The Journal of Systems and Software 106 (2015) 59–81

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

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

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

ARTICLE INFO

Article history:
Received 14 October 2014
Revised 14 April 2015
Accepted 18 April 2015
Available online 24 April 2015

Keywords:
Java
Concurrency
Software evolution

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. The main findings of our study are: (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 100,000 LoC, 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.

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 (Sutter, 2005). 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 (Lea, 2005), `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 2227 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 highest number of projects in SourceForge.

Evidence on 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 (Dig et al., 2009; Ishizaki et al., 2011; Schäfer et al., 2011a).

* Corresponding author. Tel.: +55 9191563390.
E-mail addresses: ghlp@cin.ufpe.br, gustavohenrique.86@gmail.com (G. Pinto), wst@cin.ufpe.br (W. Torres), jbfan@cin.ufpe.br (B. Fernandes), castor@cin.ufpe.br (F. Castor), roberto@cin.ufpe.br (R.S.M. Barros).

http://dx.doi.org/10.1016/j.jss.2015.04.064
0164-1212/

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. (2010) 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).

1.1. RQ1: 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.

1.2. 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%.

1.3. 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.

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

We found out that 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.

1.5. 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.

1.6. 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 mechanisms with simple semantics like `CountDownLatch` have potential to, in some contexts, replace lower-level, more traditional ones.

1.7. RQ7: Do developers attempt to capture exceptions that might cause abrupt thread failure?

Our data indicates that less than 3% of the concurrent projects 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 indications 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 about 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

[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.