Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# Feature dependencies in automotive software systems: Extent, awareness, and refactoring

Andreas Vogelsang

*Technische Universität Berlin, Germany*

ARTICLE INFO

*Article history:*
Received 21 March 2019
Revised 4 September 2019
Accepted 1 November 2019
Available online 7 November 2019

*Keywords:*
Feature interaction
Automotive
Requirements engineering
Software architecture
Technical debt
Empirical study

ABSTRACT

Many automotive companies consider their software development process to be feature-oriented. In the past, features were regarded as isolated system parts developed and tested by developers from different departments. However, in modern vehicles, features are more and more connected and their behavior depends on each other in many situations. In this article, we describe how feature-oriented software development is conducted in automotive companies and which challenges arise from that. We present an empirical analysis of feature dependencies in three real-world automotive systems. The analysis shows that features in modern vehicles are highly interdependent. Furthermore, the study reveals that developers are not aware of these dependencies in most cases. For the three examined cases, we show that less than 12% of the components in the system architecture are responsible for more than 90% of the feature dependencies. Finally, we propose a refactoring approach for implicit communal components, which makes them explicit by moving them to a dedicated platform component layer.

## 1. Introduction

Software development in automotive companies is strongly influenced by existing legacy systems, organizational constraints, and complex OEM/supplier relationships (Broy, 2006). Nevertheless, automotive companies are forced to quickly deliver increasingly complex software to keep up with their competitors and other digital products with shorter development life-cycles. In this context, like in many others, short-term goals, such as the delivery of a feature, frequently trump long-term objectives like maintainability or extensibility (Martini et al., 2014).

The development of the software system in an automobile is characterized by a decomposition into vehicle domains such as powertrain, body, chassis, driver assistance, and infotainment. Within these vehicle domains, subsystems group and structure several vehicle features that provide functionality to the driver or other external systems (Broy et al., 2007). Examples for vehicle features are *airbag, cruise control*, or *start-stop system*.

Automotive companies try to keep features as independent as possible from each other because they usually structure their organization and resources based on features (e.g., *airbag* and *cruise control* can be developed in completely different departments). However, in the past years, the different features of a vehicle got more and more interconnected to provide innovative behavior (Broy, 2006). For example, the central locking system integrates the pure functionality of locking and unlocking car doors with comfort features (such as adjusting seats, mirrors, and radio tuners according to the specific key used during unlocking), with safety/security features (such as locking the car beyond a minimum speed, arming a security device when the car is locked, and unlocking the car in case of a crash), and with human-machine-interface features, such as signaling the locking and unlocking using the car's interior and exterior lighting system.

A feature is implemented through a network of communicating components. Technically, a component is a piece of software deployed to a hardware execution unit, which is connected to one or more bus systems that provide signals from all kinds of other components. The signals on a bus system are available to all components connected to that bus. Therefore, it is a common practice of developers to (re)use any signal that is available on the bus system to implement or adapt a feature, regardless of the origin of that signal. This practice leads to behavioral dependencies between features, some of which are intended and some of which are unintended.

Behavioral dependencies between features (a.k.a. feature interactions (Zave, 1999)) have been observed and addressed first in telecommunication systems (Calder et al., 2003) followed by studies on Internet applications (Crespo et al., 2007), service systems (Weiss et al., 2005), automotive systems (Vogelsang and Fuhrmann, 2013), software product lines (Jayaraman et al., 2007), computational biology (Donaldson and Calder, 2012), and in many

*E-mail address:* andreas.vogelsang@tu-berlin.de

https://doi.org/10.1016/j.jss.2019.110458
0164-1212/© 2019 Elsevier Inc. All rights reserved.

other fields outside of computer science. Several studies show that feature dependencies have a negative impact on maintenance efforts (Ribeiro et al., 2011; Cafeo et al., 2016), increase the likelihood of integration failures (Cataldo and Herbsleb, 2011), and prevent modular reasoning (Kästner et al., 2008).

Since the development of automotive systems is structured according to features, our research goal is to analyze the extent and awareness of feature dependencies in practice empirically. We are interested in how many feature dependencies actually exist in real-world automotive systems, whether the developers are aware of these, and whether the dependencies play a role in the way the systems are built.

To answer these questions, we had the chance to examine three automotive software systems from practice. More specifically, each system was characterized by a set of features that it provides, a set of components with interface descriptions that implement the features, and a feature-component mapping that indicates which components contribute to the implementation of which features. Since there was no notion of feature dependencies in the datasets (nor in any other artifact of the company), we developed an algorithm to extract feature dependencies from the component architecture.

With this algorithm, we found numerous feature dependencies that crosscut the whole system. In a follow-up interview study, we found that the respective developers were unaware of almost 50% of the dependencies. Moreover, we were able to show that feature dependencies are not considered systematically when it comes to restructuring the system's architecture although implicit feature dependencies can be considered as technical debt (Vogelsang et al., 2016). Therefore, we propose a dependency-based refactoring approach that suggests shifting components from features to a dedicated platform component layer if they are strongly affected by feature dependencies.

In summary, we describe the following contributions in this paper:

1. We propose an algorithmic approach for extracting feature dependencies from component architectures.
2. By analyzing three automotive software systems from practice, we show that feature dependencies are numerous and crosscut the whole system.
3. By confronting developers with these dependencies and analyzing so-called *service* features, we show that feature dependencies are hardly known and considered in the development of the system's architecture.
4. We propose a dependency-based refactoring approach for system components, which is able to reduce the number of feature dependencies by 90% by refactoring less than 12% of the components in the examined systems.

*Structure of the paper:* This paper is structured along the questions of extent, awareness, and refactoring of feature dependencies. After providing some background information and introducing the dependency extraction algorithm in Section 2, we analyze the study object systems with respect to extent of feature dependencies (Section 3) and awareness of feature dependencies (Section 4). In Section 5, we introduce our refactoring approach and show its application to the three systems. In Section 7, we present alternative solutions to ours before concluding the paper with a discussion and summary.

*Relation to previous work:* This article summarizes and extends the work of previous publications (Vogelsang et al., 2016; 2012; Vogelsang and Fuhrmann, 2013). We extend the previous work by the following contributions:

- We extend the analysis of RQs 1–2 and 5–6, which have already been addressed in previous work, by an additional dataset that is larger than the existing two datasets. By this, we enhance the external validity of our previous work. In addition, we provide a more in-depth discussion of the results.
- We extend RQ2 with a new analysis that correlates the number of feature dependencies associated with a component with the position of a that component in a feature processing chain. This analysis shows details about the role of dependencies in different architectural stages of a feature (e.g., sensing, processing, actuation).
- We address a new research question RQ4 in the context of the new dataset. In this RQ, we examine the relation between feature dependencies and so-called service features that developers defined in the new dataset. The purpose of these service features is that they provide platform functionality available for use in other features of the vehicle. The explicit definition of service features in the new dataset allowed us to examine whether feature dependencies are more frequent in service features compared with regular features. This analysis provides an additional viewpoint to the question of how aware developers are of feature dependencies.
- We explain the dependency extraction algorithm in more detail and provide a characterization as pseudo code. In addition, we publish the tool that we developed to perform the feature dependency analysis. This increases the reproducibility and transparency of our analysis and allows other researchers to reuse the analysis.

## 2. Background

### 2.1. Features and feature dependencies

The term feature is associated with a great variety of meanings and interpretations in research and industry. Additional terms that are often mentioned in this context are the terms *function* or *service*. Depending on the focus, the term feature may be used to describe distinctive characteristics of a system (Kang et al., 1990; Chen et al., 2005), elements of a functional specification (Shaker et al., 2012; Schätz, 2008), or increments and configuration options in a design or implementation (Liu et al., 2006; Apel et al., 2010).

In this article, we focus on features as elements of a functional specification for a multifunctional system (cf. Broy, 2010; Batory et al., 2004). This means features are used to structure the functionality of a system with the goal to decompose the specification. Decomposition into completely independent features is usually not possible and also not desirable in many cases. The goal is to break down the functionality into features with small and clear interfaces to each other to allow for a modular and distributed development. For our work, it is not important whether a feature also represents a configuration option. Our analysis focuses on features and their dependencies that are part of one specific product.

Based on the different notions of a feature, the notion of feature dependencies also differs. In the context of software product lines, feature dependencies are understood as constraints over the possible configuration space of the product line (Apel et al., 2013a). The constraints may be specified by logic relations between features such as *requires* or *excludes*. We do not focus on this interpretation of feature dependencies in this article. Several researchers focus on code-level implementations of software product lines and the challenges of feature dependencies for the development process. Cafeo et al. define: "In the source code, a feature dependency occurs whenever one or more program elements within the boundaries of a feature depend on elements external to that feature, such as a method defined in one feature and called by another feature" (Cafeo et al., 2016). The effects of such dependencies have been extensively studied in preprocessor-based implementations (Kästner et al., 2008). Ribeiro et al. (2011) and