REACT - A Process for Improving Open-Source Software Reuse

Alexander Lampropoulos[1], Apostolos Ampatzoglou[2], Stamatia Bibi[3], Alexander Chatzigeorgiou[4], Ioannis Stamelos[2]

[1] Arx.NET Mobile Innovations, Thessaloniki, Greece
[2] Department of Informatics, Aristotle University of Thessaloniki, Greece
[3] Department of Informatics & Telecommunications, University of Western Macedonia, Greece
[4] Department of Applied Informatics, University of Macedonia, Greece
a.lampropoulos@ihu.edu.gr, apamp@csd.auth.gr, sbibi@uowm.gr, achat@uom.gr, stamelos@csd.auth.gr

Abstract— **Software reuse is a popular practice, which is constantly gaining ground among practitioners. The main reason for this is the potential that it provides for reducing development effort and increasing the end-product quality. At the same time, Open-Source Software (OSS) repositories are nowadays flourishing and can facilitate the reuse process, through the provision of a variety of software artifacts. However, up-to-date OSS reuse processes have mostly been opportunistic, leading to not fully capitalizing existing reuse potentials. In this study we propose a process (namely REACT) for improving planned OSS reuse practices, i.e., we define the activities that a software engineer can perform to reuse OSS artifacts. To illustrate the applicability of REACT, we provide an example, in which a mobile application is developed based upon the reuse of OSS artifacts. To validate the proposed process we compared the effort required to develop the application with and without adapting REACT process. Our preliminary results suggest that REACT may reduce up to 50% the effort required to build an application from scratch.**

Keywords— Software Reuse, Process Improvement, Pilot Study.

I. INTRODUCTION

The fact that open-source software (OSS) code reuse is being increasingly adopted by software companies and individual developers, becomes apparent if we take into consideration the continuous growth of the open-source software community. Reuse of OSS components in other OSS projects is intense: the reuse of code from OSS projects represents undoubtedly thousands of staff years and enormous amounts in development costs. In the literature software reuse appears in two major forms, planned and opportunistic reuse [6]. However, the results on the most fitting practice are controversial. Large organizations report on employing more formalized methods and software product lines; whereas small and medium size companies perform more opportunistic reuse [6]. In fact, when reusing open-source code many developers reuse code opportunistically by copying and pasting classes or packages to their own projects. Although the Bazaar approach in open-source development and reuse seems to be working pretty well, the more OSS components become widely available, the larger the need to analyze and systematically handle the reuse process. Ajila et al. conducted an empirical study which suggested that an organization can have important productivity and quality gains, if it implements OSS reuse in a planned way [1]. Therefore the need of a lightweight process for opportunistic reuse is considered beneficial for facilitating OSS reuse.

In this paper, we propose the *Reusable Artifact and Components Adoption* (ReACt) process, i.e., a lightweight process for applying reuse practices exploiting the impressive amount of source code that is available as open source. The process is decomposed into three main phases, each one split to a well-defined number of steps, and producing specific artifacts. To ease the understanding of the process we present an illustrative example. The usefulness of applying REACT is assessed through a pilot study. The rest of the paper is organized as follows: Section II presents related work, and Section III outlines the proposed process. In Section IV, we illustrate REACT by building a mobile application, based on OSS reuse. In Section V, we describe the pilot study design that has been performed for preliminarily assessing the usefulness of REACT. Finally, Section VI concludes the paper by summarizing our findings and referring to possible future work.

II. RELATED WORK

Software reuse has been on the radar of many researchers and practitioners due its potential to deliver quantum leaps in production efficiencies, while minimizing quality degradation during the maintenance phase [1]. Early studies focused on defining processes and tools that would allow the exploitation of the full potentials of software reuse [3]. Most of these studies focus on in-house reuse by establishing processes that a company should follow to develop reusable artifacts for being reused by the same company. However, in the last years, through the spread of open source software, third-party software reuse is very popular [7] Nowadays in most organizations software reuse is employed as opportunistic reuse, integrating third-party software that was not initially developed to be integrated or reused [6].

Open-source Software Reuse is a dominant form of third-party reuse mainly due to the availability of a variety of open-source libraries that cover a great range of horizontal and vertical application domains, in many cases being more intense compared to in-company closed source reuse [7]. Schwittek and Eicker [10] isolated 36 Java proprietary applications and explored the level of OSS reuse concluding that on average, seventy, third-party components are being reused. Research on OSS reuse the last years concentrates on the types of OSS reuse and the domains where OSS reuse is more intense. Among the most popular types of OSS reuse is black-box reuse [5] that usually requires changes to the existing architecture that will allow the integration of the reusable assets. White-box reuse on the other hand is limited to components relatively small in size and in some cases without performing any changes to the reused artifacts [7], [8]. This paper goes beyond current research by providing:

- A process to retrieve, evaluate and select the relevant third-party OSS components for reuse according to the application domain of a project.
- A mechanism for assessing the integration cost of the 3rd party OSS component by evaluating its quality.
- An exemplar application of REACT in a real project for the implementation of a Mobile Movie Management app.

III. REACT – AN OSS REUSE PROCESS

In this section, we discuss the OSS *Reusable Artifact and Components Adoption* (ReACt) process that can be used for facilitating the reuse opportunities offered by the plethora of available open-source projects. The proposed process is divided into three main phases: (a) Reuse Conception, (b) Reusable Assets Identification, and (c) Reusable Assets Adaptation. The steps of each phase and the developed artifacts (documentation artifacts) are depicted in Figure 1. An illustrative example that will assist the reader understanding the process is presented in Section 4.

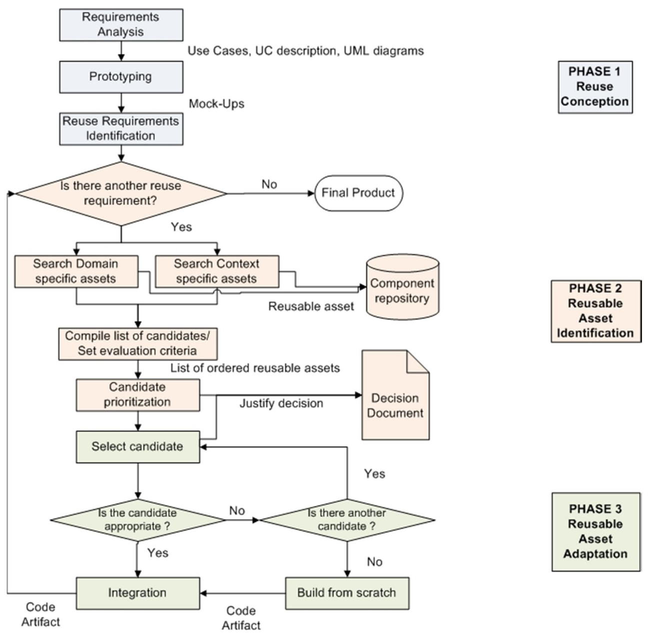
Requirements Analysis
Use Cases, UC description, UML diagrams
Prototyping
Mock-Ups
Reuse Requirements Identification
PHASE 1 Reuse Conception
Is there another reuse requirement?
No
Final Product
Yes
Search Domain specific assets
Search Context specific assets
Component repository
Reusable asset
PHASE 2 Reusable Asset Identification
Compile list of candidates/ Set evaluation criteria
List of ordered reusable assets
Candidate prioritization
Justify decision
Decision Document
Select candidate
Yes
Is the candidate appropriate ?
No
Is there another candidate ?
PHASE 3 Reusable Asset Adaptation
Yes
No
Integration
Build from scratch
Code Artifact
Code Artifact

Fig. 1. The REACT process for OSS reuse.

Phase 1: Reuse Conception Phase. During this phase the software engineer is expected to conceptualize the target system and defines the basic functional requirements of the application. The proposed representation method for functional requirements is the development of a use case model, i.e., use case diagram and use case specification (step 1.1), Next, in order for the software engineer to visualize the target application and identify additional requirements that stem from user interface (UI) design, in step 1.2, the reuser is expected to build mock-up of UIs (i.e., development of UI prototypes). As a final step of this phase (step 1.3) the software engineer is expected to catalogue the requirements that he/she aims to reuse from existing system. The outcomes of this phase are: the requirements specifications (ar.1.a), the UI mock-ups (ar.1.b), and a list of requirements to reuse (ar.1.c).

Phase 2: Reusable Asset Identification. This phase aims at the detection, evaluation, and prioritization of reusable assets that fit the concept of the target application. The efficient execution of this phase requires the existence of a component repository, in which the reuser can store candidate reusable artifacts. For the case of our study, we have selected to use the Percerons repository (*www.percerons.com*), which is an online platform that supports empirical software engineering. Percerons Client automatically identifies reusable assets, provides and initial structural evaluation, and then stores them in a repository which enables simple and advanced searches [2]. For searching purposes, the reuser has two options: (a) to use the global repository, which is populated with more than 1 million reuse candidates, or (b) use a local version of the repository, which will populate him/her-self.
This phase consists of 4 steps. In step 2.1, the *search process needs to be defined*, and the following sub-steps need to be made: (2.1.a) the reuser needs to identify OSS projects that will analyze and store in the repository. The projects should be context- and/or domain-specific. On the one hand, as context-related we consider those applications that aim the same deployment target (e.g., mobile, web, desktop, etc.), the same development paradigm (e.g., object-oriented), the same organization structure (e.g., Model-View-Controller, Model-View-Presenter, etc.), etc. On the other hand, as domain-related, we consider those applications that can be categorized in the same application domain (e.g., movies, games, audio, video, etc.). Upon the popularization of the repository, the reuser needs to define search terms (step 2.1.b) for each target requirement (ar.1.c), and execute the search (step 2.1.c). The next step (2.2) is the *compilation of a list of candidate reusable artifacts* for each requirement (artifact: ar.2.a). Next, during step (2.3) the reuser needs to define a set of criteria for evaluating his/her alternatives and develop a decision document (artifact: ar.2.b).
Phase 3: Reusable Asset Adaptation. The goal of this phase is the iterative parsing of the prioritized candidate reusable assets list, for all requirements. In this phase, first the software engineer *picks the top reuse alternative* from the stack (step 3.1), *attempts to adopt it into the current version of the target system* (step 3.2), if this is feasible, *he/she integrates it into the target system* (step 3.2.a) or goes back to step 3.1. When the reuse alternatives list has been exhaustively parsed, the software engineer *implements this requirement from scratch* (step 3.2.b). Steps 3.1 and 3.2 are performed iteratively for all targeted requirements (artifact: ar.1.c).

IV. ILLUSTRATIVE EXAMPLE

This section presents an overview of the Movie Management “demo” application destined for mobile devices that exemplifies REACT. We selected the particular domain due to the existence: (a) of many relevant open-source projects and (b) of available APIs that allows obtaining movie data.

Hence, we start from the ***Conception phase*** where we collect the *Functional Requirements* of the application and form the *Use Case Diagram* and the *Use case Description* along with the corresponding *Activity Diagrams*. In brief the end-user can search movies, see their details along with the trailers, he can navigate through the available genres and he can handle his watchlist. We acknowledge that the demo offers the minimum functionality to the end-user but still it is a fully operating Movie management app that serves the purposes of the illustrative example. In the next step we continue *Prototyping* the app by designing the *Mock-Ups* for the movie application that will enable us to visualize the use cases of the demo app in the display of a smart phone. During this step we had the opportunity to clarify the components that constitute the movie application and decide upon functionalities that are: (a) offered by Android and Google native components, and (b) custom functionalities that should be searched to external libraries. The mock-ups, help us in the next step to identify all the necessary requirements native and custom for building the app, and create a list of the *Reuse Requirements*. In this step we create a list of the required types of assets along with their description and we record the functionality expected from them so as to facilitate the search procedure of the components for reuse. The Reuse Requirements, grouped based on desired functionality (see Table I), serve as an input to Phase 2.

TABLE I. REUSE REQUIREMENTS AND RETRIEVED ASSETS

Functionality	Keywords/ Attributes	Max. extern. depend.	Min. Func.	Min. Reus.	No. of results
	SQLite	0	0	0	27
Database	Remove SQLite	0	0	0	6
	Database	5	5	2	21
	CRUD DB	5	5	2	29
	Request	5	5	2	60
	API	5	5	2	130
	Retrofit	5	5	2	10
	Glide	5	5	2	6
	Dagger	5	5	2	1
	Toast message	0	0	0	1
	Recycler view	5	5	2	8
Request	Adapter	5	5	2	84
	Dialog	5	5	2	47
	Search	5	5	2	108
	Bottom layout	0	0	0	0
	Floating button	0	0	0	0
	YouTube	5	5	2	16
	Tab Layout	0	0	0	0
	Pager	5	5	2	25
Models	Movie	5	5	2	200
	Genre	5	5	2	42

The **second phase** incorporates *the **Reused Artifacts Identification***. In the first step for each requirement recorded in the list of Reuse Requirements we searched OSS projects that could offer the relevant functionality. During the *Search process* we performed google searches and targeted searches to specific libraries and repositories as "*AndroidArsenal*", "*GitHub*" and "*Sourceforge*". The purpose of the searches was twofold: a) find projects within the same *Application Domain* (movie management) b) find projects designed within the same *Context*. In the last case we would like to reuse assets designed with the same approach as ours, employing clean architecture. The search terms employed in this demo application are presented in Table 1 and were derived as part of the requirements list identified in the previous phase. The next step after retrieving the relevant OSS projects was to isolate candidate reusable assets from these projects. We employed the Percerons repository for this purpose and created a local repository containing the candidate reusable assets from the relevant OSS projects. The *Prioritization of the Candidate assets* is performed at the end of this phase by defining a set of three criteria for evaluating the alternative candidate assets for reuse that include: (a) the functionality offered, (b) the assessment of the changes to the reusable asset (as an indicator the reusability index is taken into consideration), and (c) design- adaptation changes (as an indicator the number of external dependencies is used). Therefore for each candidate asset three metrics are calculated, with the help of Percerons [2]: *Functionality* is measured as the ratio of the number of classes outside the candidate component that use at least one class inside the component, to the total number of classes outside the component. *External dependencies* are calculated as the number of classes outside the component that are essential for the component to compile. *Reusability* is a compiled index that uses size, coupling, cohesion and messaging metrics.

As an output of this phase we have a decision document that justifies the selection of the candidate assets, provides the reasoning for their prioritization and summarizes the risks of each candidate selection. An example of the information provided in the decision document is presented in Table II. For each candidate asset a table is created, presenting among others the order of the asset in terms of reusability potentials when compared to the other candidates assets (see second line, second column of Table II).

In the third and final phase we performed the **Adaptation of the Reusable Assets**. In the majority of cases we selected the first candidate for being adopted in the target system. This is probably due to the fact that the intended architecture (i.e., use of a clean architecture through the Model-View-Presenter pattern) was an important criterion for prioritization, and it strictly defines the interaction among components. The final product was a mosaic of artifacts retrieved from various products, and thus made full benefit of the reuse opportunities offered by them (see Table III—Section V).

TABLE II. DECISION TABLE FOR REUSE ALTERNATIVE #1

Attribute Documented	Value
Functionality	add & SQLite (result: 1 of 27)
Name	MovieList
# Classes	1
Ext. Depend.	0.000
Functionality	1.000
Reusability	1.688
Description	Contains Create and Upgrade database, along with some useful tables (Overview, Rating, Release Date, Title Films)
Design rules	Some modifications might need, in order to use it with Room Add data method, should be created
Constraints	No design constrains seems to be occurred
Risks	One class only, so the risk for a fall-back is very low.
Consequences	General consequences, like usage of SQLite as a DB
Pros	Very low cost, some modifications needed
Cons	No Room integration, No add data method, Tables not useful

V. EMPIRICAL EVALUATION

In this section we present the design and the results of a pilot study for initially validating ReACT, with respect to: (a) the gained effort from its application, and (b) the dissimilarity level of the end product, compared to existing OSS solutions:

RQ$_1$: *What is the time benefit of applying ReACT to develop a product by reusing OSS artifacts, compared to development from scratch?*

We aim to validate that ReACT leads to decreased development effort, and the parameters that can affect its productivity.

RQ$_2$: *To what extend is the end-product developed by applying ReACT different from existing OSS solutions that have been fed to the ReACT process?*

We aim to reveal the extent to which the end-product is different compared to existing solutions that have been for identifying reusable assets. This is an important parameter, since reuse would be considered unsuccessful if in the end a product of very high similarity with an existing one would be built.

To collect the required data for answering the aforementioned research questions, the following variables have been recorded: [V1] ***effort estimation*** for developing (from scratch) each functional requirement at the analysis and design phase. The estimation has been performed by the experienced Android developer that implemented the demo application. [V2] **actual effort** for developing (by applying ReACT or from scratch—if no artifact could be used) each functional requirement, and [V3] ***source OSS system*** from which reusable asset has been retrieved, for each functional requirement.

To answer RQ$_1$, we compared the values of [V1] and [V2] in pairs. The analysis was three-fold: (a) we compared the total time required to develop the complete mobile application in both scenarios; (b) we compared the number of requirements for which ReACT provided a development effort benefit, and cases for which it lead to worse productivity; and (c) we performed hypothesis testing to check if the observed difference in productivity is statistically significant. To answer RQ$_2$, provide descriptive statistics on the number of reusable artifacts that have been obtained from every source (for reuse) OSS system. An interesting observation from Table 3, is that in most of the cases, the effort estimation of the developer at the analysis/design phases ([V1]) was very close to the actual effort required to develop from scratch ([V2]-scratch): this is an indication that other estimations are also accurate and thus the comparisons of estimations and actual values is quite safe.

TABLE III. STUDY DEMOGRAPHICS

Functional Requirement	[V1]	[V2]-reuse	[V2]-scratch	[V3]
Room Entity	960	480		MovieList
Request (Retrofit)	480	240		OMDB 4
Glide	120	100		UdacityPopularMovies
Toast	20	30		Vineyard
Recycler View & Adapter	240	60		MoviesTraktTV
Dialogs	60	180		None
Search	180		120	None
Bottom Sheet Layout	240		240	None
Floating Button	30		30	None
YouTube Trailer	20		30	None
Tab Layout & Pager	480	420		Traktoid
Movie Entity	60	5		RetrieveMovieList
Genre Entity	60	5		RetrieveMovieList
Cast Entity	60		60	None
Actor Entity	60		60	None
Dagger	480	340		MoviesTraktTV

Based on the results presented in Table III, the estimated total time to build this application from scratch was 59 hours, whereas the effort required to build the complete application, using ReACT, was 40 hours; leading to a time gain of approximately 30%. Regarding specific functional requirements, the reuse-based approach was faster than the estimation in 66% of the cases (8 requirements), whereas in the rest 33% it was more time consuming (occurred in cases when the selected reusable artifact had assigned a low priority). The average time to develop one requirement from scratch was 296 minutes (±301.8), compared to 186 (±176.6) when developing with ReACT. The difference between the two efforts is statistically significant, based on the Wilkoxon Signed Rank test (Z = -2.091 and p = 0.037). Regarding RQ$_2$, the results suggest that reusable artifacts from 6 different OSS projects have been used. The maximum functionality has been reused from *RetrieveMovieList* and *MoviesTraktTV* projects (each one contributing to 22.2% of the functionalities). Thus, we suggest that the similarity of the end outcome and the OSS projects that have been used to populate the repository is rather limited.

VI. CONCLUSIONS

In this paper we presented a process for facilitating OSS reuse. The main implications of this study can be summarized as follows: To reduce the adaptation and modification time of reusable components, we need to ***find components from projects with common specifications like clean architecture and MVP pattern etc***. Based on our experience, we suggest that, the adaptation time is lower for in-house reuse, since common specifications (structure, architecture, domain, design guidelines, patterns etc.) could be expected. An interesting future work would be to expand our study in the industrial sector, in cases where companies develop systems with common domains and guidelines, studying the effectiveness of the proposed approach, as a type of planned reuse strategy.

ACKNOWLEDGEMENT

This work was supported by the action "Strengthening Human Resources Research Potential via Doctorate Research" of the Operational Program "Human Resources Development Program, Education and Lifelong Learning, 2014-2020", implemented from State Scholarship Foundation (IKY) and co-financed by the EU Social Fund and the Greek public (National Strategic Reference Framework (NSRF) 2014 – 2020).

REFERENCES

[1] S. A. Ajila and D. Wu, "Empirical study of the effects of open-source adoption on software development economics", Journal of Systems and Software, Elsevier, 80 (9), pp. 1517-1529, September 2007.

[2] A. Ampatzoglou, I. Stamelos, A. Gkortzis, and I. Deligiannis, "Methodology on Extracting Reusable Software Candidate Components from Open-source Games", 16th MindTrek Conference, ACM, 2012.

[3] I. Crnkovic, B. Hnich, T. Johnson and Z. Kiziltan, "Specification, implementation, and deployment of components", Communications, Association of Computing Machinery, 45 (10), pp. 35-40, October 2002.

[4] W. B. Frakes and C. J. Fox, "Quality Improvement Using A Software Reuse Failure Modes Model", Transactions on Software Engineering, IEEE Computer Society, 22 (4), pp. 274–279, April 1996.

[5] S. Haefliger, G. von Krogh, and S. Spaeth, "Code Reuse in Open-source Software", Management Science, PubsOnline, 54 (1), Nov. 2007.

[6] S. Jansen, S. Brinkkemper, I. Hunink, and C. Demir," Pragmatic and opportunistic reuse in innovative start-up companies", IEEE software, 25(6), pp. 42-49, 2008.

[7] A. Mockus, "Large-Scale Code Reuse in Open-source Software", 1st Int. Workshop on Emerging Trends in FLOSS Research and Development (FLOSS' 07), IEEE, 2007.

[8] M. E. Paschali, A. Ampatzoglou, S. Bibi, A. Chatzigeorgiou and I. Stamelos , " Reusability of open-source software across domains: A case study". Journal of Systems and Software, Vol. 134, pp. 211-227, 2017.

[9] S. Raemaekers, A. van Deursen, and J. Visser, "An Analysis of Dependence on Third-party Libraries in Open-source and Proprietary Systems", 6th Int. Work. on Software Quality and Maintainability, 2012.

[10] W. Schwittek, and S. Eicker, "A Study on Third Party Component Reuse in Java Enterprise Open-source Software", 16th Symp. on Component-based Software Engineering (CBSE' 13), ACM, 75–80, 2013.

[11] M. Sojer and J. Henkel, "Code Reuse in Open-source Software Development: Quantitative Evidence, Drivers, and Impediments", Journal of the Association for Information Systems, 11 (12), 2010.