How Do Developers Use APIs? A Case Study in Concurrency

Stefan Blom
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
Enschede, Netherlands
Email: S.Blom@utwente.nl

Joseph Kiniry
Department of Informatics and Mathematical Modeling
Technical University of Denmark
Copenhagen, Denmark
Email: josr@itu.dk

Marieke Huisman
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
Enschede, Netherlands
Email: M.Huisman@utwente.nl

***Abstract*—With the omnipresent usage of APIs in software development, it has become important to analyse how the routines and functionalities of APIs are actually used. This information is in particular useful for API developers, to make decisions about future updates of the API. However, also for developers of static analysis and verification tools this information is highly important, because it indicates where and how to put the most efficient effort in annotating APIs, to make them usable for the static analysis and verification tools.**

This paper presents an analysis of the usage of the routines and functionalities of the Java concurrency library `java.util.concurrent`. It discusses the Histogram tool that we developed for this purpose, *i.e.*, to efficiently analyse a large collection of bytecode classes. The Histogram tool is used on a representative benchmark set, the Qualitas Corpus. The paper discusses the results of the analysis of this benchmark set in detail. This covers both an analysis of the important classes and methods used by the current releases of the benchmark collection, as well as an analysis of the time it took for the Java concurrency library to start being used in released software.

***Keywords*-API usage, java.util.concurrent, static analysis, Qualitas Corpus.**

I. Introduction

The widespread use of Application Programming Interfaces (APIs) has had a significant impact on programming. Typically, APIs implement many standard routines and data structures, and can shield the programmer from many implementation details. For example, the Java language includes an extensive standard API [1].

Because of the widespread and varied use of APIs, it is difficult to predict which routines and functionality of an API are used most often. However, there are many reasons why it is important to actually have this usage information. First of all, like all other software, APIs need maintenance. For API developers, knowing how heavily an API is used, and which functionality of the API is *actually* used, gives a good indication where to put effort during this maintenance. Spending much effort on optimising performance of a method that is never used might not be worth the effort, whereas improving performance or resource use of a method that is heavily used will be appreciated by many developers. Usage information also can help to predict impact of refactorings, and help to decide between different refactoring or deprecation options.

Knowledge about API use is not only useful for API developers, but also for developers of tools that increase software quality, such as tools for static analysis (*e.g.*, FindBugs [2] and PMD [3]); validation (*e.g.*, JMLunitNG [4] and KeYTestGen [5]), and program verification (*e.g.*, Clousot [6], Code Contracts [7], [8], various JML tools [9], [10], [11], KeY [12], ESC/Java [13], and VerCors [14]). To be used effectively, these tools require annotations of some or all of an application, including those API classes and methods that are actually used. Extending APIs with such annotations is a great deal of work without much reward. Therefore, usage information of APIs is important to do this work strategically, and to focus on those parts of the APIs that are most-widely used.

In this paper, we study the Java concurrency library (`java.concurrent.util`). This top-quality, highly scalable library provides many concurrency and synchronisation mechanisms (essentially as described in [15]). Our main concern is to find out which (parts of) classes and interfaces must be specified in order to make design by contract possible for realistic Java programs.

To do the analysis, we wrote the Histogram tool [16]. This tool has been developed to perform efficient, simple usage analyses of APIs for Java at the bytecode level. We applied the Histogram tool to a large corpus of roughly half a million Java classes of various sizes, in bytecode format (collected as the Qualitas Corpus benchmark set [17]). The classes are written by thousands of developers, are of production-quality, and are typically using multiple threads of execution. The results of this analysis, *i.e.*, an overview of usage frequency and patterns for different concurrency constructs, has been used to guide annotation writing of library classes for the VerCors (Verification of Concurrent Data Structures) project [14].

The results of the analysis show that it is important to consider several different ways to count the usage of classes. It also revealed that developers use concurrency construct

not always in the same way as they are taught in university, including using synchronisation mechanisms that are not in the standard textbooks, or using unexpected routines. Finally, the analysis also shows that the uptake of the concurrency library has not been as fast and complete as one might hope for.

The remainder of this paper is organised as follows. First of all, Section II discusses the Java concurrency library, what we want to know about its use and how the applications to be analysed are determined, and what are the important characteristics to ensure representativeness of such a set. Next, Section III discusses our way of determining the importance of classes and methods and the Histogram tool that we used to compute the raw data. Then Section IV shows the application of Histogram to determine the usage of the Java concurrency library on the set of example classes. Finally, Section V concludes and discusses related and future work.

II. Setup of Experiment

This section describes the basics for our experiment: it first gives an overview of the Java concurrency package, and then it introduces the benchmark collection that we use to obtain information about the use of the concurrency package: the Qualitas Corpus [17].

The Java Concurrency Package

The Java concurrency package has been added to the Java standard with the release of Java 5 on September 30, 2004. It is a standardisation of the concurrency package developed by Doug Lea [15], [18]. Since its release in Java 5, the package has been relatively stable. In both Java 6 and 7, the queueing functionality has been extended with new variants. Moreover in Java 7, the notion of fork-join task was added to the task-based computing framework.

The package provides the most important building blocks for developing concurrent programs:

- Implementations of *synchronisation primitives*, such as locks, including read-write locks, semaphores, count down latches, and barriers.
- The atomic classes, which are wrapper classes for *volatile variables*, supporting set, get and atomic compare-and-set operations.
- Implementations of typical concurrent *data structures*, such as concurrent maps and queues.
- The Executor framework, supporting *task-based parallelism*.

The synchronisation primitives are implemented on top of the *Synchronizer* framework, providing basis synchronisation mechanisms. Several different locking mechanisms are implemented, all implementing a common `Lock` interface. This interface declares methods `lock()` and `unlock`, allowing arbitrary locking patterns. Before introduction of this interface, locking in Java could only be done by using a `synchronized` code block.

Volatile variables can be used for lock-free synchronisation. Updates to volatile variables are immediately visible to other threads. Essentially, get and set-operations are updates and lookups of these volatile variable, whereas the compare-and-swap operation uses the hardware-specific CAS operator natively.

The `java.util.concurrent` API provides a wide range of common concurrent data structures, such as queues, maps, stacks etc. For many of these data structures different implementations are provided, providing different blocking behaviour (*e.g.*, lock-free implementations, fine-grained and coarse-grained locking). As an example, the multi-threaded version of the map `ConcurrentMap` extends the standard `Map` with atomic operations. Threads can atomically attempt to add, replace and remove mappings. For example, if an add operation succeeds, the thread knows that it was the first thread to add that particular key.

The basic task-based framework is based on the notion of tasks that run from start to finish without blocking. These tasks are submitted to an execution queue, that is backed by an unknown number of worker threads. The essential functionality of the framework is provided by the `Callable`, `ExecutorService`, and `Future` interfaces. In particular, the `Callable` interface encapsulates a computational task, which is queued for execution by calling an implementation of the `Executor` interface. To obtain the result of the task, the creator of the task obtains a reference to a `Future` object that was returned by the enqueue operation. When the task is done, its result will become available in this `Future` object.

In the initial version of the framework, tasks are not supposed to block. Therefore, it is unsafe for a task to wait for another task, which makes writing divide-and-conquer style algorithms awkward. This is fixed in Java 7, with the notion of a fork-join task which can not only spawn new tasks, but can also wait for the completion of the tasks that it submitted, without deadlocking.

The Qualitas Corpus

To analyse the use of Java's concurrency API, we need to derive statistics from a representative benchmark set. To avoid unfair bias, this should be a data set collected by somebody else. The Qualitas Corpus collection of software system has been collected exactly for purposes such as ours: its primary goal is to provide a resource that supports *reproducible* studies of software [17]. It contains open source Java projects, in source and bytecode format. Criteria for inclusion in the set are only based on technical conditions (such as being open source, written in Java), but not on software quality or purpose. This makes this collection suitable for the kind of analysis that we intend to do: it provides a good average sample of large software projects.

Table I
QUALITAS CORPUS SYSTEMS WITH EVOLUTION HISTORY.

Project	Description
ant	A Java library and command-line tool for supporting processes described in build files as targets and extension points dependent upon each other.
antlr	A parser-generator framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages.
argouml	ArgoUML diagram generator/data visualization.
azureus	A P2P file sharing client using the bittorrent protocol.
eclipse	An open extensible development platform.
freecol	Turn-based strategy game.
freemind	Mind-mapping tool.
hibernate	Projects allowing utilisation of POJO-style domain models.
jgraph	Graph management and visualisation.
jmeter	Application for measuring performance.
jung	An extendible language for the modelling, analysis, and visualization of data that can be represented as a graph or network
junit	Unit testing framework.
lucene	Indexing and search implementation.
weka	A collection of machine learning algorithms for data mining tasks.

Moreover, the Qualitas Corpus set also maintains old releases of the systems included in the benchmark set. This makes the collection also suitable for a historical analysis, to investigate the penetration of the Java concurrency library.

We used the 2012-04-01 version of the Qualitas Corpus set. It contains the most recent versions of 111 software projects, including 14 projects with 10 or more historic releases (see Table I for a short description of these systems). These historic releases go back in time to well before even the prototype of the concurrency framework. To give an idea of the size of the data set, the 111 most recent projects together add up to roughly half a million of Java classes and 1.3GB of bytecode, while the historic releases altogether add up to 5.5GB.

We analyse the collection of current releases to establish what are the most important and widely-used classes and methods in the concurrency library. Moreover, we looked at the historic series to study how fast the concurrency library was adopted by developers. The next section discusses the tool that we used to analyse the bytecode, and the main considerations underlying the analysis; Section IV then discusses the results of the analysis.

III. ANALYSIS TOOL CHAIN AND METHODOLOGY

This section provides the details of the analysis process. Essentially, we scan the bytecode of every project in the benchmark set. Every invokation of a method on an object of a certain class counts as one use of class, and also as one use of the method for that class. Once we have obtained the counting information, we pass (a subset of) the data into a spreadsheet and analyse the spreadsheet.

To support the counting process, we have developed a simple tool: the Histogram tool. We briefly describe its implementation. Then we argue that the analysis method of counting uses of classes and methods has its limitations, but still can provide useful information.

Histogram Tool Details

To count the method invokations, we developed the Histogram tool, publicly available via [16]. Histogram is a command-line tool that is given one or more projects as parameters. Given a project, it will search for bytecode files in that project. Specifically, given a directory it will find all `.class` files, and it will scan all sub-directories. The tool will also consider all `.class` files contained in archives, such as `.jar` and `.war` files. This approach mimics the behaviour of a JVM, which will scan directories and archives in its CLASSPATH.

Each of the bytecode files is opened using the ASM library [19] and scanned for method invokations. For each method invokation, depending on the settings either the class of the object on which the method invoked is counted, or the class/method pair is counted.

The result of the scan is a table whose columns are indexed with the projects and whose rows are indexed with classes or methods. The entries are the computed counts. The result is written to disk as a CSV file, which can then be further analysed using a standard spreadsheet tool.

The Relevance and Correctness of Invokation Counts

As mentioned above, the tool counts invokations of methods on objects. This might seem a rather trivial measure, but we claim that even though it has its limitations, it still provides useful information.

First of all, it should be observed that the analysis will answer the question whether a class is used at all much accurately, because if a class is used, its constructor has to be invoked. And constructor calls can only be hidden from our analysis if they either happen in an external library (just the Java standard library in our case) or if reflection is used. Moreover, the simplicity of the approach allows us to analyse very large collections of bytecode quickly, and thus to provide this information efficiently, and without any extra effort. Furthermore, even though the information is incomplete, the really important methods are likely to show up, and the differences in frequency still can be used to decide which method is more relevant.

To understand this, it is important to understand in which cases our analysis results are incomplete. As mentioned above, for each method invokation, the method call is counted as a call to the method of the static type of the object. Since Java uses dynamic dispatch in many cases this is not the method that will be executed. There is one specific case where this difference in result is especially relevant, namely when the static receiver type is not part of the concurrency library. In Fig. 1, we give an example of two situations in which a method invokation is counted for a different class than

Figure 1. Example with imperfect method counts.

```
import java.util.concurrent.locks.ReentrantLock;

public class LockDemo extends ReentrantLock {

  public static void main(String[] args) {
    Talker t=new LockedTalker();
    t.say("hello world");
  }

}

interface Talker {
  public void say(String sentence);
}

class LockedTalker extends ReentrantLock
                   implements Talker
{

  public void say(String sentence){
    lock();
    System.out.printf("%s%n",sentence);
    unlock();
  }
}
```

desired. First, the main program creates a `LockedTalker` but assigns it to a `Talker` variable. Thus the call to `LockedTalker.say` in the main class is counted as a call to `Talker.say`. Second, the `LockedTalker` class inherits from `ReentrantLock`, in order to be able to directly call the `lock()` and `unlock()` methods. But because those method calls occur in the body of method `say()`, they are counted as uses of the `LockedTalker.lock()` and `LockedTalker.unlock()` methods rather than as uses of the same methods in `ReentrantLock`. The end result is that no method calls to `ReentrantLock` are counted. Note however that there will always be a call to the constructor of `ReentrantLock` (from the constructor of `LockedTalker`), so it cannot happen that the use of this class is completely ignored.

As a real example of the first case, consider the `ConcurrentHashMap`, which extends `Map` from the Java collection API. If the static type of the receiver object is a `Map`, then any call to a method that was inherited (*e.g.*, put), will mean that this use of the concurrent hashmap will be uncounted.

The other case in the example was constructed, due to the fact that we observed that two projects did use a `ReentrantLock`, but never used the `lock()` or `unlock()` method. Detailed inspection of the two projects revealed that these cases were due to extension of the `ReentrantLock` class, thus hiding the use of the `lock()` and `unlock()` methods.

It is future work to combine the counting results of the Histogram tool with static analysis techniques for dynamic method call resolution (such as *e.g.*, Rapid Type Analysis [20]) to increase the precision of the analysis regarding the first effect. But these analyses can be expensive and must be incomplete, as the underlying problem is undecidable. In addition, if we extract the class hierarchy first then we can count every method invokation for every class for which the method is defined and thus also eliminate the second effect.

Finally, for our particular application domain, the annotation of API methods to be used in design by contract-based verification tools, the imprecisions in the analysis results often coincides with the methodological approach. In particular, it is the static receiver type that determines which contract is used for verification, and thus the counting information that we generate correctly indicates how often the contract will be used.

IV. Results of analyzing java.util.concurrent

This section discuss the results of our analysis, *i.e.*, how is the `java.util.concurrent` packaged used in the Qualitas Corpus benchmark set. As mentioned above, the results are based on the benchmark set with version number 2012-04-01, which contains 111 projects in total; our analysis revealed that 53 out of these 111 projects used the Java concurrency API. For comparison, the `java.lang.Thread` class is used in 102 projects.

First, we present the results of the analysis for class usage. We present the results in two different ways: first we sort by absolute number of references to the class; second we sort by the number of different projects that use the class.

Next, for two widely used classes – the `Lock` interface (most commonly used by absolute count) and the `ConcurrentHashMap` class (most commonly used by project count) – we discuss the results of the analysis for method usage. Finally, we discuss the data we obtained regarding the uptake of the concurrency package from analysing the systems with more than 10 systems in the Qualitas Corpus benchmark set.

The complete results of the analysis are available online at the download page of [16]. This allows the interested reader to inspect for example method usage information of classes of his or her own interest.

Most Often Used Classes

First, we discuss the results of the analysis of counting the references to `java.util.concurrent` classes in the current release variant of the Qualitas Corpus. We computed both the total number of references to each class in all projects and the number of projects referencing a class. The top 25 classes according to reference counts and project counts can be found in Table II and III, respectively.

It is important to compute the most important classes both by considering the absolute number of references and by considering the number of projects. The size difference between the smallest and the largest project is about 3 orders of magnitude. By looking at just the absolute number a big project that intensively uses a single class could make that class seem more important than it really is. By looking at

Table II
TOP 25 CLASSES BY REFERENCE COUNT

	refs	projs	class name
	3401	21	Lock
	2148	33	ConcurrentHashMap
	1826	17	ConcurrentMap
	1625	29	AtomicInteger
5	1596	22	AtomicLong
	1295	26	ReentrantLock
	1279	17	AtomicBoolean
	619	21	ReentrantReadWriteLock
	607	16	ReadWriteLock
10	551	26	CopyOnWriteArrayList
	544	23	Future
	502	20	ThreadPoolExecutor
	440	12	ConcurrentLinkedQueue
	429	11	AtomicReference
15	413	16	CountDownLatch
	393	13	ReentrantReadWriteLock$WriteLock
	371	27	ExecutorService
	338	16	BlockingQueue
	325	13	Condition
20	313	11	ReentrantReadWriteLock$ReadLock
	303	25	LinkedBlockingQueue
	234	18	TimeUnit
	213	8	Semaphore
	189	12	FutureTask
25	171	29	Executors

Table III
TOP 25 CLASSES BY PROJECT COUNT

	refs	projs	class name
	2148	33	ConcurrentHashMap
	1625	29	AtomicInteger
	171	29	Executors
	371	27	ExecutorService
5	1295	26	ReentrantLock
	551	26	CopyOnWriteArrayList
	303	25	LinkedBlockingQueue
	544	23	Future
	1596	22	AtomicLong
10	3401	21	Lock
	619	21	ReentrantReadWriteLock
	502	20	ThreadPoolExecutor
	234	18	TimeUnit
	1826	17	ConcurrentMap
15	1279	17	AtomicBoolean
	607	16	ReadWriteLock
	413	16	CountDownLatch
	338	16	BlockingQueue
	157	16	Executor
20	108	15	ExecutionException
	393	13	ReentrantReadWriteLock$WriteLock
	325	13	Condition
	86	13	CopyOnWriteArraySet
	82	13	ArrayBlockingQueue
25	440	12	ConcurrentLinkedQueue

just the number of projects, a group of tiny projects that all use a class once could make that class seem more important than it really is too.

What is also a factor is that certain classes, by design are used less often than others. For example, the `Executors` class has many factory methods. It is an important class, ranking number 3 in the project count top 25, yet it ranks as number 25 for total number of invokations. Similarly, the `Lock` interface ranks as number 10 on the project count, but is is the undisputed number 1 when considering absolute numbers.

There are a small handful of surprises that arise out of this objective, quantitative analysis.

Firstly, the `CountDownLatch` is far, far more popular than `CyclicBarrier`, even though the latter has a simpler semantics and is, in our experience, typically taught much more frequently in concurrency courses. Apparently, the differences in the signalling protocol, make the latch more important in practice.

Archetypical constructs available at the language level, like semaphores, various kind of queues, and futures, occur with very low frequencies. We have no working hypothesis for why this is the case beyond the suspicion that, as instruction in concurrency theory has fallen out of favour in universities, so too has appreciation for these basic constructs.

Finally, atomic *primitive* type wrappers are used enormously more often than atomic *reference* wrappers. We suspect this is due to the fact that developers rightly believe that reference updates are atomic, but forget that a test-and-set sequence on a reference is not and/or can behave in unexpected ways. (E.g. the double-checked locking problem [21].) Therefore, they mistakenly believe it is not necessary to use the atomic reference wrappers. Another explanation that we see is that `AtomicInteger` and other primitive wrappers are relatively easy to understand and to reason about in global invariants, so developers use them. But the major use of `AtomicReference` is in the design of lock-free algorithms, which are much harder to understand, and therefore probably avoided by many developers.

Most Important Methods

This section discusses the results of the method usage analysis for two widely-used classes: the `Lock` interface and the `ConcurrentHashMap`.

Lock-Related Methods: The `Lock` interface is the most widely used according to reference count, therefore we further analyse which methods are used. As it is difficult to look at an interface in isolation, we also consider the method counts for the cluster around this interface. Specifically, we looked at the `Condition`, `Lock` and `ReadWriteLock` interfaces and the `ReentrantLock` and `ReentrantReadWriteLock` classes.

The top 10 of methods, sorted by project counts can be found in Table IV. The top 10 by reference counts is not included because it only differs in the ranking of the constructors (as expected, objects are constructed less often than that they are used).

As expected, the lock/unlock methods are by far the most often used. Out of 6428 method invokations in total, the

Table IV
TOP 10 LOCK RELATED METHODS BY PROJECT COUNT

refs	projs	method name and signature
260	26	ReentrantLock.<init>() : void
2333	21	Lock.unlock() : void
956	21	Lock.lock() : void
80	20	ReentrantReadWriteLock.<init>() : void
568	16	ReentrantLock.unlock() : void
312	16	ReadWriteLock.readLock() : Lock
295	16	ReadWriteLock.writeLock() : Lock
286	16	ReentrantLock.lock() : void
252	12	ReentrantReadWriteLock$WriteLock.unlock() : void
122	12	ReentrantReadWriteLock$WriteLock.lock() : void

Table V
TOP 10 CONCURRENT MAP RELATED METHOD BY PROJECT COUNT

refs	projs	method name and signature
991	33	ConcurrentHashMap.<init>() : void
124	18	ConcurrentHashMap.<init>(int) : void
447	16	ConcurrentMap.get(Object) : Object
285	16	ConcurrentHashMap.get(Object) : Object
205	15	ConcurrentHashMap.put(Object,Object) : Object
182	14	ConcurrentMap.putIfAbsent(Object,Object) : Object
171	13	ConcurrentMap.put(Object,Object) : Object
59	12	ConcurrentHashMap.<init>(int,float,int) : void
59	12	ConcurrentHashMap.putIfAbsent(Object,Object) : Object
87	11	ConcurrentMap.clear() : void

various lock methods account for 1458 invokations and the various unlock methods for 3352 invokations. A good third of the projects using locks also use condition variables, which is not unexpected. What came as a surprise is that no less than 11 projects use `isHeldByCurrentThread`. This particular method is not covered at all in the theoretical standard API for locks, which consists of (variants of) lock/unlock plus wait/notify.

By taking a look at the available sources, we could find 10 uses of the `isHeldByCurrentThread` method. In 4 cases, the method is used in an assertion. In 3 other cases, the method is used to actually find out if a lock is held or not. Finally, there are 3 cases where the method is used inside what seems to be a utility library that is part of the project.

Considering our particular application domain, the annotation of API methods to be used in design by contract-based verification tools, this means that our specification techniques have to be able to describe the behaviour of this `isHeldByCurrentThread` method: design by contract verification can only be successful is all methods that are invoked have been specified.

ConcurrentMap-Related Methods: The most important artefact by project count (33) is the `ConcurrentMap` interface. This interface extends `java.util.Map`, is in turn extended by `ConcurrentNavigableMap`. The latter two interfaces are implemented by `ConcurrentHashMap` and `ConcurrentSkipListMap`, respectively. For the method analysis, we ignore `ConcurrentNavigableMap` and `ConcurrentSkipListMap` because they are used twice only.

The analysis of this interface is complicated by the fact that it extends the common `java.util.Map` interface from the collection API. Hence, as discussed above, there are cases where calls to the concurrent map are not counted as such, but counted as calls to the `Map` interface instead.

Table V shows the top 10 of concurrent map methods. Note that 33 projects create a concurrent map, while the first non-constructor method is used only in 16 projects. This suggests that the concurrent map is indeed used as a thread-safe replacement of the sequential map and not because of the extra functionality it offers. When looking at the usage information of the four atomic methods that form the difference between the `Map` and `ConcurrentMap` interfaces, we get similar indications. If we add up their uses in the concurrent map interface and implementation, we get:

method	count	projects
`putIfAbsent(key,value)`	241	19
`remove(key,value)`	68	12
`replace(key,value)`	13	8
`replace(key,old,new)`	35	8
total	357	21

Thus according to our counts, only 21 out of 33 projects use the new atomic methods, again indicating that `ConcurrentMap` is typically used as a thread-safe replacement of the sequential map.

Results

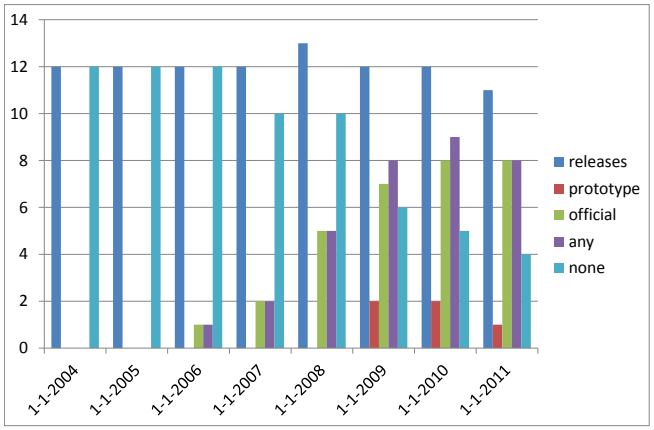
This section gives a list of the classes from the Java concurrency API that we consider the most important according to the results from our analysis. As a starting point, we merged the analysis results of Tables II and III. To complete the list, we also added their parent classes and implemented interfaces to our list. Finally, because the various atomic classes are highly similar, we added all of them instead of just the popular ones (because a change to or a specification for a method in one of these classes is easily carried over to the other atomic classes).

This list is relevant for application developers and specifiers. In particular, further developments of the `java.util.concurrent` API are best focussed on these classes, and also annotation-based verification methods should focus on annotating these classes first.

The resulting list is:

- From the *lock hierarchy*:
  Condition, Lock, ReentrantLock, ReadWriteLock, ReentrantReadWriteLock, ReentrantReadWriteLock$ WriteLock, and ReentrantReadWriteLock$ReadLock.
- From the *atomic variable classes*:
  AtomicInteger, AtomicLong, AtomicReference, and AtomicBoolean.

Figure 2. Concurrency use in releases during time period.



- From the *concurrent data structures*: ConcurrentHashMap, ConcurrentMap, BlockingQueue, ArrayBlockingQueue, LinkedBlockingQueue, ConcurrentLinkedQueue, and CopyOnWriteArrayList.
- From the *executor framework*: Executors, ExecutorService, Future, Executor, ThreadPoolExecutor, FutureTask, Callable, ExecutionException, RejectedExecutionException, and ThreadFactory.
- From the *remaining classes*: CountDownLatch and Semaphore.

To confirm the *relevance* of the classes on this list, we have counted the number of projects from the Qualitas Corpus set, whose use of concurrency classes is completely covered by classes on our list of most frequently used classes. This indicated that this list fully covers the concurrency-related aspects of 29 out of the 53 projects that use Java's concurrency API.

Adding a few more classes to the list can make a big difference. For example, the coverage is increased to 39 out of 53 by adding the following classes: TimeUnit, CopyOnWriteArraySet, ScheduledExecutorService, ScheduledThreadPoolExecutor, SynchronousQueue, ThreadPoolExecutor$AbortPolicy, and TimeoutException.

The Uptake of the Concurrency Library

The historic releases in the Qualitas Corpus allowed us to also look at the uptake of the framework over time. The analysis gives us a lot of data points, with a system name, a system version, a release date and the class counts. Unfortunately the releases are not in all cases linear, meaning that computing precisely how many systems use the concurrency library at a certain point in time is difficult. *E.g.*, if version 3.0 released in January uses the library, while version 2.8 released in February does not, should we say that the system does or does not use the library in March?

We chose to answer the question in both ways. That is, for each year we count the number of systems with at least one release that uses the library and we also count the number of systems with at least one release that does not use the library. Hence, when a system phases in the library in some year, the year before the transition it contributes 0 to the *using* count and 1 to the *non-using* count. During the transition year, it contributes 1 to both counts and after the transition year it contributes 1 to the *using* count and 0 to the *non-using* count.

Further, this counting is not just based on using and not using the library, but we also looked at several other criteria, including the use of of Doug Lea's prototype of the `java.util.concurrent` library and its variants.

Figure 2 shows the counts for the years 2004-2011 (before 2004 the concurrency library was not standardized and our data points end early in 2012). It displays the following five criteria:

criterium	Matches releases during the year that ...
releases	... merely happen,
prototype	... use one of the prototypes of the concurrency library,
official	... use the official release of the official concurrency library,
any	... use either variant of the concurrency library,
none	... use none of the variants of the concurrency library.

The picture shows us several things, which are not necessarily all surprising. For example, in each year there are typically two systems that do not release during the year, which means that there is more than a year between releases. Also, the first year in which the concurrency library is used is 2006, which is more than a full year after the release of the official version in September 2004 as part of Java 5. Neither of which is surprising, considering that development cycles of a year or two are not unusual.

What can be considered strange is that the prototypes show up during 2009-2011 only, instead of already being in use before 2004. The use of the prototypes seems to decline, but because our data ends early in 2012, this is guessing rather than fact.

What is clearly visible is that the releases that do not use the concurrency library have decreased over the years from all 12 to just 4 in 2012. From those 4, three are applications which do not need concurrency very much: freemind, argouml, and antlr. The one application that needs concurrency but does not use the concurrency library is ant. Assuming the sample is representative, this means that two-thirds of all systems released use the concurrency library in some way. Moreover, nearly all systems that need concurrency seem to use the library in one way or another.

V. Conclusion

Contributions/Summary

This paper describes how we have analysed the use of the classes and methods of the Java Concurrency library (`java.util.concurrent`). We have done this by implementing a tool that counts the number of method invokations in bytecode. The tool has been used on the Qualitas Corpus benchmark set as a collection of representative product-quality Java projects. Both the tool and the results of the analysis are available at the download page of [16].

We have presented and analysed the usage data obtained by the analysis. In addition, we have used the historic information in the Qualitas Corpus benchmark set to derive information about the uptake of the concurrency library for Java. Most results are as expected, but even so a few surprises show up, showing that developers do not always use concurrency mechanisms in the way they are taught in textbooks.

The data that is provided by the tool conclusively indicates whether a class is used at all. However, because of the difference between dynamic and static typing, the analysis is not precise enough to count all method usages. However, even though the data about method usage is incomplete, it still shows important trends about method usage. The results from the analysis can be used to determine where to put effort when maintaining and improving the library, it can provide information about the consequences of refactoring the library, or deprecating certain methods, and it can be used by developers of annotation-based analysis tools, to decide where to put the effort in annotating library classes. The results from the analysis have in particular been used to determine the focus on the Java concurrency library for the VerCors project [14].

Related Work

Peer-reviewed large-scale code analysis is relatively unusual given the volume of code created and available via various Open Source projects and websites. Within the realm of researchers focused on reasoning about programs, one would think that developers, as stakeholders, and code, as the primary artefact about which we reason, would hold the attention of researchers more.

Chalin has published two pieces of work whose results directly, objectively derived from developer surveys or code analysis, that directly impacted the semantics of the Java Modelling Language. In his ICSE paper he summarizes a developer survey which, in part, made the community realize that an assertion semantics based upon strong validity was necessary [22]. In the complementary paper by Chalin and James, large-scale static code analysis drove the community to decide to switch to a non-null reference default semantics to relieve annotation burden. Our work differs from his mainly in the sheer size and scope of our analysis, given he only analysed 700 KLOC of code, which is several orders of magnitude smaller than our sample.

Raemaekers et al.'s work is quite similar to ours because it focuses on answering a particular question, one that is perhaps presumed known in the folklore, through objective static analysis [23]. They analyse the contents of the Qualitas Corpus as well as nearly two hundred proprietary systems to which they have access. Their analysis is different from ours because it is syntactic, applied to source rather than bytecode, and is thus potentially erroneous and overcounts library use, since import statements do not mean that a class is actually used.

Rocha and Valente gather empirical evidence about the use of Java annotations by developers by statically analysing the Qualitas Corpus as well [24]. They attempted to use the JDK's `apt` tool to analyse source, but quickly found the tool not up to the task. They then developed a textual search program, which they do not describe in any detail. Consequently, they suffer the same flaws as Raemaekers et al.

Finally, Beckman et al. study nearly two millions lines of code to understand how object protocols are used in practice [25]. Their work is based upon a conservative static analysis of source code, based upon their own static analyser. Thus, their method is more similar to ours in both scale and method than any of the other aforementioned papers.

Threats to Validity

The main threat to validity for our work is the question of whether or not the code base we have analysed is truly representative of what developers write in practice. We are confident of our validity due to the size and scale of our analysis, the recognized validity of other published studies based upon the same corpus, and the fact that we precisely statically analyse bytecode. Originally we had prepared to analyse a much larger corpus, which included every Open Source system ever shipped by the Apache Foundation, IBM, Sun, and the Eclipse Foundation. We have kept this corpus in reserve, in case we need to do a larger-scale analysis in the future.

Future Work

Future work breaks down into two categories: further analyse the results of our analysis, and improve the tool and analysis method to obtain more detailed results.

In the first category, one of the open ends is the unexplained popularity of the `CountDownLatch`. The main purpose of this class is to synchronise between computations, which until Fork/Join tasks were added in Java 7, was not easily achieved between tasks. Therefore, we believe it will be interesting to see if the new Fork/Join tasks will take over market share from the old style tasks and/or the `CountDownLatch`. But given the uptake periods observed

before, it will take a few more years until this effect might become visible.

In the second category, an obvious improvement would be to make the counts produced by the tool more detailed. As mentioned above, the source of this incompleteness is that the tool counts the method as being invoked on the static receiver type of the object, instead of the dynamic type. It would be interesting to extract the subtype hierarchy and to count a method usage for every supertype that has the same method, in order to avoid artificially low counts. In addition, we could also integrate an efficient static analysis for method call resolution that allows us to consider the dynamic type of the receiver object in as many cases as possible.

In addition, the tool could also do the counting at different levels. With the analysis in this paper, we obtained an overview of which classes and methods are important to projects, but projects themselves are almost always a collection of sub-projects and it might be interesting to also look at that level. However, this would increase the number of data items by an order of magnitude, which requires that the analysis is supported by a database, and that more advanced queries can be made.

Finally, it would also be useful to extend the tool so that it can automatically give use and coverage information for a user-specified list of methods.

Acknowledgments

This work was supported by ERC grant 258405 for the VerCors project (Blom and Huisman), and Artemis grant 2008-100039 for the CHARTER project (Blom). We are grateful to the anonymous referees, who took the time to provide us with many useful detailed comments.

References

[1] K. Arnold and J. Gosling, *The Java Programming Language*. Addison-Wesley, 1996.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Using findbugs on production software," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 805–806.

[3] T. Copeland, *"PMD" Applied*. Centennial Books, Nov. 2005.

[4] D. M. Zimmerman and R. Nagmoti, "JMLUnit: The next generation." in *FoVeOOS*, ser. Lecture Notes in Computer Science, B. Beckert and C. Marché, Eds., vol. 6528. Springer, 2010, pp. 183–197.

[5] W. Ahrendt, W. Mostowski, and G. Paganelli, "Real-time java api specifications for high coverage test generation," in *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, 2012, pp. 145–154.

[6] M. Fähndrich and F. Logozzo, "Static contract checking with abstract interpretation," in *FoVeOOS*, ser. Lecture Notes in Computer Science, B. Beckert and C. Marché, Eds., vol. 6528. Springer, 2010, pp. 10–30.

[7] M. Fähndrich, "Static verification for code contracts," in *SAS*, ser. Lecture Notes in Computer Science, R. Cousot and M. Martel, Eds., vol. 6337. Springer, 2010, pp. 2–5.

[8] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *SAC*, S. Y. Shin, S. Ossowski, M. Schumacher, M. J. Palakal, and C.-C. Hung, Eds. ACM, 2010, pp. 2103–2110.

[9] D. R. Cok, "OpenJML: JML for Java 7 by extending OpenJDK," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science, M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617. Springer, 2011, pp. 472–479.

[10] A. Sarcar and Y. Cheon, "A new Eclipse-based JML compiler built using AST merging," in *Software Engineering (WCSE), 2010 Second World Congress on*, vol. 2, dec. 2010, pp. 287 –292.

[11] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *STTT*, vol. 7, no. 3, pp. 212–232, 2005.

[12] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNCS. Springer-Verlag, 2007, no. 4334.

[13] D. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system," in *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, ser. LNCS, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., vol. 3362. Springer-Verlag, 2005, pp. 108–128.

[14] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski, "The VerCors project: Setting up basecamp," in *Programming Languages meets Program Verification (PLPV 2012)*. ACM Press, 2012, pp. 71–82.

[15] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00. New York, NY, USA: ACM, 2000, pp. 36–43. [Online]. Available: http://doi.acm.org/10.1145/337449.337465

[16] The histogram tool. [Online]. Available: http://fmt.ewi.utwente.nl/tools/histogram

[17] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas Corpus: A curated collection of Java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345. [Online]. Available: http://qualitascorpus.com/

[18] D. Lea, "The java.util.concurrent synchronizer framework," *Sci. Comput. Program.*, vol. 58, no. 3, pp. 293–309, 2005.

[19] E. Bruneton, R. Lenglet, and T. Coupay, "ASM: a code manipulation tool to implement adaptable systems," in *Adaptable and extensible component systems*, November 2002.

[20] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *OOPSLA*, 1996, pp. 324–341.

[21] D. C. Schmidt and T. Harrison, "Double-checked locking," *Pattern languages of program design*, vol. 3, pp. 363–375, 1997.

[22] P. Chalin, "A sound assertion semantics for the dependable systems evolution verifying compiler," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 23–33.

[23] S. Raemaekers, A. van Deursen, and J. Visser, "An analysis of dependence on third-party libraries in open source and proprietary systems," in *Sixth International Workshop on Software Quality and Maintainability, SQM*, vol. 12, 2012.

[24] H. Rocha and M. T. Valente, "How annotations are used in Java: an empirical study," in *23rd International conference on software engineering and knowledge engineering (SEKE)*, 2011, pp. 426–431.

[25] N. Beckman, D. Kim, and J. Aldrich, "An empirical study of object protocols in the wild," *ECOOP 2011–Object-Oriented Programming*, pp. 2–26, 2011.

[26] B. Beckert and C. Marché, Eds., *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 6528. Springer, 2011.