Journal of Universal Computer Science, vol. 26, no. 3 (2020), 374-401
submitted: 19/12/18, accepted: 15/3/20, appeared: 28/3/20 *CC BY-ND 4.0*

Guidelines for Structuring Object-Oriented Product Configuration Models in Standard Configuration Software

Jeppe Bredahl Rasmussen
(Technical University of Denmark, Kgs. Lyngby, Denmark
jbrras@mek.dtu.dk)

Lars Hvam
(Technical University of Denmark, Kgs. Lyngby, Denmark
lahv@dtu.dk)

Katrin Kristjansdottir
(Technical University of Denmark, Kgs. Lyngby, Denmark
katkr@dtu.dk)

Niels Henrik Mortensen
(Technical University of Denmark, Kgs. Lyngby, Denmark
nhmo@mek.dtu.dk)

Abstract: Product configuration systems (PCSs) are increasingly being used in various industries to manage product knowledge and create the required specifications of customized products. Companies applying PCS face significant challenges in modelling, structuring and documenting the systems. Some of the main challenges related to PCSs are formalising product knowledge conceptually and structuring the product features. The modelling techniques predominantly used to visualise and structure PCSs are the Unified Modelling Language (UML) notations, Generic Bill of Materials (GBOM) and Product Variant Master (PVM), associated with class collaboration cards (CRC-cards). These methods are used to both analyse and model the products and create a basis for implementation to a PCS by using an object-oriented approach. However, the modelling techniques do not consider that most commercial PCSs are not fully object-oriented, but rather, they are expert systems with an inference engine and a knowledge base; therefore, the constructed product models require modifications before implementation in the configuration software. The consequences are that what is supposedly a feasible structure of the product model is not always appropriate for the implementation in standard PCS software. To address this challenge, this paper investigates the best practice in modelling and implementation techniques for PCSs in standard software and alternative structuring methods used in object-oriented software design. The paper proposes a method for a modular design of a PCS in not fully object-oriented standard PCS software using design patterns. The proposed method was tested in a case company that suffered from a poorly structured product model in a not fully object-oriented PCS. The results show that its maintainability can be improved by using design patterns in combination with an agile documentation approach.

Keywords: Product Configuration System, Conceptual modelling, Maintenance, Product modelling, Case study, Implementation framework
Category: D.2.2, H.1.m, J.6

1 Introduction

Poor or inappropriate product structure and knowledge representation in configuration projects are known reasons for failures in configuration project development, implementation and maintenance [Haug et al. 2019]; therefore, it is important to consider how to structure and implement product models in product configuration systems (PCSs). This article discusses the structuring and implementation of product models in commercially available, standard, non-object-oriented PCSs. PCSs are expert systems that support product customization by defining how predefined entities (physical or non-physical) and their properties (fixed or variable) can be combined [Hvam et al. 2008]. To build a PCS, a product model must be developed and implemented in the software system. Product models for PCS implementations contain rules for the construction of a product with its associated features and all its variants, so that knowledge can be expressed explicitly in a software system [Hvam et al. 2008].

Product models exist at different levels, as proposed in Duffy and Andreasen [Duffy and Andreasen 1995] (Figure 1). The real world represents knowledge about the product assortment and is often unstructured and not easily accessible. This real-world knowledge can be represented in a product model as structured representations that allow domain experts [1] to represent, analyse and communicate about this reality. An example of such a product model is the Product Variant Master (PVM) [Harlou 2008]. The information models in a configuration context can be Unified Modelling Language (UML) diagrams or similar formal Information Technology (IT) modelling techniques [Felfernig et al. 2001a, Hvam et al. 2008]. The information models are usually developed by knowledge engineers [2] and implemented in a computer model by either knowledge engineers or IT developers.

PCS modelling techniques are used to provide a basis for deciding what information to include and how to structure the information in a PCS to allow for future changes [Haug 2009]. This translation from unstructured information to IT implementation is reported as a reason for PCS project failures [Forza and Salvador 2002a, Haug et al. 2019]. One explanation for the failures was reported to be that development and maintenance are more time-consuming and challenging than initially expected [Forza and Salvador 2002, Hvam 2004, Jørgensen 2001]. Furthermore, studies have shown that, if the documentation of the PCS is not maintained, it can lead to companies having to restructure or abandon their PCSs [Forza and Salvador 2002b, Haug 2009]. This indicates a need for improved modelling and documentation approaches to develop and maintain PCS.

---

[1] Domain experts possess knowledge of the products and contribute to process analysis, product analysis and further development. Domain experts could be employees from product development or production [Hvam et al. 2008].

[2] Knowledge engineers translate the information obtained from domain experts to implement the knowledge into IT models.

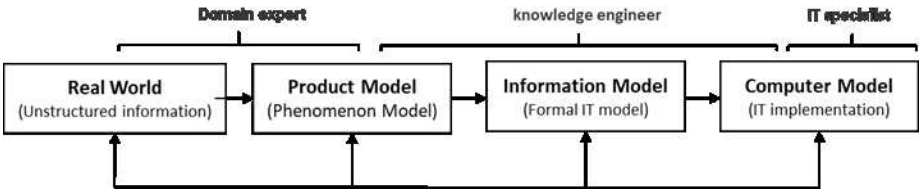


Figure 1: Different representation of product knowledge in PCS projects [Duffy and Andreasen 1995]

Configuration ontologies have been developed to provide a basis for communicating and documenting configuration knowledge in an easy-to-understand form describing concepts like attributes, attribute types, referencing to attributes and inheritance and its relation to the definition of a part-of hierarchy [Helo et al. 2010, Soininen et al. 1998, Yang et al. 2009].

Modelling techniques have been developed specifically to develop and maintain PCS models, which include 'Product Variant Masters' (PVM) associated with 'Class Collaboration Respensibility cards' (CRC-cards) [Haug 2010, Hvam et al. 2008, Hvam et al. 2003], UML diagrams [Felfernig et al. 2000a, 2001b], feature models [Kang et al. 1990] and Generic Bills of Materials (GBOMs) [Forza and Salvador 2007, Olsen and Saetre 1997, Tseng et al. 2005]. The mentioned methods assume that the concepts of object orientation can be used to model and implement a particular family of products in a class, defined as 'a description of a set of objects with the same structure, behavior, patterns and attributes' [Hvam et al. 2008]. Inheritance is a key concept in object-oriented modelling that allows for reuse and structuring of code; that is, a *subclass* can inherit properties from a superclass. Inheritance in object-oriented modelling makes it possible to define the class 'car' with certain properties (motor, colour, bodywork, chassis number, etc.), which can be inherited to a specific instance, such as an 'Opel' [Hvam et al. 2008].

Methods based on object-oriented design have proven to be successful in the development of PCS models. However, PCS models are commonly implemented in commercially available standard PCS software, representing expert systems with an inference engine and a knowledge base that do not always support fully object-oriented notations and implementations. In this paper, configuration systems that support some object-oriented features are referred to as not fully object-oriented, and software that does not support object orientation at all is referred to as non-object-oriented. When an expert system cannot handle object-oriented knowledge representations, the knowledge is non-hierarchical and cannot handle class–object relationships, inheritance or encapsulation [Hvam et al. 2008], making the available modelling techniques presented in the literature impractical to use for implementation of IT models in standard non-object-oriented software; this may result in redundant implementations. This creates a need for manually translating between the product models, based on object-oriented assumptions, and computer models, based on not fully object-oriented systems. Implementation and documentation techniques specifically for non-object-oriented standard PCSs are currently not addressed in the literature. This article aims to fill this gap. The consequence is that the product knowledge must be maintained in two different systems [Shafiee et al. 2015], and as

the product models grow bigger and become more complex, the time needed for documentation becomes a significant task in keeping the product models up to date [Hvam et al. 2005].

A class of software tools dedicated to supporting product modelling and documentation of PCS models was proposed by Haug [2010], but it was never used in the industry. To avoid redundant documentation, extraction of the information model and automatic generation of a PVM and CRC-cards have been proposed and successfully implemented [Shafiee et al. 2017]. However, the ability to extract documentation does not deal with redundant product model implementation. Other modelling approaches are emerging to mitigate modelling challenges, such as variant table representation [Haag 2017]. Based on the challenges experienced with implementation of product models and documentation of PCSs, this article proposes a framework for implementation of a modular design for PCSs that can be implemented in a standard non-object-oriented system along with an agile documentation approach.

Inspired by best practices from object-oriented programming principles and the aim of creating reliable, flexible and maintainable IT product models, the current article explores a way to structure product models for IT implementations in commercially available, not fully object-oriented PCSs by providing answers to the following research questions (RQs):

> ***RQ 1:*** *How should companies structure modular product models for PCSs that are not fully object-oriented?*
>
> ***RQ 2:*** *What could be the possible benefits for companies of using the proposed structuring principles on the usability and maintainability of a PCS that is not fully object-oriented?*

2 Research method

The current research adopts a four-phase approach from the Design Research Methodology (DRM) framework [Blessing and Chakrabarti 2009] (Figure 2). The *first phase* was identifying a worthwhile research objective, which was done from literature searches and observed challenges in companies working with PCSs. The research goal of this study was to improve the current methods to structure modular product models for not fully object-oriented PCSs by developing a framework for not fully object-oriented implementations.

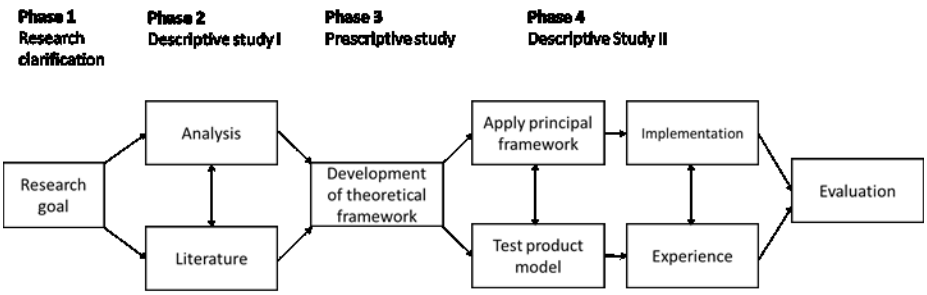


Figure 2: Research methodology for the development of a modular PCS structure, adapted from the design research methodology [Blessing and Chakrabarti 2009]

The *second phase* was an investigation of the current product modelling methods described in the literature on PCS. The identified methods were reviewed in parallel with the literature on object-oriented software design. The purpose was to learn the current best practices from IT professionals to solve configuration-modelling challenges and develop a theoretical framework for a modular PCS design adapted to not fully object-oriented systems. The proposed framework relies on the literature on product modelling [Harlou 2008, Hvam et al. 2008], combined with best practices in object-oriented software design [Coplien et al. 1998, Martin and Micah 2006] and agile documentation approaches [Staples 2004].

The *third phase* aimed to synthesise the knowledge gathered in the second phase to develop a framework for structuring modular product models for not fully object-oriented PCSs. In addition, the implications of software maintenance were identified and used as evaluation criteria in the fourth phase.

The *fourth phase* focussed on the implementation and validation of the proposed framework in a detailed case study in the company Altan.dk, allowing the theory to be tested in practice. Altan.dk was selected because of the illustrative product assortment and industry challenges experienced in structuring and maintaining its PCS. The case company has been using PCS since 2011 and has suffered from poorly structured PCS as the system has grown over the last 7 years. This has resulted in difficulties in maintaining and further developing the system. During the case study, the company's PCS was restructured based on the proposed theoretical framework developed by following the adapted DRM. The restructuring of the PCS was done over 3 months by a configuration engineer who was responsible for the PCS. This study was performed as a single case study because of the possibility of studying a phenomenon in its natural setting and allowing 'how' and 'what' questions to be answered [Karlsson 2016, Meredith 1998]. The single case study design allows the phenomenon of the structures' influence on PCS features and maintainability to be studied in detail, but this has the downside of reduced generalizability [Karlsson 2016]. This drawback can be mitigated by repeating the results in other case companies [Eisenhardt 1991]; however, this is not always possible because of resource constraints. Another way to improve reliability is using triangulation with data sources [Karlsson 2016]. In this case, data triangulation was used for collecting performance measures from different

stakeholders; here, the end users of the system (salespeople), the backend users of the system (configuration engineers) and observations from researchers were considered to obtain a full picture of the implications of the proposed framework in the case company. This was possible because the company maintained both the unstructured and newly structured PCS over a certain period, allowing for a direct comparison. The evaluation was done by the researcher in collaboration with the system users, represented by the responsible configuration engineer, and daily users of the PCS, represented by salespeople. The testing and validation for the case study were performed by investigating the benefits from the modular PCS design compared with the old PCS, which was validated by 12 PCS users in a workshop and follow-up interviews of a few selected users (Appendix). The users validating the PCS were end users, salespeople who sought to evaluate the new features and give some indications of the reductions in resource consumption. In addition, interviews were performed with an experienced configuration engineer with prior experience working with the same PCS to detail the knowledge of the strengths and weaknesses of the ad hoc structured PCS. An overview of the consulted stakeholders in relation to the data requirements for the current research can be seen in Table 1. The interviews were semi-structured because of unclear terminology in the area of configuration at the company; therefore, a need to clarify meaning as the interview progressed was identified.

Required data	**Data source**
Information on the PCS structure	Interviews with two configuration engineers Hands-on investigation of PCS done by the researcher Literature study to evaluate alternatives
Time spent introducing new product variants into the system model	Interviews with two configuration engineers
Time required to reconfigure between product variants from different product platforms	Interviews with 12 end users (salespeople) before and after a workshop presenting the new PCS
Documentation usefulness	Interviews with two configuration engineers

Table 1: Required data and sources used in the case study

3 Theoretical background

The literature review aims to identify theories for product modelling of PCSs. Section 3.1 gives an overview of the current literature on PCS modelling and its applications. Section 3.2 establishes a link between software structure and maintenance needs, as well as identifying the criteria for the evaluation of software design based on noted

challenges. Finally, Section 3.3 gives a brief introduction to the current methods used in software design to improve software quality; it serves as a basis for the development of the structuring method proposed in this article.

3.1 Product modelling for PCS projects in not fully object-oriented expert systems

In configuration research, the representation of domain knowledge is agreed to be one of the main challenges [Mailharro 1998], and most studies on the knowledge representation of PCSs address this topic from the knowledge engineering perspective [Zhang 2014]. Examples of knowledge engineering approaches to knowledge representation are the works of Felfernig et al. [Felfernig et al. 2001] , who describe how to model PCS by means of UML, and Falkner et al. [Falkner et al. 2011], who describe how to apply Artificial Intelligence (AI) methods when developing custom software. Methods to support domain engineers in representing product models for PCS have been proposed in various forms, including object-oriented approaches like UML [Felfernig et al. 2000a, Felfernig et al. 2000b], GBOM relationships [Forza and Salvador 2007], PVM and CRC-cards [Hvam et al. 2008] and product family classification trees (PFCTs) [O'Donnell et al. 1996, Yu and MacCallum 1995]. These methods have proven successful in the development of PCS. However, there has not been much follow up in the literature when it comes to evaluating the maintainability of the systems after they are implemented; they all lack specific implementation guidelines or use an UML-based representation that assumes object-oriented implementations [Hvam et al. 2008]. Most commercially available PCS software comprises non-object-oriented expert systems, meaning that the classes, attributes and methods (rules) have no hierarchical structure or do not follow the basic object-oriented principles [Hvam et al. 2008]. Such not fully object-oriented expert systems are referred to as 'non-object-oriented standard PCSs' throughout the paper. This makes the product structure depend on the folder structure and requires modifications to the product model representation developed using object-oriented approaches [Hvam et al. 2008].

PCSs can also be developed in applications not meant to support object-oriented features, for example, Excel [Wielinga and Schreiber 1997] or BOM configurations in Enterprise Resource Planning systems [Hvam et al. 2008]. Furthermore, the rapid development of commercially available configuration software has allowed domain experts to handle more of the product-modelling task [Haug et al. 2010]. In many cases, 'product domain experts' are not formally trained programmers with extensive knowledge of UML-mapping and object-oriented models; their lack of knowledge results in suboptimal product structures. Investigations of PCSs have shown that PCS applications are often developed in an ad hoc fashion, lacking formal definitions of logical relationships and hard coding, which produces severe maintenance overheads [Boucher et al. 2012]. Another reason that PCS maintenance and development is a challenge is the simultaneous development of the PCS and product model, which adds to the differences between the documentation and software [Hvam et al. 2008]; this leads to redundancy in the model and documentation [Haug 2010], and addressing this has been identified as a laborious and time-consuming task [Hvam et al. 2005].

3.2 Implications of structure for software maintenance

The challenges experienced in PCS development and maintenance are not unique to PCS; rather, they are a challenge in software development and maintenance in general. Most software projects involve understanding legacy code [Sharon 1996], and most of the time is spent identifying the errors resulting from unexpected effects rather than the time needed to correct them [Shalloway and Trott 2002]. The challenges in software stem from the following: (1) poor system design and structure, (2) excessive system complexity, (3) limited system flexibility, (4) limited or non-existing documentation, (5) inadequate project and process management, (6) inadequate change and version management, (7) inadequate release management and (8) inadequate maintenance tools [Sharon 1996]. All the mentioned areas are relevant to PCSs, but this paper focusses on addressing the first four challenges. The remaining challenges are not addressed in this article because they are not directly affected by product structure decisions.

3.3 Approaches to object-oriented software design

One way of approaching the challenges of software maintenance is by addressing the first four challenges outlined above, which can be restated as follows: (1) poor system design, (2) excessive system complexity, (3) limited system flexibility and (4) limited documentation. Various approaches have been proposed by computer scientists to mitigate these challenges [Dijkstra 1982, Freeman 2015, Martin 2002, McConnell 2004]. One approach is the concept of 'separation of concerns', which is a design principle used to aid modular programming by dividing problem spaces into distinct elements where no elements share the responsibility of others [Dijkstra 1982]. Numerous methods have been developed to aid the design of software that upholds the separation of concerns [Larman 2004, Martin and Micah 2006, Thomas and Wesley 1999].

Three dominating views of recognised object-oriented software design practices are as follows: general, responsibility, assignment, software, patterns (GRASP) [Larman 2004]; 'don't repeat yourself' (DRY) [Thomas and Wesley 1999]; and single responsibility, open-closed, Liskov substitution, dependency inversion and interface segregation (SOLID) [Martin and Micah 2006]. The GRASP guidelines for object-oriented design lay out how to assign responsibilities to classes and objects to develop software with high cohesion and low coupling [Larman 2004]. The DRY principle aims at the reduction of repetitions by ensuring that every piece of system knowledge has one authoritative, unambiguous representation, reducing the chance of errors and minimising inconsistencies [Thomas and Wesley 1999]. Martin and Micah [Martin and Micah 2006] list five principles for agile software design, known as SOLID; as indicated above, these principles are as follows: (1) single responsibility, (2) open-closed, (3) Liskov substitution, (4) dependency inversion and (5) interface segregation. All three views of object-oriented design revolve around the same topic—simplifying the code by controlling interfaces, module sizes and interdependencies to make the source code maintainable and flexible. The three frameworks all primarily focus on isolated concepts when it comes to the separation of concerns, such as couplings, cohesion, dependencies and abstractions. To reduce the level of knowledge needed to understand and implement the concepts the concept

of design patterns was introduced as a 'solution to a problem in a context' [Gamma et al. 2002].

Design patterns have proven to be reusable structuring principles between classes that can solve specific software design challenges. Some pattern examples are the facade pattern, which aims to introduce a higher level interface that makes a system easier to use by others without the need for an overview of the entire system; the adapter pattern, which is used to create new interfaces to connect with incompatible interfaces; and the bridge pattern, which strives to decouple abstraction from implementation [Gamma et al. 2002]. Numerous other design patterns exist, and more evolve as software design problems are solved and the same solution repeatedly emerges and is eventually consolidated into a design pattern. Shalloway and Trott [Shalloway and Trott 2002] state that the use of design patterns in the specification phase enables a programmer to abstract and implement code that is more flexible and open to changes. Coplien et al. [Coplien et al. 1998] suggest a scope, commonality and variability (SCV) analysis as a starting point for the development of software family lines. Here, S is the product line driven by the market, C comprises the characteristics common to all products and V represents the variation among the products.

The current paper seeks to contribute to the literature on PCS knowledge representation by applying design patterns inspired by software engineering to provide standard design patterns for knowledge representation of PCSs. This will enable domain engineers to design and take better care of the knowledge base. Furthermore, the present article suggests a shift from code-level documentation to design rationale documentation as used in agile documentation [Staples 2004].

4 The proposed approach for the development of a modular-structured PCS

This section presents a three-step framework to structure modular product models for PCSs based on both the literature relating to product modelling for PCS projects and approaches to object-oriented software design. The first step aims to analyse the PCS requirements and decide on a structure based on the strategic goals for the business and the product assortment by using a commonality and variability analysis. The second step is structuring the PCS according to the best practice of object-oriented design using design patterns. Finally, the third step is to create 'light but sufficient' documentation to provide an overview and understanding of the interfaces and dynamics of the model. Figure 3 gives an overview of the different steps of the framework, which are described in more detail in the next sections.

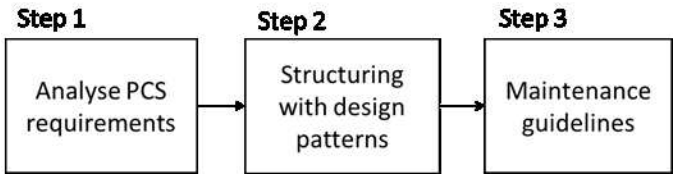


Figure 3: Proposed framework for the development of a modular PCS

4.1 Step 1: Analysis of the PCS structural requirements using an SCV analysis

The development of IT systems is costly and time consuming, and a proper analysis before making design decisions for product lines has ramifications for cost and quality [Ramachandran and Allen 2005]. The SCV framework [Coplien et al. 1998] has been used to identify the scope of the product model and what parts of it are changing. In a PCS context, the scope (S) can be viewed as the number of product platforms to be included in the PCS. If the S in a PCS handles numerous (different) product families, the structure can be complex and suffer from too little commonality. If the S is too small, there are a lot of commonalities, but there may not be enough to justify the need for structuring efforts. Guidelines for what to look for when identifying the scope in a PCS are outlined below.

1. *How many product platforms should be handled by a single product model, or would it be beneficial to split the product model into different, simpler models?*

The commonality (C) is the main source of reusable code and interfaces, and the variability (V) is the necessary differentiation in the programme or product. To identify commonality and variability, the key questions to ask are as follows:

2. *What parts of the product structure are **not** likely to change significantly within the next 3–5 years?*
3. *What parts of the product structure are **likely** to change significantly within the next 3–5 years?*

The first question finds the right coverage of the PCS, addressing system complexity, and the second and third questions aid the designer in performing a commonality and variability analysis to identify what part of the model should be modelled as abstractions and implementations in Step 2. The purpose of Step 1 is to address the structural challenges of poor system design, poor structure and model flexibility. An example of the practical use of Step 1 is described in Section 5.2.1.

4.2 Step 2: Structuring of a PCS with design patterns

The product assortments change over time in companies, and so does the need for knowledge representation of the valid product combinations in the PCS. Some parts of the PCS logic will be relatively stable over time, while others will need to be updated frequently. Design patterns are a way of describing general solutions to a design problem that recurs repeatedly in projects [Khwaja and Alshayeb 2013]. Many patterns exist that could be relevant to the PCS. In this case, the bridge pattern is chosen as the base pattern to be adapted to standard PCS modelling because of its properties of dividing abstraction and implementation [Shalloway and Trott 2002] and ability to isolate the effects of changes [Freeman 2015]. The purpose of the bridge pattern (Figure 4) is decoupling an abstraction from implementation to create well-defined interfaces. The abstraction defines the interface for the objects being implemented, and the implementor defines the interface for the specific implementation classes [Shalloway and Trott 2002]. In other words, the abstraction class contains few details that will change, and the implementor class contains the details that may change in the future. The stability of the abstraction class ensures stable interfaces for the actual implementation class.

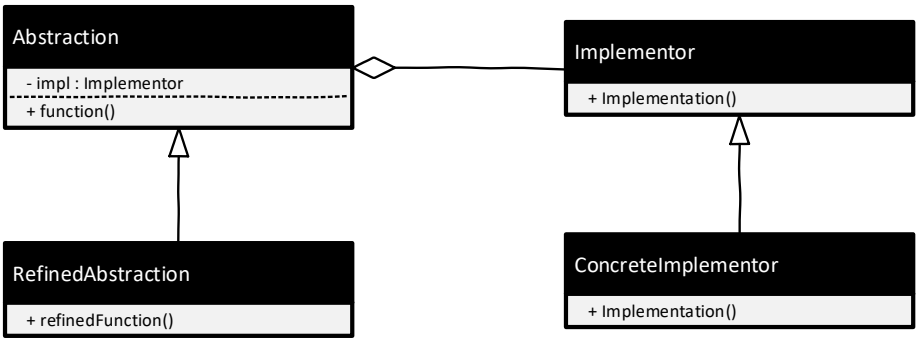


Figure 4: Bridge pattern adapted from Shalloway and Trott [Shalloway and Trott 2002], *denoted in* Unified Modelling Language *(UML) notation*

4.2.1 Modifying the bridge pattern to a non-object-oriented standard PCS

The purpose of the bridge pattern is to decouple an abstraction from implementation [Shalloway and Trott 2002]. Figure 5 depicts the bridge pattern adapted to PCS implementations with the UML notation. In PCS, an abstract class can be created to handle attributes that describe a product architecture from an abstract view, such as functional elements and abstract variations. This corresponds to the identified commonalities in Step 1. The abstract class can be further refined to contain abstractions of sub-parts. The implementor class comprises variants and rules related to product platforms at a higher level and includes the implementation of specific instances of a platform derived from abstract attributes corresponding to the variability identified in Step 1. By dividing the structure into abstraction and implementation, the methods relevant to the product architecture at a general level are

given in the abstract class, while the rules related to different product platform variants are found in the implementor classes. This allows for increased extensibility of implementations [Shalloway and Trott 2002]. The benefits of this structure are that product changes related to the product architecture and product variant implementations are situated in different objects. The clear interface between the objects helps the modeller identify where to implement changes without the need to introduce corresponding changes in other parts of the model [Freeman 2015]. Note that the generalization arrows have been removed in Figure 5 for the PCS-specific bridge pattern for non-object-oriented standard PCS because inheritance does not exist in such a system. By definition, the rules and methods work globally. The aggregation arrow remains because it describes that the abstraction class shares a reference to the implementor class. The purpose is to address the structural challenges of poor system design, poor structure and model flexibility.

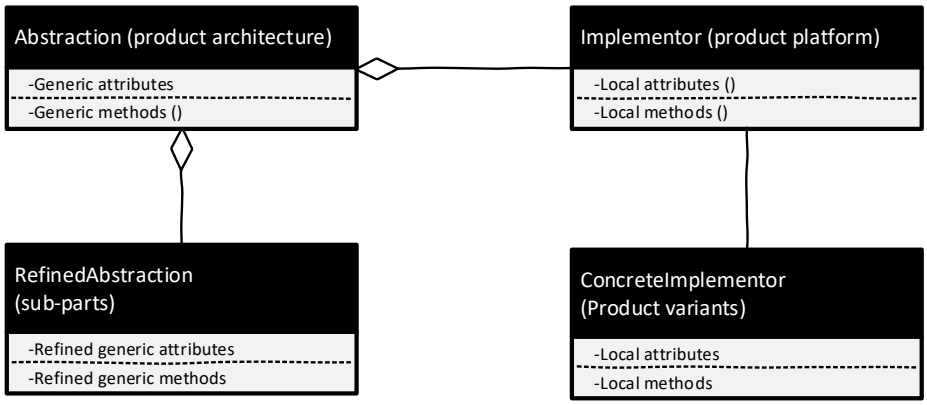


Figure 5: Bridge pattern modified for PCS implementations in UML notation

4.2.2 Example of the bridge pattern for PCS

The purpose of the bridge pattern is to decouple an abstraction from implementation [Shalloway and Trott 2002]. In Figure 6, an example of a PVM describing a simple car family is presented from Harlou [Harlou 2008]. In Figure 7, a class diagram is drawn in terms of the bridge pattern. The car family is described as an abstraction class with the car model containing generic attributes, but these can include subclasses describing subparts that are used across all product variants, for example, the engine, windshield, door and wheel parts. The descriptions used in the abstract classes should be as abstract and generic as possible to secure the flexibility of the model. The implementor class refers to the abstract class as the interface, and it contains classes with rules and attributes concerning specific product variants. This concept is close to the PVM concept, incorporating a generic architecture (sub-part structure) and family-specific sub-types (specialization). The difference is that it is not a real object-oriented representation, where different sub-classes can have different sub-structures beneath them. Consequently, the links between the generic architecture and family-specific sub-types must be specified, implemented and

maintained manually. The benefits of the bridge pattern structure are that, when changes occur, the rules related to the product architecture at a general level and product variant implementations at a model-specific level are located in different object classes. Therefore, the rules are easy to find and modify manually.

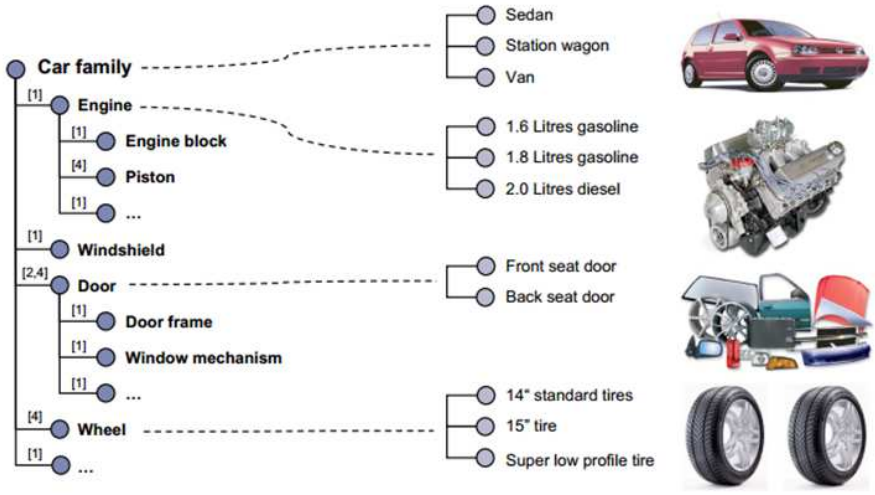


Figure 6: PVM notation of a simple car adopted from Harlou [Harlou 2008]

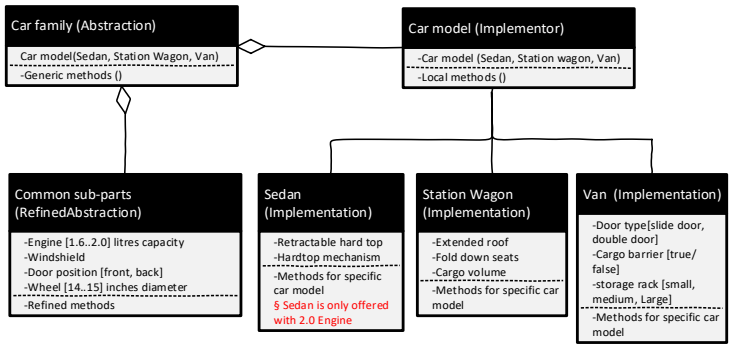


Figure 7: Class diagram drawn in relation to the bridge pattern in UML notation

4.3 Step 3: 'Light but sufficient' maintenance guidelines

The idea of step 3 is to document the design rationale over details. By documenting the design pattern and model dynamics used in the system instead of making comprehensive lists of all the available attributes and rules in the PCS, it will be easier to make changes that comply with the original design rationale and purpose [Selic 2009]. The division between abstraction and implementation classes is shown to increase the system overview and clarify the purpose of different classes [Staples

2004], and this approach has proven to make the code more reusable and result in lower maintenance costs in software development [McConnell 2004]. The same is likely true for PCS modelling. By documenting the guidelines, not the details, the correct documentation is always available in the system, and the need for redundant documentation is reduced to the need for documentation of design pattern principles and examples of the most common product model modifications. The examples can be generated by saving screenshots and writing a corresponding text describing the changes. A practical way to guide the use of in-system documentation can simply be adding detailed and consistent naming to the classes, such as _AbstractionClass or _ImplementationClass. A good test to make sure the documentation is relevant is to ask the following: 'What would you want to know if you joined the team tomorrow?' [Wiseman 2007]. The purpose of Step 3 is to address the challenges of limited model flexibility and limited documentation.

5 Case study: Configuration of balconies

The case company introduced in the study is a medium-sized Danish company; since 2009, the company has used a PCS to generate quotation letters for system deliveries of balconies. The PCS was initially developed by a consultant trained in the modelling methods proposed by Hvam et al. [Hvam et al. 2008] in the Configit Model® to handle a single product platform, but as the business evolved, new major updates, including new product platforms, were added to the existing model a few years apart by different modellers. As the product model grew in complexity over time, maintaining an in-depth understanding of the model became a challenge, and no one understood all the facets of the model and its interactions. This resulted in major problems when products and product features were added, removed or changed in the configuration model. In addition, some structural problems affected the PCS's ability to handle reconfigurations, meaning users had to configure complicated products from scratch. A decision to redesign and improve the product model was agreed on to improve the current situation and avoid future problems. The focus of the project was to build a flexible and maintainable model. To illustrate the principle, a subset of the model related to balcony bottom plates is used as an example. First, a description of the situation before intervention in the PCS of the company was elaborated on. Second, the structural redesign was made by following the three-step framework proposed in this article. Third, each of the proposed steps was validated for usefulness by conducting interviews with a configuration expert who had worked for the company and had experience with maintenance of its configuration models.

5.1 Case example: A poorly structured configuration model of balcony bottom plates

The configuration model contains three different kinds of balcony bottom plates that represent three kinds of product platforms named —'aluminium', 'steel' and 'plate'. The bottom plates are designed with three different principles, allowing different dimensions to be used. The steel plates can be any width or depth, the aluminium offers free width and a fixed number of depths and the plate comes in predetermined combinations of fixed depth and fixed width (Table 2). Since the PCS used at the

company does not support object-oriented features, such as inheritance, specific instances of each combination of width/depth needed to be generated manually for every family-specific sub-type to define the solution space and relate to the rules and constraints. The new product families were added as a parallel design principle in the product model without relating in depth to the existing ones. The result was multiple implementations of length/width combinations based on different entries in the system that randomly related to other width/depth product combinations without any clear pattern.

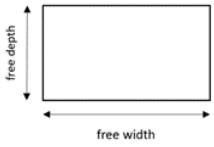
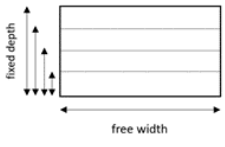
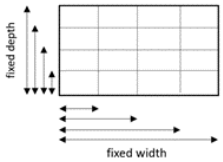
	Steel	**Aluminium**	**Plate**
Bottom plate dimen-sions	Free width, free depth	Free width, fixed depth	Fixed width, fixed depth
Drawing			

Table 2: Three different types of bottom plates representing different product families and their valid selections of width/depth

A simplified class diagram showing an example of the content from poorly structured 'bottom plate' classes is presented in Figure 8, and a screenshot from the PCS is shown in Figure 9. Note that the representations of rules, attributes and constraints are written in natural language, since syntax and implementations will vary from system to system.

The diagram reveals that a single class is responsible for several different bottom-plate-related variables, including a comprehensive set of rules for selecting and deselecting relevant product numbers, dimensions, colour choices and product-specific rules determining the legal combinations with other parts of the model. The class contains no abstraction, only implementations, resulting in a model that is difficult to comprehend. The documentation present in the company only reflected the first implementation of the product model and included a long list of outdated rules and relations. The consequence of the structural choice of the ad hoc product model was that it was difficult to find and change relevant parts in the model because of excessive information and unclear dynamically interconnected parts, making it difficult to implement changes without unforeseen consequences. The challenges resulted in maintenance difficulties and required extensive modelling and product knowledge competencies to make small changes in the model.

The same ad hoc structuring principles with unclear structuring had been used for the rest of the configuration model (i.e. railings, hand railings, floors, doors, accessories, etc.). The result was a complicated dynamic model where rule interactions did not reflect the structure of the configuration model. The most tangible consequence of the structure was a reconfiguration problem experienced by the users,

as the system did not allow changes from one kind of balcony to another without serious problems. It was not possible to change from one kind of bottom plate to another without starting over. In the past, some initiatives were taken to fix the problem, but all attempts ended up being rolled back because of unforeseen consequences relating to the restriction of product variations that were not supposed to be restricted.

Figure 8: Bottom plate 'class' representing different choices of bottom plates, widths and depths. Represented in UML notation

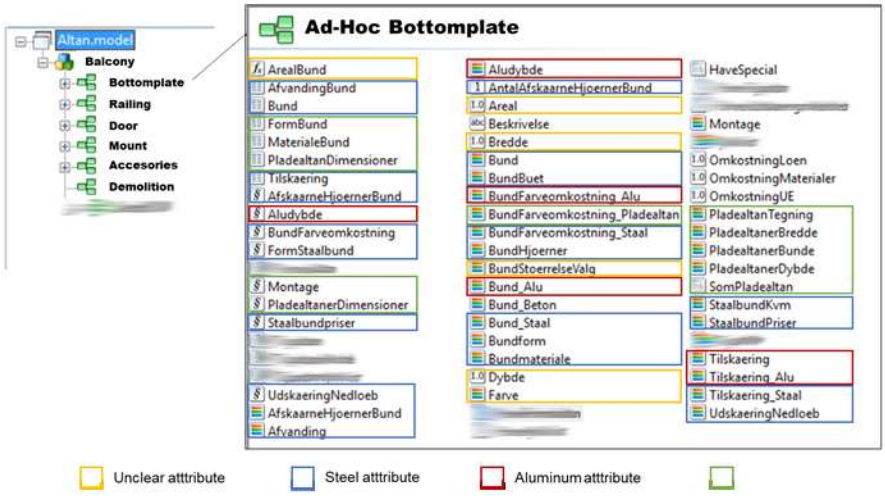


Figure 9: Screenshot of the in-system ad-hoc bottom plate group

5.2 A redesigned configuration model of balcony bottom plates

To address the problem at the case company, the proposed framework was tested and validated. The next sections describe the results from testing the individual steps of the framework.

5.2.1 Step 1: Analysing PCS structural requirements by SCV analysis

The first step was to analyse the requirements of the PCS through the SCV analysis. The scope analysis was based on the SCV framework. Question 1 (Section 4.1) initiated a discussion about the possibility of splitting the PCS model into different models representing every product platform individually to simplify the configuration model. However, changes in the scope of the PCS were dismissed by the company because of synergy effects between product platforms, many shared configuration elements and convenience for the salespeople, who required a single model to configure all the variants. The commonality and variability analyses were performed with domain engineers to identify which parts of the model would change frequently and which parts would not (questions 2 and 3; Section 4.1). The product offerings and architecture were not going to change significantly over the next couple of years, but the product components, variant possibilities and rules would be subject to many updates in the same period. Consequently, the abstraction class would consist of the most generic balcony description possible, such as selection of the product platform, dimensions, number of components and so forth, and the implementor classes would contain the specific balcony variations, such as steel, aluminium and plate, and their legal variants, specific component choices and product-specific rules at a sub-type level. It was agreed on by the users and responsible configuration employees that the new product structure would be a better fit for the business.

5.2.2 Step 2: Structuring of a PCS with design patterns

To distinguish between abstractions and implementations, a redesign of the model was performed using the bridge pattern modified to the PCS (Section 4.2). In Step 2, abstract attributes of the bottom plate class were identified as common elements representing a generic architecture. An example relates to the dimensions and balcony model; no matter what kind of bottom plate is chosen for a given balcony, the dimensions must always be specified. The same is true for the choice of the balcony model. This indicates that the dimension and bottom plate types could be considered time-stable abstract elements belonging to an abstraction class. The specific balcony variants would be subject to variation, as identified in Step 1. The variations could then be realised in an implementor class representing product-specific sub-type knowledge, such as the possible dimensions that depend on product series, colour variations and legal combinations with other parts of the balcony model. These elements were grouped in logical modules according to the product platforms. The new structure based on the bridge pattern that was designed for the new system can be seen in a UML model representation in Figure 10, and a screenshot from the actual implementation in the configuration software can be seen in Figure 11. The new solution includes an abstraction class called ‘start’, containing all abstract generic attributes in the model, such as *Abstract_width*, *Abstract_depth* and *Balcony model*. These names are chosen to be consistent with the terminology used to describe the

bridge pattern and illustrate generic attributes that specific implementations must relate to as the interface. In addition, a group for every kind of bottom plate relating to different *Balcony model* sub-types was generated, comprising the rules for the corresponding variant and checking against the abstract interface (Figure 10). The steel plate was free in any dimension, so the implementation could obtain depth and width definitions directly from the abstract class. The aluminium plate allowed for all widths and restricted depths, making it possible to refer directly to *Abstract_width* but necessary to create new attributes containing '*Alu_depth*' specifically for that product platform. If *Balcony model* has the value '*Alu*', a rule is made to restrict that *Abstract_depth* is only allowed to take values defined in *Alu_depth*, and the correct *Alu_depth* is selected. The plate balcony is restricted in both width and depth, and it is designed with the same principle, needing rules to check against both *Abstract_depth* and *Abstract_width*. In this way, all product platforms refer to the same generic definition of depth and width, and the rules related to the different plates are placed in 'implementation classes'. This approach makes it easier to find relevant rules when a product changes because the rules are encapsulated and always relate to the abstract attributes of the dimensions and choice of the bottom plate. The product variants now check the rules against the stable *Abstract_width* and *Abstract_depth* instead of between many definitions of width and depth that depend on the implementations of specific bottom plates and other related rules. The use of the bridge pattern in the modular PCS allowed the company to solve the reconfiguration between the product platforms and implement changes that the company had failed to make in the ad hoc model.

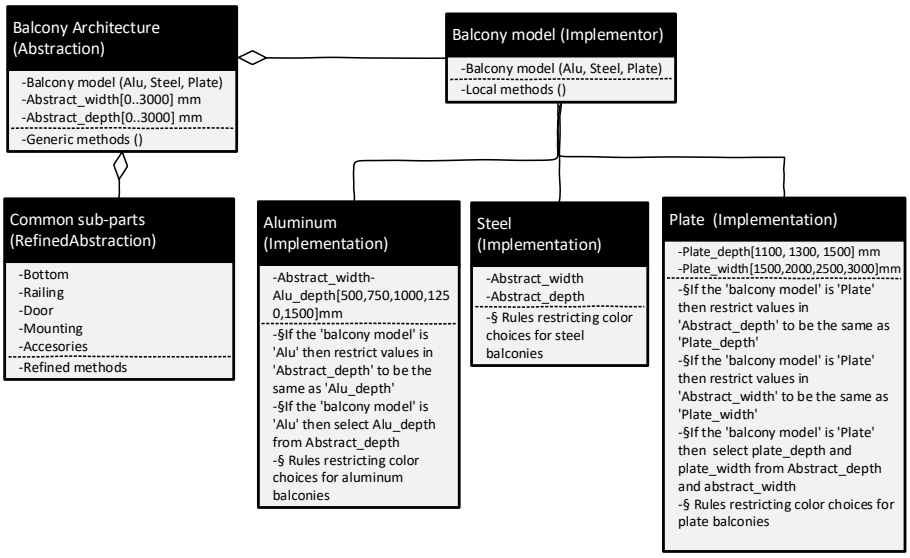


Figure 10: Bottom plate groups related to Master_width and Master_depth with local rules in UML notation

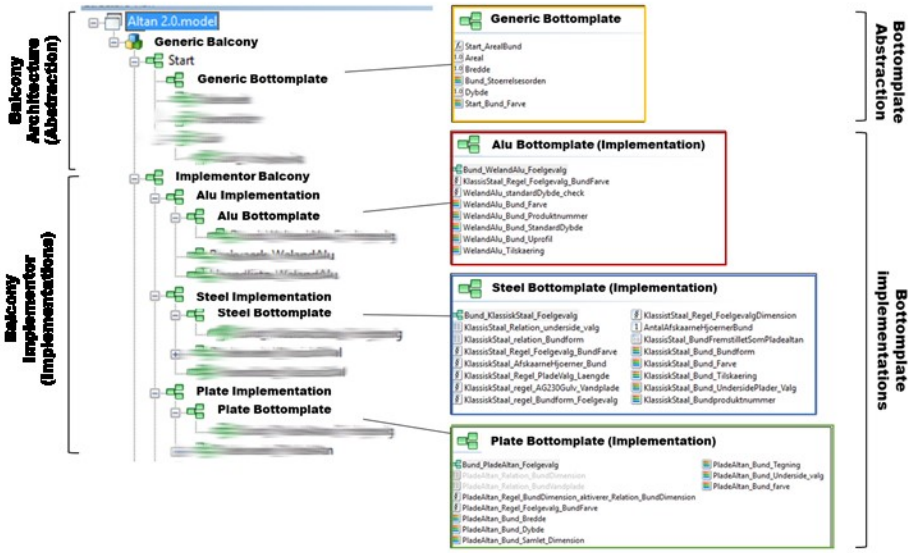


Figure 11: Actual implementation of the bridge pattern with an abstraction class as the interface for product-specific variables and rules.

5.2.3 Step 3: 'Light but sufficient' maintenance guidelines

The third step was to improve the documentation of the PCS model by documenting implementation guidelines. The case company had detailed documentation of all the rules and product combinations of the first version of the system; however, they had never used it, and instead, based implementations on in-system data without the use of guidelines. Since the documentation was never used, the company decided to document details on an in-system basis along with implementation guidelines. Therefore, 'light but sufficient' documentation of the design pattern and guidelines for common update tasks was created by documenting the design rationale—not the details. Screenshots of the product structure, design pattern overview and guidelines on where to implement certain common product changes were created as documentation. The documentation included guidelines on how to perform the most common tasks, such as changes in architecture (requires changes in both the abstraction and implementation classes) and changes at a product variant level (only requiring updates in the implementation classes) to avoid violations of the selected structure. The consequence of the approach was that correct documentation could always be found in the system, and the need for redundant documentation was reduced because of a clear and understandable structure. The focus on the design rationale over the details was presented to a configuration engineer who had experience working in the PCS, and the engineer found this new system to be more useful than was the original documentation that had described each and every rule in the system externally. The engineer further mentioned that the approach was

comparable to how 'model documentation would be handled in his new job in a big company', stating that 'the IT architects were usually lacking behind on the documentation and rarely corresponding to the data in-system'.

5.2.4 Benefits of the restructured modular PCS-based method

The improvements in the modular configuration model were, first, an improved ability to identify what parts of the model would be relevant to investigate and change for the desired outcome. Second, some annoyances in the model were corrected, allowing for reconfiguration; third, the maintenance and product updates were made less dependent on the original developer or configuration specialists. The number of rules and attributes in the ad hoc PCS structure and the new PCS structure were practically identical, indicating a product model of a similar scope. However, the claim is that the relevant rules, attributes and interactions will be much easier to identify and modify because of the improved structural overview.

5.3 Evaluation of the modular PCS case implementation

The company decided to develop and support both the ad hoc model and the restructured modular model for a certain period to secure backward compatibility and test the ad hoc system's and new structure's performance side by side. To test the differences between the ad hoc PCS performance and the new modular PCS, updates to the product assortment were made in both systems by the responsible configuration engineer. At that point, both PCSs contained approximately the same number of rules, relations and attributes. An update on the product variant level by adding new standard dimensions for the plate balconies was a longstanding wish of the company. In the ad hoc model, the company never succeeded in implementing the change because of unforeseen consequences from coupled relations. The requested change could be made in the modular PCS in 15 minutes because of a better overview and understanding of what the rules referred to. Furthermore, the configuration engineer added a new product platform to the ad hoc and new product models, and the time used was registered. Adding a new product to the ad hoc model took four working days on top of a half year of experience working with the model and product assortment. The product platform was added to the new model in a single working day and had fewer bugs. The PCS restructuring project took approximately 4 days of work (not including the time to develop the theoretical framework). In addition, persistent annoyances for the users generated by the ad hoc structured PCS were fixed, such as the reconfiguration problem, now allowing the users to revise an offer from one product type to another without starting over. This feature was tested in a workshop comparing the two systems side by side, and by itself, the new system has the potential to save 3–4 hours for every salesperson each time the customer requests a revised price based on another product platform. The company makes over 1,000 offers yearly, with multiple revisions in approximately half of them, so the new capability amounts to substantial reductions in time expenditures. The company estimated the reduced time expenditure to represent approximately half a full-time salesperson's working hours. This was a direct consequence of linking implementations to abstractions and using the bridge pattern, allowing for fast reconfigurations. The experienced benefits reported in the case include better

understandability of the product model, increased modularity, solving the reconfiguration and easier expandability of the product assortment. This is in alignment with research showing that the strength of the bridge pattern is mostly related to expandability, understandability and modularity [Khomh et al. 2009].

An overview of some measurable benefits such as reduced time expenditure for maintenance and increased flexibility for users can be seen in Table 3. In addition to these benefits, more subtle benefits were noted, such as reduced dependence on single employees, increased ownership of the product model, and consequently, reduced risk for the company.

	Ad hoc structured PCS	Modular PCS	Difference
PCS maintainability			
Addition of new standard dimensions for plate balconies	Never succeeded without unforeseen implications, multiple attempts performed	15 minutes	Now possible
Time needed to introduce a new product platform to the system with new standard dimensions by a configuration engineer	**4 working days** (and half a year of experience working with the model)	**1 working day** (and half a year of experience working with the model)	3–5 working days
PCS flexibility for users			
Reconfiguration between platforms	Time consumption 3–4 hours, depending on the configuration scope by starting from scratch Introduction of unforeseen configuration mistakes likely	5–10 minutes* Introduction of mistakes unlikely	3–4 hours

**Tested and quantified by salespeople in the company by reconfiguring problems.*

Table 5: Comparison of the ad hoc structured PCS and modular PCS

6 Discussion

The following identified challenges of software design were used as the evaluation criteria: (1) poor system design, (2) excessive complexity, (3) limited system flexibility and (4) limited documentation. The challenges in software development are complex, and it is difficult to determine when one structure is better than another. Evidence for the benefits of design patterns in software development is inconclusive and depends on the pattern used, context and software developer assessment [Khomh et al. 2009, Khomh and Gueheneuce 2008]. The bridge pattern has been found to have an overall positive impact on code quality [Abul Khaer 2007], especially when it comes to expandability, understandability and modularity [Khomh et al. 2009]. This is in alignment with the benefits reported in the case with better understandability of

the product model, increased modularity, solving the reconfiguration problem between products and easier expandability of the product assortment.

The proposed framework in the current study should ideally contribute to improvements in the area of modelling product models for PCS computer model implementations. The method proposed in the present paper was inspired by software development, and we found that the method had similar benefits to the approaches experienced in other software contexts [Coplien et al. 1998, Shalloway and Trott 2002].

Poor system design is often a result of ad hoc structuring, creating a high risk of poor structural design. By using design patterns and considering the SCV framework [Coplien et al. 1998] to introduce abstraction classes in the design phase of PCS, the structural quality and flexibility should improve when it comes to software product lines [Ramachandran and Allen 2005]; according to what was expressed in the interviews, these features improved in the presented case study.

Excessive complexity in the PCS arguably stems from the design of the products to be implemented in the system. If the scope of the PCS can be redesigned to contain more similar variants, the complexity can be reduced [Falkner and Haselböck 2009]. However, it may not be possible to change the product design to reduce the complexity of the PCS, as occurred in this case study; therefore, the PCS must be adapted to fit the product.

In the case study, limited system flexibility and maintainability were improved by clear differentiation between abstractions and implementations, as validated with expert interviews. This indicates that, if the principles are followed, it should result in a loosely coupled model and provide maintenance guidelines to update the model in accordance with the known benefits of the bridge pattern [Shalloway and Trott 2002].

The most prevalent documentation approach in PCS projects presented in the literature is documentation of everything relating to the product configuration model, which is time consuming, bothersome and results in mistakes [Hvam 2004]. However, the agile approach to software documentation, namely, documenting a high-level design rationale, is considered useful for the implementation of finer details. The agile documentation practice was preferred by the experts interviewed in the case study. The structuring approach and resulting improved flexibility of the product model may enable domain engineers to handle a greater part of the model and possibly enable outsourcing of product modelling responsibilities in some companies with relatively stable product assortments. A well-structured PCS with 'light but sufficient' documentation may also help companies reduce the risk of experiencing a situation with an overcomplicated PCS, which can be difficult to oversee and maintain. Furthermore, the approach may speed up new development in non-object-oriented standard PCSs by providing a clear structure during development.

7 Conclusion

This paper analysed the current modelling practices of standard PCSs and developed a framework based on best practices from object-oriented design for implementation in non-object-oriented standard PCS software. The framework was tested in a case company, and the results provided an improved product structure and a solution to longstanding problems with configuration maintainability and usability. In

conclusion, the contributions of the current paper are a case study and a three-step framework for the structuring of modular product models for standard PCS software implementations. The proposed framework was tested in a case company both for validation and to assess its usefulness.

As mentioned above, the proposed framework to build modular PCSs consists of three steps, which are as follows: (1) analysing the PCS structural requirements through an SCV analysis [Coplien et al. 1998, Ramachandran and Allen 2005], (2) structuring of a PCS with design patterns [Gamma et al. 2002, Martin and Micah 2006] and (3) modifying the bridge pattern to non-object-oriented standard PCS software. The framework was tested in a case company by redesigning a poorly structured PCS to aid in the creation of an improved structure of a PCS in non-object-oriented standard PCS software. Comparison of an ad hoc structured PCS and a modular PCS based on the guidelines in the proposed framework improved its usability for both salespeople and configuration engineers due to the improved understandability of system and bug fixes that were noticeable to the salespeople.

The long-term effects of the proposed method are still unknown. The presented results are based only on a single case study in a single commercially available PCS, which limits their generalizability. Therefore, further studies should both aim to test the long-term effects of using the proposed framework and test the framework in more companies in different industries, as well as in different software systems. Despite these limitations, it is our opinion that the field of product configuration modelling can learn a great deal from advancements in object-oriented design, design patterns and agile documentation. It is highly likely that other suitable patterns exist that can be explored in future research or derived from practical experiences in the industry.

Acknowledgements

We wish to thank the two anonymous reviewers for their valuable suggestions and comments, which helped improve both the presentation and quality of the paper. We would like to acknowledge Rasmuss Fiil Svarrer for his major contributions to the collection of the empirical data, which created the basis for the evaluation of the conceptual model presented here. The permission to evaluate systems and provide feedback in the company at a detailed level is gratefully acknowledged.

References

[Abul Khaer 2007] Abul Khaer, M.: “An Empirical Analysis of Software Systems for Measurement of Design Quality Level Based on Design Patterns”; 10th International Conference on Computer and Information Technology, Dhaka, Bangladesh (2007), 1-6. https://doi.org/10.1.1.465.2467

[Blessing and Chakrabarti 2009] Blessing, L. T. M., Chakrabarti, A.: “DRM, a Design Research Methodology”; Springer London, London (2009). https://doi.org/10.1007/978-1-84882-587-1

[Boucher et al. 2012] Boucher, Q., Abbasi, E. K., Hubaux, A., Perrouin, G., Acher, M., Heymans, P.: “Towards More Reliable Configurators: A Re-engineering Perspective”; 2012 3rd International Workshop on Product LinE Approaches in Software Engineering, PLEASE

2012—Proceedings (2012), 29-32. https://doi.org/10.1109/PLEASE.2012.6229766

[Coplien et al. 1998] Coplien, J., Hoffman, D., Weiss, D.: "Commonality and Variability in Software Engineering"; IEEE Software, 15, 6 (1998), 37-45. https://doi.org/10.1109/52.730836

[Dijkstra 1982] Dijkstra, E. W.: "On the Role of Scientific Thought"; Selected Writings on Computing: A Personal Perspective, Springer-Verlag, New York (1982), 60-66. https://doi.org/10.1007/978-1-4612-5695-3

[Duffy and Andreasen 1995] Duffy, A., Andreasen, M.: "Enhancing the Evolution of Design Science"; Proceedings of ICED 95, Praha, August 22-24, (1995).

[Eisenhardt 1991] Eisenhardt, K. M.: "Better Stories and Better Constructs : The Case for Rigor and Comparative Logic Author"; Academy of Management Review Journal, 16, 3 (1991), 620-627. https://doi.org/10.1105/tpc.019349.Rolland

[Falkner and Haselböck 2009] Falkner, A., Haselböck, A.: "A Simple Evaluation Process for Configurability"; Proceedings of the IJCAI–09 Workshop on Configuration (ConfWS–09) (2009), 17–22.

[Falkner et al. 2011] Falkner, A., Haselböck, A., Schenner, G., Schreiner, H.: "Modeling and Solving Technical Product Configuration Problems"; Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 25, 02 (2011), 115-129. https://doi.org/10.1017/S0890060410000570

[Felfernig et al. 2000a] Felfernig, A., Jannach, D., Zanker, M.: "Contextual Diagrams as Structuring Mechanisms for Designing Configuration Knowledge Bases in UML"; Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 1939 (2000), 240-254. https://doi.org/10.1007/3-540-40011-7

[Felfernig et al. 2000b] Felfernig, A., Friedrich, G. E., Jannach, D.: "UML as Domain Specific Language for the Construction of Knowledge-Based Configuration Systems"; International Journal of Software Engineering and Knowledge Engineering, 10, 4 (2000), 449-469. https://doi.org/10.1016/S0218-1940(00)00024-9

[Felfernig et al. 2001] Felfernig, A., Friedrich, G., Jannach, D.: "Conceptual Modeling for Configuration of Mass-Customizable Products"; Artificial Intelligence in Engineering, 15, 2 (2001a), 165-176. https://doi.org/10.1016/S0954-1810(01)00016-4

[Forza and Salvador 2002a] Forza, C., Salvador, F.: "Product Configuration and Inter-Firm Co-ordination: An Innovative Solution from a Small Manufacturing Enterprise"; Computers in Industry, 49, 1 (2002), 37-46. https://doi.org/10.1016/S0166-3615(02)00057-X

[Forza and Salvador 2002b] Forza, C., Salvador, F.: "Managing for Variety in the Order Acquisition and Fulfilment Process: The Contribution of Product Configuration Systems'; International Journal of Production Economics, 76, 1 (2002), 87-98. https://doi.org/10.1016/S0925-5273(01)00157-8

[Forza and Salvador 2007] Forza, C., Salvador, F.: "Product Information Management for Mass Customization', Palgrave Macmillan, New York (2007).

[Freeman 2015] Freeman, A.: "The Bridge Pattern"; In Pro Design Patterns in Swift, Apress, Berkeley, CA (2015), 271-292. https://doi.org/10.1007/978-1-4842-0394-1_13

[Gamma et al. 2002] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: "Design Patterns—Elements of Reusable Object-Oriented Software"; In A New Perspective on Object-Oriented

Design Addison-Wesley (2002). https://doi.org/10.1093/carcin/bgs084

[Haag 2017] Haag, A.: "Managing Variants of a Personalized Product"; Journal of Intelligent Information Systems, 49, 1 (2017), 59-86. https://doi.org/10.1007/s10844-016-0432-5

[Harlou 2008] Harlou, U.: "Developing Product Families Based on Architectures: Contribution to a theory of product families (PhD thesis)"; Orbit (Vol. 2007) (2008).

[Haug 2009] Haug, A.: "Implementation of Conceptual Product Models into Configurators: From Months to Minutes"; Proceedings of MCPC (2009).

[Haug 2010] Haug, A.: "A Software System to Support the Development and Maintenance of Complex Product Configurators"; International Journal of Advanced Manufacturing Technology (2010), 393-406. https://doi.org/10.1007/s00170-009-2396-x

[Haug et al. 2010] Haug, A., Hvam, L., Mortensen, N. H.: "A Layout Technique for Class Diagrams to Be Used in Product Configuration Projects"; Computers in Industry, 61, 5 (2010), 409-418. https://doi.org/10.1016/j.compind.2009.10.002

[Haug et al. 2019] Haug, A., Shafiee, S., Hvam, L.: "The Causes of Product Configuration Project Failure"; Computers in Industry, 108 (2019), 121-131. https://doi.org/10.1016/j.compind.2019.03.002

[Helo et al. 2010] Helo, P. T., Xu, Q. L., Kyllönen, S. J., Jiao, R. J.: "Integrated Vehicle Configuration System—Connecting the Domains of Mass Customization"; Computers in Industry, 61, 1 (2010), 44-52. https://doi.org/10.1016/j.compind.2009.07.006

[Hvam 2004] Hvam, L.: "A Multi-Perspective Approach for the Design of Product Configuration Systems—An Evaluation of Industry Applications"; Conference Proceedings (2004).

[Hvam et al. 2008] Hvam, L., Mortensen, N. H., Riis, J.: "Product Customization"; Springer-Verlag, Berlin (2008). https://doi.org/10.1007/978-3-540-71449-1

[Hvam et al. 2005] Hvam, L., Pape, S., Jensen, K. L., Riis, J.: "Development and Maintenance of Product Configuration Systems: Requirements for a Documentation Tool"; International Journal of Industrial Engineering : Theory Applications and Practice, 12, 1 (2005), 79-88.

[Hvam et al. 2003] Hvam, L., Riis, J., Hansen, B. L.: "CRC Cards for Product Modelling"; Computers in Industry, 50, 1 (2003), 57-70. https://doi.org/10.1016/S0166-3615(02)00143-4

[Jørgensen 2001] Jørgensen, K. A.: "Product Configuration—Concepts and Methodology"; Manufacturing Information Systems, Proceedings of the Fourth Smesme International Conference (2001).

[Kang et al. 1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: "Feature-Oriented Domain Analysis Feasibility Study (FODA)"; Technical Report, Pittsburgh, PA (1990).

[Karlsson 2016] Karlsson, C.: "Research Methods for Operations Management"; Taylor & Francis, New York (2016).

[Khomh et al. 2009] Khomh, F.; Guéhéneuc, Y.-G.; "An Empirical Study of Design Patterns and Software Quality"; 12th European Conference on Software Maintenance and Reengineering (2009), 274-278. https://doi.org/10.1.1.150.1235

[Khomh and Gueheneuce 2008] Khomh, F., Gueheneuce, Y.-G.: "Do Design Patterns Impact Software Quality Positively?"; 2008 12th European Conference on Software Maintenance and Reengineering. IEEE (2008), 274-278. https://doi.org/10.1109/CSMR.2008.4493325

[Khwaja and Alshayeb 2013] Khwaja, S., Alshayeb, M.: "Towards Design Pattern Definition Language"; Software: Practice and Experience, 43, 7 (2013), 747-757. https://doi.org/10.1002/spe.1122

[Larman 2004] Larman, C.: "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process"; Prentice Hall, (2001). https://doi.org/10.1016/j.nec.2006.05.008

[Mailharro 1998] Mailharro, D.: "A Classification and Constraint-Based Framework for Configuration"; AI EDAM, 12, 4, 383-397 (1998). https://doi.org/10.1017/S0890060498124101

[Martin 2002] Martin, R. C.: "Agile Software Development: Principles, Patterns, and Practices"; Pearson (2002).

[Martin and Micah 2006] Martin, R. C., Micah, M.: "Agile Principles, Patterns, and Practices in C#"; Pearson, (2006).

[McConnell 2004] McConnell, S. C.: "Code Complete"; Microsoft Press, Seattle (2004). https://doi.org/10.1039/c0an90005b

[Meredith 1998] Meredith, J.: "Building Operations Management Theory Through Case and Field Research"; Journal of Operations Management, vol. 16 issue 4, 441-454 (1998).

[O'Donnell et al. 1996] O'Donnell, F. J., MacCallum, K. J., Hogg, T. D., Yu, B.: "Product Structuring in a Small Manufacturing Enterprise"; Computers in Industry, 31, 3 (1996), 281-292. https://doi.org/10.1016/S0166-3615(96)00056-5

[Olsen and Saetre 1997] Olsen, K. A., Saetre, P.: "Managing Product Variability by Virtual Products"; International Journal of Production Research, 35, 8 (1997), 2093-2108. https://doi.org/10.1080/002075497194750

[Ramachandran and Allen 2005] Ramachandran, M., Allen, P.: "Commonality and Variability Analysis in Industrial Practice for Product Line Improvement"; Software Process: Improvement and Practice, 10, 1 (2005), 31-40. https://doi.org/10.1002/spip.212

[Selic 2009] Selic, B.: "Agile Documentation, Anyone?"; IEEE Software, 26, 6 (2009), 11-12. https://doi.org/10.1109/MS.2009.167

[Shafiee et al. 2015] Shafiee, S.; Hvam, L.; Kristjansdottir, K.: "An Agile Documentation System for Highly Engineered, Complex Product Configuration Systems"; Proceedings of the 22nd Euroma Conference (2015).

[Shafiee et al. 2017] Shafiee, S., Hvam, L., Haug, A., Dam, M., Kristjansdottir, K.: "The Documentation of Product Configuration Systems: A Framework and an IT Solution"; Advanced Engineering Informatics, 32 (2017), 163-175. https://doi.org/10.1016/j.aei.2017.02.004

[Shalloway and Trott 2002] Shalloway, A., Trott, J. R.: "Design Patterns Explained: A New Perspective on Object-Oriented Design"; AddisonWesley Publ Co, United States (2002).

[Sharon 1996] Sharon, D.: "Meeting the Challenge of Software Maintenance"; IEEE Software, 13, 1 (1996), 122-125. https://doi.org/10.1109/52.476304

[Soininen et al. 1998] Soininen, T., Tiihonen, J., Tomi, M., Sulonen, R., Männistö, T.: "Towards a General Ontology of Configuration"; Aiedam, 12, 4 (1998), 357-372. https://doi.org/10.1017/S0890060498124083

[Staples 2004] Staples, J.: "Agile Documentation: A Pattern Guide to Producing Lightweight

Documents for Software Projects"; Technical Communication, (Vol. 51, Issue. 4 p. 560-561) (2004).

[Thomas and Wesley 1999] Thomas, D., Wesley, P. A.: "The Pragmatic Programmer"; Addison Wesley, United States (1999).

[Tseng et al. 2005] Tseng, H.-E., Chang, C.-C., Chang, S.-H.: "Applying Case-Based Reasoning for Product Configuration in Mass Customization Environments"; Expert Systems with Applications, 29, 4 (2005), 913-925. https://doi.org/10.1016/j.eswa.2005.06.026

[Wielinga and Schreiber 1997] Wielinga, B. J., Schreiber, A. T.: "Configuration Design Problem Solving"; Special Issue on AI and Design, 12 (1997), 49-56.

[Wiseman 2007] Wiseman, G.: "Do Agile Methods Require Documentation?"; (2007). Retrieved 19 December 2017, from https://www.infoq.com/news/2007/07/agile-methods-documentation

[Yang et al. 2009] Yang, D., Miao, R., Wu, H., Zhou, Y.: "Product Configuration Knowledge Modeling Using Ontology Web Language"; Expert Systems with Applications, 36, 3 (2009), 4399-4411. https://doi.org/10.1016/j.eswa.2008.05.026

[Yu and MacCallum 1995] Yu, B., MacCallum, K.: "A Product Structure Methodology to Support Configuration Design"; WDK International Workshop on 'Product Structuring', 22-23 June, Delft, The Netherland (1995), 1-8.

[Zhang 2014] Zhang, L. L.: "Product Configuration: A Review of the State-of-the-Art and Future Research"; International Journal of Production Research, 52, 21 (2014), 6381-6398. https://doi.org/10.1080/00207543.2014.942012

Appendix

Workshop presenting of the new PCS to end users (Salespeople)

Program 9.30 AM to 3.00 PM.

Participants were twelve salespeople, including head of sales, head of research and development (R&D), a configuration engineer and a researcher.

Presentation of the new PCS compared with ad hoc PCS

- New user interface
- How to reconfigure from one product series to another

Testing the new system (3 hours)

- Assignment 1: Find a relevant case from real-life experiences with a requirement to change between product series.
- Assignment 2: Re-create relevant case in modular PCS and change between product series
- Assignment 3: Estimate how much time consumption has been reduced and the benefits of the restructured model

Evaluation of the system in plenum

- Discussion of the new system, including change recommendations

Evaluation of system structure and documentation (semi-structured informal interviews with prior configuration engineer, answers not transcribed)

Presentation of the framework and new documentation

Presentation of new structure compared to ad hoc structure

- Can you see the value of Step 1?
- Can you see the value of Step 2?
- Can you see the value of Step 3?
- Do you think this structure is better/worse than the ad hoc structure?
- Would the division of the abstraction/implementation be useful for future modelling?
- Do you see any drawbacks?
- Can you see other benefits of the modular structure?

Questions regarding new documentation approach based on the structure

- Do you find this documentation more useful than the old documentation?
- Do you believe it will be more useful over time than the old documentation?
- Could you see yourself starting to adopt some of the guidelines in your modelling?