Software Unbundling: Challenges and Perspectives

João Ferreira Filho Bosco, Mathieu Acher, Olivier Barais

Software Unbundling: Challenges and Perspectives

João Bosco Ferreira Filho
School of Computer Science, University of Birmingham, UK
j.ferreirafilho@cs.bham.ac.uk

Mathieu Acher Olivier Barais
Inria and Irisa, Université Rennes 1, France
mathieu.acher@irisa.fr, olivier.barais@irisa.fr

Abstract

Unbundling is a phenomenon that consists of dividing an existing software artifact into smaller ones. It can happen for different reasons, one of them is the fact that applications tend to grow in functionalities and sometimes this can negatively influence the user experience. 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 point out that it has been performed unsystematically and arbitrarily. In this article, our main goal is to present this novel and relevant concept and its underlying challenges in the light of software engineering, also 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 [35]. Meeting these requirements allows software companies to acquire new users and to stay competitive. 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 [21] – 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: unplanned because it is very hard to foresee that an application will, in some point of the future, need to be split into smaller ones; while regarding the granularity, the unbundled applications can be seen as coarse modules composed by dozens of classes and packages.

The main claim and goal of modularization techniques when applied to mature code is to improve software properties like maintainability and understandability [30]. When 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. Meanwhile, the good properties of modularity may work as enablers to accomplish unbundling.

Despite the importance of unbundling in today's software industry, no studies have been conducted to conceptualize or analyse these challenges. This article 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, showing examples of this current phenomenon and explaining how it relates to existing software engineering approaches for software modularization (Section 3); and to draw perspectives on how to facilitate and better exploit unbundling (Section 4).

This article builds upon our original paper [15] and the feedbacks we gathered at MODULARITY'15 [17]. We extend the work [15] as follows: (1) we further discuss challenges related to the user acceptance of unbundled software, discussing the risks for a company's user base; (2) we describe the reasoning over the consequences in the application's infrastructure and surrounding services; (3) we include a concrete case of a mobile client when explaining the granularity challenges; and (4) we extend the literature review.

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 [20].

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 primary 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 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 major purpose per application, so the user knows 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 applications Δ_1, Δ_2 because they serve two distinct end user purpose. This isolation implies changing the structure of the part to make it decoupled or even adjusting 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.

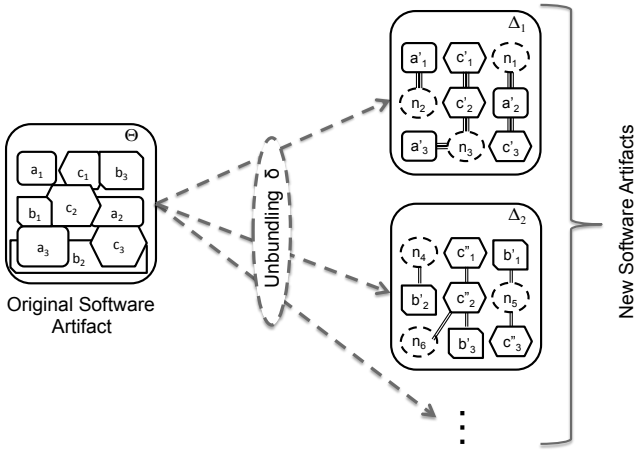

Figure 1. Unbundling.

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 help reducing the original application, removing nearly-unused/dead code without the need to make it usable in another application. The difference between actual dead code elimination [24, 45] 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 summary, if we think of *unbundling as a process*, we can enumerate the following key activities that it encompasses:

- Identifying distinct end user purposes when using a software candidate to unbundling;
- 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 seven 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 [42], or aspectualizing [4, 29, 36] software. Since its origins, modularization seeks to improve flexibility and comprehensibility of a given software [34]; these goals have been enlarged to also consider maintainability and testability [28, 31].

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 to the ends of dividing the application, while before, modularizing and dividing the software artifact in modules were means to reach such good properties.

Therefore, unbundling is triggered by business goals, it creates new opportunities with separable markets [20]; it is a specialization of software businesses. From a 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 a software engineering perspective, modularization is an 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 about 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 become a trigger for unbundling, probably if the way the application is structured starts to interfere with 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) [9], 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 interfere with this relationship; this interference is very likely to happen if a bad release of the unbundled software is launched.

3.3 Coarse-Grained Modules

The criteria used when decomposing systems into modules have been studied for decades [34]; from sub-routines to features [22], the unity of modularity is a primary concern in software design and implementation. These units are conceived 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-grained modularization raises the challenge of having to categorize the existing fine units and associate them to greater goals, which may or may not be explicitly modeled in the software. These purposes 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* (i.e., the act of confirming your current location within a place, possibly sharing it with friends) 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.

In Figure 2, we illustrate, using the case of `wordpress.com` mobile client, that sometimes these different end-user purposes are evidently identifiable and are even reflected in the application's GUI. On the left-hand side of Figure 2, we see activated the tab of the mobile app that allows the user to manage her website, while on the right-hand side, there is the tab for reading news from other users' websites. From the end-user point of view, these are dissociated functionalities that could therefore be in two different applications. Wordpress has chosen to keep the news feed in the same application, in the opposite way of LinkedIn, which has launched an application (Pulse) that isolates this functionality. A look into the Wordpress mobile application code `https://github.com/wordpress-mobile/WordPress-Android` reveals that there are 73 classes associated with the Reader tab, scattered through 8 different packages, evidencing the coarser nature of the end-user purpose.

3.4 Dividing and Isolating

Dividing software has always proved challenging; many different software engineering approaches have been proposed to decompose [3, 33] code or slicing programs [43].

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 [5]. Therefore, parts of the original software product may end up present in different purposes, or the functioning of a feature can 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, aspectualization, feature extraction techniques [1, 40] to work seamlessly with an additional level of abstraction: the purpose. These techniques work by analysing the software and evaluating its

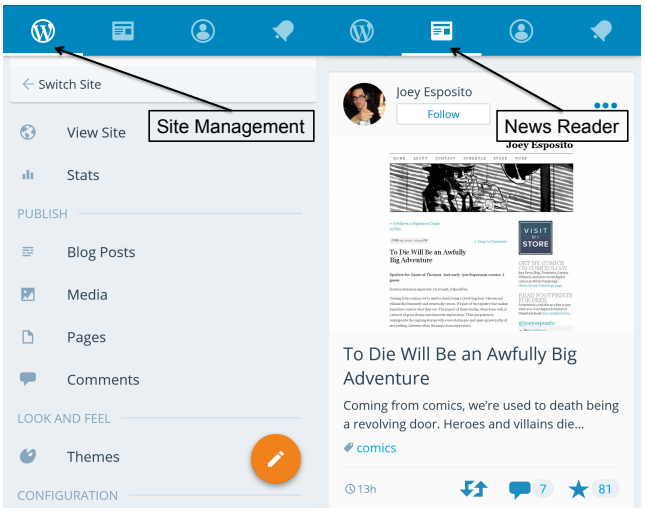

Figure 2. Wordpress mobile client.

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—we cannot assume that end user purposes correspond directly to aspects/concerns of software development—driving the division itself.

Considering the steps of the unbundling process, we can state that the essential challenges of it are comprehended in dividing and isolating parts of software: (1) identifying, (2) extracting and (3) 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 become more difficult according to how strong is the relationship between modules of the original software artifact that belong to different purposes (i.e., the 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 functions 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 3 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 get more important as the number of unbundled applications increases.

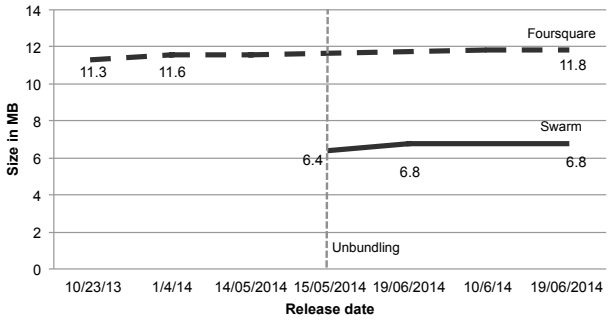

Figure 3. Foursquare and Swarm size evolution.

3.6 Services and Infrastructure

Software is becoming increasingly pervasive; it spreads over different devices and can use several remote services running in the cloud. It is very likely that the end-user purpose to be unbundled is also scattered through these different locations. Therefore, unbundling an application may also require refactoring of currently consumed/provided services and infrastructures. One example is an application's related API. API's must reflect the functionalities provided by an application and therefore, if these functionalities change, the API should be adapted to reflect it. Whether engineers should also split the API into the same number of sub-API's than of sub-apps will depend on several aspects: how interesting (economically) is dividing also the API, if it was designed generically enough to be intuitive to use even after the apps division, if it is also becoming a monolith that needs unbundling, how difficult is it to migrate to a new version and if the old version must be deprecated.

Although unbundling is relatively recent, we can already notice these challenges in current cases. For instance, Dropbox is deprecating its old API to give room to a simpler new one:

> *"As part of this effort to simplify our platform, we've decided to deprecate the Sync and Datastore APIs over the next 12 months."* Dropbox team.

The Foursquare team decided to move forward smoother, deprecating only features from the old API that are not used anymore within the new apps. On another note, they chose to keep the same User Model for both apps, so services like authentication are shared by both applications, even though that a user of Swarm may not be a Foursquare user too, which can have implications to the API's users:

> *"We expect the majority of Swarm users to also have Foursquare installed, but implementing a fallback to the web view-based implementation of auth has always been recommended as well, especially to account now for Swarm users that may not have Foursquare installed."* Foursquare team.

3.7 User Acceptance

Evolving software is challenging for many engineering reasons [26], and it can also be risky if we consider the end-user adoption [37]. Many studies [8, 11, 18, 41] have investigated how software is perceived by users (or even how user's personality influences this perception [46]); essentially, these studies try to model and relate the user's intention to use an application, which goes beyond the applications' easiness of use, it also encompasses expectations about applications' privacy, social influence, cost, performance, etc.

Unbundling must take into account these issues, specially the ones related to user's resistance to changes. An unbundling candidate application has a user base that is already acquainted to the way that the app works, which functionalities it has, and how to use them or where to find them on the GUI. On the one hand, moving these features from one application to another can cause unsatisfied users, who can give negative feedback in the application's reviews/comments page in their different mobile application stores. Critics such as *"I hear the argument why you split the features between this and Swarm, but you're bleeding a fan base"* or *"It used to be better when check-ins were all in a single app"* are numerous at the Foursquare reviews; it happens similarly with the Messenger application that was unbundled from Facebook: *"I never liked the fact that we are forced to download this 2nd app just to chat"*.

On the other hand, an unbundled app can also please more flexible users if it clearly brings new features or eases user experience. In recent interview to a global media company, Stan Chudnovsky—Messenger's head of product—claimed the importance of having unbundled Messenger from Facebook: *"If Messenger were still buried within the main app, video calling would be buried within that, and it just wouldn't ever find the light of day"*. Indeed, many users have written positive feedback in Messenger's reviews page, complimenting the new features brought by the unbundled app, such as *"I just love it it's now updated and added video calls , hd video calls, voice call ,voice chat and much more things available in it"*.

Hence, companies need to comprehend and assess the risks involved in splitting their applications from the point of view of the user. Scientifically identifying and studying the correlations between unbundling and the user adoption can be very challenging, as it involves many dependent variables that are hard to be isolated in an empirical study.

4. Perspectives

In the 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 [6, 14]) 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 [19, 23, 32], going beyond the simple synthesis of feature models [7, 12, 39, 44]. Similarly there is a large amount of approaches to locate concerns, to mine aspects or to automate the refactoring of applications in a more modular way [2, 13, 16, 25, 38].

Another perspective is to systematize the unbundling process. Even though unbundling is not a desired or planned event in software lifetime, it can be exploited as a first step to move to-

wards a product line paradigm or a software ecosystem [10, 27]. 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. Besides numerous artefacts of a project can be considered as part of the unbundling process. Source code, but not only: documentation, bug reports, or mailing lists can also be used for identifying an unbundling criterion. The unbundling can also be applied over multiple artefacts (e.g., source code together with the documentation).

5. Conclusion

We presented in this article the novel phenomenon of software unbundling, 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 unpredictability, its high-level abstractions and its need 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: (1) using separation of concerns techniques for unbundling and analysing quantitatively and qualitatively how they perform, (2) explore automated techniques for unbundling, (3) develop systematic methods and processes, (4) leverage unbundling in different kinds of software-intensive systems. 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

We would like to thank the participants of MODULARITY'15 for the feedbacks, suggestions, and discussions. 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).

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] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can we refactor conditional compilation into aspects? In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 243–254, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-442-3. . URL `http://doi.acm.org/10.1145/1509239.1509274`.

[3] O. Agesen and D. Ungar. Sifting out the gold: Delivering compact applications from an exploratory object-oriented programming environment. In *ACM SIGPLAN Notices*, volume 29, pages 355–370. ACM, 1994.

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

[5] 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.

[6] 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.

[7] G. Bcan, M. Acher, B. Baudry, and S. Nasr. Breathing ontological knowledge into feature model synthesis: an empirical study. *Empirical Software Engineering*, 2015. ISSN 1382-3256. URL `http://dx.doi.org/10.1007/s10664-014-9357-1`.

[8] B. Chen, S. Sivo, R. Seilhamer, A. Sugar, and J. Mao. User acceptance of mobile technology: A campus-wide implementation of blackboard's mobileł learn application. *Journal of Educational Computing Research*, 49(3):327–343, 2013.

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

[10] 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.

[11] F. D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.

[12] J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Cleland-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions. In *ESEC/FSE'13*, 2013.

[13] M. Eaddy, A. V. Aho, G. Antoniol, and Y.-G. Guéhéneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3176-2. . URL `http://dx.doi.org/10.1109/ICPC.2008.39`.

[14] 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.

[15] J. B. Ferreira Filho, M. Acher, and O. Barais. Challenges on software unbundling: Growing and letting go. In *14th International Conference on Modularity'15*, Fort Collins, CO, United States, mar 2015. URL `https://hal.inria.fr/hal-01116694`.

[16] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. . URL `http://doi.acm.org/10.1145/1368088.1368124`.

[17] R. B. France, S. Ghosh, and G. T. Leavens, editors. *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, 2015. ACM.

[18] M. D. Gallego, P. Luna, and S. Bueno. User acceptance model of open source software. *Computers in Human Behavior*, 24(5):2199–2216, 2008.

[19] N. Hariri, C. Castro-Herrera, M. Mirakhorli, J. Cleland-Huang, and B. Mobasher. Supporting domain analysis through mining and recommending features from online product listings. *IEEE Transactions on Software Engineering*, 99:1, 2013. ISSN 0098-5589. .

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

[21] 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.

[22] 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.

[23] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semiautomatic detection of product-line features. *IEEE Transactions on Software Engineering*, 2013.

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

[25] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2243-2. URL `http://dl.acm.org/citation.cfm?id=1038267.1039053`.

[26] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering*, 2015. .

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

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

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

[36] 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.

[37] N. Sanakulov and H. Karjaluoto. Consumer adoption of mobile technologies: a literature review. *International Journal of Mobile Communications*, 13(3):244–275, 2015.

[38] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: Feature location and textual tracing tool. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 255–258, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. . URL `http://doi.acm.org/10.1145/1810295.1810345`.

[39] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. Efficient synthesis of feature models. *Information & Software Technology*, 56(9):1122–1143, 2014. . URL `http://dx.doi.org/10.1016/j.infsof.2014.01.012`.

[40] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(6):625–666, 2002.

[41] V. Venkatesh, M. G. Morris, G. B. Davis, and F. D. Davis. User acceptance of information technology: Toward a unified view. *MIS quarterly*, pages 425–478, 2003.

[42] 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.

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

[44] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC*, 2009.

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

[46] T. Zhou and Y. Lu. The effects of personality traits on user acceptance of mobile commerce. *Intl. Journal of Human–Computer Interaction*, 27(6):545–561, 2011.