Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2015) 1–33
Procedia Computer Science

A Large-Scale Study on the Usage of Java's Concurrent Programming Constructs

Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, Roberto S. M. Barros
{ghlp, wst, jbfan, castor, roberto}@cin.ufpe.br

Informatics Center, Federal University of Pernambuco (CIn-UFPE), Av. Jornalista Anibal Fernandes, S/N, Recife-PE, 50.740-560, Brazil.

Abstract

In both academia and industry, there is a strong belief that multicore technology will radically change the way software is built. However, little is known about the current state of use of concurrent programming constructs. In this work we present an empirical work aimed at studying the usage of concurrent programming constructs of 2227 real world, stable and mature Java projects from SourceForge. We have studied the usage of concurrent techniques in the most recent versions of these applications and also how usage has evolved along time. Our study presents a number of interesting findings, for example: (I) More than 75% of the latest versions of the projects either explicitly create threads or employ some concurrency control mechanism; (II) More than half of these projects exhibit at least 47 `synchronized` methods and 3 implementations of the `Runnable` interface per 100KLoC, which means that not only concurrent programming constructs are used often but they are also employed intensively; (III) The adoption of the `java.util.concurrent` library is only moderate (approximately 23% of the concurrent projects employ it); (IV) efficient and thread-safe data structures, such as `ConcurrentHashMap`, are not yet widely used, despite the fact that they present numerous advantages.

Keywords:
Java, Concurrency, Software Evolution, OSS

1. Introduction

Multicore systems offer the potential for cheap, scalable, high-performance computing and also for significant reductions in power consumption. To achieve this potential, it is essential to take advantage of new heterogeneous architectures comprising collections of multiple processing elements. To leverage multicore technology, applications must be concurrent, which poses a challenge, since it is well-known that concurrent programming is hard [1]. A number of programming languages provide constructs for concurrent programming. These solutions vary greatly in terms of abstraction, error-proneness, and performance. The Java programming language is particularly rich when it comes to concurrent programming constructs. For example, it includes the concept of monitor, a low-level mechanism supporting both mutual exclusion and condition-based synchronization, as well as a high-level library [2], `java.util.concurrent`, also known as `j.u.c.`, introduced in version 1.5 of the language.

In both academia and industry, there is a strong belief that multicore technology will radically change the way software is built. However, to the best of our knowledge, there is a lack of reliable information about the current state of the practice of the development of concurrent software in terms of the constructs that developers employ. In this work, we aim to partially fill this gap.

Specifically, we present an empirical study aimed at establishing the current state of the practical usage of concurrent programming constructs in Java applications. We have analyzed 2,227 stable and mature Java projects comprising more than 600 million lines of code (LoC – without blank lines and comments) from SourceForge, one of the most popular open source code repositories. Our analysis encompasses several versions of these applications and is based on more than 50 source code metrics that we have automatically collected. We have also studied correlations among some of these metrics in an attempt to find trends in the use of concurrent programming constructs. We have chosen Java because it is a widely used object-oriented programming language. Moreover, as we said before, it includes support for multithreading with both low-level and high-level mechanisms. Additionally, it is the language with the most projects in SourceForge.

Evidence of how concurrent programs are written can raise developer awareness about available mechanisms. It can also indicate how well-accepted some of these mechanisms are in practice. Moreover, it can inform researchers designing new mechanisms about the kinds of constructs that developers may be more willing to use. Tool vendors can also benefit by supporting developers in the use of lesser-known, more efficient mechanisms, for example, by implementing novel refactorings [3, 4, 5]. Furthermore, results such as those uncovered by this study can support lecturers in more convincingly arguing students into the importance of concurrent programming, not only for the future of software development, but also for the present.

Mining data from the SourceForge repository poses several challenges. Some of them are inherent to the process of obtaining reliable data. These derive mainly from two factors: scale and lack of a standard organization for source code repositories. Others pertain to transforming the data into useful information. Grechanik et al. [6] discussed a few challenges that make it difficult to obtain evidence from source code. For example, getting the source code of all software versions is difficult because there is no naming pattern to define if a compressed file contains source code, binary code or something else. Furthermore, it is difficult to be sure that an error has not occurred during measurement, due to the number of projects and project versions. We address these challenges by creating an infrastructure for obtaining and processing large code bases, specifically targeting SourceForge. In addition, we have conducted a survey with the committers of some of these projects as an attempt to verify whether their beliefs are supported by our data.

Based on the data we have obtained, we propose to answer a number of research questions (RQ):

RQ1: Are mature Java applications concurrent? Do Java applications use concurrent programming constructs? We found out that more than 75% of the most recent versions of the examined projects include some form of concurrent programming, e.g., at least one occurrence of the `synchronized` keyword. In medium projects (20,001 - 100,000 LoC) this percentage grows to more than 90% and reaches 100% for large projects (over 100,000 LoC). In addition, the mean numbers (per 100,000 LoC) of `synchronized` methods, classes extending `Thread`, and classes implementing `Runnable` are, respectively, 66.75, 13, and 13.85. These results indicate that projects often use concurrent programming constructs and a considerable number do so intensively[1]. On the other hand, perhaps counterintuitively, the overall percentage of concurrent projects has not seen significant change throughout the years, despite the pervasiveness of multicore machines.

RQ2: Have developers moved to library-based concurrency? Our data shows that only 23.21% of the analyzed concurrent projects employ classes of the `java.util.concurrent` library. On the other hand, there has been a growth in the adoption of this library. However, this growth does not in general seem to be related to a decrease in the use of Java's traditional concurrent programming constructs, with a few exceptions. Furthermore, projects that have been in active development more recently, i.e., had at least one version released since 2009, employ the `java.util.concurrent` library more intensively than the mean. Therefore, the percentage of active, mature projects that use that library is actually higher than 23.21%.

RQ3: How do developers protect shared variables from concurrent threads? Most of the projects use `synchronized` blocks and methods. The `volatile` modifier, explicit locks (including variations such as read-write locks), and atomic variables are less common, albeit some of them seem to be growing in popularity. We also noticed a tendency of growth in the use of `synchronized` blocks. In particular, the growth in their use correlates positively with the growth in the use of atomic data types, explicit locks, and the `volatile` modifier.

RQ4: Do developers still use the `java.lang.Thread` class to create and manage threads? We found out that

[1]Throughout the paper, we often employ the terms "frequent" and "intensive". We use the first one to refer to the number of projects that employ a given construct. We use the term "often" as a synonym to "frequently". We employ the term "intensive" to refer to the number of uses of a given construct within a single project. For example, `synchronized` methods are used both frequently and intensively because a large number of projects use this construct and most of them use it many times.

implementing the Runnable interface is the most common approach to define new threads. Moreover, a considerable number of projects employ Executors to manage thread execution (11.14% of the concurrent projects). It was possible to observe that projects that employ executors exhibit a weak tendency to reduce the number of classes that explicitly extend the Thread class.

RQ5: Are developers using thread-safe data structures? We observed that developers are still using mostly Hashtable and HashMap, even though the former is thread-safe but inefficient and the latter is not thread-safe. Notwithstanding, there is a tendency towards the use of ConcurrentHashMap as a replacement for other associative data structures in a number of projects.

RQ6: How often do developers employ condition-based synchronization? A large number of concurrent projects include invocations of the notify(), notifyAll(), or wait() methods. At the same time, we noticed that a small number of projects have eliminated many uses of these methods, employing the CountDownLatch class, part of the java.util.concurrent library, instead. This number is not large enough for statistical analysis. Nevertheless, it indicates that general mechanisms with simple semantics, such as is the case with CountDownLatch, have potential to replace lower-level, more traditional ones.

RQ7: Do developers worry about exceptions that might cause abrupt thread failure? Our data indicate that less than 3% of the concurrent projects with more than 1KLoC implement the Thread.UncaughtExceptionHandler interface, which means that, in 97% of the concurrent projects, an exception stemming from a programming error might cause threads to die silently, potentially affecting the behavior of threads that interact with them. Moreover, analyzing these implementations, we discovered that developers often do not know what to do with uncaught exceptions in threads, even when they do implement a handler. This provides some evidence that new exception handling mechanisms that explicitly address the needs of concurrent applications are called for.

To provide a basic intuition as to what developers believe to be true for the usage of concurrent programming constructs, we have also conducted a survey with more than 160 software developers. These developers are all committers of projects whose source code we have analyzed. This survey presented respondents with various questions, such as "What do you believe to be the most often used concurrent/parallel programming construct of the Java language?". Throughout the paper, we contrast the results of this survey with data obtained by analyzing the Java source code.

This work makes the following contributions:

- It is the first large-scale **study** on the usage of concurrent programming constructs in the Java language, including an analysis on how the usage of these constructs has evolved along time.
- It presents a considerable amount of **data** pertaining to the current state-of-the-practice of real concurrent projects and the evolution of these projects along time.
- It presents results from a **survey** conducted with committers of some of the analyzed projects. This survey provides an overview of the perception of developers about the use of concurrent programming constructs.

The rest of the paper is organized as follows: Section 2 presents some background on concurrent programming in Java. Section 3 describes our survey setup and some initial results. Next, in Section 4, we describe the infrastructure we employed to download and extract the analyzed data. In Section 5 we present the results of our study organized in terms of the research questions. We then present the threats to the validity of this work in Section 6 and some implications in Section 7. Section 8 is dedicated to the related work. Finally, in Section 9, we present our conclusions and discuss future directions.

2. Background

INICIO DE TEXTO ADICIONADO

Before presenting our study, we provide a brief background on concurrent programming. A detailed presentation about concurrent programming concepts is available elsewhere [7].

Generally speaking, processes and threads are the main abstractions of concurrent programming. A process is a container that keeps all necessary information needed to run a program, for instance, the memory location where the process can read and write data. A thread, on the other hand, can be seen as a lightweight process. Even though threads have different implementations, threads and processes differ from each other in a way that multiple threads can exist within the same process and share its own data, while different processes do not share resources. Also, threads can share source code information. This feature is a double-edge sword, since it can came with the cost of well-known concurrency bugs such as race conditions.

However, one of the main reason to work with threads is because they are easier and faster than processes since threads have no resources associated. For instance, creating a thread can be one hundred times faster than creating a process [7].

On a single processor, multithreading generally occurs by time-division multiplexing. That is, the processor switches between different threads. This context switching generally happens fast and the end-user perceives that the threads are running at the same time. On a multiprocessor, or in a multi-core system, the threads or tasks will actually run at the same time, with each processor or core running a particular thread. The number of threads running at the same time is now bounded by the number of available processors.

FIM DE TEXTO ADICIONADO

Concurrent programming has been an exciting area of research in the last decade. Although no consensus has emerged on a single model of concurrency, many advances have been made with the development of various contending models [8, 9]. Besides that, regardless of the model of concurrency, many researchers [3, 4, 10, 11] argue that high level concurrency libraries can improve software quality.

The `java.util.concurrent` library aims to simplify the development of concurrent applications in the Java language. Using this framework, even a less experienced programmer can write working concurrent applications. The `java.util.concurrent` library offers several features to make the task of concurrent programming easier. In addition, the library is optimized for performance. Below we discuss some of its most well-known constructs. We assume the reader is familiar with the Java programming language and with basic concepts of concurrent programming, such as locks, mutual exclusion, and condition-based synchronization. The `java.util.concurrent` library includes some constructs, such as semaphores and exchangers, that we do not discuss in this paper because they are very seldom used. For instance, we found out that the `Semaphore` class has never been used in the analyzed projects.

Locks: Implementations of the `Lock` interface, such as `ReentrantLock`, support more flexible locking than can be performed using `synchronized` methods and blocks. They promote more versatile structuring, may have different properties depending on how threads access data, and may support multiple associated `Condition` (an interface defining condition variables associated with a lock) objects. A lock is a tool for controlling access to a shared resource by multiple threads. In general, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and every access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a `ReadWriteLock`. `Lock` implementations provide additional functionality over the use of `synchronized` methods and blocks by supporting non-blocking attempts to acquire a lock (`tryLock()`), and attempts to acquire lock that can be interrupted.

Atomic Data Types: These data types are provided by a small toolkit of classes that support lock-free, thread-safe programming on single variables. In essence, the classes in the `java.util.concurrent.atomic` package extend the notion of `volatile` values, fields, and array elements, providing an atomic conditional update operation using the `compareAndSet()` method. This method atomically sets a variable if its current value equals that of the method's first argument, returning `true` on success. The classes in this package also contain methods to get and unconditionally set values, and to increment and decrement the value of the variable. Examples of classes in this package are `AtomicBoolean`, `AtomicInteger` and `AtomicIntegerArray`.

Concurrent Collections: It is a group of Collections designed for use in multithreaded contexts. This group includes `ConcurrentHashMap`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, and `ArrayBlockingQueue`. The `Concurrent` prefix used with some classes in this package is a shorthand indicating several differences from similar *synchronized* classes, which employ a single lock for the entire collection. For example the classes `Hashtable` and `Collections.synchronizedMap(...)` are synchronized, but `ConcurrentHashMap` is "concurrent". A concurrent collection is thread-safe, but not governed by a single lock. `ConcurrentHashMap`, in particular, safely permits any number of concurrent reads as well as a tunable number of concurrent writes.

Condition-based synchronization: `java.util.concurrent` provides some classes that can replace the `wait()` and `notify()` methods. `CountDownLatch` is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads have all been completed. A `CountDownLatch` waits for N threads to finish before allowing all of them to proceed. `CyclicBarrier` is another synchronization aid. It allows a set of threads to all wait for each other to reach a common barrier point.

Executors: Executors, embodied by the `Executor` interface and its implementations and sub-interfaces, support multiple approaches for managing thread execution. They provide an asynchronous task execution framework. An `ExecutorService` manages queuing and scheduling of tasks, and allows controlled shutdown. `ExecutorService` interface and its implementations provide methods to asynchronously execute any function expressed as a `Callable`, the result-bearing analog of `Runnable`. The `ScheduledExecutorService` subinterface adds support for delayed and periodic task execution. A `Future` returns the results of a function, allows determining whether the execution has completed, and provides the means to cancel execution. Its implementations provide tunable, flexible thread pools. The `Executors` class provides factory methods for the most common kinds and configurations of `Executors`, as well as a few utility methods for using them[2].

3. Survey

We have conducted a survey with programmers in order to gather information about the perception of developers about the usage of concurrent programming constructs in Java. Using this information we can check whether the intuition of these developers is reflected by the source code of real systems. The questionnaire was designed to the recommendations of Kitchenham et al. [12], following the phases prescribed by the authors: planning, creating the questionnaire, defining the target audience, evaluating, conducting the survey, and analyzing the results. Firstly, we defined the topics for the questions. The topics are: respondents' experience, how familiar they are with concurrent programming and, lastly, we asked direct questions about the state of use of concurrent programming techniques. The questionnaire had 9 questions and is structured to limit responses to multiple-choice, Likert scales (responses given in a scale which starts from 0 until 10, where 0 means no knowledge at all and 10 means super expert), and also free-forms. It includes a single question (#9) where the respondents could answer using free text.

After defining all the questions in the questionnaire, we obtained feedback iteratively and clarified and rephrased some questions and explanations. This feedback was obtained from analysis and discussion with a group of specialists and also from one pilot of the survey. Together with the instructions of the questionnaire, we included some simple examples as an attempt to clarify our intent. Table 1 presents the questions of the questionnaire. The complete list of questions as well as all the responses of the survey are available at the companion website of the paper [3].

Our target population consists of programmers who have performed at least one commit to an open-source software analyzed in this work. It is important to mention that this work analyzed projects on SourceForge, which uses Subversion as its default version control system. Nonetheless, Subversion does not necessarily keep track of the email address of the commit author. For example, the commit author could use either an anonymous id or a pseudonym. The latter is, in fact, more commonly used than the email address. Another problem with SourceForge is that old repositories are fairly often external to SourceForge, which makes it hard to track them for a large number of projects. Then, in order to gather the email address of these programmers, we investigated which projects have moved to Github, since it makes it easier to find the email address of committers. We have found 72 projects that have moved to Github. In these projects, we have found 2,353 unique email addresses, but only 1,953 of them were valid. When sending the survey to these programmers, 273 email messages have been rejected by the server with unknown domain notices and another 18 have been auto-responsed with out-of-office messages. Over a period of 20 days, we obtained 164 responses, resulting in a 9.75% response rate. This response rate is almost twice higher than the response rates found in surveys in the software engineering field [12]. Table 2 synthesizes the survey data.

As we can see in the above table, 26% of the respondents have more than 12 years of software development experience and, on average, the respondents consider themselves to be moderately experienced in concurrent programming (a value 6 on a scale from 0 to 10, where 0 means no knowledge at all, and 10 means an expert). In their experience, the top 5 most used concurrent programming constructs are the same found on the first versions of the Java language.

[2]Throughout the paper, we often employ the term "executors constructs". We use it to refer to classes related to the Executor framework, such as `Executor`, `ExecutorService`, `ScheduledExecutorService`, `Executors`, among others.

[3]http://www.cin.ufpe.br/~groundhog

Table 1. The Survey Questions

1.	How many years do you have developing Java projects?
2.	Which is your experience using the default Java concurrent/parallel constructs?
3.	Choose one or more of the following concurrent/parallel programming constructs that you have used in a Java project
4.	What do you believe to be the percentage of open-source Java projects that use at least one concurrent/parallel construct (explicitly on the source code, not as a third-party library)?
5.	What do you believe to be the percentage of open-source Java projects that use at least one construct of the `java.util.concurrent` library (explicitly on the source code, not as a third-party library)?
6.	What do you believe to be the most often used concurrent/parallel programming construct of the Java language?
7.	Which you believe to be the most often used construct of the `java.util.concurrent` library?
8.	Have you ever been involved in, or heard about, some sort of initiative within a Java project in which you work, or have worked, aiming to improve the performance or the scalability of the application through the use of concurrent/parallel programming techniques?
9.	If so, could you briefly describe this experience?

Also, on average, they believe that half of the open-source projects use at least one basic concurrent construct and 30% of the projects employ the `java.util.concurrent` library.

In addition, 53% of the respondents said they have used concurrent programming techniques to improve the performance and/or the scalability of an application. One of the anonymous respondents has detailed how difficult it is to write correct concurrent programs – and how they have achieved performance improvements:

> *Concurrency is hard on many levels - effectively parallelizing code, avoiding potential deadlocks, etc. If not all the developers in the project are disciplined, it is also easy to slip on practices such as the pedantically correct use of try-finally when managing locks, etc. and to create fragile concurrent code. Java constructs help somewhat with the details, but the main burden still falls on the programmer understanding concurrency and its implications thoroughly. There are numerous pitfalls also in the language (e.g. long not guaranteed to be atomic in all environments) that java.util.concurrent utilities can help with, but only when the programmer understands the problem and knows what approaches and utilities to use to avoid it. The newer JLS versions have patched up some problems (e.g. if I remember correctly, now you can count on all statements in a constructor having completed before the constructor returns, which was not the case before), but the idiosyncrasies of the language still put a great burden on the developer to know all the pitfalls or to develop ultra-defensively.*

In the remainder of this paper, we discuss the main findings of the survey based on the seven research questions stated in Section 1.

Table 2. The Survey Responses

# Question	Response
1.	1 to 2 years ⇒ 5% 2 to 5 years ⇒24% 5 to 8 years ⇒ 22% 8 to 12 years ⇒ 24% more than 12 years ⇒ 26%
2.	(no knowledge at all) 0 ⇒ 0% 1 ⇒ 6% 2 ⇒ 5% 3 ⇒ 7% 4 ⇒ 9% 5 ⇒ 12% 6 ⇒ 15% 7 ⇒ 20% 8 ⇒ 20% 9 ⇒ 3% (a super-expert) 10 ⇒ 2%
3.	synchronized keyword (block statement) ⇒ 5% synchronized keyword (method statement) ⇒ 5% java.lang.Thread ⇒ 5% java.lang.Runnable ⇒ 5% Object.wait() method ⇒ 4%
4.	Median ⇒ 50% Mean ⇒ 51.43% SD ⇒ 28.48%
5.	Median ⇒ 30% Mean ⇒ 36.63% SD ⇒ 25.69%
6.	synchronized keyword (method statement) ⇒ 36% synchronized keyword (block statement) ⇒ 21% java.lang.Thread ⇒ 19% java.lang.Runnable ⇒ 16% Others ⇒ 7%
7.	java.util.concurrent.ConcurrentHashMap ⇒ 21% va.util.concurrent.ExecutorService ⇒ 13% java.util.concurrent.ConcurrentMap ⇒ 12% java.util.concurrent.Executor ⇒ 10% java.util.concurrent.Future ⇒ 10%
8.	Yes ⇒ 53% No ⇒ 47%

4. Study Setting

This section describes the configuration of our study: our basic assumptions, our mining infrastructure, and the metrics suite that we employed.

We have built a set of tools to download projects from SourceForge, analyze the source code, and collect metrics from these projects. It comprises a crawler, a metrics collection tool, and some auxiliary shell scripts. We call this infrastructure Groundhog. Figure 1 depicts the infrastructure we employed. Initially, the crawler populates the project repository with Java projects from SourceForge, including their various versions (a).

We obtain the projects by means of HTTP requests, instead of directly accessing the source code repositories of the projects. We use this approach because we are only interested in analyzing project releases, stable versions that are available to the general public. Source code repositories often do not clearly identify releases and, when they do, they employ inconsistent approaches. On the other hand, SourceForge makes it relatively easy to obtain release versions by means of HTTP requests.

When the projects have all been downloaded, all the compressed files are extracted into our local repository (b). We are currently capable of uncompressing zip, rar, tar, gz, tgz, bz2, tbz, tbz2, bzip2, and 7z files. After that, the metrics collection tool parses the source code, collects metrics, and stores the results in the metrics repository (c). Finally, it generates input, as CSV files, to be statistically analyzed by R[13].

The crawler is an extension of Crawler4j[4], an open source web crawler framework. This framework is multi-threaded and written in Java. We also implemented additional scripts to organize project versions based on dates available at SourceForge and to check if the target project was ready to be analyzed, fixing its structure when necessary. To collect concurrency metrics we used the `JavaCompiler`[5] class to parse the source code and build parse trees. The trees are traversed and the metrics are extracted and stored in text files.

The metrics consist of counting LoC of classes that extend the `Thread` class, of classes that implement the `Runnable` interface, and of uses of some Java keywords such as `synchronized` and `volatile`, as well as number of instantiations of types belonging to the `j.u.c.` library, such as `AtomicInteger`, `ConcurrentHashMap`, `ReentrantLock`, and many others. Table 3 lists the elements whose number of occurrences we have measured.

Our analysis focuses exclusively on mature and stable projects, as identified by the project developers. Furthermore, projects that did not have at least one release after 2004 are not considered, because `java.util.concurrent` was released as part of the JDK in December 2004. Moreover, we have only examined projects with at least 1,000 LoC, to avoid trivial systems. We have analyzed the projects considering both their most recent versions and their evolution along time. In the latter case, we have studied multiple versions of the projects. To better understand their evolution, we have also computed the differences in the values of some metrics considering recent and old versions of the systems. We then calculated the Pearson correlation [14] between these differences. This has helped us to identify, for example, that a number of projects exhibit a tendency to switch from extending the `Thread` class directly to using executors to manage thread execution.

[4]http://code.google.com/p/crawler4j/
[5]http://docs.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html

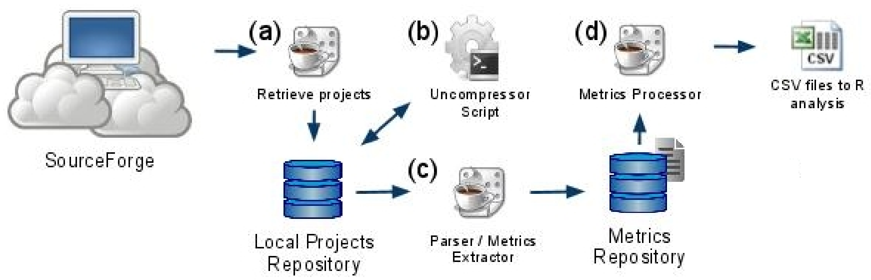

Figure 1. In (a) the crawler populates the infrastructures repository with Java Projects from Sourceforge. In (b) a shell script extracts all compressed files into our local repository. In (c) the metrics collection tool parses the source code, collects metrics, and stores the results in the metrics repository. In (d) the metric collection tool generates input CSV files to be statistically analyzed by R.

Table 3. Collected metrics: constructs that had their number of occurrences counted, e.g., "implements Runnable" is the number of classes implementing the `Runnable` interface in the program.

Group	Metrics
Light Weight Threads	AbstractExecutorService, Executor, ExecutorService, Future, FutureTask, ForkJoinTask, ForkJoinPool, ThreadPoolExecutor, RecursiveAction, RecursiveTask, RunnableFuture, RunnableScheduledFuture, ScheduledFuture, ScheduledExecutorService, ScheduledThreadPoolExecutor
Atomic Data Type (ADT)	AtomicBoolean, AtomicInteger, AtomicLong
Synchronized Collections methods	Collections.synchronizedCollection(), Collections.synchronizedList(), Collections.synchronizedMap(), Collections.synchronizedSortedMap(), Collections.synchronizedSet(), Collections.synchronizedSortedSet()
Concurrent collections	HashMap, ConcurrentHashMap, ConcurrentMap, ConcurrentSkipListMap, ConcurrentNavigableMap, ArrayBlockingQueue, PriorityBlockingQueue, LinkedBlockingQueue, SynchronousQueue, LinkedTransferQueue, BlockingQueue, DelayQueue, LinkedBlockingDeque
Locks	Locks, ReentrantReadWriteLock, ReentrantLock
Lock methods	ReentrantReadWrite.Writelock, ReentrantReadWrite.Readlock
Thread class methods	Thread.setDefaultUncaughtExceptionHandler, Thread.UncaughtException, Thread.setUncaughtExceptionHandler, Thread.UncaughtExceptionHandler
Thread creation	extends Runnable, implements Runnable, implements Callable, extends Thread, new Thread()
Synchronization	synchronized (blocks), synchronized (methods), Condition, CountDownLatch, CyclicBarrier, notify(), notifyAll(), Semaphore, volatile, wait()
Other	Hashtable

All results presented in this article are normalized to avoid distortions caused by very large absolute values and to make them more directly comparable. For example, to calculate the result for the metric `implements Runnable` for the version of the Dr.Java project released in 22-08-2011, we have divided the number of occurrences of `implements Runnable`, which is 6, by the number of lines of code, which is 112,703, resulting in 0.000053238. This result was then multiplied by 100,000 and the final result is 5.3238. All the collected metrics were normalized in this fashion and we use these normalized values in the remainder of the paper. References to absolute values throughout the paper are clearly presented as such. Both the absolute and the normalized values for all the metrics are available in the companion website of the paper.

Finally, based on the survey results, we pose a number of assumptions that represent the expectations of developers regarding the state-of-the-use of some concurrent programming techniques. Our assumptions are the following:

A1 Java projects frequently employ concurrent programming constructs (mean estimate: 51,43%);

A2 Java projects frequently employ constructs from the `j.u.c` library (mean estimate: 36.63%);

A3 `synchronized` methods are the most frequently used concurrent programming construct;

A4 `ConcurrentHashMap` is the most frequently used concurrent programming construct from the `j.u.c.` library;

A5 Initiatives to reengineer existing systems so as to leverage multicore architectures are commonplace.

5. Study Results

This section presents the results of our study. We organized the results in terms of the research questions.

5.1. RQ1: Are mature Java applications concurrent? Do Java applications use concurrent programming constructs?

This study analyzed 2,227 projects, of which 1,723 include at least one occurrence of a concurrent programming construct (77.5% of them). Also, the fourth question of the survey is directly associated with this research question. On average, the respondents believe that 51.12% of the projects use at least one concurrent programming construct (median of 50%), with standard deviation of 28.67. Hereafter, we refer to these projects as "concurrent projects". Among these projects, only 400 (23.21%) use the `java.util.concurrent` library. Moreover, this library had been

Table 4. General information about the projects.

#Projects	2,227
#Small Projects	1,700
#Medium Projects	616
#Large Projects	197
#Concurrent projects	1,723
#Concurrent projects that use `java.util.concurrent`	400
#Non concurrent projects	504
# of LoC (all versions of all projects)	623,440,010
# of LoC (all versions of concurrent projects)	612,897,893
# of LoC (all versions of non concurrent projects)	10,542,117
Size on disk (all versions of all projects in GB)	124

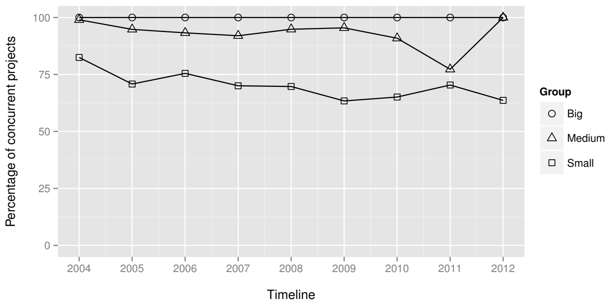

Figure 2. Percentage of project releases, per year, that include concurrent programming constructs.

available for general use for at least five years before it was incorporated into the JDK. According to question 5 of the survey (Table 1), on mean, survey respondents believe that 36.1% (median 30.0%, standard deviation 25.42) of the concurrent projects employ these constructs. Based on these results, it is fair to say that, for the population under study, the use of concurrent programming constructs is more common than most developers believe (A1). On the other hand, their intuition about the use of the `java.util.concurrent` library varies widely, with the actual percentage of concurrent projects that employ it being smaller than the standard deviation of the responses of the developers (25.42).

Table 4 presents some general size metrics for the analyzed projects. The sum of large (100,001+ LoC), medium (20,001 - 100,000 LoC) and small (1,000 - 20,000 LoC) projects is not equal to the overall number of projects. This happens because some projects can be in more than one category, e.g., a small project might become a medium project along its evolution and be counted in both categories. Moreover, it is interesting to notice that, even though more than 22% of the projects are not concurrent, the sum of the LoC of these projects corresponds to just 1.65% of the total. The mean non-concurrent project has 28,946 LoC and presents median 6,313 LoC, whereas the mean concurrent project has 41,173 LoC and presents median 12,419 LoC.

Tables 10 to 13 introduce descriptive statistics pertaining to some of the collected metrics. This information refers only to the latest versions of the projects. More than 60% of the small projects, more than 90% of the medium projects, and all large projects employed some concurrent programming mechanism up to May 2012. Figure 2 presents the percentages of project releases, per year, that include concurrent programming constructs, considering small, medium, and large projects. The data in the aforementioned figure shows that the percentage of concurrent projects released per year has not changed significantly since 2005.

We consider that a project is concurrent if its last version has a value greater than zero for any of the collected metrics. In practice, this means at least one occurrence of `implements Runnable`, `extends Thread`, or the `synchronized` keyword. This is consistent with the survey respondents in question 6: among them, 92% believe that these constructs are the most often used in concurrent projects. In fact, most concurrent projects go well beyond a single occurrence of a concurrent programming construct. According to Table 12, most concurrent projects have over 48 `synchronized` methods and 31 `synchronized` blocks per 100KLoC. Furthermore, in Table 10 the median numbers of classes per 100KLoC implementing the `Runnable` interface for small, medium, and large concurrent projects are 27.29, 7.18, and 3.73, respectively.

Most of the concurrent projects use low-level concurrency control mechanisms and a smaller number employ the high-level abstractions of the `java.util.concurrent` library. We observed that the basic concurrent Java programming constructs are more popular than the `java.util.concurrent` constructs, which is consistent with the survey results. These constructs have been introduced in the first version of the Java language and have been available ever since. The most usual way to create threads is through the implementation of the `Runnable` interface. It occurs in 47.16%, 66.27%, and 86.89% of small, medium, and large concurrent projects, respectively. To control access to shared state, `synchronized` methods are the most frequently used construct, being present in 70.98%, 92.37%, and 97.57% of small, medium, and large concurrent projects, respectively. Tables 10 and 12 also shows that ensuring mutual exclusion and managing concurrent/parallel execution are recurring problems even for small applications: 47.16% of them include classes that implement the `Runnable` interface and more that 70% include `synchronized` methods. The latter suggests that A3 is realistic. New techniques to solve these problems have therefore ample opportunity for adoption.

Tables 11 and 13 show that among the constructs of the `java.util.concurrent` library, atomic variables and `ConcurrentHashMap` have the strongest adoption. In particular, 29.94% of the large projects employ the former and 27.91% use the latter. Medium and small projects use these constructs less often. The survey respondents (21% of them) also believe that `ConcurrentHashMap` plays an important role in the high-level constructs (Assumption A4). Nonetheless, 51 respondents believe that `Executors` are more often used than atomic data types (11 respondents) in the `java.util.concurrent` library. It should be noted that some constructs have rarely been adopted. For example, the `PriorityBlockingQueue` and `ConcurrentSkipListMap` are employed by approximately 1.45% of the large concurrent projects (3 projects each). However, among the survey respondents, 48 (29.26%) claim to have used these constructs (`PriorityBlockingQueue`: 25 respondents, `ConcurrentSkipListMap`: 23 respondents). One system can take advantace of `ConcurrentSkipListMap` in order to guarantees good performance on a wide variety of operations. Also, besides the fact that these collections have a number of operations that `ConcurrentHashMap` does not have (such as `ceilingEntry`, `ceilingKey`, `floorEntry` among others), it also maintains a sort order, which would otherwise have to be calculated. Among the small projects, less than 0.4% use these constructs. This result suggests that there is room for improving existing systems. Skip lists [15] are known to be scalable and fast for search operations even in the presence of concurrent threads. Nevertheless, few systems use them, possibly due to developers not being familiar with them. As mentioned before, we did not find projects that employ semaphores, though 47 respondents (28.7% of all the respondents) have mentioned they have used this construct in a professional Java project.

INICIO DE TEXTO ADICIONADO

Moreover, we analyzed the domain of our projects in order to understand how concurrent constructs usage are related to the project domain. We used the default domain metadata available at the SourceForge webpage of the project. Although important, developers are not required to inform the category of their projects. In earlier version of SourceForge, it was possible to set more than 3 categories for a project. Nowadays, however, developers can set up to 3 categories for a given project. Table 5 shows the how the concurrent constructs are used over these categories.

As Table 5 shows, the categories that have more projects are almost the same in both concurrent and non-concurrent projects. For example, the first three categories are the same and among the top 10, the only difference are "Communication" for concurrent projects and "Education" for non concurrent projects.

Since categorization is not a required feature, not all downloaded projects had been categorized. Table 6 presents the number of categories per project.

According to the subcategorization presented in SourceForge, the "System" category should be used to projects related to "Operating System Kernels", "Emulators" and others low level technical features. Similarly to "System" category, the "Games/Entertainment" one also has a significative difference between concurrent projects (78%) and non-concurrent ones (22%). Another interesting fact is that 94% of all "Communication" projects are concurrent.

We also analyzed the top 10 metrics most used in categories that have more than 100 projects . The most popular metrics among them are: sync methods, HashMap, sync blocks, Hashtable, implements Runnable, wait(), extends Thread, j.u.c, notifyAll(),notify() and volatile. Generally speaking, we found that the 10 most used metrics were the same used across the project categories. According to the Table 7 the most widely used metric is the `synchronized` method, used in 7 out of 10 categories.

As presented in Table 8, at least half of the all projects do not use 4 of the most used metrics for each category, for example, in category System the metrics j.u.c, wait() , volatile and notifyAll() the median is zero, it means that half of

Table 5. Project Category.

Category	# Concurrent Projects		Domain Category	Non Concurrent Projects
Software Development	625		Software Development	233
Internet	267		Scientific / Engineering	74
Scientific / Engineering	265		Internet	58
System	242		Office / Business	56
Office / Business	233		Games / Entertainment	39
Multimedia	192		Multimedia	39
Communication	190		Formats and Protocols	38
Database	139		Education	35
Games Entertainment	134		Database	33
Formats and Protocols	116		System	24
Education	96		Text Editors	15
Text Editors	67		Communication	14
Security	60		Security	11
Other	28		Other	9
Desktop Environment	25		Printing	7
Mobile	12		Mobile	3
Printing	9		Desktop Environment	2
Terminals	8		Terminals	1
Social Engineering	3			
Religion / Philosophy	1			

Table 6. Summary Project Domain Category.

# Projects with categories	2090
#Projects with 1 category	1168
#Projects with 2 categories	611
#Projects with 3 categories	253
#Projects with 3+ categories	50
#Projects without categories	137

these projects did not employ these metrics.

In addition, we figured out that the preferred way to create threads in almost all categories was through the implementation of the `Runnable` interface. The only category witch did not follow this rule was Communication. Communication projects prefer create threads extends the `Thread` class.

The `synchronized` blocks and `synchronized` methods are the most popular way of synchronization in the analyzed domains, these metrics are the top 3 most used metrics in all the categories. The other possible way could be through the `Lock` interface, but we did not find usage of this interface in the top 10 metrics in our analyzed categories.

Finally, although the `java.util.concurrent` library was used in all domains, the developers did not used `ConcurrentHashMap` instead of `HashTable` and `HashMap`. Both `HashTable` and `HashMap` were widely used in all categories. A detailed report with descriptive data can be found at the companion website [6].

FIM DE TEXTO ADICIONADO

5.2. RQ2: Have developers moved to library-based concurrency?

To try to answer this question, we have studied the temporal evolution of concurrent Java programs. To obtain these results, we analyzed the latest versions of projects launched each year, until May 2012. It is important to say that it was not possible to download all the versions of some projects due to name changes between versions. In general, we found out that `java.util.concurrent` has been adopted more intensively along the years. Figure 3 shows this trend: the usage of `java.util.concurrent` increased from less than 10 uses per 100KLoC in 2006 to almost 30 uses per 100KLoC in 2011. To obtain the results presented in Figure 3, we have not taken into account the size groups (small, medium and large). We used the median of occurrences of `java.util.concurrent` constructs in the latest version of all projects released each year. In parallel we summed the number of lines of code of the selected project versions. Finally we divided the former by the latter and multiplied the result by 100K to normalize it (Section 4).

Figure 4 provides a different perspective. It shows, among the small, medium, and large projects, the percentage of project releases that are concurrent, per year, as well as the percentage of project releases that employ

[6] http://www.cin.ufpe.br/~groundhog

Table 7. The most used metric by category.

Cateogory	The most used metric
SoftwareDevelopment	sync methods
Internet	HashMap
ScientificEngineering	sync methods
System	sync methods
OfficeBusiness	HashMap
Multimedia	sync methods
Communication	sync methods
Database	sync blocks
GamesEntertainment	sync methods
FormatsandProtocols	sync methods

Table 8. The most used metric by “System” projects category.

Metric	Total	Mean	Median	Standard Deviation
sync methods	41806,34	172.75	75.64	245.68
sync blocks	22627,46	93.50	34.03	155.33
HashMap	22571,22	93.27	60.57	112.01
Hashtable	5846,93	24.16	1.85	56.34
implements Runnable	5324,07	22.00	6.58	39.74
j.u.c.	5174,92	21.38	0	73.15
extends Thread	4642,36	19.18	2.19	40.70
wait()	4095,02	16.92	0	36.09
volatile	2905,51	12	0	33.80
notifyAll()	2517,94	10.40	0	29.64

`java.util.concurrent`. The figure also shows, for example, that more than 20% of all versions of small projects released in 2011 use the `java.util.concurrent` library. For medium and large projects, that percentage reaches almost 40% and almost 60%, respectively. However in 2012 (up to may) no released large project had employed `java.util.concurrent`. According to Table 4, 17.96% of all analyzed projects use the `java.util.concurrent` library. Figure 4 shows that versions of medium and large projects (and of the small ones, after 2008) consistently use the library more frequently than the 17.96% presented in Table 4. These results suggest that projects using the library are released more often than the mean. Otherwise, we would see mean frequencies of use that would more closely match those 17.96%. The bottom line is that, even though a large percentage of the projects does not use the `java.util.concurrent` library, many of them have not seen releases in the last few years. On the other hand, projects using the library have been in more active development, which suggests that it is used more frequently in practice than one would believe on a first examination.

We have analyzed the evolution of the usage of concurrent programming constructs by comparing, in pairs, how the corresponding metrics have evolved for the projects in our study. For each of the metrics of Table 3, we obtained the difference between its values in the first and last versions. We call this difference the delta. As an example, we can calculate the deltas for uses of `ReentrantLock` and `synchronized` blocks in the Liferay Portal project. This project, one of the largest in this study, comprising more than 1,661,000 LoC. The oldest version of this proejct that we analyzed does not use `ReentrantLock`. On the other hand, the most recent version has 0.6621 occurences of `ReentrantLock` per 100KLoC. In this case, the delta is 0.6621, which represents the difference between 0.6621 (recent version) - 0 (first version). On the other hand, the delta for `synchronized` blocks is -40.271, which refers to the difference between the most recent version, 6.199, and the first version, 46.471. We then proceeded to calculate the Pearson correlation between deltas for each pair of metrics we wanted to analyze. In other words, we did not compare different projects but the evolution of two different metrics for the same project. It is important to notice that selected projects had to have at least one occurrence of both analyzed metrics in at least one of the two versions. Some examples of metric correlations are presented in Table 18. We present the correlation[7] results in Sections 5.3–5.6.

In this study, we have also identified projects that exhibited large increases or decreases in the use of a concurrent programming construct during the analyzed period. In the cases where we detected more intense changes between releases, we decided to perform a manual analysis to find out the reasons behind these changes. Since we could not

[7]Due to the extensive number of possible correlations, most of the time we only present results that have, at least, weak correlations (correlation value between 0.1 to 0.3 and -0.3 to -0.1). We make a few exceptions to this rule for illustrative purposes.

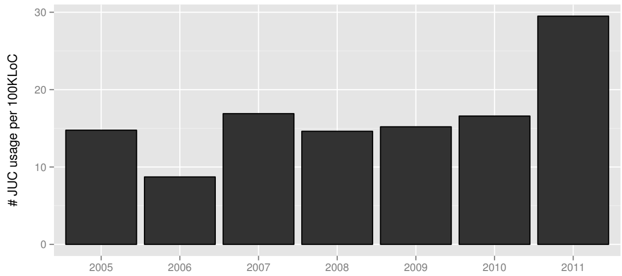

Figure 3. Uses of `j.u.c.` constructs per 100KLoC, considering the project versions released each year.

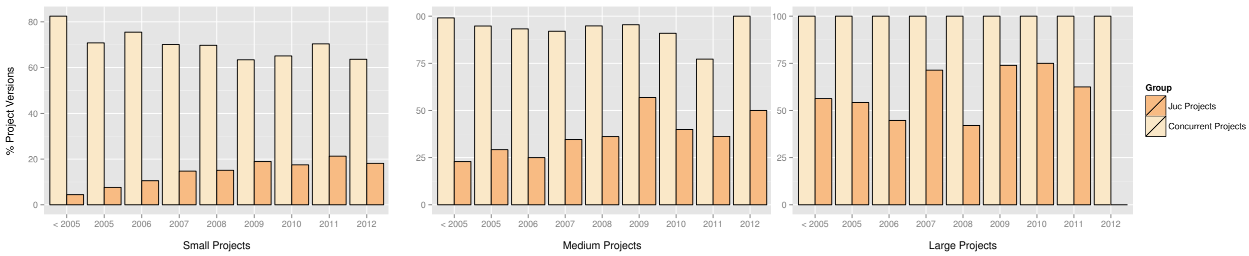

Figure 4. Percentage of concurrent project releases, with and without the use of `j.u.c.`, released between 2005 and 2012. Only the most recent versions of these projects were taken into account.

perform this kind of analysis for more than 2000 thousand projects, we established criteria to determine what it meant for a project to exhibit a large increase or decrease in the value of the metrics. The selected projects were those that had at least two releases and an increment of over 50% in a given metric between these releases. In addition, the highest absolute (not normalized) value of the metric should be higher than the value of the third quartile of all projects in that metric. In this manner, we can avoid having to examine projects with small values for the metrics, e.g., a project that had one `synchronized` block in the first release and two in the following one. By applying these criteria, we selected a total of 55 projects for manual inspection. Table 9 shows the metrics that exhibited large increases or decreases and the number of projects it was observed. Projects may have more than one metric involved in large changes.

We used the following methodology to guide our manual analysis of the code. First, we analyzed the source code of each project release searching for one of these string patterns: "AtomicBoolean", "AtomicInteger", "ConcurrentHashMap", "Executor", "ExecutorService", "ScheduledExecutorService", "extends Threads", "implements Runnable", "import j.u.c", "notify()", "notifyAll()", "synchronized", "synchronized methods", "volatile", and "wait()". These patterns were chosen because all the large changes involved the corresponding metrics. We compared the source code modifications in the newer release against the older one. We also analyzed bug reports and the on-line release notes of the projects, whenever they were available.

We identified three main reasons for dramatic changes in the values of the metrics: refactoring (30 instances), new features (30 instances), and testing/sample code (6 instances). Table 14 describes the categories and the number of occurrences for each one of them. A refactoring occurs when, for instance, the project includes a new use of a `synchronized` (method or block) or replaces a built-in type by the corresponding atomic data type. For example, in version 5.0.0 CR1, the JBOSS project was using a pre-j.u.c. library called `EDU.oswego.cs.dl.util.concurrent` and, in its 5.0.0 CR2 version, the project migrated to `java.util.concurrent`. Another interesting case is the Grinder project, which replaced its own concurrent programming library by `java.util.concurrent`. This fact is evidenced by a comment in the source code: "*This package should probably be deprecated in favor of JSR 166*". In another example of refactoring, the metric `implements Runnable` decreased after the refactoring in Stripper project, replaced by the scheduler implementation with threads.

In the second group, we observed that new functionalities are playing an important role in the use of concurrent

Table 9. Metrics involved in large changes.

Metric	Projects	How they changed
AtomicBoolean	2	All increased
AtomicInteger	4	All increased
Concurrent collections	1	All increased
Executors Contructs	2	All increased
extends Thread	5	All increased
implements Runnable	4	3 increased, 1 decreased
import j.u.c	2	All increased
notify()	1	All increased
notifyAll()	3	All increased
sync blocks	14	All increased
sync methods	17	16 increased, 1 decreased
volatile	7	All increased
wait()	1	All increased

Table 10. Thread creation projects metrics per 100KLoC by categories (small/medium/large projects, respectively), considering only concurrent projects.

Metrics	Median				Mean				%Concurrent Projects			
	S	M	L	ALL	S	M	L	ALL	S	M	L	ALL
# extends Runnable	19.32	2.38	0.80	2.29	19.78	4.87	1.45	7.43	1.14	2.41	10.15	2.49
# extends Thread	25.04	6.05	2.81	12.21	42.62	9.98	5.61	27.07	40.48	63.44	76.64	50.55
# Executors Constructs	20.92	4.07	1.79	5.16	34.82	13.51	3.68	18.99	6.12	14.65	32.48	11.14
# implements Runnable	27.29	7.18	3.73	14.42	46.85	12.21	7.54	31.31	47.10	66.03	86.80	56.99
# implements Future	27.17	1.63	0.76	1.60	23.73	2.21	1.21	4.76	0,24	1,89	5,58	1,27
# FutureTask	27.17	2.25	0.91	1.82	38.02	3.64	1.71	9.80	0.40	1.89	5.07	1.39

programming constructs. For instance, in the Choco project, we observed an increase of 1,800% in the number of `synchronized` methods when the project introduced a feature called geost, a generic geometrical kernel for handling polymorphic k-dimensional objects. In a similar case, the Hippopotams project increased the use of `synchronized` methods in more than 300% when it introduced a framework for building GUI for Java desktop applications. Additionally, we observed that some major changes involved projects that were using concurrent constructs inside the testing/sample code. These are the projects in the Testing/Sample code category. For instance, the project backport-util-concurrent version 1.1_01, the construct `extends Thread` was added to test classes, increasing concurrency behaviour.

The three aforementioned groups are not disjoint, i.e., a particular project may fit in two or more groups. For example, we observed that the TASSEL Project fits in two categories: refactoring and new features. Its 3.0-20110324 version refactored a pipeline feature to include multi-threaded behavior and, at the same time, it introduced a new module that used several concurrent constructs. Also, some projects do not make available information on bug reports or release notes for some versions. When analyzing these projects, we restricted our analysis to the source code. But, sometimes, only the source code is not enough to understand the reason behind the large change in a project. These projects are classified as no on-line information category.

Figure 5 presents a different perspective. It compares the usage of a given metric only in projects that has more than 3 versions. Then we compare the first against the last version, and the figure draws the percentage of increase in the usage of those metrics. We did not plot all metrics beacuse we consider only metrics that have at least five projects. We have observed most of the metrics present an increase of about 30% 40%, but some metrics have increase more, such as `implements Runnable` (49%) and `volatile` (56%). However, `synchronized blocks` and `synchronized methods` are the ones which present the highest increase (150% and 241%, respectively). The mean of increase was 44.36% (SD: 57%, 3rd quartile: 32.26%).

INICIO DE TEXTO ADICIONADO

In order to try to understand which features are used inside a thread call, we have analyzed the native code that is invoked in the method `run` of the 67 CONFERIR DADO projects concurrent project. In order to analyze what kind of code is being used inside the `run` methods, we used Wala[8], which is a set of Java libraries for static and dynamic program analysis for Java bytecode. To perform the analysis, we have selected a sample of 100 random concurrent projects that have their binary file available on the SourceForge page.

[8]http://wala.sourceforge.net/wiki/index.php/Main$_{page}$

Table 11. Atomic data type (ADT) projects metrics per 100KLoC by categories (small/medium/large projects, respectively), considering only concurrent projects.

Metrics	Median				Mean				%Concurrent Projects			
	S	M	L	ALL	S	M	L	ALL	S	M	L	ALL
# Atomic variables	22.89	8.48	1.56	8.25	48.46	17.87	8.29	22.51	3.26	10.68	29.94	8.53
# AtomicBoolean	15.90	3.01	1.53	3.16	22.41	6.74	3.69	8.69	1.30	5.68	16.24	4.41
# AtomicInteger	17.28	6.85	1.10	6.18	45.85	11.29	4.54	16.88	2.28	9.13	24.36	6.79
# AtomicLong	20.21	4.62	1.43	2.40	30.84	10.38	5.01	11.11	0.89	4.82	15.73	3.77

Table 12. Synchronization mechanisms projects metrics per 100KLoC by categories (small/medium/large projects, respectively), considering only concurrent projects.

Metrics	Median				Mean				%Concurrent Projects			
	S	M	L	ALL	S	M	L	ALL	S	M	L	ALL
# Condition	21.76	5.03	0.93	5.43	37.33	9.69	3.54	13.96	0.89	3.62	11.67	2.66
# CountDownLatch	17.63	6.42	1.53	6.40	27.01	15.04	9.08	15.95	1.38	4.31	12.69	3.48
# CyclicBarrier	9.24	3.11	1.13	2.55	16.84	10.71	2.42	9.35	0.32	2.37	4.85	1.26
# sync blocks	73.47	33.13	31.42	51.63	152.29	74.87	72.25	116.57	53.55	83.44	93.90	65.93
# sync methods	81.29	54.49	48.20	69.61	184.28	99.20	89.86	152.02	71.18	92.75	97.96	79.16
# notify()	26.26	5.83	2.41	7.89	41.66	10.86	5.44	21.01	14.36	37.24	53.29	24.60
# notifyAll()	23.32	6.42	2.77	9.44	50.56	11.67	7.47	25.33	15.91	40.86	62.43	27.27
# volatile	30.75	8.86	3.64	11.82	70.04	31.16	19.76	41.27	14.12	31.37	62.94	24.26
# wait()	25.36	7.22	3.97	11.02	46.86	14.21	11.18	27.93	27.75	57.75	72.08	40.22
# ReentrantLock	20.31	3.28	0.94	2.86	35.19	10.67	3.06	12.83	1.71	8.44	20.81	5.51
# ReentrantReadWriteLock	10.96	4.02	0.64	1.89	17.42	10.16	1.23	7.08	0.89	3.62	15.22	3.13

In this analysis, we selected projects that implement the `Runnable` interface or extends the `Thread` class. Also, we selected classes that have a super class that implements the `Runnable` interface or extends the `Thread` class.

Our analysis dived into the code that is invoked in the method `run()`. To navigate through the methods invokations we have built the callgraph using all public methods as entrypoints. The code sniped in Figure 1 shows an example of the analysis. The method `run()` call three others methods, the first one is a call for a method made by the programmer `methodA()` and the second and the third one are calls for a natives methods `sleep()` and `printStackTrace()`. The `methodA()` call for a native method `System.out.println()` and for `methodB()`. The `methodB()` only call for a native method. In this scenario our analysis reached the call made in `methodB`, but it could be deeper, ie. Our analysis was made to dive to up 10 call methods deep. We only count the number of native methods invokation, in this example, we count the methods `sleep()` and `printStackTrace()` once and the `println()` twice.

Listing 1. Example of code

```
public void run() {
        while(true) {
                this.methodA();
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}

public void methodA() {
        System.out.println(" ... ");
        this.methodB();
}

public void methodB() {
        System.out.println(" ... ");
}
```

Although all selected projects are concurrent, we found that 33 CONFERIR DADO projects do not have the method `run()`. Project delicious-1.14 is an example of such projects. Although concurrent, it does have a synchronized method but no `run()` method.

Table 13. Projects metrics per 100KLoC by categories (small/medium/large projects, respectively), for concurrent collections, considering only concurrent projects.

Metrics	Median				Mean				%Concurrent Projects			
	S	M	L	ALL	S	M	L	ALL	S	M	L	ALL
# Concurrent collections	22.82	4.51	1.60	6.90	48.02	11.15	6.44	23.49	6.77	15.51	34.51	12.01
# ConcurrentSkipListMap	11.80	2.57	0.66	2.01	11.80	2.57	0.65	4.78	0.16	0.33	1.45	0.34
# ConcurrentHashMap	15.49	6.36	2.86	6.61	52.50	12.27	5.53	23.43	4.08	10	27.91	7.89
# ConcurrentMap	39.95	1.67	1.07	1.44	39.95	1.81	1.25	4.44	0.08	1.01	3.88	0.74
# LinkedBlockingQueue	17.27	3.13	1.00	3.71	26.88	4.31	2.58	12.36	3.02	6.37	15.73	5.39
# PriorityBlockingQueue	22.47	1.79	0.86	2.47	34.22	1.82	0.77	15.06	0.40	0.67	1.45	0.68
# SynchronousQueue	9.04	2.95	0.50	2.32	8.87	3.13	1.48	3.51	0.32	1.35	4.36	1.10

Table 14. The categories that we found in our manual analysis.

Category	Occurrences
Refactorings	14
New Features	17
Refactoring + New Features	13
Testing/Sample code	3
Testing/Sample code + Refactorings	3
No on-line information	5

We also have organized the result in order to find which language constructs are the most used inside the `run` methods. Figure 6 shows the results. To get this result we have summed all invokation methods inside the `run` method. As we can see, the most used class is the `StringBuffer` and its most used method is `append()` which represents 30% of all `StringBuffer` method invokations.

The high number of `StringBuffer` method invokation may suggests that developers are aware that instances of `StringBuilder` are not safe for use by multiple threads. According to Java official documentation[9], if synchronization is required then `StringBuffer`, should be used instead.

Table 15. Top 10 most used classes / interfaces and the number of time of their most used methods.

Class / Interface	Method	Occurrences
java.lang.StringBuffer	append	797697
java.util.Map	get	502395
java.lang.String	length	79509
java.lang.StringBuilder	append	180129
java.lang.Character	isDigit	26945
java.util.ArrayList	add	44390
java.lang.Math	max	79325
java.lang.Object	getClass	39967
java.util.Iterator	hasNext	60404
java.lang.System	arraycopy	57712

Figure 7 shows the top 10 most used methods over all. Nine out of the top ten classes are related to string processing, and eight are `StringBuffer` methods, which may confirm that developers are aware about the synchronization.

We also investigated the usage of the `java.util.concurrent` package. The Table 16 shows all `java.util.concurrent.` classes and interfaces that were used in analyzed projects. The Table also shows the most used method of each class / interface.

The Figure 8 shows the top 10 most used methods of `j.u.c.` library. The most used method are `unlock()` and `lock()` from `ReentrantLock` class.

An interesting finding is that although the developers might be aware of the use of `StringBuffer` instead of others string related classes, they did not used any concurrent collection from `j.u.c.`. They still prefere to use `HashMap` or even `Hashtable` instead of `ConcurrentHashMap` as presented in Table 17. A closer look is need to confirm, but we think that developers are missing opportunities to make their projects faster and safer.

FIM DE TEXTO ADICIONADO

[9]http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

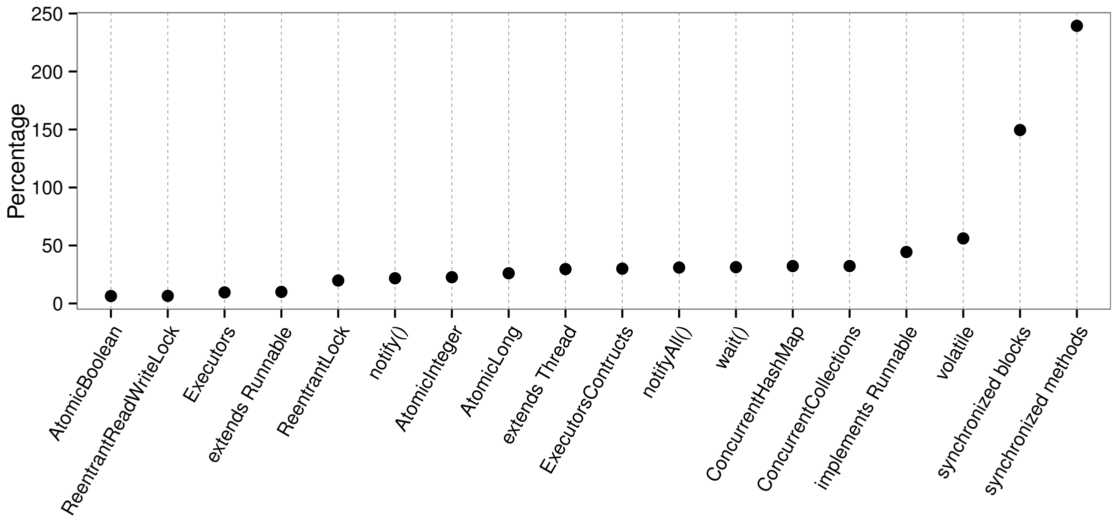

Figure 5. The difference between metrics usage when comparing the first and the last version.

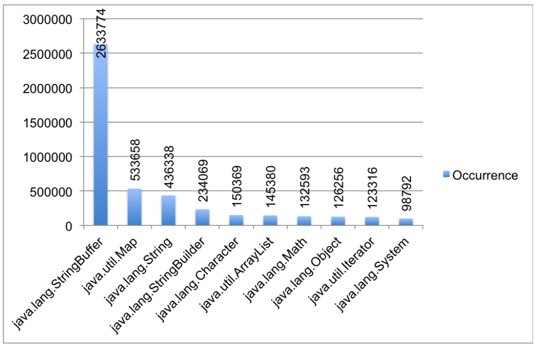

Figure 6. The top 10 most used classes

5.3. RQ3: How do developers protect shared variables from concurrent threads?

Most developers use `synchronized` blocks and methods to protect shared variables. The `volatile` modifier, explicit locks (including variations such as read-write locks), and atomic variables are less common. This is confirmed by the survey results: 57% of the respondents believe that the `synchronized` blocks and methods are the most frequently used concurrent programming construct, in comparison to the `volatile` modifier (only 1% of them) and explicit locks (less than 1% of them).

All concurrent projects we have analyzed include uses of `synchronized`, either as a method modifier or a block. This means that all the projects that use `java.util.concurrent` also use `synchronized`. The `synchronized` blocks are present in 53.56%, 83.38%, and 93.68% of concurrent small, medium, and large projects, respectively. The corresponding percentages for `synchronized` methods are 70.98%, 92.37%, and 97.57%, respectively.

Table 18 present correlations between deltas for some of the metrics we have collected. The correlations involving `synchronized` methods per 100KLoC are negative and non-negligible, except for occurrences of `synchronized` blocks. At the same time, the deltas for the latter correlate positively with the deltas for constructs whose usage has been growing in the last few years: atomic data types (a strong positive correlation of 0.5528598) and explicit locks (a moderate positive correlation of 0.4728947). We hypothesize that this trend stems from the need to improve performance of applications in multicore computers. In several situations, synchronizing an entire method could be considered a mistake. This is so because, when one synchronizes an entire method, code regions that often do not need

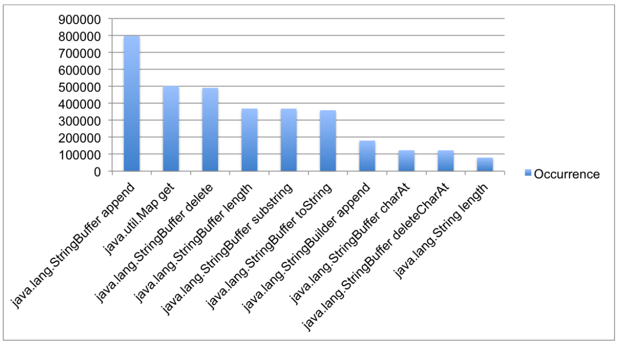

Figure 7. The top 10 most used methods over all

Table 16. All `j.u.c.` classes / interfaces used in analyzed projects ordered by usage and their most used methods.

Class / Interface	Occurrence	Method	Occurrence
j.u.c.locks.ReentrantLock	1491	unlock	1035
j.u.c.locks.AbstractQueuedSynchronizer	192	tryRelease / unparkSuccessor	71
j.u.c.atomic.AtomicInteger	173	get	83
j.u.c.atomic.AtomicReference	111	get	98
j.u.c.atomic.AtomicBoolean	72	compareAndSet	50
j.u.c.CopyOnWriteArrayList	50	size . interator	24
j.u.c.Semaphore	14	release	10
j.u.c.atomic.AtomicLong	6	set	6
j.u.c.ExecutorService	5	submit	2
j.u.c.locks.Lock	5	unlock	4
j.u.c.ScheduledThreadPoolExecutor	3	triggerTime / decorateTask / delayedExecute	1
j.u.c.LinkedBlockingQueue	3	take	2
j.u.c.Executor	2	execute	2
j.u.c.Executors	2	newFixedThreadPool / newSingleThreadExecutor	1
j.u.c.Future	2	get	2

to be synchronized are also locked. A `synchronized` block can be used instead to implement finer-grained locking. One of the anonymous respondents has described that s/he had improved the performance of the application just by *"removing synchronized bottlenecks with lock free code or finer-granularity locking"*. The `synchronized` keyword is favored by developers for mutual exclusion. It can also be noted that `synchronized` methods are employed more intensively than `synchronized` blocks. In Figure 10, we take the size group out of consideration, like we did in Figure 3.

Another interesting point in Figure 9 pertains to the `volatile` keyword. Notice that projects have employed it intensively, though nowhere near the pervasiveness of `synchronized` blocks and methods. Nonetheless, on average, the use of `volatile` variables has not changed significantly throughout the years. Even though `volatile` variables can be read and written atomically, with better performance than regular variables accessed by means of `synchronized` blocks and methods or atomic variables and also require less coding effort, the `volatile` modifier cannot be used to solve the problem of achieving mutual exclusion and, therefore, has more limited applicability. In fact, we found a weak positive correlation (0.2765276) between the deltas for `volatile` variables and `synchronized` blocks.

The `java.util.concurrent` library also provides constructs to protect shared data, such as atomic variables and the `Lock` interface. In Figure 10, Atomic Variables refer to classes of the `java.util.concurrent.atomic` package, such as `AtomicBoolean`, `AtomicInteger`, and `AtomicLong`. Although the numbers pertaining to the use of atomic variables do not seem to be large (Figure 9), a single occurrence of an `AtomicInteger` can replace many uses of `synchronized` blocks or methods. Atomic variables can be employed as a replacement for `synchronized` blocks and methods in a number of situations, while providing a non-blocking solution that completely avoids deadlocks. Nonetheless, it was possible to identify a moderate/strong positive correlation (0.5528598) between the deltas for `synchronized` blocks and atomic variables, as presented in Table 18. As for `synchronized` methods, there was no correlation. These results indicate that atomic variables are not replacing `synchronized` methods or `synchronized` blocks in applications. Instead, they suggest that atomic variables are more of a complement than a replacement

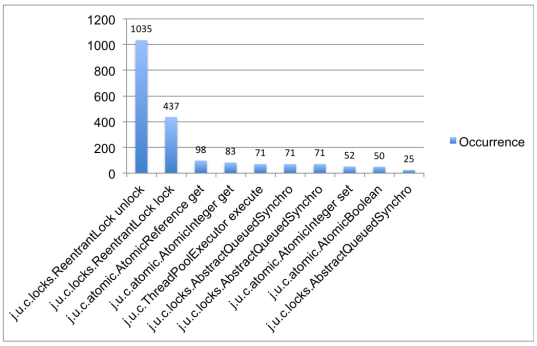

Figure 8. The top 10 most used `j.u.c.` methods

Table 17. The number os occurrence of HashMap and Hashtable and their most used methods

Hash	Occurrence	Method	Occurrence
HashMap	64440	clone	21040
Hashtable	7587	get	4021

for `synchronized` methods or `synchronized` blocks. In addition, we found a weak/moderate positive correlation (0.3631938) between the deltas for `volatile` and atomic variables. They exhibit similar properties; atomic classes have `get` and `set` methods that work like reads and writes on volatile variables.

One possible reason for the weak adoption of atomic variables is the fact that they have limitations, when compared to `synchronized` blocks and methods. The former are easier to use in situations where they are applicable, but they are also less general. Firstly because they only cover integers, booleans, and arrays of these types. Additionally, if two or more shared variables must be accessed by multiple threads, atomic variables cannot be used, except as lock replacements that require fairly complex non-blocking algorithms [16].

In some projects, explicit locks, in general, and the `ReentrantLock` class, in particular, seem to be replacing uses of `synchronized` methods. However it was not possible to find a correlation between the deltas for `synchronized` methods and uses of the `java.util.concurrent.locks` package, nor between `synchronized` methods and uses of the `ReentrantLock` class. It was possible to identify a moderate positive correlation (0.4728947) between the deltas for `synchronized` blocks and the `ReentrantLock` class and a week positive correlation (0.154635) between the deltas for `synchronized` blocks and the uses of the `java.util.concurrent.locks` package. These results indicate that explicit locks, as atomic variables, are more of a complement than a replacement for `synchronized` methods and `synchronized` blocks – they are more flexible (tryLock, non-scoped, multiple conditions). If we consider the overall number of concurrent projects, less than 9% employ atomic variables and 9.53% use the `Locks` from `java.util.concurrent` library.

5.4. RQ4: Do developers still use the `java.lang.Thread` class to create and manage threads?

We found out that developers are still using `java.lang.Thread` intensively. At the same time, they are also making more use of the high-level library to create and manage threads. Figure 11 gives a temporal perspective of the most often used constructs for thread management. Apparently, classes that directly extend the `Thread` class seem to be losing some space to the combination of executors and classes that implement the `Runnable` interface. Moreover, the use of the `Runnable` interface has increased over time, albeit slightly, and implementing it remains the standard way of defining new thread classes. However, as we can see in the bars for 2011, some projects use `Thread` more extensively than `Runnable`. In 2011 we found 2 projects that use `Thread` at least 3 times more than the mean (43.46 uses). They are rssamantha (148.50 uses) and jnrpe (204.22 uses). Without these two projects, the mean values for `Thread` would be roughly similar to the mean values for `Runnable` construct.

Figure 11 shows that `Executors` have been used more intensively in the last few years and Figure 12 presents its usage year by year. Nevertheless, we have examined several constructs of the executors family and still only a minority of the projects use them. Executors are used mainly by means of the `java.util.concurrent.Executors`

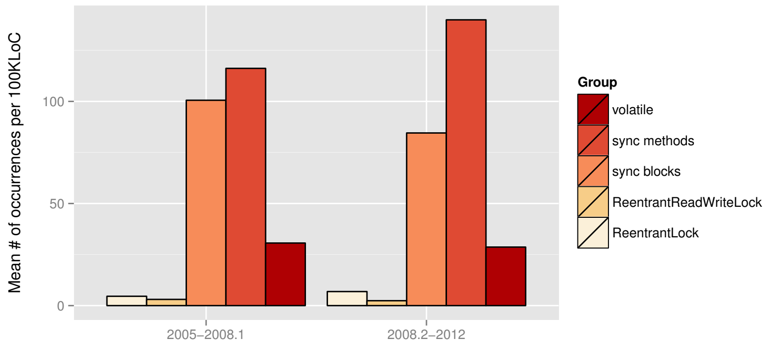

Figure 9. A temporal perspective of the most common synchronization mechanisms. Both groups contains only the most recent version in each time frame.

Table 18. Pearson correlation between synchronization types

Metrics	Projects	Correlation	mean deltas
Atomic variables + Concurrent collections	99	0.3786147	3.493009 x 5.320464
ReentrantLock + sync methods	78	-0.001103894	-0.4336 x 7.5729
ReentrantLock + sync blocks	78	0.4728947	-0.3693 x -21.75982
sync blocks + Atomic variables	128	0.5528598	-7.724 x 3.633
sync methods + Atomic variables	122	0.02918461	7.9887 x 4.493
sync blocks + java.util.concurrent	295	0.2477545	-9.5175 x 5.3835
sync methods + java.util.concurrent	295	0.01489112	-13.1194 x 7.4551
sync blocks + juc.Locks	131	0.154635	-27.9565 x 3.217
sync methods + Juc.Locks	132	-0.0478107	-0.1152 x 2.889
sync methods + sync blocks	751	0.1499019	-21.47 x -4.217
volatile + sync blocks	308	0.2765276	4.3353 x -6.214
volatile + java.util.concurrent	185	0.08287506	11.4736 x 8.914
volatile + Atomic variables	101	0.3631938	6.2826 x 6.562

class, which is a factory that provides the fundamental mechanisms to spawn threads whose lifecycle is managed by executors. However, this class is used in slightly more than 9% of all the concurrent projects (156 concurrent projects). Only a small number of projects directly employ the `Executor` and `ExecutorService` interfaces (32 and 9 concurrent projects, respectively). `Callable`, which is a useful interface, similar to `Runnable`, except for the fact that it represents computations that can produce a value as their result, has appeared in 68 concurrent projects, i.e. 3,9% of all concurrent projects. Although programmers can use `Callable` for non-concurrent purposes, it is closely associated with the use of executors, as suggested by Table 19. Furthermore, futures, a well-known mechanism for concurrent execution, are not used frequently. They are employed in only 1.27% of the concurrent projects. In constrast, 10% of the survey respondents believe that futures are the most often used construct of the `java.util.concurrent` library.

Table 19 shows a weak negative correlation between the deltas for `extends Thread` and `Executors` construct (-0.137388). This negative correlation might suggest that developers are moving from manually managing thread execution to employing the thread management policies that executors implement. This seems resonates with the intuition of developers – more than 30% of them believe executors or related types, such as `Future`, to be the most frequently used constructs of the `java.util.concurrent` library. Developers may have a great advantage when using executors. First, because they are easy to use. Second, because they provide various classes supporting advanced strategies to manage thread lifecycle, such as scheduled execution and thread pools. Finally, because they can improve performance by avoiding unnecessary thread creation. We have found 5 anonymous respondents who commented that they have used `Executors` to improve application performance. One of them has stated the following: "*Add concurrency in certain areas, improve concurrency control using executors instead of manually starting threads, make better use of multiple CPU cores, etc*".

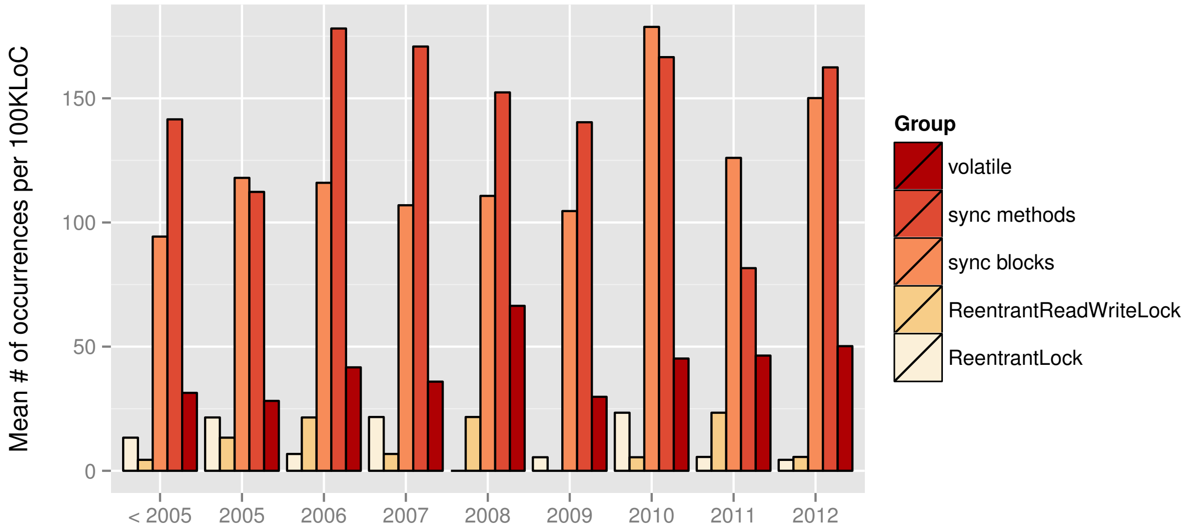

Figure 10. The mean # of uses of the most common synchronization mechanisms per 100KLoC in projects released between 2005 and 2012. Only the most recent version of each project was taken into consideration.

Table 19. Pearson correlation between Thread and Executors.

Metrics	Projects	Correlation	mean deltas
extends Thread + Executor constructs	112	-0.137388	-3.3756 x -2.922
implements Runnable + Executor constructs	145	0.01767673	-3.779 x -0.02007

5.5. *RQ5: Are developers using thread-safe data structures?*

Developers are still intensively using the older associative collections of the Java language, such as `Hashtable` and `HashMap`. Some factors make them unsuitable for highly concurrent applications [10]. Hashtable, albeit thread-safe, uses only a global lock in their methods, which makes it unscalable. HashMap, on the other hand, is not thread-safe.

We have examined several families of concurrent collections and most of them appear in few applications. `ConcurrentHashMap` and `LinkedBlockingQueue` are the only collections which present some significant use. They appear in 65.87% and 45.02% of the projects that employ concurrent collections, respectively, representing 7.98% and 5.45% of all concurrent projects, respectively. Moreover, 21% of the survey respondents agreed that `ConcurrentHashMap` is one of the most frequently used concurrent programming constructs of the Java language and 60.36% of them have used it in a Java project. In fact, they also believed that `ConcurrentMap` was frequently used. Nevertheless, since `ConcurrentMap` is an interface, it could be used frequently, albeit indirectly. One of the anonymous respondents have said that the use of `ConcurrentHashMap` has helped him to improve the application performance: "*java.util.HashMap is not thread-safe. And what's more worse [sic], it might end-up in an infinite loop. We usually simply replace it with a ConcurrentHashMap*". Moreover, only one respondent mentioned `ConcurrentSkipListMap` and another one has employed `PriorityBlockingQueue`. No respondent has mentioned any of the following collections: `LinkedBlockingQueue`, `LinkedBlockingDeque`, `LinkedTransferQueue` and `ConcurrentNabigableMap`. In our study, we found out that `LinkedBlockingQueue` is employed by 4.22% of the analyzed projects.

Figure 13 presents the mean number of uses per 100KLoC of some concurrent and non-concurrent collections. It shows that `HashMap` is used much more intensively than the other collections. `HashMap` is used six times more frequently than `Hashtable` by small projects, and more than 4 times more frequently by large projects. In addition, its usage has been steadily intensive throughout the years. In Figure 13 it is also possible to notice that `Hashtable` usage has been decreasing steadily and the numbers of uses of both `Hashmap` and `ConcurrentHashMap` have been increas-

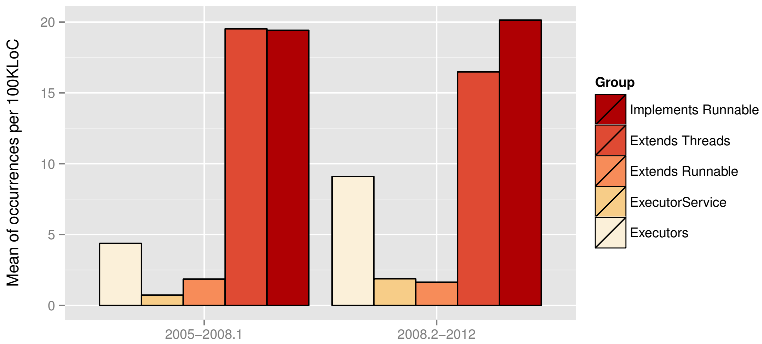

Figure 11. A temporal perspective of the most often used constructs for creating and managing threads. Both groups contains only the most recent version in each time frame.

Table 20. Pearson correlation between Collections and Concurrent Collections.

Metrics	Projects	Correlation	mean deltas
HashMap + ConcurrentHashMap	118	-0.4655915	9.313 x 6.4312
Hashtable + ConcurrentHashMap	78	-0.01810971	-14.03834 x 4.460697
Hashtable + HashMap	574	-0.1699068	-11.3671 x 10.169

ing over time. However, if we look at the most recent versions of projects released each year (Figure 14), we notice that the use of these constructs did not change much throughout the years. One exception is `PriorityBlockingQueue`. We observed a greate increase in the usage of this metric in 2010. Nonetheless, analyzing the data, we observed that only two projects were using this construct in 2010, one with 2.55 uses, and another one with 89.52 uses.

We have found a moderate negative correlation between the deltas for `HashMap` and `ConcurrentHashMap` (-0.4655915). Although we had analyzed 118 projects, this result seems to be biased by an outlier (the jade4spring project), which uses `ConcurrentHashMap` intensively and four times more than `HashMap`. Without this unique project, the result of the Pearson correlation would be much weaker (0.01158263). We also identified a negative correlation between the deltas for `Hashtable` and `HashMap` (-0.1699068). Considering the lack of scalability and bad performance of `Hashtable`, this result suggests that developers are wasting opportunities to improve the performance of their applications.

5.6. *RQ6: How often do developers employ condition-based synchronization?*

A large number of concurrent projects include invocations of `notify()`, `notifyAll()`, or `wait()`. The last one is the most popular among them, appearing in 705 projects (40% of the concurrent projects). Table 21 shows a moderate (0.6351526) and strong (0.97487) positive correlation between the deltas for the `wait()` and `notify()` methods and between `wait()` and `notifyAll()` respectively.

The `j.u.c` library provides high-level constructs for condition-based synchronization, such as `CyclicBarrier`, `CountDownLatch`, and the `BlockingQueue` interface and its implementations, e.g., `LinkedBlockingQueue`. Except for the latter, developers rarely employ these constructs. This data is confirmed by our survey: less than 1% of the respondents believe these constructs are frequently used. Nevertheless, the few projects that do use `CountDownLatch` seem to be using it to replace uses of `wait()` and `notify()`. We have noticed this upon manual examination of the projects. The correlations in Table 21 do not emphasize this because the number of projects is small. The correlations involving deltas for `CoundDownLatch` use too few samples to be relevant.

Classes such as `Condition` and `CyclicBarrier` are also rarely used: only 1.2% of the concurrent projects employ `CyclicBarrier` as we can see in Figures 16 and 15. In 2007, an outlier project called backport-util-concurrent-Java (100+ uses per 100KLoC) significantly impacted the `CyclicBarrier` data. We have calculated correlations using the delta for these constructs. However, we have not taken them into consideration due the low number of projects that use them. Table 21 presents the most interesting correlations.

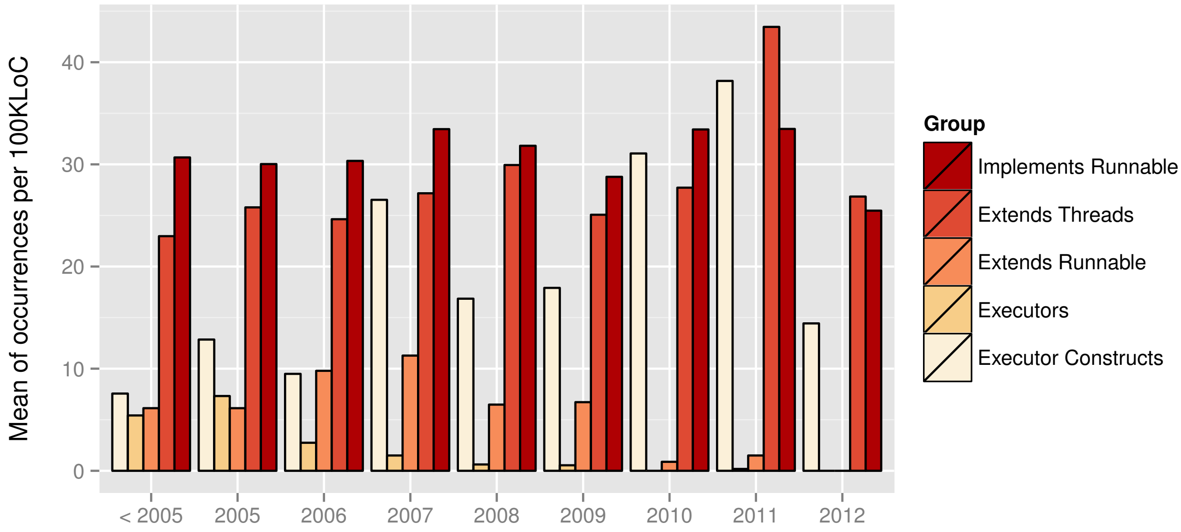

Figure 12. The mean # of uses of the most common mechanisms to create threads per 100KLoC in projects released between 2005 and 2012. Only the most recent versions of the projects were taken into consideration.

5.7. RQ7: Do developers worry about exceptions that might cause threads to end abruptly?

Developers do not worry about errors that might cause threads to end abruptly, at least that is what we conclude from our data. To gather this information, we have checked if there were project versions that implemented the `Thread.UncaughtExceptionHandler` interface. Implementations of this interface define handlers for uncaught exceptions thrown within a thread. Furthermore, we have also looked for invocations of the methods that associate these methods with a thread: `setUncaughtExceptionHandler` and `setDefaultUncaughtExceptionHandler` from `Thread` class. Less than 3% of the concurrent projects implement the `Thread.UncaughtExceptionHandler` interface.

In addition, we would like to know if developers are worried about abnormal thread death as a consequence of uncaught exceptions. When a single threaded console application terminates due to an uncaught exception, the program stops running and produces a stack trace that is different from typical program outputs. On the other hand, the death of the thread might have non-obvious consequences. Moreover, when threads have dependencies where one thread can only proceed when another one performs a certain action, the death of a thread causes the application to hang. Catching exceptions that threads throw is not only important because it can avoid these problems, but it is also encouraged by the Java official documentation[10]. In all these cases, if no handler catches the exception that caused the thread to die, finding the causes of the problem may be hard.

We observed that only a small amount of concurrent projects (46 projects), which represents 2.6% of the concurrent projects, implemented the `Thread.UncaughtExceptionHandler` interface once and only 0.22% of them (4 projects) implemented this interface four times, which is the highest number of implementations per project version. However, most of the handlers go against the official Java recommendation. We observed that most of the uses simply print a stacktrace or an error message. We also observed some naive implementations. For instance, a developer who implements this interface, but leaves the `uncaughtException` method empty, wasting the opportunity to treat the abnormal behaviour and to properly notify their clients. Also, comments in the code suggest the developers are worried about these exceptions but have no idea about what to do with them. Yet, no survey respondent have mentioned the use of this interface. Even though we did not have a question related to exception handling, the respondents were able to elaborate about it in our 9th question.

[10]http://docs.oracle.com/javase/7/docs/api/java/lang/ThreadDeath.html

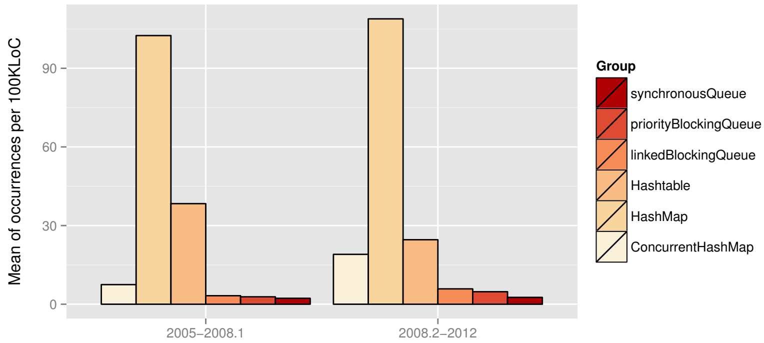

Figure 13. A temporal perspective of the most used collections. Both groups contains only the most recent version in each time frame.

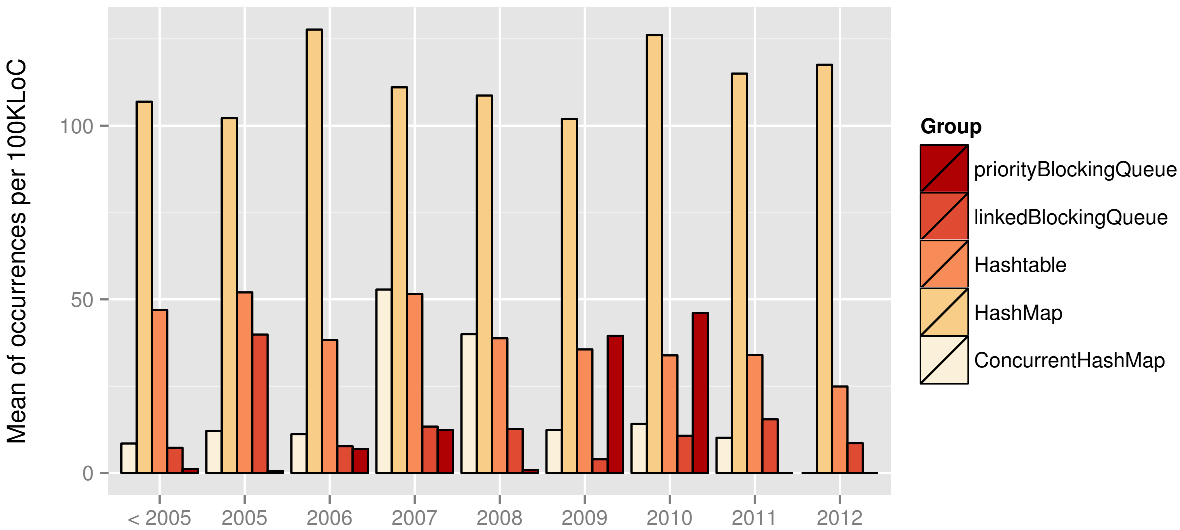

Figure 14. The mean # of uses of the most common concurrent collections per 100KLoC in projects released between 2005 and 2012. Only the most recent version of these projects were taken into consideration.

Table 21. Pearson correlation between condition based synchronization.

Metrics	Projects	Correlation	mean deltas
wait() + CountDownLatch	38	0.04138129	-5.1346 x 5.303660
notifyAll() + CountDownLatch	35	0.08461613	-1.1426 x 5.664033
wait() + notifyAll()	361	**0.97487**	3.0509 x 9.31662
wait() + notify()	333	0.6351526	-6.7499 x -5.7836
notify() + notifyAll()	201	-0.07556611	-8.5184 x -1.261

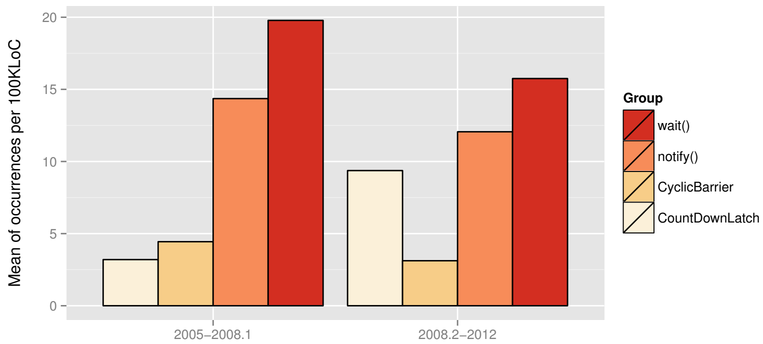

Figure 15. A temporal perspective of the most often used constructs for condition based synchronization. Both groups contains only the most recent version in each time frame.

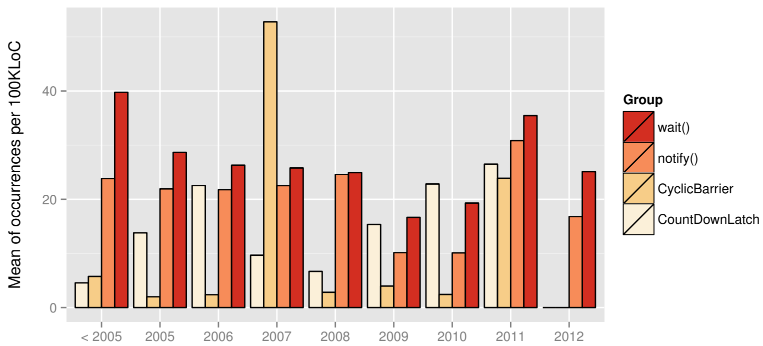

Figure 16. mean # of uses of most common condition-based synchronization mechanisms per 100KLoC in projects released between 2005 and 2012. Only the most recent version of these projects were taken into consideration.

6. Threats to Validity

In a study such as this, there are always many limitations and threats to validity. First, to download the source code of the projects, we assumed that the source files were packaged in a file with the keywords "src" or "source" in its name. This is common practice in open source repositories. Nonetheless, it is not a rule and some projects are bound to adopt different naming conventions. We have ignored such projects. Moreover, obtaining the release date of some project versions was not possible. In some cases, only the latest versions were dated, so some undated versions were ignored in order to collect data regarding temporal analysis. Furthermore, we assumed that most projects would contain either versions or subprojects in each directory. However, a small number of projects contain both in the same directory. It is difficult to infer this automatically if no conventions are followed or if the conventions are unknown. Hence, it is possible that some of the subprojects were analyzed as versions of the main project and that some versions were analyzed as subprojects. To minimize this problem, we manually examined the directory structure of projects that have multiple versions and projects that include subprojects. It is important to emphasize that previous studies with similar scope [6] did not address this issue and, as a consequence, may exhibit similar or higher bias.

Accuracy of measurement represents another threat to validity. Due to the large number of complex projects, it is impossible to automatically resolve all the dependencies on external libraries. Thus, we must rely on purely syntactic analysis. This is sufficient to precisely measure occurrences of `synchronized` and uses of monitor-based synchronization. However, to accurately collect some of the metrics, type information is necessary. Moveover, in cases where a class extends another class that extends `Thread` or implements `Runnable`, our infrastructure only counts the metric "extends Thread" or "implements Runnable" once, without looking at subclasses of the extending/implementing classes.

To verify whether the measurement error is significant, we have adopted the following procedure: we looked for class names that are common in the `java.util.concurrent` package, such as `ConcurrentHashMap`, `Executors` and `AtomicIntegers`, and for classes that extend `Thread` or implement `Runnable` which are using a fully-qualified name that does not refer to the `java.lang` package. This would indicate that the programmers are explicitly using their own `Thread` implementations, which are strong false positive candidates. We then proceeded to manually analyze all the candidate files to check whether they represented actual false positives. In general, measurement error was low, as presented in Table 22.

The calculation of the correlations to better understand the evolution of the projects was affected by the low number of project versions. Approximately 1/3 of the concurrent projects have only one version. This is a problem because at least two project versions are required to calculate correlations. This restriction greatly reduced the number of projects available for analysis. The small number of projects that employ the j.u.c. library further aggravated the problem. However, the chosen approach has produced results that we believe are more reliable than might have been achieved if we had adopted a more liberal approach.

In spite of the size of this study, it could also be argued that its results only apply to a very specific population: that of Java open source projects hosted by SourceForge. It does not cover other popular programming languages, such as C and C++, and does not study other source code repositories, e.g., Github and Google Code. Furthermore, it does not analyze proprietary software systems, whose source code is not available. However, even though the results of this work might not be generalizable, we believe that the focus on the Java language is not a problem to an experienced programmer, since it is one of the most popular programming languages according to a number of different sources and considering different measures of popularity [17, 18, 19, 20]. Moreover, SourceForge hosts an enormous number of projects [21], including large-scale, well-known, active ones, such as the Liferay Portal, jEdit, and the JBoss application server. Hence, it is representative of the current practice of open source Java software development. As for proprietary applications, they are outside the scope of this study.

Another limitation is that the Java programming language has a number of external libraries, and several of them are related to concurrent/parallel programming [22, 23]. In fact, the `java.util.concurrent` package was an external library for a long time, until the Java Community Process[11] included it in version 1.5 of the language. Understanding the use of these third-party libraries is outside the scope of this study. We also do not analyze the use of constructs added in version 1.7 of the language, which include all classes related to the Fork/Join framework, new concurrent collections (`ConcurrentSkipListMap` and `ConcurrentSkipListSet`), and a few other classes.

[11] http://www.jcp.org/

Table 22. The total of the false positive metrics.

Metric	Projects	% of false positives
java.lang.Thread	12	1,38%
java.lang.Runnable	19	1,86%
Executor Classes	2	1,05%
Atomic Variables	0	0%
Concurrent Collections	0	0%

Using the first and the last version of each project, we can employ statistical correlation to understand the evolution in the use of `java.util.concurrent`. We have also tried to analyze short periodic snapshots (quarterly, semesterly, or even yearly). Nevertheless, several issues have hindered this analysis and no interesting finding was produced. Firstly, about 70% of the concurrent projects do not have more than 3 versions released between 2005 and 2012. Thus, to proceed with the analysis considering shorter periodic snapshots, but with a small number of versions released to fill in these snapshots, would produce meaningless data. Moreover, if we restricted the study to projects with more than 3 releases, it would not be useful to a large study like this one, because it would drastically reduce our population of software systems.

Additionally, we have tried to analyze the snapshots from the source code repository of each project, instead of the releases, since they are much more frequent. However, conducting that kind of study using the SourceForge population is infeasible. Projects in SourceForge employ different source control systems, hosted in different places, and without following any standard. We found cases where the source code under development was hosted in one repository (sometimes a private repository), but the releases were available at SourceForge. Even worse, there were some projects that did not have the default repository URL.Hence, obtaining the source code snapshots for all the projects requires obtaining the URLs for the repositories one by one, fetching the snapshots, which may require the use of multiple version control systems (for example, Mercurial, Git and SVN, to name a few), and manually organizing these snapshots, since there is no standard organization. The impossibility of quickly analyzing a large number of projects under these circumstances is what motivated us to focus on releases in the first place, because SourceForge makes them available through a standard Web interface that can be crawled. However, releases are much less frequent than repository snapshots, which highlights the difficulty of performing this fine-grained analysis.

Another issue is related to the code organization inside the repository. Most of the code is not standardized, i.e., some projects have several directories on the root dir, and the released code could be in any one of them. We also found testing and documentation code in the same folder as the core business code. In addition, we have found a mix of projects inside the same repository. For example, a project had several child projects and these projects were all on the same directory. In summary, it is not possible to automatically understand and extract the code related to the core business of these projects. On the other hand, project releases usually only contain the code related to the core of the project.

Finally, this paper does not address the problem of understanding whether these constructs are used correctly or appropriately. It is well-known that programmers often misuse concurrent programming constructs, which may result in bugs or deterioration in the application's performance. e.g., code runs sequentially instead of concurrently. Nevertheless, this work does not perform any static or runtime analysis nor similar techniques in order to investigate concurrent programming errors such as deadlocks or race conditions. Analyzing these characteristics of a program is a computationally intensive task that is still difficult to perform on such a large scale.

7. Study Implications

This research has implications for different kinds of stakeholders. Five of these possible groups are discussed below.

Developers: Developers are now facing the problem of developing concurrent applications with more frequency, while keeping cost as low as possible and quality as high as possible. The results of our study provide some assistance to these developers. First, by showing that concurrent programming is already in widespread use and that they cannot ignore it (RQ1). Second, by indicating that there are many opportunities to make applications capable of benefitting from multicore machines (RQ1, RQ2, RQ3, RQ4, RQ6). Uses of `synchronized` can often be replaced by more efficient and more flexible solutions, better suited for parallel execution [3, 5, 4]. Third, by suggesting, based on actual adoption, alternatives to Java's basic concurrent programming constructs and data structures in some common situations (RQ3, RQ4, RQ5). Fourth, by showing that some developers are already switching from lower level constructs to the ones available in the `java.util.concurrent` library (RQ2, RQ3, RQ6), at least in some projects. Finally, by raising awareness about uncaught exceptions in threads and the fact that most applications are vulnerable to them (RQ7).

API Designers: Our results (RQ2, RQ3, RQ5, RQ6) suggest that simple, general-purpose mechanisms that can be employed in many situations seem to be preferred by programmers. API designers should consider this carefully when devising new libraries. `LinkedBlockingQueue` is one of the most widely used concurrent collections and, not coincidentally, it makes it trivial to solve producer-consumer-like problems. Analogously, `CountDownLatch` can be used to replace `wait`/`notify` (and `notifyAll`) pairs with a simpler solution in most situations. Atomic variables may seem restrictive at first, since the `java.util.concurrent` library only supports thread-safe versions of three primitive types: `int`, `long`, and `boolean`. However, as reported elsewhere [24], these are the types of more than 30% of all the fields appearing in Java programs. Also, protected regions involving access to a single shared variable are in widespread use [25]. Hence, their applicability is wide. In addition, they are easy to use and simplify reasoning about program behavior [16]. On the other hand, explicit `Locks` are known to be more difficult to use than monitors, though more flexible [7]. Murphy-Hill et al. [26] had tried to predict some innovations in programming languages. They believe that programming languages have had influence over IDE features and vice-versa. They also believe that sites like GitHub and StackOverflow can influence the way programmers program. They think that studies like ours can influence programming language designers how to improve programming languages.

Researchers: Researchers can also benefit from our results. First, to the best of our knowledge, this is the first report on the current state of the practice of the usage of concurrent programming constructs in Java. Second, because it hints (RQ1-6) that general purpose solutions being devised by researchers [9, 8] help developers to build parallel applications have some potential for adoption. Third, it suggests there is much room for improving the ways in which exceptions in threads are addressed (RQ7), and that existing proposals [27] are not mere academic exercises, as it indicates that guidance on how to deal with exceptions is necessary. This can be a starting point for new empirical studies and for the design of new mechanisms for handling exceptions in multithreaded systems.

Tool Vendors: Vendors can evolve their tools to improve support for code refactorings. Since concurrent programming often leverages low-level constructs, and the most popular high-level concurrent library is still not in widespread use but has clear benefits, vendors might be willing to develop intelligent tools that would suggest which constructs developers should use in particular contexts, and would also perform code transformations to introduce these constructs. Our results also suggest (RQ4, RQ5, RQ6) that developers are willing use high-level constructs provided that they are easy to use.

Lecturers in Concurrent Programming: Based on the results of this study, lecturers can provide students with information about the most widely used concurrent programming constructs in the Java language. Moreover, they can use these results to tailor the subjects they teach, for example, discussing why some concurrent programming constructs are not used in practice (RQ1) or to better highlight the advantages of these constructs, so that they become more widespread.

8. Related Work

This section discusses related research.

Studies on Large Software Populations of Java Software. The first study that we know of to conduct an in-depth study of the structure of Java programs was made by Baxter et al. [28]. In their work they examined 56 open source projects. Many of the applications were chosen because they have been used in other studies [29, 30, 31]. Other applications were added because they were popular, i.e. frequently downloaded and actively developed open-source Java applications from various websites. They did not describe how these projects have been downloaded, whether automatically or manually. Nevertheless, due to the low number of projects, it would be straightforward to manually download them. The analysis was made from the bytecode generated by the Java compiler. This study did not analyze the usage of concurrent programming constructs. Collberg et al. [24] also analyzed Java bytecode, but in a population of hundreds of Java applications. They discovered, for example, that `int` is the type of approximately 25% of all the variables in Java programs.

Grechanik et al. [6] collected and analyzed data at the source code level of open source projects in large repositories. They described an infrastructure for conducting empirical research in source code artifacts and obtained insights into over 2,080 Java applications. While they randomly chose those java applications to study, we focus on mature, stable, and recently updated Java projects.

More recently, Meyerovich et al. [32] analyzed programming language adoption through the lenses of several characteristics, including large-scale statistics and programmer decisions. Some of their findings include: (i) popular languages are consistently popular across domains of use, and less-popular languages tend to have specic domains; and (ii) developers consider ease of use and exibility to be more important than correctness. Similarly, McDonnell et al. [33] studied the impact of API evolution on software ecosystems. They conducted an investigation on a set of Android APIs and applications using data from Github. They observed that Android API is evolving at a mean rate of 115 API updates per month. On the other hand, client adoption is not catching up with the pace of the evolution – about 28% of API references in client application are outdated.

None of the aforementioned papers analyzes the usage of concurrent programming constructs, focusing instead on different characteristics of Java programs. Therefore, we could say the results presented in this paper complement their results.

Studies Targeting Concurrent Software. Lu et al. [34] analyzed 105 randomly selected real-world concurrency bugs. In particular, they found out that 73% of the non-deadlock concurrency bugs were not fixed by a simple fix strategy or were incorrectly fixed. Li et al. [27] studied bug characteristics in open source software, including concurrency bugs. One of their findings is that, although concurrency bugs represent a small portion of bug reports, 55.5% of them cause hangs or crashes, which means they can cause more severe impact on systems than non-concurrency bugs.

These previous studies complement ours because they have examined the documentation of the processes that developers follow to build concurrent systems. On the other hand, our study investigates the products of these processes, the actual concurrent systems. This approach makes it harder to understand some phenomena, such as bugs and their manifestation, but makes it possible to analyze other features, such as the usage of language constructs and how it has evolved over time. In addition, we can work at a much larger scale, because we analyze artifacts that were written in a programming language.

Dig et al. [35] analyzed five open-source projects in order to find, among other things, the most common transformations to retrofit concurrency into sequential programs, and whether these transformations are random or belong to certain categories. They analyzed qualitatively and quantitatively the concurrency-related transformations. Some of their findings are that, in 73.9% of the cases, concurrency was successfully retrofitted in existing program elements; in 5.4% of the cases, concurrency was modified in existing elements; and, in 20.5% of the cases, it was designed into new program elements. Their findings suggest that programmers follow an orderly process where they focus on well defined objectives: to improve responsiveness, throughput, or scalability, or to fix concurrency errors. This Study complements ours because they studied the process of transforming sequential code for parallelism. However, we analyzed concurrent projects in order to find which concurrent constructors were used.

More recently, Sadowski and colleagues [36] examined the evolution of data races by analyzing samples of the committed code in two open source projects over a multi-year period. They identified how the data races in these programs change over time. To gather data from the source code, they performed dynamic analysis, which has no false positives, and so gives a lower bound to the number of races that exist at a particular revision. This study complements ours by focusing on bugs in concurrent and parallel programs at the source code level. However, due to the cost of the analysis that was performed, it examines a small number of systems.

Lin et al. [37] had analyzed 104 open-source Android applications in order to understand how AsyncTask (a high-level concurrent construct) is used by programmers, if it is misused and underused. The authors also presented Asynchronizer, a refactor tool to extract sequential code into concurrent one using `AsyncTask`.

Marinescu [38] also analyzed SourceForge, but she focus on MPI (Message Passing Interface), a concurrent programming construct used in the C programming language. One of the main different between her work and ours is that she have performed, an investigation regarding the complexity based on the LOC (lines of code) and CYCLO (cyclomatic complexity) of the methods of MPI based applications .

To the best of our knowledge, the study that is closest in nature to ours is the one conducted by Okur and Dig [11]. Their study worked on a smaller scale (655 projects) and their focus was on open-source applications that use Microsoft's new parallel libraries - Task Parallel Library (TPL) and Parallel Language Integrated Query (PLINQ). Both the characteristics of these libraries and practices of the developer community make it difficult to directly compare the results of these studies. With the assistance of VisualStudio, they resolved all project dependencies and, as a consequence, avoided measurement errors. However, as discussed in Section 6, measurement error was small in our study. On the other hand, their temporal analysis only covers two years of development, and they did not look for statistical correlations in their results. Also, the research questions the two studies attempt to answer are different, with few exceptions (RQ1 and RQ3 in our study). In particular, our RQ3 was inspired by their RQ4. However, the results reported in the two papers greatly differ in terms of both their nature and the depth with which they were studied.

Okur and collegues [39] studied how windows phone applications (WP) are using asynchronous programming. In this study, they analyzed over 1,300 WP apps, and observed that developers are (i) missing opportunities to use this framework and (ii) they are misusing the constructs, creating problems that might hurt performance and introduce deadlocks. Based on these facts, they proposed two refactoring tools able to (i) convert callback-based asynchronous code to use the asynchronous framework and (i) to find and correct common misues. More recently, Okur et al. [40] had downloaded 880 open-source concurrent C# applications from Github in order to understand, among others things, the level of parallel abstractions developers used, i.e. if they used high-level abstractions instead of low-level ones. They also presented two refactoring tools which could help developers to migrate from low-level parallel abstractions to higher-level abstractions.

In a preliminary paper [41], we have reported a few of the results that we discuss in this work, most of them related to RQ1. This previous paper did not attempt to analyze larger trends in the usage of concurrent programming constructs, such as whether projects that have more recent releases are more likely to use the `java.util.concurrent` library. Moreover, the evolution of the usage of the concurrent programming constructs was only discussed superficially. In sharp contrast, here, it was the subject matter of most of the research questions. In addition, this new effort attempted to statistically correlate the obtained data about the evolution in the usage of some concurrent programming constructs.

9. Conclusion

This paper presents an empirical study into a large-scale Java open source repository. We found out that developers employ mainly simple mutual exclusion constructs. These constructs are easy to understand (though difficult to reason about) and have been available in Java since its initial version, released more than 15 years ago. Almost 80% of the concurrent projects include at least one `synchronized` method. Still, less than 25% of the projects employ the abstractions implemented by the `java.util.concurrent` library. We have noticed a tendency, nonetheless, of growth in the use of this library. In particular, more active projects seem to be using this library more frequently than the less active ones, which suggests this percentage is a conservative lower bound.

The most frequently and intensively used mechanisms to protect shared variables from concurrent threads are `synchronized` blocks and methods. The `volatile` modifier, explicit locks (including variations such as read-write locks), and atomic data types are less common, albeit growing in popularity. Developers are still using `Hashtable` and `HashMap`, even though the former is thread-safe but inefficient and the latter is not thread-safe. Although almost 80% of the concurrent projects have employed `HashMap`, only 12.11% and 50.14% of these projects have used some concurrent collections and `Hashtable`, respectively. We found out that the `Runnable` interface is the most common approach to define new threads and that executors have been growing in popularity. We also found out that developers are apparently not worried about errors that might cause threads to end abruptly.

This study has revealed many opportunities for researchers working on program reestructuring approaches. We have identified that developers waste a large number of opportunities to use high level constructs for concurrent programming, in favor of lower-level, more error-prone constructs. This suggests that previous [3, 4, 42, 11, 39] and future work on the introduction of these high-level constructs in existing programs have fertile ground to work on. At the same time, it is important to point out that using the high-level constructs is often not feasible, because it would require a large amount of refactoring, which indicates yet another opportunity for future research.

In the future, we intend to investigate recent proposals [43] for automatically resolving dependencies in large-scale repositories. This will allow us to use type information in our study, which will support more interesting analyses to be conducted. We also intend to investigate the organization of concurrency code in the analyzed projects. Furthermore, we intend to assess the extent to which exception handling constructs complicate concurrent/parallel programming. Another interesting point of study is to analyze source code evolution in order to identify finer-grained modifications, which can tell us if programmers are using refactoring techniques in their applications. Finally, we plan to investigate additional source repositories, such as CodePlex and Github, as well as to investigate other programming languages, especially Scala, which runs on the JVM, has numerous constructs for concurrent/parallel programming, and has more than 18,000 projects at Github.

10. Acknowledgments

This work is supported by CAPES/Brazil, CNPq/Brazil (306619/2011- 3, 487549/2012-0 and 477139/2013-2), FACEPE/Brazil (APQ-1367-1.03/12) and INES (CNPq 573964/ 2008-4 and FACEPE APQ-1037-1.03/08). Any opinions expressed here are from the authors and do not necessarily reflect the views of the sponsors.

References

[1] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, Dr. Dobb's Journal 30 (3).
[2] D. Lea, The java.util.concurrent synchronizer framework, Sci. Comput. Program. 58 (3) (2005) 293–309.
[3] D. Dig, J. Marrero, M. D. Ernst, Refactoring sequential java code for concurrency via concurrent libraries, in: Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada, 2009, pp. 397–407.
[4] K. Ishizaki, S. Daijavad, T. Nakatani, Refactoring java programs using concurrent libraries, in: Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD '11, ACM, 2011, pp. 35–44.
[5] M. Schäfer, M. Sridharan, J. Dolby, F. Tip, Refactoring java programs for flexible locking, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 71–80.
[6] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, C. Ghezzi, An empirical investigation into a large-scale java open source code repository, in: Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement, Bolzano-Bozen, Italy, 2010.
[7] A. S. Tanenbaum, Modern operating systems (3. ed.), Pearson Education, 2008.
[8] S. Burckhardt, A. Baldassin, D. Leijen, Concurrent programming with revisions and isolation types, in: Proceedings of OOPSLA'2010, Reno, USA, 2010.
[9] J. Yi, C. Sadowski, C. Flanagan, Cooperative reasoning for preemptive execution, in: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming, PPoPP '11, ACM, New York, USA, 2011.
[10] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, Java Concurrency in Practice., Addison-Wesley, 2006.
[11] S. Okur, D. Dig, How do developers use parallel libraries, in: Proceedings of the 21st ACM SIGSOFT Symposium on Foundations of Software Engineering, 2012.
[12] B. A. Kitchenham, S. L. Pfleeger, Personal opinion surveys, in: F. Shull, J. Singer, D. I. K. Sjoberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer, London, 2008, pp. 63–92.
[13] R. Ihaka, R. Gentleman, R: A language for data analysis and graphics, Journal Of Computational And Graphical Statistics 5 (3) (1996) 299–314.
[14] E. Pearson, Karl pearson: an appreciation of some aspects of his life and work, Biometrika 28 (3/4) (1936) 193–257.
[15] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, in: Proceedings of the Workshop on Algorithms and Data Structures, 1989, pp. 437–449.
[16] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[17] Github, Top languages, https://github.com/languages, last checked: September 17th 2013. (2013).
[18] Ohloh, Tools: All languages, sorted by commits, http://www.ohloh.net/languages?query=&sort=commits, last checked: September 17th 2013. (2013).
[19] G. de Montmollin, The transparent language popularity index, http://lang-index.sourceforge.net/, last checked: September 17th 2013. (2013).
[20] StackOverflow, Tags, http://stackoverflow.com/tags, last checked: September 17th 2013. (2013).
[21] R. Dyer, H. Nguyen, H. Rajan, T. N. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, in: ICSE '13: 35th International Conference on Software Engineering, 2013.
[22] A. F. Ferrari, Jpvm: Network parallel computing in java, in: In ACM 1998 Workshop on Java for High-Performance Network Computing, 1997.
[23] B. O. Christiansen, P. R. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, D. Wu, Javelin: Internet-based parallel computing using java, Concurrency - Practice and Experience 9 (11) (1997) 1139–1160.
[24] C. S. Collberg, G. Myles, M. Stepp, An empirical study of java bytecode programs, Softw., Pract. Exper. 37 (6) (2007) 581–641.
[25] R. Xin, et al., An automation-assisted empirical study on lock usage for concurrent programs, in: Proc. of the 29th ICSM, 2013.
[26] E. Murphy-Hill, D. Grossman, How programming languages will co-evolve with software engineering: A bright decade ahead, in: Proceedings of the on Future of Software Engineering, FOSE 2014, ACM, New York, NY, USA, 2014, pp. 145–154. doi:10.1145/2593882.2593898. URL `http://doi.acm.org/10.1145/2593882.2593898`

[27] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, C. Zhai, Have things changed now? an empirical study of bug characteristics in modern open source software, in: Proceedings of the 1st workshop on Architectural and system support for improving software dependability, 2006, pp. 25–33.

[28] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, E. Tempero, Understanding the shape of java software, SIGPLAN Not. 41 (10) (2006) 397–412.

[29] J. Y. Gil, I. Maman, Micro patterns in java code, SIGPLAN Not. 40 (10) (2005) 97–116.

[30] C. Grothoff, J. Palsberg, J. Vitek, Encapsulating objects with confined types, ACM Trans. Program. Lang. Syst. 29 (6).
URL `http://doi.acm.org/10.1145/1286821.1286823`

[31] A. Potanin, J. Noble, M. Frean, R. Biddle, Scale-free geometry in OO programs, Commun. ACM 48 (5) (2005) 99–103.
URL `http://doi.acm.org/10.1145/1060710.1060716`

[32] L. A. Meyerovich, A. S. Rabkin, Empirical analysis of programming language adoption, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 1–18. doi:10.1145/2509136.2509515.
URL `http://doi.acm.org/10.1145/2509136.2509515`

[33] T. McDonnell, B. Ray, M. Kim, An empirical study of api stability and adoption in the android ecosystem, in: ICSM, 2013, pp. 70–79.

[34] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, SIGOPS Oper. Syst. Rev. 42 (2) (2008) 329–339.

[35] D. Dig, J. Marrero, M. D. Ernst, How do programs become more concurrent? a story of program transformations, Tech. Rep. MIT-CSAIL-TR-2008-053, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA (September 5, 2008).

[36] C. Sadowski, J. Yi, S. Kin, The evolution of data races, in: Proceedings of the The 9th Working Conference on Mining Software Repositories, ICSE '12, IEEE, Zurich, Switzerland, 2012.

[37] D. D. Yu Lin, Cosmin Radoi, Retrofitting concurrency for android applications through refactoring.

[38] C. Marinescu, An empirical investigation on MPI open source applications, in: 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014, 2014, p. 20. doi:10.1145/2601248.2601298.
URL `http://doi.acm.org/10.1145/2601248.2601298`

[39] S. Okur, D. L. Hartveld, D. Dig, A. van Deursen, A study and toolkit for asynchronous programming in c#, in: ICSE, 2014, pp. 1117–1127.

[40] S. Okur, C. Erdogan, D. Dig, Converting parallel code from low-level abstractions to higher-level abstractions, in: ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings, 2014, pp. 515–540.

[41] W. Torres, G. Pinto, B. Fernandes, J. a. P. Oliveira, F. A. Ximenes, F. Castor, Are java programmers transitioning to multicore?: a large scale study of java floss, in: Proceedings of the workshop on Transition to Multicore, SPLASH '11 Workshops, ACM, 2011, pp. 123–128.

[42] M. Schäfer, M. Sridharan, J. Dolby, F. Tip, Refactoring java programs for flexible locking, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, 2011, pp. 71–80.

[43] J. Ossher, S. K. Bajracharya, C. V. Lopes, Automated dependency resolution for open source software, in: Proceedings of the 7th International Working Conference on Mining Software Repositories, Cape Town, South Africa, 2010, pp. 130–140.