed to manage the workflow or process being measured. Such tools or processes are the primary source. Examples include source code version control systems when code churn is being analyzed.

The data accuracy is lower when the data is collected indirectly as it is, for example, when attempting to enumerate bug fixes in code from defect data (which is not the same as the data from source code). And when data needs to be interpreted, the accuracy depends on how well the interpretation aligns with the goal of the metric. For instance, metrics relating to the size of change can take the form of counting changed files or changed lines of code. In some circumstances a “changed file” would also include files newly added or deleted, the definition is dependent on the goal of the metric. In addition, defining a “changed line of code” is even more complex, as it depends on a chosen similarity measure and the format of files. For example, measuring changes in “lines” for XML files, is not very useful. In such cases, the interpretation tries to be as close as possible to the intuitive understanding of the concept of a “change size” by the engineers on a particular project. In the case of changed lines of code, this metric is best approximated by using the same tool the particular engineering team uses for viewing the code diffs. Here, the measurement is a side-effect of a tool.

Special care needs to be applied to any metric that concerns measuring people or teams. Especially if there are incentives attached to the metrics, the data will over time become unreliable as far as its initial objectives. A classic example is applying a “number of defects found” metric to judge the effectiveness of testing efforts. In such circumstances, it is not unusual to see multiple defects opened for each found symptom of one underlying problem where a single more comprehensive bug report would be more useful and would require less work filling out, triaging, and tracking. Counteracting the effects of gaming is hard because causes of the changes in metrics are often not easily distinguishable from naturally occurring events and requires in-depth knowledge of the project and the team. Normalization and using a set of metrics more resilient to gaming often helps. For example, instead of “defects opened”, it might be worth applying “defects resolved through a code change” metric, since the latter is harder to game; performing a code change is typically more visible across the team than performing a defect record change. Since the problem of gaming might occur in any dataset involving events produced by incentivized people, recognition of its potential existence and applying countermeasures is therefore very important.

13.5.1.3 Process semantics

In many cases, specifics of the workflow are not explicit; but required for correct interpretation of data. Such workflow semantics can be understood in the context of and with deep knowledge of the process and the team working on a project. How important it is to understand the assumptions being made on the project is best illustrated with challenges faced when interpreting defect data.

Defect reports come from many sources: internal testing efforts, limited deployments, production deployments, telemetry collected directly from users, either automatically or through surveys and studies. The source of a defect report often determines the quality of the report and its fate. For example, crash data being reported automatically is precise as far as the symptom and, at least partially through the stack trace, the path software took to trigger the problem. Such defects often have a determinate

root cause and a solution. On the other hand, when reporting issues found by usability studies, there is more subjectivity and consequently less precision in the defect report.

There is also a lot of subjectivity around categorization of defects. Assigning severity to a defect, even on teams with strong policies, is often unequal. A simple issue of deciding whether a task is tracking a bug fix or a more general work item can be difficult to deconstruct and varies even among teams working on the same product. Applying the concept of “bug triage” is frequently a way for a team to impose some measure of consistency of information gathered and resolution on the bug queue. Bug triage often removes noise from the defect data, for example, by requesting that all defects come into the queue with some minimum level of information provided, that work items tagged as defects are not in fact features (or vice versa), or that duplicates of the same issue are recognized and tagged properly. From the metrics point of view, defects which have gone through triage are typically carrying more data, are of higher quality and more readily amenable to analysis. In the absence of a team triage, the data clean up falls on the analyst who may be less knowledgeable about the team’s policies and consequently less able to produce a high quality dataset.

Another class of problems with defect data stems from the fact that records for some of the defects do not exist (are omitted) so they may not reflect the reality of the product. Most commonly, omission of records happens when fixing a defect is less costly than the cost of tracking it in a formal system, or the visibility requirements at that stage of the project are not high. Code reviews, for example, depending on the level of formality attached to them, may result in defects being communicated to an engineer through a side channel, an informal conversation, or a comment attached to an e-mail. Customer-reported issues, on the other hand, are treated with more formality, in no small measure to ensure that there is a record of the interaction with the customer.

13.5.2 DATA INTERPRETATION

This section discusses some of the most common issues regarding interpreting metrics, and some methods to mitigate them.

13.5.2.1 Metrics as indicators

While metrics ideally should reflect the development process, they are often only indicators of the status of the project and not precise statements of fact about the project. Software development is a complex process and metrics provide windows into the development process but rarely a complete view. As such the metrics should assist in managing software development but they are not a substitute for management.

The people responsible for interpreting the metrics should continually validate their interpretation with the software engineers developing the product. Where divergence occurs between the interpretation and the engineer’s perception, then the reasons for the divergences needs to be understood which may result in changes to how the metrics are interpreted.

13.5.2.2 Noisy data

There are two main classes of noisy data: missing data and unreliable data.

Missing data can occur due to tool issues or changes in practices. In these circumstances it is necessary to capture this data via different methods. Missing data should be investigated as it may be indicative of problems of either the development tools or the development process.

Unreliable data will always exist irrespective of the quality of the tools and the collection process. A common issue is the focus of development tools (which is not on metrics generation) so the metrics may not be totally accurate. For example, code diff'ing tools are inaccurate when the difference between files is large. Another common issue is default values in data fields; when data is entered manually it may not be clear if the person intended the field to be set to the value or they forgot to set the field to the correct value. A common issue is that a lot of defect reporting processes will have a default value for the defect priority and severity. These default values can bias any results. The greatest reason for noisy data, however, is data generated manually that is not strictly necessary for completion of primary tasks. Examples of auxiliary data often generated manually include: time spent on actions, ownership data, organization structure data. This data is often not kept up-to-date, or is liable to be gamed.

The models used for interpretation need to be able to handle missing data. If data is missing or becomes unreliable then it should be excluded from the model. But the model should be able to differentiate between a zero value for data and a null value, which represents missing data.

13.5.2.3 Gaming

Gaming of the system occurs when groups or individuals use the rules and procedures that are meant to protect or allow for understanding of a system, to manipulate the systems for a desired outcome. Gaming is more likely to occur when individuals or teams are being measured implicitly or explicitly by metrics. A common example of this is the effectiveness of test teams being measured in terms of the number of defects they find or code coverage achieved. This places pressure on the test team to find defects in a product irrespective of its quality or to add coverage in places that are not substantially at risk of failure only to achieve a desired number. If such gaming becomes prevalent, it invalidates the using of susceptible metrics as measures of quality.

Gaming can often be identified through correlations between metrics, so for instance, if the rate of defects opened is not correlated to the rate of defects fixed through code changes, this raises issues regarding the validity of using opened defects as a quality metric.

Where gaming has been identified then those metrics should not be used and other non-gamed metrics need to be identified. In the example above, where the rate of opened defects is not reflective of quality, then it can be replaced by the rate of code churn due to bug fixes.

13.5.2.4 Outliers

Metric outliers can skew statistics, such as averages, and so the temptation is to automatically ignore these values. Unfortunately, an outlier may either be due to noisy data or actual product issues. Monitoring and interpreting metrics from a single product makes it difficult to automatically interpret outliers. To understand the cause of outliers requires manual inspection of the data.

When applying the same set of metrics to multiple products it is possible to automatically differentiate between noisy data and truly problematic areas. Where the same set of outliers occurs across multiple products then it is more likely that the data is representative of normal product behavior and not corrupt data. Outliers should be discussed with the engineering team as some outliers indicating bad behavior may be normal development practice. For instance, the development team may decide to follow a different development methodology for a specific subtask of the development (as in certain cases of refactoring).

13.6 EXAMPLE OF CHANGING PRODUCT DEVELOPMENT THROUGH THE USE OF DATA

During the development of Windows Server 2012, the product team was frustrated by the length of time it was taking engineers to integrate completed features into the trunk branch.

The product team set two goals for themselves:

1. Increase the velocity of code integration from the feature branches into the trunk branch.
2. At least maintain but preferably improve product quality.

To achieve these goals the product team had to investigate what was the current code velocity for the product and what were the influencing factors. Windows Server is developed in a large branch tree, which shares the same trunk branch with Windows Client (Windows 8 and 8.1). The Windows Server development cycle is split into multiple milestones; some milestones are allocated for refactoring, some for feature development, and some for stabilization. The focus on code velocity is mainly during the feature development periods, so the investigation focused on that period.

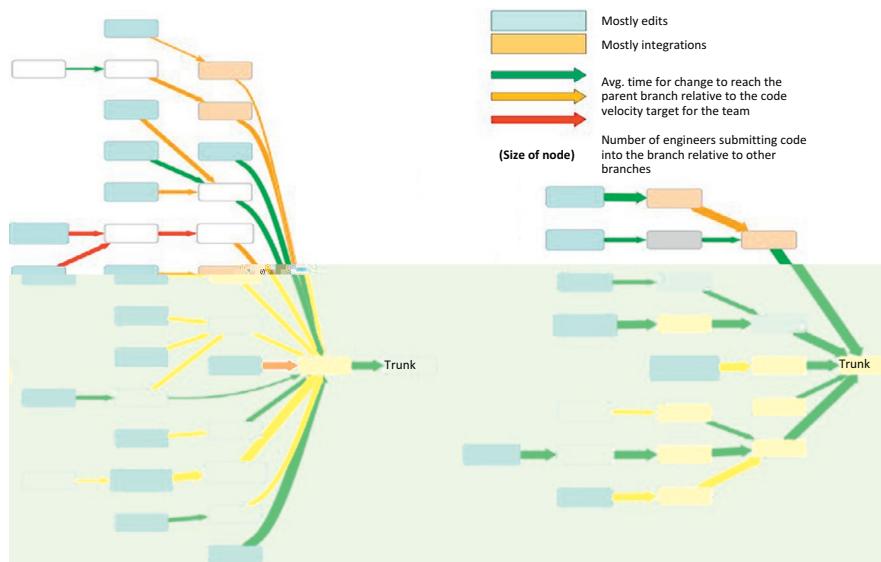
To generate a benchmark for Windows Server 2012 R2, data analysts and engineers from the product team analyzed the metrics generated during the Windows Server 2012 development. Where the metrics were at odds with the perceptions of what the engineers believed occurred (e.g., a particular feature was problematic) the data analysts and the engineers met to understand these differences. The most common reasons for differences between the metrics and engineering perception was misinterpretation of a metric, the requirement for the metric to take into account other related measurements, or the engineering teams “forgetting” issues with the product development. The result of the exercise was to improve the accuracy of the metrics and also to increase the confidence of the engineering team with the metrics.

Analysis of the data identified that the branches with more engineers developing code on, had the fastest code velocity; and the more shallow the branch tree the faster the code velocity was. Analysis of factors influencing code velocity identified the necessity of treating implementation and integration velocity separately. For code quality, the main focus was on avoidance of code integration issues. Typical examples of such issues are: (i) merge conflicts where code simultaneously developed on two parallel branches does not cleanly merge into a branch dedicated to verification, resulting in failure of the merge; (ii) runtime issues discovered in testing. Analysis identified a correlation between the number of code conflicts and the integration velocity.

Based on the analysis, the decision was made to increase the number of teams who developed code simultaneously on the same branch and to decrease the depth of the branch tree. The resulting architectural changes are shown in [Figure 13.2](#).

Reducing the depth of the branch tree was perceived as an increased quality risk, as certain branches where dedicated to verification purposes and now these branches were being removed. Additionally, the product group, recognizing that changing a complex process can have unforeseen side-effects, wanted to continuously monitor the impact of the changes in development architecture.

At the end of the first feature development milestone for Windows Server 2012 R2, a full analysis was performed comparing Windows Server 2012 R2 (the new branch model) behavior against that of Windows Server 2012 (old branch model serving as a benchmark). The product team was interested in comparison between old and new processes, not the absolute values achieved. The results of the analysis are displayed in [Table 13.2](#), as interpretation of the metric value is dependent on the metric

**FIGURE 13.2**

Branching structure used in Windows Server 2012 and 2012 R2, respectively.

Table 13.2 Comparable Metrics Between Product Developments

Category	Metric	Windows Server 2012	Windows Server 2012 R2	Status
Branch size	Number of Branches	1×	0.74×	Positive
	Engineers per Branch	1×	2.1×	Positive
	Teams Per Branch	1×	2.1×	Positive
Code velocity	Implementation Velocity	1×	0.88×	Positive
	Integration Velocity	1×	1.23×	Negative
Quality	Integration Conflicts	1×	0.56×	Positive
Volume of churn	Files	1×	1.28×	Positive
	Lines	1×	1.28×	Positive

itself, a status column is included to assist the readers. In the Status column, we indicate whether the change between releases was viewed as a positive or negative change.

Meetings were held with the product team to better understand these results. Specifically, questioned about changes to working practices, it was identified that while individual teams preferred to work separately in their own branches, they all felt that they benefited from working in branches with their related teams. The teams felt that the sharing of branches is the reason for the reduction in code conflicts. Previously, teams would develop their features separately and then merge the feature

into a larger branch for verification. Often, during that verification stage, which could be days after the code was originally written, conflicts were identified. During Windows Server 2012 R2, as more teams worked simultaneously in the same branch conflicts were identified and corrected immediately. Further analysis confirmed that the teams were moving more towards a continuous integration style of development.

Ironically, the metric that showed the smallest overall improvement was the implementation velocity, additionally the integration velocity showed a decrease, which was the opposite of what was intended. The combination of these metrics identified that code was spending less time in the branch used to develop the feature but more time being processed through the branch structure. Further investigation highlighted that separate decisions had been made to increase the amount of system testing that the software had to undergo prior to being allowed to check into the trunk branch; this increase may have been a reaction to the decrease in the number of integration branches dedicated to testing.

Overall the amount of code generated by the development team during the milestone period increased by 28%, indicating a general improvement in productivity. This improvement of the Windows Server 2012 R2 development process was considered successful by the product team. Consequently and following that, the team embarked upon a second major round of changes to the branch architecture using the same metrics-based methodology.

The three main findings that readers should take from this example are:

1. *The importance for the product team to have clear measureable goals:* Through the considerable effort in analyzing Windows 8, the teams could both quantify goals and have confidence that the metric generation process would accurately characterize the development of Windows 8.1.
2. *Continuously meeting with the product team to interpret the metrics, during the product development:* These meetings quickly identified where changes in the development process resulted in incorrect interpretation of the metrics, due to missing data or changes in naming conventions, etc. This ensured that when the product teams needed the data to make decisions then the data was accurate and available.
3. *Changing a complex process will always have unpredicted side-effects:* The changes to the development process had a positive side-effect of improving the working practices of the engineering teams. But equally the changes could have resulted in negative impact on other aspects of the development process (e.g., an increase in the velocity measures could have resulted in a decrease in quality). Therefore, it is important to not only monitor those aspects of the development process that are planned to change, it is also important to monitor areas that are supposed to be unaffected by the change.

The next section takes the learnings gained from a number of similar studies within Microsoft and provides a generalized approach to develop a data-driven engineering process.

13.7 DRIVING SOFTWARE ENGINEERING PROCESSES WITH DATA

This section summarizes the overall approach a software engineering team should apply to use data to drive their engineering processes, based on the factors discussed previously in this chapter.

The minimum requirement to enable data to help drive the software engineering process is for companies or product teams to define goals for their software development process.

The act of setting development goals forces the product groups to define the metrics that will be used to monitor those goals. Through defining the goals the product teams will also define the relative importance of different aspects of the development process. While ideally, product groups will want all aspects of their process to improve, realistically, improving one development characteristic will often have, at least initially, a negative impact on another characteristic. For example, it is challenging to increase the speed of product release and maintain quality without sacrificing the number of features in the release without a period of adjustment.

Without goals, it is also difficult to interpret the collected metrics. If the product performance is monitored and the average response time increases, without understanding the intention of the product teams, it is difficult to determine if that change was good or bad. Under such conditions, discussions that are supposed to be about the actions to be taken from the metrics inevitably result in time spent trying to resolve the definition of the metrics. Discussions about the implicit goals of the product and the actions are deferred.

Since a standard set of metrics that product groups can use to monitor their development process does not exist, in [Table 13.1](#) we provide a sample set of metrics that can help product teams in establishing their own scorecards. For all the metric categories, we provide examples of specific metrics, the goals for the metrics, and some of the challenges in interpreting those metrics.

Once the product group has determined the goals of the software engineering process and their metrics, they need to develop a process to collect and report on those metrics. The process of collecting metrics should be as automated as possible. The metrics and the data collection process can be verified through applying the process to prior releases of the same or similar products. The data analyst should characterize the behavior of the development of the prior product release and then verify their interpretation with the engineers who worked on the product. Through these discussions, the data analyst can improve their interpretation methods and also become aware of the accuracy and relevance of the chosen metrics. Ideally, this process should be applied to more than one development project, as that allows the data analysts to be better able to interpret whether statistical anomalies should be ignored or if they should be flagged as issues that need addressing.

Benchmarking the software engineering process based on the prior releases will allow the data analysts to translate the product goals into metric values. These metric values may vary over the total development cycle, for instance the code velocity during feature development periods should be longer than during stabilization periods.

This background work provides the information to allow the data analysts to track the performance of the product through comparing the collected metrics to the product goals. The focus of the data analysts during the development process should be to:

1. Continually validate the collected metrics through discussions with the engineering team and through correlate-related metrics to ensure their accuracy and lack of gaming.
2. Validate that the product development is still performing to plan based on the values of the collected metrics in relation to the product goals.
3. Examine the main drivers of any specific metric to identify opportunities to improve the metrics through optimizing the development process.

The process of creating product goals and translating those goals to validated metrics is expensive, but once a product group has a trusted set of metrics, and a set of metric-based goals, they can use those to drive product development and to optimize their development process.

REFERENCES

- [1] Dongmei Z, Shi H, Yingnong D, Jian-Guang L, Haidong Z, Tao X. Software analytics in practice. *IEEE Softw* 2013;30(5):30–37. doi:10.1109/MS.2013.94.
- [2] Murphy B, Gent T. Measuring system and software reliability using an automated data collection process. *Qual Reliabil Eng Int* 1995;11(5):341–53.
- [3] Jalote P, Murphy B. Reliability growth in software products. In: IEEE international symposium on software reliability engineering; 2004.
- [4] Murphy B. Automating software failure reporting. *Queue* 2004;November:42–48.
- [5] Czerwonka J, Nagappan N, Schulte W, Murphy B. Codemine: Building a software analytic platform for collecting and analysing engineering process data at Microsoft. Microsoft Technical Report; 2013. MSR-TR-2013-7
- [6] Williams L, Brown G, Nagappan N. Scrum + engineering practices: experiences of three Microsoft teams. In: International symposium on empirical software engineering and measurement; 2011. p. 463–71.
- [7] Sanchez J, Williams L, Maximilien M. A longitudinal study of the test-driven development practice in industry. Washington, DC: Agile; 2007. p. 5–14.
- [8] Ho CW, Johnson M, Williams L, Maximilien E. On agile performance requirements specification and testing. Minneapolis, DC: Agile; 2006. 6 p. ISBN 0-7695-2562-8/06. Electronic proceedings.
- [9] Layman L, Williams L, Cunningham L. Motivations and measurements in an agile case study. *J Syst Architect* 2006;52(11):654–67.
- [10] Williams L, Krebs W, Layman L, Antón A. Toward a framework for evaluating extreme programming. In: Empirical assessment in software engineering (EASE); 2004.
- [11] Rolt LTC. Red for Danger. Stroud, UK: Sutton Publishing Limited; 1955.
- [12] Musa J. A theory of software reliability and its applications. *IEEE Trans Softw Eng* 1975;1(3):312–30.
- [13] Littlewood B. Software reliability model for modular program structure. *IEE Trans Reliabil* 1979;R-28(3):241–6.
- [14] O'Connor P. Practical reliability engineering. Chichester, UK John Wiley & Sons; 1985. p. 133, 233–34.
- [15] Siewiorek D, Swan R. The theory and practice of reliable system design. Bedford, MA: Digital Press; 1982.
- [16] Gray J. Why do computers stop and what can be done about it. In: Proceedings of the 5th symposium on reliability in distributed software and database systems. Los Angeles, CA; 1986. p. 3–12.
- [17] Moran P, Gaffney P, Melody J, Condon M, Hayden M. System availability monitoring. *IEEE Trans Reliabil* 1990;39(4):480–85.
- [18] Jones C. Measuring programming quality and productivity. *IBM Syst J* 1978;17(1):39–63.
- [19] Boehm B. Software engineering economics. Englewood Cliffs, NJ: Prentice-Hall; 1981.
- [20] Brooks F. The mythical man month. Reading, MA: Addison-Wesley; 1986.
- [21] Lehman M, Belady L. Program evolution—processes of software change. London: Academic Press; 1985.
- [22] Lehman MM. Laws of software evolution revisited. In: Proceedings of European workshop on software process technology. LNCS, vol. 11491. Nancy: Springer Verlag; 1996. p. 108–24.
- [23] ISO 9000 standard. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=42180.
- [24] Software Engineering Institute. URL: <http://www.sei.cmu.edu/>.
- [25] Fenton N, Pfleeger S. Software metrics. Boston, MA: PWS Publishing Company; 1997.

- [26] Jones C. Software assessments. Benchmarks and best practices. Reading, MA: Addison-Wesley; 2000.
- [27] Beck K, et al. Manifesto for agile software development. Agile Alliance. 14 June 2001.
- [28] Li P, Kivett R, Zhan Z, Jeon Se, Nagappan N, Murphy B, et al. Characterizing the differences between pre and post release versions of software. International conference on software engineering; 2011.
- [29] Mockus A. Organizational volatility and its effect on software defect. In: ACM SIGSOFT international symposium on foundations of software engineering; 2010. p. 117–26.
- [30] Zimmermann T, Nagappan N, Williams L. Searching for a needle in a haystack: predicting security vulnerabilities for Windows Vista, software testing, verification and validation (ICST); 2010.
- [31] Cataldo M, Herbsleb J. Coordination breakdowns and their impact on development productivity and software failures. *Trans Softw Eng* 2013;39(3):343–60.
- [32] Zimmerman T, Nagappan N, Predicting defects using network analysis on dependency graphs. In: International conference on software engineering. Leipzig, Germany; 2008.
- [33] Ducasse S, Lanza M, Marinescu R. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. New York: Springer; 2010.
- [34] Basili V, Caldiera G, Rombach D. The goal question metric paradigm. In: Encyclopaedia of software engineering, vol. 2. New York: John Wiley and Sons Inc.; 1994. p. 528–32.
- [35] Sim S, Easterbrook S, Holt RC. Using benchmarking to advance research: a challenge to software engineering; In: International conference on software engineering. Portland; 2003.
- [36] Basili V, Shull F, Lanubile F. Building knowledge through families of experiments. *IEEE Trans Softw Eng* 1999;25:4.
- [37] Seaman C, Guo Y. Measuring and monitoring technical debt. *Adv Comput* 2011;82:25–46.
- [38] Conway ME. How do committees invent? *Datamation* 1968;14(4):28–31.
- [39] Nagappan N, Murphy B, Basili V. The influence of organizational structure on software quality: an empirical case study In: International conference on software engineering; 2008.
- [40] Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In: International conference on software engineering; 2005.
- [41] Nagappan N, Ball T, Zeller A. Mining metrics to predict component failure In: International conference on software engineering; 2006.

PAST, PRESENT, AND FUTURE OF ANALYZING SOFTWARE DATA

1

Christian Bird*, Tim Menzies†, Thomas Zimmermann*

*Microsoft Research, Redmond, WA, USA** *Computer Science, North Carolina State University, Raleigh, NC, USA†*



CHAPTER OUTLINE

1.1 Definitions	3
1.2 The Past: Origins	5
1.2.1 Generation 1: Preliminary Work	6
1.2.2 Generation 2: Academic Experiments.....	6
1.2.3 Generation 3: Industrial Experiments	6
1.2.4 Generation 4: Data Science Everywhere	7
1.3 Present Day	7
1.4 Conclusion	11
Acknowledgments	12
References.....	12

So much data, so little time.

Once upon a time, reasoning about software projects was inhibited by a lack of data. Now thanks to the Internet and open source, there's so much data about software projects that it's impossible to manually browse through it all. For example, at the time of writing (December 2014), our Web searches shows

that Mozilla Firefox has over 1.1 million bug reports, and platforms such as GitHub host over 14 million projects. Furthermore, the PROMISE repository of software engineering data (openscience.us/repo) contains data sets, ready for mining, on hundreds of software projects. PROMISE is just one of more than a dozen open source repositories that are readily available to industrial practitioners and researchers; see the following table.

The screenshot shows the homepage of the tera-PROMISE repository. At the top, there's a navigation bar with links for 'Blog', 'About', and 'People'. Below the navigation is a large graphic of blue filing cabinets labeled 'PROMISE'. To the right of the graphic, the text 'tera-PROMISE' is displayed in large letters, followed by 'Computer Science' and 'NC STATE UNIVERSITY'. A prominent blue button labeled 'Contribute' is centered below the main title. On the left side, there are several sections with bullet-pointed lists: 'Defect' (CK, McCabe & Halsted), 'Effort' (COBOL, Cocomo, Function Points Analysis, ISBSG, Personnel, Other), 'Requirements' (NRP, Other), and 'Issues'. In the center, under the heading 'Welcome to the Tera-Promise Repository', it says: 'This repository is an update to the older PROMISE repository of SE data. Our goal is to be a long term storage facility for SE data. Many researchers take the time to carefully host their data on their own web sites. But as people move through their career, those web sites can fade away and take away their data. Hopefully, using this repo, we can make conclusions in SE repeatable, and repeatable for a longer time.' It also notes size restrictions (1TB total) and chasing funds for expansion. Below this, a section titled 'How to Contribute' provides instructions for sending messages to opensciences.content@gmail.com.

Repositories of Software Engineering Data

Repository	URL
Bug Prediction Dataset	http://bug.int.usi.ch
Eclipse Bug Data	http://www.st.cs.uni-saarland.de/softevo/ bug-data/eclipse
FLOSSMetrics	http://flossmetrics.org
FLOSSMole	http://flossmole.org
International Software Benchmarking Standards Group (IBSBSG)	http://www.isbsg.org
Ohloh	http://www.ohloh.net
PROMISE	http://promisedata.googlecode.com
Qualitas Corpus	http://qualitascorpus.com
Software Artifact Repository	http://sir.unl.edu
SourceForge Research Data	http://zeriot.cse.nd.edu
Sourcerer Project	http://sourcerer.ics.uci.edu
Tukutuku	http://www.metriq.biz/tukutuku
Ultimate Debian Database	http://udd.debian.org

It is now routine for any project to generate gigabytes of artifacts (software code, developer emails, bug reports, etc.). How can we reason about it all? The answer is data science. This is a rapidly growing field with immense potential to change the day-to-day practices of any number of fields. Software companies (e.g., Google, Facebook, and Microsoft) are increasingly making decisions in a data-driven way and are in search of data scientists to help them.

1.1 DEFINITIONS

It is challenging to define software analytics for software engineering (SE) since, at different times, SE analytics has meant different things to different people. [Table 1.1](#) lists some of the more recent definitions found in various papers since 2010. Later in this introduction, we offer a short history of work dating back many decades, any of which might be called “SE data analytics.”

One reason for this wide range of definitions is the diversity of services and the diversity of audiences for those services. SE data science covers a very wide range of individuals and teams including, but not limited to, the following:

1. Users deciding what funds to allocate to that software;
2. Developers engaged in software development or maintenance;
3. Managers deciding what functionality should be assigned to which developers engaged in that development or maintenance;

Table 1.1 Five Definitions of “Software Analytics”

Hassan A, Xie T. Software intelligence: the future of mining software engineering data. FoSER 2010: 161-166.	[Software Intelligence] offers software practitioners (not just developers) up-to-date and pertinent information to support their daily decision-making processes.
Buse RPL, Zimmermann T. Analytics for software development. FoSER 2010:77-90.	The idea of analytics is to leverage potentially large amounts of data into real and actionable insights.
Zhang D, Dang Y, Lou J-G, Han S, Zhang H, Xie T. Software analytics as a learning case in practice: approaches and experiences. MALETS 2011.	Software analytics is to enable software practitioners to perform data exploration and analysis in order to obtain insightful and actionable information for data driven tasks around software and services (and software practitioners typically include software developers, tests, usability engineers, and managers, etc.).
Buse RPL, Zimmermann T. Information needs for software development analytics. ICSE 2012:987-996.	Software development analytics . . . empower(s) software development teams to independently gain and share insight from their data without relying on a separate entity.
Menzies T, Zimmermann T. Software analytics: so what? IEEE Softw 2013;30(4):31-7.	Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions.
Zhang D, Han S, Dang Y, Lou J-G, Zhang H, Xie T. Software analytics in practice. IEEE Softw 2013;30(5):30-7.	With software analytics, software practitioners explore and analyze data to obtain insightful, actionable information for tasks regarding software development, systems, and users.

4. Analysts trying to reduce code runtimes;
5. Test engineers developing work arounds to known problems;
6. And many more besides these five.

It would be very brave, and very inaccurate, to say that one definition of “analytics” holds across this diverse range. For example, Table 1.2 shows nine different information needs seen in interviews with 100+ software managers and developers [1].

Other work has also shown the broad range of information needs for different audiences. For example, the paper “Analyze This! 145 Questions for Data Scientists in Software Engineering” lists over 12 dozen different kinds of questions that have been seen in the information needs of software developers [2]. Note that each of these may require a different kind of analysis before an SE data scientist can answer the particular questions of specific users (as shown in the last column of Table 1.3).

We use the term “data scientist” to denote a person who can handle all these techniques (and more) as well as and adapt them to different information needs. As the following historical notes show, organizations have had “data scientists” for many years—albeit not as high demand or highly paid as in the current environment. Organizations hire these data scientists to explore the local data to find models that most answer the questions of most interest to the local business users [3]. These scientists know

Table 1.2 Space of Information Needs that can be Addressed by Data Science in SE

	Past	Present	Future
Exploration Find important conditions.	Trends Quantifies how an artifact is changing. Useful for understanding the direction of a project. <ul style="list-style-type: none"> • Regression analysis. 	Alerts Reports unusual changes in artifacts when they happen. Helps users respond quickly to events. <ul style="list-style-type: none"> • Anomaly detection. 	Forecasting Predicts events based on current trends. Helps users make pro-active decisions. <ul style="list-style-type: none"> • Extrapolation.
Analysis Explain conditions.	Summarization Succinctly characterizes key aspects of artifacts or groups of artifacts. Quickly maps artifacts to development activities or other project dimensions. <ul style="list-style-type: none"> • Topic analysis. 	Overlays Compares artifacts or development histories interactively. Helps establish guidelines. <ul style="list-style-type: none"> • Correlation. 	Goals Discovers how artifacts are changing with respect to goals. Provides assistance for planning. <ul style="list-style-type: none"> • Root-cause analysis.
Experimentation Compare alternative conditions.	Modeling Characterizes normal development behavior. Facilitates learning from previous work. <ul style="list-style-type: none"> • Machine learning. 	Benchmarking Compares artifacts to established best practices. Helps with evaluation. <ul style="list-style-type: none"> • Significance testing. 	Simulation Tests decisions before making them. Helps when choosing between decision alternatives. <ul style="list-style-type: none"> • What-if? analysis.

Table 1.3 Mapping Information Needs (Left) to Automatic Technique (Right)

Information Need	Description	Insight	Relevant Techniques
Summarization	Search for important or unusual factors associated with a time range.	Characterize events, understand why they happened.	Topic analysis, NLP
Alerts (& Correlations)	Continuous search for unusual changes or relationships in variables	Notice important events.	Statistics, Repeated measures
Forecasting	Search for and predict unusual events in the future based on current trends.	Anticipate events.	Extrapolation, Statistics
Trends	How is an artifact changing?	Understand the direction of the project.	Regression analysis
Overlays	What artifacts account for current activity?	Understand the relationships between artifacts.	Cluster analysis, repository mining
Goals	How are features/artifacts changing in the context of completion or some other goal?	Assistance for planning	Root-cause analysis
Modeling	Compares the abstract history of similar artifacts. Identify important factors in history.	Learn from previous projects.	Machine learning
Benchmarking	Identify vectors of similarity/difference across artifacts.	Assistance for resource allocation and many other decisions	Statistics
Simulation	Simulate changes based on other artifact models.	Assistance for general decisions	What-if? analysis

that before they apply technique XYZ, they first spend much time with their business users learning their particular problems and the specific information needs of their domain.

1.2 THE PAST: ORIGINS

Moving on from definitions, we now offer a historical perspective on SE data analytics. Looking back in time, we can see that this is the fourth generation of data science in SE. This section describes those four generations.

However, before doing that, we add that any historical retrospective cannot reference all work conducted by all researchers (and this is particularly true for a field as large and active as data science in software engineering). Hence, we apologize in advance to any of our colleagues not mentioned in the following.

1.2.1 GENERATION 1: PRELIMINARY WORK

As soon as people started programming, it became apparent that programming was an inherently buggy process. As recalled by Wilkes [4], speaking of his programming experiences from the early 1950s:

“It was on one of my journeys between the EDSAC room and the punching equipment that ‘hesitating at the angles of stairs’ the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.”

It took several decades to gather the experience required to quantify the size/defect relationship. In 1971, Akiyama [5] described the first known “size” law, stating that the number of defects D was a function of the number of LOC; specifically:

$$D = 4.86 + 0.018^*i.$$

In 1976, Thomas McCabe argued that the number of LOC was less important than the complexity of that code [6]. He argued that code is more likely to be defective when its “cyclomatic complexity” measure was over 10.

Not only is programming an inherently buggy process, it’s also inherently difficult. Based on data from 63 projects, in 1981 Boehm [7] proposed an estimator for development effort that was exponential on program size:

$$\text{effort} = a^* \text{ KLOC } b^* \text{ EffortMultipliers} \quad (\text{where } 2.4 \leq a \leq 3 \text{ and } 1.05 \leq b \leq 1.2).$$

At the same time, other researchers were finding repeated meta-level patterns in software development. For example, in the late 1970s, Lehman proposed a set of *laws of software evolution* to describe a balance between (1) forces driving new developments and (2) forces that slow down progress [8]. For example, the *law of continuing change* states that an “e-program” (whose behavior is strongly connected to the environment where it runs) must always be continually adapted or it will become progressively less satisfactory.

1.2.2 GENERATION 2: ACADEMIC EXPERIMENTS

From the late 1980s, some data scientists starting analyzing software data using algorithms taken from artificial intelligence research. For example, Selby and Porter found that decision-tree learners could identify which components might be error-prone (or having a high development cost) [9]. After that, very many researchers tried very many other AI methods for predicting aspects of software projects. For example, some researchers applied decision trees and neural networks to software effort estimation [10] or reliability-growth modeling [11]. Yet other work explored instance-based reasoning by analogy [12] or rough sets [13] (again, for effort estimation).

1.2.3 GENERATION 3: INDUSTRIAL EXPERIMENTS

From around the turn of the century, it became more common for workers at industrial or government organizations to find that data science can be successfully applied to their software projects.

For example, Norman Schneidewind explored quality prediction via Boolean discriminant functions for NASA systems [14]. Also at NASA, Menzies and Feather et al. used AI tools to explore trade-offs in early lifecycle models [15] or to guide software inspection teams [16].

Further over at AT&T, Ostrand and Weyuker and Bell used binomial regression functions to recognize 20% of the code that contained most (over 80%) of the bugs [17]. Other prominent work in this time frame included:

- Zimmermann et al. [18] who used association rule learning to find patterns of defects in a large set of open source projects.
- Nagappan, Ball, Williams, Vouk et al. who worked with Nortel Networks and Microsoft to show that data from those organizations can predict for software quality [19, 20].

1.2.4 GENERATION 4: DATA SCIENCE EVERYWHERE

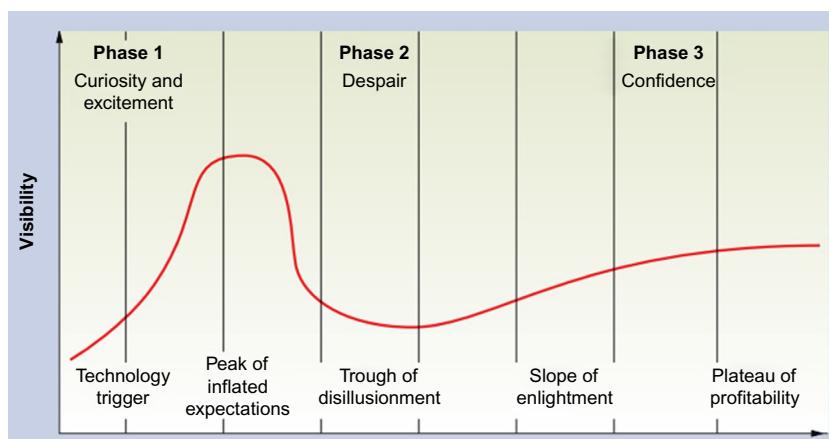
After the above, it became harder to track the massive growth in data science for SE. Many organizations such as Google, Facebook, and Microsoft (and others) routinely apply data science methods to their data. Since 2005, at many conferences, it has become routine to find papers from industrial practitioners and/or academics applying data science methods to software projects. Additionally, since 2010, we have seen a dramatic increase in the starting salaries of our graduate students who take industrial data science positions.

Further, various conferences have emerged that devote themselves to SE data science. At the time of writing, it is now routine for many SE conference papers to use data miners. However, during the last decade, two conferences lead the way: the Mining Software Repositories conference and the PROMISE conference on repeatable experiments in SE. Both communities explore data collection and its subsequent analysis with statistics or data mining methods, but each community has a particular focus: the MSR community is mostly concerned with the initial data collection while PROMISE community is more concerned with improving the efficacy and repeatability of that data's analysis. This book focuses more on the MSR community, while another book (*Sharing Data and Models in Software Engineering*, Morgan Kaufmann, 2014) offers the PROMISE perspective on data analysis.

1.3 PRESENT DAY

Data science has become so popular that at the time of writing (May, 2014), there is something of an unsustainable “bubble” in this area. Over the past few years, the volume and types of data related to SE has grown at an unprecedented rate and shows no sign of slowing. This turn of events has led to a veritable gold rush, as many “big data” enthusiasts mine raw data and extract nuggets of insight. A very real danger is that the landscape may become a Wild West where inexperienced software “cowboys” sell hastily generated models to unsophisticated business users, without any concern for best or safe practices (e.g., such as the best practices documented in this book).

To some extent, the current “bubble” in data is not surprising. New technologies such as data science typically follow the “hype curve” of [Figure 1.1](#). Just like the dot-coms in the late 1990s, the current boom in “big data” and data science is characterized by unrealistic expectations. However, as shown in [Figure 1.1](#), if a technology has something to offer, it won’t stay in the trough of disillusionment.

**FIGURE 1.1**

Standard hype cycle for new technology.

Internet-based computing survived the “dot-gone” meltdown of 1999–2001 and has now risen to a new sustainable (and growing) level of activity. We are confident that data science will also endure and thrive as a important technology for the decades to come.

One goal of this book is to go “behind the hype” and demonstrate proven principles for a sustainable data science industry. One standout result from this book is that *data science needs data scientists*. As shown by the examples in this book, data science for SE can be an intricate task involving extensive and elaborate combinations of tools. Further, it is not enough to merely use those tools—it is also necessary to understand them well enough to adapt and integrate them into some human-level process. Hence, to use data science properly, organizations need skilled practitioners, extensive knowledge of humans and of organizations, a broad skill set, and a *big toolkit* of methods.

For example, consider the technology used in the chapter “Analytical Product Release Planning” by Maleknaz Nayebi and Guenther Ruhe. As shown below, this list of technology is not short. Further, it is representative of the kind of data science solutions being deployed today in many organizations.

The automatic methods discussed in that chapter include:

- Analogical reasoning;
- DBScan (which is a kind of density-based clustering algorithm);
- Attribute weighting; Preprocessing of data (e.g., to filter out erroneous values or to fill in missing values); and
- Methodological tools for assessing a learned model (e.g., leave-n-out experiments; different performance measures).

However, that chapter does not stop at mere automatic methods. Recalling part our definition (shown above), we said that the goal of data science is “*to gain and share insight from data to make better decisions*.” Note how this “insight” is a human-reaction to data analysis. Therefore, it is vital that automatic tools be augmented with human-in-the-loop interaction. As examples of that kind of human-level analysis, Nayebi and Ruhe use:

- Qualitative business modeling;
- Amazon's Mechanical Turk;
- Combining algorithmic vs. expert judgement;
- Aggregation of expert judgments.

The bad news is that this kind of analysis is impossible without trained and skilled data scientists. The good news is that the community of trained and skilled data scientists, while not large, is growing. Maryalene LaPonsie calls "data scientists" the "the hottest job you haven't heard of" [21]. She writes that "The University of California San Diego Extension lists data mining and analytics as the second hottest career for college graduates in 2011. Even the Cheezburger Network, home of the web's infamous LOLCats, recently brought a data scientist on board."

In any field with such rapid growth, there are two problems:

1. How do we train the newcomers?
2. How do we manage them?

As to the issue of *training*, most people in SE data science currently get their knowledge about analysis from general data analysis texts such as data mining texts [22], statistical texts [23], etc. While these are useful, they are aimed at a broader audience and do not include common issues in SE (as a simple example, unlike many fields, most SE metrics are not normally distributed). This aim of this book is to focus on how analysis is applied to real-world SE. This book will discuss a range of methods (from manual to automatic and combinations in the middle). Our goal in this book is to give readers an understanding of the breadth of analysis methods possible for a wide range of data taken from SE projects. A companion book *Sharing Data and Models in Software Engineering* takes a different in-depth approach (where the same data is analyzed in-depth by many different ways using many different methods) [24].

As for the issue of *management*, it can be difficult for senior managers to effectively lead teams where the teams are working on technologies that are so novel and disruptive as data science. For such senior managers, we offer the following advice:

- **It ain't all hardware:** In the era of Google-style inference and cloud computing, it's a common belief that a company can analyze large amounts of data merely by building (or renting) a CPU farm, then running some distributed algorithms, perhaps using Hadoop (<http://hadoop.apache.org>) or some other distributed inference mechanism. This isn't the case. In our experience, while having many CPUs is (sometimes) useful, the factors that determine successful software analytics rarely include the hardware. More important than the hardware is how that hardware is used by skilled data scientists.
- **It ain't all tools:** Another misconception we often see relates to the role of software. Some managers think that if they acquire the right software tools—Weka, Matlab, and so on—then all their analytical problems will be instantly solved. Nothing could be further from the truth. All the standard data analysis toolkits come with built-in assumptions that might be suitable for particular domains. Hence, a premature commitment to particular automatic analysis tools can be counterproductive.

In our view, one vital role for a data scientist is to uncover the right hardware and the right tools for a particular application. At the start of a data science project, when it isn't clear what the important

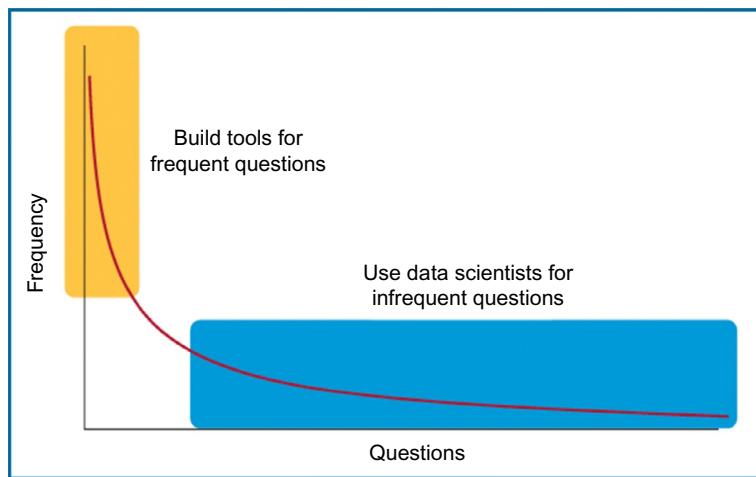


FIGURE 1.2

Data science projects may mature from many ad hoc queries to the repeated use of a limited number of queries.

factors in a domain are, a good data scientist will make many ad hoc queries to clarify the issues in that domain. Subsequently, once the analysis method stabilizes, it will then be possible to define what hardware and software tools would best automate any of the routine and repeated analysis tasks.

We can illustrate this process in Figure 1.2. As shown most data science projects mature by moving up along a curve. Initially, we might start in the darker region (where the queries are many, but the repeated queries are few) and then move into the lighter region (where we repeatedly make a small number of queries). This diagram leads to one of our favorite mantras for managing data science projects:

For new problems, deploy the data scientists before deploying tools or hardware.

As for other principles for data science, in our Inductive Engineering Manifesto [25], we made some notes on what characterizes best practices in industrial data mining. Combining them with the above, we arrive at six points from this chapter and five further points:

1. *It ain't all hardware.*
2. *It ain't all software.*
3. *Data science needs data scientists*—especially during that initial analysis stage where the goal is to find (a) the most informative queries (that should be automated) and (b) the right hardware and the right tools to automate that query.
4. *Users before algorithms.* Data mining algorithms are only useful in industry if users fund their use in real-world applications. The user perspective is vital to inductive engineering. The space of models that can be generated from any dataset is very large. If we understand and apply user goals, then we can quickly focus an inductive engineering project on the small set of most crucial issues.

5. *Broad skill set, big toolkit.* Successful inductive engineers routinely try multiple inductive technologies. To handle the wide range of possible goals an inductive engineer should be ready to deploy a wide range of tools. Note that the set of useful inductive technologies is large and constantly changing. Therefore, use tools supported by a large ecosystem of developers who are constantly building new learners and fixing old ones.
6. *Deploy the data scientists before deploying tools or hardware*—especially for new problems.
7. *Plan for scale:* In any industrial application, data mining is repeated many times to (a) answer additional questions raised by the users; or (b) make some enhancement and/or bug fix to the method, or (c) to deploy it to a different set of users. That is, for serious studies, to ensure repeatability, the entire analysis should be automated using some high-level scripting language.
8. *Early feedback.* Continuous and early feedback from users allows needed changes to be made as soon as possible and without wasting heavy up-front investment. Prior to conducting very elaborate studies, try applying very simple tools to gain rapid early feedback.
9. *Be open-minded.* It's unwise to enter into an inductive study with fixed hypotheses or approaches, particularly for data that hasn't been mined before. Don't resist exploring additional avenues when a particular idea doesn't work out. We advise this because data likes to surprise: initial results often change the goals of a study when business plans are based on issues irrelevant to local data.
10. *Do smart learning.* Important outcomes are riding on your conclusions. Make sure you check and validate them. There are many such validation methods such as repeat the analysis N times on, say, 90% of the available data—then check how well your conclusions hold across all those samples.
11. *Live with the data you have.* You go mining with the data you have, not the data you might want or wish to have at a later time. Because we may not have control over how data is collected, it's wise to clean the data prior to learning. For example, before learning from a dataset, conduct instance or feature selection studies to see what spurious data can be removed.

1.4 CONCLUSION

This is an exciting time for those of us involved in data science and the analysis of software data. Looking into the very near future, we can only predict more use of data science in SE. By 2020, we predict

- more and different data,
- more algorithms,
- faster decision making with the availability of more data and faster release cycles,
- more people involved in data science as it becomes more routine to mine data,
- more education as more people analyze and work with data,
- more roles for data scientists and developers as this field matures with specialized subareas,
- more real-time data science to address the challenges of quickly finding patterns in big data,
- more data science for software systems such as mobile apps and games, and
- more impact of social tools in data science.

As an example of this last point, check out *Human Boosting* by Harsh Pareek and Pradeep Ravikumar, which discusses how to boost human learning with the help of data miners [26]. In the very near future, this kind of human(s)-in-the-loop analytics will become much more prevalent.

ACKNOWLEDGMENTS

The work of this kind of book falls mostly on the authors and reviewers, and we're very appreciative of all those who took the time to write and comment on these chapters. The work of the reviewers was particularly challenging because their feedback was required in a very condensed timetable. Accordingly, we offer them our heartfelt thanks.

We're also grateful to the Morgan Kaufmann production team for their hard work in assembling this material.

REFERENCES

- [1] Buse RPL, Zimmermann T. Information needs for software development analytics. In: ICSE 2012; 2012. p. 987–96.
- [2] Begel A, Zimmermann T. Analyze this! 145 questions for data scientists in software engineering. In: ICSE'14; 2014.
- [3] Menzies T, Butcher A, Cok D, Marcus A, Layman L, Shull F, et al. Local vs. global lessons for defect prediction and effort estimation. IEEE Trans Softw Eng 2013;29(6).
- [4] Wilkes M. Memoirs of a computer pioneer. Cambridge, MA: MIT Press; 1985.
- [5] Akiyama F. An example of software system debugging. Inform Process 1971;71:353–9.
- [6] McCabe T. A complexity measure. IEEE Trans Softw Eng 1976;2(4):308–20.
- [7] Boehm B. Software engineering economics. Englewood Cliffs: Prentice-Hall; 1981.
- [8] Lehman MM. On understanding laws, evolution, and conservation in the large-program life cycle. J Syst Softw 1980; 1:213–21.
- [9] Porter AA, Selby RW. Empirically guided software development using metric-based classification trees. IEEE Softw 1990;7(2):46–54.
- [10] Srinivasan K, Fisher D. Machine learning approaches to estimating software development effort. IEEE Trans Softw Eng 1995;21(2):126–37.
- [11] Tian J. Integrating time domain and input domain analyses of software reliability using tree-based models. IEEE Trans Softw Eng 1995;21(12):945–58.
- [12] Shepperd M, Schofield C. Estimating software project effort using analogies. IEEE Trans Softw Eng 1997;23(11):736–43.
- [13] Ruhe G. Rough set based data analysis in goal oriented software measurement. In: Proceedings of the 3rd international symposium on software metrics: from measurement to empirical results (METRICS '96); 1996.
- [14] Schneidewind NF. Validating metrics for ensuring space shuttle flight software quality. Computer 1994;27(8):50,57.
- [15] Feather M, Menzies T. Converging on the optimal attainment of requirements. In: IEEE RE'02; 2002.
- [16] Menzies T, Stefano JSD, Chapman M. Learning early lifecycle IV and V quality indicators. In: IEEE symposium on software metrics symposium; 2003.
- [17] Ostrand TJ, Weyuker EJ, Bell RM. Where the bugs are. SIGSOFT Softw Eng Notes 2004; 29(4):86–96.
- [18] Zimmermann T, Weißgerber P, Diehl S, Zeller A. Mining version histories to guide software changes. In: Proceedings of the 26th international conference on software engineering (ICSE 2004), Edinburgh, United Kingdom; 2004. p. 563–72.

- [19] Nagappan N, Ball T. Use of relative code churn measures to predict system defect density. In: ICSE 2005; 2005.
- [20] Zheng J, Williams L, Nagappan N, Snipes W, Hudepohl JP, Vouk MA. On the value of static analysis for fault detection in software. IEEE Trans Softw Eng 2006; 32(4):240–53.
- [21] LaPonsie M. The hottest job you haven't heard of; 2011. July 5, 2011, URL: <http://www.onlinedegrees.com/>, <http://goo.gl/OjYqXQ>.
- [22] Witten IH, Frank E, Hall MA. Data mining: practical machine learning tools and techniques. 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2011.
- [23] Duda RO, Hart PE, Stork DG. Pattern classification. 2nd ed. Oxford: Wiley-Interscience; 2000.
- [24] Menzies T, Kocaguneli E, Minku L, Peters F, Turhan B. Sharing data and models in software engineering. Waltham, MA: Morgan Kaufmann Publishers; 2014.
- [25] Menzies T, Bird C, Kocaguneli E. The inductive software engineering manifesto: principles for industrial data mining; 2011. URL: <http://menzies.us/pdf11manifesto.pdf>.
- [26] Pareek H, Ravikumar P. Human boosting. In: Proceedings of the international conference on machine learning; 2013. URL: <http://jmlr.csail.mit.edu/proceedings/papers/v28/pareek13.pdf>.

Chapter 2

The Software Heritage Open Science Ecosystem



Roberto Di Cosmo and Stefano Zacchiroli

Abstract Software Heritage is the largest public archive of software source code and associated development history, as captured by modern version control systems. As of July 2023, it has archived more than 16 billion unique source code files coming from more than 250 million collaborative development projects. In this chapter, we describe the Software Heritage ecosystem, focusing on research and open science use cases.

On the one hand, Software Heritage supports empirical research on software by materializing in a single Merkle direct acyclic graph the development history of public code. This giant graph of source code artifacts (files, directories, and commits) can be used –and has been used– to study repository forks, open source contributors, vulnerability propagation, software provenance tracking, source code indexing, and more.

On the other hand, Software Heritage ensures availability and guarantees integrity of the source code of software artifacts used in any field that relies on software to conduct experiments, contributing to making research reproducible. The source code used in scientific experiments can be archived –e.g., via integration with open-access repositories – referenced using persistent identifiers that allow downstream integrity checks and linked to/from other scholarly digital artifacts.

R. Di Cosmo (✉)
Inria and Université Paris Cité, Paris, France
e-mail: roberto@dicosmo.org

S. Zacchiroli
LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France
e-mail: stefano.zacchiroli@telecom-paris.fr

2.1 The Software Heritage Archive

Software Heritage [1, 12] is a nonprofit initiative started by Inria in partnership with UNESCO to build a long-term universal archive specifically designed for software source code, capable of storing source code files and directories, together with their full development histories.

Software Heritage’s mission is to collect, preserve, and make easily accessible the source code of *all publicly available software*, addressing the needs of a plurality of stakeholders, ranging from cultural heritage to public administrations and from research to industry.

The key principles that underpin this initiative are described in detail in two articles written for a broader audience in the early years of the project [1, 12]. One of these principles was to avoid any *a priori* selection of the contents of the archive, to avoid the risk of missing relevant source code, whose value will only become apparent later on. Hence, one of the strategies enacted for collecting source code to archive is the large-scale automated crawling of major software development forges and distributions, as shown in Fig. 2.1.

As a consequence of this automated harvesting, there is no guarantee that the content of the archive only contains quality source code or only code that builds properly: curation of the contents will need to happen at a later stage, via human or automated processes that build a view of the archive for specific needs. It may also happen that the archive ends up containing content that needs to be removed, and this required the creation of a process to handle take down requests following current legal regulations.¹

The sustainability plan is based on several pillars. The first one is the support of Inria, a national research institution that is involved for the long term. A second one is the fact that Software Heritage provides a common infrastructure catering to the needs of a variety of stakeholders, ranging from industry to academia and from cultural heritage to public administrations. As a consequence, funding comes from a diverse group of sponsors, ranging from IT companies to public institutions. Finally, an extra layer of archival security is provided by a network of independent international mirrors that maintain each a full copy of the archive.²

We recall here a few key properties that set Software Heritage apart from other scholarly infrastructures:

- Software Heritage *proactively* archives *all software*, making it possible to store and reference any piece of publicly available software relevant to a research result, independently from any specific field of endeavor, and even when the author(s) did not take any step to have it archived [1, 12];

¹ See <https://www.softwareheritage.org/legal/content-policy/> for details.

² More details can be found at <https://www.softwareheritage.org/support/sponsors> and <https://www.softwareheritage.org/mirrors>.

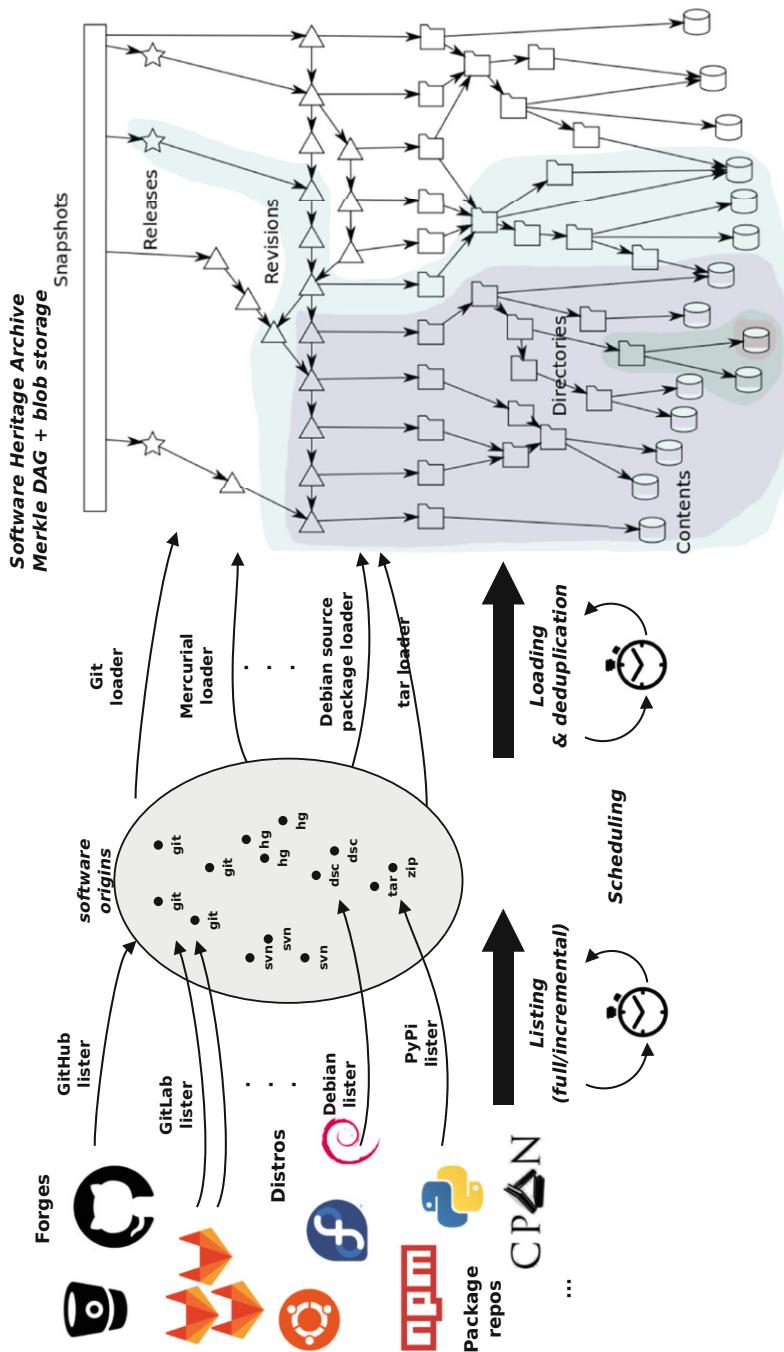


Fig. 2.1 Software Heritage data flow: crawling (on the left) and archival (right)

- Software Heritage stores source code with its development history in a uniform data structure, a Merkle Directed Acyclic Graph (DAG) [32], which allows to provide uniform, *intrinsic* identifiers for tens of billions archived software artifacts, independently of the version control system (VCS) or package distribution technology used by software developers [15].

Relevance for Software Ecosystems Software Heritage relates to software ecosystems, according to the seminal definition of Messerschmitt et al. [33] in two main ways. On the one hand, software products are associated with source code artifacts that are versioned and stored in VCSs. For Free/Open Source Software (FOSS), and more generally public code, those artifacts are distributed publicly and can be mined to pursue various goals. Software Heritage collects and preserves observable artifacts that originate from open-source ecosystems, enabling others to access and exploit them in the foreseeable future.

On the other hand, Software Heritage provides the means to foster the sharing of even more of those artifacts in the specific case of open scientific practices—what we refer to as the “open science ecosystem” in this chapter. Contrary to software-only ecosystems, the open science ecosystem encompasses a variety of software and non-software artifacts (e.g., data, publications); Software Heritage has contributed to this ecosystem the missing piece of long-term archival and referencing of scientifically relevant software source code artifacts.

2.1.1 Data Model

Modern software development produces multiple kinds of source code artifacts (e.g., source code files, directories, commits), which are usually stored and tracked in version control systems, distributed as packages in various formats, or otherwise.

When designing a software source code archive that stores source code with its version control history coming from a disparate set of platforms, there are different design options available. One option is to keep a verbatim copy of all the harvested content, which makes it easy to immediately reuse the package or version control tool. However, this approach can result in storage explosion: as a consequence of both social coding practices on collaborative development platforms and the liberal licensing terms of open-source software, those source code artifacts end up being massively duplicated across code hosting and distribution platforms.

Choosing a data structure that minimizes duplication is better for long-term preservation and the ability to identify easily code reuse and duplication.

This is the choice made by Software Heritage. Its data model is a Direct Acyclic Graph (DAG) that leverages classical ideas from content addressable storage and Merkle trees [32], which we recall briefly here.

As shown in Fig. 2.2, the Software Heritage DAG is organized in five logical layers, which we describe below from bottom to top.

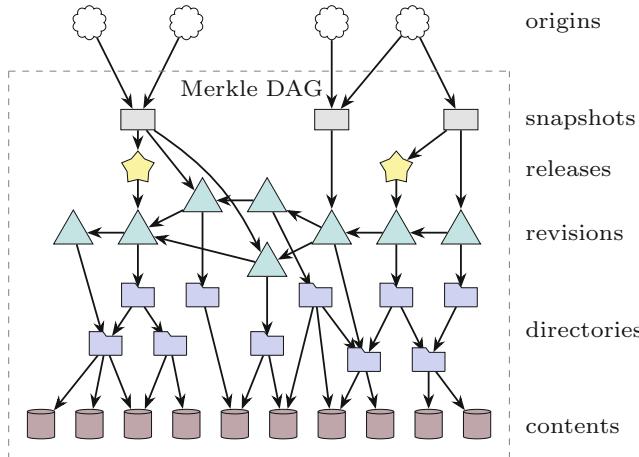


Fig. 2.2 Data model of the Software Heritage archive: a directed acyclic graph (DAG) linking together deduplicated software artifacts shared across the entire body of (archived) public code

Contents (or “blobs”) form the graph’s leaves and contain the raw content of source code files, not including their filenames (which are context-dependent and stored only as part of directory entries).

Directories are associative lists mapping names to directory entries and associated metadata (e.g., permissions). Each entry can point to content objects (“file entries”), revisions (“revision entries,” e.g., to represent git submodules or subversion externals), or other directories (“directory entries”).

Revisions (or “commits”) are point-in-time representations of the entire source tree of a development project. Each revision points to the root directory of the project source tree and a list of its parent revisions (if any).

Releases (or “tags”) are revisions that have been marked by developers as noteworthy with a specific, usually mnemonic, name (e.g., a version number like “4.2”). Each release points to a revision and might include additional metadata such as a changelog message, digital signature, etc.

Snapshots are point-in-time captures of the full state of a project development repository. While revisions capture the state of a single development line (or “branch”), snapshots capture the state of *all* branches in a repository and allow to reconstruct the full state of a repository that has been deleted or modified destructively (e.g., rewriting its history with tools like “git rebase”).

Origins represent the places where artifacts have been encountered in the wild (e.g., a public Git repository) and link those places to snapshot nodes and associated metadata (e.g., the timestamp at which crawling happened), allowing to start archive traversals pointing into the Merkle DAG.

The Software Heritage archive is hence a giant graph containing nodes corresponding to all these artifacts and links between them as graph edges.

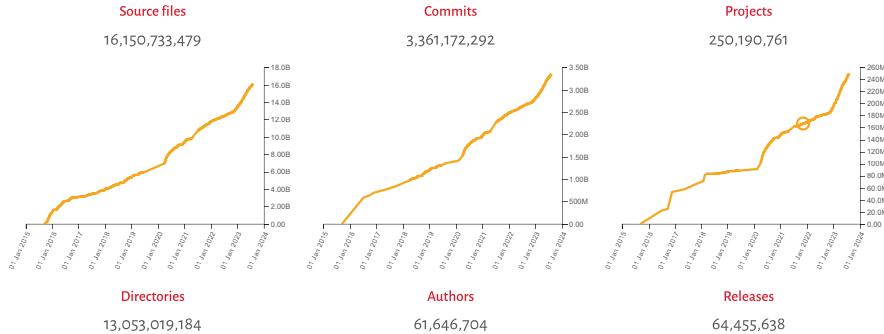


Fig. 2.3 Evolution of the Software Heritage archive over time (July 2023)

What makes this DAG capable of deduplicating identical content is the fact that each node is identified by a cryptographic hash that concisely represent its contents, and that is used in the SWHID identifier detailed in the next section. For the blobs that are the leaves of the graph, this identifier is just a hash of the blob itself, so even if the same file content can be present in multiple projects, its identifier will be the same, and it will be stored in the archive only once, like in classical content addressable storage [39]. For internal nodes, the identifier is computed from the aggregation of the identifiers of its children, following the construction originally introduced by Ralph Merkle [32]: as a consequence, if the same directory, possibly containing thousands of files, is duplicated across multiple projects, its identifier will stay the same, and it will be stored only once in the archive. The same goes for revision, releases, and snapshots.

In terms of size, the archive grows steadily over time as new source code artifacts get added to it, as shown in Fig. 2.3. As of July 2023, the Software Heritage archive contained over 16 billion unique source code files, harvested from more than 250 million software origins.³

2.1.2 Software Heritage Persistent Identifiers (SWHIDs)

As part of the archival process, a *Software Heritage Persistent Identifier (SWHID)*, is computed for each source code artifact added to the archive and can be used later to reference, look up, and retrieve it from the archive. The general syntax of SWHIDs is shown in Fig. 2.4.⁴

³ See <https://archive.softwareheritage.org> for these and other up-to-date statistics.

⁴ See <https://docs.softwareheritage.org-devel/swh-model/persistent-identifiers.html> for the full specification of SWHIDs.



Fig. 2.4 Schema of the Software Heritage identifiers (SWHID)

SWHIDs are URIs [5] with a simple syntax. *Core* SWHIDs start with the “`swh`” URI scheme; the colon (`:`) is used as separator between the logical parts of identifiers; the schema version (currently 1) is the current version of this identifier schema, then follows the type of source code artifacts identified, and finally comes a hex-encoded (using lowercase ASCII characters) cryptographic signature of this object, computed in a standard way, as detailed in [13, 15].

Core SWHIDs can then be complemented by *qualifiers* that carry contextual *extrinsic* information about the referenced source code artifact:

origin: the *software origin* where an object has been found or observed in the wild, as a URI;

visit: persistent identifier of a *snapshot* corresponding to a specific *visit* of a repository containing the designated object;

anchor: a *designated node* in the Merkle DAG relative to which a *path to the object* is specified;

path: the *absolute file path*, from the *root directory* associated with the *anchor node*, to the object;

lines: *line number(s)* of interest, usually pointing within a source code file.

The combination of core SWHIDs and qualifiers provides a powerful means of referring in a research article all source code artefacts of interest.

By keeping all the development history in a single global Merkle DAG, Software Heritage offers unique opportunities for *massive analysis of the software development landscape*. By archiving and referencing all the publicly available source code, the archive also constitutes the ideal place to *preserve research software artifacts* and offers powerful mechanisms to *enhance research articles* with precise references to relevant fragments of source code and contributes an essential building block to the software pillar of Open Science.

2.2 Large Open Datasets for Empirical Software Engineering

The availability of large amounts of source code that came with the growing adoption of open source and collaborative development has attracted the interest of software engineering researchers since the beginning of the 2000s and opened the way to large-scale empirical software engineering studies and a dedicated conference, Mining Software Repositories.

Several shared concerns emerged over time in this area, and we recall here some of the ones that are relevant for the discussion in this chapter.

One issue is the significant overhead involved in the systematic extraction of relevant data from the publicly available repositories and their analysis for testing research hypotheses. Building a very large-scale dataset containing massive amounts of source code with its version control history is a complex undertaking and requires significant resources, as shown in seminal work by Mockus in 2009 [34]. The lack of a common infrastructure spawned a proliferation of ad hoc pipelines for collecting and organizing source code with its version control history, a duplication of efforts that were subtracted to the time available to perform the intended research and hindered their reusability. A few initiatives were born with the intention of improving this unsatisfactory state of affairs: Boa [17] provides selected datasets (the largest and most recent one at the time of writing consists of about eight million GitHub repositories sampled in October 2019) and a dedicated domain specific language to perform efficient queries on them, while World of Code [31] collects git repositories on a large scale and maintains dedicated data structures that ease their analysis.

The complexity of addressing the variety of existing code hosting platforms and version control systems resulted in focusing only on subsets of the most popular ones, in particular the GitHub forge and the git version control system, which raises another issue: the risk of introducing bias in the results. In empirical sciences, *selection bias* [24] is the bias that originates from performing an experiment on a non-representative subset of the entire population under study. It is a methodological issue that can lead to threats to the *external validity* of experiments, i.e., incorrectly concluding that the obtained results are valid for the entire population, whereas they might only apply to the selected subset. In empirical software engineering, a common pattern that could result in selection bias is performing experiments on software artifacts coming from a relatively small set of development projects. It can be mitigated by ensuring that the project set is representative of the larger set of projects of interest, but doing so could be challenging.

Finally, there is the issue of enabling *reproducibility* of large-scale experiments—i.e., the ability to replicate the findings of a previous scientific experiment, by the same or a different team of scientists, reusing varying amounts of the artifacts used

in the original experiment [29].⁵ Large-scale empirical experiments in software engineering might easily require shipping *hundreds* of GiB up to a few TiB of source code artifacts as part of replication packages, whereas current scientific platform for data self archival usually cap at tens of GiB.⁶

The comprehensiveness of the Software Heritage archive, which makes available the largest public corpus of source code artifacts in a single logical place, helps with all these issues:

- reduces the *opportunity cost* of conducting large-scale experiments by offering at regular intervals as *open datasets* full dumps of the archive content
- contributes to *mitigate* selection bias and the associated external validity threats by providing a corpus that strives to be *comprehensive* for researchers conducting empirical software engineering experiments targeting large project populations.
- the persistence offered by an independent digital archive, run by a nonprofit open organization, eases the process of ensuring the *reproducibility* of large-scale experiments, avoiding the need to re-archive the same open-source code artifacts in multiple papers, a wasteful practice that should be avoided if possible. Using Software Heritage is enough to thoroughly document in replication packages the SWHIDs (see Sect. 2.1.2) of all source code artifacts⁷ used in an empirical experiment to enable other scientists to reproduce the experiments later on [11].

Table 2.1 summarizes the above points, comparing with a few other infrastructures designed specifically for software engineering studies.

In the rest of this section, we briefly describe the datasets that Software Heritage curates and maintains to the benefit of other researchers in the field of empirical software engineering.

Before detailing the available datasets, we recall that building and maintaining the Software Heritage infrastructure that is instrumental to build them is a multi-million dollar undertaking. We are making significant efforts to reduce the burden on the prospective users, by providing dumps at regular intervals that help with reproducibility and making them directly available on public clouds like AWS. Researchers can then either run their queries directly on the cloud, paying only the compute time, or download them for exploiting them on their own infrastructure.

To give an idea of the associated costs for researchers, SQL queries on the graph datasets described in Sect. 2.2.1.1 can be performed using Amazon Athena for approximately 5\$ per Terabyte scanned at the time of writing. For example, an

⁵ For the sake of conciseness, we do not differentiate here between repeatability, reproducibility, and replicability; we refer instead the interested reader to the ACM terminology available at <https://www.acm.org/publications/policies/artifact-review-and-badging-current>. To varying degrees, Software Heritage helps with all of them, specifically when it comes to mitigating the risk of losing availability to source code artifacts.

⁶ For comparison: the total size of source code archived at Software Heritage is ≈ 1 PiB at the time of writing.

⁷ As it will become clear in Sect. 2.1.2, in most cases, it will be sufficient to list the SWHIDs of the releases or repository snapshots.

Table 2.1 Comparison of infrastructures for performing empirical software engineering research

\ Infrastructure Criteria	SWH on S3	SWH graph (on premise)	Boa	World of Code
host organisation	non profit foundation		research project	research project
purpose	archival & research			
scope	all platforms			
dataset	open			
access	free			
query language	SQL Athena	graph API		
cost	5\$/TB	10K\$ setup		
dataset update frequency	6 months		≈ yearly	≈ yearly
reproducibility	named dataset		named dataset	named dataset
	SWHID list			

SQL query to get the 4 topmost commit verb stems from over two billion revisions scans approximately 100 Gigabytes of data and provides the user with the answer in less than a minute, for a total cost of approximately 50 cents, a minimal fraction of the cost one would incur to set up an on-premise solution.

When SQL queries are not enough (typically when a graph traversal is needed), the cost of a cloud solution may quickly become significant, and it may become more interesting to set up an on-premise solution. The full compressed graph dataset can be exploited using medium range server grade machines that are accessible for less than 10,000 dollars.

2.2.1 *The Software Heritage Datasets*

The entire content of the Software Heritage archive is publicly available to researchers interested in conducting empirical experiments on it. At the simplest level, the content of the archive can be browsed interactively using the Web user interface at <https://archive.softwareheritage.org/> and accessed programmatically using the Web API documented at <https://archive.softwareheritage.org/api/>. These access paths, however, are not really suitable for large-scale experiments due to protocol overheads and rate limitations enforced to avoid depleting archive resources. To address this, several curated datasets are regularly extracted from the archive and made available to researchers in ways suitable for mass analysis.

2.2.1.1 The Software Heritage *Graph* Dataset

Consider the data model discussed in Sect. 2.1.1. The entire archive graph is exported periodically as the *Software Heritage Graph Dataset* [38]. Note the word “graph” in there, which characterizes this particular dataset and denotes that *only* the graph is included in the dataset, up to the content of its leave nodes, excluded (for size reasons). This dataset is suitable for analyzing source code *metadata*, including commit information, filenames, software provenance, code reuse, etc., but not for textual analyses of archived source code, as that is stored in graph leaves (see the blob dataset below for how to analyze actual code).

The data model of the graph dataset is a relational representation of the archive Merkle DAG, with one “table” for each type of node: blobs, directories, commits, releases, and snapshots. Each table entry is associated with several attributes, such as multiple checksums for blobs, filenames and attributes for directories, commit messages and timestamps for commits, etc. The full schema is documented at <https://docs.softwareheritage.org-devel/swh-dataset/graph/schema.html>.

In practical terms, the dataset is distributed as a set of Apache ORC files for each table, suitable for loading into scale-out columnar-oriented data processing frameworks such as Spark and Hadoop. The ORC files can be downloaded from the public Amazon S3 bucket <s3://softwareheritage/graph/>. At the time of writing, the most recent dataset export has timestamp 2022-12-07, so, for example, the first ORC files of the commit table are:

```

1 $ aws s3 ls --no-sign-request
   ↪ s3://softwareheritage/graph/2022-12-07/orc/revision/
2 2022-12-13 17:41:44 3099338621 revision-[...]-f9492019c788.orc
3 2022-12-13 17:32:42 4714929458 revision-[...]-42da526d2964.orc
4 2022-12-13 17:57:00 3095895911 revision-[...]-9c46b558269d.orc
5 [...]
```

The current version of the dataset contains metadata for 13 billion source code files, ten billion directories, 2.7 billion commits, 35 million releases, and 200 million VCS snapshots, coming from 189 M software origins. The total size of the dataset is 11 TiB, which makes it unpractical for use on personal machines, as opposed to research clusters. For that reason, hosted versions of the dataset are also available on Amazon Athena and Azure Databricks. The former can be queried using the Presto distributed SQL engine without having to download the dataset locally. For example, the following query will return the most common first word stems used in commit messages across more than 2.7 billion commits in just a few seconds:

Listing 2.1 Simple SQL query to get the 4 topmost commit verb stems

```

1 SELECT count(*) as c,word FROM (
2   SELECT word_stem(lower(split_part(trim(from_utf8(message)), ' ', 1)))
   ↪ as word
3   from revision WHERE length(message) < 1000000
4   WHERE word != ''
5   GROUP BY word ORDER BY c DESC LIMIT 4
```

For the curious reader, the (unsurprising) results of the query look like this:

Count	Word
294 369 196	updat
178 738 450	merg
152 441 261	add
113 924 516	fix

More complex queries and examples can be found in previous work [38]. For more details about using the graph dataset, we refer the reader to its technical documentation at <https://docs.softwareheritage.org/devel/swh-dataset/graph/>.

In addition to the research highlights presented later in this chapter, the Software Heritage graph dataset has been used as subject of study for the 2020 edition of the MSR (Mining Software Repositories) mining challenge, where students and young researchers in software repository mining have used it to solve the most interesting mining problems they could think of. To facilitate their task “teaser” datasets –data samples with exactly the same shape of the full dataset, but much smaller– have also been produced and can be used by researchers to understand how the dataset works before attacking its full scale. For example, the `popular-3k-python` teaser contains a subset of 2.197 popular repositories tagged as implemented in Python and being popular according to various metrics (e.g., GitHub stars, PyPI download statistics, etc.). The `gitlab-all` teaser corresponds to all public repositories on www.gitlab.com (as of December 2020), an often neglected ecosystem of Git repositories, which is interesting to study to avoid (or compare against) GitHub-specific biases.

2.2.1.2 Accessing Source Code Files

All source code files archived by Software Heritage are spread across multiple copies and also mirrored to the public Amazon S3 bucket www.s3://softwareheritage/content/. From there, individual files can be retrieved, possibly massively and in parallel, based on their SHA1 checksums. Starting from SWHIDs, one can obtain SHA1 checksums using the `content` table of the graph dataset and then access the associated content as follows:

```

1 $ aws s3 cp s3://softwareheritage/content/\
2   8624bcdcae55baeef00cd11d5dfcfa60f68710a02 .
3 download: s3://softwareheritage/content/8624b[...] to ./8624b[...]
4
5 $ zcat 8624bcdcae55baeef00cd11d5dfcfa60f68710a02 | sha1sum
6 8624bcdcae55baeef00cd11d5dfcfa60f68710a02 -
7
8 $ zcat 8624bcdcae55baeef00cd11d5dfcfa60f68710a02 | head
9           GNU GENERAL PUBLIC LICENSE
10          Version 3, 29 June 2007
11 [...]
```

Note that individual files are gzip-compressed to further reduce storage size.

The general empirical analysis workflow involves three simple steps: identify the source code files of interest using the metadata available in the graph dataset, obtain their checksum identifiers, and then retrieve them in batch and in parallel from public cloud providers. This process scales well up to many million files to be analyzed. For even larger-scale experiments, e.g., analyzing *all* source code files archived at Software Heritage, research institutions may consider setting up a local mirror of the archive.⁸

2.2.1.3 License Dataset

In addition to datasets that correspond to the actual content of the archive, i.e., source code artifacts as encountered among public code, it is also possible to curate *derived* datasets extracted from Software Heritage for the specific use cases or fields of endeavors.

As of today one notable example of such a derived dataset is the *license blob dataset*, available at <https://annex.softwareheritage.org/public/dataset/license-blobs/> and described in [51]. It consists of the largest known dataset of the complete texts of free/open-source software (FOSS) license variants. To assemble it, the authors collected from the Software Heritage archive all versions of files whose names are commonly used to convey licensing terms to software users and developers, e.g., COPYRIGHT, LICENSE, etc. (the exact pattern is documented as part of the dataset replication package).

The dataset consists of 6.5 million unique license files that can be used to conduct empirical studies on open-source licensing, training of automated license classifiers, natural language processing (NLP) analyses of legal texts, as well as historical and phylogenetic studies on FOSS licensing. Additional metadata about shipped license files are also provided, making the dataset ready to use in various empirical software engineering contexts. Metadata include file length measures, detected MIME type,

⁸ See <https://www.softwareheritage.org/mirrors/> for details, including storage requirements. At the time of writing, a full mirror of the archive requires about 1 PiB of raw storage.

detected SPDX [45] license (using ScanCode [35], a state-of-the-art tool for license detection), example origin (e.g., GitHub repository), and oldest public commit in which the license appeared. The dataset is released as open data as an archive file containing all deduplicated license files, plus several portable CSV files for metadata, referencing files via cryptographic checksums.

2.3 Research Highlights

The datasets discussed in the previous section have been used to tackle research problems in empirical software engineering and neighboring fields. In this section, we provide brief highlights on the most interesting of them.

2.3.1 Enabling Artifact Access and (Large-Scale) Analysis

Applied research in various fields has been conducted to ease access to such a huge amount of data as the Software Heritage archive for empirical researchers. This kind of research is not, strictly speaking, research *enabled* by the availability of the archive to solve software engineering problems but rather research *motivated* by the practical need of empowering fellow scholars to do so empirically.

As a first example, *SwfFS* (*Software Heritage File System*) [2] is a virtual filesystem developed using the Linux FUSE (Filesystem in User SpacE) framework that can “mount,” in the UNIX tradition, selected parts of the archive as if they were available locally as part of your filesystem. For example, starting from a known SWHID, one can, for instance:

```

1 $ mkdir swhfs
2 $ swf fs mount swhfs/ # mount the archive
3 $ cd swhfs/
4
5 $ cat archive/swf:1:c839dea9e8e6f0528b468214348fee8669b305b2
#include <stdio.h>
6
7 int main(void) {
8     printf("Hello, World!\n");
9 }
10
11
12 $ cd archive/swf:1:dir:1fee702c7e6d14395bbf\
13 5ac3598e73bcbf97b030
14 $ ls | wc -l
15 127
16 $ grep -i antenna THE_LUNAR_LANDING.s | cut -f 5
17 # IS THE LR ANTENNA IN POSITION 1 YET
18 # BRANCH IF ANTENNA ALREADY IN POSITION 1

```

In the second example, we are grepping through the code of Apollo 11 guidance computer code, searching for reference to antennas.

SwhFS allows to bridge the gap between classic UNIX-like mining tools, which are often relied upon in the fields of empirical software engineering and software repository mining, as well as by the Software Heritage APIs. However, it is not suitable for very-large-scale mining, due to the fact that seemingly local archive access pass through the public Internet (with caching, but still not suitable for large experiments).

swh-graph [7] is a way to enable such large-scale experiments. The main idea behind its approach is to adapt and apply *graph compression* techniques, commonly used for graphs such as the Web or social network, to the Merkle DAG graph that underpins the Software Heritage archive. The main research question addressed by *swh-graph* is:

Is it possible to efficiently perform software development history analyses at ultra-large scale, on a single, relatively cheap machine?

The answer is affirmative. As of today, the entire structure of the Software Heritage graph (≈ 25 billion nodes + 350 billion edges) can be loaded in memory on a single machine equipped with ≈ 200 GiB of RAM (roughly: 100 GiB for the direct graph + 100 GiB for its transposed version, which is useful in many research use cases such as source code provenance analysis). While significant and not suitable for personal machines, such requirements are perfectly fine for server-grade hardware on the market, with an investment of a few thousand US dollars in RAM. Once loaded, the entire graph can be visited in full in just a few hours and a single path visit from end to end can be performed in tens of nanoseconds per edge, close to the cost of a single memory access per edge.

In practical terms, this allows to answer queries such as “where does this file/directory/commit come from” or “list the entire content of this repositories” in fractions of seconds (depending just on the size of the answer, in most cases) fully in memory, without having to rely on a DBMS or even just disk accesses. The price to pay for this is that (1) the compressed graph representation loaded in memory is derived from the main archive and not incremental (it should periodically be recreated) and (2) only the graph structure and selected metadata fit in RAM; others reside on disk (although using compressed representations as well [37]) and need to be memory mapped for efficient access to frequently accessed information.

Finally, the archive also provides interesting use cases for database research. Recently, Wellenzohn et al. [48] has used it to develop a novel type of *content-and-structure (CAS) index*, capable of indexing over time the evolution of properties associated to specific graph nodes, e.g., a file content residing at a given place in a repository changing over time together with its metadata (last modified timestamp, author, etc.). While these indexes existed before, their deployment and efficient pre-population were still unexplored at this scale.

2.3.2 Software Provenance and Evolution

The peculiar structure –a fully deduplicated Merkle DAG– and comprehensiveness of the Software Heritage archive provides a powerful observation point and tool on the evolution and provenance of public source code artifacts. In particular, it is possible, on the one hand, to navigate the Merkle DAG *backward*, starting from any artifact of interest (source code file, directory, commit, etc.), to obtain the full list of all places (e.g., different repositories) where it has ever been distributed from. This area is referred to as *software provenance* and, in its simplest form, deals with determining the *original* (i.e., earliest) distribution place of a given artifact. More generally, being able to identify *all* places that have ever distributed it provides a way to measure software impact, track out-of-date copies or clones, and more.

Rousseau et al. [42] used the Software Heritage archive in a study that made two relevant contributions in this area. First, exploiting the fact that commits are deduplicated and timestamped, they verified that *the growth of public code* as a whole, at least as it is observable from the lenses of Software Heritage, is *exponential*: the amount of original commits (i.e., commits never observed before throughout the archive, no matter the origin repository) in public source code doubles every ≈ 30 months and has been doing so for the past 20 years. If, on the other hand, we look at original source code blobs (i.e., files whose content has never been observed before throughout the archive, up to that point in time), the overall trends remain the same, and only the speed changes: the amount of original public source code blobs doubles every ≈ 22 months. These are remarkable findings for software evolution, which had never been verified before at this macro-level.

Second, the authors showed how to model software provenance compactly, so that it can be represented (space-)efficiently at the scale of Software Heritage and can be used to address software audit use cases, which are commonplace in open-source compliance scenarios, merger and acquisition audits, etc.

2.3.3 Software Forks

The same characteristics that enable studying the evolution and provenance of public code artifacts can be leveraged to study the global ecosystem of software forks. In particular, the fact that commits are fully deduplicated allows to detect forks –both collaborative ones, such as those created on social coding platforms to submit pull requests, and hostile ones used to bring the project in a different direction– even when they are not created on the same platform. It is possible to detect the fork of a project originally created on GitHub and living on GitLab.com, or vice versa, based on the fact that the respective repositories share a common commit history.

This is important as a methodological point for empirical researchers, because by relying only on platform metadata (e.g., the fact that a repository *has been created* by clicking on a “fork” button on the GitHub user interface), researchers

risk overlooking other relevant forks. In previous work, Zacchiroli [51] provided a classification of the type of forks based on whether they are explicitly tracked as being forks of one another on a coding platform (Type 1 forks), they share at least one commit (Type 2), or they share a common root directory at some point in their histories (Type 3). He empirically verified that between 3.8% and 16%, forks could be overlooked by considering only type 1 forks, possibly inducing a significant threat to validity for empirical analyses of forks that strive to be comprehensive.

Along the same lines, Bhattacharjee et al. [6] (participants in the MSR 2020 mining challenge) focus their analyses on “cross-platform” forks between GitHub and GitLab.com, identifying several cases in which interesting development activity can be found on GitLab even for projects initially mirrored from GitHub.

2.3.4 Diversity, Equity, and Inclusion

Diversity, equity, and inclusion studies (DE&I) are hot research topics in the area of human aspects of software engineering. Free/open-source software artifacts, as archived by Software Heritage, provides a wealth of data for analyzing evolutionary DE&I trends, in particular in the very long term and at the largest scale attempted thus far.

A recent study by Zacchiroli [50] has used Software Heritage to explore the trend of *gender diversity* over a time period of 50 years. He conducted a longitudinal study of the population of contributors to publicly available software source code, analyzing 1.6 billion commits corresponding to the development history of 120 million projects, contributed by 33 million distinct authors over a period of 50 years. At this scale, authors cannot be interviewed to ask their gender, nor cross-checking with large-enough complementary dataset was possible. Instead, automated detection based on census data from around the world and the gender-guesser tool (benchmarked for accuracy and popular in the field) was used. Results show that while the amount of commits by female authors remains very low overall (male authors have contributed more than 92% of public code commits over the 50 years leading to 2019), there is evidence of a stable long-term increase in their proportion over all contributions (with the ratio of commits by female authors growing steadily over 15 years, reaching in 2019 for the first time 10% of all contributions to public code).

Follow-up studies have added the spatial dimension investigating the *geographic gap* in addition to the gender one. Rossi et al. [40] have developed techniques to detect the geographic origin of authors of Software Heritage commit, using as signals the time-zone offset and the author names (compared against census date from around the world). Results over 50 years of development history show evidence of the early dominance of North America in open-source software, later joined by Europe. After that period, the geographic diversity in public code has been constantly increasing, with more and more contributions coming from Central and South Asia (comprising India), Russia, Africa, and Central and South America.

Finally, Rossi et al. [41] put together the temporal and spatial dimension using the Software Heritage archive to investigate whether the ratio of women participation over time shows notable differences around the world, at the granularity of 20 macro-regions. The main result is that the increased trend of women participation is indeed a worldwide phenomenon, with the exception of specific regions of Asia where the increase is either slowed or completely flat. An incidental finding is also worth noting: the positive trend of increased women participation observed up to 2019 has been reversed by the COVID-19 pandemic, with the *ratio* of both contributions by and active female authors decreasing sharply starting at about that time.

These studies show how social aspects of software engineering can benefit from large-scale empirical studies and how they can be enabled by comprehensive, public archives of public code artifacts.

2.4 Building the Software Pillar of Open Science

Software plays a key role in scientific research, and it can be a tool, a result, and a research object. [...] France will support the development and preservation of source code – inseparable from the support of humanity’s technical and scientific knowledge – and it will, from this position, continue its support for the Software Heritage universal archive. So as to create an ecosystem that connects code, data and publications, the collaboration between the national open archive HAL, the national research data platform Recherche Data Gouv, the scientific publishing sector and Software Heritage will be strengthened.

Second french national plan for open science, July 2021 [22]

Software is *an essential research output*, and its source code implements and describes data generation and collection, data visualization, data analysis, data transformation, and data processing with a level of precision that is not met by scholarly articles alone. Publicly accessible software source code allows a better understanding of the process that leads to research results, and open-source software allows researchers to build upon the results obtained by others, provided proper mechanisms are put in place to make sure that software source code is preserved and that it is referenced in a persistent way.

There is a growing general awareness of its importance for supporting the research process [9, 25, 46]. Many research communities focus on the issue of *scientific reproducibility* and strongly encourage making the source code of the artefact available by archiving it in publicly accessible long-term archives; some have even put in place mechanisms to assess research software, like the *Artefact Evaluation* process introduced in the ESEC-FSE 2011 conference and now widely adopted by many computer science conferences [10] and the ACM *Artifact Review and Badging* program.⁹ Others raise the complementary issues of making

⁹ <https://www.acm.org/publications/policies/artifact-review-badging>.

it easier to discover existing research software and giving academic credit to its authors [26, 30, 44].

These important issues are similar in spirit to those that led to the now-popular FAIR data movement [49], and as a first step, it is important to clearly identify the different concerns that come into play when addressing software, and in particular its source code, as a research output. They can be classified as follows:

Archival: software artifacts must be properly **archived**, to ensure we can *retrieve* them at a later time;

Reference: software artifacts must be properly **referenced** to ensure we can *identify* the exact code, among many potentially archived copies, used for reproducing a specific experiment;

Description: software artifacts must be equipped with proper **metadata** to make it easy to *find* them in a catalog or through a search engine;

Citation: research software must be properly **cited** in research articles in order to give *credit* to the people that contributed to it.

These are not only different concerns but also *separate* ones. Establishing proper *credit* for contributors via *citations* or providing proper metadata to *describe* the artifacts requires a *curation* process [3, 8, 14] and is way more complex than simply providing stable, intrinsic identifiers to *reference* a precise version of a software source code for reproducibility purposes [4, 15, 26]. Also, as remarked in [4, 25], research software is often a thin layer on top of a large number of software dependencies that are developed and maintained outside of academia, so the usual approach based on institutional archives is not sufficient to cover all the software that is relevant for reproducibility of research.

In this section, we focus on the first two concerns, *archival* and *reference*, that can be addressed fully by leveraging the Software Heritage archive, but we also describe how Software Heritage contributes through its ecosystem to the two other concerns.

2.4.1 Software in the Scholarly Ecosystem

Presenting results in journal or conference articles has always been part of the research activity. The growing trend, however, is to include software to support or demonstrate such results. This activity can be a significant part of academic work and must be properly taken into account when researchers are evaluated [4, 44].

Software source code developed by researchers is only *a thin layer* on top of the complex web of software components, most of them developed outside of academia, which are necessary to produce scientific results: as an example, Fig. 2.5 shows the many components that are needed by the popular `matplotlib` library [27].

As a consequence, scholarly infrastructures that support software source code written in academia must go the extra mile to ensure they adopt standards and provide mechanisms that are compatible with the ones used by tens of millions

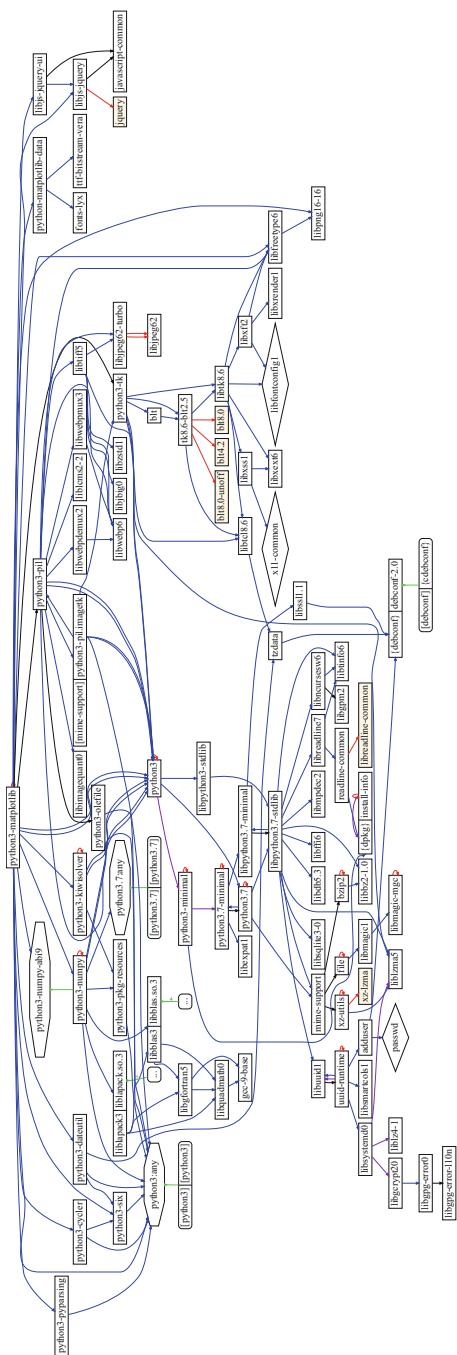


Fig. 2.5 Direct and indirect dependencies for a specific python package (matplotlib). In blue are the Python dependencies; in red are the “true” system dependencies incurred by python (e.g., the libc or libjpeg62); in green are some dependencies triggered by the package management system but which are very likely not used by python (e.g., adduser or dpkg)

of non-academic software developers worldwide. They also need to ensure that the large amount of software components that are developed outside academia, but are relevant for research activities, are properly taken into account.

Over the recent years, there have been a number of initiatives to add support for software artifacts in the scholarly world, which fall short of satisfying these requirements. They can be roughly classified in two categories:

overlays on public forges provide links from articles to the source code repository of the associated software artifact as found on a public code hosting platform (forge); typical examples are websites like <https://paperswithcode.com/>, <http://www.replicabilitystamp.org/>, and the *Code and data* links recently introduced in ArXiv.org.

deposits in academic repositories take snapshots of a given state of the source code, usually in the form of a .zip or .tar file, and store it in the repository exactly like an article or a dataset, with an associated publisher identifier; a typical example in computer science is the ACM Digital Library, but there are a number of general academic repositories where software artefacts have been deposited, like FigShare and Zenodo.

The approaches in the first category rely on code hosting platforms that do not guarantee *persistence* of the software artifact: the author of a project may alter, rename, or remove it, and we have seen that code hosting platforms can be discontinued or decide to remove large amount of projects.¹⁰

The approaches in the second category do take into account persistence, as they archive software snapshots, but they lose the *version control history* and do not provide the *granularity* needed to reference the internal components of a software artifact (directories, files, snippets).

And none of the initiatives in these categories provide a means to properly archive and reference the numerous external dependencies of software artefacts.

This is where Software Heritage comes into play for Open Science, by providing an *archive designed for software* that provides persistence, preserves the version control history, supports granularity in the identification of software artefacts and their components, and harvests all publicly available source code.

The differences described above are summarized in Table 2.2, where we only consider infrastructures in the second category described above, as they are the only one assuming the mission to archive their contents. We also take into account additional features found in academic repositories, like the possibility of depositing content with an *embargo* period, which is not possible on Software Heritage, and the existence of a curation process to obtain qualified metadata, which is currently out of scope of Software Heritage.

¹⁰ Google Code and Gitorious.org were shut down in 2015, Bitbucket removed support for the Mercurial VCS in 2020, and in 2022, Gitlab.com considered removing all projects inactive for more than a year.

Table 2.2 Comparison of infrastructures for archiving research software. The various granularities of identifiers are abbreviated with the same convention used in SWHIDs (*snp* for snapshot, etc.), plus the abbreviation *frg* that stands for the ability to identify a *code fragment*

Criteria \ Infrastructure	Software Heritage	ACM DL	HAL	Figshare	Zenodo
identifier	intrinsic	extrinsic	extrinsic + <i>intrinsic</i> (via SWH)	extrinsic	extrinsic
granularity	snp, rel, rev dir, cnt, frg harvest	dir	dir	dir	rel, dir
archival	deposit save code now	deposit	deposit	deposit	deposit
history	full VCS	no	no	no	releases
browse code	yes	no	no	no	no
scope	universal	discipline	academic	academic	academic
embargo	no	no	yes	yes	yes
curation	no	yes	yes	no	no
integration	BitBucket, SourceForge, GitHub, Gitea, Gitlab, HAL, etc.		SWH		GitHub

2.4.2 Extending the Scholarly Ecosystem Architecture to Software

In the framework of the European Open Science Cloud initiative (EOSC), a working group has been tasked in 2019 to bring together representatives from a broad spectrum of scholarly infrastructures to study these issues and propose concrete ways to address them. The result, known as the EOSC Scholarly Infrastructures for Research Software (SIRS) report [16], was published in 2020 and provides a detailed analysis of the existing infrastructures, their relationships, and the workflows that are needed to properly support software as a research result on par with publications and data.

Figure 2.6 presents the main categories of identified actors:

Scholarly repositories: services that have as one of their primary goals the long-term preservation of the digital content that they collect.

Academic publishers: organizations that prepare submitted research texts, possibly with associated source code and data, to produce a publication and manage the dissemination, promotion, and archival process. Software and data can be part of the main publication or assets given as supplementary materials depending on the policy of the journal.

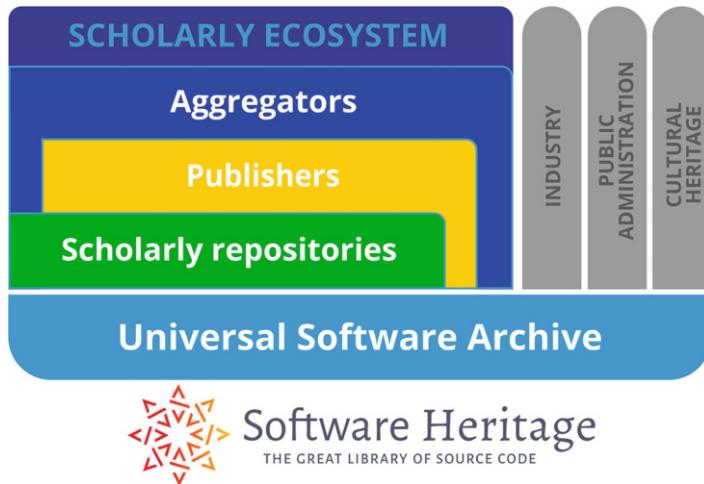


Fig. 2.6 Overview of the high-level architecture of scholarly infrastructures for research software, as described in the EOSC SIRS report

Aggregators: services that collect information about digital content from a variety of sources with the primary goal of increasing its discoverability and possibly adding value to this information via processes like curation, abstraction, classification, and linking.

These actors have a long history of collaboration around research articles, with well-defined workflows and collaborations. The novelty here is the fact that to handle research software, it is no longer possible to work in isolation inside the academic world, for the reasons explained previously: one needs a means to share information and work with other ecosystems where software is present, like in industry and public administration.

One key finding of the EOSC SIRS Report is that Software Heritage provides the shared basic architectural layer that allows to interconnect all these ecosystems, because of its unified approach to archiving and referencing all software artefacts, independently of the tools or platforms used to develop or distribute the software involved.

2.4.3 *Growing Technical and Policy Support*

In order to take advantage of the services provided by Software Heritage in this setting, a broad spectrum of actions have been started and are ongoing. We briefly survey here the ones that are most relevant at the time of writing.

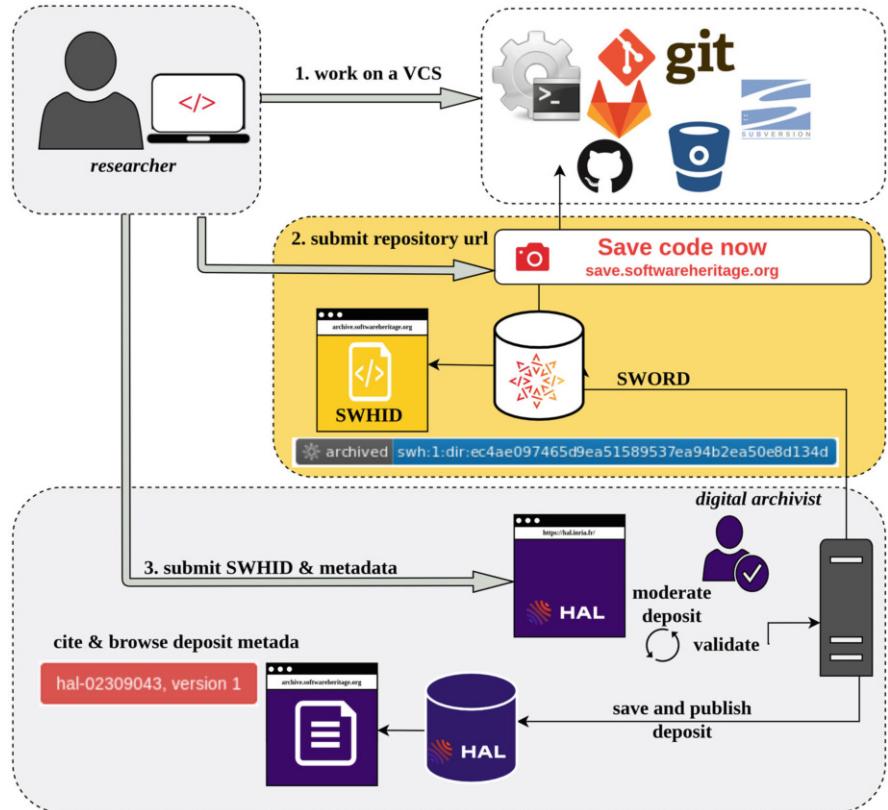


Fig. 2.7 Overview of the interplay between HAL and Software Heritage for research software

At the national level, France has developed a multi-annual plan on Open Science that includes research software [21, 22] and consistently implemented this plan through a series of steps that range from technical development to policy measures.

On the technical side, the French national open-access repository HAL [14] (analogous to the popular arXiv service¹¹) has been integrated with the Software Heritage archive. The integration allows researchers to have their software projects archived and referenced in Software Heritage, while curated rich metadata and citation information are made available on HAL [14], with a streamlined process depicted in Fig. 2.7.

On the policy side, the second French national plan for open science [22], published in July 2021, prescribes the use of Software Heritage and HAL for all the research software produced in France, and Software Heritage is now listed

¹¹ <https://arxiv.org>.

in the official national roadmap of research infrastructures published in February 2022 [23].

This approach is now being pushed forward at the European level, through funding for consortia that will build the needed connectors between Software Heritage and several infrastructures and technologies used in academia, using the French experience as a reference. Most notably, the FAIRCORE4EOSC [19] European project includes plans to build connectors with scholarly repository systems like Dataverse [47] and InvenioRDM [28] (the white-label variant of Zenodo), publishers like Dagstuhl [43] and Episcience [18], and aggregators like swMath [20] and OpenAire [36].

2.4.4 *Supporting Researchers*

The growing awareness about the importance of software as a research output will inevitably bring new recommendations for research activity, which will eventually become obligations for researchers, as we have seen with publications and data.

Through the collaboration with academic infrastructures, Software Heritage is striving to develop mechanisms that minimize the extra burden for researchers, and we mention here a few examples.

A newly released extension, codename `updateswh`, for the popular Web browsers Firefox and Google Chrome allows to trigger archival in just one click for any public repository hosted on Bitbucket, GitLab (.com and any instance), GitHub, and any instance of Gitea. It also allows to access in one click the archived version of the repository and obtain the associated SWHID identifier.

Integration with Web hooks is available for a variety of code hosting platforms, including Bitbucket, GitHub, GitLab.com, and SourceForge, as well as for instances of GitLab and Gitea, which enable owners of projects hosted on those platforms to trigger archival automatically on any new release, reducing the burden on researchers even more.

Software Heritage will try to detect and parse intrinsic metadata present in software projects independently of the format chosen, but we see the value of standardizing on a common format. This is why, with all academic platforms, we are working with, we are advocating the use of `codemeta.json`, a machine readable file based on the CodeMeta extension of `schema.org`, to retrieve automatically metadata associated to software artifact, in order to avoid the need for researchers to fill forms when declaring software artifacts in academic catalogs, following the schema put in place with the HAL national open-access portal.

Finally, we have released the `biblatex-software` bibliographic style extension to make it easy to cite software artefacts in publications written using the popular `LATEX` framework.

2.5 Conclusions and Perspectives

In conclusion, the Software Heritage ecosystem is a useful resource for both software engineering studies and for Open Science. As an infrastructure for research on software engineering, the archive provides numerous benefits. The SWHID intrinsic identifiers make it easier for researchers to identify and track software artifacts across different repositories and systems. The uniform data structure used by the archive abstracts away all the details of software forges and package managers, providing a standardized representation of software code that is easy to use and analyze. The availability of the open datasets makes it possible to tailor experiments to one's needs and improves their reproducibility. An obvious direction at the time of writing is to leverage Software Heritage's extensive source code corpus for pre-training large language models. Future collaborations may lead to integrate functionalities like the domain-specific language from the Boa project or the efficient data structures of the World of Code project, enabling researchers to run more specialized queries and achieve more detailed insights.

Regarding the Open Science aspect, Software Heritage already offers the reference archive for all publicly available research software. The next step is to interconnect it with a growing number of scholarly infrastructures, which will increase reproducibility of research in all fields and support software citation directly from the archive, contributing to increasing visibility of research software.

Going forward, we believe that Software Heritage will provide a unique observatory for the whole software development ecosystem, both in academia and outside of it. We hope that with growing adoption, it will play an increasingly valuable role in advancing the state of software engineering research and in supporting the software pillar of open science.

References

1. Abramatic, J.F., Di Cosmo, R., Zacchiroli, S.: Building the universal archive of source code. *Commun. ACM* **61**(10), 29–31 (2018). <https://doi.org/10.1145/3183558>
2. Allançon, T., Pietri, A., Zacchiroli, S.: The software heritage filesystem (SwfFS): integrating source code archival with development. In: International Conference on Software Engineering (ICSE). IEEE, Piscataway (2021). <https://doi.org/10.1109/ICSE-Companion52605.2021.00032>
3. Allen, A., Schmidt, J.: Looking before leaping: creating a software registry. *J. Open Res. Softw.* **3**(e15) (2015). <https://doi.org/10.5334/jors.bv>
4. Alliez, P., Di Cosmo, R., Guedj, B., Girault, A., Hacid, M.S., Legrand, A., Rougier, N.: Attributing and referencing (research) software: best practices and outlook from INRIA. *Comput. Sci. Eng.* **22**(1), 39–52 (2020). <https://doi.org/10.1109/MCSE.2019.2949413>. Available from <https://hal.archives-ouvertes.fr/hal-02135891>
5. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform resource identifier (URI): Generic syntax. RFC 3986, RFC Editor (2005)

6. Bhattacharjee, A., Nath, S.S., Zhou, S., Chakraborti, D., Roy, B., Roy, C.K., Schneider, K.A.: An exploratory study to find motives behind cross-platform forks from software heritage dataset. In: International Conference on Mining Software Repositories (MSR), pp. 11–15. ACM, New York (2020). <https://doi.org/10.1145/3379597.3387512>
7. Boldi, P., Pietri, A., Vigna, S., Zaccarioli, S.: Ultra-large-scale repository analysis via graph compression. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 184–194. IEEE, Piscataway (2020). <https://doi.org/10.1109/SANER48275.2020.9054827>
8. Bönisch, S., Brickenstein, M., Chrapary, H., Greuel, G., Sperber, W.: swMATH - a new information service for mathematical software. In: MKM/Calculemus/DML. Lecture Notes in Computer Science, vol. 7961, pp. 369–373. Springer, Berlin (2013)
9. Borgman, C.L., Wallis, J.C., Mayernik, M.S.: Who's got the data? Interdependencies in science and technology collaborations. In: Computer Supported Cooperative Work (CSCW), vol. 21, pp. 485–523 (2012). <https://doi.org/10.1007/s10606-012-9169-z>
10. Childers, B.R., Fursin, G., Krishnamurthi, S., Zeller, A.: Artifact evaluation for publications (Dagstuhl Perspectives Workshop 15452). Dagstuhl Rep. **5**(11), 29–35 (2016). <https://doi.org/10.4230/DagRep.5.11.29>
11. Di Cosmo, R.: Archiving and referencing source code with software heritage. In: International Conference on Mathematical Software (ICMS). Lecture Notes in Computer Science, vol. 12097, pp. 362–373. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-52200-1_36
12. Di Cosmo, R., Zaccarioli, S.: Software Heritage: Why and how to preserve software source code. In: International Conference on Digital Preservation (iPRES) (2017)
13. Di Cosmo, R., Gruenpeter, M., Zaccarioli, S.: Identifiers for digital objects: the case of software source code preservation. In: International Conference on Digital Preservation (iPRES) (2018). <https://doi.org/10.17605/OSF.IO/KDE56>
14. Di Cosmo, R., Gruenpeter, M., Marmol, B.P., Monteil, A., Romary, L., Sadowska, J.: Curated Archiving of Research Software Artifacts: lessons learned from the French open archive (HAL) (2019). Presented at the International Digital Curation Conference. Submitted to IJDC
15. Di Cosmo, R., Gruenpeter, M., Zaccarioli, S.: Referencing source code artifacts: a separate concern in software citation. Comput. Sci. Eng. **22**(2), 33–43 (2020). <https://doi.org/10.1109/MCSE.2019.2963148>
16. Di Cosmo, R., Lopez, J.B.G., Abramatic, J.F., Graf, K., Colom, M., Manghi, P., Harrison, M., Barborini, Y., Tenhunen, V., Wagner, M., Dalitz, W., Maassen, J., Martinez-Ortiz, C., Ronchieri, E., Yates, S., Schubotz, M., Candela, L., Fenner, M., Jeangirard, E.: Scholarly Infrastructures for Research Software. European Commission. Directorate General for Research and Innovation (2020). <https://doi.org/10.2777/28598>
17. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: International Conference on Software Engineering (ICSE), pp. 422–431 (2013)
18. Episciences. <https://www.episciences.org>. Accessed 15 April 2023
19. FAIRCORE4EOSC project. <https://faircore4eosc.eu>. Accessed 15 April 2023
20. FIZ Karlsruhe GmbH: swMATH mathematical software. <https://swmath.org> (2023). Accessed 15 April 2023
21. French Ministry of Research and Higher Education: French National Plan for Open Science. <https://www.enseignementsup-recherche.gouv.fr/fr/le-plan-national-pour-la-science-ouverte-les-resultats-de-la-recherche-scientifique-ouverts-tous-49241> (2018)
22. French Ministry of Research and Higher Education: French second national plan for open science: Support and opportunities for universities' open infrastructures and practices. <https://www.enseignementsup-recherche.gouv.fr/fr/le-plan-national-pour-la-science-ouverte-2021-2024-vers-une-generalisation-de-la-science-ouverte-en-48525> (2021)
23. French Ministry of Research and Higher Education: Feuille de route nationale des infrastructures de recherche. <https://www.enseignementsup-recherche.gouv.fr/fr/feuille-de-route-nationale-des-infrastructures-de-recherche> (2022)

24. Heckman, J.: Varieties of selection bias. *Am Eco Rev* **80**(2), 313–318 (1990)
25. Hinsen, K.: Software development for reproducible research. *Comput. Sci. Eng.* **15**(4), 60–63 (2013). <https://doi.org/10.1109/MCSE.2013.91>
26. Howison, J., Bullard, J.: Software in the scientific literature: problems with seeing, finding, and using software mentioned in the biology literature. *J. Assoc. Inf. Sci. Technol.* **67**(9), 2137–2155 (2016). <https://doi.org/10.1002/asi.23538>
27. Hunter, J.D.: Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* **9**(3), 90–95 (2007). <https://doi.org/10.1109/MCSE.2007.55>
28. Invenio: InvenioRDM. <https://inveniosoftware.org/products/rdm/>. Accessed 15 April 2023
29. Ivie, P., Thain, D.: Reproducibility in scientific computing. *ACM Comput. Surv.* **51**(3), 63:1–63:36 (2018). <https://doi.org/10.1145/3186266>
30. Lamprecht, A.L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., Dominguez Del Angel, V., van de Sandt, S., Ison, J., Martinez, P.A., McQuilton, P., Valencia, A., Harrow, J., Psomopoulos, F., Gelpi, J.L., Chue Hong, N., Goble, C., Capella-Gutierrez, S.: Towards FAIR principles for research software. *Data Sci.* **3**(1), 37–59 (2020). <https://doi.org/10.3233/DS-190026>
31. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., Mockus, A.: World of code: an infrastructure for mining the universe of open source VCS data. In: International Conference on Mining Software Repositories (MSR), pp. 143–154. IEEE, Piscataway (2019). <https://doi.org/10.1109/MSR.2019.00031>
32. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Advances in Cryptology (CRYPTO), pp. 369–378 (1987). https://doi.org/10.1007/3-540-48184-2%5C_32
33. Messerschmitt, D.G., Szyperski, C.: Software Ecosystem: Understanding an Indispensable Technology and Industry. MIT Press, Cambridge (2003)
34. Mockus, A.: Amassing and indexing a large sample of version control systems: towards the census of public source code history. In: International Working Conference on Mining Software Repositories (MSR), pp. 11–20. IEEE, Piscataway (2009). <https://doi.org/10.1109/MSR.2009.5069476>
35. nexB: ScanCode. <https://www.aboutcode.org/projects/scancode.html>. Accessed 15 April 2023
36. Openaire. <https://www.openaire.eu>. Accessed 15 April 2023
37. Pietri, A.: Organizing the graph of public software development for large-scale mining. (organisation du graphe de développement logiciel pour l'analyse à grande échelle). Ph.D. Thesis, University of Paris (2021)
38. Pietri, A., Spinellis, D., Zacchiroli, S.: The Software Heritage graph dataset: public software development under one roof. In: International Conference on Mining Software Repositories (MSR), pp. 138–142 (2019). <https://doi.org/10.1109/MSR.2019.00030>
39. Quinlan, S., Dorward, S.: Venti: a new approach to archival data storage. In: Conference on File and Storage Technologies (FAST). USENIX Association, Berkeley (2002). <https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage>
40. Rossi, D., Zacchiroli, S.: Geographic diversity in public code contributions: an exploratory large-scale study over 50 years. In: International Conference on Mining Software Repositories (MSR), pp. 80–85. ACM, New York (2022). <https://doi.org/10.1145/3524842.3528471>
41. Rossi, D., Zacchiroli, S.: Worldwide gender differences in public code contributions (and how they have been affected by the COVID-19 pandemic). In: International Conference on Software Engineering – Software Engineering in Society Track (ICSE-SEIS), pp. 172–183. ACM, New York (2022). <https://doi.org/10.1109/ICSE-SEIS55304.2022.9794118>
42. Rousseau, G., Di Cosmo, R., Zacchiroli, S.: Software provenance tracking at the scale of public source code. *Empirical Software Eng.* **25**(4), 2930–2959 (2020). <https://doi.org/10.1007/s10664-020-09828-5>
43. Schloss Dagstuhl. <https://www.dagstuhl.de>. Accessed 15 April 2023
44. Smith, A.M., Katz, D.S., Niemeyer, K.E.: Software citation principles. *PeerJ Comput. Sci.* **2**, e86 (2016). <https://doi.org/10.7717/peerj-cs.86>
45. Stewart, K., Odence, P., Rockett, E.: Software package data exchange (SPDX) specification. *IFOSS L. Rev.* **2**, 191 (2010)

46. Stodden, V., LeVeque, R.J., Mitchell, I.: Reproducible research for scientific computing: tools and strategies for changing the culture. *Comput. Sci. Eng.* **14**(4), 13–17 (2012). <https://doi.org/10.1109/MCSE.2012.38>
47. The Dataverse Project. <https://dataverse.org>. Accessed 15 April 2023
48. Wellenzohn, K., Böhlen, M.H., Helmer, S., Pietri, A., Zacchiroli, S.: Robust and scalable content-and-structure indexing. *VLDB J.* (2022). <https://doi.org/10.1007/s00778-022-00764-y>
49. Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., Bouwman, J., Brookes, A.J., Clark, T., Crosas, M., Dillo, I., Dumon, O., Edmunds, S., Evelo, C.T., Finkers, R., Gonzalez-Beltran, A., Gray, A.J., Groth, P., Goble, C., Grethe, J.S., Heringa, J., 't Hoen, P.A., Hooft, R., Kuhn, T., Kok, R., Kok, J., Lusher, S.J., Martone, M.E., Mons, A., Packer, A.L., Persson, B., Rocca-Serra, P., Roos, M., van Schaik, R., Sansone, S.A., Schulz, E., Sengstag, T., Slater, T., Strawn, G., Swertz, M.A., Thompson, M., van der Lei, J., van Mulligen, E., Velterop, J., Waagmeester, A., Wittenburg, P., Wolstencroft, K., Zhao, J., Mons, B.: The FAIR guiding principles for scientific data management and stewardship. *Sci. Data* **3**(1), 160018 (2016). <https://doi.org/10.1038/sdata.2016.18>
50. Zacchiroli, S.: Gender differences in public code contributions: a 50-year perspective. *IEEE Softw.* **38**(2), 45–50 (2021). <https://doi.org/10.1109/MS.2020.3038765>
51. Zacchiroli, S.: A large-scale dataset of (open source) license text variants. In: International Conference on Mining Software Repositories (MSR), pp. 757–761. ACM, New York (2022). <https://doi.org/10.1145/3524842.3528491>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Chapter 1

An Introduction to Software Ecosystems



Tom Mens and Coen De Roover

Abstract This chapter defines and presents the kinds of software ecosystems that are targeted in this book. The focus is on the development, tooling, and analytics aspects of “software ecosystems,” i.e., communities of software developers and the interconnected software components (e.g., projects, libraries, packages, repositories, plug-ins, apps) they are developing and maintaining. The technical and social dependencies between these developers and software components form a socio-technical dependency network, and the dynamics of this network change over time. We classify and provide several examples of such ecosystems, many of which will be explored in further detail in the subsequent chapters of the book. The chapter also introduces and clarifies the relevant terms needed to understand and analyze these ecosystems, as well as the techniques and research methods that can be used to analyze different aspects of these ecosystems.

1.1 The Origins of Software Ecosystems

Today, *software ecosystems* are considered an important domain of study within the general discipline of *software engineering*. This section describes its origins, by summarizing the important milestones that have led to its emergence. Figure 1.1 depicts these milestones chronologically.

The software engineering discipline emerged in 1968 as the result of a first international conference [126], sponsored by the NATO Science Committee, based on the realization that more disciplined techniques, engineering principles, and theoretical foundations were urgently needed to cope with the increasing complexity, importance, and impact of software systems in all sectors of economy and industry.

T. Mens (✉)
University of Mons, Mons, Belgium
e-mail: tom.mens@umons.ac.be

C. De Roover
Vrije Universiteit Brussel, Etterbeek, Belgium
e-mail: Coen.De.Roover@vub.be

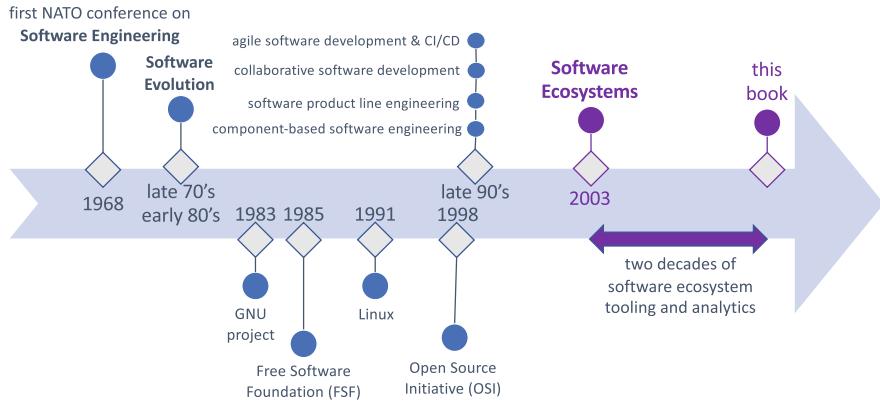


Fig. 1.1 Milestones that contributed to the domain of research (analytics) and development (tooling) of *software ecosystems*

Even the key idea of *software reuse* [61, 97], which suggests to reduce time-to-market, cost, and effort when building software while at the same time increasing reuse, productivity, and quality, is as old as the software engineering discipline itself. During the aforementioned conference, Malcolm Douglas McIlroy proposed to face increasing software complexity by building software through the reuse of high-quality software components [112].

In the late 1970s, awareness increased that the development of large-scale software needs to *embrace change* as a key aspect of the development process [186]. This has led Manny Lehman to propose the so-called laws of *software evolution*, focusing on how industrial software systems continue to evolve after their first deployment or public release [19, 101, 102]. The software evolution research domain is still thriving today [114, 116], with two dedicated annual conferences: the IEEE International Conference on Software Maintenance and Evolution (ICSME) and the IEEE Software Analysis, Evolution and Reengineering Conference (SANER).

Another important factor having contributed to the popularity of software ecosystems is the emergence and ever-increasing importance of *free software* and *open-source software* (OSS) since the early 1980s, partly through the creation of the GNU project¹ in 1983 and the Free Software Foundation (FSF) in 1985 by Richard Stallman, as well as the creation of the Linux operating system in 1991. Strong open-source advocates such as Eric Raymond [144] further contributed to the popularity through the creation of the Open-Source Initiative (OSI) in 1998, and by contrasting cathedral-style closed development process models with the bazaar-style open development process models for open-source and free software in which the code is publicly developed over the Internet. This bazaar-style model evolved into

¹ <https://www.gnu.org>.

geographically distributed *global software development* [73, 78] models, supported by the immensely popular *social coding platforms* [43] such as GitHub, GitLab, Gitea, and BitBucket.

In parallel, the importance of software reuse in the late 1990s gave rise to additional subfields of software engineering such as the domain of *component-based software engineering* [96, 159], focusing on methods and principles for composing large systems from loosely coupled and independently evolving software components. Around the same time, it was joined by another subfield, called *software product line engineering* [179], which explicitly aims to enable developing closely related software products using a process modelled after product line manufacturing, separating the *domain engineering* phase of producing reusable software artefacts that are common to the product family, from the *application engineering* phase that focuses on developing concrete software applications that exploit the commonalities of the reusable artefacts created during the domain engineering phase. Software product lines have allowed many companies to reduce costs while at the same time increasing quality and time to market, by providing a product line platform and architecture that allows to scale up from the development and maintenance of individual software products to the maintenance of entire families of software products. However, these product families still remain within the organizational boundaries of the company.

Around the same time, the lightweight and iterative process models known as *agile software processes* started to come to the forefront, with a user-centric vision requiring adaptive and continuous software change. Different variants, such as Scrum [150] and eXtreme Programming (XP) [17], led to the foundation of the Agile Alliance and the creation of the agile manifesto [18]. In support of agile software processes, various development practices and tools for *continuous integration and delivery* (CI/CD) emerged later on in the decade.

Since the 2003 seminal book by Messerschmitt and Szyperski [117], *software ecosystems* have become an active topic of research in software engineering. As argued by Jan Bosch [27, 28], software ecosystems expand upon software product lines by allowing companies to cross the organizational boundaries and make their software development platforms available to third parties that, in turn, can contribute to the popularity of the produced software through externally developed components and applications. The key point of software ecosystems is that software products can no longer be considered or maintained in isolation, since they have become heavily interconnected.

1.2 Perspectives and Definitions of Software Ecosystems

Messerschmitt and Szyperski [117] were arguably among the first to use the term *software ecosystem* and defined it rather generically as *a collection of software products that have some given degree of symbiotic relationships*. Since then, the

research literature has provided different definitions of software ecosystems, from many different perspectives.

From an **ecological perspective**, several researchers have tried to exploit the analogy between software ecosystems and natural ecosystems. The term software ecosystem quite obviously originates from its ecological counterpart of biological ecosystems that can be found in nature, in a wide variety of forms (e.g., rainforests, coral reefs, deserts, mountain zones, and polar ecosystems). In 1930, Roy Clapham introduced the term *ecosystem* in an ecological context to denote the *physical and biological components of an environment considered in relation to each other as a unit* [184]. These components encompass all living organisms (e.g., plants, animals, microorganisms) and physical constituents (e.g., light, water, soil, rocks, minerals) that interact with one another in a given environment. Dunghana et al. [53] compared the characteristics of natural and software ecosystems. Mens [113] provided a high-level historical and ecological perspective on how software ecosystems evolve. Moore [123] and Iansiti and Levien [82] focused on the analogy between business ecosystems and ecology.

From an **economic and business perspective**, Jansen et al. [83] provide a more precise definition: *a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them*. In a similar vein, Bosch et al. [27] say that a software ecosystem *consists of a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs*. Hanssen [76] defines it as *a networked community of organizations, which base their relations to each other on a common interest in a central software technology*. An excellent entry point to this business-oriented viewpoint on software ecosystems is the book edited by Jansen et al. [84]. In contrast, the chapters in the current book focus mostly on the complementary technical and social perspectives.

From a more **technical perspective**, the focus is on technical aspects such as the software tools that are being used (e.g., version control systems, issue and bug trackers, social coding platforms, integrated development environments, programming languages) and the software artefacts that are being used and produced (e.g., source code, executable code, tests, databases, documentation, trace logs, bug and vulnerability reports). Within this technical perspective, Lungu [105] defined a software ecosystem as *a collection of software projects that are developed and evolve together in the same environment*. The notion of *environment* can be interpreted rather broadly. The environment can correspond to a software-producing organization, including the tools and libraries used by this organization for developing its software projects, as well as the clients using the developed software projects. It can correspond to an academic environment, composed of software projects developed and maintained by students and researchers in research units. It can also correspond to an entire OSS community consisting of geographically dispersed project collaborators focused around similar philosophies or goals.

From a **social perspective**, the focus is on the social context and network structure that emerges as a result of the collaboration dynamics and interaction

between the different contributors to the projects that belong to the software ecosystem. This social structure is at least as important as the technical aspects and includes the various stakeholders that participate in the software ecosystem, such as developers, end users, project managers, analysts, designers, software architects, security specialists, legal consultants, clients, QA teams, and many more. Chapter 5 focuses on these social aspects from an emotion analysis viewpoint.

Manikas [111] combined all these perspectives into a single all-encompassing definition of a software ecosystem as *the interactions of a set of actors on top of a common technological platform that results in a number of software solutions or services. Each actor is motivated by a set of interests or business models and connected to the rest of the actors and the ecosystem as a whole with symbiotic relationships, while the technological platform is structured in a way that allows the involvement and contribution of the different actors.*

1.3 Examples of Software Ecosystems

Following the wide diversity of definitions of software ecosystem, the kinds of software ecosystems that have been studied in recent research are equally diverse. An interesting entry point into how the research literature on software ecosystems has been evolving over the years are the many published systematic literature reviews, such as [14, 29, 110, 111, 151].

Without attempting to be complete, Table 1.1 groups into different categories some of the most popular examples of software ecosystems that have been studied in the research literature. These categories are not necessarily disjoint, since software ecosystems tend to contain different types of components that can be studied from different viewpoints.

The remaining subsections provide more details for each category, illustrating the variety of software ecosystems that have been studied and providing examples of well-known ecosystems and empirical research that has been conducted on them.

1.3.1 Digital Platform Ecosystems

Hein et al. [77] define a *digital platform ecosystem* as a software ecosystem that *comprises a platform owner that implements governance mechanisms to facilitate value-creating mechanisms on a digital platform between the platform owner and an ecosystem of autonomous complementors and consumers*. This is in line with the previously mentioned definition by Bosch et al. [27] that a software ecosystem *consists of a software platform, a set of internal and external developers and a community of domain experts in service to a community of users that compose relevant solution elements to satisfy their needs*.

Table 1.1 Categories of software ecosystems

Category	Examples	Components	Contributors
digital platforms	mobile app stores, integrated development environments	mobile apps, software plug-ins, or extensions	third-party app or plug-in developers and their users
social coding platforms	SourceForge, GitHub, GitLab, Gitea, Bitbucket	software project repositories	software project contributors
component-based software ecosystems	software library registries (e.g., CRAN, npm, RubyGems, PyPi, Maven Central), OS package registries (e.g., Debian packages, Ubuntu package archive)	interdependent software packages	consumers and producers of software packages and libraries
software automation ecosystems	Docker Hub, Kubernetes, Ansible Galaxy, Chef Supermarket, Puppet Forge	container images, configuration and orchestration scripts, CI/CD pipelines and workflows	creators and maintainers of workflow automation, containerization and orchestration solutions
communication-oriented ecosystems	mailing lists, Stack Overflow, Slack	e-mail threads, questions, answers, messages, posts, etc.	programmers, developers, end users, researchers
OSS communities	Apache Software Foundation, Linux Foundation	OSS projects	community members, code contributors, project maintainers, end users

Well-known examples of digital platform ecosystems are the *mobile software ecosystems* provided by companies such as Microsoft, Apple, and Google. The company owns and controls an *app store* as a central platform to which other companies or individuals can contribute apps, which in turn can be downloaded and installed by mobile device users. The systematic mapping studies by de Lima Fontao et al. [51] and [156] report on the abundant research that has been conducted on these mobile software ecosystems.

Any software system that provides a mechanism for third parties to contribute plug-ins or extensions that enhance the functionalities of the system can be considered as a digital software ecosystem. Examples of these are configurable text editors such as Emacs and Vim and integrated software development environments (IDEs) such as IntelliJ IDEA, VS Code, NetBeans, and Eclipse. The latter ecosystem in particular has been the subject of quite some research on its evolutionary dynamics (e.g., [4, 30–33, 91, 115, 128, 166]). These examples show that digital platform ecosystems are not necessarily controlled by a single company. In many cases, they are managed by a consortium, foundation, or open-source community. For example, NetBeans is controlled by the Apache Foundation, and Eclipse is controlled by the Eclipse Foundation.

Another well-known digital platform ecosystem is WordPress, the most popular content management system in use today, which features a plugin architecture and template system that enables third parties to publish themes and extend the core functionality. Um et al. [170] presented a recent study of this ecosystem. Yet another example is OpenStack, an open-source cloud computing platform involving more than 500 companies. This ecosystem has been studied by several researchers (e.g., [60, 162, 166, 193]).

1.3.2 Component-Based Software Ecosystems

A very important category of software ecosystems are so-called *component-based software ecosystems*. They constitute large collections of reusable software components, which often have many interdependencies among them [1]. Empirical studies on component-based software ecosystems tend to focus on the technicalities of dependency-based reuse, which differentiates them from studies on digital platform ecosystems, which have a more business-oriented and managerial focus.

As explained in Sect. 1.1, the idea of building software by reusing existing software components is as old as the software engineering discipline itself, since it was proposed by McIlroy in 1968 during the very first software engineering conference [112]. The goal was to reduce time-to-market, cost, and effort when building software while at the same time increasing reuse, productivity, and quality. This has given rise to a very important and abundant subfield of software engineering that is commonly referred to as component-based software engineering. Despite the large body of research in this field (e.g., [34, 97, 159]), it was not able to live up to its promises due to a lack of a standard marketplace for software components, combined with a lack of proper component models, terminology, and scalable tooling [95]. All of this has changed nowadays, probably due to a combination of the increasing popularity of OSS and the emergence of affordable cloud computing solutions.

Among the most important success stories of component-based software ecosystems are undoubtedly the many interconnected *software packages* for OSS operating systems such as the GNU Project since 1983, Linux since 1991, Debian since 1993 (e.g., [1, 36, 39, 40, 69]), and Ubuntu since 2004. They come with associated *package management systems* (or *package managers* for short) such as DPKG (since 1994) and APT (since 1998), which are systems that automate the process of selecting, installing (or removing), upgrading, and configuring those packages. Package managers typically maintain a database of software dependencies and version information to prevent software incompatibilities.

Another popular type of ecosystems of reusable components are *software libraries*. Software developers, regardless of whether they are part of an OSS community or software company, rely to a large extent on such reusable third-party *software libraries*. These library ecosystems tend to come with their own specific package managers and package registries and are available for all major

programming languages. Examples include the CPAN archive network (created in 1995 for the Perl programming language, the CRAN archive network (created in 1997) and Bioconductor for the R statistical programming language) [62, 138], npm and Bower for JavaScript [2, 41, 46, 47, 49, 190], PyPI for Python [171], Maven (Central) for JVM-based languages such as Java and Scala [22, 130, 155], Packagist for PHP, RubyGems for Ruby [49, 87, 190], NuGet for the .NET ecosystem [103], and the Cargo package manager and its associated crates registry for the Rust programming language [48, 149]. Another example is the Robot Operating System (ROS), the most popular middleware for robotics development, offering reusable libraries for building a robot, distributed through a dedicated package manager [59, 94, 136].

Decan et al. [48] studied and compared seven software library ecosystems for programming languages, focusing on the evolutionary characteristics of their package dependency networks. They observed that library dependency networks tend to grow over time but that some packages are more impactful than other. A minority of packages are responsible for most of the package updates, a small proportion of packages accounts for most of the reverse dependencies, and there is a high proportion of fragile packages due to a high number of transitive dependencies. This makes software library ecosystems prone to a variety of technical, dependency-related issues [2, 40, 45, 155], licensing issues [108], security vulnerabilities [6, 47, 190], backward compatibility [26, 44, 49], and reliance on deprecated components [41], as well as obsolete or outdated components [46, 100, 158, 191]. Versioning practices, such as the use of semantic versioning, can be used to a certain extent to reduce some of these risks [44, 55, 98, 130]. Library ecosystems also face many social challenges, such as how to attract and retain contributors and how to avoid contributor abandonment [42].

1.3.3 Web-Based Code Hosting Platforms

The landscape of web-based code hosting platforms has seen many important changes over the last two decades, as can be seen in Fig. 1.2. SourceForge was created in 1999 as a centralized web-based platform for hosting and managing the version history of free and OSS projects. It used to be a very popular data source for empirical research (e.g., [79, 93, 129, 147]). This is no longer the case today, as the majority of OSS projects have migrated to other hosting platforms. Google started running a similar open-source project hosting service, called Google Code, in 2006 but shut it down in January 2016. The same happened to Gitorious, which ran from 2008 to 2015.

GitHub replaced Google Code as the most popular and largest hosting platform for open-source (and commercial) software projects that use the git version control system. Other alternatives such as Bitbucket (also created in 2008), GitLab (created in 2014), and the likes are much less popular for hosting OSS projects. Even older is Gitee (created in 2013), an online git forge mainly used in China for hosting open-

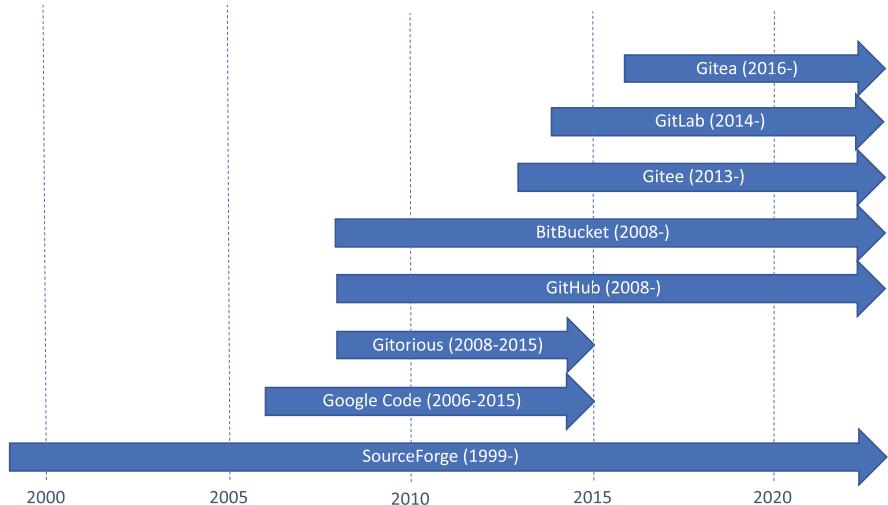


Fig. 1.2 Historical overview of source code hosting platforms

source software. A relatively new contender in the field is Gitea, created in 2016 and funded by the Open Source Collective.

GitHub maintains historical information about hundreds of millions of OSS repositories and has been the subject of many empirical studies focusing on different aspects [88]. GitHub is claimed to be the first *social coding* platform [43], as it was the first hosting platform to provide a wide range of mechanisms and associated visualizations to increase collaboration by making socially significant information visible: watching, starring, commits, issues, pull requests, and commenting. Being an enabler of social coding, the social aspects around GitHub projects have been studied extensively [135, 168], including communication patterns [134], collaboration through pull requests [72, 142, 187], variation in contributor workload [172], gender and tenure diversity [173, 174], geographically distributed development [143, 160, 177], socio-technical alignment between projects [25], the impact of gamification on the behavior of software developers [120], and sentiment and emotion analysis [74, 86, 139, 153, 180, 185]. The latter will be presented in more detail in Chap. 5. The phenomenon of project forking has also been actively studied in the context of GitHub [23, 85, 194], as will be discussed in more detail in Chap. 6. The automation of development activities in GitHub projects has also been studied, such as the use of CI/CD tools [20, 67, 174], and the use of development bots [3, 66, 178, 181]. The latter perspective on the ecosystem will be discussed in Chap. 8. The same chapter also explains how GitHub can be studied from the point of view of a digital platform ecosystem (cf. Sect. 1.3.1), as it offers a marketplace of apps and Actions that can be provided by third parties.

1.3.4 Open-Source Software Communities

Quite some research on software ecosystems has focused on collections of OSS projects maintained by decentralized communities of software developers. Such OSS ecosystems have clear advantages over closed, proprietary software ecosystems. For example, their openness guarantees the accessibility to all. Following the adagio that “given enough eyeballs, all bugs are shallow” [145], OSS ecosystems benefit from a potentially very large number of people that can report bugs, review the code, and identify potential security issues. Provided that the software licenses being used are compatible, organizations and companies can save money by relying on OSS components rather than reinventing the wheel and developing those components themselves.

At the downside, OSS ecosystems and their constituent components are frequently maintained on a volunteer basis by unpaid developers. This imposes an increased risk of unmaintained components or slow response time. Organizations that rely on OSS ecosystems could significantly reduce these risks by financially sponsoring the respective communities of OSS developers. Many fiscal and legal initiatives for doing so exist, such as the Open Collective, the Open Source Collective, and the Open Collective Foundation.

OSS ecosystems are often controlled, maintained, and hosted by a nonprofit software foundation. A well-known example is the *Apache Software Foundation* (www.apache.org). It hosts several hundreds of OSS projects, involving tens of thousands of code contributors. This ecosystem has been a popular subject of research (e.g., [16, 35, 38, 119, 161]). Another example is the *Linux Foundation* (www.linuxfoundation.org), whose initial goal was to support the development and evolution of the Linux operating system but nowadays hosts hundreds of OSS projects with hundreds of thousands of code contributors. As can be expected, the OSS project communities of third-party components that surround a digital platform ecosystem (cf. Sect. 1.3.1) also tend to be managed by nonprofit foundations. For example, the Eclipse Foundation controls the Eclipse plug-ins, the WordPress Foundation controls the WordPress plug-ins, and the Open Infrastructure Foundation manages the OpenStack projects.

Much in the same way as public OSS ecosystems, there exists a multitude of entirely private and company-controlled software ecosystems. We defer to the book by Jansen et al. [84] that focuses on the business aspects of such commercial software ecosystems. Given their proprietary nature, they have been much less the subject of quantitative empirical research studies, but it is likely that such private ecosystems share many of the characteristics known to OSS ecosystems. As a matter of illustration of such ecosystems, among the countless available examples, we mention Wolfram’s Mathematica and MathWorks’ MATLAB with their large collections of –often third-party– add-ons and the ecosystem surrounding SAP, the world’s largest software vendor of enterprise resource planning solutions.

1.3.5 *Communication-Oriented Ecosystems*

The previous categories of software ecosystems have in common that the main components they focus on are *technical* code-related software artefacts (e.g., software library packages and their metadata, mobile software applications, software plug-ins, project repositories, application code and tests, software containers, configuration scripts).

The current category focuses on what we will refer to as *communication-oriented ecosystems*, in which the main component is some *social* communication artefact that is shared among members of a software community through some communication channel. Examples of these are mailing lists, developer discussion fora, question-and-answer (Q&A) platforms such as Stack Overflow, and modern communication platforms such as Slack and Discord. Each of them constitute software ecosystems in their own right. A particularity of these ecosystems is that the main components they contain (e.g., questions, answers, posts, e-mail, and message threads) are mostly based on unstructured or semi-structured text. As a consequence, extracting and analyzing relevant information from them requires specific techniques based on Natural Language Processing (NLP). These “social programmer ecosystems” [127] have been analyzed by researchers for various reasons, mostly from a social viewpoint.

Mailing Lists Mailing lists are a common communication medium for software development teams, although they are gradually being replaced by more modern communication technologies. As the same person may have multiple email addresses, disambiguation techniques are often required to uniquely identify a given team member [183]. They have been the subject of multiple empirical studies (e.g., [75, 188]). Some of these studies have tried to identify personality traits or emotions expressed through e-mails [99, 146, 167].

Discussion Fora Software development discussion fora support mass communication and coordination among distributed software development teams [157]. They are a considerable improvement over mailing lists in that they provide browse and search functions, as well as a platform for posting questions within a specific domain of interest and for receiving expert answers to these questions.

A generic, and undoubtedly the most popular, discussion forum is Stack Overflow, dedicated to questions and answers related to computer programming and software development. It belongs to the Stack Exchange network, providing a range of websites covering specific topics. Such Q&A platforms can be considered as a software ecosystem where the “components” are questions and their answers (including all the metadata that comes with them), and the contributor community consists of developers that are asking questions and experts that provide answers to these questions. The Stack Overflow ecosystem has been studied for various purposes and in various ways [5, 8, 13, 15, 109, 124, 125, 175, 176, 188, 192]. The open dataset SOTorrent has been made available on top of a data dump with all posts from 2018 to 2020 [10–12]. Some researchers [52, 127, 169] have applied sentiment

and emotion analysis techniques on data extracted from Stack Overflow. We refer to Chap. 5 for a more detailed account on the use of such techniques in software ecosystems.

Next to generic discussion fora such as Stack Overflow, some software project communities prefer to use project-specific discussion fora. This is, for example, the case for Eclipse. Nugroho et al. [128] present an empirical analysis of how this forum is being used by its participants.

Modern Communication Platforms Several kinds of modern communication platforms, such as Slack and Discord, are increasingly used by software development teams. Lin et al. [104] reported how Slack facilitates messaging and archiving, as well as the creation of automated integrations with external services and bots to support the work of software development teams.

1.3.6 Software Automation Ecosystems

Another category of software ecosystems is what we would refer to as software automation ecosystems. They revolve around technological solutions that aim to automate part of the management, development, packaging, deployment, delivery, configuration, and orchestration of software applications, often through cloud-based platforms. We can mention at least three categories: containerization solutions, orchestration tools based on Infrastructure as Code, and tools for automating DevOps and CI/CD.

Containerization Containerization allows developers to package all (software and data) components of their applications into so-called containers, which are lightweight, portable, and self-contained executable software environments that can run on any operating system or cloud platform. By isolating the software applications from the underlying hardware infrastructure, they become easier to manage and more resilient to change. Docker is the most popular containerization tool, and it comes with multiple online registries to store, manage, distribute, and share containers (e.g., Google Container Registry, Amazon ECR, JFrog Container Registry, RedHat's Quay, and of course Docker Hub). While each of these registries come with their own set of features and benefits, Docker Hub is by far the largest of these registries. The corresponding ecosystem is studied in Chap. 9 and more specifically in Sect. 9.2.

Infrastructure Management Through *Infrastructure as Code* (IaC), infrastructure management tools enable automating the provisioning, configuration, deployment, scaling, and load balancing of the machines used in a digital infrastructure. Different infrastructure management tools have been proposed, including Ansible, Chef, and Puppet. Each of them comes with their own platform or registry for sharing configuration scripts (Ansible Galaxy, Chef Supermarket, and Puppet Forge). Sharma et al. [152] studied best practices in Puppet configuration code, analyzing 4,621

Puppet repositories for the presence of implementation and design configuration smells. Opdebeeck et al. conversely studied variable-related [131] and security-related [132] bad smells in Ansible files respectively. The Ansible Galaxy ecosystem has been an active subject of study in general, as will be shown in Chap. 9 and more specifically Sect. 9.3.

DevOps and CI/CD Collaborative distributed software development processes, especially for software projects hosted on social coding platforms, tend to be streamlined and automated using continuous integration, deployment, and delivery tools (CI/CD), which are a key part of DevOps practices. CI/CD tools enable project maintainers to specify project-specific workflows or pipelines that automate many repetitive and error-prone human activities that are part of the development process. Examples are test automation, code quality analysis, dependency management, and vulnerability detection. A wide range of CI/CD tools exist (e.g., Jenkins, Travis, CircleCI, GitLab CI/CD, and GitHub Actions to name just a few). Coming with a registry or marketplace of reusable workflow components that facilitate the creation and evolution of workflows, an ecosystem has formed around many of these tools. In particular, Chap. 8 will focus on the ecosystems surrounding GitHub Actions, the integrated CI/CD service of GitHub. Since its introduction, the CI/CD landscape on GitHub has radically changed [50, 92, 182].

1.4 Data Sources for Mining Software Ecosystems

The Mining Software Repositories (MSR) research community relies on a wide variety of publicly accessible raw data, APIs or other data extraction tools, data dumps, curated datasets, and data processing tools (e.g., dedicated parsers) depending on the specific purpose and needs of the research being conducted.

The Pros These data sources and their associated tooling form a gold mine for empirical research in software engineering, and they have allowed the MSR field to thrive. Relying on existing publicly accessible data substantially reduces the laborious and error-prone effort of the data extraction and processing phases of empirical research. As such, it has allowed researchers and software practitioners to learn a great deal about software engineering practices in the field and how to improve these practices. Moreover, this allows multiple researchers to rely on the same data, facilitating comparison and reproducibility of research results [68].

The Cons At the same time, these data sources and tools come with a variety of negative consequences, such as:

- Existing data and tools can quickly become obsolete, as it is difficult and effort-intensive to keep up with changes in the original data source or in the APIs required to access them. Many initiatives to create and maintain data extraction tools or curated datasets have been discontinued, mainly due to a lack

of continued funding or because the original maintainers have abandoned the initiative due to career changes.

- Ethical, legal, or privacy reasons may prevent specific parts of the data of interest to be made available [65]. Examples are proprietary copyrighted source code or personal information that cannot be revealed due by GDPR regulations.
- Specific analyses may need specific types of data that are not readily available in existing datasets, requiring the creation of new datasets or the extension of existing ones. Talking from a personal experience, it often takes several months of effort to obtain, preprocess, validate, curate, and improve the quality of the obtained data. Not doing so may lead to results that are inaccurate, biased, not replicable, or not generalizable to other situations.
- Existing datasets may not be appropriate for specific analyses, because of how the data has been gathered or filtered. As an illustration of this problem, suppose, for example, that we want to analyze the effort spent by human contributors in some software ecosystem, based on an available dataset containing contributor accounts and their associated activities over time. If this dataset does not distinguish between human accounts and automated bots, then the results will be biased by bot activities being considered as human activities, calling for the use of bot identification approaches and associated datasets (e.g., [66]).
- Research that is relying on raw data sources instead of curated datasets may reduce reproducibility since, unlike for a published dataset, there is no guarantee that the original data will remain the same after publication of the research results. For example, GitHub repositories may be deleted, and the history of a git repository may be changed at any time [24, 89].

The following subsections provide a list of data sources that have been used in empirical research on a wide variety of software ecosystems. This list is non-exhaustive, given the plethora of established and newly emerging ecosystems, data sources about them, and research studies on them.

1.4.1 Mining the GitHub Ecosystem

For git repositories hosted on the GitHub social coding platform, different ways have been proposed to source their data. GitHub provides public REST and GraphQL APIs to interact with its huge dataset of events and interaction with the hosted repositories. As an alternative, different attempts have been made to provide datasets and data dumps containing relevant data extracted from GitHub, with varying success:

- *GHArchive*² records, archives, and makes available the public GitHub timeline for public consumption and analysis. It is available on Google BigQuery, and

² <https://www.gharchive.org>.

it contains datasets, aggregated into hourly archives, based on 20+ event types, ranging from new commits and fork events to opening new tickets, commenting, and adding members to a project.

- In a similar way, *GHTorrent* aimed to obtain data from GitHub public repositories [70, 71], covering a large part of the activity from 2012 to 2019. The latest available data dump was created in March 2021,³ and the initiative has been discontinued altogether.
- *TravisTorrent* was a dataset created in 2017 based on Travis CI and GitHub. It provides access to over 2.6 million Travis builds from more than 1,000 GitHub projects [21].

1.4.2 Mining the Java Ecosystem

Multiple datasets have been produced for use in studies on the ecosystem surrounding the Java programming language. The Qualitas Corpus [163], a curated dataset of Java software systems, aimed to facilitate reproducing these studies. Only two data dumps have been released, in 2010 and in 2013.

More recent datasets for Java focused on Apache’s Maven Central Repository, a software package registry maintaining a huge collection of libraries for the Java Virtual Machine. For example, Raemaekers et al. provide the Maven Dependency Dataset with metrics, changes, and a dependency graph for 148,253 jar files [140]. The dataset was used to study the phenomena of semantic versioning and breaking changes [141]. Mitropoulos et al. [118] provide a complementary dataset containing the FindBugs results for every project version included in the Maven Central Repository.

More recently, Benelallam et al. [22] created the Maven Dependency Graph, an open-source data set containing a snapshot of the whole Maven Central Repository taken on September 2018, stored in a temporal graph database modelling all dependencies. This dataset has been used for various purposes, such as the study of dependency bloat [155] and diversity [154].

1.4.3 Mining Software Library Ecosystems

Beyond the Java ecosystem, many software library ecosystems have been studied for a wide range of programming languages. For the purpose of analysing the dependency networks of these ecosystems, *Libraries.io* [90] has been used by several researchers (e.g., [48, 108, 158, 189, 190]). Five successive data dumps have been made available from 2017 to 2020, containing metadata from a wide range of

³ <http://ghtorrent-downloads.ewi.tudelft.nl/mysql/>.

different package managers. No more recent data dumps have been released since Tidelift decided to discontinue active maintenance of the dataset.

As a kind of successor to *Libraries.io*, the *Ecosyste.ms* project⁴ was started in 2022. Currently sponsored by the Open Collective,⁵ it focuses on expanding available data and APIs, as such providing a foundational basis for researchers to better analyze open-source software and for funders to better prioritize which projects need to be funded most. The *Ecosyste.ms* platform provides a shared collection of openly accessible services to support, sustain, and secure critical open-source software components. Each service comes with an openly accessible JSON API to facilitate the creation of new tools and services. The APIs and data structures are designed to be as generic as possible, to facilitate analyzing different data sources in an ecosystem-agnostic way. Some of the supported services include:

- An index of several millions of open-source packages from dozens of package registries (for programming languages and Linux distributions), with tens of thousands of new package versions being added on a daily basis.
- An index of the historical timeline of several billions of events that occurred across public git repositories (hosted on GitHub, GitLab, or Gitea) over many years, with hundreds of thousands of events being added on an hourly basis.
- An index of dozens of millions of open-source repositories and Docker projects and their dependencies originating from a dozen of different sources, with tens of thousands of new repositories being added on a daily basis.
- A range of services to provide software repository, contributor, and security vulnerability metadata, parse software dependency and licensing metadata, resolve software package dependency trees, generate diffs between package releases, and many more.

1.4.4 Mining Other Software Ecosystems

Beyond the data sources mentioned above, a wide variety of other initiatives to mine, analyze, or archive software ecosystems have been proposed through a plethora of datasets or data sources that are –or have been– available for researchers or practitioners.

Of particular relevance is the *Software Heritage* ecosystem [54]. It is the largest public software archive, containing the development history of billions of source code files from more than 180 million collaborative software development projects. Supported by a partnership with UNESCO, its long-term mission is to collect, preserve, and make easily accessible the source code of publicly available software.

⁴ <https://ecosyste.ms>.

⁵ <https://opencollective.com>.

It comes with its own filesystem [7] and graph dataset [137]. For more details, we refer to Chap. 2, which is entirely focused on this ecosystem.

World of Code (WoC) [106, 107] is another ambitious initiative to create a very large and frequently updated collection of historical data in OSS ecosystems. The provided infrastructure facilitates the analysis of technical dependencies, social networks, and their interrelationships. To this end, WoC provides tools for efficiently correcting, augmenting, querying, and analyzing that data—a foundation for understanding the structure and evolution of the relationships that drive OSS activities.

Boa [57, 58, 81] is yet another initiative to support the efficient mining of large-scale datasets of software repository data. Boa provides a domain-specific language and distributed computing infrastructure to facilitate this.

Many other attempts have been made in the past to create and support publicly available software datasets and platforms, but these are no longer actively maintained today. We mention some notable examples below. The *PROMISE* Software Engineering Repository is a collection of publicly available datasets to serve researchers in conducting predictive software engineering in a repeatable, verifiable, and refutable way [148]. *FLOSSmole* is another collaborative collection of OSS project data [80]. *Candoia* is a platform and ecosystem for building and sharing software repository mining tools and applications [164, 165]. *Sourcerer* is a research project aimed at exploring open-source projects and provided an open-source infrastructure and curated datasets for other researchers to use [9].

DebSources is a dataset containing source code and metadata spanning two decades of history related to the Debian Linux distribution until 2016 [37].

Jira is one of the most popular issue tracking systems (ITSS) in practice. A first Jira repository dataset was created in 2015, containing more than 700K issue reports and more than two million issue comments extracted from the Jira issue tracking system of the Apache Software Foundation, Spring, JBoss, and CodeHaus OSS communities [133]. A more recent dataset created in 2022 gathers data from 16 public Jira repositories containing 1822 projects and spanning 2.7 million issues with a combined total of 32 million changes, nine million comments, and one million issue links [121, 122].

1.5 The CHAOSS Project

In an introductory chapter on software ecosystems, it is indispensable to also mention the CHAOSS initiative (which is an acronym for Community Health Analytics in Open Source Software) [64].⁶ It is a Linux Foundation project aimed at better understanding OSS community health on a global scale [63]. Unhealthy OSS projects can have a negative impact on the community involved in them,

⁶ <https://chaoss.community>.

as well as on organizations that are relying on them. CHAOSS therefore focuses on understanding and supporting health through the creation of metrics, metrics models, and software development analytics tools for measuring and visualizing community health in OSS projects.

Two main OSS tools are proposed by CHAOSS to do so: *Augur* and *Grimoire-Lab* [56]. The latter is an open-source toolkit with support for extracting, visualizing, and analyzing activity, community, and process data from 30+ data sources related to code management, issues, code reviewing, mailing list, developer fora, and more. Perhaps one shortcoming of these tools is that they have not been designed to scale up to visualize or analyze health issues at the level of ecosystems containing thousands or even millions of interconnected projects.

1.6 Summary

This introductory chapter served as a first stepping stone for newcomers in the research field of software ecosystems. We aimed to provide the necessary material to get up to speed in this domain. After a historical account of where software ecosystems originated from, we highlighted the different perspectives on software ecosystems and their accompanying definitions. We categorized the different kinds of software ecosystems, providing many examples for each category.

Since the book to which this introductory chapter belongs focuses on software ecosystem tooling and analytics, we presented a rich set of data sources and datasets that have been or can be used for mining software ecosystems. Given that the field of software ecosystems is evolving at a rapid pace, it is difficult to predict the direction into which it is heading and the extent to which the current tools and data sources will evolve or get replaced in the future.

References

1. Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S.: Strong dependencies between software components. In: International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 89–99 (2009). <https://doi.org/10.1109/ESEM.2009.5316017>
2. Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E.: Why do developers use trivial packages? An empirical case study on npm. In: Joint Meeting on Foundations of Software Engineering (FSE), pp. 385–395 (2017). <https://doi.org/10.1145/3106237.3106267>
3. Abdellatif, A., Wessel, M., Steinmacher, I., Gerosa, M.A., Shihab, E.: BotHunter: an approach to detect software bots in GitHub. In: International Conference on Mining Software Repositories (MSR), pp. 6–17. IEEE Computer Society, Washington (2022). <https://doi.org/10.1145/3524842.3527959>
4. Abou Khalil, Z., Constantinou, E., Mens, T., Duchien, L.: On the impact of release policies on bug handling activity: a case study of Eclipse. J. Syst. Software **173** (2021). <https://doi.org/10.1016/j.jss.2020.110882>

5. Ahasanuzzaman, M., Asaduzzaman, M., Roy, C.K., Schneider, K.A.: CAPS: a supervised technique for classifying Stack Overflow posts concerning API issues. *Empir. Software Eng.* **25**(2), 1493–1532 (2020). <https://doi.org/10.1007/s10664-019-09743-4>
6. Alfadel, M., Costa, D.E., Shihab, E., Shihab, E.: Empirical analysis of security vulnerabilities in Python packages. In: International Conference on Software Analysis, Evolution and Reengineering (SANER) (2021). <https://doi.org/10.1109/saner50967.2021.00048>
7. Allançon, T., Pietri, A., Zacchioli, S.: The software heritage filesystem (SwHFS): integrating source code archival with development. In: International Conference on Software Engineering (ICSE). IEEE, Piscataway (2021). <https://doi.org/10.1109/ICSE-Companion52605.2021.00032>
8. Anderson, A., Huttenlocher, D., Kleinberg, J., Leskovec, J.: Discovering value from community activity on focused question answering sites: a case study of Stack Overflow. In: SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 850–858. ACM, New York (2012). <https://doi.org/10.1145/2339530.2339665>
9. Bajracharya, S., Ossher, J., Lopes, C.: Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Sci. Comput. Program.* **79**, 241–259 (2014). <https://doi.org/10.1016/j.scico.2012.04.008>
10. Baltes, S.: SOTorrent dataset (2021). <https://doi.org/10.5281/zenodo.4415593>
11. Baltes, S., Dumani, L., Treude, C., Diehl, S.: SOTorrent: reconstructing and analyzing the evolution of Stack Overflow posts. In: International Conference on Mining Software Repositories (MSR), pp. 319–330. ACM, New York (2018). <https://doi.org/10.1145/3196398.3196430>
12. Baltes, S., Treude, C., Diehl, S.: SOTorrent: studying the origin, evolution, and usage of Stack Overflow code snippets. In: International Conference on Mining Software Repositories (MSR), pp. 191–194. IEEE, Piscataway/ACM, New York (2019). <https://doi.org/10.1109/MSR.2019.00038>
13. Bangash, A.A., Sahar, H., Chowdhury, S., Wong, A.W., Hindle, A., Ali, K.: What do developers know about machine learning: a study of ML discussions on StackOverflow. In: International Conference on Mining Software Repositories (MSR), pp. 260–264 (2019). <https://doi.org/10.1109/MSR.2019.00052>
14. Barbosa, O., Alves, C.: A systematic mapping study on software ecosystems. In: International Workshop on Software Ecosystems (IWSECO), CEUR Workshop Proceedings, vol. 746, pp. 15–26 (2011)
15. Barua, A., Thomas, S.W., Hassan, A.E.: What are developers talking about? An analysis of topics and trends in Stack Overflow. *Empir. Software Eng.* **19**(3), 619–654 (2014). <https://doi.org/10.1007/s10664-012-9231-y>
16. Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S.: The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In: International Conference on Software Maintenance (ICSM), pp. 280–289 (2013). <https://doi.org/10.1109/ICSM.2013.39>
17. Beck, K.: Embracing change with extreme programming. *Computer* **32**(10), 70–77 (1999). <https://doi.org/10.1109/2.796139>
18. Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al.: Manifesto for agile software development. Technical report, Snowbird, UT (2001)
19. Belady, L.A., Lehman, M.M.: A model of large program development. *IBM Syst. J.* **15**(3), 225–252 (1976). <https://doi.org/10.1147/sj.153.0225>
20. Beller, M., Gousios, G., Zaidman, A.: Oops, my tests broke the build: an explorative analysis of Travis CI with GitHub. In: International Conference on Mining Software Repositories (MSR), pp. 356–367. IEEE, Piscataway (2017). <https://doi.org/10.1109/MSR.2017.62>
21. Beller, M., Gousios, G., Zaidman, A.: TravisTorrent: synthesizing Travis CI and GitHub for full-stack research on continuous integration. In: International Conference on Mining Software Repositories (MSR), pp. 447–450 (2017). <https://doi.org/10.1109/MSR.2017.24>

22. Benelallam, A., Harrand, N., Soto-Valero, C., Baudry, B., Barais, O.: The Maven dependency graph: a temporal graph-based representation of Maven Central. In: International Conference on Mining Software Repositories (MSR), pp. 344–348 (2019). <https://doi.org/10.1109/MSR.2019.00060>
23. Biazzini, M., Baudry, B.: May the fork be with you: novel metrics to analyze collaboration on GitHub. In: International Workshop on Emerging Trends in Software Metrics, pp. 37–43. ACM, New York (2014). <https://doi.org/10.1145/2593868.2593875>
24. Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., Germán, D.M., Devanbu, P.T.: The promises and perils of mining git. In: International Working Conference on Mining Software Repositories (MSR), pp. 1–10. IEEE, Piscataway (2009). <https://doi.org/10.1109/MSR.2009.5069475>
25. Blincoe, K., Harrison, F., Kaur, N., Damian, D.: Reference coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems. *Inform. Software Technol.* **110**, 174–189 (2019). <https://doi.org/10.1016/j.infsof.2019.03.005>
26. Bogart, C., Kästner, C., Herbsleb, J., Thung, F.: When and how to make breaking changes: policies and practices in 18 open source software ecosystems. *Trans. Software Eng. Methodol.* **30**(4) (2021). <https://doi.org/10.1145/3447245>
27. Bosch, J.: From software product lines to software ecosystems. In: International Software Product Line Conference (SPLC) (2009)
28. Bosch, J., Bosch-Sijtsema, P.: From integration to composition: on the impact of software product lines, global development and ecosystems. *J. Syst. Software* **83**(1), 67–76 (2010). <https://doi.org/10.1016/j.jss.2009.06.051>
29. Burström, T., Lahti, T., Parida, V., Wartiovaara, M., Wincent, J.: Software ecosystems now and in the future: a definition, systematic literature review, and integration into the business and digital ecosystem literature. *Trans. Eng. Manag.*, 1–16 (2022). <https://doi.org/10.1109/TEM2022.3216633>
30. Businge, J., Serebrenik, A., van den Brand, M.G.J.: Survival of Eclipse third-party plug-ins. In: International Conference on Software Maintenance (ICSM), pp. 368–377 (2012). <https://doi.org/10.1109/ICSM.2012.6405295>
31. Businge, J., Serebrenik, A., van den Brand, M.G.J.: Analyzing the Eclipse API usage: putting the developer in the loop. In: European Conference on Software Maintenance and Reengineering (CSMR), pp. 37–46. IEEE Computer Society, Washington (2013). <https://doi.org/10.1109/CSMR.2013.14>
32. Businge, J., Serebrenik, A., Brand, M.G.: Eclipse API usage: the good and the bad. *Software Qual. J.* **23**(1), 107–141 (2015). <https://doi.org/10.1007/s11219-013-9221-3>
33. Businge, J., Kawuma, S., Openja, M., Bainomugisha, E., Serebrenik, A.: How stable are Eclipse application framework internal interfaces? In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 117–127 (2019). <https://doi.org/10.1109/SANER.2019.8668018>
34. Caldiera, G., Basili, V.: Identifying and qualifying reusable software components. *Computer* **24**(2), 61–70 (1991). <https://doi.org/10.1109/2.67210>
35. Calefato, F., Lanubile, F., Vasilescu, B.: A large-scale, in-depth analysis of developers' personalities in the Apache ecosystem. *Inf. Software Technol.* **114**, 1–20 (2019). <https://doi.org/10.1016/j.infsof.2019.05.012>
36. Caneill, M., Zacchiroli, S.: Debsources: live and historical views on macro-level software evolution. In: International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, New York (2014). <https://doi.org/10.1145/2652524.2652528>. <http://sources.debian.net>
37. Caneill, M., German, D.M., Zacchiroli, S.: The debsources dataset: two decades of free and open source software. *Empir. Software Eng.* **22**, 1405–1437 (2017). <https://doi.org/10.1007/s10664-016-9461-5>
38. Chen, B., (Jack) Jiang, Z.M.: Characterizing logging practices in Java-based open source software projects – a replication study in Apache software foundation. *Empir. Software Eng.* **22**(1), 330–374 (2017). <https://doi.org/10.1007/s10664-016-9429-5>

39. Claes, M., Mens, T., Di Cosmo, R., Vouillon, J.: A historical analysis of Debian package incompatibilities. In: Working Conference on Mining Software Repositories (MSR), pp. 212–223 (2015). <https://doi.org/10.1109/MSR.2015.27>
40. Claes, M., Decan, A., Mens, T.: Intercomponent dependency issues in software ecosystems. In: Software Technology: 10 Years of Innovation in IEEE Computer, chap. 3, pp. 35–57. Wiley, Hoboken (2018). <https://doi.org/10.1002/9781119174240.ch3>
41. Cogo, F.R., Oliva, G.A., Hassan, A.E.: Deprecation of packages and releases in software ecosystems: a case study on npm. *Trans. Software Eng.* (2021). <https://doi.org/10.1109/TSE.2021.3055123>
42. Constantinou, E., Mens, T.: An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations Syst. Software Eng.* **13**(2), 101–115 (2017). <https://doi.org/10.1007/s11334-017-0303-4>
43. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in GitHub: transparency and collaboration in an open software repository. In: International Conference on Computer Supported Cooperative Work (CSCW), pp. 1277–1286. ACM, New York (2012). <https://doi.org/10.1145/2145204.2145396>
44. Decan, A., Mens, T.: What do package dependencies tell us about semantic versioning? *Trans. Software Eng.* **47**(6), 1226–1240 (2021). <https://doi.org/10.1109/TSE.2019.2918315>
45. Decan, A., Mens, T., Claes, M.: An empirical comparison of dependency issues in OSS packaging ecosystems. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, Piscataway (2017). <https://doi.org/10.1109/SANER.2017.7884604>
46. Decan, A., Mens, T., Constantinou, E.: On the evolution of technical lag in the npm package dependency network. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 404–414. IEEE, Piscataway (2018). <https://doi.org/10.1109/ICSME.2018.00050>
47. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories (MSR), pp. 181–191 (2018). <https://doi.org/10.1007/s10664-022-10154-1>
48. Decan, A., Mens, T., Grosjean, P.: An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Software Eng.* **24**(1), 381–416 (2019). <https://doi.org/10.1007/s10664-017-9589-y>
49. Decan, A., Mens, T., Zerouali, A., De Roover, C.: Back to the past – analysing backporting practices in package dependency networks. *Trans. Software Eng.* (2021). <https://doi.org/10.1109/TSE.2021.3112204>
50. Decan, A., Mens, T., Mazraei, P.R., Golzadeh, M.: On the use of GitHub Actions in software development repositories. In: International Conference on Software Maintenance and Evolution (ICSME). IEEE, Piscataway (2022). <https://doi.org/10.1109/ICSME55016.2022.00029>
51. de Lima Fontão, A., Pereira dos Santos, R., Dias-Neto, A.C.: Mobile software ecosystem (MSECO): a systematic mapping study. In: Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 653–658. IEEE, Piscataway (2015). <https://doi.org/10.1109/COMPSAC.2015.121>
52. de Lima Fontão, A., Ekwoge, O.M., dos Santos, R.P., Dias-Neto, A.C.: Facing up the primary emotions in mobile software ecosystems from developer experience. In: Workshop on Social, Human, and Economic Aspects of Software (WASHES), pp. 5–11. ACM, New York (2017). <https://doi.org/10.1145/3098322.3098325>
53. Dhungana, D., Groher, I., Schludermann, E., Biffl, S.: Guiding principles of natural ecosystems and their applicability to software ecosystems. In: Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry, chap. 3, pp. 43–58. Edward Elgar, Cheltenham (2013). <https://doi.org/10.4337/9781955628.00010>
54. Di Cosmo, R., Zacchiroli, S.: Software Heritage: why and how to preserve software source code. In: International Conference on Digital Preservation (iPRES) (2017)

55. Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K.: Dependency versioning in the wild. In: International Conference on Mining Software Repositories (MSR), pp. 349–359. IEEE, Piscataway (2019). <https://doi.org/10.1109/MSR.2019.00061>
56. Dueñas, S., Cosentino, V., Gonzalez-Barahona, J.M., del Castillo San Felix, A., Izquierdo-Cortazar, D., Cañas-Díaz, L., Pérez García-Plaza, A.: GrimoireLab: a toolset for software development analytics. PeerJ Comput. Sci. (2021). <https://doi.org/10.7717/peerj-cs.601>
57. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: International Conference on Software Engineering (ICSE), pp. 422–431. IEEE, Piscataway (2013). <https://doi.org/10.1109/ICSE.2013.6606588>
58. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: Ultra-large-scale software repository and source-code mining. Trans. Software Eng. Methodol. **25**(1) (2015). <https://doi.org/10.1145/2803171>
59. Estevo, P., Simmonds, J., Robbes, R., Fabry, J.: The Robot Operating System: package reuse and community dynamics. J. Syst. Software **151**, 226–242 (2019). <https://doi.org/10.1016/j.jss.2019.02.024>
60. Foundjem, A., Constantinou, E., Mens, T., Adams, B.: A mixed-methods analysis of micro-collaborative coding practices in OpenStack. Empir. Software Eng. **27**(5), 120 (2022). <https://doi.org/10.1007/s10664-022-10167-w>
61. Frakes, W., Kang, K.: Software reuse research: status and future. Trans. Software Eng. **31**(7), 529–536 (2005). <https://doi.org/10.1109/TSE.2005.85>
62. German, D.M., Adams, B., Hassan, A.E.: The evolution of the R software ecosystem. In: European Conference on Software Maintenance and Reengineering (CSMR), pp. 243–252 (2013). <https://doi.org/10.1109/CSMR.2013.33>
63. Goggins, S., Lumbard, K., Germonprez, M.: Open source community health: analytical metrics and their corresponding narratives. In: International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal), pp. 25–33 (2021). <https://doi.org/10.1109/SoHeal52568.2021.00010>
64. Goggins, S.P., Germonprez, M., Lumbard, K.: Making open source project health transparent. Computer **54**(8), 104–111 (2021). <https://doi.org/10.1109/MC.2021.3084015>
65. Gold, N.E., Krinke, J.: Ethics in the mining of software repositories. Empir. Software Eng. **27**(1), 17 (2022). <https://doi.org/10.1007/s10664-021-10057-7>
66. Golzadeh, M., Decan, A., Legay, D., Mens, T.: A ground-truth dataset and classification model for detecting bots in GitHub issue and PR comments. J. Syst. Software **175** (2021). <https://doi.org/10.1016/j.jss.2021.110911>
67. Golzadeh, M., Decan, A., Mens, T.: On the rise and fall of CI services in GitHub. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, Piscataway (2021). <https://doi.org/10.1109/SANER53432.2022.00084>
68. Gonzalez-Barahona, J.M., Robles, G.: On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. Empir. Software Eng. **17**(1), 75–89 (2012). <https://doi.org/10.1007/s10664-011-9181-9>
69. Gonzalez-Barahona, J.M., Robles, G., Michlmayr, M., Amor, J.J., German, D.M.: Macro-level software evolution: a case study of a large software compilation. Empir. Software Eng. **14**(3), 262–285 (2009). <https://doi.org/10.1007/s10664-008-9100-x>
70. Gousios, G., Spinellis, D.: GHTorrent: Github's data from a firehose. In: Working Conference of Mining Software Repositories (MSR), pp. 12–21 (2012). <https://doi.org/10.1109/MSR.2012.6224294>
71. Gousios, G., Spinellis, D.: Mining software engineering data from GitHub. In: International Conference on Software Engineering (ICSE), pp. 501–502 (2017). <https://doi.org/10.1109/ICSE-C.2017.164>
72. Gousios, G., Storey, M.A., Bacchelli, A.: Work practices and challenges in pull-based development: the contributor's perspective. In: International Conference on Software Engineering (ICSE), pp. 285–296. ACM, New York (2016). <https://doi.org/10.1145/2884781.2884826>

73. Grinter, R.E., Herbsleb, J.D., Perry, D.E.: The geography of coordination: dealing with distance in R&D work. In: International ACM SIGGROUP conference on Supporting group work (GROUP), pp. 306–315 (1999). <https://doi.org/10.1145/320297.320333>
74. Guzman, E., Azócar, D., Li, Y.: Sentiment analysis of commit comments in GitHub: an empirical study. In: International Conference on Mining Software Repositories (MSR), pp. 352–355. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597118>
75. Guzzi, A., Bacchelli, A., Lanza, M., Pinzger, M., van Deursen, A.: Communication in open source software development mailing lists. In: Working Conference on Mining Software Repositories (MSR), pp. 277–286. IEEE, Piscataway (2013)
76. Hanssen, G.K.: A longitudinal case study of an emerging software ecosystem: implications for practice and theory. *J. Syst. Software* **85**(7), 1455–1466 (2012). <https://doi.org/10.1016/j.jss.2011.04.020>
77. Hein, A., Schreieck, M., Riasanow, T., Setzke, D.S., Wiesche, M., Böhm, M., Krcmar, H.: Digital platform ecosystems. *Electron. Mark.* **30**(1), 87–98 (2020). <https://doi.org/10.1007/s12525-019-00377-4>
78. Herbsleb, J.D., Moitra, D.: Global software development. *IEEE Software* **18**(2), 16–20 (2001). <https://doi.org/10.1109/52.914732>
79. Howison, J., Crowston, K.: The perils and pitfalls of mining SourceForge. In: International Workshop on Mining Software Repositories (MSR), pp. 7–11 (2004). <https://doi.org/10.1049/ic:20040467>
80. Howison, J., Conklin, M., Crowston, K.: Flossmole: a collaborative repository for FLOSS research data and analyses. *IJITWE* **1**(3), 17–26 (2006). <https://doi.org/10.4018/jitwe.2006070102>
81. Hung, C.S., Dyer, R.: Boa views: easy modularization and sharing of MSR analyses. In: International Conference on Mining Software Repositories (MSR), pp. 147–157. ACM, New York (2020). <https://doi.org/10.1145/3379597.3387480>
82. Iansiti, M., Levien, R.: Strategy as ecology. *Harvard Bus. Rev.* **82**(3), 68–81 (2004)
83. Jansen, S., Finkelstein, A., Brinkkemper, S.: A sense of community: a research agenda for software ecosystems. In: International Conference on Software Engineering, pp. 187–190 (2009). <https://doi.org/10.1109/ICSE-COMPANION.2009.5070978>
84. Jansen, S., Brinkkemper, S., Cusumano, M.A.: Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry. Edward Elgar, Cheltenham (2013)
85. Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P.S., Zhang, L.: Why and how developers fork what from whom in GitHub. *Empir. Software Eng.* **22**(1), 547–578 (2017). <https://doi.org/10.1007/s10664-016-9436-6>
86. Jurado, F., Rodríguez Marín, P.: Sentiment analysis in monitoring software development processes: an exploratory case study on GitHub’s project issues. *J. Syst. Software* **104**, 82–89 (2015). <https://doi.org/10.1016/j.jss.2015.02.055>
87. Kabbedijk, J., Jansen, S.: Steering insight: An exploration of the Ruby software ecosystem. In: Software Business, pp. 44–55. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21544-5%5C_5
88. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining GitHub. In: Working Conference on Mining Software Repositories (MSR), MSR 2014, pp. 92–101. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597074>
89. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D., Damian, D.: An in-depth study of the promises and perils of mining GitHub. *Empir. Software Eng.* **21**(5), 2035–2071 (2016). <https://doi.org/10.1007/s10664-015-9393-5>
90. Katz, J.: Libraries.io open source repository and dependency metadata (2020). <https://doi.org/10.5281/zenodo.3626071>
91. Kawuma, S., Businge, J., Bainomugisha, E.: Can we find stable alternatives for unstable Eclipse interfaces? In: International Conference on Program Comprehension (ICPC), pp. 1–10 (2016). <https://doi.org/10.1109/ICPC.2016.7503716>

92. Kinsman, T., Wessel, M., Gerosa, M.A., Treude, C.: How do software developers use GitHub Actions to automate their workflows? In: International Conference on Mining Software Repositories (MSR), pp. 420–431. IEEE, Piscataway (2021). <https://doi.org/10.1109/MSR52588.2021.00054>
93. Koch, S.: Exploring the effects of SourceForge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empir. Software Eng.* **14**(4), 397–417 (2009). <https://doi.org/10.1007/s10664-008-9086-4>
94. Kolak, S., Afzal, A., Le Goues, C., Hilton, M., Timperley, C.S.: It takes a village to build a robot: an empirical study of the ROS ecosystem. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 430–440 (2020). <https://doi.org/10.1109/ICSME46990.2020.00048>
95. Kotovs, V.: Forty years of software reuse. *Sci. J. Riga Tech. Univ.* **38**(38), 153–160 (2009). <https://doi.org/10.2478/v10143-009-0013-y>
96. Kozaczynski, W., Booch, G.: Component-based software engineering. *IEEE Software* **15**(5), 34–36 (1998). <https://doi.org/10.1109/MS.1998.714621>
97. Krueger, C.W.: Software reuse. *ACM Comput. Surv.* **24**(2), 131–183 (1992). <https://doi.org/10.1145/130844.130856>
98. Lam, P., Dietrich, J., Pearce, D.J.: Putting the semantics into semantic versioning. In: International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), pp. 157–179. ACM, New York (2020). <https://doi.org/10.1145/3426428.3426922>
99. Lanovaz, M.J., Adams, B.: Comparing the communication tone and responses of users and developers in two R mailing lists: measuring positive and negative emails. *IEEE Software* **36**(5), 46–50 (2019). <https://doi.org/10.1109/MS.2019.2922949>
100. Lauinger, T., Chaabane, A., Wilson, C.B.: Thou shalt not depend on me. *Commun. ACM* **61**(6), 41–47 (2018). <https://doi.org/10.1145/3190562>
101. Lehman, M.M.: On understanding laws, evolution and conservation in the large program life cycle. *J. Syst. Software* **1**(3), 213–221 (1980). [https://doi.org/10.1016/0164-1212\(79\)90022-0](https://doi.org/10.1016/0164-1212(79)90022-0)
102. Lehman, M.M.: Programs, life cycles, and laws of software evolution. *Proc. IEEE* **68**(9), 1060–1076 (1980). <https://doi.org/10.1109/PROC.1980.11805>
103. Li, Z., Wang, Y., Lin, Z., Cheung, S.C., Lou, J.G.: Nufix: Escape from NuGet dependency maze. In: International Conference on Software Engineering (ICSE), pp. 1545–1557. ACM, New York (2022). <https://doi.org/10.1145/3510003.3510118>
104. Lin, B., Zagalsky, A., Storey, M.A., Serebrenik, A.: Why developers are slacking off: understanding how software teams use Slack. In: International Conference on Computer Supported Cooperative Work (CSCW), pp. 333–336. ACM, New York (2016). <https://doi.org/10.1145/2818052.2869117>
105. Lungu, M.: Towards reverse engineering software ecosystems. In: International Conference on Software Maintenance (ICSM), pp. 428–431. IEEE, Piscataway (2008). <https://doi.org/10.1109/ICSM.2008.4658096>
106. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., Mockus, A.: World of code: an infrastructure for mining the universe of open source VCS data. In: International Conference on Mining Software Repositories (MSR), pp. 143–154. IEEE, Piscataway (2019). <https://doi.org/10.1109/MSR.2019.00031>
107. Ma, Y., Dey, T., Bogart, C., Amreen, S., Valiev, M., Tutko, A., Kennard, D., Zaretzki, R., Mockus, A.: World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empir. Software Eng.* **26**(2) (2021). <https://doi.org/10.1007/s10664-020-09905-9>
108. Makari, I.S., Zerouali, A., De Roover, C.: Prevalence and evolution of license violations in npm and RubyGems dependency networks. In: International Conference on Software and Systems Reuse (ICSR), pp. 85–100. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-08129-3_6

109. Manes, S.S., Baysal, O.: How often and what StackOverflow posts do developers reference in their GitHub projects? In: International Conference on Mining Software Repositories (MSR), pp. 235–239 (2019). <https://doi.org/10.1109/MSR.2019.00047>
110. Manikas, K.: Revisiting software ecosystems research: a longitudinal literature study. *J. Syst. Software* **117**, 84–103 (2016). <https://doi.org/10.1016/j.jss.2016.02.003>
111. Manikas, K., Hansen, K.M.: Software ecosystems: a systematic literature review. *J. Syst. Software* **86**(5), 1294–1306 (2013). <https://doi.org/10.1016/j.jss.2012.12.026>
112. McIlroy, M.D.: Mass produced software components. In: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. Garmisch, Germany (1969)
113. Mens, T.: Evolving software ecosystems: a historical and ecological perspective. NATO Sci. Peace Sec. Ser. D Inform. Commun. Sec. **Volume 40: Dependable Software Systems Engineering**, 170–192 (2015). <https://doi.org/10.3233/978-1-61499-495-4-170>
114. Mens, T., Demeyer, S. (eds.): Software Evolution. Springer, Berlin (2008)
115. Mens, T., Fernández-Ramil, J., Degrandart, S.: The evolution of Eclipse. In: International Conference on Software Maintenance (ICSM). IEEE, Piscataway (2008). <https://doi.org/10.1109/ICSM.2008.4658087>
116. Mens, T., Serebrenik, A., Cleve, A.: Evolving Software Systems. Springer, Berlin (2014)
117. Messerschmitt, D.G., Szyperski, C.: Software ecosystem: understanding an indispensable technology and industry. MIT Press, Cambridge (2003)
118. Mitropoulos, D., Karakoidas, V., Louridas, P., Gousios, G., Spinellis, D.: The bug catalog of the Maven ecosystem. In: Working Conference on Mining Software Repositories (MSR), pp. 372–375. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597123>
119. Mockus, A., Fielding, R.T., Herbsleb, J.D.: Two case studies of open source software development: apache and mozilla. *Trans. Software Eng. Methodol.* **11**(3), 309–346 (2002). <https://doi.org/10.1145/567793.567795>
120. Moldon, L., Strohmaier, M., Wachs, J.: How gamification affects software developers: cautionary evidence from a natural experiment on GitHub. In: International Conference on Software Engineering (ICSE), pp. 549–561 (2021). <https://doi.org/10.1109/ICSE43902.2021.00058>
121. Montgomery, L., Lüders, C., Maalej, P.D.W.: The public Jira dataset (2022). <https://doi.org/10.5281/zenodo.5901804>
122. Montgomery, L., Lüders, C., Maalej, W.: An alternative issue tracking dataset of public Jira repositories. In: International Conference on Mining Software Repositories (MSR), pp. 73–77. ACM, New York (2022). <https://doi.org/10.1145/3524842.3528486>
123. Moore, J.: Predators and prey: a new ecology of competition. *Harvard Bus. Rev.* **71**(3), 75–83 (1993)
124. Nagy, C., Cleve, A.: Mining stack overflow for discovering error patterns in SQL queries. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 516–520. IEEE, Piscataway (2015). <https://doi.org/10.1109/ICSM.2015.7332505>
125. Nasehi, S.M., Sillito, J., Maurer, F., Burns, C.: What makes a good code example? A study of programming Q&A in StackOverflow. In: International Conference on Software Maintenance (ICSM), pp. 25–34. IEEE, Piscataway (2012). <https://doi.org/10.1109/ICSM.2012.6405249>
126. Naur, P., Randell, B.: Software Engineering: Report of a Conference Sponsored by the NATO Science Committee. NATO, Garmisch (1969)
127. Novielli, N., Calefato, F., Lanubile, F.: The challenges of sentiment detection in the social programmer ecosystem. In: International Workshop on Social Software Engineering (SSE), pp. 33–40. ACM, New York (2015). <https://doi.org/10.1145/2804381.2804387>
128. Nugroho, Y.S., Islam, S., Nakasai, K., Rehman, I., Hata, H., Kula, R.G., Nagappan, M., Matsumoto, K.: How are project-specific forums utilized? A study of participation, content, and sentiment in the Eclipse ecosystem. *Empir. Software Eng.* **26**(6), 132 (2021). <https://doi.org/10.1007/s10664-021-10032-2>
129. Nyman, L., Mikkonen, T.: To fork or not to fork: Fork motivations in SourceForge projects. *Int. J. Open Source Software Proces.* **3**(3) (2011). <https://doi.org/10.4018/jossp.2011070101>

130. Ochoa, L., Degueule, T., Falleri, J.R., Vinju, J.: Breaking bad? Semantic versioning and impact of breaking changes in Maven Central. *Empir. Software Eng.* **27**(3), 61 (2022). <https://doi.org/10.1007/s10664-021-10052-y>
131. Opdebeeck, R., Zerouali, A., De Roover, C.: Smelly variables in Ansible infrastructure code: detection, prevalence, and lifetime. In: International Conference on Mining Software Repositories (MSR). ACM, New York (2022). <https://doi.org/10.1145/3524842.3527964>
132. Opdebeeck, R., Zerouali, A., De Roover, C.: Control and data flow in security smell detection for infrastructure as code: Is it worth the effort? In: International Conference on Mining Software Repositories (MSR). ACM, New York (2023)
133. Ortú, M., Destefanis, G., Adams, B., Murgia, A., Marchesi, M., Tonelli, R.: The JIRA repository dataset: understanding social aspects of software development. In: International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE). ACM, New York (2015). <https://doi.org/10.1145/2810146.2810147>
134. Ortú, M., Hall, T., Marchesi, M., Tonelli, R., Bowes, D., Destefanis, G.: Mining communication patterns in software development: a GitHub analysis. In: International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE), pp. 70–79. ACM, New York (2018). <https://doi.org/10.1145/3273934.3273943>
135. Padhye, R., Mani, S., Sinha, V.S.: A study of external community contribution to open-source projects on GitHub. In: Working Conference on Mining Software Repositories (MSR), pp. 332–335. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597113>
136. Pichler, M., Dieber, B., Pinzger, M.: Can i depend on you? Mapping the dependency and quality landscape of ROS packages. In: International Conference on Robotic Computing (IRC), pp. 78–85. IEEE, Piscataway (2019). <https://doi.org/10.1109/IRC.2019.00020>
137. Pietri, A., Spinellis, D., Zacchiroli, S.: The software heritage graph dataset: large-scale analysis of public software development history. In: International Conference on Mining Software Repositories (MSR). IEEE, Piscataway (2020). <https://doi.org/10.1145/3379597.3387510>
138. Plakidas, K., Schall, D., Zdun, U.: Evolution of the R software ecosystem: metrics, relationships, and their impact on qualities. *J. Syst. Software* **132**, 119–146 (2017). <https://doi.org/10.1016/j.jss.2017.06.095>
139. Pletea, D., Vasilescu, B., Serebrenik, A.: Security and emotion: sentiment analysis of security discussions on GitHub. In: Working Conference on Mining Software Repositories (MSR), pp. 348–351. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597117>
140. Raemaekers, S., van Deursen, A., Visser, J.: The Maven repository dataset of metrics, changes, and dependencies. In: Working Conference on Mining Software Repositories (MSR), pp. 221–224 (2013). <https://doi.org/10.1109/MSR.2013.6624031>
141. Raemaekers, S., Van Deursen, A., Visser, J.: Semantic versioning versus breaking changes: a study of the Maven repository. In: International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 215–224. IEEE, Piscataway (2014). <https://doi.org/10.1109/SCAM.2014.30>
142. Rahman, M.M., Roy, C.K.: An insight into the pull requests of GitHub. In: Working Conference on Mining Software Repositories (MSR), pp. 364–367. ACM, New York (2014). <https://doi.org/10.1145/2597073.2597121>
143. Rastogi, A., Nagappan, N., Gousios, G., van der Hoek, A.: Relationship between geographical location and evaluation of developer contributions in GitHub. In: International Symposium on Empirical Software Engineering and Measurement (ESEM). ACM, New York (2018). <https://doi.org/10.1145/3239235.3240504>
144. Raymond, E.: The cathedral and the bazaar. *Knowl. Technol. Policy* **12**(3), 23–49 (1999). <https://doi.org/10.1007/s12130-999-1026-0>
145. Raymond, E.S.: The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol (1999)
146. Rigby, P.C., Hassan, A.E.: What can OSS mailing lists tell us? A preliminary psychometric text analysis of the Apache developer mailing list. In: International Workshop on Mining Software Repositories (MSR), pp. 23–23 (2007). <https://doi.org/10.1109/MSR.2007.35>

147. Robles, G., Gonzalez-Barahona, J.M.: Geographic location of developers at SourceForge. In: International Workshop on Mining Software Repositories (MSR), pp. 144–150. ACM, New York (2006). <https://doi.org/10.1145/1137983.1138017>
148. Sayyad Shirabad, J., Menzies, T.: The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa (2005). URL <http://promise.site.uottawa.ca/SERepository>
149. Schueler, W., Wachs, J., Servedio, V.D.P., Thurner, S., Loreto, V.: Evolving collaboration, dependencies, and use in the rust open source software ecosystem. *Sci. Data* **9**(1), 703 (2022). <https://doi.org/10.1038/s41597-022-01819-z>
150. Schwaber, K.: SCRUM development process. In: *Business Object Design and Implementation*, pp. 117–134. Springer, Berlin (1997)
151. Seppänen, M., Hyrynsalmi, S., Manikas, K., Suominen, A.: Yet another ecosystem literature review: 10+1 research communities. In: European Technology and Engineering Management Summit (E-TEMS), pp. 1–8. IEEE, Piscataway (2017). <https://doi.org/10.1109/E-TEMS.2017.8244229>
152. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: Working Conference on Mining Software Repositories (MSR), pp. 189–200 (2016). <https://doi.org/10.1145/2901739.2901761>
153. Singh, N., Singh, P.: How do code refactoring activities impact software developers' sentiments? An empirical investigation into GitHub commits. In: Asia-Pacific Software Engineering Conference (APSEC), pp. 648–653. IEEE, Piscataway (2017). <https://doi.org/10.1109/APSEC.2017.79>
154. Soto-Valero, C., Benelallam, A., Harrand, N., Barais, O., Baudry, B.: The emergence of software diversity in Maven Central. In: International Conference on Mining Software Repositories (MSR), pp. 333–343 (2019). <https://doi.org/10.1109/MSR.2019.00059>
155. Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B.: A comprehensive study of bloated dependencies in the Maven ecosystem. *Empir. Software Eng.* **26**(3), 1–44 (2021). <https://doi.org/10.1007/s10664-020-09914-8>
156. Steglich, C., Marczak, S., Guerra, L.P., Mosmann, L.H., Perin, M., Figueira Filho, F., de Souza, C.: Revisiting the mobile software ecosystems literature. In: International Workshop on Software Engineering for Systems-of-Systems (SESoS) and Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems (WDES), pp. 50–57 (2019). <https://doi.org/10.1109/SESoS/WDES.2019.00015>
157. Storey, M.A., Zagalsky, A., Filho, F.F., Singer, L., German, D.M.: How social and communication channels shape and challenge a participatory culture in software development. *Trans. Software Eng.* **43**(2), 185–204 (2017). <https://doi.org/10.1109/TSE.2016.2584053>
158. Stringer, J., Tahir, A., Blincoe, K., Dietrich, J.: Technical lag of dependencies in major package managers. In: Asia-Pacific Software Engineering Conference (APSEC), pp. 228–237 (2020). <https://doi.org/10.1109/APSEC51365.2020.00031>
159. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 1st ed. Addison-Wesley, Boston (1997)
160. Takhteyev, Y., Hilts, A.: Investigating the geography of open source software through GitHub. <https://flosshub.org/sites/flosshub.org/files/Takhteyev-Hilts-2010.pdf> (2010)
161. Tan, J., Feitosa, D., Avgeriou, P., Lungu, M.: Evolution of technical debt remediation in Python: a case study on the Apache software ecosystem. *J. Software Evol. Proces.* **33**(4) (2020). <https://doi.org/10.1002/smri.2319>
162. Teixeira, J., Hyrynsalmi, S.: How do software ecosystems co-evolve? A view from OpenStack and beyond. In: International Conference of Software Business (ICSOB), pp. 115–130. Springer, Berlin (2017). <https://doi.org/10.1007/978-3-319-69191-6>
163. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: The Qualitas Corpus: a curated collection of Java code for empirical studies. In: Asia Pacific Software Engineering Conference (APSEC), pp. 336–345 (2010). <https://doi.org/10.1109/APSEC.2010.46>

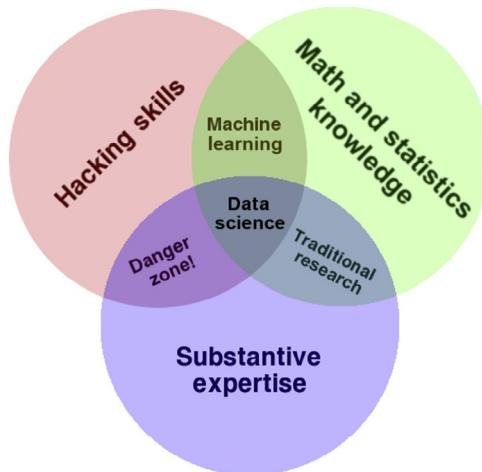
164. Tiwari, N.M., Upadhyaya, G., Rajan, H.: Candoia: a platform and ecosystem for mining software repositories tools. In: International Conference on Software Engineering (ICSE), pp. 759–764 (2016). <https://doi.org/10.1145/2889160.2892662>
165. Tiwari, N.M., Upadhyaya, G., Nguyen, H.A., Rajan, H.: Candoia: A platform for building and sharing mining software repositories tools as apps. In: International Conference on Mining Software Repositories (MSR), pp. 53–63 (2017). <https://doi.org/10.1109/MSR.2017.56>
166. Tourani, P., Adams, B.: The impact of human discussions on just-in-time quality assurance: an empirical study on OpenStack and Eclipse. In: International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 189–200. IEEE, Piscataway (2016). <https://doi.org/10.1109/SANER.2016.113>
167. Tourani, P., Jiang, Y., Adams, B.: Monitoring sentiment in open source mailing lists: exploratory study on the Apache ecosystem. In: International Conference on Computer Science and Software Engineering (CASCON), pp. 34–44. IBM, Armonk/ACM, New York (2014)
168. Tsay, J., Dabbish, L., Herbsleb, J.: Influence of social and technical factors for evaluating contribution in GitHub. In: International Conference on Software Engineering (ICSE), pp. 356–366. ACM, New York (2014). <https://doi.org/10.1145/2568225.2568315>
169. Uddin, G., Khomh, F.: Automatic mining of opinions expressed about APIs in Stack Overflow. *Trans. Software Eng.*, 1–1 (2019). <https://doi.org/10.1109/TSE.2019.2900245>
170. Um, S., Zhang, B., Wattal, S., Yoo, Y.: Software components and product variety in a platform ecosystem: a dynamic network analysis of WordPress. *Inform. Syst. Res.* (2022). <https://doi.org/10.1287/isre.2022.1172>
171. Valiev, M., Vasilescu, B., Herbsleb, J.: Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 644–655. ACM, New York (2018). <https://doi.org/10.1145/3236024.3236062>
172. Vasilescu, B., Serebrenik, A., Goeminne, M., Mens, T.: On the variation and specialisation of workload: a case study of the Gnome ecosystem community. *Empir. Software Eng.* **19**(4), 955–1008 (2014). <https://doi.org/10.1007/s10664-013-9244-1>
173. Vasilescu, B., Posnett, D., Ray, B., van den Brand, M.G., Serebrenik, A., Devanbu, P., Filkov, V.: Gender and tenure diversity in GitHub teams. In: Conference on Human Factors in Computing Systems (CHI), pp. 3789–3798. ACM, New York (2015). <https://doi.org/10.1145/2702123.2702549>
174. Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in GitHub. In: Joint meeting on Foundations of Software Engineering (ESEC/FSE), pp. 805–816 (2015). <https://doi.org/10.1145/2786805.2786850>
175. Velázquez-Rodríguez, C., Constantinou, E., De Roover, C.: Uncovering library features from API usage on Stack Overflow. In: International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 207–217. IEEE, Piscataway (2022). <https://doi.org/10.1109/SANER53432.2022.00035>
176. Velázquez-Rodríguez, C., Di Nucci, D., De Roover, C.: A text classification approach to API type resolution for incomplete code snippets. *Sci. Comput. Programm.* **227**, 102941 (2023). <https://doi.org/10.1016/j.scico.2023.102941>
177. Wachs, J., Nitecki, M., Schueler, W., Polleres, A.: The geography of open source software: evidence from GitHub. *Technol. Forecast. Soc. Change* **176** (2021). <https://doi.org/10.1016/j.techfore.2022.121478>
178. Wang, Z., Wang, Y., Redmiles, D.: From specialized mechanics to project butlers: the usage of bots in OSS development. *IEEE Software* (2022). <https://doi.org/10.1109/MS.2022.3180297>
179. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley (1999). ISBN 0201694387, 9780201694383
180. Werder, K., Brinkkemper, S.: MEME: toward a method for emotions extraction from GitHub. In: International Workshop on Emotion Awareness in Software Engineering (SEmotion), pp. 20–24. ACM, New York (2018). <https://doi.org/10.1145/3194932.3194941>

181. Wessel, M., Serebrenik, A., Wiese, I., Steinmacher, I., Gerosa, M.A.: Quality gatekeepers: investigating the effects of code review bots on pull request activities. *Empir. Software Eng.* **27**(5), 108 (2022). <https://doi.org/10.1007/s10664-022-10130-9>
182. Wessel, M., Vargovich, J., Gerosa, M.A., Treude, C.: Github actions: the impact on the pull request process. Preprint. arXiv:2206.14118 (2022)
183. Wiese, I.S., Da Silva, J.T., Steinmacher, I., Treude, C., Gerosa, M.A.: Who is who in the mailing list? Comparing six disambiguation heuristics to identify multiple addresses of a participant. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 345–355. IEEE, Piscataway (2016). <https://doi.org/10.1109/ICSME.2016.13>
184. Willis, A.: The ecosystem: an evolving concept viewed historically. *Funct. Ecol.* **11**, 268–271 (1997)
185. Yang, B., Wei, X., Liu, C.: Sentiments analysis in GitHub repositories: an empirical study. In: Asia-Pacific Software Engineering Conference Workshops (APSEC Workshops), pp. 84–89. IEEE, Piscataway (2017). <https://doi.org/10.1109/APSECW.2017.13>
186. Yau, S., Collofello, J., MacGregor, T.: Ripple effect analysis of software maintenance. In: International Computer Software and Applications Conference (COMPSAC), pp. 60–65. IEEE, Piscataway (1978). <https://doi.org/10.1109/CMPSCA.1978.810308>
187. Yu, Y., Wang, H., Filkov, V., Devanbu, P., Vasilescu, B.: Wait for it: determinants of pull request evaluation latency on GitHub. In: Working Conference on Mining Software Repositories (MSR), pp. 367–371 (2015). <https://doi.org/10.1109/MSR.2015.42>
188. Zagalsky, A., German, D.M., Storey, M.A., Teshima, C.G., Poo-Caamaño, G.: How the R community creates and curates knowledge: an extended study of Stack Overflow and mailing lists. *Empir. Software Eng.* **23**(2), 953–986 (2018). <https://doi.org/10.1007/s10664-017-9536-y>
189. Zerouali, A., Constantinou, E., Mens, T., Robles, G., González-Barahona, J.: An empirical analysis of technical lag in npm package dependencies. In: International Conference on Software Reuse (ICSR). Lecture Notes in Computer Science, vol. 10826, pp. 95–110. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-90421-4_6
190. Zerouali, A., Mens, T., Decan, A., De Roover, C.: On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empir. Software Eng.* **27**(5), 1–45 (2022). <https://doi.org/10.1007/s10664-022-10154-1>
191. Zerouali, A., Mens, T., Gonzalez-Barahona, J., Decan, A., Constantinou, E., Robles, G.: A formal framework for measuring technical lag in component repositories—and its application to npm. *J. Software: Evol. Process* **31**(8) (2019). <https://doi.org/10.1002/smrv.2157>
192. Zerouali, A., Velázquez-Rodríguez, C., De Roover, C.: Identifying versions of libraries used in Stack Overflow code snippets. In: International Conference on Mining Software Repositories (MSR), pp. 341–345. IEEE, Piscataway (2021). <https://doi.org/10.1109/MSR52588.2021.00046>
193. Zhang, Y., Liu, H., Tan, X., Zhou, M., Jin, Z., Zhu, J.: Turnover of companies in openstack: prevalence and rationale. *Trans. Software Eng. Methodol.* **31**(4) (2022). <https://doi.org/10.1145/3510849>
194. Zhou, S., Vasilescu, B., Kästner, C.: How has forking changed in the last 20 years? A study of hard forks on GitHub. In: International Conference on Software Engineering (ICSE), pp. 445–456. ACM, New York (2020). <https://doi.org/10.1145/3377811.3380412>

RULE #2: KNOW THE DOMAIN

4

What are the skills needed by a successful data scientist? The previous chapter made the case that such scientists need to be skilled facilitators for meetings of business users. But is anything more needed?



According to the famous Conway diagram (shown right), a good data scientist is part hacker, part mathematician, and part domain expert. In his comments on the diagram,¹ Conway warns that all three are essential.

Note the “danger zone” where people know enough to be dangerous but not enough to be cautious. This is the region of “lies, damned lies, and statistics” where novices report models without any real understanding of the limitations of those models or their implication for the business.

Note also the important of substantive domain expertise. Novice data scientists do not take the time to learn the context around the data they are studying. Hence, these novices can either miss important results or report spurious results.

¹ <http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram>.

4.1 CAUTIONARY TALE #1: “DISCOVERING” RANDOM NOISE

As an example of the discovery of spurious patterns, consider a data miner trying to find structures within the tick marks of Figure 4.1. Each of the squares in that figure was generated by some process that

- Reflects over the last N characters (and a random process has $N = 0$);
 - Then writes either a vertical or horizontal dash.

Before reading the next paragraph, the reader is invited to decide which square in Figure 4.1 comes from a random process.

Pause.

Worked it out? In Figure 4.1, square (c) has the longer runs of identical characters; in other words, it seems the least random. It may surprise the reader to learn that square (c) was actually generated by

(a)

(b)

(c)

FIGURE 4.1

Random, or structured, data? One of these squares was generated with each tick mark independent of the one before it (in left-to-right top-to-bottom order). From <http://norvig.com/experiment-design.html>.

the random process. To understand that, please consider the mathematics of random numbers. The odds of writing either character is 0.5 so the odds of writing, say, six in a row is $0.5^6 = (1/64)$. In the 312 characters of each square, we should therefore expect at least four such randomly generated structures; that is, the squares with the fewest number of long runs was generated by the least random process.

4.2 CAUTIONARY TALE #2: JUMPING AT SHADOWS

One way to avoid spurious conclusions based on random noise (such as the structures in Figure 4.1c) is to use domain knowledge. This is an important point as there are many cautionary tales about the mistakes made by data scientists who lacked that domain knowledge.

For example, in the 1990s, it was standard practice at the NASA's Software Engineering Laboratory [117] to share data with outside research groups. The only caveat was that research teams who wanted that data had to come to the lab and spend one week on domain familiarization. This policy was put in place after too many outside researchers kept making the same mistake.

- A repeated conclusion from those outsiders was that a certain class of NASA subsystems was inherently most bug prone.
- The problem with that conclusion was that, while certainly true, it missed an important factor. It turns out that that particular subsystem was the one deemed least critical by NASA. Hence, it was standard policy to let newcomers work on that subsystem in order to learn the domain.
- Because such beginners make more mistakes, it is hardly surprising that this particular subsystem saw the most errors.

Note that, for this subsystem, an ounce of domain knowledge would have been worth a ton of data mining algorithms.

4.3 CAUTIONARY TALE #3: IT PAYS TO ASK

Another cautionary tale (about the value of domain knowledge) comes from when one of the authors worked at Microsoft. During his research on the quality implications of distributed development [232], Kocagüneli, had to determine which code files were created by a distributed or centralized development process. This, in turn, meant mapping files to their authors, and then situating some author in a particular building in a particular city and country.

After six weeks of work (which was mostly spent running and debugging some very long SQL queries), it appeared that a very small number of people produced most of the core changes to certain Microsoft products.

- Initially, this result seemed very plausible as it had some precedence in the field of software engineering. In the 1970s, Fred Brooks [59] proposed organizing programming like a surgical team around one highly skilled chief programmer and other, lesser-trained support personnel.
- This “chief programmer” model is suitable when the success of a project is determined by a relatively small percent of the workforce.
- And, if this was the reality of work at Microsoft, it would mean that product quality would be most assured by focusing more on this small group.

However, the conclusions from that first six weeks of work were completely wrong. Microsoft is a highly optimized organization that takes full advantage of the benefits of auto-generated code. That generation occurs when software binaries are being built and, at Microsoft, that build process is controlled by a small number of skilled engineers. As a result, most of the files appeared to be “owned” by these build engineers even though these files are built from code provided by a very large number of programmers working across the Microsoft organization.

Now, to repeat the lesson from the last chapter, the erroneous nature of the initial conclusions was first revealed after Kocagüneli talked to his business users. That is, by showing the wrong conclusions, Kocagüneli’s users were prompted to explain more of the details of the build process at Microsoft. As a result, the study could be changed and some interesting and insightful final results were generated [232].

4.4 SUMMARY

Be good at the math, be good at the programming, but also take the time to carefully understand the domain.

Measuring individual productivity

T. Fritz

University of Zurich, Zurich, CHE

CHAPTER OUTLINE

No Single and Simple Best Metric for Success/Productivity	68
Measure the Process, Not Just the Outcome	68
Allow for Measures to Evolve	69
Goodhart's Law and the Effect of Measuring	69
How to Measure Individual Productivity?	70
References	71

In the last century, one company was said to have determined the price of a software product by the estimated number of lines of code written. In turn, the company also paid their software developers based on the number of lines of code produced per day. What happened next is that the company’s developers started “gaming the system”—they suddenly wrote a lot more code, while the functionality captured in the code decreased. As one might imagine, adding more lines of code to a program without changing the behavior is easy. So incentivizing employees on a single outcome metric might just foster this behavior.

Overall, this example shows that it is not easy to find a good measure for a developer’s success or productivity. In some domains, such as car manufacturing, specific measures on the quantity of the outcome, such as the number of cars produced in a day, have worked well to incentivize employees and to measure their success. In the software development domain, however, where the outcome and the overall process of developing software is less clearly defined and less tangible, such outcome measures are difficult to define. In particular, trying to reduce the complex activity of developing software into a single measure of one particular outcome of the development process is probably impossible.

NO SINGLE AND SIMPLE BEST METRIC FOR SUCCESS/PRODUCTIVITY

In a study we conducted, we asked professional developers how they measure and assess their personal productivity [1]. The results show that software developers generally want a combination of measures to assess productivity, and these combinations varied significantly across developers. Even for the one metric that developers rated highest overall for assessing productivity—the number of work items (tasks, bugs) closed—developers stated that it heavily depends on the complexity and size of the task, so that further information is needed to interpret this metric.

These findings further indicate that there is no single and simple best measure for assessing a developer's productivity. Given the variety of artifacts produced during software development, such as code artifacts, test cases, bug reports, and documentation, it is also not surprising that just focusing on the code would does not adequately reflect the progress a developer makes in his/her work. As one example of the variety of artifacts generated by developers every day, are the around 10,000 Java source files, 26,000 Bugzilla bug reports and the 45,000 news-group entries that were created over one development cycle of Eclipse—an open source IDE [2]. While the code is being compiled and executed in the end, the other artifacts are just as important to the process to make sure the software product is developed the right way and works.

MEASURE THE PROCESS, NOT JUST THE OUTCOME

The variety in outcomes or artifacts generated in the process is just one important aspect for measuring a developer's work and productivity. A majority of the participants in our study also mentioned that they are particularly productive when they get into the “flow” without many context switches. So rather than just focusing on a measure of the outcomes of the development activity, such as the code artifacts, the process of developing the software is important as well. Software development is a complex process that is comprised of various activities and frequent interruptions and interactions with multiple stakeholders involved in the development process, such as fellow developers, requirements engineers, or even customers [3–5].

Measuring aspects of the process of developing software, such as flow or context switches, is difficult since their cost and impact on productivity vary and are difficult to determine. For example, take a developer who is testing a system and has to wait for a build or for the application to start up. Switching context in this situation to perform a shorter task and filling the time in between might actually increase his/her productivity. On the other hand, take a developer who is programming and in the “flow.” When the developer is interrupted by another developer and asked about

last weekend's football scores, the forced context switch is expensive, decreases productivity, and can even result in more errors being made overall, as studies have shown [6]. So overall, while aspects of the process are important for measuring productivity, they are difficult to quantify and measure. Recently emerging biometric sensing technologies might provide new means to measure such aspects of individual productivity better, especially due to their pervasiveness and their decreasing invasiveness.

ALLOW FOR MEASURES TO EVOLVE

When it comes to a person's fitness and health, wearable fitness tracking devices, such as the Fitbit [7] or the Withings Pulse Band [8] have recently gained widespread adoption. Most of these devices employ a simple step count measure that has also been shown to be very successful in providing users valuable insights on their activity level and in promoting and improving health and fitness (eg, [9,10]). In an interview-based study with 30 participants who had used and adopted wearable activity tracking devices "in the wild" for between 3 and 54 months, we found that the step count provided long-term motivation for people to become and stay active. Yet, while for several participants, the devices helped foster engagement in fitness, the device, and in particular the step count measure, did not support their increasingly sophisticated fitness goals. As one example, the step count helped some participants to walk more and start running, but when they started adding new activities to further increase fitness, such as weight lifting or yoga, the devices failed to capture these [11].

Similarly, the activities and responsibilities of an individual developer evolve over time and thus the measures capturing his/her productivity have to evolve. For example, someone might start out as a developer in a team, and later on be promoted to manage teams. While in the first role, the number of bug fixes completed might be a reasonable proxy for their productivity, but as a manager, they will focus mostly on their team being productive and have little to no time to work on bug fixes themselves.

GOODHART'S LAW AND THE EFFECT OF MEASURING

Goodhart's law states that "When a measure becomes a target, it ceases to be a good measure." This effect happened in the first example presented in this chapter. As soon as the lines of code metric was used to measure a developer's productivity and affected the developer's salary, it ceased to be a good indicator, with developers gaming the system to benefit. Coming back to the fitness-tracking domain, we found from our interviews that the numerical feedback provided by the devices effected users' behavior in ways other than the intended fitness improvement. In many cases,

the accountability and getting credit for activities became important, and even more important than the original goal, so that users adjusted their sports activities for better accountability, and became unhappy and annoyed when they forgot their devices despite being very active, or were merely “fishing for numbers” or gaming the system. One user, for example, stopped going backward on an elliptical machine since the device did not pick up on it.

The effect depends on how and what a measure is being used for, eg, is it just used for personal retrospection or is it used for adjusting someone’s salary? In any case, one needs to assess the influence a certain measure might have on a developer’s behavior and the risks it bears.

HOW TO MEASURE INDIVIDUAL PRODUCTIVITY?

Measuring productivity of individual developers is challenging given the complex and multifaceted nature of developing software. There is not a single and simple best measure that works for everyone. Rather, you will have to tailor the measurement to the specific situation and context you are looking at and the specific goal you have in mind with your measurement. For example, if you are conducting a research study on the productivity gain of a code navigation support tool, a measure such as the edit ratio [12], ie, the ratio between edit and select events, might be a good indicator. However, this neglects other aspects, such as interruptions that occur or the resumption lag, which might be important measures when looking at a more general developer productivity definition. Similarly, due to the collaborative nature of software development, it is not necessarily possible to isolate an individual’s productivity from the team’s productivity. Take, for example, the idea of reducing interruptions for an individual developer. While reducing interruptions for the single developer might very well increase her/his productivity, it could block other team members from working and decrease their productivity, as well as the team’s overall.

One also has to assess who will or should have access to the measurement data and how might this affect the developer’s behavior and thus the productivity measure in the end. If the developer knows that her/his boss will have access to certain parts of the data, it is likely that the developer will make sure that the data that the boss has access to fits the purpose rather than the reality; whereas, if the data is just for the developer or, for example, independent research that anonymizes data completely, it is more likely that the data will reflect reality. Finally, a very important point is the privacy of the collected data. Are you actually able to collect the data needed from the developers, are you allowed to analyze it and did you make sure that only the intended people have access to it? Since any productivity measure will most likely require fairly sensitive information that could be used for or against someone, you need to make sure to pay attention to privacy concerns, treat them carefully, and be transparent about who will have access to the data.

REFERENCES

- [1] Meyer AN, Fritz T, Murphy GC, Zimmermann T. Software developers? Perceptions of productivity. In: Proc. of the ACM SIGSOFT 22nd international symposium on the foundations of software engineering 2014 (FSE'14); 2014.
- [2] http://www.eclipse.org/eclipse/development/eclipse_3_0_stats.html.
- [3] Müller S, Fritz T. Stakeholders' information needs for artifacts and their dependencies in a real world context. In: Proc. of the IEEE international conference on software maintenance 2013 (ICSM'13); 2013.
- [4] Perry D, Staudenmayer N, Votta L. People, organizations, and process improvement. *IEEE Softw* 1994;11(4):36–45.
- [5] Singer J, Lethbridge T, Vinson N, Anquetil N. An examination of software engineering work practices. In: CASCON first decade high impact papers (CASCON '10); 2010.
- [6] Bailey BP, Konstan JA. On the need for attention-aware systems: measuring effects of interruption on task performance, error rate, and affective state. *Comput Hum Behav* 2006;22(4):685–708.
- [7] <http://www.fitbit.com/>.
- [8] <http://www2.withings.com/us/en/products/pulse>.
- [9] Bravata D, Smith-Spangler C, Sundaram V, Gienger A, Lin N, Lewis R, et al. Using pedometers to increase physical activity and improve health. *JAMA* 2007;298:2296–304.
- [10] Consolvo S, McDonald D, Toscos T, Chen M, Froehlich J, Harrison B, et al. Activity sensing in the wild: a field trial of UbiFit garden. In: Proc. CHI'08; 2008.
- [11] Fritz T, Huang EM, Murphy GC, Zimmermann T. Persuasive technology in the real world: a study of long-term use of activity sensing devices for fitness. In: Proc. of the ACM conference on human factors in computing systems 2014 (CHI'14); 2014.
- [12] Kersten M, Murphy GC. Using task context to improve programmer productivity. In: Proc. of the ACM SIGSOFT 14th international symposium on the foundations of software engineering 2006 (FSE'06); 2006.

Once is not enough: Why we need replication

N. Juristo

Universidad Politécnica de Madrid, Spain

CHAPTER OUTLINE

Motivating Example and Tips	299
Exploring the Unknown	300
Types of Empirical Results	301
Do's and Don't's	301
Further Reading	302

MOTIVATING EXAMPLE AND TIPS

Imagine you land on an uncharted planet where no human has been before. For one reason or another, you are unable to leave the spaceship to explore the unknown, and you can only gather information about the planet through the spaceship windows. You look out and see a strange being: an alien of a shape, size, and color that you could have never imagined. You are extremely surprised, and you carry on looking at it in order to unlock the mystery and form an idea of what sort of thing you have in your sights. Are you sure that what your eyes are seeing (and your brain is perceiving) through the window really is a true likeness of the outside world? What if the food you had to eat last night had gone off? What if you are seeing things, and there really is nothing out there? What would you do to make sure?

- **Tip:** *Empirical studies results should not be blindly trusted. Empirical data can also lie. How to be sure that what you observe exists?*

For ruling out the possibility that you are seeing things, you ask one of your traveling companions to look out of the window as well. You will not tell her anything about what you have seen. You lead her to the window and ask her what she can see, taking care not to manipulate her in any way. She looks out of the window next to yours and cannot see anything at all. Is that so? You go round to the window where she is stationed and find that you cannot see the alien from there either, while she takes her place at your window. Aha! There is a light of some kind shining into this window,

and it makes the alien invisible from here. Your traveling companion confirms that she, too, can see the alien from your window. It does exist then; it's out there!

- **Tip:** *Site and researchers are variables that might induce different results of the same identical empirical study. Different researchers being able to reproduce your results in their labs increase confidence in your empirical results. Is that all we need?*

“There is a red living being out there,” she says. Are you two sure enough that the alien is really red? Could it be that the glass in the window is so thick that it is deforming the alien’s profile? What if the glass is not perfectly transparent and is altering his color? “Al is watching us!” your colleague shouts, astonished (by this time you have given the creature a pet name). “Watching us?” you ask with surprise, “I can’t see any eyes.” You and she each take turns looking out of the window, and you discuss and compare what you see. What she referred to as an eye looks to you like a blotch on its skin (if you can call the alien’s outer covering skin...). She decides to climb up to the observatory on top of the spaceship to study Al using the onboard telescope. Great! Thanks to this new instrument, she can distinguish details that were unappreciable by just looking through the windows. The information that she proffers from the telescope is crucial in making out the eye-blotch controversy, as well as some particulars that were unclear from the window.

- **Tip:** *Instruments matter. Even the best-designed empirical study interferes with the phenomenon at hand. It is of utmost importance that the same phenomenon is studied with different instruments in order to guarantee that the observation is independent of the instrument.*

EXPLORING THE UNKNOWN

This imaginary scenario has a lot in common with empirical research work. We empirical software engineering researchers are every inch the spaceship’s crew on our way to an unexplored planet. We are on a voyage into the unknown: software development. Unfortunately, we cannot *penetrate* the unknown. We cannot travel to the place where the strings of software development are pulled; the place where the development variables causing the external behavior that we observe in software projects are cooked up. The software *backstage*, where hidden variables are ruling development behavior, is equally as impenetrable to our senses as gravity is. We human beings cannot perceive the gravitational forces that are at work behind the behaviors that we observe (for example, when we spill our coffee or when the Earth obediently moves in its orbit around the Sun). Likewise, we cannot directly observe the relationships between the variables causing the behaviors that we observe in software development. The only option open to us is to gather information about software development indirectly through empirical studies.

Any empirical study is a window onto the software development backstage. But the prospect from such a window gives us only one view of the reality that we are

scrutinizing. Unfortunately, the window is not open; it contains a piece of glass that has a bearing on what we see. Thus, we have to take the same precautions as our friends on the spaceship that just landed on the unexplored planet.

TYPES OF EMPIRICAL RESULTS

If an experiment is not replicated, there is no way to distinguish whether results were produced by chance (the observed event occurred accidentally), results are artifactual (the event occurred because of the experimental configuration but does not exist in reality), or results conform to a pattern existing in reality. Different replication types help to clarify which of these three types of results an experiment yields.

When you see something shocking, striking, and unexpected, what do you do to be sure? You look twice. To build a piece of reliable knowledge from an empirical study, you need to be sure that the observed results did not happen accidentally. So, do it twice! Repeat the study: same experimenters, same lab, same instrumentation, and same protocol. To rule out chance results and get an estimation of the natural variation of the observation, we need identical repetitions.

Still, if the results are repeated in your lab, we need to know if they can be generalized to other sites and researchers. The way to get such a level of certainty in empirical findings is for other researchers in their labs to replicate your study. To rule out local and researcher-dependent results, we need identical replications.

One empirical study, no matter how well designed it is, might produce artifactual results. There exists a relationship between reality and the observation instrument; no one individual empirical study can yield definitive results, as the observation instrument itself (the study setting) may be affecting some aspects of the findings. Other designs, other protocols, and other instrumentations will be able to confirm if results hold. To rule out artifactual results, we need conceptual replications.

One type of empirical study offers one view of the phenomenon under observation. In order to make our evidence more reliable, we have to go a step further and observe the phenomenon using another type of instrument. Other types of empirical studies observing the same phenomenon provide new perspectives. Different types of empirical studies (experiments, observational studies, historical studies, case studies, surveys...) and empirical paradigms (qualitative and quantitative approaches) provide complementary views of the reality that we are studying, and we can piece together a more accurate picture of the development phenomenon under study by synthesizing their results.

DO'S AND DON'T'S

What is the moral of this story? Results from one single empirical study are just a preliminary piece of information; we should even consider it just an anecdote. Our studies have to be repeated, replicated, and triangulated in order to form a

reliable idea of any SE phenomenon that we investigate. Building a reliable piece of knowledge out of empirical studies requires:

- Do the same study
 - by the same researchers in the same site. This type of repetition is needed to get away from fortuity and start walking toward evidence.
 - by other researchers in a different site. Other researchers replicate the study with the same protocol as the baseline study. This strategy confirms that the results are independent of the researcher, site, and sample.
 - with a different protocol. Studies (of the same type) should be performed using different settings or protocols (operationalize the variables differently, use other study designs, other measurement processes, etc.). This strategy guarantees that the results are independent of the instrument used.
- Do a different study with same goals. Alternative studies should be conducted. Observing the same reality through different types of studies provides new information that cannot be gathered from the baseline empirical study type.

Notice that leaving out steps might lead to uncertain situations. If you move directly to step three and the results you get are different (of which there is a high probability), you will be unable to trace back the source of variations since there are so many. The new information, and the old information will not fit together and nothing new can be learnt. Through baby steps, each study will contribute, with pieces fitting together like parts of a puzzle, and the bigger picture will emerge.

Just because it once happened that you observed something emerging from a set of data (being either produced by your study or borrowed from a repository) *don't* trust it. Replication is the tool science has for being sure that something observed really exists. Do it again!

FURTHER READING

- [1] Gómez O, Juristo N, Vegas S. Understanding replication of experiments in software engineering: a classification. *Inf Softw Technol* 2014;56(8):1033–48.
- [2] Juristo N, Vegas S. The role of non-exact replications in software engineering experiments. *Empir Softw Eng* 2011;16(3):295–324.

Seven principles of inductive software engineering: What we do is different

T. Menzies

North Carolina State University, Raleigh, NC, United States

CHAPTER OUTLINE

Different and Important	13
Principle #1: Humans Before Algorithms	13
Principle #2: Plan for Scale	14
Principle #3: Get Early Feedback	15
Principle #4: Be Open Minded	15
Principle #5: Be Smart with Your Learning	15
Principle #6: Live with the Data You Have	16
Principle #7: Develop a Broad Skill Set That Uses a Big Toolkit	17
References	17

DIFFERENT AND IMPORTANT

Inductive software engineering is the branch of software engineering focusing on the delivery of data-mining based software applications. Within those data mines, the core problem is *induction*, which is the extraction of small patterns from larger data sets. Inductive engineers spend much effort trying to understand business goals in order to inductively generate the models that matter the most.

Previously, with Christian Bird, Thomas Zimmermann, Wolfram Schulte, and Ekrem Kocaganeli, we wrote an *Inductive Engineering Manifesto* [1] that offered some details on this new kind of engineering. The whole manifesto is a little long, so here I offer a quick summary. Following are seven key principles which, if ignored, can make it harder to deploy analytics in the real world. For more details (and more principles), refer to the original document [1].

PRINCIPLE #1: HUMANS BEFORE ALGORITHMS

Mining algorithms are only good if humans find their use in real-world applications. This means that humans need to:

- understand the results
- understand that those results add value to their work.

Accordingly, it is strongly recommend that once the algorithms generate some model, then the inductive engineer *talks to humans* about those results. In the case of software analytics, these humans are the subject matter experts or business problem owners that are asking you to improve the ways they are generating software.

In our experience, such discussions lead to a second, third, fourth, etc., round of learning. To assess if you are talking in “the right way” to your humans, check the following:

- Do they bring their senior management to the meetings? If yes, great!
- Do they keep interrupting (you or each other) and debating your results? If yes, then stay quiet (and take lots of notes!)
- Do they indicate they understand your explanation of the results? For example, can they correctly extend your results to list desirable and undesirable implications of your results?
- Do your results touch on issues that concern them? This is *easy* to check... just count how many times they glance up from their notes, looking startled or alarmed.
- Do they offer more data sources for analysis? If yes, they like what you are doing and want you to do it more.
- Do they invite you to their workspace and ask you to teach them how to do XYZ? If yes, this is a real win.

PRINCIPLE #2: PLAN FOR SCALE

Data mining methods are usually repeated multiple times in order to:

- answer new questions, inspired by the current results;
- enhance data mining method or fix some bugs; and
- deploy the results, or the analysis methods, to different user groups.

So that means that, if it works, you will be asked to do it again (and again and again). To put that another way, *thou shalt not click*. That is, if all your analysis requires lots of pointing-and-clicking in a pretty GUI environment, then you are definitely *not* planning for scale.

Another issue is that as you scale up, your methods will need to scale up as well. For example, in our *Manifesto* document, we discussed the CRANE project at Microsoft that deployed data mining methods to the code base of Microsoft Windows. This was a *very large* project, so the way it started was *not* the way it ended:

- Initially, a single inductive engineer did some rapid prototyping for a few weeks, to explore a range of hypotheses and gain business interest (and get human feedback on the early results).

- Next, the inductive engineering team spent a few months conducting many experiments to find stable models (and to narrow in on the most important business goals).
- In the final stage, which took a year, the inductive engineers integrated the models into a deployment framework that was suitable for a target user base.

Note that the team size doubled at each stage—so anyone funding this work needs to know that increasingly useful conclusions can be increasingly expensive.

PRINCIPLE #3: GET EARLY FEEDBACK

This is mentioned previously, but it is worth repeating. Before conducting very elaborate studies (that take a long time to reach a conclusion), try applying very simple tools to gain rapid early feedback.

So, simplicity first! Get feedback early and often! For example, there are many linear time discretizers for learning what are good divisions of continuous attributes (eg, the Fayyad-Irani discretizer [2]). These methods can also report when breaking up an attribute is *not* useful since that attribute is not very informative. Using tools like these, it is possible to discover what attributes can be safely ignored (hint: usually, it's more than half).

PRINCIPLE #4: BE OPEN MINDED

The goal of inductive engineering for SE is to find better ideas than what was available when you started. So if you leave a data mining project with the same beliefs as when you started, you really wasted a lot of time and effort. Hence, some mantras to chant while data mining are:

- Avoid a fixed hypothesis. Be respectful but doubtful of all human-suggested domain hypotheses. Certainly, explore the issues that they raise, but also take the time to look further afield.
 - Avoid a fixed approach for data mining (eg, just using decision trees all the time), particularly for data that has not been mined before.
 - The most important initial results are the ones that radically and dramatically improve the goals of the project. So seek important results.
-

PRINCIPLE #5: BE SMART WITH YOUR LEARNING

Let's face it, any inductive agents (human or otherwise) have biases that can confuse the learning process. So don't torture the data to meet preconceptions (that is, it is ok to go "fishing" to look for new insights).

It is also true that any inductive agent (and this includes you) can make mistakes. If organizations are going to use your results to change policy, the important outcomes are riding on your conclusions. This means you need to check and validate your results:

- Ask other people to do code reviews of your scripts.
- Check the conclusion stability against different sample policies. For example:
 - Policy1: divide data into (say) ten 90% samples (built at random). How much do your conclusions change across those samples?
 - Policy2: sort data by collection date (if available). Learn from the far past, the nearer past, then the most recent data. Does time change your results? Is time *still* changing your results? It is important to check.
 - Policy3,4,5, etc.: Are there any other natural divisions of your data (eg, east coast versus west coast, men versus women, etc.)? Do they affect your conclusions?
- When reporting multiple runs of a learner, don't just report mean results—also report the wriggle around the mean.
- In fact, do not report mean results at all since outliers can distort those mean values. Instead, try to report median and IQR results (the inter-quartile range is the difference between the 75th and 25th percentile).

PRINCIPLE #6: LIVE WITH THE DATA YOU HAVE

In practice, it is a rare analytics project that can dictate how data is collected in industrial contexts. Usually, inductive engineers have to cope with whatever data is available (rather than demand more data collected under more ideal conditions). This means that often you have to go mining with the data you have (and not the data you hope to have at some later date). So it's important to spend some time on data quality operators. For example:

- Use *feature selection* to remove spurious attributes. There are many ways to perform such feature selection, including the Fayyad-Irani method discussed herein. For a discussion of other feature selection methods, see [3].
- Use *row selection* to remove outliers and group-related rows into sets of clusters. For a discussion on row selection methods, see [4].

One benefit of replacing rows with clusters is that any signal that is spread out amongst the rows can be “amplified” in the clusters. If we cluster, then learn one model per cluster, then the resulting predictions have better median values and smaller variances [5].

In any case, we've often found row and feature selection discards to be up to 80% to 90% of the data, without damaging our ability to learn from the data. This means that ensuring quality of *all* the data can sometimes be less important than being able to extract quality data from large examples.

PRINCIPLE #7: DEVELOP A BROAD SKILL SET THAT USES A BIG TOOLKIT

The reason organizations need to hire inductive engineers is that they come equipped with a very broad range of tools. This is important since many problems need specialized methods to find good solutions.

So, to become an inductive engineer, look for the “big ecology” toolkits where lots of developers are constantly trying out new ideas. Languages like Python, Scala (and lately, Julia) have extensive online forums where developers share their data mining tips. Toolkits like R, MATLAB, and WEKA are continually being updated with new tools.

What a great time to be an inductive engineer! So much to learn, so much to try. Would you want to have it any other way?

REFERENCES

- [1] Menzies T, Bird C, Zimmermann T, Schulte W, Kocaganeli E. The inductive software engineering manifesto: principles for industrial data mining. In: Proceedings of the international workshop on machine learning technologies in software engineering (MALETS '11). New York, NY, USA: ACM; 2011. p. 19–26. <http://dx.doi.org/10.1145/2070821.2070824>.
- [2] Fayyad UM, Irani KB. On the handling of continuous-valued attributes in decision tree generation. *Mach Learn* 1992;8(1):87–102. <http://dx.doi.org/10.1023/A:1022638503176>.
- [3] Hall MA, Holmes G. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans Knowl Data Eng* 2003;15(6):1437–47. <http://dx.doi.org/10.1109/TKDE.2003.1245283>.
- [4] Olvera-López JA, Carrasco-Ochoa JA, Martínez-Trinidad JF, Kittler J. A review of instance selection methods. *Artif Intell Rev* 2010;34(2):133–43. <http://dx.doi.org/10.1007/s10462-010-9165-y>.
- [5] Menzies T, Butcher A, Cok D, Marcus A, Layman L, Shull F, Turhan B, Zimmermann T. Local versus global lessons for defect prediction and effort estimation. *IEEE Trans Softw Eng* 2013;39(6):822–34. <http://dx.doi.org/10.1109/TSE.2012.83>.

RULE #1: TALK TO THE USERS

3

The most important rule in industrial data science is this: talk more to your users than to your algorithms.

Why is it most important? Well, the main difference we see between academic data mining research and industrial data scientists is that the former is mostly focused on algorithms and the latter is mostly focused on “users.”

Note that by this term “user,” we do not mean the end user of a product. Rather, we mean the community providing the data and domain insights vital to a successful project. Users provide the funding for the project and, typically, need to see a value-added benefit, very early in a project.

At one level, talking to the people who hold the purse strings is just good manners and good business sense (because it is those people who might be willing to fund future projects). As the Mercury astronauts used to say, “No bucks, no Buck Rodgers.”

But there is another, more fundamental, reason to talk to business users:

- The space of models that can be generated from any data set is very large. If we understand and apply user goals, then we can quickly focus a data mining project on the small set of most crucial issues.
- Hence, it is vital to talk to users in order to leverage their “biases” for better guiding the data mining.

As discussed in this chapter, any inductive process is fundamentally biased. Hence, we need to build ensembles of opinions (some from humans, some from data miners), each biased in their own particular way. In this way, we can generate better conclusions than using some an overreliance on a single bias.

The rest of this chapter expands on this notion of “bias,” as it appears in humans and data miners.

3.1 USERS BIASES

The Wikipedia page on “List of cognitive biases”¹ lists nearly 200 ways that human reasoning is systematically biased. That list includes

- Nearly a hundred decision-making, belief, and behavioral biases such as *attentional bias* (paying more attention to emotionally dominant stimuli in one’s environment and neglecting relevant data).
- Nearly two dozen social biases such as the *worse than average effect* (believing that we are worse than others at tasks that are difficult).
- Over 50 memory errors and biases such as *illusory correlation* (inaccurately remembering a relationship between two events).

¹ http://en.wikipedia.org/wiki/List_of_cognitive_biases.

**FIGURE 3.1**

Gorillas in our midst. From [408]. Scene from video used in “invisible gorilla” study. Figure provided by Daniel Simons. For more information about the study and to see the video go to www.dansimons.com or www.theinvisiblegorilla.com.

As documented by Simons and Chabris [400], the effects of these human imperfections can be quite startling. For example, in the “Gorilla in our midst” experiment, subjects were asked to count how often a ball was passed between the members of a basketball team wearing white shirts. Nearly half the subjects (48%) were so focused on “white things” that they did not notice a six-foot tall hairy black gorilla walk slowly into the game, beat its chest, then walk out (see Figure 3.1). According to Simons and Chabris, humans can suffer from “sustained in attentional blindness” (which is a kind of cognitive bias) where they do not see effects that, to an outside observer, are glaringly obvious.

The lesson here is that when humans analyze data, their biases can make them miss important effects. Therefore, it is wise to run data miners over that same data in order to find any missed effects.

3.2 DATA MINING BIASES

When human biases stop them from seeing important effects, data miners can uncover those missed effects. In fact, best results often come from *combining* the biases of the humans and data miners (i.e., the whole may be greater than any of the parts).

That kind of multilearner combination is discussed below. First, we need to review the four biases of any data miner:

1. No data miner has access to all the data in the universe. Some *sampling bias* controls which bits of the data we offer to a miner.
2. Different data miners have a *language bias* that controls what models they write. For example, decision tree learners cannot write equations and Bayesian learners cannot write out categorical rules.
3. When growing a model, data miners use a *search bias* that controls what is the next thing they add to a model. For example, some learners demand that new rules cannot be considered unless they are supported by a minimum number of examples.
4. Once a model is grown, data miners also use an *underfit bias* that controls how they prune away parts of a model generated by random noise (“underfit” is also called “overfitting avoidance”).

For example, consider the task of discussing the results of data mining with management. For that audience, it is often useful to display small decision trees that fit on one slide. For example, here's a small decision tree that predicts who will survive the sinking of the *Titanic* (Hint: do not buy third-class tickets).

```

sex = male
|   class = first
|   |   age = adult : died
|   |   age = child : survived
|   class = second
|   |   age = adult : died
|   |   age = child : survived
|   class = third   : died
|   class = crew    : died
sex = female
|   class = first   : survived
|   class = second  : survived
|   class = third   : died
|   class = crew    : survived

```

To generate that figure, we apply the following biases:

- A *language bias* that generates decision trees.
- A *search bias* that does not grow the model unless absolutely necessary.
- An *underfit bias* that prunes any doubtful parts of the tree. This is needed because, without it, our trees may grow too large to show the business users.

For other audiences, we would use other biases. For example, programmers who want to include the learned model in their program have a language bias. They often prefer *decision lists*, or a list of conditions (and the *next* condition is only checked if the *last* condition fails). Such decision lists can be easily dropped into a program. For example, here is a decision list learned from the *S.S. Titanic* data:

```

(sex = female) and (class = first) => survived
(sex = female) and (class = second) => survived
(sex = female) and (class = crew)   => survived
                           => died

```

So bias is a blessing to any data miner because, if we understand it, we can tune the output to the audience. For example, our programmer likes the decision list output as it is very simple to (say) drop it into a "C" program.

```

if ((sex == "female") && (class == "first")) return "survived"
if ((sex == "female") && (class == "second")) return "survived"
if ((sex == "female") && (class == "crew"))   return "survived"
return "died"

```

On the other hand, bias is a challenge because, the same data can generate different models, depending on the biases of the learner. This leads to the question: Is bias avoidable?

3.3 CAN WE AVOID BIAS?

The lesson of the above example (about the *Titanic*) is that, sometimes, it is possible to use user bias to inform the data mining process. But is that even necessary? Is not the real solution to avoid bias all together?

In everyday speech, bias is a dirty word. For example:

Judge slammed for appearance of bias

A Superior Court judge has barred an Ontario Court colleague from presiding over a man's trial after concluding he appeared biased against the defendant.

(www.lawtimesnews.com, September 26, 2011. See <http://goo.gl/pQZkF>).

But bias' bad reputation is not deserved. It turns out that bias is very useful:

- Without bias, we cannot say which bits of the data matter most to us.
- That is, without bias, we cannot divide data into the bits that matter and the bits that do not.

This is important because if we cannot ignore anything, we cannot *generalize*. Why? Well, any generalization must be smaller than the thing it generalizes (otherwise we might as well just keep the original thing and ignore the generalization). Such generalizations are vital to data mining. Without generalization, all we can do is match new situations to a database of old examples. This is a problem because, if the new situation has not occurred before, then nothing is matched and we cannot make any predictions. That is, while bias can blind us to certain details, it can also let us see patterns that let us make predictions in the future.

To put that another way:

Bias make us blind but it also let us see.

3.4 MANAGING BIASES

Bias is unavoidable, even central, to the learning process. But how to manage it?

Much of this book is concerned with automatic tools for handling bias:

- Above, we showed an example where different learners (decision tree learners and rule learners) were used to automatically generate different models for different users.
- Later in this book, we discuss state-of-the-art ensemble learners that automatically build committees of experts, each with their own particular bias. We will show that combining the different biases of different experts produces better predictions than relying exclusively on the bias of one expert.

But even without automatic support, it is possible to exploit expert biases to achieve better predictions. For example, at Microsoft, data scientists conduct "user engagement meetings" to review the results of a data mining session:

- Meetings begin with a quick presentation on the analysis method used to generate the results.
- Some sample data is then shown on the screen, at which point the more informed business users usually peer forward to check the data for known effects in that business.

- If the data passes that “sanity check” (that it contains old conclusions), the users start investigating the data for new effects.

Other measures of success of such meetings are as follows. In a good user engagement meeting

- The users keep interrupting to debate the implications of your results. This shows that (a) you are explaining the results in a way they understand, and (b) your results are commenting on issues that concern users.
- Data scientists spend more time listening than talking as the users propose queries on the sample data, or if they vigorously debate implications of the displayed data.
- At subsequent meetings, users bring their senior management.
- The users start listing more and more candidate data sources that you could exploit.
- After the meeting, the users invite you back to their desks inside their firewalls to show them how to perform certain kinds of analysis.

3.5 SUMMARY

The conclusions made by any human- or computer-based learning will be biased. When using data miners, the trick is to match the biases of the learner with the biases of the users. Hence, data scientists should talk more, and listen more, to business users in order to understand and take advantage of the user biases.

There's never enough time to do all the testing you want

K. Herzog

Software Development Engineer, Microsoft Corporation, Redmond, United States

CHAPTER OUTLINE

The Impact of Short Release Cycles (There's Not Enough Time)	91
Testing Is More Than Functional Correctness (All the Testing You Want)	92
Learn From Your Test Execution History	92
Test Effectiveness	93
Test Reliability/Not Every Test Failure Points to a Defect	93
The Art of Testing Less	93
Without Sacrificing Code Quality	94
Tests Evolve Over Time	94
In Summary	94
References	95

Software is present in nearly every aspect of our daily lives and also dominates large parts of the high-tech consumer market. Consumers love new features, and new features are what makes them buy software products, while properties like reliability, security, and privacy are assumed. To respond to the consumer market demand, many software producers are following a trend to shorten software release cycles. As a consequence, software developers have to produce more features in less time while maintaining, or even increasing, product quality. Here at Microsoft (as well as other large software organizations [1]) we have learned that testing is not free. Testing can slow down development processes and cost money in terms of infrastructure and human involvement. Thus, the effort associated with testing must be carefully monitored and managed.

THE IMPACT OF SHORT RELEASE CYCLES (THERE'S NOT ENOUGH TIME)

To enable faster and more “agile” software development, processes have to change. We need to cut down time required to develop, verify, and ship new features or code changes in general. In other words, we need to increase *code velocity* by increasing the effectiveness, efficiency, and reliability of our development processes.

Focusing on testing processes, it is important to realize that verification time is a lower bound on how fast we can ship software. However, nowadays this lower bound frequently conflicts with the goal of faster release cycles. As a matter of fact, we simply cannot afford to execute all tests on all code changes anymore. Simply removing tests is easy, the challenge is to cut tests without negatively impacting product quality.

TESTING IS MORE THAN FUNCTIONAL CORRECTNESS (ALL THE TESTING YOU WANT)

Often, testing is associated with checking for functional correctness and unit testing. While these tests are often fast, passing or failing in seconds, large complex software systems require tests to verify system constraints such as backward compatibility, performance, security, usability, and so on. These *system and integration tests* are complex and typically time-consuming, even though they relatively rarely find a bug. Nevertheless, these tests must be seen as an insurance process verifying that the software product complies with all necessary system constraints at all times (or at least at the time of release). Optimizing unit tests can be very helpful, but usually it is the system and integration testing part of verification processes that consumes most of the precious development time.

LEARN FROM YOUR TEST EXECUTION HISTORY

Knowing that we cannot afford to run all tests on all code changes anymore, we face a difficult task: find the best combination of tests to verify the current code change, spending as little test execution time as possible. To achieve this goal, we need to think of testing as a risk management tool to minimize the risk of elapsing code defects to later stages of the development process or even to customers.

The basic assumption behind most test optimization and test selection approaches is that for given scenarios, or context C , not all tests are equally well suited. Some tests are more effective than others. For example, running a test for *Internet Explorer* or the *Windows kernel* code base is unlikely to find new code defects.

However, determining the effectiveness and reliability of tests and when to execute which subset is not trivial. One of the most popular metrics to determine test quality is *code coverage*. However, coverage is of very limited use in this case. First, coverage does not imply verification (especially not for system and integration tests). Second, it does not allow us to assess the effectiveness and reliability of single test cases. Last but not least, collecting coverage significantly slows down test runtime, which would require us to remove even more tests.

Instead, we want to execute only those tests that, for a given code change and a given execution context C , eg, branch, architecture, language, device type, has high reliability and high effectiveness. Independent from the definition of reliability and

effectiveness, all tests that are not highly reliable and effective should be executed less frequently or not at all.

TEST EFFECTIVENESS

Simplistically, a test is effective if it finds defects. This does not imply that tests that find no defects should be removed completely, but we should consider them of secondary importance (see “The Art of Testing Less” section). The beauty of this simplistic definition of test effectiveness is that we can use *historic test execution data* to measure test effectiveness. For a given execution context C , we determine how often the test failed due to a code defect. For example, a test T that has been executed 100 times on a given execution context C and that failed 20 times due to code defects has a historic code defect probability of 0.2. To compute such historic code defect probabilities, it usually suffices to query an existing test execution database and to link test failures to issue reports and code changes, a procedure that is also commonly used to assess the quality of source code. Please note that coverage information is partially included in this measurement. Not covering code means not being able to fail on a code change, implying a historic failure probability of 0.

TEST RELIABILITY/NOT EVERY TEST FAILURE POINTS TO A DEFECT

Tests are usually designed to either pass or fail, and each failure should point to a code defect. In practice, however, many tests tend to report so-called *false test alarms*. These are test failures that are not due to code defects, but due to test issues or infrastructure issues. Common examples are: wrong test assertions, non-deterministic (flaky) tests, and tests that depend on network resources that fail when the network is unavailable.

Tests that report false test alarms regularly must be considered a serious threat to the verification and development processes. As with any other test failure, false alarms trigger manual investigations that must be regarded as wasted engineering time. The result of the investigation will not increase product quality, but rather, slow down code velocity of the current code change under the test.

Similar to test effectiveness, we can measure test reliability as a historic probability. Simplistically, we can count any test failure that did not lead to a code change (code defect) as false test alarm. Thus, a test T that has been executed 100 times on a given execution context C and that failed 10 times but did not trigger a product code change has a historic test unreliability probability of 0.1.

THE ART OF TESTING LESS

Combining both measurements for effectiveness and reliability (independent from their definition) allows a development team to assess the quality of individual tests and to act on it. Teams may decide to statically fix unreliable tests or to dynamically

skip tests. Tests that show low effectiveness and/or low reliability should be executed only where necessary and as infrequently as possible. For more details in how to use these probabilities to design a system that dynamically determines which test to execute and which to not execute, we refer to Herzig et al. [2] and to Elbaum et al. [1].

WITHOUT SACRIFICING CODE QUALITY

However, the problem is that some tests might get disabled completely, either because they are too unreliable, or because they never found a code defect (in the last periods). To minimize the risk of elapsing severe bugs into the final product and in order to boost the confidence of development teams in the product under development, it is essential to prevent tests from being disabled completely. One possible solution is to regularly force test executions, eg, once a week. Similarly, you can also use a version control branch-based approach, eg, executing all tests on the trunk or release branch, but not on feature and integration branches.

TESTS EVOLVE OVER TIME

Any complex test infrastructure evolves over time. New tests are added, while older tests might become less important or even deprecated. Maintaining tests and preventing test infrastructures from decay can grow to be a significant effort. For products with some history, some of the older tests may not be “owned” by anybody anymore, or show strongly distributed ownership across multiple product teams. Such ownership can impact test effectiveness and reliability, slowing down development speed [3]. Determining and monitoring new test cases being added, or changes to existing tests, can be very useful in assessing the healthiness of the verification process. For example, adding lots of new features to a product without experiencing an increase in a new test being written might indicate a drop in quality assurance. The amount of newly introduced code and newly written or at least modified test code should be well balanced.

IN SUMMARY

Software testing is expensive, in terms of time and money. Emulating millions of configurations and devices requires complex test infrastructures and scenarios. This contradicts today’s trend of releasing complex software systems in ever-shorter periods of time. As a result, software testing became a bottleneck in development processes. The time spent in verification defines a lower bound on how fast companies can ship software. Resolving test bottlenecks requires us to rethink development and testing processes to allow new room for newer and better tests and to regain confidence in testing. We need to accept that testing is an important part of our daily development process, but that “There's never enough time to do all the testing we want.”

REFERENCES

- [1] Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE); 2014.
- [2] Herzig K, Greiler M, Czerwonka J, Murphy B. The art of testing less without sacrificing quality. In: Proceedings of the 2015 international conference on software engineering; 2015.
- [3] Herzig K, Nagappan N. Empirically detecting false test alarms using association rules. In: Companion proceedings of the 37th international conference on software engineering; 2015.

Why provenance matters

M.W. Godfrey

University of Waterloo, Waterloo, ON, Canada

CHAPTER OUTLINE

What's Provenance?	228
What are the Key Entities?	228
What are the Key Tasks?	228
Another Example	230
Looking Ahead	231
References	231

Here's a problem: You're a lead developer for a company that produces a web content management system, and you get email from an open source project leader who claims that your closed source project probably contains some of their open source code, in violation of their license. She lists three features of your product that she suspects are built around code copied from her system, and she asks you to investigate and respond. Can you determine if her claim has merit? How do you investigate it?

Here's another problem: Your company uses a commercial Java application that ships with embedded third-party libraries. One of these libraries is known to have had a major security flaw in version 2.1 that was fixed in version 2.1.1. Are you vulnerable? How can you tell?

Here's one more: You've recently agreed to take on the role of bug triage for a large open source project; that is, you are to examine each incoming bug report, decide if the report is legitimate and worth proceeding with, and, if so, assign a developer to investigate. However, often a single bug will manifest itself in different ways; obviously you don't want to assign multiple developers to fix the same underlying defect. So how can you tell if a given bug report is truly new, and not just a manifestation of something you've seen before? That is, how can you effectively "de-dup" a voluminous set of bug reports?

WHAT'S PROVENANCE?

What these problem scenarios have in common is that they depend on the *provenance* of software entities [1]; that is, we need to be able to analyze various kinds of evidence that pertain to the origin, history, and ownership of software entities to answer questions such as:

- What is the history of this entity? How did it come to be where it is currently?
- What other entities is it related to, and how?
- What is the evidence? How reliable is it?

Understanding the raw design artifacts that comprise our system—such as the source code, documentation, build recipes, etc.—gives us only a small part of the larger story. We need also to be able to reason about the relatedness of a wide variety of development artifacts, the processes that produce and manage them, their history and evolution, and the people involved and their roles.

WHAT ARE THE KEY ENTITIES?

To be able to address these issues, we need to be able to identify which particular entities we are interested in, and how they relate to each other. There are several kinds of entities that we might want to consider:

- software artifacts that are managed by some tool as first-class entities, such as source code files, commits in a version control system (VCS), or bug reports in an issue tracking system;
- attributes of these artifacts, such as version identifiers, current status, comment histories, and timestamps;
- synthetic entities such as software features and maintenance tasks, which have strong meaning to developers but may not have crisp definitions or even independent existence; and
- relationships between any of the preceding items, which may be explicit, implicit, inferred, or even probabilistic.

WHAT ARE THE KEY TASKS?

Armed with a broad understanding of the kinds of entities in our project, we can then consider how we might use them to answer our questions. There are two key tasks that we need to be able to perform here:

- defining and scoping the entities that are of particular interest, and
- establishing artifact linkage and ground truth.

Additionally, we often need techniques to be able to explore the entity space, such as:

- scalable matching algorithms to query the large entity space, and
- various kinds of historical analyses to answer questions about the evolution of the system.

Depending on the task at hand, *defining and scoping the entities of interest* may be straightforward, or it may require tool support and/or manual intervention. For example, in the first problem scenario, we need to decide how to define and scope a feature. Then, we need to be able to map these features to sets of code fragments for both our system and the open source system. Finally, we need to examine the VCS commit history of the code that is related to these features within our system (and if the VCS history of the open source system is available to us, we can do the same for the open source system).

At this point we have a set of features, decomposed into code fragments across two systems, and a set of related VCS commits that touch these code fragments. *Establishing artifact linkage and ground truth* is the next problem we need to address. When the open source project leader mentioned three high level features she thought had been copied, we performed the mapping of the features to source code on both systems. Did we get it right? On the other hand, if we trust our decomposition, then the commit history gleaned from the VCS should be accurate, unless someone has rebased our git repository [2].

Now we probably want to compare code fragments of the two systems on a feature-by-feature basis using a comparison tool, such as diff, which compares raw text, or a code clone detector like CCFinder [3] or ConQAT [4], which has some understanding of the programming language. Because we've narrowed down our field of interest to a manageable set of code fragments, this step will probably be doable quickly.

But let's suppose that our company's lawyer now steps in with a bigger challenge: he wants us to make sure that nothing in our codebase closely matches any code in any of the five open source systems that are in our problem domain. Lucky for us, code clone detector tools are usually designed to be able to process large systems. In provenance analysis in general, we often have to compare complex structures such as source code against large datasets; *scalable matching algorithms* are required to make this feasible. This can often be achieved via a two-stage process:

1. Preprocessing of the artifacts reduces the dimensionality of the data, say by hashing each line of code; the preprocessed data can then be compared relatively quickly as sets or sequences of hash values.
2. When a quick comparison suggests a "hit," more complicated approaches can then be used on the original artifacts to prune away false positives.

Of course, in reducing the dimensionality of the data, we are losing information; for example, if I copy a method and then make a few changes, any line that is even slightly different from the original will result in a different hash value. But if we move to the granularity of tokens or characters, then the comparisons become much more time consuming. So we are always trading off accuracy against performance.

Finally, if we decide that some of the code in the two systems appears to be unusually similar, we can *perform a historical analysis* on the commit trails, to see when and how any “borrowed” code fragments might have made it into our codebase, and which developers might have been responsible. We may also wish to look for other historic trends, such as unusually large commits by inexperienced developers, since that might also indicate the incorporation of third-party code into the system.

ANOTHER EXAMPLE

Let’s return to the second problem scenario now, since it’s one we’ve explored in real life [5]. A company approached us with the problem of how to authoritatively ascertain the version identifier of a third-party Java library binary that has been packaged within a commercial application. This sounds like a simple problem, but it’s surprisingly tricky to solve. Sometimes, the version identifier forms part of the name of the included jar file; however, this is merely developer convention, it is not always observed and isn’t enforceable technically. Worse, in our investigations we found instances of where the version identifier had been edited to remove the “release candidate” designation, falsely implying that the included library was the final version.

Our first step was to decide on our entities of interest: versions of Java libraries that are in common use. Next, we needed to establish ground truth; that is, we decided to build a master database that effectively represented all recent versions of these libraries. We decided to use the Maven2 [6] repository as our data source, since it is very large and we found it to contain instances of almost all recent versions of almost all common Java libraries.

Our next problem was how to achieve scalable matching of our candidate library against the master repository. We couldn’t just compare source code against source code because our candidate library might not contain any; similarly, we couldn’t compare byte code against byte code because the binaries of two identical Java programs might differ if one used a different compiler or command line options. Instead, we extracted the signatures of the methods that the library contained; these would be extractable from both source code and byte code, and would not be affected by compiler choice. We then took the SHA1 hash of the library’s API (declaration plus method signatures), and let that one hash value represent the library version in our database. So we downloaded all of Maven—no mean feat!—extracted the API information from every version of every library in it, and created the hashes using the provided version identifier as ground truth. This took some time and effort. However, once the master repository had been built, analyzing an application was very fast: we extracted the API from the candidate libraries, found the corresponding SHA1 hash value, and looked it up in the database to see if there was an authoritative version identifier.

Of course, there are a couple of problems with our approach: the library or the particular version we’re looking for might not be in Maven at all. Worse, if the

API of a library remains stable across several versions, we may match multiple versions, requiring manual intervention to disambiguate. In practice, however, we found that these were rarely serious problems. New library histories could be added as needed to the master repository for next time, and manual intervention rarely required manually examining more than a small handful of possible matches.

LOOKING AHEAD

As a profession, we are getting better at instrumenting our processes and tracking our steps. As a profession, we are also increasingly concerned with questions of ownership, origin, evolution, and transparency of the various development artifacts we interact with on a daily basis. As more and more of our artifacts and processes are managed by tools, automatically logged, and annotated with metadata, so will the job of provenance analysis become simpler. And, of course, while we have examined the space of software development artifacts here, the issues and techniques of provenance are no less important for the broader field of data science [7].

REFERENCES

- [1] Godfrey MW. Understanding software artifact provenance. *Sci Comput Program* 2015;97(1):86–90.
- [2] Kalliamvakou E, Gousios G, Blincoe K, Singer L, German D, Damian D. An in-depth study of the promises and perils of mining GitHub. *Empir Softw Eng* [Online 05.09.15].
- [3] <http://www.ccfinder.net>.
- [4] <http://www.conqat.org/>.
- [5] Davies J, German DM, Godfrey MW, Hindle A. Software Bertillonage determining the provenance of software development artifacts. *Empir Softw Eng* 2013;18(6):1195–237.
- [6] <http://maven.apache.org/>.
- [7] <http://cacm.acm.org/blogs/blog-cacm/169199-data-science-workflow-overview-and-challenges/fulltext>.

Why theory matters

D.I.K. Sjøberg*†, G.R. Bergersen*, T. Dybå†

*University of Oslo, Norway** SINTEF ICT, Trondheim, Norway†

CHAPTER OUTLINE

Introduction	29
How to Use Theory	30
How to Build Theory	30
Constructs	31
Propositions	31
Explanation	32
Scope	32
In Summary: Find a Theory or Build One Yourself	32
Further Reading	33

INTRODUCTION

Data without theory is blind, but theory without data is mere intellectual play.

Paraphrased from Kant

It is relatively easy to generate and acquire much data from software engineering (SE) activities. The challenge is to obtain meaning from the data that represents something true, rather than spurious. To increase knowledge and insight, more theories should be built and used.

- Theories help predict what will happen in the future.
- Theories explain why things happen, that is, causes and effects as opposed to only identifying correlations.
- Theories help reduce the apparent complexity of the world.
- Theories summarize, condense and accumulate knowledge.

HOW TO USE THEORY

More than 10 years ago, we conducted an experiment on the effect of two different control styles in Java programs. The “centralized control style” was supposed to represent poor object-oriented programming. The alternative, the “delegated control style,” was supposed to represent good object-oriented programming. Among the 156 consultants who participated, we found no difference between the control styles regarding the time spent on solving the given tasks, given that the solutions were correct.

Does this imply that type of control style does not matter? No, when digging into the data, we found that only the senior consultants performed consistently better on the “good” solution. For the junior consultants, the result was reversed; they consistently performed better on the “poor” solution.

So, we found the opposite effect of the control style depending on the category of consultants. That one style benefitted juniors and another one benefitted seniors, did that happen by coincidence in our experiment or did we encounter a more general phenomenon described in an existing theory? First, we searched through a literature review on the use of theories in SE experiments conducted by Hannay et al., but found no relevant theories. Then we searched in the scientific psychology literature and found Sweller’s cognitive load theory (admittedly, after searching quite a long time). It states that the cognitive load of novices in learning processes may require other strategies than those that may benefit experts. One effect of the cognitive load theory is the “the expertise reversal effect,” a term coined by Kalyuga et al. to denote the phenomenon that dictates that the best instructional technique may depend on the skills of the learner.

If we replace “instructional technique” with “control style” and consider senior consultants as experts and junior consultants as novices, the expertise reversal effect may also be applied to our case with control style.

HOW TO BUILD THEORY

To make SE a more mature, scientific discipline, we need theories that describe SE phenomena. Theory building may involve adapting an existing theory or developing a theory from scratch. Either case is certainly challenging, but one needs to be brave enough to start theorizing and then seek critique for the proposed theories. Through successive improvements made by the community, the theories will become better and better.

Even though it has hardly been discussed in the SE literature, could it be that the reversal effect that we found in the aforementioned experiment is a general phenomenon in SE? To start theorizing on this issue, let us propose a theory on the reversal effect of SE technology according to the elements recommended in Sjøberg et al.:

- Constructs (the basic concepts of the theory)
- Propositions (the interactions among the constructs)

- Explanations (the reasons for the claimed interactions)
- Scope (the circumstances under which the constructs, propositions, and explanations are valid)

Following is the theory on the reversal effect described according to this framework.

CONSTRUCTS

- Software development skill (the ability of a person to develop software)
- SE technology (method, technique, tool, language, etc. used to support software development)
- Performance (the quality gained and the time saved from using the technology)
- SE technology difficulty (how difficult it is to master the technology to exploit its potential; the more difficult the technology is, the better skills and/or more practice is needed to master it)
- Competing SE technologies (technologies that are supposed to support the same kind of SE activities)

PROPOSITIONS

- P1: Given that one masters two competing SE technologies, using the most difficult SE technology offers the highest development performance.
- P2: Given two competing SE technologies, T1 with little difficulty and T2 with great difficulty: T1 gives higher development performance than T2 for low-skilled developers; T2 gives higher development performance than T1 for high-skilled developers; see [Fig. 1](#).

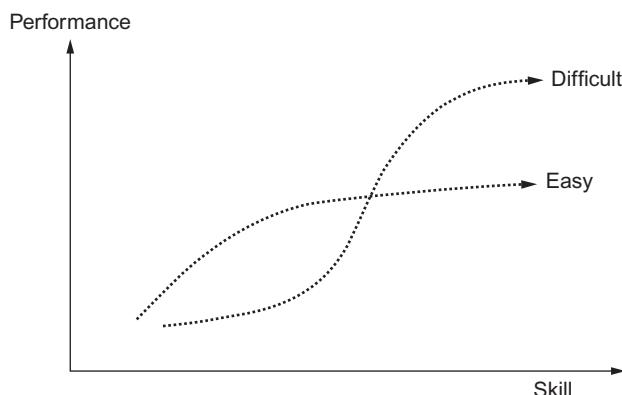


FIG. 1

A illustration of the propositions of the theory.

EXPLANATION

P1 may be explained by the general case that a more sophisticated technology may have many powerful features that lead to increased performance, but at the same time may be difficult to use. For example, a helicopter may transport more people and goods longer and faster, and across more diverse terrain than a bike, but a helicopter is also more difficult to use.

P2 may be explained by the general case that if you have not mastered the sophisticated technology, you will benefit more from using a less sophisticated technology. When asked to solve an unfamiliar problem, powerful features of the technology may require the understanding of certain abstractions. If one has no idea of what these abstractions represent, solving the problem becomes even harder. In those cases, one will benefit from stepping through the details that are available rather than dealing with abstractions that one does not understand. Think of a helicopter versus a bicycle if one does not know how to operate either. The pedals have a clear connection with the movement of the rear wheel. The operations of a helicopter are much more abstract and thus more difficult to operate without pre-established skills.

SCOPE

A theory should represent knowledge that is somewhat widely applicable, but still not trivial, or one that follows from a definition. For example, stating in a theory that “software development is a human activity” provides no new knowledge. In disciplines that involve human behavior, one would hardly find such universal theories as Einstein’s general theory of relativity in physics. Given the many context factors that affect software development, it is a tough challenge to define an appropriate scope for an SE theory. In the case of the theory proposed here, we consider the scope to include only technologies that are either in use in industry or are likely to be useful on the basis of evidence from experiments in an artificial setting.

Kurt Lewin stated more than 60 years ago, “there is nothing so practical as a good theory.” Particularly in an engineering discipline like SE, theory should have practical applications. For example, the theory proposed herein could be used in cost-benefit considerations: if an organization has low-skilled (or high-skilled) developers, an easy (or difficult) SE technology should be used. Of course, there are many trade-offs, but then one should consider whether it is sensible to let the high-skilled developers construct a system if only the low-skilled developers will perform maintenance in the future.

IN SUMMARY: FIND A THEORY OR BUILD ONE YOURSELF

So, the message here is that if you have obtained a set of data, you should interpret your data in the context of theory. If you do not find such a theory, use your data as a starting point for building theory yourself.

Note, paraphrasing Bunge, “premature theorizing is likely to be wrong—but not sterile—and that a long deferred beginning of theorizing is worse than any number of failures.” Our proposed theory is obviously not generally valid. There are clearly cases where no reversal effect exists; that is, between two competing technologies, one of them may give better performance for all skill levels. A more refined theory may classify SE technologies into two more categories: easy to use technologies that give increased performance for all skill levels (this is a mega success) and more difficult to use technologies that give decreased performance for all skill levels (there are many examples of such failures).

Furthermore, a construct like efficiency or cost-benefit may be included in the theory. Two technologies may have similar efficiency if one of them is twice as difficult to learn as the other, one but also gives twice the performance. However, measuring difficulty and performance on a ratio scale may be difficult. Further theorizing and critiquing are therefore required.

It is the task of the research community to collect or produce more data that may strengthen, refine or refute a proposed theory. But first someone has to propose the initial theory. Why not you?

FURTHER READING

- [1] Arisholm E, Sjøberg DIK. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Trans Softw Eng* 2004;30(8):521–34.
- [2] Hannay JE, Sjøberg DIK, Dybå T. A systematic review of theory use in software engineering experiments. *IEEE Trans Softw Eng* 2007;33(2):87–107.
- [3] Kalyuga S, Ayres P, Chandler P, Sweller J. The expertise reversal effect. *Educ Psychol* 2003;38:23–31.
- [4] Lewin K. The research center for group dynamics at Massachusetts Institute of Technology. *Sociometry* 1945;8:126–35.
- [5] Sjøberg DIK, Dybå T, Anda BCD, Hannay JE. Building theories in software engineering. In: Shull F, Singer J, Sjøberg D, editors. *Advanced topics in empirical software engineering*. Berlin: Springer; 2008. p. 312–36.
- [6] Sweller J. Cognitive load during problem solving: effects on learning. *Cognit Sci* 1988;12:257–85.

Actionable metrics are better metrics

A. Meneely

Rochester Institute of Technology, Rochester, NY, United States

CHAPTER OUTLINE

What Would You Say... I Should DO?	284
The Offenders	284
Number of Bugs	285
Code Churn	285
Actionable Heroes	285
Number of Developers	286
Number of Callees (Coupling) and Number of Parameters	286
Cyclomatic Complexity: An Interesting Case	286
Are Unactionable Metrics Useless?	287
Reference	287

- “Frank, I’m sorry, we’re letting you go.”
- “Why? What did I do wrong? I’ve been your best developer for 30 years!”
- “We ran some metrics and found that your code used the letter ‘m’ 10 times more than anyone else.”
- “What does that matter?”
- “We’ve found that code using the letter ‘m’ is more likely to have memory leaks, and you use it all the time, don’t you?”
- “You mean malloc? But, you asked me to rewrite our memory manager!”
- “Not our problem. Data never lie.”

True, data is just data. But humans use data to lie all the time... with metrics. Metrics are a huge part of our evidence-based society. Metrics drive what data we collect, determine the exact phrasing of our conclusions, and shape our perspectives. One wrong step with a metric, and we can grossly misunderstand the world around us.

In software development, a common data misinterpretation is to always assume that a metric is *actionable*. That is, are we using this metric to directly drive decisions? Does the metric actually tell us what to do next? Or is it more of an indication of a condition?

WHAT WOULD YOU SAY... I SHOULD DO?

An actionable metric is one that inspires us to improve. Consider these metrics in our everyday lives:

- Hours spent in the gym
- Speed of your car in relation to the speed limit
- Number of dollars spent on movies

In each of those situations, a specific property is being measured—and that property is something that the person being measured has direct control over. What happens when “Number of hours spent in the gym” decreases? Improving upon that metric is easy: go to the gym!

Now consider these metrics:

- Number of home runs hit this season
- Number of hackers who attacked this operating system
- Body temperature
- Exam grade

A general manager would be unfair to approach his players and say “hit more home runs,” or a CIO telling her system administrators: “get hacked less.” That’s not useful. Everybody wants those metrics to improve, but *how* improvement is to be made is not a directly controllable action for the people involved. The preceding metrics, rather, are *emergent* properties: they are the result of many good forces cooperating at once.

Consider body temperature as another example. In medicine, a high body temperature is a *symptom* of an illness, not the *cause*. Note that body temperature is still useful as a metric because it indicates problems and can guide treatment, but it does not provide an obvious direction to doctors on a cure.

The distance between actionable metrics and emergent properties can be a wide chasm. Notice how “Hours spent in the gym” is not necessarily a direct measure of your overall physical fitness: perhaps you get exercise in other ways. Gym attendance metrics in general would be meaningless if you lacked a gym membership to begin with. Thus, actionable metrics are typically tied to a *process* in which they are intended to be used. In software development, a developer who is slow at responding to pull requests could be a large problem, or a small one depending on how pull requests are used. Bridging the gap between actionable metrics and emergent properties is a regular part of good project management.

THE OFFENDERS

Unactionable metrics are commonplace in software engineering. Two metrics stand out as being particularly frustrating to developers: number of bugs and code churn.

NUMBER OF BUGS

The “number of bugs” metric is ubiquitous in software quality discussions. Defect density, failure rate, fault occurrences, and many other variations all try to show the same picture: how many problems have we had with this system? Ideally, software quality is much more nuanced than simply the number of bug reports, but managers count what they have, and the number of bugs is easy to count.

Unfortunately, the existence of a bug is not the result of one simple action. A single bug can be a combination of all kinds of software engineering failures: requirements mistakes, design flaws, coding errors, lack of testing, miscommunication, or poor process. To aggregate all of these complex scenarios into a single metric and then ask developers to reduce them is simply not helpful to developers. To behave as if it were actionable is indicative of a get-it-right-the-first-time mentality instead of continuous improvement. Number of bugs, therefore, is *emergent*, not actionable.

CODE CHURN

Code churn is a metric that describes the amount of change a module has undergone. As a software project evolves, source code is added and deleted constantly. By mining the source code management system (such as the Git or Subversion repository), researchers can collect the number of lines of code that have changed for a given file over time. Historically, churn is a predictor of quality problems, such as bugs and vulnerabilities: files that change a lot overall are more likely to have big problems later. In the empirical software engineering research world, code churn has become a staple in bug prediction models.

But, code churn is difficult to act upon. What should developers do... not change code? No bugfixes? No new features? No compatibility upgrades? Of course not! That code churn works in a bug prediction model means that we have a good indication of where problems will be, but we don’t understand why. Code can churn for all sorts of reasons, most of which are unavoidable.

While code churn by itself is unavoidable, specific variations of code churn can be more actionable. For example, suppose we define a metric called “un-reviewed churn” that counts the number of changes that did not undergo a code inspection. A high un-reviewed churn has a clear recommendation. If we have empirical support that the metric is also correlated with bugs (and we do from some recent studies), then the recommendation is clear: do more code reviews on those changes.

ACTIONABLE HEROES

After seeing the worst, let’s take a look at some metrics that are actionable in interesting ways.

NUMBER OF DEVELOPERS

Source code management is not just a record of what code changed, it can also be a record of how developers are working together. When two developers make changes to the same source code file, they have a common cause. One useful metric that comes out of this is the “number of different developers who changed this file.” The metric typically defined over a period of time, such as 1 year or the most recent release. Historically, this metric is correlated with later having bugs and vulnerabilities—the more developers who work on a file, the higher the probability of bugs later on.

“Number of developers” is important because it can indicate organizational problems in your team. If a file has 200 developers committing to a single source code file, then chances are they are not all coordinating with each other to make joint decisions. Unintended effects of many changes at once can plausibly introduce bugs. A high number of developers might also indicate design problems that force developers to maintain a monolithic file. Build files, for example, might need to be touched by many developers if not properly broken down according to a subsystem.

NUMBER OF CALLEES (COUPLING) AND NUMBER OF PARAMETERS

When examining the source code itself, metrics such as “number of method callees” and “number of parameters” can indicate actionable problems in your product. If a method makes many external calls, or accepts many different parameters, perhaps that method has too much responsibility. A well-designed method should have a single responsibility that is clearly understood. Reducing the number of callees and number of parameters through refactoring can directly help this responsibility problem.

CYCLOMATIC COMPLEXITY: AN INTERESTING CASE

Metrics can be actionable, but not empirically useful at the same time. McCabe’s cyclomatic complexity is one such metric. Broadly speaking, cyclomatic complexity is derived by counting the number of potential paths through the system (typically at the method level). Originally designed to estimate the number of unit tests a method needs, cyclomatic complexity is built into a lot of metric tools and static analysis tools. Developers can and often do use it to measure their notion of “cognitive” complexity, and use it to target their refactoring efforts. Lower the cyclomatic complexity, and improve the testability of your code. However, many empirical studies have shown that, historically, cyclomatic complexity is strongly correlated with the number of lines of code [1]. In fact, cyclomatic complexity provides no new information in predicting bugs than the size of the method (or file or other unit of measure). Well-designed code will be both more concise and have fewer conditionals. Therefore, in comparison to the easier-to-collect “number of lines of code,” cyclomatic complexity provides very little new information: it is actionable, but typically not useful.

ARE UNACTIONABLE METRICS USELESS?

Of course not! Unactionable metrics (aka emergent metrics) can be quite useful in telling us about the symptoms of a problems. Like “number of bugs,” we can get an overall assessment of how the product is doing. Emergent metrics are most useful when they are a surprise, for example, when a student’s grade comes back lower than expected. We simply cannot rely upon emergent metrics to diagnose the problem for us.

Thus, an *actionable* software development metric is one that measures a property directly under a developer’s control. An *emergent* metric is one that aggregates a variety of potential problems into a single measurement without an obvious diagnosis. Actionable metrics tell us how to improve, whereas emergent metrics tell us where we are. Both are useful in their own right, but our expectations and applications of these metrics must be consistent with the information they provide.

REFERENCE

- [1] Shin Y, Meneely A, Williams L, Osborne JA. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans Softw Eng* 2011;37(6):772–87.

Correlation is not causation (or, when not to scream “Eureka!”)

T. Menzies

North Carolina State University, Raleigh, NC, United States

CHAPTER OUTLINE

What Not to Do	327
Example	327
Examples from Software Engineering	328
What to Do	329
In Summary: Wait and Reflect Before You Report	330
References	330

WHAT NOT TO DO

Legend has it that Archimedes once solved a problem sitting in his bathtub. Crying *Eureka!* (“I have it!”), Archimedes leapt out of the bath and ran to tell the king about the solution. Legend does not say if he stopped to get dressed first.

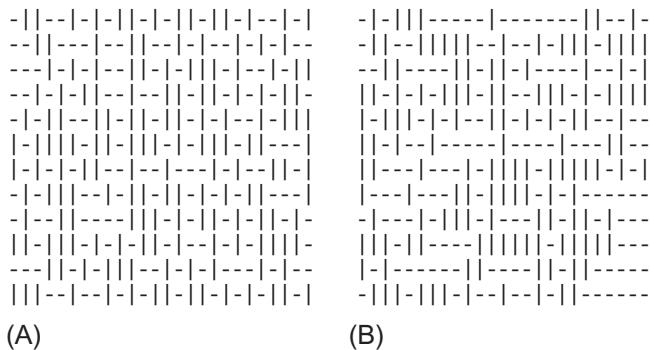
When we stumble onto some pattern in the data, it is so tempting to send a *Eureka!* text to the business users. This is a natural response that stems from the excitement of doing science and discovering an effect that no one has ever seen before.

Here’s my warning: don’t do it. At least, don’t do it straight away.

I say this because I have often fallen into the trap of *correlation is not causation*. Which is to say, just because some connection pattern has been observed between variables does not necessarily imply that a real-world causal mechanism has been discovered. In fact, that “pattern” may actually just be an accident—a mere quirk of cosmic randomness.

EXAMPLE

For an example of nature tricking us and offering a “pattern” where, in fact, no such pattern exists, consider the following two squares (see Fig. 1). (This example comes from Norvig.) One of these was generated by people pretending to be a coin toss

**FIG. 1**

Coin toss patterns (which one is truly random?)

while the others were generated by actually tossing a coin, then writing vertical and horizontal marks for heads or tails.

Can you tell which one is really random? Clearly, not (B) since it has too many long runs of horizontal and vertical marks. But hang on—is that true? If we toss a coin 300 times, then at probability $1/4$, $1/8$, $1/16$, $1/32$ we will get a run of the same mark that is three, four, five, or six ticks long (respectively). Now $1/32 \times 300 = 9$ so in (B), we might expect several runs that are at least six ticks long. That is, these “patterns” of long ticks in (B) are actually just random noise.

EXAMPLES FROM SOFTWARE ENGINEERING

Sadly, there are many examples in SE of data scientists uncovering “patterns” which, in retrospect, were more “jumping at shadows” than discovering some underlying causal mechanism. For example, Shull et al. reported one study at NASA’s Software Engineering Laboratory that “discovered” a category of software that seemed inherently most bug prone. The problem with that conclusion was that, while certainly true, it missed an important factor. It turns out that particular subsystem was the one deemed least critical by NASA. Hence, it was standard policy to let newcomers work on that subsystem in order to learn the domain. Since such beginners make more mistakes, then it is hardly surprising that this particular subsystem saw the most errors.

For another example, Kocaguneli et al. had to determine which code files were created by a distributed or centralized development process. This, in turn, meant mapping files to their authors, and then situating some author in a particular building in a particular city and country. After weeks of work they “discovered” that a very small number of people seemed to have produced most of the core changes to certain Microsoft products. Note that if this was the reality of work at Microsoft, it would mean that product quality would be most assured by focusing more on this small core group of programmers.

However, that conclusion was completely wrong. Microsoft is a highly optimized organization that takes full advantage of the benefits of auto-generated code. That generation occurs when software binaries are being built and, at Microsoft, that build process is controlled by a small number of skilled engineers. As a result, most of the files appeared to be “owned” by these build engineers even though these files are built from code provided by a very large number of programmers working across the Microsoft organization. Hence, Kocaguneli had to look elsewhere for methods to improve productivity at Microsoft.

WHAT TO DO

Much has been written on how to avoid spurious and misleading correlations to lead to bogus “discoveries.” Basili and Easterbrook and colleagues advocate a “top-down” approach to data analysis where the collection process is controlled by research questions, and where those questions are defined *before* data collection.

The advantage of “top-down” is that you never ask data “what have you got?”—a question that can lead to the “discovery” of bogus patterns. Instead, you only ask “have you got X?” where “X” was defined before the data was collected.

In practice, there are many issues with top-down, not the least of which is that in SE data analytics, we are often processing data that was collected for some other purpose than our current investigation. And when we cannot control data collection, we often have to ask the open-ended question “what is there?” rather than the top-down question of “is X there?”

In practice, it may be best to mix up top-down with some “look around” inquiries:

- Normally, before we look at the data, there are questions we think are important and issues we want to explore.
- After contact with the data, we might find that other issues are actually more important and that other questions might be more relevant and answerable.

In defense of a little less top-down analysis, I note that many important accidental discoveries might have been overlooked if researchers restricted themselves to just the questions defined before data collection. Here is a list of discoveries, all made by researchers pursuing other goals:

- North America (by Columbus)
- Penicillin
- Radiation from the big bang
- Cardiac pacemakers (the first pacemaker was a badly built cardiac monitor)
- X-ray photography
- Insulin
- Microwave ovens
- Velcro
- Teflon
- Vulcanized rubber
- Viagra

IN SUMMARY: WAIT AND REFLECT BEFORE YOU REPORT

My message is *not* that data miners are useless algorithms that torture data until they surrender some spurious conclusion. By asking open-ended “what can you see?” questions, our data miners can find unexpected novel patterns that are actually true and useful—even if those patterns fly in the face of accepted wisdom. For example, Schmidt and Lipson’s Eureqa machine can learn models that make no sense (with respect to current theories of biology), yet can make accurate predictions on complex phenomena (eg, ion exchanges between living cells).

But, while data miners can actually produce useful models, sometimes they make mistakes. So, my advice is:

- Always, always, always, wait a few days.
- Most definitely, *do not* confuse business users with such recent raw results.

In summary, do not rush to report the conclusions that you just uncovered, just this morning. For example, in the case of the Kocaguneli et al. study, if a little more time had been taken reading the raw data, then they would have found the files written by the core group all had funny auto-generated names (eg, “S0001.h”). This would have been a big clue that something funny was happening here.

And while you wait, critically and carefully review how you reached that result. See if you can reproduce it using other tools and techniques or, at the very least, implement your analysis a second time using the same tools (just to check if the first result came from some one-letter typo in your scripts).

REFERENCES

- [1] Basili VR. Software modeling and measurement: the goal/question/metric paradigm: Technical report. College Park, MD: University of Maryland; 1992.
- [2] Easterbrook S, Singer J, Storey M-A, Damian D. Selecting empirical methods for software engineering research. In: Guide to advanced empirical software engineering. London: Springer; 2008. p. 285–311.
- [3] Kocaguneli E, Zimmermann T, Bird C, Nagappan N, Menzies T. Distributed development considered harmful? In: Proceedings of the 2013 international conference on software engineering (ICSE ‘13). Piscataway, NJ: IEEE Press; 2013. p. 882–90.
- [4] Norving P. Warning signs in experimental design and interpretation, <http://goo.gl/x0rI2>.
- [5] Schmidt M, Lipson H. Distilling free-form natural laws from experimental data. *Science* 2009;324(5923):81–5.
- [6] Shull F, Mendoncaa MG, Basili V, Carver J, Maldonado JC, Fabbri S, et al. Knowledge-sharing issues in experimental software engineering. *Empir Softw Eng* 2004;9(1–2):111–37.

RULE #4: DATA SCIENCE IS CYCLIC

6

A conclusion is the place where you got tired thinking.
(Martin H. Fischer)

Data mining is inherently a cyclic activity. For example, if you are talking to your users, then any successful data mining analysis will generate conclusions *and* comments on those conclusions that require further investigation. That is, finding one pattern will prompt new questions such as “Why does that effect hold?” or “Are we sure there is no bug in step X of the method?” Each such question refines the goals of the data mining, which leads to another round of the whole data mining process.

The repetitive nature of data science implies that the data mining method is repeated multiple times to either

- Answer an extra user question;
- Make some enhancement and/or bug fix to the method;
- Or deploy the analysis to a different set of users.

This chapter discusses cyclic development in data mining and its implications for how to develop and deploy a data mining application.

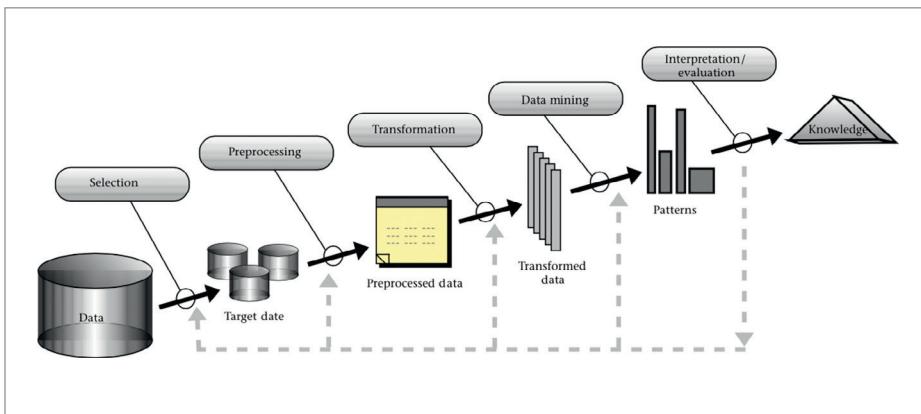
6.1 THE KNOWLEDGE DISCOVERY CYCLE

In 1996, Fayyad et al. [123] noted the cyclic nature of knowledge discovery in data (KDD). [Figure 6.1](#) summarizes their view of KDD. That figure highlights numerous significant aspects of real-world data mining:

- The “data mining” part of KDD is actually a small part of the whole process.
- Even just gaining permission to access data can be a long process requiring extensive interaction with business user groups.
- Once data is accessed, then raw data typically requires extensive manipulation before it is suitable for mining.

The general point here is that, before any learner can execute, much effort must be expended in selecting and accessing the data to process, preprocessing, and transforming it into some learnable form. And after the data mining is done, further effort is required to investigate and understand the results. That is,

Most of “data mining” is actually “data pre/postprocessing.”

**FIGURE 6.1**

The Fayyad KDD cycle. From [123].

Note the dashed feedback arrows, shown in gray in [Figure 6.1](#). In this feedback, lessons from one step tell how to improve prior work. This feedback lets users refine and mature the goals of the project. Such maturation is useful because, sometimes, the initial goals of the data mining project may not directly match the needs of the users and the content of the data. In fact, we would characterize the cyclic nature of data mining as a search to increase the overlap between the three regions shown in [Figure 6.2](#) that describes:

1. *The questions the users care about.* This can change with feedback when (say) the users realize that their data cannot comment on “A,B,C,” but might be able to comment on another set of related issues “C,D,E.”
2. *The questions the data scientist cares to ask.* For example, data scientists might take the time to learn more about the domain, thus letting them pose more insightful questions.
3. *The answers that can be extracted from the data.* This can change if the data scientists work with the user to collect different kinds of data from the domain.

Novice data scientists do not understand the cyclic nature of data mining. Hence, these novices make too much use of the click-and-point GUIs seen in the data mining toolkits. Such click-and-pointing makes it hard to exactly reproduce complex prior results. Further, such clicking makes it hard to mass produce future results on new data, or small variants of existing data.

For serious studies, to ensure repeatability, the entire analysis should be automated using some high-level scripting language. To put that another way:

Thou shall not click.

That is, experts know to eschew a data miner’s GUI and to code the analysis in (say) R script, MATLAB, or Bash [339] or Python.

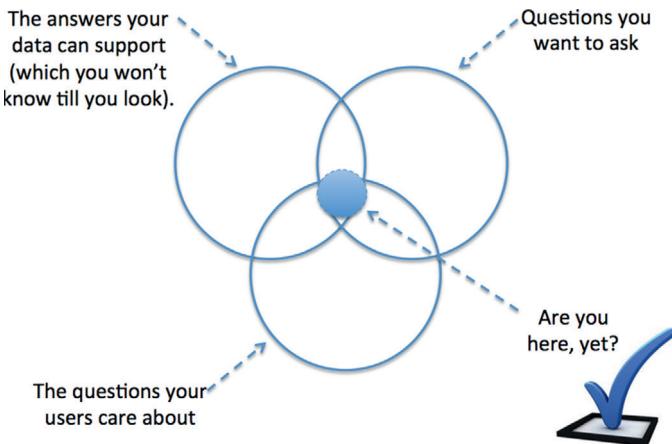


FIGURE 6.2

The questions asked, the answers available, the answers required. Data mining projects need to find the intersection between the questions being asked and the answers supported by the data.

6.2 EVOLVING CYCLIC DEVELOPMENT

In our experience, not only is data science a cyclic process, but *the cycles evolve as the project matures*. For example, consider the CRANE application developed by inductive engineers at Microsoft [86]. CRANE is a risk assessment and test prioritization tool used at Microsoft that alerts developers if the next software check-in might be problematic. CRANE makes its assessments using metrics collected from static analysis of source code, dynamic analysis of tests running on the system, and field data.

CRANE was developed in a cyclic manner, and those cycles changed as the project developed. We characterize that change as follows:

$$\textit{scout} \Rightarrow \textit{survey} \Rightarrow \textit{build}$$

6.2.1 SCOUTING

In the *scout* phase, rapid prototyping is used to try many mining methods on the data. In this phase, experimental rigor is less important than exploring the range of user hypotheses. The other goal of this phase is to gain the interest of the users in the induction results.

It is important to stress that feedback to the users can and must appear very early in a data mining project. Users, we find, find it very hard to express what they want from their data. This is especially true if they have never mined it before. However, once we start showing them results, their requirements rapidly mature as initial results help them sharpen the focus of the inductive study. Therefore, we recommend:

Simplicity first. Prior to conducting very elaborate studies, try applying very simple tools to gain rapid early feedback.

For example, simple linear-time column pruners, such as those discussed in the last chapter, comment on what factors are not influential in a particular domain. It can be insightful to discuss information with the users.

6.2.2 SURVEYING

In the *survey* phase, after securing some interest and goodwill amongst the user population, data scientists conduct careful experiments focusing on the user goals found during the scouting phase.

Of these three phases, the *survey* phase is closest to [Figure 6.1](#).

6.2.3 BUILDING

In the final *build* phase, after the *survey* has found stable models of interest to the users, a systems engineering phase begins. Here, the learned models are integrated into some turnkey product that is suitable for deployment to a very wide user base.

6.2.4 EFFORT

In terms of development effort, the specific details of CRANE's development schedule are proprietary. But to give an approximate idea of the effort required for each phase, we offer the following notes.

For *greenfield applications*, which have not been developed previously:

- The development effort often takes weeks/months/years of work for *scout/survey/build* (respectively).
- The team size doubles and then doubles again after the *scout* and *survey* phases; e.g., one scout, two surveyors, and four builders (assisted by the analysis of the surveyors).

For *product line applications* (where the new effort is some extension to existing work), the above numbers can be greatly reduced when the new effort reuses analysis or tools developed in previous applications.

6.3 SUMMARY

It is unwise to enter into a data mining with fixed hypotheses or approaches, particularly for data that has not been mined before. Organize your work such that you obtain the experience that can change your mind. Deploy your data scientists to explore the domain, before prematurely committing to any particular tools. Plan to repeat a data mining study many times (and, as the project develops, mature those cycles from *scout* to *survey* to *build*). Finally, once you know what is going on, automate your work so you can run the study again and again (perhaps with a slightly different focus for future runs).

Don't embarrass yourself: Beware of bias in your data



C. Bird

Microsoft Research, Redmond, WA, United States

CHAPTER OUTLINE

Dewey Defeats Truman	309
Impact of Bias in Software Engineering	310
Identifying Bias	312
Assessing Impact	313
Which Features Should I Look At?	315
References	315

DEWEY DEFEATS TRUMAN

The 1948 U.S. presidential election proved to be one of the greatest blunders in applied statistics history.

As is often the case, many polls were conducted in the run up to the election. Gallup, still one of the most trusted polling organizations today, predicted that republican Thomas Dewey would handily defeat democrat Harry Truman. In fact, the press was so convinced by the “empirical evidence” that the Chicago Daily Tribune had already printed the first edition of the paper with the headline “Dewey Defeats Truman” before final election results were in (see Fig. 1). Unfortunately for them, the election results the next morning were anything but expected, as Truman had won the electoral vote with 303 votes to Dewey’s 189. This was a landslide, of course, but in the opposite direction.

In the modern era of data collection and statistics, how could such a thing have happened? The answer lies not in the analysis of the data, but in the hidden biases it contained. Consider just one of many errors in the polling methodology. Like today, polling was conducted by selecting people randomly and contacting people via telephone. However, in 1948, telephones were mostly owned by individuals who were more financially well-off. At that time, those with higher income levels tended to lean republican. While the polling was indeed random, the population sampled (people that had telephones) was biased with respect to the entire voting population. Thus, any results drawn from the polling data were similarly biased. The problem

**FIG. 1**

The press was so sure that Dewey would win based on telephone polling that they printed results in the newspaper before official vote counts were completed!

has not completely been solved even today as certain demographics may be more likely to answer the phone or less likely to vote.

This is an interesting cautionary tale, but surely such a mistake couldn't happen in the 21st century on data drawn from software engineering ...

IMPACT OF BIAS IN SOFTWARE ENGINEERING

A *biased* sample is a sample that is collected in such a way that some members of the intended population are less likely to be included than others. In layman's terms, a sample of data has bias if some characteristic in the population being sampled is significantly over or under-represented. Bias is usually a result of how the data is created, collected, manipulated, analyzed, or presented. As a trivial example, if you were measuring the height of students at a university and used the basketball team as your sample, the data would be biased and inaccurate because short students are much less likely to make the team. Any analysis and results arising from this data would also be biased, and most likely invalid. If bias is not recognized and accounted for, results can be erroneously attributed to the phenomenon under study rather than to the method of sampling.

Unfortunately, bias exists in software engineering data as well. If left unchecked and undetected, such bias in data can lead to misleading conclusions and incorrect predictions.

A few years ago [1], we examined defect data sets to determine if there was bias in the “links” between a defect in the defect database and the corresponding defect correcting change in the source code repository. Knowing which code changes fix which defects can be quite valuable because this can provide much more context

about a code change, and also allows us to determine which prior code changes actually introduced a bug. It also allows us to see who is introducing the defects, who is correcting the defects, and what types of defects are corrected in different parts of the code base. Research has shown that this information can be used to learn characteristics of code changes that lead to defects or to teach machine learning methods to accurately predict where in the source code a defect is based only on the bug report. Because of the value of these “links,” a long line of research exists on techniques to infer these links. See Sliwerski et al. for one of the most well-known examples [2]. In our study of these links in five software projects, we found that there was bias in the severity level of defects that could be linked to defects in four of the projects. That is, the lower the severity level for a fixed bug, the higher the likelihood that there was a link between the defect and the commit. As an extreme example, out of all defects labeled “minor” in the Apache Webserver that were indicated in the defect database to have been fixed, we were able to identify the corresponding fixing commit for 65% of them. In contrast, for those bugs in the category of “blocker” that were fixes, we were only able to find the fixing commit 15% of the time.

Fig. 2 shows the proportions for all projects. Note that AspectJ appears to suffer far less from bias in bug severity for links between defects and commits.

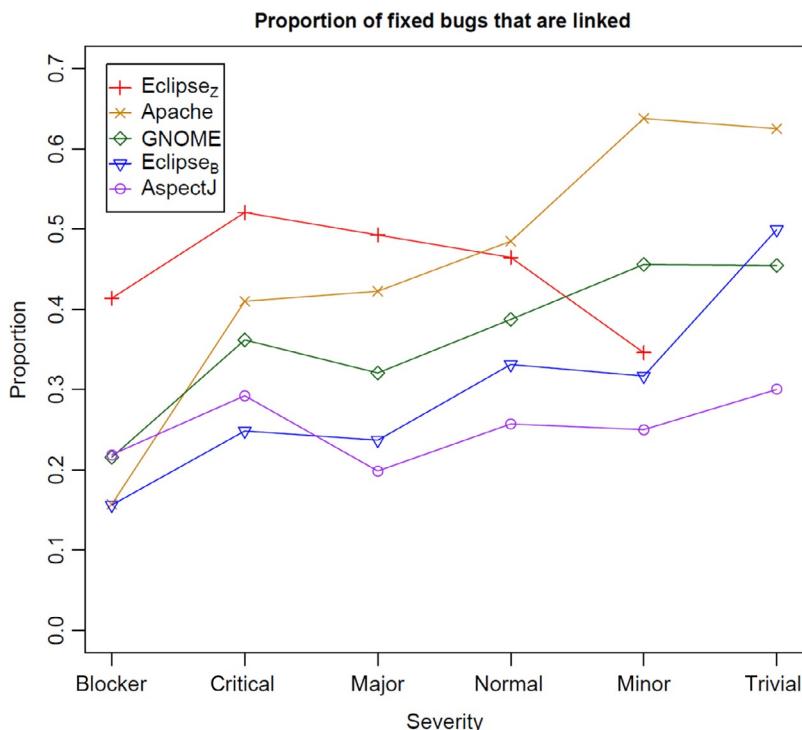


FIG. 2

Proportion of fixed bugs that are linked to corresponding source code changes ordered by severity for five bug data sets.

While we identified bias in the data, what's worse is that this bias appeared to affect the results of research that used the data. We used the linked defects and fixing commits to train a defect prediction model (a statistical model that would predict what parts of the code were most likely to contain defects). When we evaluated our prediction model, it was much better at predicting those defects that had lower severity than those that had higher severity. In practice, one would likely prefer a machine learning method that points to likely defective areas in the code to either be agnostic of the severity of the defects, or favor indicating locations with higher severity defects. We were getting the opposite due to bias in the data. Inadvertently using biased data can impact the quality of tools or models and the validity of empirical findings.

IDENTIFYING BIAS

The first step in avoiding the use of biased data is in determining if bias exists in the first place. This can be done via visualization or statistics, but often requires some *a priori* knowledge or expectations about the data as well. To do this, you first decide what *feature* you are interested in examining. A feature is any individual measurable property of the data or phenomenon being observed. Concretely, features of defects in a defect tracking system can include (but are certainly not limited to) the severity, how long it took for a defect to be fixed, who fixed the defect, and the textual summary of the defect. Features of a source code commit could include the number of lines in a change, the person that made the change, the type of file or files being changed, and whether a comment was added to the code.

In the best case scenario, you may have information about the distribution of an important feature in the population. In the study of defects, we had the severity levels for all fixed defects in a project. This forms our *population* distribution. We then compared that to the severity levels for fixed defects that we could find the corresponding commits for, our *sample* distribution. Generating histograms for the population and sample severity distributions is relatively easy to do in R or Excel, and if there is bias, it is often easy to spot them visually using such visualizations. One nice aspect of using histograms is that they work for both categorical *and* numerical data.

Statistically, one can compare distributions of categorical data using a Pearson's chi-squared test or a Fisher's exact test. Both are readily available in any statistical environment such as R, SPSS, or SAS. For numerical data, the Kolmogorov-Smirnov test (also called the K-S test) can tell you if there is a statistically significant difference between the two populations. Note that in all of these cases, the result of the test is a likelihood that the two distributions come from the same population (a so-called *p*-value). The tests *do not* indicate how the distributions differ (eg, which way a sample is biased or which category is over-represented). It is up to you to investigate further and determine what exactly the bias is and how extreme it is.

While statistics can help, it is important that you understand the data being collected and that you know what you expect to see in the data and why. In our study, we examined the way the development happened and that defects were fixed, and based

on our knowledge of the projects, we believed that each defect that was fixed should have the same likelihood of being linked to a commit as any other. However, consider a software project where minor bugs are assigned to junior developers and their fixes must be reviewed by senior developers. In this case, it may make more sense that minor bugs would be explicitly linked to their commits (for review), and so our expectations would be that lower severity defects have a higher link rate. Understanding your data and the processes that it came from can aid greatly when examining your data visually or statistically for bias.

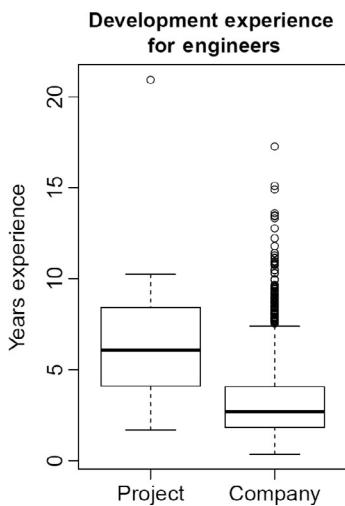
Unfortunately, you may not always have in-depth knowledge about some characteristic of your data. In the absence of information about the population, distributions from samples expected to be similar could be used. For instance, if you are examining the number of developers actively contributing to Python over the past year, you could compare the distribution of active developers this year to previous years for Python. You might also compare it to other projects that you consider to be similar to Python in some way, such as Perl or Ruby. For these types of comparisons, statistical tests are unlikely to provide much value, as the distributions will likely be different to some degree, and that may not indicate real bias. A visual inspection will help you determine if they are different enough to warrant further investigation.

If you lack any other distribution for comparison, the best approach is to calculate descriptive statistics from your sample, visualize the sample via histograms, PDFs, or boxplots, and make a judgement as to how well the distribution of a feature matches your expectations. If they differ widely, then either the data or your expectations are incorrect. In both cases, you should likely “dig in” and do some more manual investigation into where the data came from, how it was produced, and anything that may be out of the ordinary.

As a concrete example, consider an investigation into the impact of years of experience on the time to complete tasks in a software company. As part of this study, one would need to select one or more projects to investigate. Suppose that after selecting a project and gathering data, such as the experience of developers and the time taken to complete various tasks, the investigator wants to determine if there is any bias (and if so how much) in his data. One step would be to collect the years of experience from developers in the entire company (or from a purely random sample of developers). Putting the data into R and drawing a simple boxplot (shown in Fig. 3) will quickly show that the project selected is highly biased with respect to age of developers, and thus the findings may not generalize to the entire company. A Kolmogorov-Smirnov test in R also indicates that the sample is statistically different from the population of developers in the company.

ASSESSING IMPACT

Just because bias exists in a data set does not mean that the bias will have an impact on the results of using the data. In our study, we found that when defects used to train a model were biased with respect to severity, the predictions from the model were

**FIG. 3**

Years of experience for all developers in the company compared to developers just in the project selected for a study.

also biased in a similar way. However, consider a defect model trained on defects that were fixed mostly (though not completely) on even days of the month (eg, Jan. 2, Oct. 24, etc.). While the data is biased with regard to the parity of the fix day, it is unlikely that such a model would do much better when evaluated on defects fixed on even days than on defects fixed on odd days.

How can we assess the impact of bias?

If we had access to all defects for all days, that would help. We could train one model on the biased sample and another on the larger, less-biased sample and look at the difference to assess the impact of the bias. However, usually if we have a biased sample, we don't have access to a larger, less-biased sample. In the absence of this less-biased sample, one approach is to select subsets of your sample such that they are biased in different ways. In the preceding example, we could remove all of the odd days so that the model is *only* trained on defects fixed on even days. Does the performance of this second model differ from the original model? What about training the model only on days that are multiples of four or ten? These are “super-biased” data sets. We could go the other way and create a subset from our sample that has the same number of defects fixed on odd and even days. Does a model trained on this data set perform differently? If we see (as I suspect we would), that the amount of “day parity” bias does not affect model results, then we may not need to worry about the bias. If in your investigations, you find that there *is* a feature (such as age of a developer, size of a commit, or date of a defect report) that is biased and that does effect the results of a study, accuracy of a model, or utility of a technique, you are not completely out of luck. This just means that the bias needs to be reported so that others consuming your research have all the salient facts they need.

WHICH FEATURES SHOULD I LOOK AT?

Having said all of this, an additional key question to ask is what features to examine for bias. Data collected from software repositories have nearly endless dimensions (features). A developer working on a project has an associated age, gender, experience level, education, location, employment history, marital status, etc. A code review has an author, a date, the contents of the changed code, the phase of the development cycle it occurs in, the files that are modified, and the number of lines changed. These are just a few of the features that exist for just a few of the artifacts that may be examined as part of a research endeavor. Exhaustively investigating bias for all possible features will take an inordinate amount of time and most of that time will be wasted.

A better approach is to start by reading related research and brainstorming those features that you hypothesize may be related to the outcome of interest. That is, if you are conducting a study related to collaboration of developers, identify those features whose bias you believe is most likely to impact results and validity. Next, identify those features that you can actually measure in your data (sadly, this often a much shorter list). Then rank these features and investigate them as outlined in this chapter. Whether or not you do find bias, be sure to report your investigation and the outcome in any publication, most often in a “Threats to Validity” or “Evaluation” section.

REFERENCES

- [1] Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu D. Fair and balanced? Bias in bug-fix datasets. In: Proceedings of the ACM SIGSOFT foundations of software engineering. New York: ACM; 2009.
- [2] Sliwerski J, Zimmermann T, Zeller A. When do changes induce fixes? On Fridays. In: Proceedings of the international workshop on mining software repositories; 2005.
- [3] Friedenson B. Dewey defeats Truman and cancer statistics. J Natl Cancer Inst 2009; 101(16):1157.

On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension

Leon Moonen¹, Arie van Deursen^{1,2}, Andy Zaidman¹, and Magiel Bruntink^{2,1}

¹ Delft University of Technology, The Netherlands

² CWI, The Netherlands

Summary. We know software evolution to be inevitable if the system is to survive in the long-term. Equally well-understood is the necessity of having a good test suite available in order to (1) ensure the quality of the current state of the software system and (2) to ease future change. In that light, this chapter explores the interplay that exists between software testing and software evolution, because as tests ease software evolution by offering a safety net against unwanted change, they can equally be experienced as a burden because they are subject to the very same forces of software evolution themselves.

In particular, in this chapter, we describe how typical refactorings of production code can invalidate tests, how test code can (structurally) be improved by applying specialized test refactorings. Building upon these concepts, we introduce “test-driven refactoring”, or refactorings of production code that are induced by the (re)structuring of the tests. We also report on typical source code design metrics that can serve as indicators for testability. To conclude, we present a research agenda that contains pointers to—as yet—unexplored research topics in the domain of testing.

8.1 Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less satisfactory [317, 321]. We also know that due to ever changing surroundings, new business needs, new regulations and also due to the people working with the system, the software is in a semi-permanent state of flux [319]. Combined with the increasing life-span of most software systems [56], this leads to a situation where an ever higher fraction of the total budget of a software system is spent during the maintenance or evolution phase of a software system, considerably outweighing the initial development costs of a system [329].

For many people, evolving a software system has become a synonym for adapting the source code as this concept stands central when thinking of software. Software, however, is multidimensional, and so is the development process behind it. This multidimensionality lies in the fact that to develop high-quality source code, other artifacts are needed. Examples of these are: specifications, which are needed

to know what should be developed, constraints, which are defined so that the software has to adhere to them, documentation, which needs to be written to ease future evolution, and tests, which need to be set up and exercised to ensure quality [436]. The central question then is how evolution should happen: in a unidimensional way, where only the source code is changed, or in a multidimensional way, where (all) the other artifacts are also evolved?

Within this chapter we will explore two dimensions of the multidimensional software evolution space, as we will focus on how the production software evolves with regard to the accompanying tests of the software system. To characterize why tests are so important during evolution, we first discuss some general focal points of tests:

Quality assurance Tests are typically engineered and run to ensure the quality of a software system [131]. Other facets that are frequently tested are the robustness and stress-resistance of a software system.

Documentation In *agile software development* methods such as *extreme programming* (XP), tests are explicitly used as a form of documentation, and as such, the tests serve as a means of communication between developers [516, 149].

Confidence At a more psychological level, test code can help the software (re-) engineer become more confident, because of the safety net that is provided by the tests. Furthermore, the confidence within the development team can be improved when they see that the system they are trying to deliver, is working correctly [119, 149].

An aspect of testing that cannot be neglected is the impact on the software development process: testing is known to be very time-intensive, thus driving up the total costs of the software system. Estimates by Brooks put the total time devoted to testing at 50% of the total allocated time [85, 447], while Kung et al. suggest that 40 to 80% of the development costs of building software is spent in the testing phase [301].

Several types of testing activities can be distinguished. The focus of this chapter is on *developer testing* (often also called *unit testing*), i.e., testing as conducted by the development team in order to assess that the system that is being built is working properly. In some cases, such tests will be set up with knowledge of the inner workings of the system (*white box testing*)—in others the test case will be based on component requirements, (design) models or public interfaces (*black box testing*) [66, 346].

One of the alternatives to developer testing is *acceptance testing*, i.e., testing as conducted by end user representatives in order to determine whether the system meets the stated criteria. Although acceptance testing is not the primary focus of this chapter, it has many techniques in common with developer testing (as observed by Binder [66]), which is why we believe that the results that we discuss will to a large extent be valid for acceptance testing as well.

Having discussed the necessity of a software system's evolution and also the importance of having a test suite available for a system, we can turn our attention to the interactions that occur between tests and the system under evolution. To this end, we define a number of research questions that we will investigate in the remainder of this chapter:

1. How does a system’s test suite influence the program comprehension process of a software engineer trying to understand a given system? What are the possible side effects with regard to evolving the software system?
2. Are there typical code characteristics that indicate which test code resists evolution? And if so, how can we help alleviate these, so called, *test smells*?
3. Given that production code evolves through e.g. refactorings—behavior preserving changes—, what is the influence of these refactorings on the associated test code? Does that test code need to be refactored as well or can it remain in place unadapted? And what will happen to its role as safety net against errors?
4. Can we use metrics to understand the relation between test code and production code? In particular, can object-oriented metrics on the production code be used to predict key properties of the test code?

In order to find answers to the above questions, we have studied how the test suites of a number of applications evolve through time. We have specifically looked at software developed using agile software development methods since these methods explicitly include a number of evolutionary steps in their development process. Furthermore, such projects typically make use of testing frameworks, such as JUnit [49, 262]. To sketch this context, we give a short introduction to agile methods in Section 8.2.

The four research questions introduced above, are discussed in Sections 8.3 through 8.6: we investigate the effects of test suites on comprehension in Section 8.3. We present a catalogue of test smells and test refactorings in Section 8.4. In Section 8.5 we make a classification of classical refactorings [183] into categories, so that one can easily see which refactorings (possibly) break a test. Finally, we discuss a study that shows how certain object-oriented metrics correlate to testing effort in Section 8.6.

In our concluding remarks (Section 8.7) we present a retrospective and touch upon a number of unexplored research tracks.

8.2 Agile Software Development Methods

Agile software development methods (or Agile methods in short) refer to a collection of “lightweight” software development methodologies that adhere to the ideas in the Agile Manifesto [233]. Agile methods aim at minimizing risk and achieving customer satisfaction through a short (development) feedback loop.

Agile methods recognize that continuous change of software systems is natural, inevitable and actually a desirable aspect of successful software systems. Agile software development is typically done in short iterations, lasting only a few weeks. Each iteration includes all software engineering activities, such as planning, design, coding, and testing, that are needed to add a (small) piece of functionality to the system. Agile methods aim at having a working product (albeit not functionally complete) deliverable to the customer after each iteration.

Agile software development builds upon various existing and common sense practices and principles, such as code reviewing, testing, designing and refactoring. However, these practices are done *continuously* rather than at dedicated phases of

the software process only. On the other hand, the need for extensive documentation on an agile project is reduced by several of its practices: test-driven development and a focus on acceptance testing ensures that there is always a test suite that shows that your system works and fulfills the requirements implemented to that point. For the developers, these tests act as significant documentation because it shows how the code actually works [9], and how it should be invoked.

A particular agile method that is studied in more detail in this chapter is *Extreme Programming* (XP). XP is one of the initial and most prominent of the agile methods and applies many of the agile practices to “extreme levels”. It is a lightweight methodology for small teams of approximately 10 people developing software in the face of vague or rapidly changing requirements [50]. XP is performed in short *iterations*, which are grouped into larger *releases*. The planning process is depicted as a game in which business and development determine the scope of releases and iterations. The customer describes features via *user stories*, informal use cases that fit on an index card. The developers estimate each of the user stories. User stories are the starting point for the planning, design, implementation, and acceptance test activities conducted in XP.

Two key practices of XP play an important role within the scope of our study, namely testing and refactoring. In XP (and most other agile methods) *tests* are written in parallel with (or even *before*) the production code by the programmers. The tests are collected and they must all pass at any time. Customers write acceptance tests for the stories in an iteration, if needed supported by the development team. Tests are typically fully automatic, making it cheap to run them frequently. To write tests, testing frameworks such as JUnit [49] are used (see the next section).

The second key practice of interest is *refactoring*: improving the design of existing code without changing functionality. The guiding design principle is “do the simplest thing that could possibly work”. In XP, continuous refactoring during coding replaces the traditional (big) up front design effort.

Note that although this chapter uses agile software development methods and XP to discuss the interaction between software evolution and software testing, this does not mean that the issues observed only apply to agile methods; they are just as likely to come up in any other development process where developer testing and refactoring plays an important role. We choose agile methods as showcase because of its explicit focus on testing and inclusion of evolutionary steps in the development cycle.

8.3 Program Comprehension

A major cost factor in the life cycle of a software system is *program understanding*: trying to understand an existing software system for the purpose of planning, designing, implementing, and testing changes. Estimates put the total cost of the understanding phase at 50% of the total effort [125]. This suggests that paying attention to program comprehension issues in the software process could well pay off in terms of higher quality, longer life time, fewer defects, lower costs, and higher job satisfaction.

This is especially true in the case of extreme programming since the need for people to understand pieces of code is at the very core of XP.

Based upon a thorough analysis of (1) literature on XP [50, 254, 48]; (2) online discussion covering XP subjects³; and (3) our own experiences made during an ongoing (industrial) extreme programming project⁴, we made the following observation:

Observation 1 An extensive test suite can stimulate the program comprehension process, especially in the light of continuously evolving software.

For our study, we specifically focus on how program comprehension and unit tests interact in the XP software process. We analyze risks and opportunities, look at the effect on the team (whether and how the team gets a better understanding of the code) as well as on the source code (whether and how the code gets more understandable).

8.3.1 Program Understanding

We define program understanding (comprehension) as *the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution, and re-engineering purposes* [383].

An important research area in program understanding deals with the cognitive processes involved in constructing a mental model of the software system (see, e.g., [530]). A common element of such cognitive models is generating hypotheses about code and investigating whether they hold or must be rejected. Several *strategies* can be used to arrive at relevant hypotheses, such as bottom up (starting from code), top down (starting from a high-level goal and expectations), and opportunistic combinations of the two [125]. Strategies guide two understanding mechanisms that produce information: *chunking* creates new, higher level abstraction structures from lower level structures, and *cross referencing* relates different abstraction levels [530]. We will see how the XP practices relate to these program understanding theories.

The construction of mental models at different levels of abstraction can be supported by so called software exploration tools [378]. These tools use reverse engineering techniques to (1) identify a system's components and interrelationships; and (2) create representations of a system in other forms or at higher levels of abstraction [112].

8.3.2 Unit Testing and XP

Unit testing is at the heart of XP. Unit tests are written by the developers, using the same programming language used to build the system itself. Tests are small, take a white box view on the code, and include a check on the correctness of the

³ Most notably, the *C2 wiki* at <http://www.c2.com/cgi/wiki> and <http://groups.yahoo.com/group/extremeprogramming/>. Last visited January, 2007.

⁴ Program understanding tools by the *Software Improvement Group*: <http://www.software-improvers.com/>.

results obtained, comparing actual results with expected ones. Tests are an explicit part of the code, they are put under revision control, and all tests are shared by the development team (any one can invoke any test). A unit test is required to run in almost zero time. This makes it possible (and recommended) to run all tests before and after any change, however minor the change may be.

Testing is typically done using a testing framework such as *JUnit* developed by Beck and Gamma [49, 262]. The framework caters for invoking all test methods of a test class automatically, and for collecting test cases into test suites. Test results can be checked by invoking any of the *assert* methods of the framework with which expected values can be compared to actual values. Testing success is visualized through a graphical user interface showing a growing green bar as the tests progress: as soon as a test fails, the bar becomes red.

The XP process encourages writing a test class for every class in the system. The test code/production code ratio may vary from project to project and in practice we have seen ratios as high as 1:1. Moreover, XP encourages programmers to use tests for documentation purposes, in particular if an interface or method is unclear, if the implementation of a method is complicated, if there are circumstances in which the code should work in a special way, and if a bug report is received [50]. In each of these situations, the test is written *before* the corresponding method is written (or modified) [52].

Also, tests can be added while understanding existing code. In particular, whenever a programmer is tempted to type something into a print statement or debugger instruction, XP advises to write a test instead and add it to the system's test suite [49].

8.3.3 Comprehension Benefits

This section discusses a number of benefits that an automated unit testing regime has for program comprehension.

First, *XP's testing policy encourages programmers to explain their code using test cases*. Rather than explaining the behavior of a function using prose in comments or documentation, the extreme programmer adds a test that explicitly shows the behavior.

Second, *the requirement that all tests must run 100% at all times, ensures that the documentation via unit tests is kept up-to-date*. With regular technical documentation and comments, nothing is more difficult than keeping them consistent with the source code. In XP, all tests must pass before and after every change, ensuring that what the developer writing the tests intended to communicate remains valid.

Third, *adding unit tests provides a repeatable program comprehension strategy*. If a programmer needs to change a piece of code that he is not familiar with, he will try to understand the code by inspecting the test cases. If these do not provide enough understanding, the programmer will try to understand the nature of the code by developing and testing a series of hypotheses, as we have seen in Section 8.3.1. The advise to write tests instead of using print statements or debugger commands applies here as well: *program understanding hypotheses can be translated into unit tests*, which then can be run in order to confirm or refute the hypotheses.

Fourth, *a comprehensive set of unit tests reduces the comprehension space when modifying source code*. To a certain extent a programmer can just try a change and see whether the tests still run. This reduces the risks and complexity of conducting a painstakingly difficult impact analysis. Thus, the XP process attempts to minimize the *size* of the mental model that needs to be build and maintained since the tests help the programmer to see what parts are not affected by the current modifications.

Last but not least, *systematic unit testing helps build team confidence*. In the XP literature, it is said that the tests help the team to develop *courage* to change the code [344].

The XP testing process not only affects the way the team works, it also has a direct effect on the understandability of the production code written [254, p.199]. *Writing unit tests requires that the code tested is split into many small methods each responsible for a clear and testable task*.

In addition, if the tests are written after the production code, it is likely that the production code is difficult to test. For that reason, XP requires that the unit tests are written *before* the code (the “test-driven” approach) [52]. In this way, *testing code and production code are written hand-in-hand, ensuring that the production code is set up in a testable manner*.

8.3.4 Comprehension Risks

Using tests for documentation leads to the somewhat paradoxical situation that in order to understand a given piece of code a programmer has to read another piece of code. Thus, to support program comprehension, XP increases the code base and this code needs to be maintained as well. We experienced that maintaining such test code requires special skills and refactorings, which we describe in Section 8.5.

Also of importance is that tests are automated (with the possible exception of exploratory tests), as non-automated tests probably require knowledge or skill to activate the tests. Knowledge which is possibly not available during (initial) program comprehension [131].

Another concern is that XP uses the tests (in combination with oral communication and code written to display intent) as a *replacement* for technical documentation. The word “documentation” is mentioned once in Beck’s book, where he explains why he decided *not* to write documentation [50, p. 156]. For addressing subjects not easily expressed in the tests or code of the system under development, a *technical memorandum* can be written [134]. These are short (one or two pages) papers expressing key ideas and motivations of the design. However, if the general tendency is not to write documentation, it is unlikely that the technical memoranda actually get written, leaving important decisions undocumented.

A final concern is that some types of code are inherently hard to test, the best known examples being user interfaces and database code. Writing tests for such code requires skill, experience, and determination. This will not be always available, leaving the hardest code without tests and thus without documentation.

A possible solution for these cases can be the use of so called *mock objects* which are “simulated” objects that can mimic the behavior of complex objects in a con-

trolled way (often using a form of capture and replay) [336]. Setting up such a mock object can then serve as documentation of the interaction with the real object.

8.4 Test Smells and Refactorings

Continuous refactoring, one of the key practices of extreme programming and most other agile methods, is advocated for bringing the code into the simplest state possible. To aid in the refactoring process a catalog of “code smells” and a wide range of refactorings is available, varying from simple modifications up to ways to systematically introduce design patterns in existing code [273].

From our own experiences we know however that test code is different from production code and this has led us to the following observations:

Observation 2 Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other.

Observation 3 Improving test code involves a mixture of applying refactorings as identified by Fowler [183] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes and the way of grouping test cases.

In this section we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification.

For the remainder of this chapter, we assume some familiarity with the *xUnit* framework [49] and refactorings as described by Fowler [183]. We will refer to refactorings described in this book using *Name (F:page#)* and to our test specific refactorings described in Section 8.4.2 using *Name (#)*.

8.4.1 Test Code Smells

This section gives an overview of bad code smells that are specific for test code.

Smell 1: Mystery Guest.

When a test uses external resources, such as a file containing test data, the test is no longer self contained. Consequently, there is not enough information to understand the tested functionality, making it hard to use that test as documentation.

Moreover, using external resources introduces hidden dependencies: if some force changes or deletes such a resource, tests start failing. Chances for this increase when more tests use the same resource.

The use of external resources can be eliminated using the refactoring *Inline Resource* (1). If external resources are needed, you can apply *Setup External Resource* (2) to remove hidden dependencies.

Smell 2: Resource Optimism.

Test code that makes optimistic assumptions about the existence (or absence) and state of external resources (such as particular directories or database tables) can cause non-deterministic behavior in test outcomes. Situations where tests run fine at one time and fail miserably the next time are not where you want to find yourself in. Use *Setup External Resource* (2) to allocate and/or initialize all resources that are used.

Smell 3: Test Run War.

Such wars arise when the tests run fine as long as you are the only one testing but fail when more programmers run them. This is most likely caused by resource interference: some tests in your suite allocate resources such as temporary files that are also used by others. Apply *Make Resource Unique* (3) to overcome interference.

Smell 4: General Fixture.

In the *JUnit* framework a programmer can write a *setUp* method that will be executed before each test method to create a fixture for the tests to run in.

Things start to smell when the *setUp* fixture is too general and different tests only access part of the fixture. Such set-ups are harder to read and understand and may make tests run more slowly (because they do unnecessary work). The danger of having tests that take too much time to complete is that testing starts interfering with the rest of the programming process and programmers eventually may not run the tests at all.

The solution is to use *setUp* only for that part of the fixture that is shared by all tests using Fowler's *Extract Method* (F:110) and put the rest of the fixture in the method that uses it using *Inline Method* (F:117). If, for example, two different groups of tests require different fixtures, consider setting these up in separate methods that are explicitly invoked for each test, or spin off two separate test classes using *Extract Class* (F:149).

Smell 5: Eager Test.

When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.

The solution is simple: separate the test code into test methods that test only one method using Fowler's *Extract Method* (F:110), using a meaningful name highlighting the purpose of the test. Note that splitting into smaller methods can slow down the tests due to increased setup/teardown overhead.

Smell 6: Lazy Test.

This occurs when several test methods check the same method *using the same fixture* (but for example check the values of different instance variables). Such tests often only have meaning when considering them together so they are easier to use when joined using *Inline Method* (F:117).

Smell 7: Assertion Roulette.

You know *something* is wrong because your tests fail but it is unclear *what*. This smell comes from having a number of assertions in a single test method that do not

have a distinct explanation. If one of the assertions fails, you do not know which one it is. Use *Add Assertion Explanation* (5) to remove this smell.

Smell 8: Indirect Testing.

A test class is supposed to test its counterpart in the production code. It starts to smell when a test class contains methods that actually perform tests on other objects (for example because there are references to them in the class-to-be-tested). Such indirection can be moved to the appropriate test class by applying *Extract Method* (F:110) followed by *Move Method* (F:142) on that part of the test. The fact that this smell arises also indicates that there might be problems with data hiding in the production code.

Note that opinions differ on indirect testing. Some people do not consider it a smell but a way to guard tests against changes in the “lower” classes. We feel that there are more losses than gains to this approach: it is much harder to test anything that can break in an object from a higher level and understanding and debugging indirect tests is much harder.

Smell 9: For Testers Only.

When a production class contains methods that are only used by test methods, these methods either (1) are not needed and can be removed, or (2) are only needed to set up a fixture for testing. Depending on functionality of those methods, you may not want them in production code where others can use them. If this is the case, apply *Extract Subclass* (F:330) to move these methods in the testcode and use that subclass to perform the tests on. You will often find that these methods have names or comments stressing that they should only be used for testing.

Fear of this smell may lead to another undesirable situation: a class without corresponding test class. The reason then is that the developer (1) does not know how to test the class without adding methods that are specifically needed for the test and (2) does not want to pollute his production class with test code. Creating a separate subclass helps to deal with this problem.

Smell 10: Sensitive Equality.

It is fast and easy to write equality checks using the `toString` method. A typical way is to compute an actual result, map it to a string, which is then compared to a string literal representing the expected value. Such tests, however may depend on many irrelevant details such as commas, quotes, spaces, etc. Whenever the `toString` method for an object is changed, tests start failing. The solution is to replace `toString` equality checks by real equality checks using *Introduce Equality Method* (6).

Smell 11: Test Code Duplication.

Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem. Solutions are similar to those for normal code duplication as described by Fowler [183, p. 76]. The most common case for test code will be duplication of code in the same test class. This can be removed using *Extract Method* (F:110). For duplication across test classes, it may be helpful to mirror the class hierarchy of the production code into the test class hierarchy. A word of caution

however: moving duplicated code from two separate classes to a common class can introduce (unwanted) dependencies between tests.

A special case of code duplication is *test implication*: test *A* and *B* cover the same production code, and *A* fails if and only if *B* fails. A typical example occurs when the production code gets refactored: before this refactoring, *A* and *B* covered different code, but afterwards they deal with the same code and it is not necessary anymore to maintain both tests. Because it fails to distinguish between the various cases, test implication impedes comprehension and documentation.

8.4.2 Test Refactorings

Bad smells seem to arise more often in production code than in test code. The main reason for this is that production code is adapted and refactored more frequently, allowing these smells to escape.

One should not, however, underestimate the importance of having fresh test code. Especially when new programmers are added to the team or when complex refactorings need to be performed, clear test code is invaluable. To maintain this freshness, test code also needs to be refactored.

We define *test refactorings* as changes (transformations) of test code that: (1) do not add or remove test cases, and (2) make test code better understandable/readable and/or maintainable [518].

The remainder of this section presents refactorings that we encountered while working on test code. Not all of these refactorings are directly linked with the elimination of the test smells of Section 8.4.1, but when a link is there, it is described.

Refactoring 1: *Inline Resource*.

To remove the dependency between a test method and some external resource, we incorporate that resource in the test code. This is done by setting up a fixture in the test code that holds the same contents as the resource. This fixture is then used instead of the resource to run the test. A simple example of this refactoring is putting the contents of a file that is used into some string in the test code.

If the contents of the resource are large, chances are high that you are also suffering from *Eager Test* (5) smell. Consider conducting *Extract Method* (*F:110*) or *Reduce Data* (4) refactorings.

Refactoring 2: *Setup External Resource*.

If it is necessary for a test to rely on external resources, such as directories, databases, or files, make sure the test that uses them explicitly creates or allocates these resources before testing, and releases them when done (take precautions to ensure the resource is also released when tests fail).

Refactoring 3: *Make Resource Unique*.

A lot of problems originate from the use of overlapping resource names, either between different tests run done by the same user or between simultaneous test runs done by different users.

Such problems can easily be prevented (or repaired) by using unique identifiers for all resources that are allocated, e.g. by including a time-stamp. When you also

include the name of the test responsible for allocating the resource in this identifier, you will have less problems finding tests that do not properly release their resources.

Refactoring 4: Reduce Data.

Minimize the data that is setup in fixtures to the bare essentials. This will have two advantages: (1) it makes them better suitable as documentation, and (2) your tests will be less sensitive to changes.

Refactoring 5: Add Assertion Explanation.

Assertions in the JUnit framework have an optional first argument to give an explanatory message to the user when the assertion fails. Testing becomes much easier when you use this message to distinguish between different assertions that occur in the same test. Maybe this argument should not have been optional...

Refactoring 6: Introduce Equality Method.

If an object structure needs to be checked for equality in tests, add an implementation for the “equals” method for the object’s class. You then can rewrite the tests that use string equality to use this method. If an expected test value is only represented as a string, explicitly construct an object containing the expected value, and use the new equals method to compare it to the actually computed object.

8.4.3 Other Test Smells and Refactorings

Fowler [183] presents a large set of bad smells and refactorings that can be used to remove them. Our work focuses on smells and refactorings that are typical for test code, whereas Fowler focuses more on production code. The role of unit tests in [183] is also more geared towards proving that a refactoring did not break anything than to be used as documentation of the production code.

Instead of focusing on cleaning test code which already has bad smells, Schneider [454] describes how to prevent these smells right from the start by discussing a number of best practices for writing tests with JUnit.

The C2 Wiki contains some discussion on the decay of unit test quality and practice as time proceeds [98], and on the maintenance of broken unit tests [542]. Opinions vary between repairing broken unit tests, deleting them completely, and moving them to another class in order to make them less exposed to changes (which may lead to our *Indirect Testing* (8) smell).

Van Rompaey et al. present an approach in which test smells are detected and then ranked according to their relative significance [521]. For this, they rely on a metric-based heuristic approach. They focus on the “General Fixture” and “Eager Test” test smells (Smell 4 & 5 in Section 8.4.1).

Besides the test smells we described earlier, Meszaros [372] discusses an additional set of process-oriented test smells and their refactorings.

8.5 How Refactoring Can Invalidate Its Safety Net

When evolving a piece of software, the change activities can roughly be divided into two categories. The first category consists of those operations that preserve behavior,

i.e. refactorings, while the second category contains those changes that do not necessarily preserve behavior. Intuitively, when non-behavior-preserving changes are applied to production code, one would expect that the associated test code would need to evolve as well, as the end-result of the computation is bound to be different.

When thinking of refactorings of production code however, the picture is not that clear whether the associated unit tests need to evolve as well. Refactoring, which aims to improve the internal structure of the code, happens e.g. through the removal of duplication, simplification, making code easier to understand, adding flexibility, ... Fowler describes it as: “*Without refactoring, the design of software will decay. Regular refactoring helps code retain its shape.*” [183, p.55].

One of the dangers of refactoring is that a programmer unintentionally changes the system’s behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice, however, we have noticed that there are refactorings that will invalidate tests, as tests often rely, to a certain extent, on the code structure, which may have been affected by the refactoring (e.g., when a method is moved to another class and the test still expects it in the original class).

From this perspective, we observed the following:

Observation 4 The refactorings as proposed by Fowler [183] can be classified based on the type of change they make to the code, and therefore on the possible change they require in the test code.

Observation 5 In parallel to test-driven design, *test-driven refactoring* can improve the design of production code by focusing on the desired way of organizing test code to drive refactoring of production code (i.e., refactor for testing).

To explore the relationship between unit testing and refactorings, we take the following path: we first set up a classification of the refactorings described by Fowler [183], identifying exactly which of the refactorings affect class interfaces, and which therefore require changes in the test code as well (see Section 8.5.1). Subsequently, we look at the video store example from [183], and assess the implications of each refactoring on the test code (Section 8.5.2). We explore *test-driven refactoring*, which analyzes the test code in order to arrive at code level refactorings (Section 8.5.3), before we discuss the relationship between code-level refactorings and test-level refactorings (Section 8.5.4). We then integrate our results via the notion of a *refactoring session* which is a coherent set of steps resulting in refactoring of both the code and the tests (Section 8.5.5).

8.5.1 Types of Refactoring

Refactoring a system should not change its observable behavior. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring [50, 183].

In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that tests only can pass after the refactoring if they are modified. For example, refactoring can move a method to a new class

while some tests expect it in the original class (in that case, the code will probably not even compile).

This unfortunate behavior was also noted by Fowler: “Something that is disturbing about refactoring is that many of the refactorings do change an interface.” [183, p.64]. Nevertheless, we do not want to change the tests together with a refactoring since that will make them less trustworthy for validating correct behavior afterwards.

In the remainder of this section, we will look in more detail at the refactorings described by Fowler [183] to analyze in which cases problems might arise because the original tests need to be modified.

Taxonomy

If we start with the assumption that refactoring does not change the behavior of the system, then there is only one reason why a refactoring can break a test: *because the refactoring changes the interface that the test expects*. Note that the interface extends to all visible aspects of a class (fields, methods, and exceptions). This implies that one has to be careful with tests that directly inspect the fields of a class since these will more easily change during a refactoring⁵.

So, initially, we distinguish two types of refactoring: refactorings that do not change any interface of the classes in the system and refactorings that do change an interface. The first type of refactorings has no consequences for the tests: since the interfaces are kept the same, tests that succeeded before refactoring will also succeed after refactoring (if the refactoring indeed preserves the tested behavior).

The second type of refactorings can have consequences for the tests since there might be tests that expect the old interface. Again, we can distinguish two situations:

Incompatible: the refactoring destroys the original interface. All tests that rely on the old interface must be adjusted.

Backwards Compatible: the refactoring extends the original interface. In this case the tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the refactoring, we might need to add more tests covering the extensions.

A number of incompatible refactorings that normally would destroy the original interface can be made into backwards compatible refactorings. This is done by extending the refactoring so it will retain the old interface, for example, using the *Adapter* pattern or simply via delegation. As a side-effect, the new interface will already partly be tested. Note that this is common practice when refactoring a published interface to prevent breaking dependent systems. A disadvantage is that a larger interface has to be maintained but when delegation or wrapping was used, that should not be too much work. Furthermore, language features like deprecation can be used to signal that this part of the interface is outdated.

⁵ In fact, direct inspection of fields of a class is a test smell that could better be removed beforehand [518].

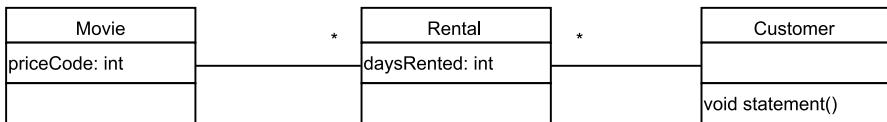


Fig. 8.1. Classes before refactoring

Classification

We have analyzed the refactorings in [183] and divided them into the following classes:

- Composite*: The four big refactorings *Convert Procedural Design to Objects*, *Separate Domain from Presentation*, *Tease Apart Inheritance*, and *Extract Hierarchy* will change the original interface, but we will not consider them in more detail since they are performed as series of smaller refactorings.
- Compatible*: Refactorings that do not change the original interface. Refactorings in this class are listed in Table 8.1.
- Backwards Compatible*: Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. Refactorings in this class are listed in Table 8.2.
- Make Backwards Compatible*: Refactorings that change the original interface and can be made backwards compatible by adapting the new interface to the new one. Refactorings in this class are listed in Table 8.3.
- Incompatible*: Refactorings that change the original interface and are not backwards compatible (for example, because they change the types of classes that are involved). Refactorings in this class are listed in Table 8.4.

Note that the refactorings *Replace Inheritance with Delegation* and *Replace Delegation with Inheritance* are listed both in the *Compatible* and *Backwards Compatible* tables since they can be of either category, depending on the actual case.

8.5.2 Revisiting the Video Store

In this section, we study the relationship between testing and refactoring using a well-known example of refactoring. We revisit the video store code used by Fowler [183, Chapter 1], extending it with an analysis of what should be going on in the accompanying video store test code.

The video store class structure before refactoring is shown in Figure 8.1. It consists of a *Customer*, who is associated with a series of *Rentals*, each consisting of a *Movie* and an integer indicating the number of days the movie was rented. The key functionality is in the *Customer's statement* method printing a customer's total rental cost. Before refactoring, this statement is printed by a single long method. After refactoring, the statement functionality is moved into appropriate classes, resulting in the structure of Figure 8.2 taken from [183, p. 51].

Fowler emphasizes the need to conduct refactorings as a sequence of small steps. At each step, you must run the tests in order to verify that nothing essential has

Table 8.1. Compatible refactorings (type B)

Change Bidirectional Association to Unidirectional	Replace Exception with Test
Replace Nested Conditional with Guard Clauses	Change Reference to Value
Replace Magic Number with Symbolic Constant	Split Temporary Variable
Consolidate Duplicate Conditional Fragments	Decompose Conditional
Replace Conditional with Polymorphism	Introduce Null Object
Replace Inheritance with Delegation	Preserve Whole Object
Replace Delegation with Inheritance	Remove Control Flag
Replace Method with Method Object	Substitute Algorithm
Remove Assignments to Parameters	Introduce Assertion
Replace Data Value with Object	Extract Class
Introduce Explaining Variable	Inline Temp

Table 8.2. Backwards compatible refactorings (type C)

Replace Inheritance with Delegation	Replace Temp with Query	Push Down Method
Replace Delegation with Inheritance	Duplicate Observed Data	Push Down Field
Consolidate Conditional Expression	Self Encapsulate Field	Pull Up Method
Replace Record with Data Class	Form Template Method	Extract Method
Introduce Foreign Method	Extract Superclass	Pull Up Field
Pull Up Constructor Body	Extract Interface	

Table 8.3. Refactorings that can be made backwards compatible (type D)

Change Unidirectional Association to Bidirectional	Remove Middle Man
Replace Parameter with Explicit Methods	Remove Parameter
Replace Parameter with Method	Add Parameter
Separate Query from Modifier	Rename Method
Introduce Parameter Object	Move Method
Parameterize Method	

Table 8.4. Incompatible refactorings (type E)

Replace Constructor with Factory Method	Remove Setting Method
Replace Type Code with State/Strategy	Encapsulate Downcast
Replace Type Code with Subclasses	Collapse Hierarchy
Replace Error Code with Exception	Encapsulate Field
Replace Subclass with Fields	Extract Subclass
Replace Type Code with Class	Hide Delegate
Change Value to Reference	Inline Method
Introduce Local Extension	Inline Class
Replace Array with Object	Hide Method
Encapsulate Collection	Move Field

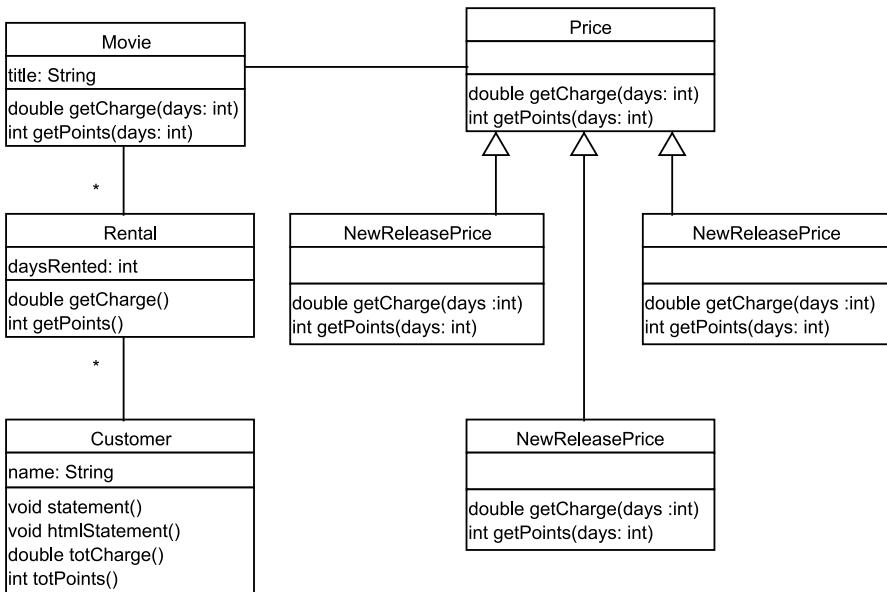


Fig. 8.2. Class structure after refactoring

changed. His testing approach is the following: “I create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. I then do a string comparison between the new string and some reference strings that I have hand checked” [183, p. 8]. Although Fowler does not list his test classes, this typically should look like the code in Figure 8.3.

Studying this string-based testing method, we make the following observations:

- The setup is complicated, involving the creation of many different objects.
- The documentation value of the test is limited: it is hard to relate the computation of the charge of 4.5 for movie *m1* to the way in which charges are computed for the actual movies rented (in this case a children’s and a regular movie, each with their own price computation).
- The tests are brittle. All test cases include a full statement string. When the format changes in just a very small way, all existing tests (!) must be adjusted, an error prone activity we would like to avoid.

Unfortunately, there is no other way to write tests for the given code. The poor structure of the long method necessarily leads to an equally poor structure of the test cases. From a testing perspective, we would like to be able to separate computations from report writing. The long statement method prohibits this: it needs to be refactored in order to be able to improve the testability of the code.

This way of reasoning naturally leads to the application of the *Extract Method* refactoring to the *statement* method. Fowler comes to the same conclusion, based on the need to write a new method printing a statement in HTML format. Thus, we

```

Movie m1 = new Movie("m1", Movie.CHILDRENS);
Movie m2 = new Movie("m2", Movie.REGULAR);
Movie m3 = new Movie("m3", Movie.NEW_RELEASE);
Rental r1 = new Rental(m1, 5);
Rental r2 = new Rental(m2, 7);
Rental r3 = new Rental(m3, 1);
Customer c1 = new Customer("c1");
Customer c2 = new Customer("c2");

public void setUp() {
    c1.addRental(r1);
    c1.addRental(r2);
    c2.addRental(r3);
}

public void testStatement1() {
    String expected =
        "Rental Record for c1\n" +
        "\tm1\t4.5\n" +
        "\tm2\t9.5\n" +
        "Amount owed is 14.0\n" +
        "You earned 2 frequent renter points";
    assertEquals(expected, c1.statement());
}

```

Fig. 8.3. Initial sample test code

extract *getCharge* for computing the charge of a rental, and *getPoints* for computing the “frequent renter points”.

Extract Method is of type C, the *backwards compatible* refactorings, so we can use our existing tests to check the refactoring. However, we have created new methods, for which we might like to add tests that document and verify their specific behavior. To create such tests, we can reuse the setup of movies, rentals, and customers used for testing the *statement* method. We end up with a number of smaller test cases specifically addressing either the charge or rental point computations.

Since the correspondence between test code and actual code is now much clearer and better focused, we can apply white box testing, and use our knowledge of the structure of the code to determine the test cases needed. Thus, we see that the *getCharge* method to be tested distinguishes between 5 cases, and we make sure our tests cover these cases.

This has solved some of the problems. The tests are better understandable, more complete, much shorter, and less brittle. Unfortunately, we still have the complicated setup method. What we see is that the setup mostly involves rentals and movies, while the tests themselves are in the customer testing class. This is because the extracted method is in the wrong class: applying *Move Method* to *Rental* simplifies the set up for new test cases. Again we use our analysis of the test code to find refactorings in the production code.

The *Move Method* is of type D, refactorings that can be made backwards compatible by adding a wrapper method to retain the old interface. We add this wrapper so we can check the refactoring with our original tests. However, since the documentation of the method is in the test, and this documentation should be as close as possible to the method documented, we want to move the tests to the method's new location. Since there is no test class for *Rental* yet, we create it, and move the test methods for *getCharge* to it. Depending on whether the method was part of a published interface, we might want to keep the wrapper (for some time), or remove it together with the original test.

Fowler discusses several other refactorings, moving the charge and point calculations further down to the *Movie* class, replacing conditional logic by polymorphism in order to make it easier to add new movie types, and introducing the *state* design pattern in order to be able to change movie type during the life time of a movie.

When considering the impact on test cases of these remaining video store refactorings, we start to recognize a pattern:

- Studying the test code and the smells contained in it may help to identify refactorings to be applied at the production code;
- Many refactorings involve a change to the structure of the unit tests as well: in order to maintain the documenting value of these unit tests, they should be changed to reflect the structure of the code being tested.

In the next two sections, we take a closer look at these issues.

8.5.3 Test-Driven Refactoring

In *test-driven refactoring*, we try to use the existing test cases in order to determine the code-level refactorings. Thus, we study *test* code in order to find improvements to the *production* code.

This calls for a set of *code smells* that helps to find such refactorings. A first category is the set of existing code smells discussed in Fowler's book [183]. Several of them, such as long method, duplicated code, long parameter list, and so on, apply to test code as well as they do to production code. In many cases solving them involves not just a change on the test code, but first of all a refactoring of the production code.

A second category of smells is the collection of *test smells* discussed in Section 8.4 (also see [518]). In fact, in our movie example we encountered several of them already. Our uneasy feeling with the test case of Figure 8.3 is captured by the *Sensitive Equality* smell [518, Smell 10]: comparing computed values to a string literal representing the expected value. Such tests depend on many irrelevant details, such as commas, quotes, tabs, ... This is exactly why the customer tests of Figure 8.3 become brittle.

Another *test smell* we encountered is called *Indirect Testing* [518, Smell 8]: a test class contains methods that actually perform tests on other objects. Indirect tests make it harder to understand the relationship between test and production code. While moving the *getCharge* and *getPoints* methods in the class hierarchy (using *Move Method*), we also moved the corresponding test cases, in order to avoid *Indirect Testing*.

The test-driven perspective may lead to the formulation of additional test smells. For example, we observed that setting up the fixture for the CustomerTest was complicated. This indicates that the tests are in the wrong class, or that the underlying business logic is not well isolated. Another smell appears when there are many test cases for a single method, indicating that the method is too complex.

Test-driven refactoring is a natural consequence of test-driven design. Test-driven design is a way to get a good design by thinking about test cases first when adding functionality. Test-driven refactoring is a way to improve your design by rethinking the way you structured your tests.

In fact, Beck's work on test-driven design [51, 52] contains an interesting example that can be transferred to the refactoring domain. It involves testing the construction of a mortality table. His first attempt requires a complicated setup, involving separate "person" objects. He then rejects this solution as being overly complex for testing purposes, and proposes the construction of a mortality table with just an age as input. His example illustrates how test case construction guides design when building new code; likewise, test case refactoring guides the improvement of design during refactoring.

8.5.4 Refactoring Test Code

In our study of the video store example, we saw that many refactorings on the code level can be completed by applying a corresponding refactoring on the test case level. For example, to avoid *Indirect Testing*, the refactoring *Move Method* should be followed by "*Move Test*". Likewise, in many cases *Extract Method* should be followed by "*Extract Test*". To retain the documentation value of the unit tests, their structure should be in sync with the structure of the source code.

In our opinion, it makes sense to extend the existing descriptions of refactorings with suggestions on what to do with the corresponding unit tests, for example in the "mechanics" part.

The topic of refactoring test code is discussed extensively in Section 8.4. An issue of concern when changing test code is that we may "lose" test cases. When refactoring production code, the availability of tests forms a safety net that guards us from accidentally losing code, but such a safety net is not in place when modifying test code. A solution is to measure coverage [346] before and after changing the tests, e.g. with the help of Clover [108] or Emma [469]. One step further is *mutation testing*, using a tool such as Jester [379, 470]. Jester automatically makes changes to conditions and literals in Java source code. If the code is well-tested, such changes should lead to failing tests. Running Jester before and after test case refactorings helps to verify that the changes did not affect test coverage.

8.5.5 Refactoring Sessions

The meaningful unit of refactoring is a sequence of steps involving changes to both the code base and the test base. We propose the notion of a *refactoring session* to capture such a sequence. It consists of the following steps:

1. Detect *smells* in the code or test code that need to be fixed. In test-driven refactoring, the test set is the starting point for finding such smells.
2. Identify candidate refactorings addressing the smell.
3. Ensure that all existing tests run.
4. Apply the selected refactoring to the code. Provide a backwards compatible interface if the refactoring falls in category D. Only change the associated test classes when the refactoring falls in category E.
5. Ensure that all existing tests run. Consider applying mutation testing to assess the coverage of the test cases.
6. Apply the testing counterpart of the selected refactoring.
7. Ensure that the modified tests still run. Check that the coverage has not changed.
8. Extend the test cases now that the underlying code has become easier to test.
9. Ensure the new tests run.

The integrity of the code is ensured since (1) all tests are run between each step; (2) each step changes either code or tests, but never both at the same time (unless this is impossible).

8.6 Measuring Code and Test Code

In the previous sections we have seen how test suites affect program comprehension, how test suites themselves can be subjected to refactoring, and how refactoring of the production code is reflected in the test code. The last thing we investigate is whether there is a relation (correlation) between certain properties of the production code and those of the test code. We look at one property in particular, namely the *testability* of production code, based on our earlier work on finding testability metrics for Java systems [89].

For our investigation, we take advantage of the popularity of the JUnit framework [262]. JUnit's typical usage scenario is to test each Java class C by means of a dedicated test class C_T , generating pairs of the form $\langle C, C_T \rangle$. The route then that we pursue is to use these pairs to find source code metrics on C that are good predictors of test-related metrics on C_T .

To elaborate this route, we first define the notion of testability that we address, then describe the experimental design that can be used to explore the hypothesis, followed by a discussion of initial experimental results.

8.6.1 Testability

The ISO defines testability as “attributes of software that bear on the effort needed to validate the software product” [240]. Binder [65] offers an analysis of the various factors that contribute to a system’s testability, which he visualizes using the fish bone diagram as shown in Figure 8.4. The major factors determining test effort that Binder distinguishes include the test adequacy criterion that is required, the usefulness of the documentation, the quality of the implementation, the reusability

and structure of the test suite, the suitability of the test tools used, and the process capabilities.

Of these factors, we are concerned with the structure of the implementation, and with source code factors in particular. One group of factors we distinguish are *test case generation factors*, which influence the *number* of test cases required. An example is the testing criterion (test all branches, test all inherited methods), but directly related are characteristics of the code itself (use of if-then-else statements, use of inheritance). The other group of factors we distinguish are *test case construction factors*, which are related to the effort needed to create a particular test case. Such factors include the complexity of creating instances for a given class, or the number of fields that need to be initialized.

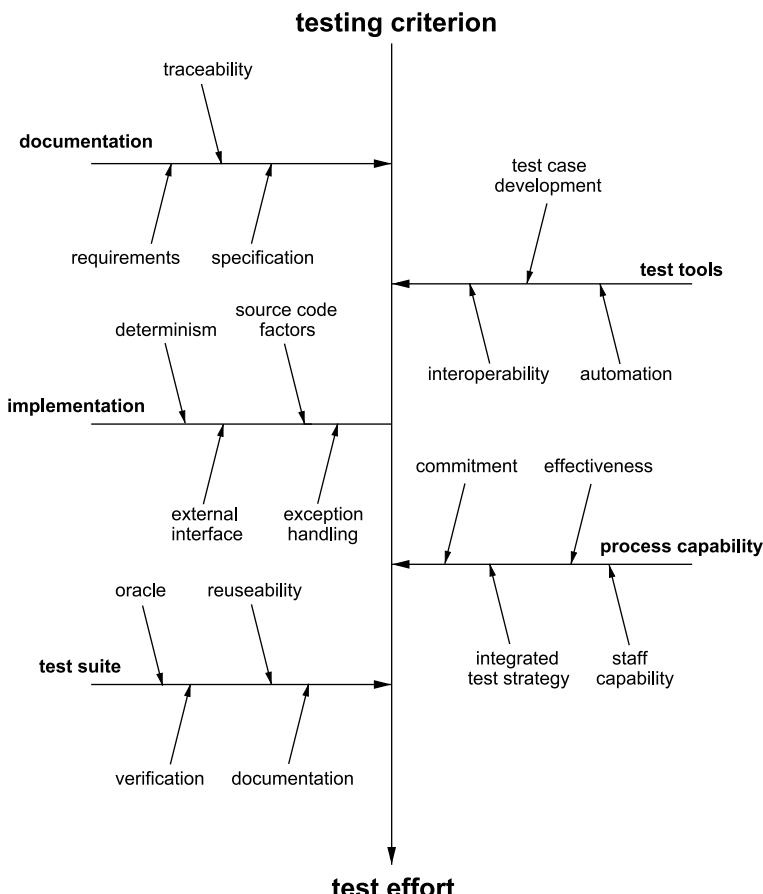


Fig. 8.4. The testability fish-bone [65, 89]

8.6.2 Experimental Design

Our goal is to assess the capability of a suite of object-oriented metrics to predict testing effort. We assess this capability from a class level perspective, i.e., we assess whether or not the values of object-oriented metrics for a given class can predict the required amount of effort needed for unit testing that class. The particular environment in which we conduct the experiments consists of Java systems that are unit tested at the class level using the JUnit testing framework.

To help us translate the goal into measurements, we pose questions that pertain to the goal:

Question 1: Are the values of the object-oriented metrics for a class associated with the required testing effort for that class?

To answer this question, we must first quantify “testing effort.” To indicate the testing effort required for a class we use the size of the corresponding test suite. Well-known cost models such as Boehm’s COCOMO [72] and Putnam’s SLIM model [421] relate development cost and effort to software size. Test suites are software in their own right; they have to be developed and maintained just like ‘normal’ software. Below we will see which metrics we use to measure the size of a test suite.

Next, we can refine our original question, and obtain the following new question:

Question 2: Are the values of the object-oriented metrics for a class associated with the size of the corresponding test suite?

From these questions we can derive a hypothesis that our experiments test:

H₀(m, n): There is *no* association between object-oriented metric *m* and test suite metric *n*,

H₁(m, n): There is an association between object-oriented metric *m* and test suite metric *n*,

where *m* ranges over our set of object-oriented metrics, and *n* over our set of test-suite based metrics.

As a candidate set of object-oriented metrics, we use the suite proposed by Binder [65] as a starting point. Binder is interested in testability as well, and uses a model distinguishing “complexity” and “scope” factors, which are similar to our test case construction and generation factors. The metrics used by Binder are based on the well known metrics suite provided by Chidamber and Kemerer [111], who for some of their metrics (such as the Coupling Between Objects and the Response for Class) already suggested that they would have a bearing on test effort. The metrics that we have used in our experiments are listed in Table 8.5.

For our experiments we propose the dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases) metrics to indicate the size of a test suite. The ‘d’ prepended to the names of these metrics denotes that they are the *dependent* variables of our experiment, i.e., the variables we want to predict. The dLOCC metric is defined like the LOCC metric.

Table 8.5. Metrics suite used for assessing testability of a class c

Metric	Description
DIT	Depth of inheritance tree
FOUT	Fan out, nr of classes used by c
LCOM	Lack of cohesion in methods—which measures how fields are used in methods
LOCC	Lines of code per class
NOC	Number of children
NOF	Number of fields
NOM	Number of methods
RFC	Response for class—Methods in c plus the number of methods invoked by c .
WMC	Weighted methods per class—sum of McCabe's cyclomatic complexity number of all methods.

The dNOTC metric provides a different perspective on the size of a test suite. It is calculated by counting the number of invocations of JUnit ‘assert’ methods that occur in the code of a test class. JUnit provides the tester with a number of different ‘assert’ methods, for example ‘assertTrue’, ‘assertFalse’ or ‘assertEqual’. The operation of these methods is the same; the parameters passed to the method are tested for compliance to some condition, depending on the specific variant. For example, ‘assertTrue’ tests whether or not its parameter evaluates to ‘true’. If the parameters do not satisfy the condition, the framework generates an exception that indicates a test has failed. Thus, the tester uses the set of JUnit ‘assert’ methods to compare the expected behavior of the class-under-test to its current behavior. Counting the number of invocations of ‘assert’ methods, gives the number of comparisons between expected and current behavior which we consider an appropriate definition of a test case.

Conducting the measurements yields a series of values $\langle m, n \rangle$ of object-oriented metric m and test suite metric n for a series of pairs $\langle C, C_T \rangle$ of a class C and its corresponding test class C_T . To test the hypotheses, we calculate Spearman’s rank-order correlation (which does not require a normal distribution of the data), yielding values $r_s(m, n)$ for metrics m and n . The significance (related to the number of observations made) of the value of r_s found is subsequently determined by calculating the t -statistic, yielding a value p indicating the chance that the observed value is the result of a chance event, allowing us to accept $H_1(m, n)$ with confidence level $1 - p$.

8.6.3 Experimental Results

Experiments were conducted on five software systems, of which four were closed source software products developed at the Software Improvement Group (SIG)⁶. Additionally, we included Apache Ant [18], an open source automation tool for software development. All systems are written in Java and the systems totaled over 290 KLOC.

⁶ <http://www.sig.nl>.

The key results for the Ant case study are shown in Table 8.6; similar results were obtained for the other case studies. The experiment shows that there is a significant correlation between test level metrics dLOCC (Lines of Code for Class) and dNOTC (Number of Testcases) and various class level metrics:

- There are several metrics related to *size*, in particular LOCC, NOM, and WMC. Since size can be considered a test case generation (we need more test cases) as well as a test case construction factor (larger classes become harder to test), it is natural that these metrics are correlated with test effort.
- The inheritance related metrics DIT (depth of inheritance tree) and NOC (number of subclasses) are *not* correlated with test metrics. In principle, test strategies in which, for example, extra subclasses lead to more intensive testing of the superclass, could cause NOC or DIT to be predictors of test effort. Apparently in the case studies these strategies were not adopted.
- Two metrics measuring external dependencies are Fan Out (FOUT) and Response-for-Class (RFC). Both are clearly correlated with both test suite metrics.
- The metrics LCOM (Lack of Cohesion of Methods) and NOF (Number of Fields) are correlated with the test metrics for the Ant case as well, but not for the four commercial case studies. One can expect NOF to be an indicator for test effort, for example, for initializing fields in a class. In cases where NOF is not an indicator, this may be due to the fact that the NOF metric only measures fields introduced in a particular class, and not fields inherited from superclasses.

Based on these findings, we conclude with the following observation:

Observation 6 Traditional object-oriented source code metrics applied to production code can indicate the effort needed for developing unit tests.

We refer to Bruntink and Van Deursen for a full account of the experiments described above [89].

Table 8.6. Correlation values and confidence levels found for Ant

r_s	dLOCC	dNOTC	p	dLOCC	dNOTC
DIT	-.0456	-.201	DIT	.634	.0344
FOUT	.465	.307	FOUT	< .01	< .01
LCOM	.437	.382	LCOM	< .01	< .01
LOCC	.500	.325	LOCC	< .01	< .01
NOC	.0537	-.0262	NOC	.575	.785
NOF	.455	.294	NOF	< .01	< .01
NOM	.532	.369	NOM	< .01	< .01
RFC	.526	.341	RFC	< .01	< .01
WMC	.531	.348	WMC	< .01	< .01

8.7 Concluding Remarks

In this section we first look back on the interplay between software testing and evolution. We then present a research agenda with a number of future research tracks, which are currently left unexplored.

8.7.1 Retrospective

Based upon *Observation 1* (see page 177), which states that an extensive test suite can stimulate the program comprehension process in the light of continuously evolving software, we have investigated the interactions between software evolution, software testing and program comprehension that exist in extreme programming in Section 8.3. Naturally, some (or all) of these elements are used in other development processes as well. For example, Humphrey stresses the importance of inspections, software quality assurance, and testing [236]. The Rational Unified Process emphasizes short iterations, architecture centric software development, and use cases [299]. Key publications on extreme programming [50, 254, 48] cover many issues related to comprehension, such as code expressing intent, feedback from the system, and tests to document code.

From our observation that test code has a distinct set of smells (see *Observation 2*, page 180), we looked at test code from the perspective of refactoring. Our own experiences are that the quality of test code is not as high as the quality of the production code. Test code was not refactored as mercilessly as production code, following Fowler's advice that it is acceptable to copy and edit test code, trusting our ability to refactor out truly common items later [183, p. 102]. When at a later stage we started refactoring test code more intensively, we discovered that test code has its own set of problems (which we translated into smells) as well as its own repertoire of solutions (which we formulated as test refactorings).

For each test smell that we identified, we have provided a solution, using either a potentially specialized variant of an existing refactoring from Fowler [183] or a dedicated test refactoring. We believe that the resulting smells and refactorings provide a valuable starting point for a larger collection based on a broader set of projects. This is in line with our *Observation 3* (see page 180).

Observation 4 (see page 185) states that when applying the refactorings as proposed by Fowler [183] on production code, a classification can be made based on whether these refactorings necessitate refactoring the test code as well. In Section 8.5 we have analyzed which of the documented refactorings affect the test code. It turns out that the majority of the refactorings are in category D (requiring explicit actions to keep the interface compatible) and E (necessarily requiring a change to the test code). We have shown the implications of refactoring tests with the help of Fowler's video store example. We then proposed the notion of *test-driven refactoring*, which uses the existing test cases as the starting point for finding suitable code level refactorings.

We have argued for the need to extend the descriptions of refactorings with a section on their implications on the corresponding test code. If the tests are to maintain

their documentation value, they should be kept in sync with the structure of the code. As outlined in *Observation 5* (see page 185), we propose, as a first step, the notion of a *refactoring session*, capturing a coherent series of separate steps involving changes to both the production and the test code.

The impact of program structure on test structure is further illustrated through *Observation 6* (page 197), which suggests that traditional object-oriented metrics can be used to estimate test effort. We described an experiment to assess which metrics can be used for this purpose. Note that some of the metrics identified (such as fan-out or response-for-class) are also indicators for class complexity. This suggests that high values for such metrics may call for refactorings, which in turn may help to reduce the test effort required for unit testing these classes.

From our studies we have learned that the interplay between software evolution and software testing is often more complex than meets the eye. The interplay that we witnessed works in two directions: software evolution is hindered by the fact that when evolving a system, the tests often need to co-evolve, making the evolution more difficult and time-intensive. On the other hand, many software evolution operations cannot safely take place without adequate tests being present to enable a safety net. This leads to an almost paradoxical situation where tests are essential for evolving software, yet at the same time, they are obstructing that very evolution.

Another important factor in this interplay is *program comprehension*, or the process of building up knowledge about a system under study, which is of critical importance during software evolution. In this context, having a test suite available can be a blessing, as the tests provide documentation about how the software works. At the same time, when no tests are available, writing tests to understand the software is a good way of building up comprehension.

We have seen that software evolution and testing are intertwined at the very core of (re)engineering software systems and continue to provide interesting and challenging research topics.

8.7.2 Research Agenda

During our study we came across a number of research ideas in the area of software testing and software evolution that are as yet still unexplored. The topics we propose can be seen as an addition or refinement to the topics that were addressed by Harrold in her “Testing: A Roadmap” [224].

Model Driven Engineering

MDE [453] is a modeling activity, whereby the traditional activity of writing code manually is replaced by modeling specifications for the application. Code generation techniques then use these models to generate (partial) code models of the application. This setup ensures the alignment between the models and the executable implementation. A similar approach can be followed when it comes to testing the application: modeling both the application and the tests through specifications. Muccini et al. consider this as the next logical step [381]. Recently, Pickin et al. have picked up on this research topic in the context of distributed systems [415].

Aspect Oriented Programming

AOP [276] is a programming paradigm that aims to offer an added layer of abstraction that can modularize system-level concerns (also see Chapter 9). However, when these aspects are woven into the base code, some unexpected effects can occur that are difficult to oversee. This can happen (1) when the pointcut is not defined precisely enough, resulting in an aspect being woven in at an unexpected place in the base program, or (2) because of unexpected results because of aspect composition, when the advice of two separate aspects is woven in. McEachen et al. describe a number of possible fault scenarios that can occur [357], but further research into this area is certainly warranted to prevent such fault scenarios through testing.

Test Case Isomorphism

Various sources indicate that test cases should be independent of each other because this decreases testing time, increases test output comprehensibility and having concise and focused tests increases their benefit as documentation of a specific aspect of the code [149, 131].

As said, having concise and focused tests decreases the testing time, which partly alleviates the problem of having to do selective regression testing [444, 445]. Another problem situation that is overcome, is the one described by Gaelli et al., whereby broken unit tests are ordered, so that the most specific unit test can be dealt with first [189].

Research questions of interest are how we can characterize and measure this isomorphism and what refactorings can be used to improve this isomorphism. These are related to detecting and removing the test implication smell described earlier.

Service-Orientation

The current trend is to build software systems from loosely coupled components or services (see Chapter 7). These services have mostly not been designed to co-exist with each other from their phase of inception and their “integration” often depends on the configuration of parameters at run-time. Although the components (or services) themselves will probably be of a higher quality, due to the fact that these are shared by many different projects (this can e.g. be in the case of *Commercial Off The Shelf* (COTS) components), testing the integration of these components or services is all the more important.

Although work related to testing components [212, 539] is readily available, not so much can be found on testing service-orientation. Although it is probable that many existing testing techniques can be adapted to work in this context, additional research is warranted. One of the first attempts at tool support for testing services is Coyote [507]. Commercial tool-support comes from SOAPSonar and Ikto’s LISA and also Apache’s Jakarta JMeter is useful when testing services [467].

Empirical Studies

Although many testing techniques are currently in circulation, there are few academic publications documenting how these testing techniques are exactly used and combined in industrial projects. Performing empirical studies that involve professional software developers and testers can lead to a better understanding of how software testing techniques or strategies are used (e.g., the study of Erdogmus et al. [161]). The results from this research can be used to build the next generation of testing techniques and test tools. An added benefit of this line of research is that by providing cutting-edge testing techniques to the industrial partners helps with knowledge and technology transfer about testing from academia to industry.

Repository Mining

The *a posteriori* analysis of software evolution, through the mining of e.g. versioning systems, provides a view on how the software *has* evolved and on how the software *might* evolve in the future (also see Chapter 3).

Up until recently however, no specific research has been carried out in this context that looks at the co-evolution of the software system and its associated test suite. Zaidman et al. performed an initial study on how this co-evolution happens in open source software systems [562]. They offer three separate views that show (1) the commit behavior of the developers, (2) the growth evolution of the system and (3) the coverage through time. The major observation that was made is that testing is mostly a *phased* activity, whereas development is more continuous.

In the same context, further research might provide answers to questions such as:

- Is every change to the production code backed up by a change to the test suite? Are there specific reasons why this should or should not happen?
- Can IDE's provide warnings when adaptations to the production code lead to reduced quality of the test suite?

Test Coverage

Even when continuous testing is becoming more and more commonplace in the development process [448], determining the test coverage [346, Chapter 7] is often not part of the fixed testing routine. In combination with the findings of Elbaum et al. [159], who have determined that even minor changes to production code can have a serious impact on the test coverage, this might lead to situations where the testing effort might prove to be insufficient. As such, the development of features in integrated developments environments that preemptively warn against drops in test coverage will lead to a more efficient and thorough test process.

Regression Testing

Regression testing provides you with a safety net when letting software evolve, because it guards against introducing bugs into functionality that previously worked fine. Ideally, these tests should be run after each modification, but regression testing

is often very expensive. Rothermel and Harrold provide a detailed survey of research in regression testing techniques, particularly in the domain of *selective* regression testing [445], where only that part of the regression test pertaining to the modification is re-run. Although selective regression testing can save costs, the process of determining which tests should be re-run is still expensive and the ultimate gain is thus relatively small. Further research into this topic is certainly warranted.