Easing IoT Development for Novice Programmers Through Code Recipes

Fulvio Corno
Politecnico di Torino
Turin, Italy
fulvio.corno@polito.it

Luigi De Russis
Politecnico di Torino
Turin, Italy
luigi.derussis@polito.it

Juan Pablo Sáenz
Politecnico di Torino
Turin, Italy
juan.saenz@polito.it

ABSTRACT

The co-existence of various kinds of devices, protocols, architectures, and programming languages make Internet of Things (IoT) systems complex to develop, even for experienced programmers. Perforce, Software Engineering challenges are even more difficult to address by novice programmers. Previous research focused on identifying the most challenging issues that novice programmers experience when developing IoT systems. The results suggested that the integration of heterogeneous software components resulted one of the most painful issues, mainly due to the lack of documentation understandable by inexperienced developers, from both conceptual and technical perspectives. In fact, novice programmers devote a significant effort looking for documentation and code samples willing to understand them conceptually, or in the worst case, at least to make them work. Driven by the research question: "How do the lessons learned by IoT novice programmers can be captured, so they become an asset for other novice developers?", in this paper, we introduce *Code Recipes*. They consist of summarized and well-defined documentation modules, independent from programming languages or run-time environments, by which non-expert programmers can smoothly become familiar with source code, written by other developers that faced similar issues. Through a use case, we show how *Code Recipes* are a feasible mechanism to support novice IoT programmers in building their IoT systems.

CCS CONCEPTS

• **Social and professional topics** → **Computational science and engineering education**; **Software engineering education**; • **Computer systems organization** → **Embedded and cyber-physical systems**;

KEYWORDS

Novice programmers, Internet of Things, Documentation, Code Fragments

1 INTRODUCTION

The development of IoT systems is challenging. On one hand, it relies on various areas such as distributed systems, mobile computing, web information systems, and cloud computing, among others. On the other hand, it differs from mainstream mobile-app and client-side web application development in a sense that IoT developers are required to consider aspects such as: the multidevice programming; the reactive, always-on nature of the system; heterogeneity and diversity; the distributed, highly dynamic, and potentially migratory nature of software [8].

Naturally, these challenging issues are even more painful for novice programmers since they are not expected to have deep knowledge or experience in all those areas or aspects [5]. Our previous research aimed at identifying the pain points that novice programmers experienced when developing IoT systems [2]. An exploratory study was conducted among Electronic and Computer Engineering undergraduate students of a university course in which, following a project-based learning approach, three to four people groups were assigned to develop an IoT system. In accordance with the course learning goals, these IoT systems had to include mobile applications, web applications, cloud computing services, wearable devices, single-board computers, and IoT sensing devices [1].

The results from this exploratory study suggested that the integration of heterogeneous software components is one of the most painful issues. It commonly implies dealing with several protocols, formats, and authentication mechanisms, that are usually unknown to the students. Moreover, the lack of clear and complete documentation, or merely, the absence of documentation that can be understood by a novice developer, make this integration issue even more difficult to overcome.

Looking for solutions to support novice IoT developers in overcoming these integration issues, we noticed that despite the specificity of each project, implementations of the integration between software components were similar across most of them, especially when third-party services were involved. However, although the source code of the projects from the past years' courses is on GitHub, it was not being reused among groups in later versions of the course. Therefore, the lessons learned by a group when implementing its project is not useful for the next year's groups.

Taking into account the results of the exploratory study and the lack of code reuse between the course groups, we envisioned that the the solutions found by the students, that were finally included in the working prototype built at the end of the course, could become a valuable asset for the novices that are about to start implementing their projects. The source code of these prototypes reveals architectural decisions and strategies adopted by other groups to achieve the integration of diverse software components. This code should, therefore, provide some guidance to other programmers that are in the process of overcoming the same learning curve issues. Moreover, if documented, this code would be a solution to the reported lack of documentation understandable by inexperienced developers [9]. In fact, being able to observe how someone else coded, what others paid attention to, and how they solved problems all support learning better ways to code and access to superior knowledge [3].

The present work is driven by the research question: "How do the lessons learned by IoT novice programmers can be captured, so they become an asset for other novice developers?". The current proposal

aims at easing the learning curve to IoT novice developers, not by automating code reusing and hiding the code from the developers, but instead, by enabling non-expert programmers to easily become familiar with source code, written by other developers that faced similar issues.

2 USE CASE

As mentioned earlier, the results from our previous research [2] suggested that among the most challenging issues novices face when developing IoT systems, the integration with other software components was perceived by many students as the most painful issue. In particular, the integration with third-party APIs that require OAuth 2.0 authentication was a time-consuming and difficult task. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. Broadly speaking, this authentication protocol consists of a flow, with a set of roles (*resource owner, resource server, client, and authorization server*) interacting across various steps (*authorization request, access token request, and protected resource request*), and exchanging several resources (*authorization grant, access token, refresh token, redirect URI*).

In the development of IoT systems, OAuth authentication protocol becomes fundamental since most of the third party service APIs use it. The integration with the Fitbit activity tracker[1] is a concrete example of the OAuth protocol usage. In order to gather the data captured by this wearable device, the third party application (i.e., the one developed by the novices) must obtain users authorization through the OAuth protocol.

However, due to the roles, steps, and resources that the protocol comprises, the adoption of the OAuth authentication is not trivial. The appropriate implementation of this protocol requires a clear understanding of the various steps, both from the conceptual and the technical perspective. Novice programmers struggle considerably with the adoption of OAuth, mainly due to the lack of documentation that might be understandable by non-expert programmers.

Fitbit, for instance, has a documentation website[2] that provides guidance about the Web API for accessing data from Fitbit activity trackers. Although the developer's site has an API explorer built in Swagger, and an API debug tool, it does not provide a fully implemented functional source code sample. Moreover, despite the clarity, readability and good overall structure of the documentation, it is targeted at experienced programmers, as with most of the developer's documentation.

In this scenario, novice programmers are required to search code samples, willing to understand them conceptually, or in the worst case, at least to make them work. Typically, this involves the reference of the Google OAuth Client Library documentation, the Fitbit developers website, several posts published in Stack Overflow, and various code samples available on GitHub. Hence, from the experience of the novice programmers adopting the OAuth protocol, we have that: (i) a significant amount of effort is devoted looking for documentation and samples; (ii) just through the source code it is not possible to understand the whole learning process behind it; (iii) the code fragments must be surrounded by summarized, structured and well-defined documentation modules, so they become an asset for other IoT novice programmers.

[1] *Fitbit*, accessed October 6, 2017, https://www.fitbit.com
[2] *Fitbit Web API*, accessed October 6, 2017, https://dev.fitbit.com/reference/web-api/quickstart/

3 CODE RECIPES

Code Recipes aims at capturing the most important information and documentation about one or more code fragments, to ease the development of an IoT system for novice developers. *Code Recipes* are specified through a set of metadata and consist of multiple code fragments along with documentation and links to ease the understanding of such code, in order to implement a given integration between subsystems of an IoT system. The joint presence of metadata and links allow novice developers to explore alternative solutions and, at their will, deepen their knowledge about a specific IoT subsystem, thus contributing to their learning process.

Our approach lies in the fact that code examples, when used effectively, can be a powerful learning resource [4]. However, while examples are a valuable resource for programmers, the rich context surrounding them is often crucial for adaptation and integration [6]. This proposal enables the integration of several software components through code fragments that might belong to different programming languages and might be deployed across various runtime environments, as it is common in IoT systems. The decoupling between the recipes and the technological stack is fundamental given the heterogeneity of the software components that are involved in an IoT system. *Code Recipes*, therefore, are defined as summarized and well-defined documentation modules, independent from programming languages or run-time environments.

By defining *Code Recipes* as documentation modules structured around code fragments, they can be incorporated in various kind of tools that might handle them in the learning process, e.g., a wiki-style web application or an Integrated Development Environment (IDE) extension.

Code Recipes, therefore, expose four features:

- Although the *Recipes* are structured around source code fragments, they are much more than just code. They encompass information that, besides providing technical solutions, includes comments and documentation sources that account for the learning process that other novice IoT developers followed and the decisions they made to reach a solution.
- *Recipes* are not constrained to a specific architecture, programming language or run-time environment. This means, first, that this proposal is aware of the heterogeneous nature of IoT environments, and second, that is suitable to be used in multiple scenarios with IoT novice developers.
- *Recipes* are not isolated from each other, they are cross-linked on the basis of three criteria: alternative versions, other language versions, and related recipes. This feature enables the sharing of diverse learning experiences with their commonalities and their divergences.
- Technical speaking, a structured representation (e.g., in JSON or XML) of the *Code Recipes* enables the implementation of various kind of tools that might handle them. For instance, a

web application (as shown in Fig. 1), a web browser extension, or an IDE plugin.

```
1  {
2      "id": "1506954092",
3      "author": [{
4          "name": "Juan Saenz"
5      }],
6      "date": "21.9.2017",
7      "name": "Integration between Fitbit and Java",
8      "description": "Recipe to consume the Fitbit API using OAuth
           2.0",
9      "tags": ["fitbit", "java", "oauth 2.0", "api"],
10     "running_environment": "Server application built in Java",
11     "endpoints": ["Fitbit API"],
12     "ingredients" : [{
13         "name": "Fitbit account",
14         "description": "Fitbit accounts set up for read/write API
               access",
15         "url": "https://dev.fitbit.com/"}],
16     "dependencies" : [{
17         "name": "Maven",
18         "description": "Maven plugin for Eclipse installed",
19         "url": "http://www.eclipse.org/m2e/"}],
20     "code_fragments": [{
21         "programming_language": "Java",
22         "description": "This is the main class",
23         "documentation_urls": ["https://github.com/google-oauth-
               client"],
24         "name": "FitbitSample",
25         "source_code_url": "./1506954092/FitbitSample.java",
26         "ide": "Eclipse Neon",
27         "parameters": [{
28             "name": "SCOPE",
29             "description": "OAuth 2.0 permission for resources",
30             "data_type": "String",
31             "sample_value": "activity, heartrate, location,
                   nutrition"
32         }]
33     }],
34     "documentation_urls": ["https://stackoverflow.com/quest
           /9863836"],
35     "rating": "4.6",
36     "alternative_versions": ["1506957773", "1507562564"],
37     "other_languages_versions": ["1496761597"],
38     "related_recipes": ["1507302404"]
39 }
```

Listing 1: Code Recipe Sample

Listing 1 describes a possible structure of a *Code Recipes* in JSON format. First, each recipe is described through an **id** (timestamp), its **author name**, **publication date**, **name**, **description**, and **tags** (lines 2 to 9). Then, the subsystems that the recipe integrates are specified in the **endpoints** fields (lines 10 and 11). **Ingredients** (line 12) correspond to the requirements of the recipe. They can be technical requirements, such as the deployment of a specific kind of web server, or data requirements, such as creating a developer account and issuing API client credentials. **Dependencies** (line 16) refers to requirements associated with the source code, which are fundamentally libraries and packages that must be installed.

Most importantly, *Code Recipes* include one or more **code fragments** that can be implemented in different **programming languages** and **IDEs** (lines 20 to 33). Each fragment has a set of **parameters**, which are values specific to each implementation of the recipe. Besides the source code, recipes include the documentation that their authors consulted, both for the whole recipe as well as for its fragments. They can be specified in the **documentation URLs** fields (lines 23 and 34). Finally, *Code Recipes* can be linked to each other in three ways (lines 36 to 38): **alternative versions**, that point to other recipes targeted at implementing the same integration; **other language versions**, that point to implementations

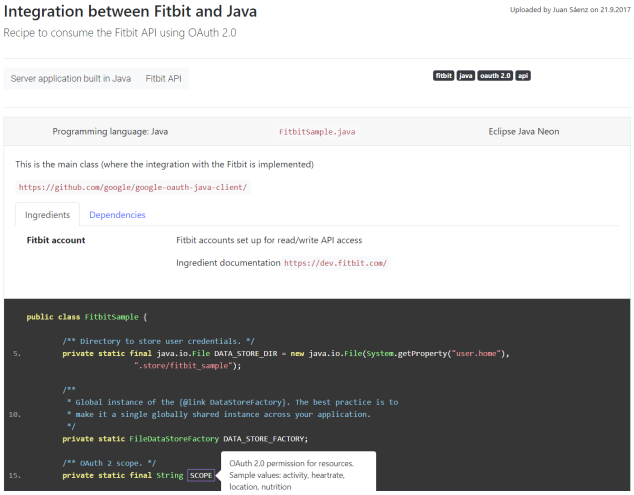

Figure 1: Code Recipe visualized in a web interface

of the same recipe in other programming languages; and **related recipes**, that correspond to other recipes that can be used as intermediate steps to implement the concerned recipe.

4 THE FITBIT OAUTH CODE RECIPE

With the use case described in Section 2 in mind, a *Code Recipe* was developed to illustrate how our approach might help novices to overcome integration issues through a collaborative approach. To develop this recipe, we took on the task of implementing a simple Java application to gather data from a Fitbit bracelet.

As mentioned before, no sample projects are provided in the Fitbit developers' website. Therefore, the first endeavor was to find a sample project in which the OAuth authentication was implemented using Java. After googling "*OAuth 2.0 Java Sample Code*", the second result took us to the documentation of the Google OAuth Client Library for Java (in the *Code Recipes*, this website would be included in the **`documentation_urls`** field). This website had setup instructions for Maven, the list of libraries that were required (in the *Code Recipes* would correspond to the **`dependencies`** field), the release notes of these libraries, and one sample code of the integration between a Java application and the Dailymotion API[3], using OAuth 2.0.

Once downloaded and imported the sample code into the IDE, the next step was to install and configure Maven, including the Project Object Model (POM) in which the dependencies of the project were defined. Later, when the Java project was already compilable, the next task was to identify which pieces of the code should be modified to achieve the integration with the Fitbit API (in the *Code Recipes*, these pieces are specified in the **`parameters`** field). Among the new data that had to be inserted into the code as parameters, there were the API key, the API secret, the Callback URL and the Scope. All of this data was obtained after completing the registration as a Fitbit developer (in the *Code Recipes* this registration accounts as an **`ingredient`**)

[3] *Dailymotion Developers - API*, accessed October 6, 2017, https://developer.dailymotion.com/

Afterwards, there was the source code itself. It consisted of three Java classes, two of which had to be parameterized. The explanation of the meaning of every parameter was available in the Fitbit developers website, along with their possible values (in the *Code Recipes*, these **`parameters`** can be documented through a **`description`**, their **`data_type`**, and a set of **`sample_values`**). Since this was the first *Recipe* that was developed, there were no other *Recipes* to link.

Across the whole implementation process, several documentation sources were consulted. The Google OAuth Client Library documentation, the Fitbit developers website, several posts published in Stack Overflow, and various code samples available in GitHub. Notwithstanding the fact that the *Code Recipe* was developed by an experienced programmer, its implementation was not trivial, and many of the issues expressed by the novices in our previous research were highlighted.

5 RELATED WORKS

Warner *et al.* [10] created *CodePilot*, a prototype IDE for novices. The tool enabled multiple users to connect to a web-based programming session and work together. According to the authors, *CodePilot* is the first attempt to integrate real-time collaborative coding, testing, bug reporting, and version control management into a unified system. This approach aims at lowering the entry barrier for novices by unifying the collaborative development workflow into a single IDE.

Oney *et al.* [6] developed and evaluated a mechanism they called *Codelets*. It consists of a block of example code and a user interactive helper widget that assists the developer in understanding and integrating the example. Through this interactive helper, web developers always have explanations attached to their code and can recall it if necessary. This approach allows maintaining a connection between example code and related documentation throughout the example's life-cycle.

Sidiroglou-Douskos *et al.* [7] presented a system named *CodeCarbonCopy* for transferring code from a donor application into a recipient application. This tool implemented an automatic data representation and naming translation between recipient and donor and a static analysis that automatically identifies and removes code that is irrelevant to the recipient.

Unlike *CodePilot* [10] and *Codelets* [6], *Code Recipes* are designed not to be tied to a specific programming language, IDE, or deployment environment. In our approach, IoT developers are intended to gain expertise understanding and adapting source code into their own implementations, despite the architectural decisions of the concerned system.

6 CONCLUSION

In view of the complexity that the development of IoT systems poses, particularly concerning the integration of heterogeneous software components, and taking into account the lack of documentation reported by novice programmers in our previous research, this paper presented *Code Recipes*. Code Recipes are summarized and well-defined documentation modules, non-dependent from programming languages or run-time environments, and structured around the code fragments that are required to implement some portions of an IoT system. Through this approach we aim at supporting novice IoT programmers, enabling them to easily become familiar with source code written by other developers that faced similar issues. Future work will concern: the development of a *Code Recipes* catalog; a web-based tool through which students can use them; and the subsequent evaluation in the context of the course.

REFERENCES

[1] Fulvio Corno, Luigi De Russis, and Dario Bonino. 2016. Educating Internet of Things Professionals: The Ambient Intelligence Course. *IT Professional* 18, 6 (Nov 2016), 50–57. https://doi.org/10.1109/MITP.2016.100

[2] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. 2017. Pain Points for Novice Programmers of Ambient Intelligence Systems: An Exploratory Study. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 250–255. https://doi.org/10.1109/COMPSAC.2017.186

[3] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work (CSCW '12)*. ACM, New York, NY, USA, 1277–1286. https://doi.org/10.1145/2145204.2145396

[4] Michelle Ichinco and Caitlin Kelleher. 2015. Exploring novice programmer example use. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 63–71. https://doi.org/10.1109/VLHCC.2015.7357199

[5] Hanna Maenpaa, Samu Varjonen, Arto Hellas, Sasu Tarkoma, and Tomi Mannisto. 2017. Assessing IOT Projects in University Education - A Framework for Problem-Based Learning. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. 37–46. https://doi.org/10.1109/ICSE-SEET.2017.6

[6] Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. ACM, New York, NY, USA, 2697–2706. https://doi.org/10.1145/2207676.2208664

[7] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. 2017. CodeCarbonCopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3106237.3106269

[8] Antero Taivalsaari and Tommi Mikkonen. 2017. A Roadmap to the Programmable World: Software Challenges in the IoT Era. *IEEE Software* 34, 1 (Jan 2017), 72–80. https://doi.org/10.1109/MS.2017.26

[9] Camilo Vieira, Alejandra J. Magana, Michael L. Falk, and R. Edwin Garcia. 2017. Writing In-Code Comments to Self-Explain in Computational Science and Engineering Education. *ACM Trans. Comput. Educ.* 17, 4, Article 17 (Aug. 2017), 21 pages. https://doi.org/10.1145/3058751

[10] Jeremy Warner and Philip J. Guo. 2017. CodePilot: Scaffolding End-to-End Collaborative Software Development for Novice Programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 1136–1141. https://doi.org/10.1145/3025453.3025876