Features and How to Find Them: A Survey of Manual Feature Location

Jacob Krüger[1,2], Thorsten Berger[3], Thomas Leich[2,4]

[1] Otto-von-Guericke University, Magdeburg, Germany

[2] Harz University of Applied Sciences, Wernigerode, Germany

[3] Chalmers | University of Gothenburg, Gothenburg, Sweden

[4] METOP GmbH, Magdeburg, Germany

Abstract

The notion of features is commonly used to maintain, evolve, reuse, or re-engineer a software system. To this end, developers need to understand the features and their – potentially scattered – locations within the codebase of a system. Unfortunately, features are rarely documented, developers' knowledge about the features fades quickly, and developers leave projects. In such cases, feature locations need to be recovered from the codebase – a costly, but still one of the most common tasks performed by developers. While automated feature-location techniques have been proposed, they typically require significant adjustments to the particular system and often fall short in their accuracy. To improve this situation, it is necessary to understand how developers perform feature location manually – a surprisingly neglected research area. In this chapter, we provide an overview on existing studies of manual feature location, focusing on the topics addressed in the studies and open issues. We find that the seven studies we identified analyze four topics of manual feature location in detail: Search tools, performance, influencing factors, and distinct phases. We observe that a focus of these studies is on understanding the process of manual feature location regarding the developers' activities and actions. We also find that, still, little is known about the actual efforts of performing feature location, how different factors influence these efforts, how automated techniques are scoped to the identified phases, and which additional information sources help developers to locate features.

Keywords

Feature location; systematic literature review; reverse variability engineering; feature identification; feature mapping

1. Introduction

Developers do not only implement software systems, but also maintain, extend, evolve, and re-engineer them (Standish 1984; Tiarks 2011). To this end, it is essential that developers understand the relevant source code (Siegmund 2016) – especially where the system's functionalities, or *features,* are located and what their relationships are among each other (von Mayrhauser, Vans, and Howe 1997). As such, feature location can arguably be seen as one of the most frequent tasks performed by developers (Biggerstaff, Mitbander, and Webster 1986; Poshyvanyk et al. 2007; Wang et al. 2013). Unfortunately, feature location can be costly and challenging – for instance, when features and their locations are not documented, when developers are newly assigned to a project, when previous developers retire, or when developers' knowledge about features fades over time (Kästner, Dreiling, and Ostermann 2014; Ji et al. 2015; Krüger et al. 2016; Krüger, Wiemann, et al. 2018). In these cases, developers have to re-obtain knowledge about features and recover their locations in the system's source code.

Several fully or semi-automated techniques have been proposed for identifying, locating, and documenting features (Rubin and Chechik 2013; Dit et al. 2013; Laguna and Crespo 2013; Assunção and Vergilio 2014). However, applying them in practice is challenging. First, automation of the feature-location process is difficult, since features are rather domain-specific entities and orthogonal to typical structures found in programs, such as components, classes or methods. This makes automated feature location –namely, deciding which source-code artifacts belong to a feature – a difficult problem that can hardly be solved by algorithms (Biggerstaff, Mitbander, and Webster 1986; Kästner, Dreiling, and Ostermann 2014). Second, the proposed techniques lack in accuracy and require substantial effort from developers to adapt them to a specific system (Wilde et al. 2003; Ji et al. 2015; Krüger, Gu, et al. 2018). As a result, these techniques may be helpful, but developers still need to largely resort to manual feature location.

Unfortunately, little research exists on understanding how developers perform feature location manually. Thus, it is not clear if the proposed automated techniques actually do (or could) help developers performing feature location. This research gap calls for an analysis on manual feature location.

In this chapter, we survey the literature about studies of developers performing manual feature location. For this purpose, we conduct a systematic literature review (Kitchenham and Charters 2007) on the literature databases DBLP and SCOPUS, and extend our results with snowballing (Wohlin 2014). By relating synonymous and related terms, we consolidate the identified research into four topics that are primarily investigated. We also clarify the term feature location, summarize the findings of the identified literature, and identify open gaps that hamper our understanding of feature-location tasks and processes.

In summary, we:

- discuss different notions of features;
- identify and unify tasks that are considered as feature location;
- provide an overview of studies on manual feature location; and
- discuss open issues and problems identified in our review.

We describe feature location from a practitioners' perspective, hoping to provide actionable insights for developers. By consolidating existing knowledge on manual feature location, we also hope to foster the development of corresponding techniques, helping researchers to scope their work, and provide a starting point for further studies.

We proceed by discussing the notion of features in Section 2, since this notion differs among different communities and developers. Likewise, we show in Section 3 that different notions of feature location exist, comprising different kinds of tasks performed by developers. We discuss these tasks based on definitions provided in the literature, thereby illustrating the different notions of feature location and consolidating the distinct tasks into individual definitions. Thereafter, we provide a systematic review on studies about manual feature location, describing the review design in Section 4. We compare the studies' research approaches, their investigations of feature location, and their results. Within Section 5, we synthesize our observations and discuss them. Then, we discuss threats to validity in Section 6, related work in Section 7, and conclude in Section 8.

2. What is a Feature?

While not only most developers have their own notion of *feature*, the exact definitions provided by researchers also vary significantly (Apel et al. 2013; Berger et al. 2015). For example, there are more than ten definitions of the term *feature* (Classen, Heymans, and Schobbens 2008), ranging from rather abstract – "user-visible aspect, quality, or characteristic" (Kang et al. 1990) – to rather technical definitions – "an increment of program functionality" (Batory 2005). As a result, the concept of *feature* can be challenging to grasp, given the missing common definition. We rely on the definition of Apel et al. (2013), who consolidate previous definitions:

"A feature is a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle."

In this definition, features are foremost a means to communicate visible characteristics or observable behaviors of a system. This definition also explains the role of features in the context of a software product line (a portfolio of software variants) (Czarnecki and Eisenecker 2000; Clements and Northrop 2002; Apel et al. 2013), where features play an even more important role. There, features guide the structure of a system and can be reused among variants: Feature can be present in some variants of the software product line and absent in others, thereby reusing the feature. Enabling feature reuse requires implementing them in a specific way, for instance, as a component, which structures the software in a way that allows reuse. Such feature reuse also requires managing the variabilities represented by the features – commonly known as variability management, which comprises activities such as modeling features or managing feature dependencies.

For different contexts, different kinds of features are relevant, leading to different characteristics of the feature locations (Krüger, Gu, et al. 2018). In the context of product line engineering, the focus is primarily on managing (e.g., modeling) optional features, less on mandatory (i.e., common to all variants) features. The locations of such optional features are typically documented using preprocessor annotations (Apel et al. 2013; Medeiros et al. 2015), which allows switching features on and off when deriving individual variants from the product line. As such, the software-product-line engineering community is primarily concerned with optional features – as units of variability – and their locations. In contrast, other communities see features primarily as units of functionality, which is relevant for communities such as software maintenance. Here, a feature can be any visible characteristic or observable behavior of a system and is not necessarily configurable. This notion is broader and corresponds to the definition of Apel et al. (2013) that we rely on.

Features are a kind of software concern. While the latter term is more general, both terms are sometimes used synonymously (Wilde et al. 2003; Apel et al. 2013). A concern is defined as an area of interest in a system; and in software-product-line engineering, features are the concerns of primary interest (Apel et al. 2013). However, the broad definition of concerns means that they comprise many kinds of entities in different contexts – for instance, studies on locating concerns include identifying code that causes a bug, corresponds to a requirement, or implements a feature (Wilde et al. 2003; Ko et al. 2006a). This results from the fact that an area of interest can be essentially anything. Thus, features are a specific kind of concern, but not all concerns are features (Wilde et al. 2003).

3. A Clarification of Feature Location

When developers analyze a system, they may investigate a specific concern according to the task at hand. In this chapter, we focus on locating features as such concerns, excluding bug localization (Hangal and Lam 2002) or requirements traceability recovery (Antoniol et al. 2002; Oliveto et al.

2010), among others. Still, feature location is a term, similar to feature, that developers have a notion of (or a vague idea about), but these notions often slightly differ among the developers. Our literature analysis in fact reveals different notions, for instance:

(1) "Detection phase: initially, relevant information is extracted from the input artefacts, e.g. source code, to understand the existing structure, functionalities, data flow, relationships, existing features, etc. It is a discovery phase, often referred as feature location;"

(Assunção and Vergilio 2014)

(2) "Feature location is the activity of identifying the source code elements (i.e., methods) that implement a feature."

(Revelle, Broadbent, and Coppit 2005)

(3) "Developers often have to identify where and how a feature is implemented in the source code in order to fix bugs, introduce new features, and adapt or enhance existing features. This activity is referred to as feature location in the context of software engineering."

(Wang et al. 2011)

(4) "Feature location is the activity of identifying an initial location in the source code that implements functionality in a software system."

(Dit et al. 2013)

(5) "While the set of available features in many cases is specified by the product documentation and reports, the relationship between the features and their corresponding implementation is rarely documented. Identification of such relationships is the main goal of feature location techniques."

(Rubin and Chechik 2013)

(6) "Concept location is a process that maps domain concepts to the software components."

(Chen and Rajlich 2000)

Observe that each of these definitions comprises a slightly different meaning related to three tasks: Identifying, locating, and mapping features. Below, we will consolidate these definitions by distinguishing these three tasks from each other. The resulting definition will scope our literature analysis on manual feature location in this chapter.

3.1. Feature Identification

Most of the aforementioned definitions assume that the features to locate are already known and documented. However, this is rarely the case for software systems – except for optional features in software product lines – and can become a problem, especially in the context of variant-rich systems that are developed in an unsystematic clone-and-own approach (Dubinsky et al. 2013; Rubin, Czarnecki, and Chechik 2015; Krüger, Nell, et al. 2017; Krüger 2017). For example, it is questionable whether documentation (e.g., source code comments, models, or requirements) is regularly updated to reflect code changes (Jiang and Hassan 2006; Fluri, Wursch, and Gall 2007). Such inconsistencies can already cause problems in a single systems and may result in one clone developing divergently from the others, including features that are not documented and may be forgotten.

For these reasons, it is often necessary to first identify features that exist in a system, collect the corresponding information, and define their relationships. This is emphasized by definition (1), which describes feature location as the process of identifying existing features in a system. However, this task is more often referred to as feature identification and can utilize different input artifacts, such as documentation, models, and source code (Laguna and Crespo 2013). While the terms feature identification and feature location are often used synonymously, we clearly distinguish them. In feature identification, an exemplary output can be a variability model (Czarnecki et al. 2012; Berger et al. 2013) that describes the features of a system and their dependencies. We define feature identification as:

Definition:
Feature identification is the task of determining the features that exist in a system based on a set of artifacts, revealing their dependencies and documenting the result.

There are several approaches that support feature identification (Laguna and Crespo 2013), such as the ones by Acher et al. (2012) and Davril et al. (2013). Most of these studies are related to recovering variability models from legacy systems. This emphasizes the importance of documenting the identified features and their dependencies in our definition, and distinguishes such tasks from feature location.

3.2. Feature Location

Feature location is solely concerned with finding the source code that implements a specific feature. Knowing these locations is necessary to update, reuse, and maintain a feature (von Mayrhauser, Vans, and Howe 1997; Krüger, Pinnecke, et al. 2017). In practice, feature locations are rarely documented explicitly, but depend on the knowledge of developers. When this knowledge fades or gets lost (Ji et al. 2015, Krüger, Wiemann, et al. 2018), developers need to recover these locations. The location of a feature can be seen as a traceability link, which is a broader term (Antoniol et al. 2002; Oliveto et al. 2010), referring to various kinds of relationships between any kind of code or non-code artifacts, such as requirements and code. The understanding of feature location as the recovery of implementation relationships between (known) features and source code is shared by most definitions above, specifically (2), (3), and (4). Consequently, we define feature location as:

Definition:

Feature location is the task of finding the source code in a system that implements a feature.

Many fully or semi-automated feature-location techniques have been proposed, as surveyed by Rubin and Chechik (2013), Dit et al. (2013), and Assunção and Vergilio (2014). Recall that automated feature location is problematic, given their adaptation needs and low accuracy (Duszynski, Knodel, and Becker 2011; Ji et al. 2015; Krüger, Nell, et al. 2017; Krüger, Gu, et al. 2018) – motivating our analysis of manual feature location.

3.3. Feature Mapping

Definitions (5) and (6) emphasize the declaration of relationships – the mapping – between features and their implementing source-code artifacts. Since this declaration can be done using different means, it is a task on its own, and we distinguish it from identifying features and locating features

(Krüger, Nell, et al. 2017; Krüger, Gu, et al. 2018). For our purpose, we consider a specific task of this research area to which we refer to as feature mapping:

Definition

Feature mapping is the task of documenting the connection between a feature and its implementation in a system's source code.

Feature mapping can be done with different complementary techniques. For example, annotation-based implementation techniques for software product lines require explicit annotations (e.g., preprocessor annotations such as #IFDEFs) for optional features (Apel et al. 2013). These annotations establish the mapping of parts of code artifacts to features defined in a variability model, which is the input to a configurator tool. Other authors, such as, Revelle, Broadbent, and Coppit (2005), Ji et al. (2015), Seiler and Paech (2017), or Krüger, Gu et al. (2018) advocate to also annotate the locations of mandatory features using annotations that are embedded into the source code, regardless whether used in the context of a software product line or a single system. Other implementation techniques, such as feature-oriented programming (Prehofer 1997), physically separate features into different files (so-called modules) with a feature's name and achieve traceability this way.

3.4. Summary

In this section, we distinguished three different tasks that are sometimes synonymously described as feature location. We argued that the task of actual feature location differs from feature identification and feature mapping. However, these three tasks partly depend on each other and are tangled. For example, all three tasks can be performed in parallel when investigating source code, which may result in inconsistencies – for instance, when a feature is identified later on and all locations need to be reinvestigated to update the corresponding mappings. Also, without knowing about existing features and their locations in the source code, it is not possible to map both types of information. Overall, there are different approaches for each task and, especially for automation, a clear distinction is necessary to define input and output, or to measure the performance of a technique. In the remainder of this chapter, we will focus only on the task of feature location as we defined it above.

4. Methodology

Surveys show that a variety of automated feature-location techniques exist that utilize a variety of inputs and outputs (Dit et al. 2013; Rubin and Chechik 2013; Assunção and Vergilio 2014). However, little and mostly recent research investigates how developers actually perform feature location without automated techniques. Such research helps to understand the needs of practitioners, to identify challenges, and to design techniques based on real-world needs. In the following, we describe our review on manual feature location, roughly following the guidelines for systematic literature reviews described by Kitchenham and Charters (2007).

4.1. Research Questions

With our review, we aim to provide an overview on manual feature location to consolidate existing knowledge and to identify open issues. In particular, we are interested in the steps developers perform naturally when searching for a feature's code, without being supported by automated techniques. We formulate two research questions:

RQ_1 What topics are of primary interest in existing studies on manual feature location?

RQ_2 What topics are neglected in existing studies on manual feature location?

The first research question focuses on showing the state of the art of studies that report on developers performing manual feature location. In contrast, with the second research question, we aim to derive open issues based on the previous results.

To answer our research questions, we consolidate the research questions defined in each of the included studies into a topic. Such topics may be observing the search patterns of developers or measuring the effort it requires to manually locate a feature. For this purpose, we start with identifying those questions related solely to manual feature location, for example, excluding the comparison of automated techniques performed by Wilde et al. (2003). Then, we abstract more general topics by relating terms used in the research questions that are either identical or closely related.

4.2. Review Design

We start with an automated search in the DBLP database – last updated on the 13th of November 2017. Our search string only contains the term *"feature location"* as a broad term, aiming to capture all indexed documents on this research area. This search string results in 271 documents that we check manually, starting with title and abstract, and continuing with the whole document if necessary. We exclude studies that are not explicitly investigating and reporting findings about manual feature location at least as a part of the study (e.g., Wilde et al. (2003) use manually derived locations as a baseline for two automated feature-location techniques), are not written in English, or are not peer-reviewed. This way, we identify two relevant documents from the automated search.

For a more complete view, we apply backwards-snowballing on the selected papers' references and forward-snowballing (Wohlin 2014) using Google Scholar – last updated on the 15th of November 2017. For each newly identified document, we again apply snowballing and the same exclusion criteria as for our automated search. Here, we find four more relevant studies. Additionally, we include one of our own studies (Krüger, Gu, et al. 2018) that is concerned with manual feature location, but not indexed at this time.

Finally, we check our search results by using the SCOPUS database – last updated on the 26th of January 2018. This time, we use the more specific search string "*manual AND feature locat*"* on abstracts, keywords, and titles. The resulting 37 documents contain no new relevant studies, wherefore we finally include seven studies in our review. The small number of mostly recent studies already indicates that more research on manual feature location may be necessary to improve our understanding of this task. We remark that DBLP as well as SCOPUS are databases that index publications from most important publishers in computer sciences, including ACM, IEEE, Springer, Elsevier, and Wiley.

5. Results

In this section, we provide an overview on the identified seven studies on manual feature location. Afterwards, we summarize and discuss the research topics addressed in these studies, as well as open issues.

5.1. Identified Studies

Here, we briefly summarize the identified studies. We provide and overview on the used research method, considered systems, and participants to put the results into context. Moreover, we focus on the investigated research questions, based on which we derive the addressed research topics. In Table 1, we provide a brief overview on all studies included in this review. We remark, that study 3b is an extension of 3a, including an additional experiment, so we will consider these studies as a single one in the remaining chapter.

Table 1: Overview on the documents included in this review.

ID	Document
1	A Comparison of Methods for Locating Features in Legacy Software (Wilde et al. 2003) Journal of Systems and Software
2	Understanding Concerns in Software: Insights Gained from Two Case Studies (Revelle, Broadbent, and Coppit 2005) International Workshop on Program Comprehension
3a	An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions (Wang et al. 2011) International Conference on Software Maintenance
3b	How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study (Wang et al. 2013) Journal of Software: Evolution and Process
4	Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads

	(Jordan et al. 2015) International Conference on Program Comprehension
5	A Field Study of How Developers Locate Features in Source Code (Damevski, Shepherd, and Pollock 2016) Empirical Software Engineering
6	Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin (Krüger, Gu, et al. 2018) International Workshop on Variability Modelling of Software-Intensive Systems

Wilde et al. (2003) report a case study that includes manual feature location. They focus on comparing two automated techniques to the results of a single developer who manually analyzes an industrial FORTRAN system containing 2,335 lines of code. The developer is allowed to use only simple search tools, like those that are integrated in most development environments and text editors. Overall, Wilde et al. investigate three research objectives:

- Analyze the necessary effort to adopt any of the three techniques to the FORTRAN code;
- Investigate the pros and cons of each technique; and
- Identify inconsistencies between the results of all techniques.

Revelle, Broadbent, and Coppit (2005) describe a case study on concern location including two experienced developers. These developers analyze a C system with 2,100, and a Java system with 3,000 lines of code. We remark that, in this study, feature location is mingled with feature identification and feature mapping (cf. our previous definitions in Section 3). The authors compare the findings of the two developers to compare different feature characteristics and derive guidelines to locate and map them, for example, the authors derive the guide that entire functions often refer to a concern or support one.

Wang et al. (2011, 2013) perform three controlled experiments. The first experiment includes 18 students and two full-time developers as participants, who analyze two Java systems with around

72,900 and 2,300 lines of code. For the other two experiments, two different Java systems with 43,950 and 18,750 lines of code are investigated by 18 students. Overall, the authors answer five research questions related to actions, search patterns, and phases of feature location. They also analyze the influence of knowing the phases and patterns of feature location on developers' performance as well as of external factors that may influence developers' choices.

Jordan et al. (2015) observe two professional developers while these locate features in an industrial system. The system is implemented in COBOL with various domain-specific languages (DSLs), adding up to around 3 million lines of code overall. The authors' focus is on the used search tools and how effective the developers locate features.

Damevski, Shepherd, and Pollock (2016) conduct a field study in which they analyze tracking data of developers that use an integrated development environment, Visual Studio, to locate features. The first part of the study is based on a dataset provided by the company ABB. The dataset contains data of 67 developers for who all interactions with Visual Studio have been tracked for an average of 25.6 days. Data for the second part of the study is derived with a plug-in that around 600 developers downloaded and that tracks search queries of the users. Again, the focus of this study is on the used search tools and queries, but also on the patterns Wang et al. (2011, 2013) defined.

Finally, Krüger, Gu, et al. (2018) report a case study on the Marlin 3D-printer firmware that comprises over 10,000 lines of C code. In this study, two of the authors manually locate features in one variant of Marlin. The focus in this study is on possible entry points to identify and locate features, as well as comparing the characteristics of mandatory and optional features.

We summarize the characteristics of these studies in Table 2. We can see that most of the conducted research is focusing solely on feature location, relying on already identified features for the location task. Revelle, Broadbent, and Coppit (2005) and Krüger, Gu, et al. (2018) explicitly state and investigate the identification and mapping of features in the source code. Considering the remaining studies, Wilde et al. (2003) compare the located features of different techniques, which requires a mapping, and Wang et al. (2011, 2013) also investigate documenting located features, which in this case means to take notes about the features, but apparently not explicitly mapping them to the code. Considering the remaining characteristics, we see that different research methods have been applied, namely case studies, experiments, and field studies, that all observe the behavior of the participants. In addition, the used systems and artifacts differ in the programming languages, size, and number of subjects that performed the location tasks.

Table 2: Characteristics of the identified studies.

ID	Identification	Location	Mapping	Method	Systems		Subjects
1	✗	✓	✓	Case Study	1 FORTRAN	2,350 LOC	1
2	✓	✓	✓	Case Study	1 C 1 Java	2, 100 LOC 3,000 LOC	2
3	✓	✓	✗	Experiment	4 Java	72,900 LOC 2,300 LOC 43,950 LOC 18,750 LOC	20/18
4	✗	✓	✗	Field study	1 COBOL, DSLs	3 Mio. LOC	2
5	✗	✓	✗	Field study	/	/	67/~600
6	✓	✓	✓	Case Study	C	/	2

5.2. RQ$_1$: What Topics are of Primary Interest in Existing Studies on Manual Feature Location?

In order to answer this research question, we analyze the questions and objectives addressed in the identified studies. Here, we identify common and related terms to generalize the investigated topics. For example, several research questions use the terms *search tools*, *actions*, *drawbacks*, and *factors*.

We show the results of this abstraction in Table 3. We will discuss each of these topics and the corresponding findings in the following.

Table 3: Identified research topics.

ID	Search Tools	Performance	Factors	Processes
1	✓	✓	✗	✗
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✗
5	✓	✗	✗	✓
6	✗	✗	✓	✗

5.2.1. Search Tools

Three of the identified studies particularly address how developers use standard *search tools*, as are included in most development environments and text editors, to locate the source code belonging to a feature. In general, these studies indicate that search tools are an important means for developers to search for terms that are related to a feature. The identified locations may correspond to the feature and can be used as starting points (a so-called seeds) to extend the search.

Jordan et al. (2015) as well as Damevski, Shepherd, and Pollock (2016) observe that their participants usually start with simple search queries, including single keywords that are later refined based on the results. For instance, a query may be specified by adding a term. Especially Jordan et al. (2015) also exhibit different triggers that result in a search and the origin of the used keywords. Their participants mainly use searches to browse the code, to refine a previous search, to inspect results, and to identify new seeds for a feature. To define their search queries, they rely on language syntax and conventions, terms in the source code, and the results of previous searches.

While Revelle, Broadbent, and Coppit (2005) find that such searches are fast and require no adaptation in contrast to automated techniques, they also remark a low accuracy of such searches. They argue that the reliance on keywords appearing in comments and source code reduces the success of simple search queries in contrast to more advanced techniques. In contrast, Jordan et al. (2015) find that searches performed by developers that are familiar with the system are likely to return relevant results, with only 29% of the investigated searches being unsuccessful. This is not only contradicting the findings of Revelle, Broadbent, and Coppit (2005), but also other studies on search behavior, for example by Ko et al. (2006). Such studies are more generally interested in how developers search for information during software development tasks, namely for debugging and program enhancement in the study of Ko et al. (2006). Thus, they may also provide additional insights into this topic, but not with a focus on feature location.

5.2.2. Performance

The topic *performance* summarizes all findings related to the effectiveness of manual feature location, as well as its advantages and disadvantages in practice. While three studies address this topic, they always mingle it with another topic, making it problematic to draw isolated results about the topic *performance*. Specifically, Revelle, Broadbent, and Coppit (2005) find that manual location, focusing on simple search tools, is fast to set up and requires no adaptation effort – but is also not reliable. The same accounts for Jordan et al. (2015) who consider the performance of search tools themselves. Finally, Wang et al. (2011, 2013) compare the performance of their participants before and after the participants gain knowledge about feature-location patterns and processes, combining the topics *factors* (i.e., knowledge) as well as *processes* (i.e., feature-location patterns and processes) with *performance*. Overall, there are no reliable findings about the efforts and *performance* that are actually related to the feature-location task or parts of it, as the identified results are always intermingled with other topics.

5.1.3. Factors

The topic *factors* summarizes all influences (e.g., system characteristics, task at hand, developer characteristics) on the feature-location task, for example, how developers perform searches or how they conduct specific steps. A similar distinction is identified by Wang et al. (2013), who differentiate between three factors, listed from most to least important (regarding how searches are performed):

(1) *Human factors* refer to knowledge, experiences, capabilities or preferences of a developer. Wang et al. (2013) find that such factors have the strongest impact on feature location, which is identical to the findings of Revelle, Broadbent, and Coppit (2005) as well as Jordan et al. (2015), stating program comprehension and knowledge as significant factors.

(2) *Task properties* include factors that are connected to the task at hand, including characteristics of the system as well as the features under investigation. These findings are again identical to Revelle, Broadbent, and Coppit (2005), who identify the abstraction layer of features and the program context as further factors. Addressing the factors of the features of a system themselves, Krüger, Gu, et al. (2018) compare the characteristics of mandatory and optional features in a software product line. The findings indicate that locating the different kinds of features may require varying effort, as they are scattered differently among their subject system.

(3) *In-process feedback* refers to any feedback the developer receives during the feature-location task, for example, the number of results a search query returns. This factor in particular seems to have less effect on the search behavior, as a search query can be reuse. However, the factors have more impact on the actual actions performed after receiving the results. An example of Wang et al. (2013) are searches with too many results being reduced by a refined search, not changing the search behavior but the developer’s actions – repeating another search instead of investigating the results.

Here, we see that several factors have been identified to influence feature-location tasks. These are reported repeatedly and, thus, to some extent verified by the other studies.

5.2.4. Processes

The actual processes and actions performed during manual feature location are intensively investigated by Wang et al. (2011, 2013). Here, we also include the guidelines for feature location by Revelle, Broadbent, and Coppit (2005) and partly consolidate them into the processes for feature location, as they resemble similar steps. In Figure 1, we show an adapted version of the feature-location phases defined by Wang et al. (2013), which is also used by Damevski, Shepherd, and Pollock (2016), and reflected by the results of Revelle, Broadbent, and Coppit (2005) – for example, considering the recommendation that functions or class attributes can indicate feature seeds. We remark that we do not show the documentation phase, which we relate to feature identification and feature mapping according to our definitions (cf. Section 3). Consequently, for this survey the documentation phase is out of the scope of processes for feature location.

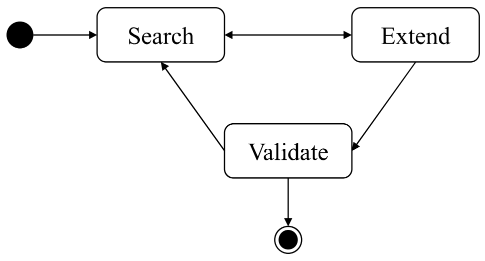

Figure 1: Feature-location phases adapted from Wang et al. (2013).

The search phase describes the initial step of feature location, in which the developers search for one or few initial seeds for a feature. Wang et al. (2013) identify three search patterns that developers commonly apply:

1. *Information retrieval-based* patterns are adaptations of information retrieval techniques, but also include simple search tools. As identified in other studies, the developers identify keywords that relate to the feature they aim to locate and then derive a search query. Depending on the results, they change the queries to achieve better results.
2. *Execution-based* patterns require the developer to derive execution scenarios of the program in which the feature is active. They then set breakpoints (as for debugging) in the source code, wherever they assume the feature may be implemented. If the program stops at a breakpoint, the

developers have to investigate the source code and refine these points to find seeds of a feature step-by-step.

3. *Exploration-based* patterns do not rely on keyword searches or program execution, but on the static dependencies in the code. To this end, developers read the source code and aim to understand its behavior, for example, by investigating method calls or type hierarchies, to locate a feature.

After this phase, the developers are aware of seeds in the source code from which they continue to refine feature locations in the extension phase. During this extension phase, developers often rely on their own code comprehension and two types of search patterns described above: Execution-based and exploration-based extension patterns. However, the purpose of this phase is different, which also reflects on the actual actions performed in this phase: To locate all code belonging to a feature, developers extend the seeds identified previously. As a result, they often return to these seeds and restart their searches from there. In their study, Damevski, Shepherd, and Pollock (2016) find that almost all their participants rely on the exploration-based patterns (97%) to extend feature locations, while only few participants use execution-based patterns (3%). Finally, the developers need to validate whether the identified code belongs to the investigated feature. This can be done by analyzing the code, for example, based on its structure, comments, or further keywords. Wang et al. (2013) find that their participants often edit the code and then execute it to validate the output, or they rely on debugging.

5.2.5. Summary

In this section, we consolidated the findings of the seven included studies. With respect to RQ$_1$, we find that the studies investigate manual feature location in terms of search tools, performance, influencing factors, and occurring processes as well as patterns. Especially the studies of Wang et al. (2011, 2013) provide detailed insights into the behavior of practitioners and can help to better understand the feature-location task.

5.3. RQ$_2$: What Topics are Neglected in Existing Studies on Manual Feature Location?

As we described, the seven studies investigate the topics search tools, performance, factors, and processes in detail. Still, there are several important issues not yet analyzed, which we discuss in this section.

5.3.1. Studies on Efforts of Feature Location

Despite the extensive analyses, we find very little information about the actual efforts of locating features, for example, only as the number of recorded material or duration of a study. While several researchers find feature location to be the most expensive task in software maintenance, detailed analysis on the actual efforts are missing. This is problematic for several reasons: First, we do not understand what actions for feature location require what efforts. This knowledge would help to identify the actions that require most support to facilitate the whole feature-location task. For example, most automated techniques propose seeds for a feature, but if simple search tools are more accessible and provide comparable or even better results than automated techniques, this may not reduce efforts in practice. Second, we do not know whether physical actions or mental actions require more support. For example, if the effort is mainly connected to deriving suitable queries, automatically providing suggestions to the developer may be more helpful than fully automated searches. Third, missing understanding of efforts makes a comparable assessment of benefits and costs of feature-location techniques impossible. Without knowing the costs of performing the automated tasks manually, we cannot evaluate if the derived technique is helpful. Overall, we argue that the limited knowledge about the efforts of feature location prevents us from understanding its actual challenges and problems. Furthermore, this missing knowledge hampers cost estimations for software maintenance and reengineering; and is a threat to feature-location techniques, as they may try to facilitate actions that are already simple to perform.

5.3.2. Performance Studies and Factors

Several authors remark that different factors have positive impact on the feature-location task. However, detailed analyses or experiments in this regard are missing. For example, it is interesting to investigate if specified notions of features or programming patterns result in identical results. The studies by Wilde et al. (2003), Revelle, Broadbent, and Coppit (2005), Wang et al. (2013), and Jordan et al. (2015) find that the experience and knowledge of the developer is important, but do not reveal the actual positive or negative effects of these factors on the performance. Corresponding studies can help to identify and assess factors that considerably impact feature location. The results can help to scope automated techniques, to let developers focus on specific parts of the system based on their knowledge, and to avoid potential pitfalls.

5.3.3. Evaluating and Scoping Automated Techniques

A common problem of existing automated feature-location techniques is their low accuracy and that it is often unclear what steps they support to what extent. In this regards, it would be helpful to conduct more studies that compare such techniques to manual feature location. We only find the study of Wilde et al. (2003) addressing this issue. While they report a lower accuracy for the manual analysis compared to a reference study, this may only indicate that there are different notions of features. Thus, such studies can be helpful to not only compare, but also to scope techniques to the notions of developers. For this purpose, it seems also necessary to clearly define what tasks, phases, and actions are addressed by a feature-location technique. Such definitions are clearly missing, especially since the first feature-location techniques were already proposed, at least, in 1995 (Rubin and Chechik 2013). It seems curious that automation has been proposed long before the manual tasks that they support have been investigated. We argue that a comparison of existing, automated feature-location techniques with the already identified phases and patterns helps to scope the application scenarios of automated techniques.

5.3.4. Information Sources

While feature location is concerned with finding feature implementations in the source code, relevant information may be contained in artifacts beyond code. In fact, for feature identification, more and more researchers propose to rely on other information sources, for example, histories of modern code repositories and version control systems, such as GitHub or BitBucket (Krüger, Gu et al. 2018), or natural language processing of requirements documentations (Li et al. 2017). Especially repositories and version control systems already are common in software development for a long time and provide detailed information about code changes. Analyzing the additional data, such as the commit messages, and correlating changes in the same commit or among branches with each other may facilitate (automated) feature location. Still, also in this context, manual studies may provide detailed insights on analysis patterns that are different from the already identified ones.

5.2.5. Summary

With respect to RQ_2, we identify four open issues for further investigations. These include studies on the effort of feature location, factors that influence the performance, the scope and evaluation of automated techniques, and the used information sources. Additional studies are needed to address these open issues in feature-location research.

Recall the mapping study of Assunção and Vergilio (2014) on automated feature location (cf. Section 3). We find some similarities of their identified open research issues to ours above. Specifically, the authors also see the need to improve evaluations, combine information sources, and to conduct further studies to improve automation. These are open issues we share and that we emphasize can only be investigated by systematically observing actual developers performing feature location.

6. Threats to Validity

A threat to the external validity is that we focused on a specific research area and used narrow search terms. Thus, we may have missed some studies that are concerned with manual feature location or closely related areas that may include valuable insights. However, we mitigated these effects by applying snowballing on our search results to broaden the scope of our review. Consequently, we argue that we provide a detailed overview on research for manual feature location.

We discussed only four open issues that are derived from our own analysis of the included studies. However, we carefully derived these issues by analyzing gaps between the investigated topics in each study. Thus, while several other issues exist, we argue that they are a call for attention to improve our understanding on manual feature location. While these issues may be biased to our own opinions, other researchers can easily derive additional ones based on an analysis of our sruvey. Also, we focused on issues that hamper the practical adoption and support for feature location to support practitioners in understanding the problems connected to these. Here, we see that especially performances and costs are the most interesting topics.

Finally, we argue that any researcher can replicate and extend our review. We followed established guidelines for systematic literature reviews, described each step, and provided an overview on all included studies. Depending on future studies and the scope of a replication, the findings may slightly differ but should comprise the same results.

7. Related Work

There are some related works to our survey on manual feature location: Several surveys have been conducted on feature-location techniques (Dit et al. 2013; Rubin and Chechik 2013; Assunção and Vergilio 2014). These surveys compare different techniques, including, for instance, their underlying approach and potential application scenarios. These techniques support different aspects of the tasks we identified previously, mainly the processes to identify, locate, and map features in the source code. This is often done by identifying and extending feature seeds. Thus, these surveys extend the search tools we identified as a research topic and mostly evaluate their performance. Still, this is often done on single case studies and by comparing feature-location techniques. As we described above, there are some open issues in connecting the manual processes of feature location with such automated techniques.

Works on concept location investigate not only features, but also bugs or requirements (Wilde et al. 2003; Robillard and Murphy 2007). As a result, different approaches, for example on requirements traceability and recovery (Antoniol et al. 2002), also cover related techniques, but with a different scope compared to feature location. Similar to feature location, few studies actually are concerned with the manual processes in these domains. In this regard, especially studies on search processes and software maintenance activities provide detailed insights, as they often observe developers during their daily work (von Mayrhauser, Vans, and Howe 1997; Ko et al. 2006; Tiarks 2011). Thus, these studies can help to better understand how developers actually perform such maintenance tasks and processes, similar to the identified studies on feature location.

8. Conclusion

In this chapter, we reviewed studies on manual feature location. We provided an overview on the processes and steps connected to feature location, summarizing the state of the art to support practitioners in applying, and researchers in improving, automated and manual feature location. To this end, we identified that the most addressed topics are the used search tools, factors that influence feature location, performance measures, and unique phases. Moreover, we discussed open issues that still prevent the practical adoption of feature-location techniques: Missing knowledge about the required efforts and the influence of external factors, missing evaluations of techniques, and the exclusion of additional information sources. In summary, further studies on manual feature location are necessary to improve our understanding about such topics, to elicit empirical data and to improve feature-location practices.

Acknowledgements

Supported by the German Research Foundation (DFG) grant number 3382/2-1, and the ITEA project REVaMP^2, funded by Vinnova Sweden (2016-02804) and the Swedish Research Council Vetenskapsrådet (257822902).

References

Acher, Mathieu, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet, and Philippe Lahire. 2012. "On Extracting Feature Models from Product Descriptions." In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 45–54. ACM. doi:10.1145/2110147.2110153.

Antoniol, Giuliano, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. “Recovering Traceability Links Between Code and Documentation.” *IEEE Transactions on Software Engineering* 28 (10): 970–83. doi:10.1109/TSE.2002.1041053.

Apel, Sven, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer. doi:10.1007/978-3-642-37521-7.

Assunção, Wesley Klewerton Guez, and Silvia Regina Vergilio. 2014. “Feature Location for Software Product Line Migration: A Mapping Study.” In *International Systems and Software Product Line Conference (SPLC)*, 52–59. ACM. doi:10.1145/2647908.2655967.

Batory, Don. 2005. “Feature Models, Grammars, and Propositional Gormulas.” In *International Systems and Software Product Line Conference (SPLC)*, 7–20. Springer. doi:10.1007/11554844_3.

Berger, Thorsten, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. “What Is a Feature? A Qualitative Study of Features in Industrial Software Product Lines.” In *International Conference on Software Product Line (SPLC)*, 16–25. ACM. doi:10.1145/2791060.2791108.

Berger, Thorsten, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. “A Survey of Variability Modeling in Industrial Practice.” In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 1–8. ACM. doi:10.1145/2430502.2430513.

Biggerstaff, Ted J., Bharat G. Mitbander, and Dallas Webster. 1986. “The Concept Assignment Problem in Program Understanding.” In *International Conference on Software Engineering (ICSE)*, 482–98. IEEE. doi:10.1109/ICSE.1993.346017.

Chen, Kunrong, and Václav Rajlich. 2000. “Case Study of Feature Location Using Dependence Graph.” In *International Workshop on Program Comprehension (IWPC)*, 241–47. IEEE. doi:10.1109/WPC.2000.852498.

Classen, Andreas, Patrick Heymans, and Pierre-yves Schobbens. 2008. “What’s in a Feature: A Requirements Engineering Perspective.” In *International Conference on Fundamental*

Approaches to Software Engineering (FASE), 16–30. Springer. doi:10.1007/978-3-540-78743-3_2.

Clements, Paul, and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Czarnecki, Krzysztof and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

Czarnecki, Krzysztof, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. “Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches.” In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, 173–82. ACM. doi:10.1145/2110147.2110167.

Damevski, Kostadin, David Shepherd, and Lori Pollock. 2016. “A Field Study of How Developers Locate Features in Source Code.” *Empirical Software Engineering* 21 (2): 724–47. doi:10.1007/s10664-015-9373-9.

Davril, Jean-Marc, Edouard Delfosse, Negar Hariri, Mathieu Acher, Jane Cleland-Huang, and Patrick Heymans. 2013. “Feature Model Extraction from Large Collections of Informal Product Descriptions.” In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 290–300. ACM. doi:10.1145/2491411.2491455.

Dit, Bogdan, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. “Feature Location in Source Code: A Taxonomy and Survey.” *Journal of Software: Evolution and Process* 25 (1): 53–95. doi:10.1002/smr.567.

Dubinsky, Yael, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. “An Exploratory Study of Cloning in Industrial Software Product Lines.” In *European Conference on Software Maintenance and Reengineering (CSMR)*, 25–34. IEEE. doi:10.1109/CSMR.2013.13.

Duszynski, Slawomir, Jens Knodel, and Martin Becker. 2011. “Analyzing the Source Code of Multiple Software Variants for Reuse Potential.” In *Working Conference on Reverse Engineering (WCRE)*, 303–7. IEEE. doi:10.1109/WCRE.2011.44.

Fluri, Beat, Michael Wursch, and Harald C. Gall. 2007. “Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes.” In *Working Conference on Reverse Engineering (WCRE)*, 70–79. IEEE. doi:10.1109/WCRE.2007.21.

Hangal, Sudheendra, and Monica S. Lam. 2002. “Tracking Down Software Bugs Using Automatic Anomaly Detection.” In *International Conference on Software Engineering (ICSE)*, 291–301. ACM. doi:10.1145/581339.581377.

Ji, Wenbin, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. “Maintaining Feature Traceability with Embedded Annotations.” In *International Systems and Software Product Line Conference (SPLC)*, 61–70. ACM. doi:10.1145/2791060.2791107.

Jiang, Zhen Ming, and Ahmed E. Hassan. 2006. “Examining the Evolution of Code Comments in PostgreSQL.” In *International Workshop on Mining Software Repositories (MSR)*, 179–80. ACM. doi:10.1145/1137983.1138030.

Jordan, Howell, Jacek Rosik, Sebastian Herold, Goetz Botterweck, and Jim Buckley. 2015. “Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads.” In *International Conference on Program Comprehension (ICPC)*, 174–77. IEEE. doi:10.1109/ICPC.2015.26.

Kang, Kyo Chul, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. “Feature-Oriented Domain Analysis (FODA) Feasibility Study.” Pittsburgh.

Kästner, Christian, Alexander Dreiling, and Klaus Ostermann. 2014. “Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features.” *IEEE Transactions on Software Engineering* 40 (1): 67–82. doi:10.1109/TSE.2013.45.

Kitchenham, Barbara A., and Stuart Charters. 2007. “Guidelines for Performing Systematic Literature Reviews in Software Engineering.” EBSE-2007-01.

Ko, Andrew J., Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006a. “An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks.” *IEEE Transactions on Software Engineering* 32 (12): 971–87. doi:10.1109/TSE.2006.116.

Krüger, Jacob, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2018. "Do You Remember This Source Code?" In *International Conference on Software Engineering (ICSE)*. ACM. doi:10.1145/3180155.3180215.

Krüger, Jacob. 2017. "Lost in Source Code: Physically Separating Features in Legacy Systems." In *International Conference on Software Engineering (ICSE)*, 461–62. IEEE. doi:10.1109/ICSE-C.2017.46.

Krüger, Jacob, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. "Extracting Software Product Lines: A Cost Estimation Perspective." In *International Systems and Software Product Line Conference (SPLC)*, 354–61. ACM. doi:10.1145/2934466.2962731.

Krüger, Jacob, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. "Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin." In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), 105–112*. ACM. doi: 10.1145/3168365.3168371.

Krüger, Jacob, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. "Finding Lost Features in Cloned Systems." In *International Systems and Software Product Line Conference (SPLC)*, 65–72. ACM. doi:10.1145/3109729.3109736.

Krüger, Jacob, Marcus Pinnecke, Andy Kenner, Christopher Kruczek, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. "Composing Annotations Without Regret? Practical Experiences Using FeatureC." *Software: Practice and Experience*. doi:10.1002/spe.2525.

Laguna, Miguel A., and Yania Crespo. 2013. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring." *Science of Computer Programming* 78 (8). Elsevier: 1010–34. doi:10.1016/j.scico.2012.05.003.

Li, Yang, Sandro Schulze, and Gunter Saake. 2017. "Reverse Engineering Variability from Natural Language Documents: A Systematic Literature Review." In *International Systems and Software Product Line Conference (SPLC)*, 133-142. ACM. doi: 10.1145/3106195.3106207

Medeiros, Flávio, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. “The Love/Hate Relationship with the C Preprocessor: An Interview Study.” In *European Conference on Object-Oriented Programming (ECOOP)*, edited by John Tang Boyland, 37:495–518. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2015.495.

Oliveto, Rocco, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2010. “On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery.” In *International Conference on Program Comprehension (ICPC)*, 68–71. IEEE. doi:10.1109/ICPC.2010.20.

Poshyvanyk, Denys, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. 2007. “Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval.” *IEEE Transactions on Software Engineering* 33 (6): 420–32. doi:10.1109/TSE.2007.1016.

Prehofer, Christian. 1997. “Feature-Oriented Programming: A Fresh Look at Objects.” In *European Conference on Object-Oriented Programming (ECOOP)*, 1241:419–43. Springer. doi:10.1007/BFb0053389.

Revelle, Meghan, Tiffany Broadbent, and David Coppit. 2005. “Understanding Concerns in Software: Insights Gained from Two Case Studies.” In *International Workshop on Program Comprehension (IWPC)*, 23–32. IEEE. doi:10.1109/WPC.2005.43.

Robillard, Martin P., and Murphy, Gail C. 2007. “Representing Concerns in Source Code.” *ACM Transactions on Software Engineering Methodology* 16 (1): Article 3. doi:10.1145/1189748.1189751.

Rubin, Julia, and Marsha Chechik. 2013. “A Survey of Feature Location Techniques.” In *Domain Engineering*, 29–58. Springer. doi:10.1007/978-3-642-36654-3_2.

Rubin, Julia, Krzysztof Czarnecki, and Marsha Chechik. 2015. “Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines.” *International Journal on Software Tools for Technology Transfer* 17 (5). Springer: 627–46. doi:10.1007/s10009-014-0347-9.

Seiler, Marcus and Barbara Paech. 2017. "Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems." In *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, 174–180. Springer.

Siegmund, Janet. 2016. "Program Comprehension: Past, Present, and Future." In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 13–20. IEEE. doi:10.1109/SANER.2016.35.

Standish, Thomas A. 1984. "An Essay on Software Reuse." *IEEE Transactions on Software Engineering* SE-10 (5): 494–97. doi:10.1109/TSE.1984.5010272.

Tiarks, Rebecca. 2011. "What Maintenance Programmers Really Do: An Observational Study." In *Workshop Software Reengineering (WSR)*, 36–37.

von Mayrhauser, Anneliese, A. Marie Vans, and Adele E. Howe. 1997. "Program Understanding Behaviour During Enhancement of Large-Scale Software." *Journal of Software Maintenance: Research and Practice* 9 (5): 299–327. doi:10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S.

Wang, Jinshui, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. "An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions." In *International Conference on Software Maintenance (ICSM)*, 213–22. IEEE. doi:10.1109/ICSM.2011.6080788.

———. 2013. "How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study." *Journal of Software: Evolution and Process* 25 (11): 1193–1224. doi:10.1002/smr.1593.

Wilde, Norman, Michelle Buckellew, Henry Page, Václav Rajlich, and La Treva Pounds. 2003. "A Comparison of Methods for Locating Features in Legacy Software." *Journal of Systems and Software* 65 (2): 105–14. doi:10.1016/S0164-1212(02)00052-3.

Wohlin, Claes. 2014. "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering." In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 1–10. ACM. doi:10.1145/2601248.2601268.