Visualizing Code Variabilities for Supporting Reuse Decisions

Anna Zamansky and Iris Reinhartz-Berger

Department of Information Systems, University of Haifa, Israel
annazam@is.haifa.ac.il, iris@is.haifa.ac.il

Abstract. Software reuse is the practice of using artifacts from existing systems to build new ones. It has been shown effective for improving quality and maintainability and for reducing cost and development time. Human factors have been identified as significant barriers to a wider adoption of reuse practices in industry. In this paper we consider a tool-supported approach for systematic reuse of object-oriented programs (written in Java) based on polymorphism-inspired mechanisms. The suggested tool gets as input implementations of multiple products, and produces a visual representation of the similarities and variabilities between their classes in terms of exhibits behaviors, as well as presents possible reuse options. We discuss the suitability of this approach for educational and training settings, and specifically for supporting reuse decisions of novice developers.

Keywords: Software reuse**,** Education**,** Decision Support**,** Software Product Line Engineering, Variability Analysis, Variability Mechanisms, Polymorphism

1 Introduction

Development has become increasingly complex while reducing time-to-market remains a critical issue. Software reuse has been shown to be effective for reducing cost and development time [14]. However, there are significant barriers in the adoption of reuse practices in industry. As pointed out in [2], "initial research on software reuse has focused on the technological issues (e.g., programming language support, creating and retrieving reusable artifacts, repositories, etc.), and only later non-technical factors (e.g., organization, processes, business drivers) were found to be important for the success of a reuse strategy".

Recently more attention has been drawn to the human factors of reuse practices, focusing mainly on decision making processes of developers [12], [20]. In particular, empirical studies suggest reuse training is an important factor for improving reuse practices [5], [4]. Nevertheless, work on how to educate for reuse is scarce. Frakes and Kang [6] stress the need for addressing reuse education: "Industry studies have shown that education is a primary factor in better reuse, yet there had been little systematic study of how best to do reuse education. Certainly, both academia and industry could improve educational practices".

In this paper we propose a tool-supported approach for educating and training novices and supporting reuse decisions. In [19] we presented a tool for comparing pairs of software artifacts (object-oriented code) and representing their similarities and variabilities. The tool, called VarMeR – a Variability Mechanisms Recommender, is based on an ontological framework that compares software behaviors rather than concrete implementations [16], [17], [18]. This way software systems that have similar intensions (i.e., exhibit similar behaviors) can be considered for reuse, even if their implementations are different (e.g., contains different components).

We particularly explore the suitability of VarMeR to assist novice developers in reuse decisions. To this end, VarMeR was extended to compare an arbitrary number of software artifacts (rather than pairs of artifacts). In other words, the input of VarMeR is object-oriented code artifacts (in Java) that belong to multi software systems and the output is a graph that captures the similarities and variabilities of the classes of those systems in terms of their exhibited behavior. The tool further recommends how to increase reuse by utilizing suitable polymorphism-inspired mechanisms.

The rest of this paper is structured as follows. Section 2 provides an overview of our proposed approach, while Section 3 presents the capabilities of the extended version of the VarMeR tool (for supporting reuse when multi software products are available). Section 4 describes the benefits of the tool and its possible use scenarios in educational and training contexts, as well as some preliminary usability feedback. Finally, Section 5 summarizes and refers to future plans.

2 The VarMeR Approach

VarMeR analyzes the commonality and variability of products behaviors and presents the analysis outcomes in the form of polymorphism-inspired mechanisms among classes that behave similarly (even if their realizations are different). Specifically, the approach is composed of three steps, shown in Figure 1: Extract Behaviors, Compare Behaviors, and Analyze Variability.

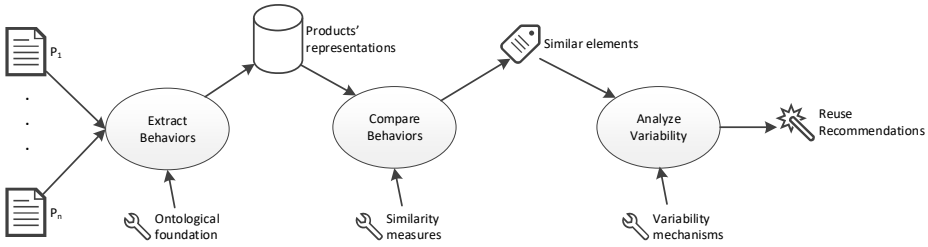

Figure 1. A high level overview of the approach

2.1 Extracting Behaviors

Based on ontological considerations [16], a software behavior can be represented as a triplet of initial state – the status of the software before the behavior occurs, external

event – that triggers the behavior, and final state– the status of the software after the behavior occurs. Those behavioral components are extracted from the public operations of the different classes[1]. Each public class operation specifies some behavior of the software product that is widely relevant within the product. We assume that the operation name captures the essence of the behavior and thus can describe its trigger (the external event), e.g., Borrow and Return of a Book Copy class in a library management system (see Figure 2).

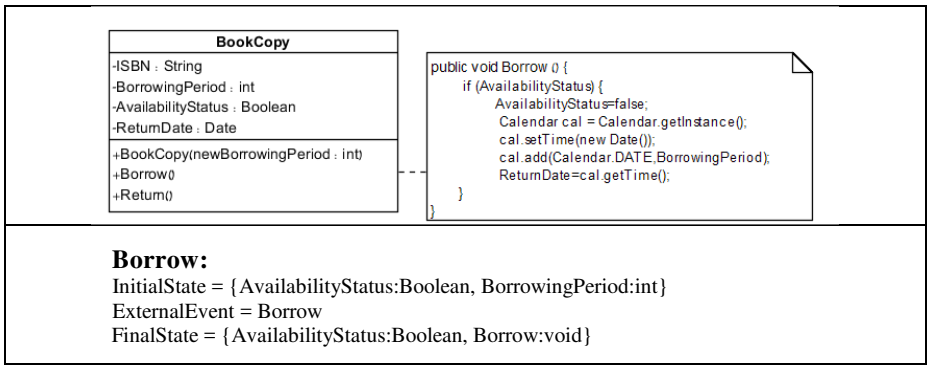

Figure 2. An example of behavior extraction

For extracting initial and final states, we distinguish between two levels of operation descriptor: shallow – which refers to the signature of the operation, and deep – which takes into consideration the behavior in terms of attributes used and modified throughout the operation (including those that are used and modified indirectly by operations called from the analyzed operation). We consider only attributes and ignore local variables, as the later can be defined for implementation and realization purposes and may hinder the operation's behavior essence. The *initial state* of the behavior is composed of all the parameters passed to the operation (part of shallow) and all the class attributes used (read) by the operation (part of deep). The *final state* consists of the operation name and its returned type (part of shallow) and all the class attributes modified (set) by the operation (part of deep). Figure 2 exemplifies the behavior extraction outcome for the operation Borrow of the Book Copy class. Note that each attribute is presents via its name and type which provide the basis for comparison.

2.2 Compare Behaviors

Different methods have been proposed for measuring the similarity of applications or software systems. McMillan et al. [11], for example, propose an approach called CLAN that measures similarity of Java applications using the notion of *semantic layers* that correspond to packages and class hierarchies. As opposed to that approach and

[1] The assumption is that private and protected operations are introduced for implementation purposes and thus hinder the exhibited behavior of the analyzed software product.

many other methods, which take into account structural implementation and realization considerations, VarMeR measures the behavioral similarity of software systems.

To this end, a similarity mapping between the behavior constituents (namely, initial state, external event, and final state) is applied. This mapping can be based on existing general-purpose or domain-specific similarity metrics or some combination of such metrics. The metrics can be based on semantic nets or statistical techniques to measure the distances among words and terms [13]. Alternatively, they can use type or schematic similarities, potentially ignoring the semantic roles or essence of the compared elements [7]. The similarity mapping associates to each operation's constituent (parameter, attribute used, or attribute modified) all of its similar counterparts in the other operation (i.e., elements whose similarity with the given constituent exceeds some predefined threshold).

Returning to our example of Book Copy, assume a class named Car which has an operation named Rent that changes the *In Agency* status of a car from true to false. It further calculates the *Back Date* according to the *Rental Period*. Figure 3 exemplified two potential similarity mappings: the first one (a) is based on a schematic type-based similarity according to which two attributes are similar if and only if their types are similar. The second mapping (b) is based on a semantic measure named Latent Semantic Analysist (LSA) [10].

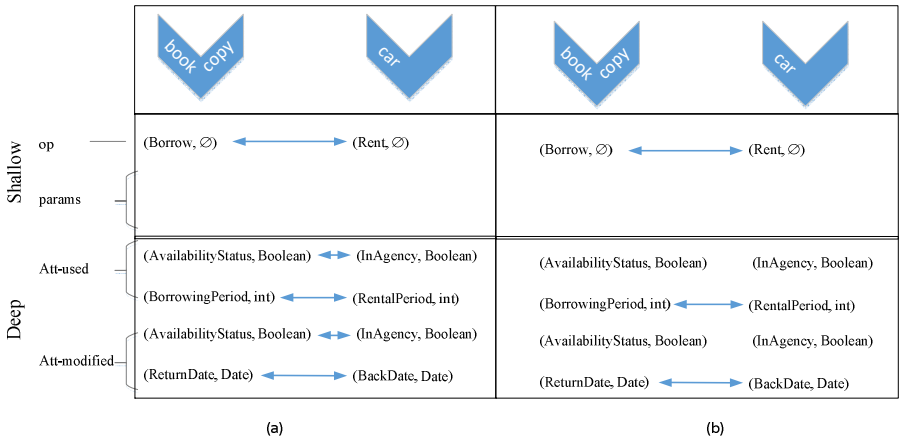

Figure 3. Examples of similarity mappings based on: (a) type-based similarity and (b) semantic similarity (LSA)

2.3 Analyze Variability

Based on the similarity mapping, we can distinguish between the following cases among (public) operations:

1. USE – the similarity mapping is bijection (each constituent of operation 1 has exactly one counterpart in operation 2 and vice versa).
2. REF (abbreviation for refinement) – at least one constituent in operation 1 has more than one counterpart in operation 2.

3. EXT (abbreviation for extension) – at least one constituent in operation 1 has no counterpart in operation 2.

Note that REF and EXT are not mutually exclusive; we refer to a combination of both as REF-EXT (abbreviation for refined extension).

Aggregating the above notions from the level of operations to the level of classes, we take inspiration from the polymorphism mechanisms. *Polymorphism* is the provision of a single interface to entities of different types. Therefore, the cases of polymorphism are characterized by similar signatures of operations (namely, the USE category in the shallow level of the operations). We further focus on three types of polymorphism which are widely used in industry:

1. Subtyping (inclusion) polymorphism which includes refinement or extension of behaviors (e.g., function pointers, inheritance).
2. Parametric polymorphism which includes name or type analogy (e.g., C++ templates).
3. Overloading which includes behavior change while maintaining the same signature.

Table 1 presents recommendations for those polymorphism-inspired mechanisms based on the reuse mapping characteristics.

Table 1. Characteristics of Polymorphism-Inspired Mechanisms

Shallow	**Deep**	**Description**	**Polymorphism-Inspired mechanism**
USE	USE	Both signatures and behaviors are similar	Parametric
USE	REF	Signatures are similar and behavior is refined	Subtyping
USE	EXT	Signatures are similar and behavior is extended	
USE	REF-EXT	Signatures are similar and behavior is both refined and extended	
USE	Not mapped	Signatures are similar and behavior is different	Overloading

3 The VarMeR Tool

In order to make our approach accessible to developers (potentially novice ones and students), we developed a tool named VarMeR – Variability Mechanisms Recommender. While the first version of the tool, described in [19], concentrated on analyzing the commonality and variability between a pair of software products, the current version extends the scope to a multi-product setting. This way the tool aims at supporting reuse decisions and particularly the selection of the most suitable products (or product parts) to reuse.

The inputs of the tool, namely the software products, are provided as (paths to) jar files. Those files are reverse engineered into class diagrams (in XMI format) and Program Dependence Graphs (PDG)[2] [8] (in JSON format). The shallow and deep levels of the behaviors are extracted from those representations and the tool proceeds in the

[2] PDG explicitly represents the data and control dependencies of a program.

three stages described in the previous section. The outcome is presented visually in three levels of analysis, using graph-based representations:

- **Product level** with nodes representing the software products and edges representing their degrees of similarity.
- **Class level** with nodes representing the classes of the analyzed software products and edges representing recommendation on polymorphism-inspired mechanisms (parametric, subtyping, and overloading).
- **Operation level** with nodes representing operations (of a certain sub-set of classes of software products) and edges representing the mechanism characteristics listed in Table 1 (USE, REF, EXT, REF-EXT).

In the product and class levels, the size of the nodes is proportional to the size of objects they represent; the larger the node is, the more operations the class have or the more classes the product has. The width of an edge, as well as its length, represents the degree of evidence (e.g., the number of operations related with a certain type of polymorphism); the thicker/longer the link is, the more evidence exist.

An example of VarMeR output at the class level is depicted in Figure 4. The comparison is done between three different software products, the classes of which are represented using different colors. To support scalability, VarMer provides the user with several possibilities for fine-tuning and information hiding, which are further discussed in the next section.

4 Potential Use of VarMeR in Education and Training Contexts

Although lack of training has been identified as a major barrier to a wider adoption of reuse practices in industry, no systematic way to address this problem has been proposed [6]. We suggest the VarMeR tool presented above as a starting point for developing methods for education and decision support of novice developers and software engineering students due to its intuitive abstractions, its visualization, and its support for scalability. These features are discussed below, as well as some usability scenarios and feedback we have collected regarding VarMeR.

4.1 VarMeR in Education and Training Contexts

Intuitive abstractions. Krueger [9] highlights the importance of choosing the right abstraction in the context of reuse: “Why is software reuse difficult? Useful abstractions for large, complex, reusable software artifacts will typically be complex. In order to use these artifacts, software developers must either be familiar with the abstractions a priori or must take time to study and understand the abstractions. The latter case can defeat some or all of the gains in reusing an artifact.”

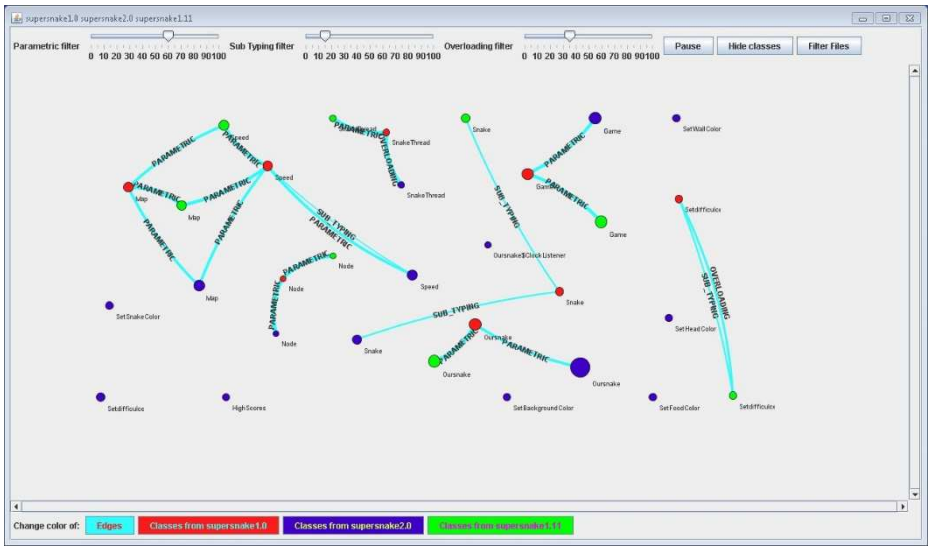

Figure 4. Class Level Analysis

VarMeR makes use of graph-based abstractions which are intuitive and easy to understand: nodes to represent (different types of) elements, and edges to represent their similarity relations. It also supports switching between different abstraction levels by supporting multi-level analysis (product-, class- and operation-levels). Another kind of abstraction made by the VarMeR approach is that comparison between software elements is made in terms of intensions (i.e., exhibited behavior) rather than in terms of realizations and implementations. Starting with understanding behavior may be much easier than starting by understanding old code, especially for novices. As highlighted by Agresti [1]: "It takes effort to understand old code. The developer is trying to establish whether the old code will meet some or all of the new requirements so it can be part of a new system. When that old code is written such that it makes it especially difficult to understand, a developer can reasonably conclude that her effort is better spent developing new code from scratch."

Visualization. While a large body of research on software visualization exists [3], to the best of our knowledge visualization for reuse has not yet been addressed. The type of visualization offered by VarMeR can be classified as what is called in [15] 'changing the perspective', or "bringing large software engineering problems within the scope of a single view,.., an attempt at complexity control, helping to keep a large problem 'in a single head' by visualizing the overall structure and providing some assistance for navigating or traversing that structure." In VarMeR's visualization we aim to increase cognitive effectiveness by the use of simple and intuitive graph-based elements (nodes and edges), employing also other visual variables such as color (to encode different products), size (to encode the number of operations/classes in each class/product) and edge thickness (to encode extent of similarity).

Scalability: **fine-tuning and information hiding**. Reuse decisions might be easy when considering two simple operations such as Borrow and Rent from Figure 3, but

dealing with large scale projects with hundreds of classes and thousands of behaviors introduces an additional dimension of complexity when making reuse decisions. To reduce the user cognitive load, VarMeR supports several ways of information hiding and fine-tuning at the class level analysis (see Figure 4):

- Modifying thresholds for presenting recommendations for each type of polymorphism (namely, the minimal percentages of "similar" operations to present parametric, subtyping, and overloading edges can be set; the lower these thresholders are, the more abstraction is needed to apply reuse for those classes).
- Hiding classes which have no similarity to other classes (*Hide Classes* button).
- Filtering the graph-based visualization according to different software packages of the product (*Filter Files* button).

4.2 Usability Scenarios and Feedback

The above features provide a starting point for developing a method for supporting novice developers and software engineering students in reuse decisions. Consider, e.g., a scenario in which a novice developer, or a student in a programming course, needs to develop a software system. In an industrial setting, the company may have already developed various similar systems and maintain a repository of software artifacts. In other settings, some open-source applications may be available. Searching in such repositories, e.g., using keywords or queries, our developer may discover several similar systems, not all of which have informative descriptions, and each of which may have many different versions. Let us assume that our developer finally decides to select five systems, which according to their descriptions are quite similar to the system he/she needs to develop. For making a decision which system (or system parts) to reuse, it is useful to comprehend how the five systems differ.

This is exactly the point where VarMeR enters the scene, providing assistance to support such decisions. The developer can run the tool on the five selected systems[3] and browse the similarity analysis and recommendations at different levels of abstraction. The product-level analysis will show him/her which of the systems are most similar. Zooming into class level, he/she can identify clusters of classes which can potentially be reused for implementing different behaviors. Zooming again into the operation level, we obtain information on the reuse relations among operations. At this stage the developer can zoom into the implementation itself, by looking at the relevant segment of code in an Eclipse-like environment and adapt the code to the task at hand.

We are currently in the process of evaluating VarMeR's usability. In a pilot setting, we gave the tool to two pairs of students studying in the Information Systems department at the University of Haifa. Each pair received two portions of similar Java projects from SourceForge. They were requested to inspect VarMeR's outcomes, and grade the appropriateness of its recommendations. Our analysis of the open-text feedback, provided by the students and classified by us according to VarMeR's features, shows that visualization was positively mentioned – and particularly the use of colors and size.

[3] Note that theoretically the developer could run VarMeR on all the game applications. However, the complexity of multi-object comparison dramatically increases as the number of objects increases. Hence, some a-priori filtering is recommended.

The intuitive abstractions, and specifically the ability to zoom-in and zoom-out between the product, class, and operation levels, was also considered very useful. The scalability support was also mentioned as important, but the comprehensibility of three independent bars (for parametric, subtyping, and overloading mechanisms) was questioned.

5 Summary and Future Work

The human factor is one of the most significant barriers in wider adoption of reuse practices in industry. Yet aspects of reuse training and decision support have so far been overlooked in the software engineering literature. We addressed this problem by presenting a tool-supported approach aiming to support developers in making reuse decisions, and discussed its applications in training settings. The VarMeR tool, supporting our approach, has several features which make it attractive in the context of reuse education. It applies intuitive abstractions of software artifacts, and allows for easy switching between abstraction levels. Furthermore, it uses visualization which employs intuitive visual constructs and variables, and aims to reduce cognitive load of developers when dealing with large scale software projects by allowing for fine-tuning and information hiding.

In the future, we intend to explore several paths for further development of VarMeR for educational purposes. First, we intend to develop a querying language on top of VarMeR, in order to retrieve the most relevant artifacts to a given development task. Second, we intend to systematically support reuse activities. After retrieving the relevant (portions of) software artifacts, we need to explore how to guide the developer in applying the reuse recommendations. Finally, we are in the process of adapting VarMeR in an academic software engineering course. Empirical studies with the students are planned to evaluate the benefits and limitations of the tool and further improve the tool. Our long term vision is for VarMeR to be fully integrated in standard development environments to promote reuse thinking as an integral part of development.

Acknowledgment. The authors thank Jonathan Liberman for his help in the implementation of the VarMeR tool. We also thank Alex Kogan and Asaf Mor for their assistance in the initial steps of the development. The first author was supported by the Israel Science Foundation under grant agreement 817/15.

References

[1] Agresti, W.W. (2011). Software reuse: developers' experiences and perceptions. Journal of Software Engineering and Applications, 4 (1), pp. 48-58.

[2] Anisa, S. (2015). Do Developers Make Unbiased Decisions? The Effect of Mindfulness and Not-Invented-Here Bias on the Adoption of Software Components. ECIS'2015.

[3] Diehl, S. (2007). Software visualization: visualizing the structure, behaviour, and evolution of software. Springer Science & Business Media, 2007.

[4] Favaro, J. (1991). What price reusability?: a case study. ACM SIGAda Ada Letters, 11 (3), ACM, pp. 115-124.

[5] Frakes, W.B. and Fox C.J. (1995). Sixteen questions about software reuse. Communications of the ACM, 38 (6): 75-ff.

[6] Frakes, W.B. and Kang, K. (2005). Software reuse research: Status and future. IEEE transactions on Software Engineering, 31 (7), pp. 529-536.

[7] Kashyap, V. and Sheth, A., (1996). Semantic and schematic similarities between database objects: a context-based approach. VLDB Journal, 5(4), pp. 276-304.

[8] Krinke, J. (2001). Identifying Similar Code with Program Dependence Graphs. 8th Working Conference on Reverse Engineering, pp. 301-309.

[9] Krueger, C. W. (1992). Software reuse. ACM Computing Surveys (CSUR) 24 (2), 131-183.

[10] Landauer, T. K., Foltz, P. W. and Laham, D. (1998). Introduction to Latent Semantic Analysis. Discourse Processes 25, pp. 259-284.

[11] McMillan, C., Grechanik, M. and Poshyvanyk, D. (2012). Detecting similar software applications. 34th International Conference on Software Engineering (ICSE'2012), pp. 364-374.

[12] Mellarkod, V., Appan, R., Jones, D. R., & Sherif, K. (2007). A multi-level analysis of factors affecting software developers' intention to reuse software assets: An empirical investigation. Information & Management, 44(7), 613-625.

[13] Mihalcea, R., Corley, C., and Strapparava, C. (2006). Corpus-based and knowledge-based measures of text semantic similarity. American Association for Artificial Intelligence (AAAI'06), pp. 775-780.

[14] Mohagheghi, P., and Conradi, R. (2007). Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering 12(5), 471-516.

[15] Petre, M., A. F. Blackwell, and T. R. G. Green. (1998) Cognitive questions in software visualization. Software visualization: Programming as a multimedia experience, 453-480.

[16] Reinhartz-Berger, I., Zamansky, A., & Wand, Y. (2016). An Ontological Approach for Identifying Software Variants: Specialization and Template Instantiation. 35th International Conference on Conceptual Modeling (ER'2016), pp. 98-112.

[17] Reinhartz-Berger, I., Zamansky, A., and Kemelman, M. (2015). Analyzing Variability of Cloned Artifacts: Formal Framework and Its Application to Requirements. Enterprise, Business-Process and Information Systems Modeling, EMMSAD'2015, pp. 311-325.

[18] Reinhartz-Berger, I., Zamansky, A., and Wand, Y. (2015). Taming Software Variability: Ontological Foundations of Variability Mechanisms. 34th International Conference on Conceptual Modeling (ER'2015), LNCS 9381, pp. 399-406.

[19] Reinhartz-Berger, I. Zamansky, A. (2017). VarMeR - A Variability Mechanisms Recommender for Software Artifacts. CAiSE-Forum-DC2017, 57-64.

[20] Sojer, M. and Henkel, J. (2010). Code reuse in open source software development: Quantitative evidence, drivers, and impediments. Journal of the Association for Information Systems, 11 (12), 868-901.