Proposals for Modular Asynchronous Web Programming: Issues and Challenges

Hiroaki Fukuda[1(✉)] and Paul Leger[2]

[1] Information Science and Engineering, Shibaura Institute of Technology, Tokyo, Japan
hiroaki@shibaura-it.ac.jp
[2] Escuela de Ciencias Empresariales, Universidad Católica Del Norte, Región de Antofagasta, Chile

Abstract. Because of the success in the Internet technologies, traditional applications such as drawing and spreadsheet software are now provided as web applications. These modern web applications adopt asynchronous programming that provides high responsive user interactions even if an application works without multi-threading. At the same time, as the scale of these applications becomes large, modular programming becomes important because it allows developers to separate concerns, meaning that the evolution of one module does not affect other modules. However, applying asynchronous and modular programming is difficult because asynchronous programming requires uncoupling of a module into two sub-modules, which are non-intuitively connected by a callback method. The separation of the module spurs the birth of other two issues: *callback spaghetti* and *callback hell*. Some proposals have been proposed without the lack of issues about modular programming. In this paper, we compare and evaluate these proposals applying them to a non-trivial open source application development. We then present a discussion on our experience in implementing the application using these proposals. Finally, we point out challenges that this kind of proposal should overcome toward a modular programming.

Keywords: Virtual block · Asynchronous programming · Aspect-oriented programming

1 Introduction

With the growth of high speed network and the variety kinds of computers that possess high computation capabilities, traditional standalone applications such as drawing and spreadsheet software are now provided using web technologies, namely web applications. Compared to traditional web applications, such modern applications adopt asynchronous behavior such as ajax that provides high responsive user interaction even if an application works without multi-threading. At the same time, the scale of such applications becomes large where modular programming becomes important because it allows separating concerns to be localized, meaning that the modification of one concern does not affect other concerns (*e.g.,* other modules). The basic idea of asynchronous programming is to decompose a blocking

© Springer International Publishing Switzerland 2015
F. Daniel and O. Diaz (Eds.): ICWE 2015 Workshops, LNCS 9396, pp. 91–102, 2015.
DOI: 10.1007/978-3-319-24800-4_8

operation that waits for its completion into a non-blocking operation that immediately returns control. Whenever the non-blocking operation execution ends, a piece of code known as *callback method*, is invoked. Therefore, the use of callbacks in asynchronous programming comes with issues that affect the modular development of software. The most notable two issues are *callback spaghetti* [7] and *callback hell* [8]. Callback spaghetti refers to the concern implementation that has a complex and tangled control structure because of continuations over callback methods. Callback hell refers to deeply-nested callbacks that have dependencies on data returned from previous asynchronous invocations. Some proposals have used to address these issues such as *async/await* from C# [1], *Promise pattern* [3] from JavaScript and SyncAS [4] from ActionScript. This paper compares and evaluates these proposals applying them to a non-trivial open source application development, called FlickrSphere [10] that is originally implemented by ActionScript3 and uses nested iterative callbacks, leading complicated control flows. The paper then presents a discussion on our experiences in its implementation.

Paper Roadmap. Section 2 gives an introduction to asynchronous programming and its problems. Section 3 briefly describes about FlickrSphere. In Section 4, we apply each one of three proposals to FlickrSphere to address asynchronous issues, we then present our experiences. Section 5 presents conclusions and future work.

2 Asynchronous Programming Problems

Asynchronous programming is now widely-adopted between mainstream programmers [1]. This section briefly describes asynchronous programming comparing to synchronous programming.

2.1 Synchronous Programming

Synchronous programming is the standard style used by programmers to write pieces of code. Listing 1.1 shows two classes: ImageViewer and Request. The ImageViewer class contains two methods: showFromURL and checkAndConvertToImage. The first method downloads data from a url and uses the second method to convert this data to an image if it passes an integrity check. The Request class has the send method that actually downloads some data by using the download method of the Downloader class. For this example, we assume download is a blocking operation, which takes a significant period of time. Because of several advantages such as reusability, maintainability and adaptability, dividing a system into a composition of modules is natural [9]. Therefore, in Listing 1.1, Request encapsulates how to get data from sources. As shown in Listing 1.1, in synchronous programming, we can obtain the result of a method invocation directly, then pass it to the next invocation as an argument, making the control flow clear.

2.2 Asynchronous Programming

Asynchronous programmign style makes it complicated to understand the control flow from pieces of code. Listing 1.2 shows the rewritten program of

Listing 1.1 replacing a blocking operation (download) with an non-blocking operation (downloadAsync). Three major changes are found in Listing 1.2. First, invoking checkAndConvertToImage is removed from showFromURL because send, that invokes downloadAsync, returns immediately without any data. Instead, the reference of the showFromURL is passed as a callback by using next, which is defined in Request. Second, a variable checksum must be defined in ImageViewer to keep the argument checksum in showFromURL because checkAndConvertToImage only accepts one parameter. Third, the show function call must be moved to checkAndConvertToImage because showFromURL does not contain the image. As described previously in this section, a module that uses a non-blocking operation requires representing the continuation as a callback. As a consequence, if the location of the continuation is far from the call-site, understanding control flow is difficult, leading to callback spaghetti. Moreover, a change of implementation inside of one module (*e.g.,* Request) may require call-site modifications in other modules (*e.g.,* ImageViewer), breaking modular principles. Besides, understanding control flow becomes difficult at a glance.

```
class ImageViewer {
  function showFromURL(url:URL,checksum:Function):void {
    var e:Event = new Request().send(url);
    var img:Image = checkAndConvertToImage(e,checksum);
    show(img);
  }
  function checkAndConvertToImage(e:Event,checksum:Function) {
    if (checksum(e.data))
      return convertToImage(e.data);
    else //throw an exception
} }
class Request {
  function send(url:URL):Event {
    return new Downloader().download(url);
} }
```

Listing 1.1. A synchronous version of a remote image viewer.

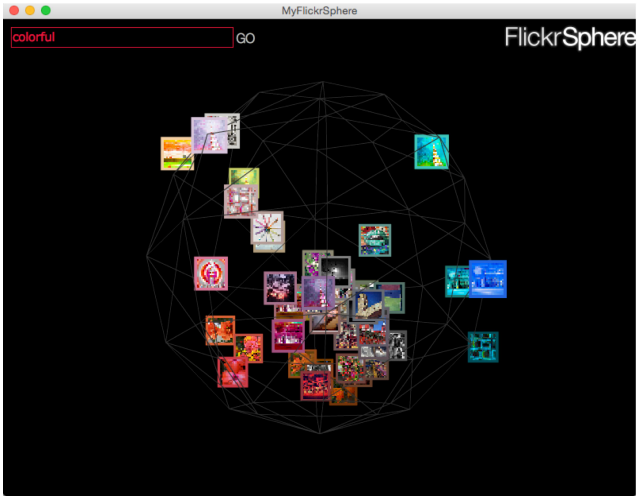

Fig. 1. A screenshot of FlickrSphere.

```
class ImageViewer {
  var checksum:Function;

  function showFromURL(url:URL, checksum:Function):void {
    this.checksum = checksum;
    var request = new Request().next(checkAndConvertToImage);
    request.send(url);
  }
  function checkAndconvertToImage(e:Event):Image {
    if (this.checksum(e.data)) {
      var img:Image = convertToImage(e.data);
      show(img);
    }
    else //throw an exception
} }

class Request {
  var nextF:Function;
  function next(f:Function):void {
    nextF = f;
  }
  function send(url:URL):void {
    var dl = new Downloader();
    dl.addEventListener(Downloader.Complete, callback);
    dl.downloadAsync(url);
  }
  function callback(e:Event):void {
      nextF(e);
} }
```

Listing 1.2. An asynchronous version of a remote image viewer.

3 Nested and Iterative Asynchronous Executions

FlickrSphere is an open source Web application implemented in ActionScript3. Since ActionScript3 runtime does not provide *threads* for concurrent executions, programmers need to use asynchronous programming if needed. This section briefly describes the behavior of FlickrSphere that uses nested and iterative asynchronous executions, then explains its original implementation.

3.1 FlickrSphere in a NutShell

FlickrSphere accepts keywords from users, and accesses to the flickr web service [2] to get all URLs matched by the keywords. Then FlickrSphere downloads all images according to these URLs. Everytime a image is completly downloaded, FlickrSphere displays the image on a circle that spins in the center of the screen. If a user searches during downloading images, FlickrSphere cancels current downloads, then starts a new search after the completion of the cancel operation. In addition to these main behaviors, FlickrSphere provides *Demo Mode* that plays a search with a certain keyword automatically to show the behavior of FlickrSphere.

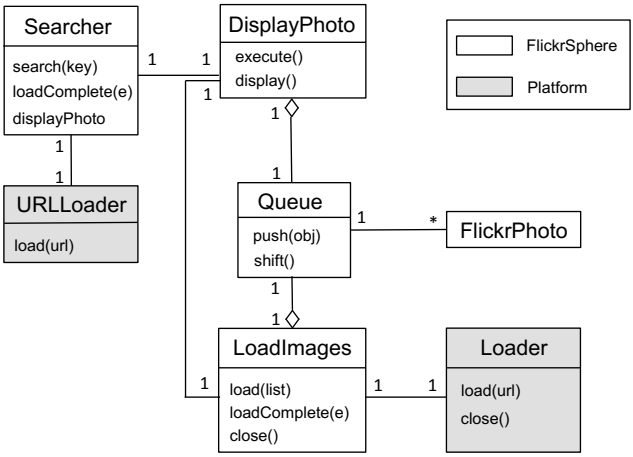

Fig. 2. A class diagram of the original implementation in FlickrSphere.

3.2 FlickrSphere Implementations

FlickrSphere is an open source application that shows images downloaded from Flickr [2]. Figure 1 and Figure 2 show a screenshot and a simplified class diagram of the original FlickrSphere[1] respectively. The main behavior of FlickrSphere is provided by two phases: downloading a list of URLs matched by a given keyword, and displaying each image after the download is completed. The main behavior carries out nested and iterative asynchronous executions. Listing 1.3 shows the piece of code that executes these two phases. For the first phase, Searcher directly uses URLLoader that is provided by the Flash runtime and uses a non-blocking operation (*i.e.,* load in Line 5), meaning that a callback (*i.e.,* loadComplete in Line 7) is required. The loadComplete method receives a list that contains all URLs of images. For the second phase, LoadImages uses a non-blocking operation like load), then pushes the downloaded images into a Queue. LoadImages also removes the URL of the image from a list (passed in Line 21), then the same process is repeated until the URL list becomes empty. At the same time, the display is invoked with a delay using a timer (lines 20 and 22). The display method gets the downloaded image from Queue, then renders it on the circle. The rendering images repeats until the the Queue becomes empty. As shown in Figure 1, LoadImages and DisplayPhoto share a queue to pass/receive images. Therefore, at a glance, it is not easy to understand the connection between downloading and rendering of images because execute does not directly invoke display using downloaded data.

Moreover, display is implicitly invoked by a timer because the rendering of images is faster than that of downloading. If this delay is not used, the repeat of

[1] We only show the classes required by the main behavior of FlickrSphere (*e.g.,* download and display images).

display unnecessarily consumes CPU resources. Although this delay is necessary, these pieces of code are difficult to understand at a glance. These pieces of code can be removed if the reference of display is passed to LoadImages then invoked inside a callback method (*i.e.,* loadComplete), making *callback spaghetti*.

```
1  class Searcher {
2    function search(key:String):void {
3      var loader:URLLoader = new URLLoader();
4      loader.addEventListener(Event.Complete, loadComplete);
5      loader.load(createURL(key));
6    }
7    function loadComplete(e:Event):void {
8      var list:Array = createList(e.data);
9      new DisplayPhoto().execute(list);
10   }
11 }
12
13 class DisplayPhoto {
14   private queue:Queue = new Queue();
15   private timer:Timer;
16   private loaded:Boolean = false;
17   function execute(list:Array) {
18     var loadImages = new LoadImages(queue);
19     timer = new Timer(2);
20     timer.addEventListener(TimerEvent.Timer, display);
21     imageLoader.load(list);
22     timer.start();
23   }
24   function display() {
25     if (queue.length == 0 && loaded) {
26       timer.stop();
27       timer.removeEventListener(TimerEvent.Timer, display);
28       return;
29     }
30     if (queue.length == 0) return;
31     var img:FlickrPhoto = queue.shift();
32     showImage(img);
33   }
34 }
```

Listing 1.3. A simplified implementations of FlickrSphere.

4 Applying Existing Proposals to FlickrSphere

This section presents different FlickrSphere implementations using existing proposals like async/await, Promise pattern, and SyncAS. While we present each implementation, we briefly explain each proposal.

4.1 The async/await Constructs

The async/await constructs is a proposal that supplies writing programs with non-blocking operations in a synchronous fashion for C# 5.0 [1]. A method invocation attached to await in order to keep the following executions as continuations that are restarted when the method execution is completed. The method usually contains an operation that takes a certain period of time (*e.g.,* LoadImages.load). Meanwhile, a method definition with async modifier lets the compiler know what the method contains a method invocation that uses non-blocking operations.

```
class Searcher {
  void async search(String key) {
    ListLoader loader = new ListLoader();
    list = await loader.load(key);
    new DisplayPhoto().execute(list);
  }
}
class DisplayPhoto() {
  LoadImages loader;
  void execute(Array list) {
    if (list) {
      loader = new LoadImages();
      display(list);
    }
  }
  void async display(Array list) {
    for (int i = 0; i < list.length; ++i) {
      Image img = await loader.load(list.get(i));
      if (img) showImage(img);
      else i--;
} } }
```

Listing 1.4. Simplified main behavior of FlickrSphere with async/await.

Listing 1.4 shows the rewritten code using async/await. In Listing 1.4, we introduce ListLoader because await can be attached only to a method invocation that uses non-blocking operations and we assume ListLoader uses URLLoader inside the load. Using async/await enables writing pieces of code that contain asynchronous executions as synchronous fashion, making them easy to understand and intuitive.

4.2 Promise Pattern

One approach to deal with asynchronous issues adopted by JavaScript communities is the Promise pattern [3]: a proxy object that represents an unknown (or future) result that is yet to be computed. The common term used for promise is *thenable*, as a programmer uses a then method to attach callback methods to a promise when it is fulfilled.

```
class Searcher() {
  function search(var key) {
    loadList(key).then(new DisplayPhoto().display, error); // use a promise using then
  }
  function loadList(var key) { // create a promise object
    var p = new Promise();
    var loader = new URLLoader();
    loader.addEventListener(Event.Complete, function(e) {
      (e.data) ? p.resolve(e.data) : p.reject("error"); // choose which methods a promise invokes (success or fail)
    });
    loader.load(key);
    return p;
  }
}

class DisplayPhoto() {
  var loader = new Loader();
  var list;
  function execute(list) {
    this.list = list;
    display();
  }
  function display() {
```

```
24      load(list.shift()).then(show, retry);
25    }
26    function load(url) {
27      var p = new Promise();
28      loader.addEventListener(Complete, function(e) {
29        (e.data) ? p.resolve(e.data) : p.reject(url);
30      });
31      loader.load(url);
32      return p;
33    }
34    function show(img) {
35      showImage(img);
36      if (list.length > 0) display();
37    }
38    function retry(url) {
39      list.unshirt(url);
40      display();
41    }
42  }
```

Listing 1.5. Simplified main behavior of FlickrSphere with Promise.

Figure 1.5 shows the rewritten code with the Promise pattern. Promise requires decomposing a set of operations into methods, then a method combines them creating a promise object and using then. Thereby, we create loadList (Line 5) to have a promise object and use it inside of the Searcher.search method. In the DisplayPhoto class, load is introduced, and display is rewritten to use the promise pattern. As a consequence, pieces of code that represent iterative display executions are non-intuitive because they use recursive invocations. Moreover, these recursions consist of a set of methods (*i.e.,* display, show, and retry), increasing its complexity. Note that, using loop statements such as while in display is impossible because it starts downloading and rendering all images at a time, leading a different behavior from the original FlickrSphere implementation.

4.3 SyncAS

SyncAS is a proof-of-concept library to provide *virtual block*, which enables a programmer to virtually block a method execution without blocking the execution of the program. A programmer specifies the points where a execution should be stopped and restarted using an aspect-oriented approach [6]. As a consequence, programmers can write programs as synchronous fashion even if they use non-blocking operations similar to async/await.

```
1  class Searcher {
2    function search(key:String):void {
3      var loader:ListLoader = new ListLoader();
4      var list:Array = loader.load(key); // a method invocation containing non-blocking operations
5      new DisplayPhoto().execute(list);  // virtually blocked by an aspect
6    }
7  }
8  class DisplayPhoto() {
9    private loader:LoadImages;
10   function execute(list:Array) {
11     if (list) {
12       loader = new LoadImages();
13       display();
14     }
15   }
16   function display(list:Array) {
17     var url:String = list.shift();
18     if (url) {
```

```
19      var img:FlickrPhoto = loader.load(url);      // a method invocation containing non-blocking operations
20      (img) ? showImage(img) : list.unshift(url); // virtually blocked by an aspect
21      display(list);                               // self recursion
22  } } }
```

Listing 1.6. Simplified main behavior of FlickrSphere with SyncAS.

Listing 1.6 shows the rewritten code with SyncAS. Similar to async/await, SyncAS enables virtually blocking a method invocation that uses non-blocking operations. With SyncAS, we can write search in a synchronous manner without the need to add constructs like found in async/await. Instead, the ListLoader.load method is a method that contains a non-blocking operation, thereby, we need to deploy an *aspect* to virtually block the execution of Line 5 and restart them when loadComplete is finished as follows.

```
SyncAS.addAsyncOperation("ListLoader.load","ListLoader.loadComplete");
```

In addition, compared to loop constructs (*e.g.,* for) used in async/await, self recursive iterations are a bit non-intuitive, however, this recursion in SyncAS is easier than iterations over multiple methods used in Promise because this iteration can be naturally written using self recursion. To virtually block Line 19 in Listing 1.6, we need to deploy another aspect as follows.

```
SyncAS.addAsyncOperation("Loader.load","Loader.loadComplete");
```

4.4 Discussion

Applying each proposal can remove a timer, which connects LoadImages and Displayphoto as shown in Figure 1 and Listing 1.3, making pieces of code more intuitive. In addition, as shown in Table 1, we evaluate these proposals from three aspects: Modularity, Expressiveness, and Overload. *Modularity* refers to how we can concentrate one concern on one place. *Expressiveness* refers to how we can write programs naturally and intuitively. Finally, *Overload* refers to the difficulty that introduces each proposal.

Table 1. Comparison of proposals in terms of Modularity, Expressiveness, Overload

	async/await	Promise	SyncAS
Modularity	Middle	Middle	High
Expressiveness	High	Low	Middle
Overload	Thread level	Very low	Additional closure execution

The async/await proposal provides an appropriate solution that enables writing pieces of code as synchronous fashion. We can define a method that contains non-blocking operations at one place and use these constructs inside loop executions, thereby, its expressiveness is considered *high*. Since the base technique of this proposal is *Thread*, the overload is equivalent to Thread. Meanwhile,

async/await does not hide asynchronous executions completely because programmers, who just use a method containing non-blocking operations, need to have concerns about behaviors (*i.e.,* synchronous/asynchronous) in addition to its definition (*e.g.,* interface). This is because these programmers explicitly need to write async/await in order to control executions. As a consequence, the modularity of this proposal is not considered high (*i.e., Middle*).

The Promise pattern enables writing nested non-blocking operations at one place with a fluent interface using then. However, callback methods are necessary to follow the promise style, bringing modularity issues like callback spaghetti (*Low* expressiveness and *Middle* modularity). Moreover, iterative executions with non-blocking operations may bring complicated control flow (*e.g.,* recursive executions over methods) because understanding iterative and recursive executions is difficult at a glance. The overload is really *low* because this proposal is only a design pattern.

Although SyncAS has similar features to async/await, a SyncAS programmer, who uses a method containing non-blocking operations, does not need to have aware about behavior of methods. However, a programmer who provides asynchronous methods also needs to provide aspects that control asynchronous executions. This fact means that SyncAS is more modular than async/await and enables dividing programmers into two categories: non-asynchronous programmers who just use a method containing non-blocking operation, and asynchronous programmers who know and control asynchronous executions, enhancing modular development. Thereby, its modularity is the *highest* of these three proposals. Meanwhile, the current SyncAS forces programmers to write iterative asynchronous executions using self recursion instead of loop statements, leading to non-intuitive programs. Therefore the expressiveness is lower than async/await but higher than Promise (*Middle* expressiveness). As SyncAS requires the support of aspect-oriented programming that uses closure to weave an advice, its overhead depends on the *additional closure execution* time. In Listing 1.6 implementation, the additional execution time was 10[ms] while the downloading one image file took 500[ms], meaning that the effect of the additional time could be limited becuase asynchronous programming is usually applied to the execution that takes certain time.

Based on previous evaluations, a proposal that enables writing asynchronous executions as synchronous fashion enhances modular asynchronous Web programming. In this context, SyncAS is more modular than other two proposals, however, its lack of expressiveness and execution overhead is worse than other proposals. However its overhead is limited in practical usages because asynchronous programming is used when an execution takes a certain period of time. Introducing loop join point [5] and a sophisticated translator, which partially encapsulates programs related to asynchronous executions, are next challenges for current drawbacks.

5 Conclusion

Based on the progress in the Internet technologies, traditional applications are now provided as web applications. These modern web applications adopt asynchronous programming for various reasons: hiding latency of the network and improving the responsiveness in the user interface. Callback is a typical solution that enables asynchronous programming. However, this solution has its drawbacks such as *callback spaghetti* – the modern goto statement in asynchronous programming. In addition, introducing asynchronous programming into module based programming requires dividing a method into call-site and its continuations, making complex control flows. In order to solve these drawbacks, some proposals are available, however, issues related to modular programming, expressiveness, complexity are still present. In this paper, we evaluated and compared three proposals: async/await, Promise pattern, and SyncAS, applying them to a non-trivial open source application called FlickrSphere. From a modular programming view point, SyncAS is better than other two proposals because can encapsulate non-blocking operations in a module completely. From an expressiveness viewpoint, async/await is better due to supporting of loops (*e.g.,* for). Finally, the Promise pattern is only useful when developers need a lightweight solution. However, as we can appreciate in Table 1, none of these proposals fully support modular programming and expressiveness without adding a significant complexity.

Acknowledgments. This work was partially supported by JSPS KAKENHI Grant Number 26330089.

References

1. Bierman, G., Russo, C., Mainland, G., Meijer, E., Torgersen, M.: Pause 'n' play: formalizing asynchronous C♯. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 233–257. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-31057-7_12
2. flickr: http://www.flickr.com/
3. Friedman, D., Wise, D.: The Impact of Applicative Programming on Multiprocessing. Technical report (Indiana University, Bloomington. Computer Science Dept.), Indiana University, Computer Science Department (1976). http://books.google.co.jp/books?id=ZIhtHQAACAAJ
4. Fukuda, H., Leger, P.: A library to modularly control asynchronous executions. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2015). ACM Press, Salamnca, April 2015 (to appear)
5. Harbulot, B., Gurd, J.R.: A join point for loops in aspectj. In: Proceedings of the 5th International Conference on Aspect-oriented Software Development, AOSD 2006, pp. 63–74. ACM, New York (2006). http://doi.acm.org/10.1145/1119655.1119666
6. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C., Maeda, C., Mendhekar, A.: Aspect oriented programming. In: Muehlhaeuser, M. (general ed.) et al. Special Issues in Object-Oriented Programming (1996)

7. Mikkonen, T., Taivalsaari, A.: Web applications - spaghetti code for the 21st century. In: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications, SERA 2008, pp. 319–328. IEEE Computer Society, Washington, DC (2008). http://dx.doi.org/10.1109/SERA.2008.16
8. Ogden, M.: Callback hell. http://callbackhell.com/
9. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972). http://doi.acm.org/10.1145/361598.361623
10. Yossy:beinteractive: Flickrsphere. http://www.libspark.org/svn/as3/Thread/tags/v1.0/samples/flickrsphere/fla/FlickrSphere.html