Challenges on Software Unbundling: Growing and Letting Go

João Bosco Ferreira Filho, Mathieu Acher, Olivier Barais

Challenges on Software Unbundling: Growing and Letting Go

João Bosco Ferreira Filho Mathieu Acher Olivier Barais
Inria and Irisa, Université Rennes 1, France
joao.ferreira_filho@inria.fr, mathieu.acher@irisa.fr, olivier.barais@irisa.fr

Abstract

Unbundling is a phenomenon that consists of dividing an existing software artifact into smaller ones. For example, mobile applications from well-known companies are being divided into simpler and more focused new ones. Despite its current importance, little is known or studied about unbundling or about how it relates to existing software engineering approaches, such as modularization. Consequently, recent cases points out that it has been performed unsystematically and arbitrarily. In this paper, our main goal is to present this novel and relevant concept and its challenges in the light of software engineering, exemplifying it with recent cases. We relate unbundling to standard software modularization, presenting the new motivations behind it, the resulting problems, and drawing perspectives for future support in the area.

Categories and Subject Descriptors D.2.9 [*SOFTWARE ENGINEERING*]: Distribution, Maintenance, and Enhancement

Keywords Unbundling, modularization, features, aspects, reengineering, refactoring, evolution

1. Introduction

Software is designed to meet user needs and requirements, which are constantly changing and evolving [19]. Meeting these requirements allows software companies to acquire new users. For example, mobile applications compete with each other to gain market share in different domains; they constantly provide new features and services for the end user, growing in size and complexity. In some cases, the software artifact absorbs several distinct features, overloading the application and overwhelming the user and his/her acceptance of the software product [9] – he/she has to carry dozens of Swiss Army knives in his smart phone.

A recent phenomenon is to unbundle these dense pieces of software into smaller ones, trying to provide simpler and more focused applications. *Unbundling consists of dividing an existing software artifact into smaller ones, each one serving to different end use purposes*; it requires an unplanned coarse-grained modularization of mature software.

The main claim and goal of modularization techniques when applied to mature code is to improve software properties like maintainability and understandability [15]. In unbundling, this is not the case, these desired good properties become means, instead of ends of the process. The goal is the division itself, in order to attend to market issues imposed by trends of usage and competition to gain market share; nevertheless, the good properties may work as enablers to accomplish unbundling.

Although the importance of unbundling in today's software industry, no studies have been conducted to conceptualize or analyse these challenges. This paper is a first step to fill this void. Our main contributions are: to define and analyse the unbundling phenomenon (Section 2), to present the main challenges of unbundling (Section 3), showing examples of this current phenomenon and explaining how it relates to existing software engineering approaches for software modularization, and to draw perspectives on how to facilitate and better exploit unbundling (Section 4).

2. The unbundling phenomenon

Recently, many well-known software companies started to divide their mobile applications into smaller ones. This is the case of Foursquare, Dropbox, LinkedIn, Evernote, Facebook and Google. Some of them, like Foursquare, have split into two, continuing with the original one and separating part of its features into a second brand new application (in this case, Swarm); some others unbundled into several applications, for example, LinkedIn originated Pulse, Connected, Job Search, Recruiter, Sales Navigator and Slide Share. IBM also adopted the strategy of dividing their software and services to better adequate to market and regulatory needs [8].

In all these cases, unbundling was essentially about identifying parts of the original software that could be isolated in separated applications. For doing this, these parts must be reengineered in a new software, according to a high-level end user *purpose*. A purpose is a high-level and often subjective end user goal when using the software product, it can be the reason why the user acquired the product (e.g., sharing photos, making todo lists, searching places, etc.), and it can gather a set of requirements or features of it. Ideally, one application should serve one highest purpose, however secondary purposes often start as small features of the application, and then they grow until they are a considerable part of the software, which can now be separated as a self-contained purpose.

Reengineering these parts that serve to different purposes comprehends: (1) decoupling them from other parts that will not be in the same application, so they can exist separately; (2) developing missing parts that were removed during refactoring or that are necessary to make the part usable (e.g., changing a user interface code to show only a group of functionalities). In the case of mobile applications, software companies are trying to follow the principle of having one big purpose per application, so the user knows clearly why and how to use it, avoiding numerous, cluttering and confusing features. In this way, application vendors create a strong identity for their products and link them well-defined purposes.

Figure 1 gives an overview of the unbundling process. Let us consider that an original software artifact Θ is a candidate to be

unbundled. Θ contains different parts $\Theta = \{a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3\}$ expressed in any unit (e.g., class, method, block, statement). These parts can be associated or intersect each other. Let us also consider that there exists subsets of parts in Θ that implement different high-level purposes, which is a motivation for unbundling Θ into different applications containing these different parts. In the example of Figure 1, the subsets of parts $\{a_1, a_2, a_3\}$ and $\{b_1, b_2, b_3\}$ are isolated into two new software Δ_1, Δ_2 because they serve to two distinct end user purpose. This isolation implies changing the structure of the part to make it decoupled or even adequating its behaviour to work in a different context (a_1 is reengineered into a'_1 in Figure 1).

As also illustrated in Figure 1, it is possible that the new software artifacts share common parts among them and with the original one (c_1, c_2, c_3). This is the case, for example, of parts responsible for the implementation of crosscutting concerns or of essential functionalities of any application derived from the original one, such as authentication, storage, cryptography modules, etc. It is also possible that the new software artifacts demand new parts ($n_1, n_2, n_3, n_4, n_5, n_6$) to implement new functionalities or to adapt the existing ones to the new context.

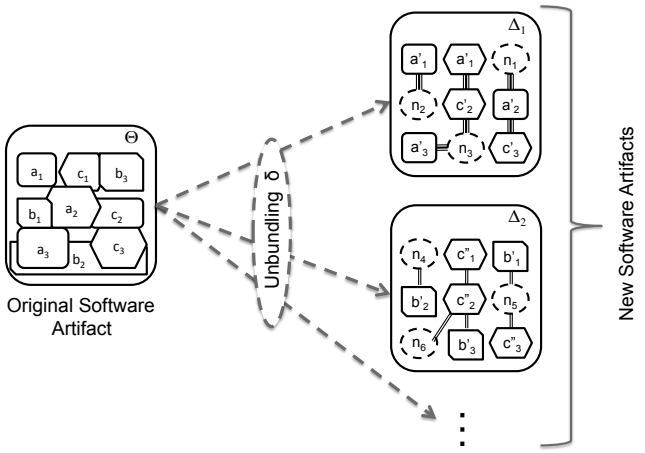

Figure 1. Unbundling.

In Figure 1, Δ_1 can be the original software without the secondary purpose parts, as it is of the company's interest to continue their original software product in order to keep its associated market share. However, if the result of the Unbundling δ was only Δ_1 without any other new applications, we would call it **Reductive Unbundling**. From this perspective, unbundling can serve to reduce the original application, removing near-unused/dead code without the preoccupation of making it usable in another application. The difference between actual dead code elimination [11, 24] and reductive unbundling is that, before the unbundling, the eliminated code could still be executed at run-time in the original application, which does not fit the definition of dead code.

In resume, if we think of *unbundling as a process*, we can enumerate key activities that it encompasses:

- Identifying distinct end user purposes when using a given original software;
- Identifying the parts of the original software that relates to each identified purpose.
- Extracting and reengineering the parts to be placed in the different new software artifacts;
- Identifying and extracting common parts so they can be reused.
- Occasionally implementing new parts that will complement the existing ones in the new software artifact.

3. Challenges

In this section, we present five key challenges of unbundling. In order to efficiently handle unbundling, it is essential to: (1) understand its new causes and objectives, which are different from standard modularization; (2) handle an unplanned need for correcting the software structure, so it can better evolve and grow; (3) manage the software parts from a high-level perspective related to an end user purpose, which is a coarser-grained abstraction when compared to existing software engineering abstractions; (4) handle the division itself and the isolation of existing parts of software to work on different applications; and (5) unbundling efficiently by avoiding code replication.

3.1 New causes and objectives

The *causes and objectives* of unbundling differentiates from the ones of modularizing, or simply componentizing [22], or aspectualizing [2, 14, 20] software. Since its origins, modularization seeks to improve flexibility and comprehensibility of a given software [18]; these goals have been enlarged to also consider maintainability and testability [13, 16].

In unbundling, the goal is beyond increasing maintainability, understandability, flexibility or testability of an application; it aims at separating the software artifact and creating new smaller and self-contained ones, which will then be managed by different engineers, used by different clients and placed in different domains. Essentially, the ends become the means: all software good properties are now means for the end of dividing the software, while before, modularizing and dividing the software artifact in modules were the means to reach such good properties.

Therefore, unbundling is triggered by business goals, it creates new opportunities with separable markets [8]; it is a specialization of software businesses. From the marketing perspective, unbundling is challenging for the company because there is a risk to lose clients in the transition to the new software. On the other hand, if the transition succeeds, it can open a new market and attract more clients.

From the software engineering perspective, modularization is a established good practice and it is driven by developers or stakeholders that are related to the development process. Market competition is often what drives unbundling – maintainability or any other software engineering concepts are less important when faced to the needs of competing to a market share or simply aligning the software to a new trend of use. The way that developers thought the software modularization can be very different from the division imposed by these high-level purposes. However, this does not prevent that software engineering best practices also come to be a trigger for unbundling, probably if the way the application is structured starts to interfere in the company's business.

3.2 Unplanned corrective evolution

Two essential characteristics present in software to be unbundled are: it is already mature, and the possibility of splitting it into several other software artifacts was not conceived in earlier stages of its development. Therefore, unlike, for example, Software Product Lines (SPL) [5], unbundling is not a long-term strategy that is planned in advance by the company. The fact that the software artifact is already mature and no longer in a conceptual phase implies that unbundling is a corrective action that has to handle code with all its existing good or bad design and implementation choices. Another challenge is that the software is up and running, having numerous clients relying on it, therefore the evolution must not in-

terfere in this relationship; this interference is very likely to happen if a bad release of the unbundled software is launched.

3.3 Coarser-grained modules

The criteria used when decomposing systems into modules have been studied for decades [18]; from sub-routines to features [10], the unity of modularity is a primary concern in software design and implementation. These units of modularity are thought by stakeholders directly involved with the code (e.g., developers, architects). As the motivation of unbundling is to separate distinct end user purposes residing in one single software artifact, it is necessary to group the set of software parts in the original software product that meets these purposes.

This coarser modularization raises the challenge of having to categorize the existing fine units and associate them to greater goals, which are not explicitly modeled in the software, but that are rather high level abstractions that are used to sell the software to a client. In the example of Foursquare and Swarm, the purpose of *checking in places* was extracted from the original application (Foursquare) and placed in a new one (Swarm); checking in places gathers functional and non-functional finer features of the software like: geographic location, map visualization, social network sharing, etc.

3.4 Dividing and isolating

Dividing software has always proved challenging; many different software engineering approaches have been proposed to decompose [17] code or slicing programs [23].

As explained in the last subsection, unbundling has to handle coarser abstractions, but still actually managing finer ones. The problem is that this is not always intuitive, as classes, components, aspects or features are not perfect modular units, they share dependencies and interactions [3]. Therefore, parts of the original software product may finish to be present in different purposes, or the functioning of a feature to be affected by the presence or absence of another, which makes the task of dividing and isolating purposes hard.

A great challenge is to make current componentization, apectualization, feature extraction [1] techniques to work seamlessly with an additional level of abstraction: the purpose. These techniques work by analysing the software and evaluating its parts, clustering them in categories according to their structure and semantics as criteria. In unbundling, this division must be enriched with the end user purpose, which will drive the division itself.

Considering the steps of the unbundling process, we can state that the essential challenges of this process are comprehended in dividing and isolating parts of software: identifying, extracting and reengineering existing features. Identifying is challenging because of the complicated matching between one single high-level and often subjective purpose to concrete fine-grained implementations. Extracting the features imposes the difficulty of not breaking the semantics of the code after removing parts of it, thoroughly analysing the dependencies from statement to component levels. And finally, reengineering the parts is challenging because it depends on the two past activities and because it demands knowledge on the new application environment in which the new parts are now placed.

Besides their own challenging nature, these activities can be hardened according to how strong is the relationship between modules of the original software artifact that belong to different purposes (i.e., coupling degree). On the other hand, unbundling may be facilitated if the modularity units belonging to a purpose present a high cohesion (i.e., their function relate only to the given purpose).

3.5 Code replication

As dividing and isolating legacy software parts is a hard task, one can be tempted to simply clone and own the original application, but only hiding the secondary purpose features from the end user. However, this leads to lots of replicated and useless code, decreasing the comprehensibility and maintainability of the software artifact, also demanding additional memory space in limited devices.

Figure 2 shows the evolution over time of Foursquare and Swarm mobile applications with respect to their sizes in MB. After the unbundling, the Foursquare user started to need two applications for doing what he/she used to do with only one application, and about 7 MB more of storage, not considering the additional RAM that Swarm consumes. Although these numbers do not clearly prove that there is code replication, it gives us initial insight that unbundling in an efficient way is challenging. This is also true in other unbundling cases and the problem can gain bigger proportions as the number of unbundled application increases.

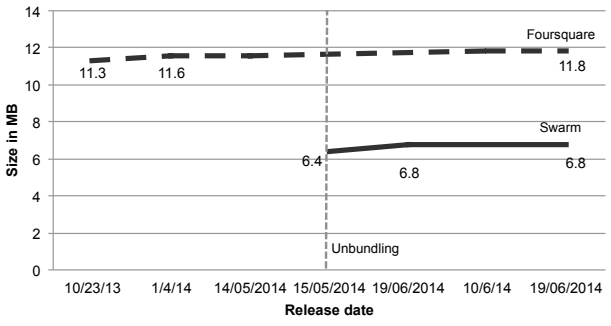

Figure 2. Foursquare and Swarm size evolution.

4. Perspectives

Following, we present the perspectives for analysing and leveraging unbundling, describing key topics that can be the starting point for a research roadmap.

Unbundling can take advantage from existing modularization approaches. Although we still need experiences on whether these techniques actually facilitate or not unbundling, our intuition is that code with cohesive and decoupled coarse-grained modularity units is easier to be unbundled. Therefore, a first perspective is that refactoring approaches for modularization of legacy code should be adapted to cope with the new challenges of unbundling, such as maximizing cohesion, minimizing coupling and mapping modularity units to usage purposes in order to facilitate the concrete division of the software.

Ideally, unbundling should be as much automated as possible. As a perspective, we envision automated techniques to analyse and execute unbundling. For example, detecting patterns of application usage (as in [4, 7]) in order to classify features that are frequently used or not, features that are always used by a profile of user and others by other profiles and so on. These patterns would serve to make explicit the different purposes that the user has when manipulating the software product, better motivating and justifying a division. As for the automation of the division itself, we envision future research on techniques to extract and isolating features in separated applications, going beyond the simple synthesis of feature models after source code [21].

Another perspective is to systematize the unbundling process. Nevertheless unbundling is not a desired or planned event in software lifetime, it can be exploited as a first step to move towards a product line paradigm or a software ecosystem [6, 12]. For

this, the original application can be seen as a source of reusable assets that, if efficiently extracted and isolated, could be the basis to construct several different new applications. This systematization is justified when the company desires, as long-term vision, to carry on building a family of applications for a specific domain, sharing commonalities and managing variabilities.

Unbundling can happen in other kinds of software. Particularly, there is the case of big and complex APIs and frameworks, which provide several features for programmers in a specific domain; they sometimes provide much more than the programmer needs or their use can vary according to different purposes. Therefore we can envision unbundling these artifacts to better suit the needs of different end user purposes, reducing the overload of using a given framework or API.

5. Conclusion

We presented in this paper the novel phenomenon of unbundling software, showing evidences in the domain of mobile applications. In order to better understand it, we explained the process of unbundling, introducing how an original software artifact is transformed into two or more new ones. We then discussed the problems and challenges of unbundling, also relating it to standard planned modularization. Finally, we discussed perspectives on the support of this new phenomenon. Our main conclusion is that unbundling needs special support and the existing modularization techniques can help in this task; because of its unpredicted nature, its high-level abstractions and its needs for concrete isolation of software parts, unbundling raises new challenges that merit to be further investigated, understood and supported. As long-term future work, we consider the points explained in the perspectives section, while in short-term, we want to proceed on analysing unbundling cases deeper, studying the division and distribution of features from the original software products into the new ones.

Acknowledgments

The research leading to these results has received funding from the European Commission under the Seventh (FP7 - 2007-2013) Framework Programme for Research and Technological Development (HEADS project).

We thank the reviewers of Modularity'15 for insightful feedback on this work; they helped us to vastly improve the concepts related to software unbundling.

References

[1] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Extraction and evolution of architectural variability models in plugin-based systems. *Software & Systems Modeling*, pages 1–28, 2013.

[2] S. A. Ajila, A. S. Gakhar, and C.-H. Lung. Aspectualization of code clonesan algorithmic approach. *Information Systems Frontiers*, pages 1–17, 2013.

[3] S. Apel, A. Von Rhein, T. Thüm, and C. Kästner. Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409, 2013.

[4] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer. Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 47–56. ACM, 2011.

[5] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001. ISBN 0201703327.

[6] G. Costa, F. Silva, R. Santos, C. Werner, and T. Oliveira. From applications to a software ecosystem platform: an exploratory study. In *Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, pages 9–16. ACM, 2013.

[7] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in smartphone usage. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 179–194. ACM, 2010.

[8] W. S. Humphrey. Software unbundling: a personal perspective. *IEEE Annals of the History of Computing*, 24(1):59–63, 2002.

[9] S. Ickin, K. Wac, M. Fiedler, L. Janowski, J.-H. Hong, and A. K. Dey. Factors influencing quality of experience of commonly used mobile applications. *Communications Magazine, IEEE*, 50(4):48–56, 2012.

[10] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, pages 311–320. ACM, 2008.

[11] J. Knoop, O. Rüthing, and B. Steffen. *Partial dead code elimination*, volume 29. ACM, 1994.

[12] D. G. Messerschmitt and C. Szyperski. Software ecosystem: understanding an indispensable technology and industry. *MIT Press Books*, 1, 2005.

[13] M. Mortensen. Improving software maintainability through aspectualization. 2009.

[14] M. Mortensen, S. Ghosh, and J. M. Bieman. A test driven approach for aspectualizing legacy software using mock systems. *Information and Software Technology*, 50(7):621–640, 2008.

[15] M. Mortensen, S. Ghosh, and J. M. Bieman. Aspect-oriented refactoring of legacy applications: An evaluation. *Software Engineering, IEEE Transactions on*, 38(1):118–140, 2012.

[16] F. Munoz, B. Baudry, R. Delamare, and Y. Le Traon. Usage and testability of aop: an empirical study of aspectj. *Information and Software Technology*, 55(2):252–266, 2013.

[17] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re) shape evolving software. *Communications of the ACM*, 44(10): 43–50, 2001.

[18] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[19] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[20] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. M. Zimmerman, M. Cornélio, and T. Thüm. Modularizing crosscutting contracts with aspectjml. In *Proceedings of the of the 13th international conference on Modularity*, pages 21–24. ACM, 2014.

[21] S. She, K. Czarnecki, and A. Wasowski. Usage scenarios for feature model synthesis. In *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, pages 15–20. ACM, 2012.

[22] H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer programming*, 56(1):99–116, 2005.

[23] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[24] H. Xi. Dead code elimination through dependent types. In *Practical Aspects of Declarative Languages*, pages 228–242. Springer, 1998.