in Proceedings of the 36th International Conference on Software Engineering (ICSE '14)
received ACM SIGSOFT Distinguished Paper Award

A Study and Toolkit for Asynchronous Programming in C#

Semih Okur*, David L. Hartveld[ϒ], Danny Dig[†], Arie van Deursen[ϒ]
*University of Illinois USA okur2@illinois.edu
[ϒ]Delft University of Technology Netherlands d.l.hartveld@student.tudelft.nl arie.vandeursen@tudelft.nl
[†]Oregon State University USA digd@eecs.oregonstate.edu

ABSTRACT

Asynchronous programming is in demand today, because responsiveness is increasingly important on all modern devices. Yet, we know little about how developers use asynchronous programming in practice. Without such knowledge, developers, researchers, language and library designers, and tool providers can make wrong assumptions.

We present the first study that analyzes the usage of asynchronous programming in a large experiment. We analyzed 1378 open source Windows Phone (WP) apps, comprising 12M SLOC, produced by 3376 developers. Using this data, we answer 2 research questions about use and misuse of asynchronous constructs. Inspired by these findings, we developed (i) Asyncifier, an automated refactoring tool that converts callback-based asynchronous code to use `async/await`; (ii) Corrector, a tool that finds and corrects common misuses of `async/await`. Our empirical evaluation shows that these tools are (i) applicable and (ii) efficient. Developers accepted 314 patches generated by our tools.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms: Design, Experimentation

Keywords: Program transformation, asynchronous, C#

1. INTRODUCTION

User interfaces are usually designed around the use of a single user interface (UI) event thread [15, 16, 23, 24]:a every operation that modifies UI state is executed as an event on that thread. The UI "freezes" when it cannot respond to input, or when it cannot be redrawn. It is recommended that long-running CPU-bound or blocking I/O operations execute asynchronously so that the application (app) continues to respond to UI events.

Asynchronous programming is in demand today because responsiveness is increasingly important on all modern devices: desktop, mobile, or web apps. Therefore, major programming languages have APIs that support non-blocking, asynchronous operations (e.g., to access the web, or for file operations). While these APIs make asynchronous programming possible, they do not make it easy.

Asynchronous APIs rely on callbacks. However, callbacks invert the control flow, are awkward, and obfuscate the intent of the original synchronous code [38].

Recently, major languages (F# [38], C# and Visual Basic [8] and Scala [7]) introduced `async` constructs that resemble the straightforward coding style of traditional synchronous code. Thus, they recognize asynchronous programming as a first-class citizen.

Yet, we know little about how developers use asynchronous programming and specifically the new `async` constructs in practice. Without such knowledge, other developers cannot educate themselves about the state of the practice, language and library designers are unaware of any misuse, researchers make wrong assumptions, and tool providers do not provide the tools that developers really need. This knowledge is also important as a guide to designers of other major languages (e.g., Java) planning to support similar constructs. Hence, asynchronous programming deserves first-class citizenship in empirical research and tool support, too.

We present the first study that analyzes the usage of asynchronous libraries and new language constructs, `async/await` in a large experiment. We analyzed 1378 open source Windows Phone (WP) apps, comprising 12M SLOC, produced by 3376 developers. While all our empirical analysis and tools directly apply to any platform app written in C# (e.g., desktop, console, web, tablet), in this paper we focus on the Windows Phone platform.

We focus on WP apps because we expect to find many exemplars of asynchronous programming, given that responsiveness is critical. Mobile apps can easily be unresponsive because mobile devices have limited resources and have high latency (excessive network accesses). With the immediacy of touch-based UIs, even small hiccups in responsiveness are more obvious and jarring than when using a mouse or keyboard. Some sluggishness might motivate the user to uninstall the app, and possibly submit negative comments in the app store [37]. Moreover, mobile apps are becoming increasingly more important. According to Gartner, by 2016 more than 300 billion apps will be downloaded annually [17].

The goal of this paper is twofold. First, we obtain a deep understanding of the problems around asynchronous programming. Second, we present a toolkit (2 tools) to address exactly these problems. To this end, we investigate 1378 WP apps through tools and by hand, focussing on the following research questions:

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

RQ1: How do developers use asynchronous programming?

RQ2: To what extent do developers misuse `async/await`?

We found that developers heavily use callback-based asynchronous idioms. However, Microsoft officially no longer recommends these asynchronous idioms [29] and has started to replace them with new idioms in new libraries (e.g., WinRT). Developers need to refactor callback-based idioms to new idioms that can take advantage of the `async/await` keywords. The changes that the refactoring requires are non-trivial, though. For instance, developers have to inspect deep call graphs. Furthermore, they need to be extra careful to preserve exception handling behavior. Thus, we implemented the refactoring as an automated tool, Asyncifier.

We also found that nearly half of WP8 apps have started to use the 9-month-old `async/await` keywords. However, developers misuse `async/await` in various ways. We define *misuse* as anti-patterns, which hurt performance and might cause serious problems like deadlocks. For instance, we found that 14% of methods that use (the expensive) `async/await` keywords do this unnecessarily, 19% of methods do not follow an important good practice [21], 1 out of 5 apps misses opportunities in async methods to increase asynchronicity, and developers (almost) always unnecessarily capture context, hurting performance. Thus, we implemented a transformation tool, Corrector, that finds and corrects the misused `async/await`.

This paper makes the following contributions:

Empirical Study: To the best of our knowledge, this is the first large-scale empirical study to answer questions about asynchronous programming and `async/await`, that will be available soon in other major programming languages. We present implications of our findings from the perspective of four main audiences: developers, language and library designers, researchers, and tool providers.

Toolkit: We implemented the analysis and transformation algorithms to address the challenges (Asyncifier and Corrector).

Evaluation: We evaluated our tools by using the code corpus and applied the tools hundreds of times. We show that our tools are highly applicable and efficient. Developers find our transformations useful. Using Asyncifier, we applied and reported refactorings in 10 apps. 9 replied and accepted each one of our 28 refactorings. Using Corrector, we found and reported misuses in 19 apps. 19 replied and accepted each of our 286 patches.

Outreach: Because developers learn new language constructs through both positive and negative examples, we designed a website, `http://LearnAsync.NET`, to show hundreds of such usages of asynchronous idioms and `async/await`.

2. BACKGROUND

When a button click event handler executes a synchronous long-running CPU-bound or blocking I/O operation, the user interface will freeze because the UI event thread cannot respond to events. Code listing 1 shows an example of such an event handler, method `Button_Click`. It uses the `GetFromUrl` method to download the contents of a URL, and place it in a text box. Because `GetFromUrl` is waiting for the network operation to complete, the UI event thread is blocked, and the UI is unresponsive.

Keeping UIs responsive thus means keeping the UI event thread free of those long-running or blocking operations. If these operations are executed asynchronously in the background, the foreground UI event thread does not have to busy-wait for completion of the operations. That frees up the UI event thread to respond to user input, or redraw the UI: the user will experience the UI to be responsive.

Code 1 Synchronous example

```
1  void Button_Click(...) {
2    string contents = GetFromUrl(url)
3    textBox.Text = contents;
4  }
5  string GetFromUrl(string url) {
6    WebRequest request = WebRequest.Create(url);
7    WebResponse response = request.GetResponse();
8    Stream stream = response.GetResponseStream();
9    return stream.ReadAsString();
10 }
```

CPU-bound operations can be executed asynchronously by (i) explicitly creating threads, or (ii) by reusing a thread from the thread pool.

I/O operations are more complicated to offload asynchronously. The naive approach would be to just start another thread to run the synchronous operation asynchronously, using the same mechanics as used for CPU-bound code. However, that would still block the new thread, which consumes significant resources, hurting scalability.

The solution is to use asynchronous APIs provided by the platform. The .NET framework mainly provides two models for asynchronous programming: (1) the Asynchronous Programming Model (APM), that uses callbacks, and (2) the Task Asynchronous Pattern (TAP), that uses `Task`s, which are similar to the concept of *futures* found in many other languages such as Java, Scala or Python.

2.1 Asynchronous Programming Model

APM, the Asynchronous Programming Model, was part of the first version of the .NET framework, and has been in existence for 10 years. APM asynchronous operations are started with a `Begin` method invocation. The result is obtained with an `End` method invocation. In Code listing 2, `BeginGetResponse` is such a `Begin` method, and `EndGetResponse` is an `End` method.

`BeginGetRepsonse` is used to initiate an asynchronous `HTTP GET` request. The .NET framework starts the I/O operation in the background (in this case, sending the request to the remote web server). Control is returned to the calling method, which can then continue to do something else. When the server responds, the .NET framework will "call back" to the application to notify that the response is ready. `EndGetResponse` is then used in the callback code to retrieve the actual result of the operation. See Figure 1 for an illustration of this flow of events.

The APM `Begin` method has two pattern-related parameters. The first parameter is the callback delegate (which is a managed, type-safe equivalent of a function pointer). It can be defined as either a method reference, or a lambda expression. The second parameter allows the developer to pass any single object reference to the callback, and is called `state`.

The .NET framework will execute the callback delegate on the thread pool once the asynchronous background operation completes. The `EndGetResponse` method is then used in the callback to obtain the result of the operation, the actual `WebResponse`.

Code 2 APM-based example

```
1  void Button_Click(...) {
2    GetFromUrl(url);
3  }
4  void GetFromUrl(string url) {
5    var request = WebRequest.Create(url);
6    request.BeginGetResponse(Callback, request);
7  }
8  void Callback(IAsyncResult aResult) {
9    var request = (WebRequest)aResult.AsyncState;
10   var response = request.EndGetResponse(aResult);
11   var stream = response.getResponseStream();
12   var content = stream.ReadAsString();
13   Dispatcher.BeginInvoke(() => {
14       textBox.Text = content;
15   });
16 }
```

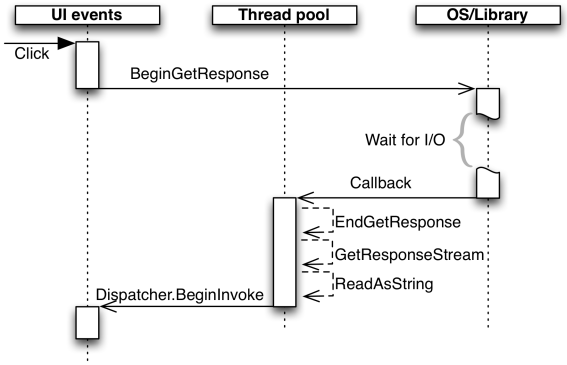

Figure 1: Where is callback-based APM code executing?

Note a subtle difference between the synchronous, sequential example in Code listing 1 and the asynchronous, APM-based example in Code listing 2. In the synchronous example, the `Button_Click` method contains the UI update (setting the download result as contents of the text box). However, in the asynchronous example, the final callback contains an invocation of `Dispatcher.BeginInvoke(...)` to change context from the thread pool to the UI event thread.

2.2 Task-based Asynchronous Pattern

TAP, the Task-based Asynchronous Pattern, provides for a slightly different approach. TAP methods have the same base operation name as APM methods, without 'Begin' or 'End' prefixes, and instead have an 'Async' suffix. The API consists of methods that start the background operation and return a `Task` object. The `Task` represents the operation in progress, and its future result.

The `Task` can be (1) queried for the status of the operation, (2) synchronized upon to wait for the result of the operation, or (3) set up with a continuation that resumes in the background when the task completes (similar to the callbacks in the APM model).

2.3 Drawbacks of APM and Plain TAP

Using APM and plain TAP directly has two main drawbacks. First, the code that must be executed after the asynchronous operation is finished, must be passed explicitly to the `Begin` method invocation. For APM, even more scaffolding is required: The `End` method must be called, and that usually requires the explicit passing and casting of an 'async state' `object` instance - see Code listing 2, lines 9-10. Second, even though the `Begin` method might be called from the UI event thread, the callback code is executed on a thread pool thread. To update the UI after completion of the asynchronous operation from the thread pool thread, an event must be sent to the UI event thread explicitly - see Code listing 2, line 13-15.

Code 3 TAP & `async/await`-based example

```
1  async void Button_Click(...) {
2    var content = await GetFromUrlAsync(url);
3    textBox.Text = content;
4  }
5  async Task<string> GetFromUrlAsync(string url) {
6    var request = WebRequest.Create(url);
7    var response = await request.GetResponseAsync()
8                              .ConfigureAwait(false);
9    var stream = response.GetResponseStream();
10   return stream.ReadAsString();
11 }
```

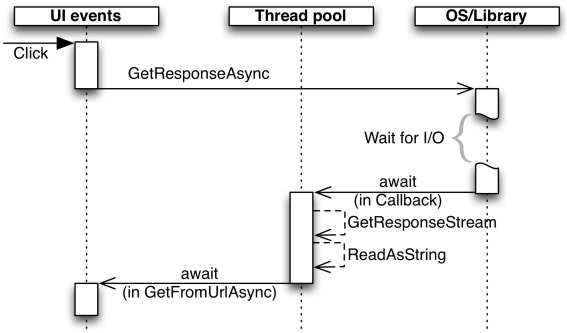

Figure 2: Where is the async/await-based code executing?

2.4 Pause & Play with `async/await`

To solve this problem, the `async` and `await` keywords have been introduced in 2012 in C# 5.0. When a method has the `async` keyword modifier in its signature, the `await` keyword can be used to define pausing points. When a `Task` is awaited in an `await` expression, the current method is paused and control is returned to the caller. When the `await`'ed `Task`'s background operation is completed, the method is resumed from right after the `await` expression. Code listing 3 shows the TAP- & `async/await`-based equivalent of Code listing 2, and Figure 2 illustrates its flow of execution.

The code following the `await` expression can be considered a continuation of the method, exactly like the callback that needs to be supplied explicitly when using APM or plain TAP. Methods that have the `async` modifier will thus run synchronously up to the first `await` expression (and if it does not have any, it will complete synchronously). Merely adding the `async` modifier does not magically make a method be asynchronously executed in the background.

2.5 Where is the Code Executing?

There is one important difference between `async/await` continuations, and APM or plain TAP callback continuations: APM and plain TAP always execute the callback on a thread pool thread. The programmer needs to explicitly schedule a UI event to interface with the UI, as shown in Code listing 2 and Figure 1.

In `async/await` continuations, the `await` keyword, by default, captures information about the thread in which it is executed. This captured context is used to schedule execu-

tion of the rest of the method in the same context as when the asynchronous operation was called. For example, if the `await` keyword is encountered in the UI event thread, it will capture that fact. Once the background operation is completed, the continuation of the rest of the method is scheduled back onto the UI event thread. This behavior allows the developer to write asynchronous code in a sequential manner. See Code listing 3 for an example.

Comparing the code examples in Code listings 1 and 3 will show that the responsive version based on TAP & `async/await` only slightly differs from the sequential version. It is readable in a similar fashion, and even the UI update (setting contents of the text box) is back at its original place.

By default, `await` expressions capture the current context. However, it is not always needed to make the expensive context switch back to the original context. To forestall a context switch, an `await`'ed `Task` can be set to ignore capturing the current context by using `ConfigureAwait(false)`. In Code listing 3, in `GetFromUrlAsync`, none of the statements following the await expressions require access to the UI. Hence, the `await`'ed `Task` is set with `ConfigureAwait(false)`. In `Button_Click`, the statement following `await GetFromUrlAsync(url)` does need to update the UI. So that `await` expression should capture the original context, and the task should not be set up with ConfigureAwait(false).

3. RESEARCH QUESTIONS

We are interested in assessing the usage of state of the art asynchronous programming in real world WP apps.

3.1 Methodology

Corpus of Data: We chose Microsoft's CodePlex [11] and GitHub [18] as sources of the code corpus of WP apps. According to a recent study [26], most C# apps reside in these two repositories. We developed WPCollector to create our code corpus. It is available online [10] and can be reused by other researchers.

We used WPCollector to download all recently updated WP apps which have a WP-related signature in their project files. It ignores (1) apps without commits since 2012, and (2) apps with less than 500 non-comment, non-blank lines of code (SLOC). The latter "toy apps" are not representative of production code.

WPCollector makes as many projects compilable as possible (e.g., by resolving-installing dependencies), because the Roslyn APIs that we rely on (see Analysis Infrastructure) require compilable source code.

WPCollector successfully downloaded and prepared 1378 apps, comprising 12M SLOC, produced by 3376 developers. Our analysis uses all apps, without sampling.

In our corpus, 1115 apps are targeting WP7, released in October 2010. Another 349 apps target WP8, released in October 2012. 86 apps target both platforms.

Analysis Infrastructure: We developed AsyncAnalyzer to perform the static analysis of asynchronous programming construct usage. We used Microsoft's recently released Roslyn [30] SDK, which provides an API for syntactic and semantic program analysis, AST transformations and editor services in Visual Studio. Because the publicly available version of Roslyn is incomplete and does not support the `async/await` keywords yet, we used an internal build obtained from Microsoft.

Table 1: Usage of asynchronous idioms. The three columns per platform show the total number of idiom instances, the total number of apps with instances of the idiom, and the percentage of apps with instances of the idiom.

	WP7			WP8		
	#	# App	%	#	# App	%
I/O APM	1028	242	22%	217	65	19%
I/O TAP	123	23	2%	269	57	16%
New Thread	183	92	8%	28	24	7%
BG Worker	149	73	6%	11	6	2%
ThreadPool	386	103	9%	52	24	7%
New Task	51	11	1%	182	28	8%

We executed AsyncAnalyzer over each app in our corpus. For each of these apps, it inspects the version from the main development branch as of August 1st, 2013. We developed a specific analysis to answer each research question.

3.2 How do Developers Use Asynchronous Programming?

Asynchronous APIs: We detected all APM and TAP methods that are used in our code corpus as shown in Table 1. Because in WP7 apps, TAP methods are only accessible via additional libraries, Table 1 tabulates the usage statistics for WP7 and WP8 apps separately. The data shows that APM is more popular than TAP for both WP7 and WP8.

We also manually inspected all APM and TAP methods used and categorized them based on the type of I/O operations: network (1012), file system (310), database (145), user interaction (102) and other I/O (e.g., speech recognition) (68). We found that asynchronous operations are most commonly used for network operations.

There are two ways to offload CPU-bound operations to another thread: by creating a new thread, or by reusing threads from the thread pool. Based on C# books and references [1], we distinguish 3 different approaches developers use to access the thread pool: (1) the BackgroundWorker class, (2) accessing the ThreadPool directly, and (3) creating `Task`s. Table 1 tabulates the usage statistics of all these approaches. Because `Task` is only available in WP7 apps by using additional libraries, the table shows separate statistics for WP7 and WP8 apps. The data shows that `Task` is used significantly more in WP8 apps, most likely because of availability in the core platform.

Language Constructs: `async/await` have become accessible for WP development in last quarter of 2012. While they are available by default in WP8, WP7 apps have to reference `Microsoft.Bcl.Async` library to use them.

We found that 45% (157) of WP8 apps use `async/await` keywords. While nearly half of all WP8 apps have started to use the new 9-month-old constructs, only 10 WP7 apps use them. In the combined 167 apps, we found that there are 2383 `async` methods that use at least one `await` keyword in their method body. An `async` method has 1.6 `await` keywords on average, meaning that `async` methods call other `async` methods.

Callback-based APM is the most widely used idiom. While nearly half of all WP8 apps have started to use async/await, only 10 WP7 apps use them.

3.3 Do Developers Misuse `async/await`?

Because `async/await` are relatively new language constructs, we have also investigated how developers misuse them. We define misuse as anti-patterns which hurt performance and might cause serious problems like deadlocks. We detected the following typical misusage idioms.

3.3.1 *Fire & Forget Methods*

799 of 2382 `async/await` methods return `void` instead of `Task`. This means they are "fire&forget" methods, which cannot be awaited. Exceptions thrown in such methods cannot be caught in the calling method, and cause termination of the app. Unless these methods are UI event handlers, this is a code smell.

However, we found that only 339 out of these 799 `async void` methods are event handlers. It means that 19% of all `async` methods (460 out of 2383) are not following this important practice [21]. Instead, such methods should return `Task`, which does not force the developer to change anything else, but it does enable easier error handling, composition and testability.

One in five async methods violate the principle that an `async` method should be awaitable unless it is the top level event handler.

3.3.2 *Unnecessary* `async/await` *Methods*

Consider the example from "Cimbalino Windows Phone Toolkit" [3]:

```
public async Task<Stream> OpenFileForReadAsync(...) {
  return await Storage.OpenStreamForReadAsync(path);
}
```

The `OpenStream` method is a TAP call, which is awaited in the `OpenFile` method. However, there is no need to await it. Because there is no statement after the `await` expression except for the return, the method is paused without reason: the `Task` that is returned by `Storage.OpenStream` can be immediately returned to the caller. The snippet below behaves exactly the same as the one above:

```
public Task<Stream> OpenFileForReadAsync(...) {
  return Storage.OpenStreamForReadAsync(path);
}
```

It is important to detect this kind of misuse. Adding the `async` modifier comes at a price: the compiler generates some code in every `async` method and generated code complicates the control flow which results in decreased performance.

We discovered that in 26% of the 167 apps, 324 out of all 2383 `async` methods unnecessarily use `async/await`. **There is no need to use `async/await` in 14% of `async` methods.**

3.3.3 *Using Long-running Operations under Async Methods*

We also noticed that developers use some potentially long-running operations under `async` methods even though there are corresponding asynchronous versions of these methods in .NET or third-party libraries. Consider the following example from indulged-flickr [14]:

```
public async void GetPhotoStreamAsync(...) {
  var response = await DispatchRequest(...);
  using (StreamReader reader = new StreamReader(...)){
    string jsonString = reader.ReadToEnd();
  }
}
```

The developer might use `await ReadToEndAsync()` instead of the synchronous `ReadToEnd` call, especially if the stream is expected to be large.

In the example below from iRacerMotionControl [22], the situation is more severe.

```
private async void BT2Arduino_Send(string WhatToSend){
  ...
  await BTSock.OutputStream.WriteAsync(datab);
  txtBTStatus.Text = "sent";
  System.Threading.Thread.Sleep(5000);  ...
}
```

The UI event thread calls `BT2Arduino_Send`, which blocks the UI thread by busy-waiting for 5 seconds. Instead of using the blocking `Thread.Sleep` method, the developer should use the non-blocking `Task.Delay(5000)` method call to preserve similar timing behavior, and `await` it to prevent the UI to freeze for 5 seconds.

We found 115 instances of potentially long-running operations in 22% of the 167 apps that use `async/await`. **1 out of 5 apps miss opportunities in at least one async method to increase asynchronicity**.

3.3.4 *Unnecessarily Capturing Context*

`async/await` introduce new risks if the context is captured without specifying `ConfigureAwait(false)`. For example, consider the following example from adsclient [2]:

```
void GetMessage(byte[] response) {
  ...
  ReceiveAsync(response).Wait(); ...
}
async Task<bool> ReceiveAsync(byte[] message) {
  ...
  return await tcs.Task;
}
```

If `GetMessage` is called from the UI event thread, the thread will wait for completion of `ReceiveAsync` because of the `Wait` call. When the await completes in `ReceiveAsync`, it attempts to execute the remainder of the method within the captured context, which is the UI event thread. However, the UI event thread is already blocked, waiting for the completion of `ReceiveAsync`. Therefore, a deadlock occurs.

To prevent the deadlock, the developer needs to set up the `await` expression to use `ConfigureAwait(false)`. Instead of attempting to resume the `ReceiveAsync` method on the UI event thread, it now resumes on the thread pool, and the blocking wait in `GetMessage` does not cause a deadlock any more. In the example above, although `ConfigureAwait(false)` is a solution, we fixed it by removing `await` because it was also an instance of unnecessary `async/await` use. The developer of the app accepted our fix as a patch.

We found 5 different cases for this type of deadlock which can happen if the caller method executes on UI event thread.

Capturing the context can also cause another problem: it hurts performance. As asynchronous GUI applications grow larger, there can be many small parts of `async` methods all using the UI event thread as their context. This can cause sluggishness as responsiveness suffers from thousands of paper cuts. It also enables a small amount of parallelism: some asynchronous code can run in parallel with the UI event thread instead of constantly badgering it with bits of work to do.

To mitigate these problems, developers should await the `Task` with `ConfigureAwait(false)` whenever they can. If the statements after the `await` expression do not update the UI, `ConfigureAwait(false)` must be set. Detecting this misuse is important because using `ConfigureAwait(false)` might prevent future bugs like deadlocks and improve the performance.

1786 out of 2383 `async` methods do not update GUI elements in their call graph after `await` expressions.

Table 2: Statistics of Misuses

Misuse	#	Method	App
(1) Fire & Forget	460	19%	76%
(2) Unnecessary Async	324	14%	26%
(3) Potential LongRunning	115	5%	22%
(4) Unnecessary Context	1770	74%	86%

We found that `ConfigureAwait(false)` is used in only 16 out of these 1786 `async` methods in `await` expressions. All 1770 other `async` methods should have used `ConfigureAwait(false)`. **99% of the time, developers did not use `ConfigureAwait(false)` where this was needed.**

4. TOOLKIT

Based on our findings from Section 3, we developed a two-fold approach to support the developer: (1) Asyncifier, a refactoring tool to upgrade legacy callback-based APM code to take advantage of `async/await` construct (see Section 4.1) and (2) Corrector, a tool for detecting and fixing misuses of `async/await` in code (see Section 4.2).

Asyncifier helps the developer in two ways: (1) the code is upgraded without errors, retaining original behavior, and (2) it shows how to correctly use `async/await` in production code. If the developer manually introduces `async/await`, Corrector will help in finding and removing misuses.

4.1 Refactoring APM to `async/await`

4.1.1 Challenges

There are three main challenges that make it hard to execute the refactoring quick and flawlessly by hand. First, the developer needs to understand if the APM instance is a candidate for refactoring based on the preconditions in Section 4.1.2. Second, he must transform the code while retaining the original behavior of the code - both functionally and in terms of scheduling. This is non-trivial, especially in the presence of (1) exception handling, and (2) APM `End` methods that are placed deeper in the call graph.

Exception handling: The refactoring from APM to `async/await` should retain the functional behavior of the original program, both in the normal case and under exceptional circumstances. In 52% of all APM instances, `try-catch` blocks are in place to handle those exceptions. The `try-catch` blocks surround the `End` method invocation, which throws an exception if the background operation results in an exceptional circumstance. These `catch` blocks can contain business logic: for example, a network error sometimes needs to be reported to the user ("Please check the data or WiFi connection"). Code listing 4 shows such an example.

The naive approach to introducing `async/await` is to replace the `Begin` method invocation with an invocation to the corresponding TAP method, and await the result immediately. However, the `await` expression is the site that can throw the exception when the background operation failed. In the APM code, the exception would be thrown at the `End` call site. Thus, the exception would be thrown at a different site, and this can drastically change behavior. Exception handling behavior can be retained by introducing the `await` expression as replacement of the `End` method call at the exact same place. This is a non-trivial insight for developers, because online examples of `async/await` only show the refactoring for extremely simple cases, where this is not a concern.

Code 4 EndGetResponse in try-catch block

```
void Button_Click(...) {
  WebRequest request = WebRequest.Create(url);
  request.BeginGetResponse(Callback, request);
}
void Callback(IAsyncResult ar) {
  WebRequest request = (WebRequest)ar.AsyncState;
  try {
    var response = request.EndGetResponse(ar);
    // Code does something with successful response
  } catch (WebException e) {
    // Error handling code
  }
}
```

Code 5 EndGetResponse on longer call graph path

```
void Button_Click(...) {
  WebRequest request = WebRequest.Create(url);
  request.BeginGetResponse(ar => {
    IntermediateMethod(ar, request);
  }, null);
}
void IntermediateMethod(IAsyncResult result,
     WebRequest request) {
  var response = GetResponse(request, result);
  // Code does something with response
}
WebResponse GetResponse(WebRequest request,
     IAsyncResult result) {
  return request.EndGetResponse(result);
}
```

Hidden End methods: The developer needs to take even more care when the `End` method is not immediately called in the `callback` lambda expression, but is 'hidden' deeper down the call chain. Code listing 5 shows such an example. In order to preserve exceptional behavior, the `Task` instance must be passed down to the call site of the `End` method. This requires an inter-procedural analysis of the code: each of the methods, through which the `IAsyncResult` 'flows', must be refactored, which makes the refactoring more tedious. The developer must trace the call graph of the callback to find the `End` method call, and in each encountered method: (1) replace the `IAsyncResult` parameter with a `Task<T>` parameter (with `T` being the return type of the TAP method, (2) replace the return type `R` with `async Task<R>`, or `void` with `async void` or `async Task`, and (3) introduce `ConfigureAwait(false)` at each `await` expression. As shown in the results of the empirical study, developers almost never use `ConfigureAwait(false)`, despite the fact that it is critical for UI responsiveness.

4.1.2 Algorithm Preconditions

An invocation of a `Begin` method is a candidate for refactoring to `async/await` constructs, if it adheres to the following preconditions and restrictions:

P1: The APM method call must represent an asynchronous operation for which a TAP-based method also exists. If the TAP-based method does not exist, the code cannot be refactored.

P2: The `Begin` method invocation statement must be contained in a regular method, i.e, not in a lambda expression or delegate anonymous method. The `Begin` method will be made `async`. While it is possible to make lambdas and delegate anonymous methods `async`, this is considered a bad practice because it usually creates an `async void` fire & forget method (see Section 3.3.1).

Code 6 Adheres to precondition

```
void Action(WebRequest request) {
  request.BeginGetResponse(asyncResult => {
    var response=request.EndGetRequest(asyncResult);
    // Code does something with response
  }, null);
}
```

Code 7 Code listing 2 refactored to meet preconditions

```
void GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  request.BeginGetResponse(asyncResult => {
    Callback(asyncResult, request);
  }, null);
}
void Callback(IAsyncResult ar, WebRequest request) {
  var response = request.EndGetResponse(ar);
  var stream = response.getResponseStream();
  var content = stream.ReadAsString();
  Dispatcher.BeginInvoke(() => {
      textBox.Text = content;
  });
}
```

P3: The `callback` argument must be a lambda expression with a body consisting of a block of statements and the call graph of that block must contain an `End` method invocation that takes the lambda `IAsyncResult` parameter as argument. The callback must actually terminate the background operation. Otherwise, there is no way to get the result from the asynchronous operation.
P4: In the callback call graph, the `IAsyncResult` lambda parameter should not be used, except as an argument to the `End` method, because it will be removed after the refactoring.
P5: In the initiating method (the method containing the `Begin` method invocation), the `IAsyncResult` return value of the `Begin` method should not be used, because it is returned by a method invocation that will disappear.

Code listing 6 shows a valid example in the context of these preconditions.

Applying these preconditions to APM instances in real-world applications would restrict the number of APM instances that can be refactored. Fortunately, many instances in other forms can be refactored into this form. Code listing 2 shows an example that fails **P3**: the `callback` argument is a method reference. This instance can be refactored into the code shown in listing 7 by applying the "Introduce Parameter" refactoring to the `request` variable in the original `Callback` method.

Based on encountered cases in the analyzed code corpus, we have identified and implemented several such refactorings in Asyncifier. An example is the rewriting of some callback argument expressions (solves violations of **P3**)

4.1.3 *Refactoring APM Instances*

Asyncifier detects all `Begin` method invocations that fulfill the preconditions. It takes the following steps to refactor the APM instance to `async/await`-based constructs.

(1)Traveling the call graph from APM Begin to End: First, Asyncifier explores the call graph of the body of the callback lambda expression to find the invocation path to the `End` invocation. It does a depth-first search of the call graph, by looking up the symbols of any non-virtual method that is encountered. There are two possible scenarios: the `End` method invocation (1) is placed directly in the lambda expression, or (2) it is found on the call graph of the lambda body in another method's body. Code listing 6 is an example of the first case.

In the second case, Asyncifier identifies three different methods which are on the call graph path: (1) the *initiating method*, i.e., the method containing the `Begin` method invocation, (2) the *result-obtaining method*, i.e., the method containing the `End` method invocation, and (3) *intermediate methods*, i.e., the remaining methods on the path. Code listing 7 is an example of the second case. We use this example in the description of the following steps.

(2) Rewriting the initiating method: In both cases, the initiating method needs to be rewritten. Asyncifier adds the *async* modifier to the signature of the initiating method. It changes the return value is to either *Task* instead of *void*, or *Task<T>* for any return type *T*.

```
void GetFromUrl(string url) { ... }
```

⇓

```
async Task GetFromUrl(string url) { ... }
```

Asyncifier replaces the `Begin` method invocation statement with a local variable declaration of a task that is assigned the result of the corresponding TAP method invocation. The parameterized type is the return type of the `End` method:

```
request.BeginGetResponse(...);
```

⇓

```
Task<WebResponse> task = request.GetResponseAsync();
```

It then concatenates the statements in the lambda expression body to the body of the initiating method:

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  var task = request.GetResponseAsync();
  Callback(asyncResult, request);
}
```

It replaces the `asyncResult` lambda parameter reference `asyncResult` with a reference to the newly declared `Task` instance.

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  var task = request.GetResponseAsync();
  Callback(task, request);
}
```

(3) Rewriting the result-obtaining method: Asyncifier updates the signature of the result-obtaining method as follows: (1) it adds the `async` modifier, (2) it replaces return type `void` with `Task`, or any other `T` with `Task<T>`, and (3) it replaces the `IAsyncResult` parameter with `Task<R>`, with `R` the return type of the `End` method.

```
void Callback(IAsyncResult asyncResult,
              WebRequest request) { ... }
```

⇓

```
async Task Callback(Task<WebResponse> task,
                    WebRequest request) { ... }
```

Then it replaces the `End` method invocation expression with `await task`, without capturing the synchronization context:

```
var response = request.EndGetResponse(asyncResult);
```

⇓

```
var response = await task.ConfigureAwait(false);
```

Asyncifier refactors the APM instance into the code shown in Code listing 8. If the introduction of new variables leads to identifier name clashes, Asyncifier disambiguates the newly introduced names by appending an increasing number to them, i.e., *task1*, *task2*, etc.

Code 8 `async/await` code after refactoring Code listing 7

```
async Task GetFromUrl(string url) {
  var request = WebRequest.Create(url);
  Task<WebResponse> task=request.GetResponseAsync();
  Callback(task, request);
}

async Task Callback(Task<WebResponse> task,
                    WebRequest request) {
  var response = await task.ConfigureAwait(false);
  var stream = response.getResponseStream();
  var content = stream.ReadAsString();
  Dispatcher.BeginInvoke(() => {
      textBox.Text = content;
  });
}
```

(4) Callbacks containing the End call: If the `End` method invocation is now in the initiating method, Asyncifier replaces it with an await expression, and the refactoring is complete. The example in Code listing 6 would be completely refactored at this point:

```
void Action(WebRequest request) {
  var task = request.GetResponseAsync();
  var response = await task.ConfigureAwait(false);
  // Do something with response.
}
```

(5) Rewriting intermediate methods: Intermediate methods must be rewritten if the `End` method is not invoked in the callback lambda expression body. Asyncifier recursively refactors every method, applying the same steps as for the result-obtaining method. Additionally, at the call site of each method, the reference to the (removed) `result` parameter is replaced with a reference to the (newly introduced) `task` parameter.

4.1.4 *Retaining Original Behavior*

It is crucial that the refactored code has the same behavior in terms of scheduling as the original code. With both the `Begin` method and the TAP method, the asynchronous operation is started. In the APM case, the callback is only executed once the background operation is completed. With `async/await`, the same *happens-before* relationship exists between the `await` expression and the statements that follow the `await` of the `Task` returned by the TAP method. Because the statements in callbacks are placed after the `await` expression that pauses execution until the completion of the background operation, this timing behavior is preserved by Asyncifier.

4.1.5 *Implementation Limitations*

The set of candidate programs that can be refactored is restricted by tool limitations related to re-use of `Begin` or `End` methods. First, there should not be other call graph paths leading from `Begin` method call to the target `End` method, which means that the specific `End` method invocation must not be shared between multiple `Begin` invocations. Second, recursion in the callback through another `Begin` call that references the same callback again is not allowed (essentially, this is also sharing of an `End` method call). Third, Asyncifier does not support multiple `End` method invocations that correspond to a single `Begin` method invocation, for example through the use of branching. However, this case is very rare.

4.2 Corrector

We implemented another tool, Corrector, that detects and corrects common misuses that we explained in Section 3.3. Corrector gets the project file as an input and automatically corrects the misuses if it finds any, thus operating in batch mode. In addition, Corrector also works in interactive mode via *Quick Fix* mode for Visual Studio. This mode shows a small icon close to the location of the misuse to fix the problem. Because Corrector is constantly scanning the source code as soon as the user typed in new code, it dramatically shortens the time between the introduction of an error and its correction.

Unnecessary async/await methods: Corrector checks whether `async` method body has only one `await` keyword and this `await` is used for a TAP method call that is the last statement of the method. Corrector does not do this for `async void` (fire&forget) methods; because if it removes `await` from the last statement in `async void` methods, it will silence the exception that can occur in that statement.

To fix these cases, Corrector removes the `async` from the method identifiers and the `await` keyword from the TAP method call. The method will return the `Task` that is the result of TAP method call.

Long-running operations under async methods: To detect these operations, Corrector looks up symbols of each method invocation in the bodies of `async` methods. After getting symbol information, Corrector looks at the other members of the containing class of that symbol to check whether there is an asynchronous version. For instance, if there is an `x.Read()` method invocation and x is an instance of the `Stream` class, Corrector looks at the members of the `Stream` class to see whether there is a `ReadAsync` method that gets the same parameters and returns Task. By checking the members, Corrector can also find asynchronous versions not only in the .NET framework but also in third-party libraries.

Corrector also maps corresponding blocking and non-blocking methods which do not follow the `Async` suffix convention (e.g., Thread.Sleep -> Task.Delay).

Corrector avoids introducing asynchronous operations of file IO operations in loops, as this could result in slower performance than the synchronous version.

After finding the corresponding non-blocking operation, Asyncifier simply replaces the invocation with the new operation and makes it `await`'ed.

Unnecessarily capturing context: Corrector checks the call graph of `async` methods to see if there are accesses of GUI elements. All GUI elements are in the namespaces `System.Windows` and `Microsoft.Phone`. If Corrector does not find any use of elements from those namespaces after `await` expressions, it introduces `ConfigureAwait(false)` in those await expressions.

Fire & Forget methods: There is no automated fix for this misuse. If fire & forget method is converted to `async Task` method and is awaited in the caller, it will change the semantics. Therefore, the developer's understanding of code is required to fix this case.

5. EVALUATION

To evaluate Asyncifier and Corrector, we studied their *applicability*, their *impact* on the code, their *performance*, and the *usefulness* of their results to developers.

Applicability: To evaluate the applicability, we executed Asyncifier over our code corpus. After each transformation, Asyncifier compiled the app in-memory and checked whether compilation errors were introduced. 54% of the 1245 `APM` instances adhere to the preconditions set in section 4.1.2, which were all successfully refactored. By manually checking 10% of all transformed instances, randomly sampled, we verified that Asyncifier refactors APM instances correctly. In the 46% of unsupported `APM` instances, Asyncifier does not change the original program.

The two main causes for unsuccessful refactorings are (1) instances that do not adhere to preconditions, and (2) tool limitations. The former consist mostly of instances that can not be refactored because of fundamental limitations of the algorithm. Examples are callback expressions that reference a field delegate, or APM `End` methods that are hidden behind interface implementations (violations of precondition **P3**). The latter consist of the examples given in Section 4.1.5.

We also applied Corrector to the full corpus (2209 times). All instances of type 2, 3, and 4 misuses (see Table 2 were corrected automatically.

Code Impact: To evaluate the impact of our refactorings on the code we investigate the size of the changes. Asyncifier changes 28.9 lines on average per refactoring. This shows that automation is needed, since each refactoring changes many lines of code. Moreover, these changes are not trivial.

Corrector changes one line per each misuse of type (3) and (4) in Section 4.2. It changes 2 or 3 lines per each misuse of type (2); 2.1 lines on average.

Tool Performance: For Asyncifier, the average time needed to refactor one instance is 508ms rendering Asyncifier suitable for an interactive refactoring mode in an IDE.

Because the detection and fixing of type (2) and (3) misuses is straightforward, we did not measure the execution time. However, detecting type (4) misuse is expensive, as it requires inspection of the call graph of the `async` method. We found that analyzing one `async` method for this misuse takes on average 47ms. This shows that Corrector can be used interactively in an IDE, even for type (4) misuse.

Usefulness of Patched Code: To further evaluate the usefulness in practice, we conducted a qualitative analysis of the 10 most recently updated apps that have APM instances. We applied Asyncifier ourselves, and offered the modifications to the original developers as a patch via a pull request.[1] 9 out of 10 developers responded, and accepted each one of our 28 refactorings.

We received very positive feedback on these pull requests. One developer would like to have the tool available right now: "I'll look forward to the release of that refactoring tool, it seems to be really useful." [33] The developer of phoneguitartab [4] said that he had "been thinking about replacing all asynchronous calls [with] new async/await style code". This illustrates the demand for tool support for the refactoring from APM to `async/await`.

For Corrector, we selected the 10 most recently updated apps for all type (2) and (3) misuses. In total, we selected 19 apps because one app had both type (2) and (3). We used the same corpus of 19 apps to also fix misuses of type (4). Developers of 19 apps replied and accepted all our patches, corresponding to 149 instances of type (2), 39 instances of type (3), and 98 instances of type (4) misuses. In total 18 apps accepted 286 instances of Corrector transformations.

Response to the fixes that removed unnecessary `async/await` keywords was similarly positive. One developer pointed out that he missed several unnecessary `async/await` instances that Corrector detected: "[...] I normally try to take the same minimizing approach, though it seems I missed these." [32] The developer of SoftbuildData [6] experienced performance improvements after removing unnecessary `async/await`: "[...] performance has been improved to 28 milliseconds from 49 milliseconds." Again, these illustrate the need for tools that support the developer in finding problems in the use of `async/await`.

Furthermore, the developers of the playerframework [5] said that they missed the misuses because the particular code was ported from old asynchronous idioms. This demonstrates the need for Asyncifier as it can help a developer to upgrade his or her code, without introducing incorrect usage of `async/await`.

6. DISCUSSION

6.1 Implications

Our study has implications for developers, educators, language and library designers, tool providers, and researchers.

Developers learn and *educators* teach new programming constructs through both positive and negative examples. Robillard and DeLine [35] study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. We provide hundreds of real-world examples of all asynchronous idioms on `LearnAsync.net`. Because developers might need to inspect the whole source file or project to understand the example, we also link to highlighted source files on GitHub [39]. We also provide negative examples anonymously, without giving app names.

Language and library designers can learn which constructs and idioms are embraced by developers, and which ones are tedious to use or error-prone. Because some other major languages (Java 9) have plans to introduce similar constructs for asynchronous programming, this first study can guide them to an improved design of similar language constructs for their languages. The architects of async constructs in C#, F#, and Scala confirmed that our findings are useful and will influence the future evolution of these constructs. For instance, capturing the context need not be the default: as we have seen developers are very likely to forget to use `ConfigureAwait(false)`.

Tool providers can take advantage of our findings on `async/await` misuse. IDEs such as Visual Studio should have built-in quick fixes (similar to ours) to prevent users from introducing misuse. For instance, if developers introduce a fire & forget method, the IDE should give a warning unless the method is the top level event handler.

Researchers in the refactoring community can use our findings to target future research. For example, as we see from Table 1, the usage of `Task` jumped to 8% from 1% in WP8. This calls for work on a tool that converts old asynchronous idioms of CPU-bound computations (e.g., `Thread`) to new idioms (e.g., `Task`).

6.2 Why is `async/await` Commonly Misused?

We have seen extensive misuse of the `async/await` keywords in the projects in our code corpus. Different authors, both

[1] All patches can be found on our web site: `LearnAsync.NET`

from Microsoft and others, have documented these potential misuses extensively. This raises the question: Why is the misuse so extensive? Are developers just uninformed, or are they unaware of risks or performance characteristics of `async/await`?

The `async/await` feature is a powerful abstraction. Asynchronous methods are more complicated than regular methods in three ways. (1) Control flow of asynchronous methods. Control is returned to the caller when awaiting, and the continuation is resumed later on. (2) Exception handling. Exceptions thrown in asynchronous methods are automatically captured and returned through the `Task`. The exception is then re-thrown when the `Task` is awaited. (3) Non-trivial concurrent behavior. Up to the first await expression, the asynchronous method is executed synchronously. The continuation is potentially executed in parallel with the main thread. Each of these is a leak in the abstraction, which requires an understanding of the underlying technology - which developers do not yet seem to grasp.

Another problem might simply be the naming of the feature: asynchronous methods. However, the first part of the method executes *synchronously*, and possibly the continuations do so as well. Therefore, the name *asynchronous* method might be misleading: the term *pauseable* could be more appropriate.

Microsoft has introduced a powerful new feature with `async/await` keywords in C# 5. This empowers developers. However, the flip side of the same coin is that they can easily misuse it. Therefore, it is important to support developers in their understanding of the feature, its pros and cons, and best practices. Documentation is one manner of doing this; a toolkit and/or IDE support is another. The authors believe that these are complimentary, and should both be provided.

6.3 Threats to Validity

External: For what C# programs are the results representative? First, despite the fact that our corpus contains only open source apps, the 1378 apps span a wide domain, from games, social networking, and office productivity to image processing and third party libraries. They are developed by different teams with 3376 contributors from a large and varied community, comprising all Windows Phone apps from GitHub and Codeplex.

Second, our *tools* can be applied to any C# system. Thus, a similar search for misuses and refactoring opportunities can be conducted on non-Phone systems, such as reusable libraries, server side code, or desktop applications.

Internal: With respect to the misuse analysis, for most cases, our analysis is an under-estimation: For example, we detect one form of *unnecessary* `async/await`, but likely there are other unnecessary uses that we do not detect. In other cases, heuristics are used, for example to discover the existence of asynchronous versions of long running methods, or for detecting GUI code. Concerning the applicability of the refactorings, our tool presently can handle 54% of the `APM` cases. This percentage can go up by addressing some of Asyncifier's tool limitations.

Reliability: To facilitate replication of our analysis, we provide a detailed description of our results with fine-grained reports online at `http://LearnAsync.NET/`. The tools can be downloaded through our webpage.

7. RELATED WORK

Empirical Studies: There are several empirical studies [9, 19, 25, 28, 31] on the usage of libraries or programing language constructs. To the best of our knowledge, there is no empirical study on asynchronous constructs and language constructs for asynchronous programming.

We have previously conducted an empirical study [27] on how developers from thousands of open source projects use Microsoft's Parallel Libraries. There is a small intersection between asynchronous and parallel libraries: only `Thread`, `Task`, and `ThreadPool` constructs. In this paper, we studied these three constructs as 3 of 5 different approaches to executing CPU-bound computations asynchronously.

Refactoring Tools: Traditionally, refactoring tools have been used to improve the design of sequential programs. There are a few refactoring tools that specifically target concurrency. We have used refactoring [12, 13] to retrofit parallelism into sequential applications via concurrent libraries. In the same spirit, Wloka et al. present a refactoring for replacing global state with thread local state [40]. Schafer et al. present Relocker [36], a refactoring tool that lets programmers replace usages of Java built-in locks with more flexible locks. Gyori et al. present Lambdaficator [20], that refactors existing Java code to use lambda expressions to enable parallelism.

To the best of our knowledge, there is no refactoring tool that specifically targets asynchronous programming. In industry, ReSharper [34] is a well-known refactoring tool, but it does not support `async/await`-specific refactorings. Our refactoring helps developer design responsive apps, which is an area that has not yet been explored.

8. CONCLUSION

Because responsiveness is very important on mobile devices, asynchronous programming is already a first-class citizen in modern programming languages. However, the empirical research community and tool providers have not yet caught up.

Our large-scale empirical study of Windows Phone apps provides insight into how developers use asynchronous programming. We have discovered that developers make many mistakes when manually introducing asynchronous programming. We provide a toolkit to support developers in preventing and curing these mistakes. Our toolkit (1) safely refactors legacy callback-based asynchronous code to the new `async/await`, (2) detects and fixes existing errors, and (3) prevents introduction of new errors. Evaluation of the toolkit shows that it is highly applicable, and developers already find the transformations very useful and are looking forward to using our toolkit. We hope that our study motivates other follow-up studies to fully understand the state of the art in asynchronous programming.

9. ACKNOWLEDGEMENTS

This research is partly funded through NSF CCF-1213091 and CCF-1219027 grants, a SEIF award from Microsoft, and a gift grant from Intel. The authors would like to thank Cosmin Radoi, Yu Lin, Mihai Codoban, Caius Brindescu, Sergey Shmarkatyuk, Alex Gyori, Michael Hilton, Felienne Hermans, Eren Atbas, Mohsen Vakilian, Alberto Bacchelli, Dustin Campbell, Jon Skeet, Don Syme, Stephen Toub, and anonymous reviewers for providing helpful feedback on earlier drafts of this paper.

10. REFERENCES

[1] J. Albahari and B. Albahari. *CSharp 5.0 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2012.

[2] Adsclient App. August'13, `https://github.com/roelandmoors/adsclient`.

[3] Cimbalino-Phone-Toolkit App. August'13, `https://github.com/Cimbalino/Cimbalino-Phone-Toolkit`.

[4] Phoneguitartab App. August'13, `http://phoneguitartab.codeplex.com/`.

[5] Playerframework App. August'13, `http://playerframework.codeplex.com/`.

[6] Softbuild.Data App. August'13, `https://github.com/CH3COOH/Softbuild.Data`.

[7] Scala Async. August'13, `http://docs.scala-lang.org/sips/pending/async.html`.

[8] Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. Pause n Play: Formalizing Asynchronous CSharp. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP '12, pages 233–257, 2012.

[9] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How developers use the dynamic features of programming languages. In *Proceeding of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 23–32, 2011.

[10] WPCollector Source Code. August'13, `https://github.com/semihokur/wpcollector`.

[11] CodePlex. August'13, `http://codeplex.com`.

[12] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, 2009.

[13] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. Relooper. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 793–794, 2009.

[14] Indulged flickr App. August'13, `https://github.com/powerytg/indulged-flickr`.

[15] Windows Forms. August'13, `http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx`.

[16] Windows Presentation Foundation. August'13, `http://msdn.microsoft.com/en-us/library/ms754130.aspx`.

[17] Gartner. August'13, `http://www.gartner.com/newsroom/id/2153215`.

[18] Github. August'13, `https://github.com`.

[19] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 11–21, 2010.

[20] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the Foundations of Software Engineering*, FSE '13, pages 543–553, 2013.

[21] Best Practices in Asynchronous Programming. August'13, `http://msdn.microsoft.com/en-us/magazine/jj991977.aspx`.

[22] iRacerMotionControl App. August'13, `https://github.com/lanceseidman/iRacer_MotionControl`.

[23] Oracle JavaAWT. August'13, `http://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html`.

[24] Oracle JavaSwing. August'13, `http://docs.oracle.com/javase/7/docs/technotes/guides/swing/`.

[25] Siim Karus and Harald Gall. A study of language usage evolution in open source software. In *Proceeding of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 13–22, 2011.

[26] Survival of the Forgest. August'13, `http://redmonk.com/sograd y/2011/06/02/blackduck-webinar/`.

[27] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 54–65, 2012.

[28] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. Adoption and use of Java generics. *Empirical Software Engineering*, 18(6):1047–1089, 2013.

[29] Microsoft Asynchronous Programming Patterns. August'13, `http://msdn.microsoft.com/en-us/library/jj152938.aspx`.

[30] The Roslyn Project. August'13, `http://msdn.microsoft.com/en-us/hh500769`.

[31] Cosmin Radoi and Danny Dig. Practical static race detection for java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA '13, pages 178–190, 2013.

[32] Cimbalino Pull Request. August'13, `https://github.com/Cimbalino/Cimbalino-Phone-Toolkit/pull/21`.

[33] OCell Pull Request. August'13, `https://github.com/gjulianm/Ocell/pull/27`.

[34] ReSharper. August'13, `http://www.jetbrains.com/resharper/`.

[35] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, December 2010.

[36] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Refactoring Java programs for flexible locking. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 71–80, 2011.

[37] Windows Store. August'13, `http://www.windowsphone.com/en-us/store`.

[38] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 175–189, 2011.

[39] Our Companion Website. August'13, `http://learnasync.net`.

[40] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the 7th Joint Meeting of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, FSE '09, pages 173–182, 2009.