JITTA

JOURNAL OF INFORMATION TECHNOLOGY THEORY AND APPLICATION

ISSN: 1532-3416

A Platform-based Design Approach for Flexible Software Components

Marcus A. Rothenberger
Lee Business School
University of Nevada Las Vegas

Hemant Jain
College of Business
University of Tennessee at Chattanooga

Vijayan Sugumaran
School of Business Administration
Oakland University

Abstract:

We develop a design method that promotes flexible component design based on a common component platform with various plug-ins. The approach increases the flexibility and expandability of software components, which improves their reuse opportunities. We argue that such a flexible component design can expand reuse from relatively small infrastructure items, such as user interfaces, printing functionality, and data access modules, to the core of the application domain. Reusing such domain-specific items helps realize the true value of component-based software development. Following a design science research approach, we evaluated the component design method by assessing its correctness and its application to different scenarios. We also recruited a panel of experts to assess it.

Keywords: Component Design, Software Reuse, Component-based Development, Service Composition, Design Science.

Timo O. Saarinen was the Senior Editor for this paper.

1 Introduction

Although software productivity has steadily increased, the demand for improvements in software development methods remains high. To address these demands, research has built reusable artifacts, such as components, services (Chengjun, 2008; Chu & Qian, 2009; Jain & Vitharana, 2000; Orriens & Yang, 2008; Vitharana, Zahedi, & Jain, 2003b; Yau et al., 2009), and design patterns (Czarnecki, 2007; Erwig & Fu, 2005; Park, Park, & Sugumaran, 2007). Researchers have suggested that one can achieve low development cost, high product quality, and low development time (Mohagheghi & Conradi, 2008; Slyngstad et al., 2006) by generating new designs through combining high-level specifications with existing artifacts (i.e., reusing artifacts) (Ramachandran, 2005). Initially, reuse research focused primarily on code-based in-house reuse. However, a comparably small repository limits such reuse (Basili, Briand, & Melo, 1996; Poulin, Caruso, & Hancock, 1993). Only after the introduction of component standards such as CORBA and JavaBeans that component markets that support the notion of inter-organizational reuse started emerging. The wide adoption of Web-service standards (Rodriguez, Crasso, Mateos, Zunino, & Campo, 2013; Silic, Delac, Krka, & Srbljic, 2013) and service-oriented architectures (Girbea, Suciu, Nechifor, & Sisakm, 2013; Li, Muthusamy, & Jacobson, 2010; Welke, Hirschheim, & Schwarz, 2011) has further enhanced this movement. In service composition and component-based development, one customizes services or components by setting appropriate parameters that prompt the service or component to act consistently with the application's requirements. However, because reuse requires a high-level of standardization in functionality and interfaces and clearly defined functionality to enable searches, organizations are achieving reuse more widely at the infrastructure level than at the application domain level (Holmes & Walker, 2012). Reusable software assets at the infrastructure level include user interface constructs, printing functions, and data-access routines with highly standardized interfaces and functionality that one can clearly define for easy retrieval through search engines (Crnkovic, Stafford, & Szyperski, 2011). These infrastructure services or components have high reuse potential because one can use their functionality across application domains. On the other hand, domain-level services or components are more difficult to standardize and have limited reuse potential since their functionality must match functional application requirements. Thus, one may attribute low domain-level reuse to low component flexibility and the difficulty of finding software assets that meet application requirements (Gill, 2003; Vitharana, Zahedi, & Jain, 2003a). Nevertheless, one cannot realize the true value of reuse until one can build a substantial portion of an application in a business domain by reusing software components/services at the domain level and the infrastructure level. To do so, one must design domain-level software assets to be flexible in order to improve the chances of matching requirements and, thus, increase reuse potential. In fact, Sharp and Ryan (2010) have pointed out the need for additional work on component adaptability.

Developing applications through component assembly and through service composition are similar in that both can benefit from a more flexible design as we discuss above; however, the service paradigm differs in the sense that a service might use multiple components to provide the functionality or a component could provide multiple services. In this study, we develop a design approach for flexible software assets that can apply to both contexts; nevertheless, we present and evaluate the design approach in the context of the component-based paradigm. Because of its lower conceptual complexity, it better suits our work to develop and illustrate this new approach.

Contribution:

This paper proposes a novel component design approach that can lead to increased domain-level reuse opportunities. The approach applies lessons learned from manufacturing about platform-based products (e.g., cars, printers, etc.) to design software components. While developers have used platform concepts at the software product level in the past, we develop a novel approach in designing each individual domain-specific component intended for reuse as a combination of a generic and highly reusable component platform and more specific plug-ins. One can repurpose component platforms to meet new domain needs by adding new plug-ins in different ways without rewriting the original component platform. The design approach defines how one can structure domain-specific components to support flexibility and maximize their reuse potential, and we evaluate the approach against its objectives.

The proposed design approach defines how one can structure domain-specific components to support their flexibility and, thus, reuse potential. The approach builds on platform-based components—that is, domain-level components that one can customize (without code changes) to meet varying needs (Jain, Rothenberger, & Sugumaran, 2006). Thus, the proposed approach enhances the flexibility of components in meeting application requirements. Motivated by the concept of product platforms in manufacturing (Sääksjärvi, 2002; Salvador, Forza, & Rungtusanatham, 2002), we develop a component design method that incorporates aspects of domain analysis and domain modeling to design core component platforms and plug-ins.

The paper proceeds as follows: in Section 1.1, we discuss the related literature to specify the problem and motivate the design approach. In Section 2, we present the flexible component design approach. In Section 3, we demonstrate how the approach works by applying it to a sample scenario. In Section 4, we evaluate the approach by having a group of software development experts interact with it and assess its utility (Hevner, March, Park, & Ram, 2004; Peffers, Tuunanen, Rothenberger, & Chatterjee, 2008). In Section 5, we conclude the paper.

1.1 Problem Definition

Researchers have argued that developing an information system from reusable components results in a more reliable product (Hissam, Seacord, & Lewis, 2002; Hopkins, 2000), increases developer productivity (Lau, 2006), reduces required skills (Sinha & Jain, 2013), shortens development lifecycle (Due, 2000; Manolios, Vroon, & Subramanian, 2007), reduces time to market (Kharb & Singh, 2008), increases the developed system's quality (Sprott, 2000), and reduces development costs (Due 2000). Beyond these operational benefits, researchers have also found component-based software development (CBSD) to provide strategic benefits, such as the opportunity to enter new markets or the flexibility to respond to competitive forces and changing market conditions (Favaro, Favaro, & Favaro, 1998; Hissam et al., 2002). Component providers have the opportunity to enter new markets because of the potential to cross-sell components with associated functionalities. Similarly, end user organizations have the flexibility to quickly substitute components with newer ones that contain additional features to respond to competitive forces and changing market conditions (Scott, Robert, & Grace, 2002).

CBSD has impacted the way organizations develop and deliver applications to end users (Forte, Claudino, de Souza, do Prado, & Santana, 2007; Heinecke et al., 2008). It has caused a shift in software development paradigms, particularly with the development of several component architecture standards such as common object request broker Architecture (CORBA), component object model (COM), and enterprise Java beans (EJB) (Gill, 2006; Szyperski, 1998). A component is a well-defined unit of software that has a published interface and can be used in conjunction with other components to form larger units (Heineman, 2000; Hopkins, 2000). As we discuss in Section 1, domain-level component reuse has been low due to the limited flexibility of parameterized components and the difficulty of finding components in a library that match an application's exact requirements(Gill, 2003; Vitharana et al., 2003a). Parameterization, an approach in which one develops components with optional functionalities and choices that one can trigger by their parameters (Gill, 2006), limits a component's possible applications to what the component developer initially anticipated. In order to increase the reuse potential of domain-level components, software components must provide the flexibility to allow their reuse in a large number of applications (Heineman, 2000).

Designing smaller size components (with less functionality per component) would make it easier for software developers to match the requirements with component functionality (Lau, 2006; Vitharana, Jain, & Zahedi, 2004) and, thus, increase each individual component's reuse potential. However, a low component granularity with low functional complexity per component is problematic (Vitharana et al., 2004) since one needs to perform more steps to retrieve and combine small components (Hong & Lerch, 2002). Further, each such component represents a low development effort, which reduces the reuse leverage of each instance; activities, such as component retrieval and integration, take up a proportionally larger share of development time and, thus, increase development costs, which can render the reuse effort economically unfeasible (Nazareth & Rothenberger, 2004). This situation leads to low developer demand for components with low functional complexity (Hong & Lerch, 2002). The platform-based approach we propose in this paper provides a mechanism to create flexible components that can meet user requirements without reducing component granularity.

In the physical world, modular product platforms have been a means to increase product variety to better meet varying customer requirements (Sääksjärvi, 2002; Salvador et al., 2002). For example, Volkswagen developed the PQ35 platform as a basis for at least 19 models, including the Audi A3, the Volkswagen Tiguan and Golf, the Skoda Octavia, and the Seat Toledo (Volkswagen group A platform, n.d.). Hewlett-Packard's (HP) OfficeJet platform combined functions of previously distinct products, such as computer printers, fax machines, scanners, and photocopiers, to meet customer demand in a flexible manner (Meyer & Lehnerd, 1997). The principles of developing modular physical products may also apply to developing software components (LaMantia, 2006; Salonen & Sääksjärvi, 2004). As in manufacturing, modularity promotes an increased fit between software and customer specifications by enabling developers to combine and customize elements of the application, which reduces the need for custom development based on each client's specifications.

The underlying concepts of physical product platforms have already found their way into software development as software product lines. The software product line approach focuses on modeling the commonalities and variations in features or functionalities in the application domain and linking them to software components (Baresi, Guinea, & Pasquale, 2012; Capilla et al., 2014). This means that, if a developer requires a particular feature in a system, then the developer can reuse the components that support the feature (Dhungana, Rabiser, Grnbacher, & Neumayer, 2007; Rosenmller, Siegmund, Saake, & Apel, 2008). The software product line approach is a means to produce software more quickly and economically (Bell, 2007; Krueger, 2006), and researchers recognize it as a successful method for improving reuse in software development (Bosch, 2000a; Kang, Lee, & Donohoe, 2002); however, it does not address the flexibility of the individual reusable component. The platform-based component design method we propose in this research fills this gap by addressing the nature of the reusable component itself.

A product line enables organizations to develop a product family from reusable core assets rather than from scratch (Sugumaran, Park, & Kang, 2006). The key requirements of developing future products drive how organizations design product lines, which means they need to identify these requirements (Clements, Jones, McGregor, & Northrop, 2006). Thus, organizations must perform a thorough requirements analysis for the product line, which involves systematically identifying and describing particular common and variant requirements (i.e., a commonality and variably (C&V) analysis) (Laguna, Gonzlez-Baixauli, & Marques, 2007). Furthermore, the identified requirements and commonality and variability must satisfy an organization's high-level business goals. Thus, organizations must carry both analyses out to satisfy these high-level business goals and provide the rationale for them. While software product lines apply platform concepts at the application domain level, our approach uses them at a lower level of functional granularity (i.e., at the individual component level), which results in more flexible components.

Thus, our approach provides the means to develop a component design that increases component flexibility over traditional parameterized components by building a platform for each component that allows one to create custom components using different combinations of available plug-ins (plug-ins are lower-level component stubs that extend the functionality of a component platform or a higher-level plug-in). Our method for designing a platform-based component combines individual functionally related components from multiple domains into a component hierarchy that is equivalent in design to the original individual components yet more flexible in that one can extend and use the hierarchy across multiple domains.

2 Development of the Platform-based Component Design Method

2.1 Platform-based Design through Unification of Domain Models

According to product platform principles, reusable flexible components comprise a core component platform and multiple hierarchical levels of plug-ins. For a specific reuse instance, selecting appropriate plug-ins from the hierarchy helps one to customize the component to meet various requirements. The method we develop in this research consolidates different component designs that are based on different requirements into a core platform component and one or more plug-in hierarchies that are consistent with the product platform principles. The method we use to design the flexible components is the artifact of this design science research. It unifies the functional requirements of the common component user base. Based on platform-development in manufacturing, customers' requirements and demands must guide the design of a flexible component (Meyer & Seliger, 1998). In other words, the projected market for the component must drive its design, which can be determined through domain analysis. We use the unified modeling language (UML) notation of the object-oriented design model to illustrate the design of a flexible component. Our method formalizes the design of the core component platform and plug-in hierarchy.

2.2 Design Methodology

Domain analysis is an activity similar to systems analysis. Domain analysis involves analyzing existing systems in the application domain and creating a domain model that characterizes it (Tolvanen, Gray, Rossi, & Sprinkle, 2008). According to Neighbors (1984), “domain analysis and modeling is an activity in which all systems in an application domain are generalized by means of a domain model that transcends specific applications”. Thus, a domain model represents the common characteristics and variations among the existing and future members of a family of software systems in a particular application domain. Researchers have used domain-analysis methods such as the feature-oriented approach (Fey, Fajta, & Boros, 2002; Kang, Lee, & Lee, 2002; Metzger & Pohl, 2007), reuse-driven software engineering business (RSEB) (Jacobson, Griss, & Jonsson, 1997), FODAcom (Vici, Argentieri, Mansour, d’Alessandro, & Favaro 1998), FeatuRSEB (Griss, Favaro, & d’Alessandro, 1998), and product line analysis (PLA) (Chastek, Donohoe, & McGregor, 2002) to analyze commonality and variability among families of software systems in application domains. In particular, researchers and practitioners have used the feature-oriented approach extensively (Clements et al., 2006; Donohoe, 2000; Kang et al., 2003). In this approach, one analyzes commonality and variability in terms of features, which provides a feature-based model to develop reusable assets (Kang et al., 2002; White et al., 2014). During feature modeling, if there are no existing products or if the existing products do not have a specified set of features associated with them, then one must identify and define the features associated with each individual product (Bosch, 2000b). Thus, feature modeling constitutes a big part of domain modeling (Fey et al., 2002; Griss et al., 1998; Kang et al., 2002, 2003; Metzger & Pohl, 2007). Once feature model is developed, the component designs that implement these features and establishes links between the features and the corresponding component designs is completed. One limitation of the existing domain model-based approaches to reuse concerns their lack of support for integrating different components that represent related functionalities from various domain models (Czarnecki, 2007; Jalender, Govardhan, Premchand, 2010; Park et al., 2007). Overcoming this limitation would allow one to extend the domain model with new features. As such, our approach provides a systematic way to combine the designs of different components that represent related functionalities from various domain models to create a higher level of abstraction.

To design a platform-based component, we use the individual component models that meet the requirements of each application domain that the platform-based component targets. One must transfer the commonalities between the individual component designs into the core platform component; the plug-in design hierarchy of the platform-based component handles the variations (we assume that commonalities exist because only sets of domain models that share at least some common design features would be good candidates for a platform component hierarchy). Finally, if necessary, one can evaluate and fine-tune the platform-based component design. Figure 1 shows the overall process diagram that depicts our method for creating the flexible component design. The proposed method has two major steps: 1) select relevant component designs from the domain models and 2) apply the transformation algorithm to generate the core component platform and plug-in hierarchy.

2.2.1 Selecting Relevant Component Designs

The domain models contain feature models and, for each feature, the corresponding component designs that implement that feature. Based on the functional requirement specified for the flexible component one seeks to design, one can identify the appropriate features incorporated in the domain models that would satisfy this requirement and gather the component designs associated with these features. For example, if one seeks to design a flexible reservation component, then one would consider domain models that support the reservation feature and identify the corresponding component designs that implement this feature as the potential components for the transformation algorithm to transform. The reservation concept is inherent to the airline, train, entertainment show, and sporting event domains. Consequently, the domain models from these domains would incorporate the reservation feature and include components that implement it. Hence, to develop a flexible component design for the reservation functionality, one would select the reservation-related features and the respective component designs from these domain models. While researchers have discussed several approaches in the domain modeling and software product line literature (Bosch, 2000b; Kang et al., 2003; Neighbors, 1984; Sugumaran, Tanniru, & Storey, 2008; Sugumaran et al., 2006) for identifying features and selecting components, we adopt the approach that Sugumaran et al. (2008) discuss. Thus, this step outputs the component designs that implement the features from various domain models that correspond to the functionality that one desires in the flexible component that one seeks to develop. The next step uses this set of components as its input.

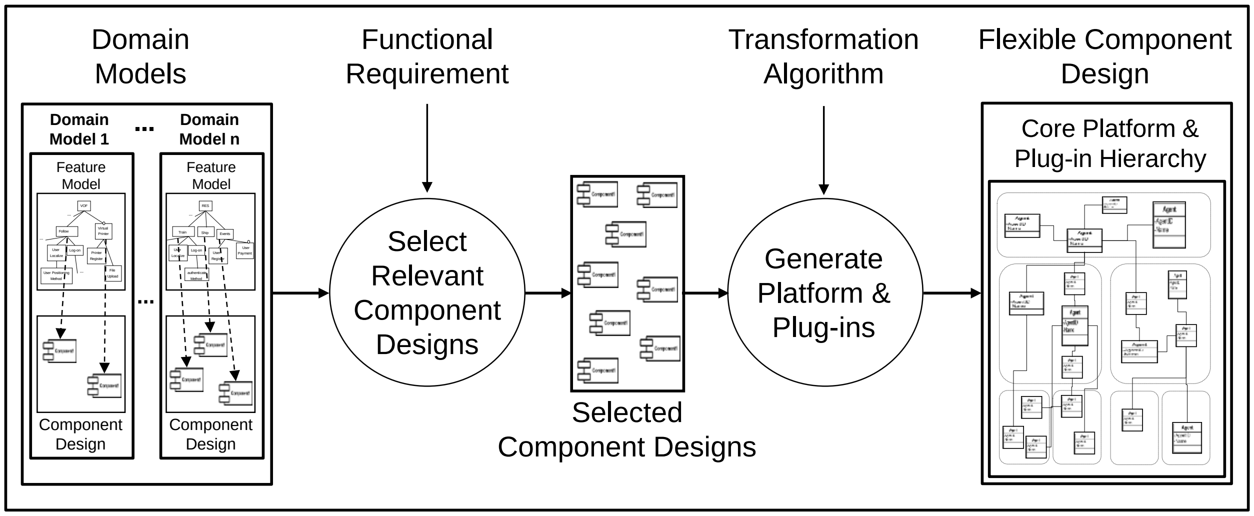
Domain Models
Functional Requirement
Transformation Algorithm
Flexible Component Design
Domain Model 1
Domain Model n
Feature Model
Component Design
Select Relevant Component Designs
Selected Component Designs
Generate Platform & Plug-ins
Core Platform & Plug-in Hierarchy

Figure 1. Overall Process Diagram

2.2.2 Applying Transformation Algorithm

In this step, one applies the transformation algorithm, which takes the set of selected component designs and generates the core component platform and plug-in hierarchy. To transform the individual component designs to the platform-based component, the algorithm identifies and integrates the commonalities that exist between the individual component designs. We use a simplified example of three component designs to illustrate the approach. All classes in the example are either distinct from or common to other designs; the example does not include classes with partially common functionality across individual component designs. We discuss how the approach deals with classes that contain partially common functionality across the individual component designs in Section 3. Figures 2 through 4 show the simplified component design examples that the process will later integrate into a component hierarchy.

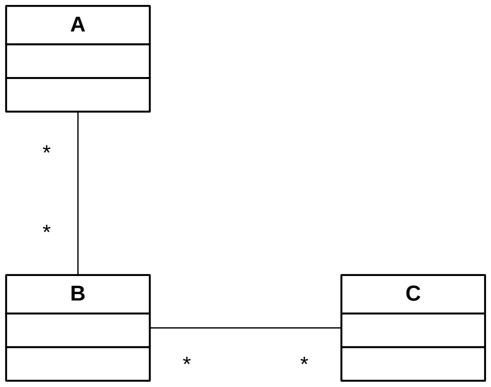

Figure 2. Example: Relevant Component from Domain Model 1

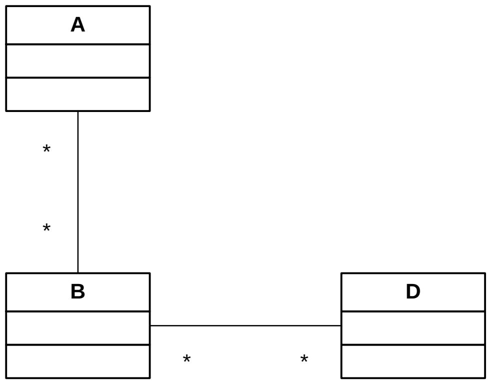

Figure 3. Example: Relevant Component from Domain Model 2

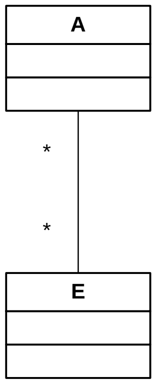

Figure 4. Example: Relevant Component from Domain Model 3

Since all three component designs have class A, the process moves it into the core component platform. As a result, class B becomes common to the largest number of different designs (two out of the three). Thus, the process moves it into one level 1 plug-in, and it builds the component hierarchy for these two designs to the lowest level by recursively executing the same steps for these designs: the two designs that have class B in common differ in that one design includes class C and the other design includes class D. Thus, the process includes classes C and D in two different level 2 plug-ins that are subordinate to the level 1 plug in that contains class B. Subsequently, the process deals with the remaining designs: only the third design is left and it includes class E, which differentiates it from the other designs. Thus, the process moves class E into a separate level 1 plug-in. Figure 5 shows the resulting plug-in hierarchy.

We developed the above transformation algorithm via an application and refinement process that involved going through multiple iterations of applying the method to different scenarios and evaluating the outcome. Table 1 depicts the resulting recursive algorithm in pseudo code that we name “CreatePlugIn”. Appendix A provides a formal specification of this algorithm. While both are equivalent, we include the pseudo code version in the main body of the paper for better readability. Like in the preceding simplified example, the process identifies commonalities across all individual component designs that it will later integrate into a plug-in hierarchy. Hereby, the process must identify commonalities based on identical functionality rather than identical names because class, method, and attribute names may vary across different designs even though they may implement the same functionality. The process moves common classes (or generalizable common parts of different classes) across the individual designs into the core platform component. Subsequently, it recursively identifies commonalities between subsets of all designs (starting with common design elements across the largest number of individual domain models) and moves them into the next level of plug-ins. We provide an illustrative application of the transformation algorithm in Section 3.

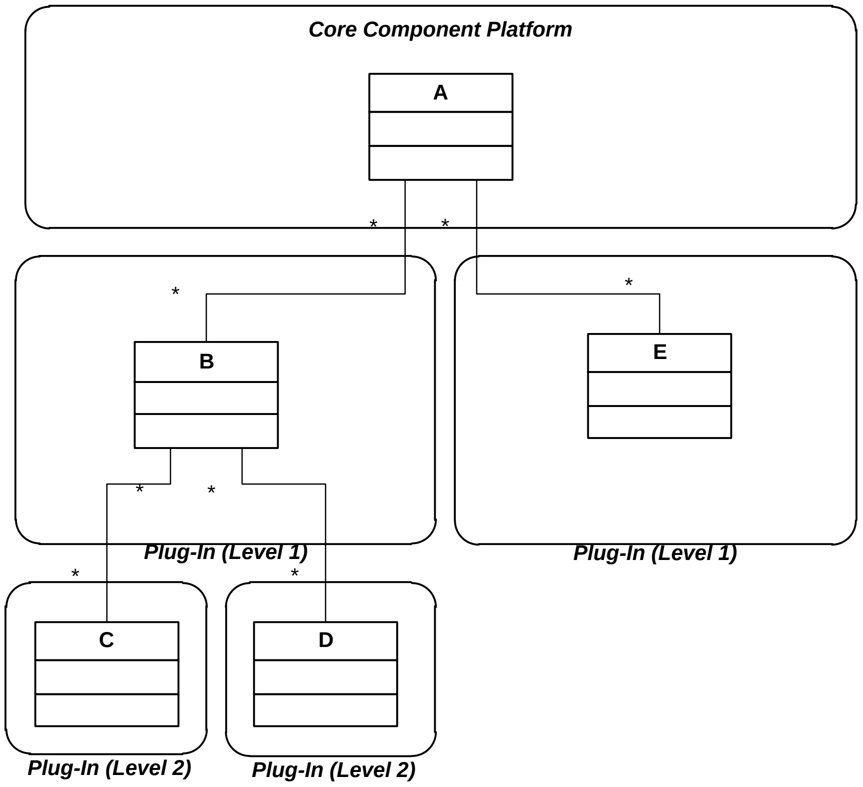

Figure 5. Example: Platform-based Component that Comprises a Core Component Platform and Plug-ins

Table 1. CreatePlugIn(S_D) in Pseudo Code)

```
1. $S_C$ is the union of all classes contained in the individual component designs $S_D$
2. For all classes (or possible class generalizations) $C_j$' that are part of the set of classes $S_c$,
   where no other class generalization exists in any individual component design that incorporates more functionality of the original class than $C_j$' does,
   add $C_j$' to the current plug-in (or core component platform if this is the highest level), remove $C_j$' from the set of classes $S_C$, and remove $C_j$' from all individual component designs $D_i$.
3. Repeat until the set of designs $S_D$ is empty
    a. Find the class (or possible class generalizations) $C_j$' that is part of the set of classes $S_C$ and that is contained in the largest number of individual component designs $D_i$
       where no other class generalization exists in the same set of individual component designs $D_i$ that incorporates more functionality of the original class than $C_j$' does,
       create a subset of all designs $S_D$' that includes only the individual designs that contain $C_j$' and remove the designs $S_D$' from $S_D$
    b. The next Plug-In will be a child of the current Plug-In $P_k$.
```

3 Demonstration of the Method Using a Sample Scenario

To demonstrate our method to design flexible components, we use a sample scenario of designing a reservation component. We use the individual component designs selected from four different but related domain models (namely, train reservations, airline reservations, show reservations, and sporting event reservations) as input. Appendix B presents these component designs. All individual component designs have common elements pertaining to ticketing, payment, customer information, booking agent, and the transaction processing. Further, transport reservations (airline and trains) and event reservations (shows and sporting events) have partial commonalities: transport reservations have itineraries, passengers, and routes in common, while the event reservations both use venues, seats, events, and price categories. We applied the method we introduce in this paper to design the core component platform and plug-in hierarchy of the reservation component. Figure 6 illustrates the resulting design of core component platform and plug-in hierarchy.

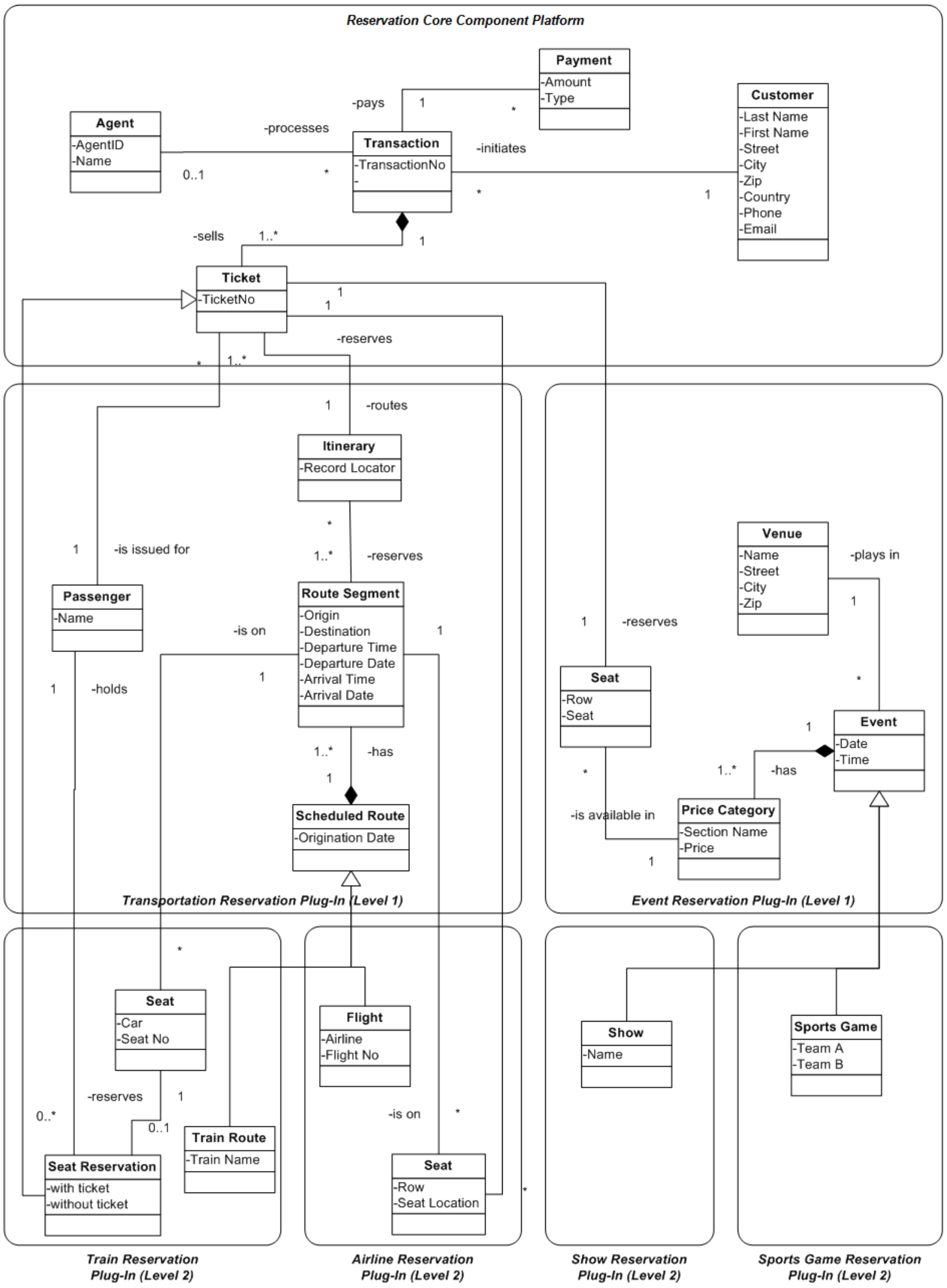

Figure 6. Reservation Component Consisting of Core Component Platform with Industry Plug-ins

This example demonstrates that a platform component can have multiple levels of plug-ins; the transformation method and the nature of the domain models used as input determine the actual number of levels for a specific design. Here, the example provides plug-ins at two levels; the first level incorporates industry segment-specific functionality for transportation and event reservations, and the second level provides the lowest level of industry functionality for train, airline, show, and sporting event reservations.

As we discuss in Section 2, in the context of the simplified example, the method will move a class with functionality that differs from those contained in the other domain models into the plug-in that implements the respective class's domain model (e.g., seat class in Figure 6). However, some classes may appear in similar yet not identical form in multiple of the component designs that the process will later transform into a platform component. The method deals with the commonalities between similar (not identical) classes in multiple component designs by implementing common attributes or common methods in a parent class that will be located in the core platform or in a higher-level plug-in (e.g., scheduled route class in Figure 6). To implement the differences between such similar classes in multiple component designs, the lower-level plug-ins contain a class that inherits the properties from the parent class that the process previously added to the core component platform or a higher-level plug-in; this child class then extends the parent class functionality according to the differences in functionality in each component design (e.g., flight class in Figure 6).

4 Evaluation of the Design Method

In Section 2, we introduce a component design method that combines individual domain component designs into a flexible platform-based component. We developed such a component to create reusable software assets that are flexible in two ways: they meet the requirements across all the individual components' domains that they have been built from, and one can extend them to meet requirements beyond the domains that the original individual components covered.

As such, our method must address both of these aspects when evaluating it. To demonstrate that the flexible platform-based components our method creates meet the requirements of the original individual components that one builds them from, we show that a resulting core component platform and its plug-in hierarchy are equivalent in design to the individual components they came from. To demonstrate that one can extend the flexible platform-based components beyond their original domains, we asked five software development experts to independently assess how one would need to extend the flexible platform-based reservation component we developed to meet the new requirements of cruise ship reservations. Their evaluation results show that the experts matched the component extensions that we developed according to our approach.

4.1 Evaluation of Equivalency of the Design

First, we evaluate whether the flexible platform component design we developed using our method is equivalent to the individual components of the various domains used as input. We evaluated the flexible platform component by decomposing it into its original individual separate components and comparing the decomposed pieces with the original component designs used as input. We evaluated the component using the demonstration scenario we discuss above. We decomposed the core component platform and plug-in hierarchy (Figure 6) into the component designs we built them from as follows: we start off with the platform component design that resulted from the application of our approach, referred to as a tree structure. Each plug-in is connected to plug-ins that are located on a higher level in that tree structure with the lowest-level plug-in being the leaf node, higher-level plug-ins being the parent nodes, and the core component platform being the root of the tree. The paths from each leaf node via the parent nodes to the core component platform represent the path of plug-ins, which we merge to represent a decomposed functionality modeled by a leaf node. One can decompose the flexible platform component in Figure 6 into four separate individual component designs because it has four leaves. Hereby, one creates each decomposed component design by adding all classes and their associations located on a path discussed above. However, the platform-based component design approach may have introduced additional parent classes into the design by breaking down a class of the original design into a parent-child pair (our design approach does that to model the common functionality across the individual component designs). These new parent-child pairs carry over into the decomposed design, and one must merge them to get the original class. One can easily identify the additional parent classes because they are the only parent classes with only one child in the decomposed design (note that the original design should not include parent classes with only one child because this would be poor modeling). After reintegrating each parent class with its child class, one can see that the four decomposed components' designs are semantically equivalent to the four original component designs shown in the Appendix B.

For space considerations, we present here only the decomposition for the airline reservation plug-in. However, we conducted the other decompositions as well. Decomposing the airlines reservation plug-in resulted in the component that Figure 7 depicts. The parent-child pair scheduled route/flight was originally a single class that our approach converted into the inheritance hierarchy. Merging these two parent-child pairs into an individual class leads to the component design that Figure 8 shows. Its design is syntactically and semantically equivalent to that of the original airline reservation component depicted in the Appendix B (please note that different names for the individual classes do not affect the equivalency of two designs). Thus, given that we demonstrate the designs' equivalency, we can conclude that flexible platform component designs that we developed according to the method we introduce in this study meet the same utility as the sum of the individual component designs that we developed them from. The evaluation shows that the core component platform and plug-in hierarchy that resulted by applying the proposed method to the individual component designs is equivalent to the individual component designs. Thus, one can deem the transformation algorithm lossless and correct.

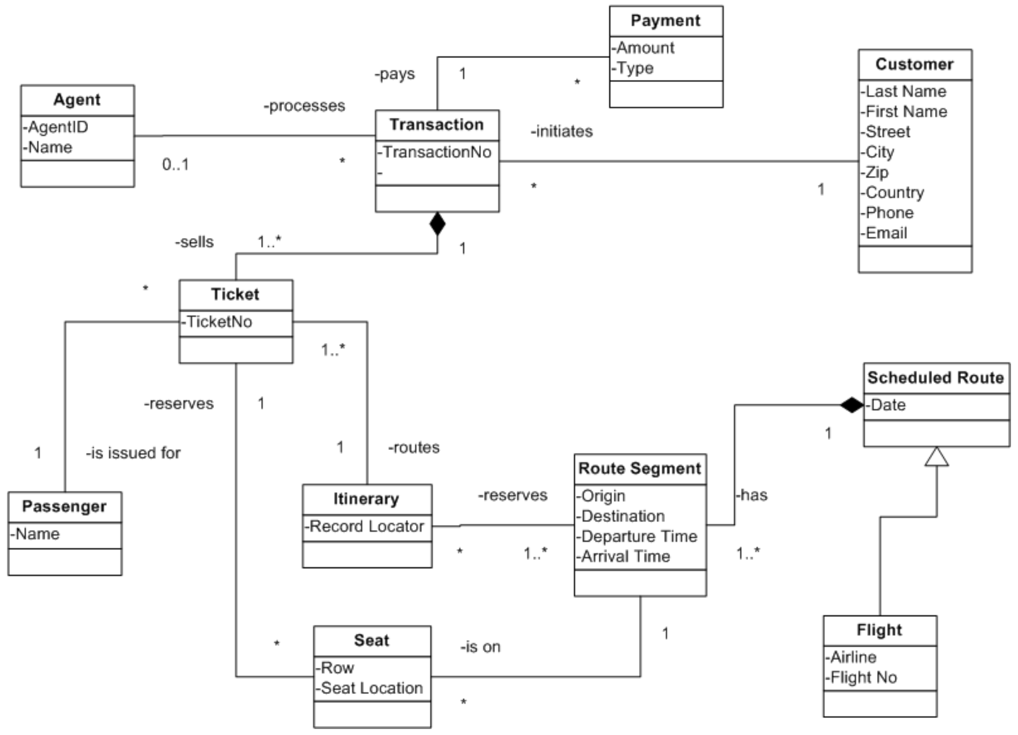

Figure 7. Decomposed Airlines Reservation Component (Step 1)

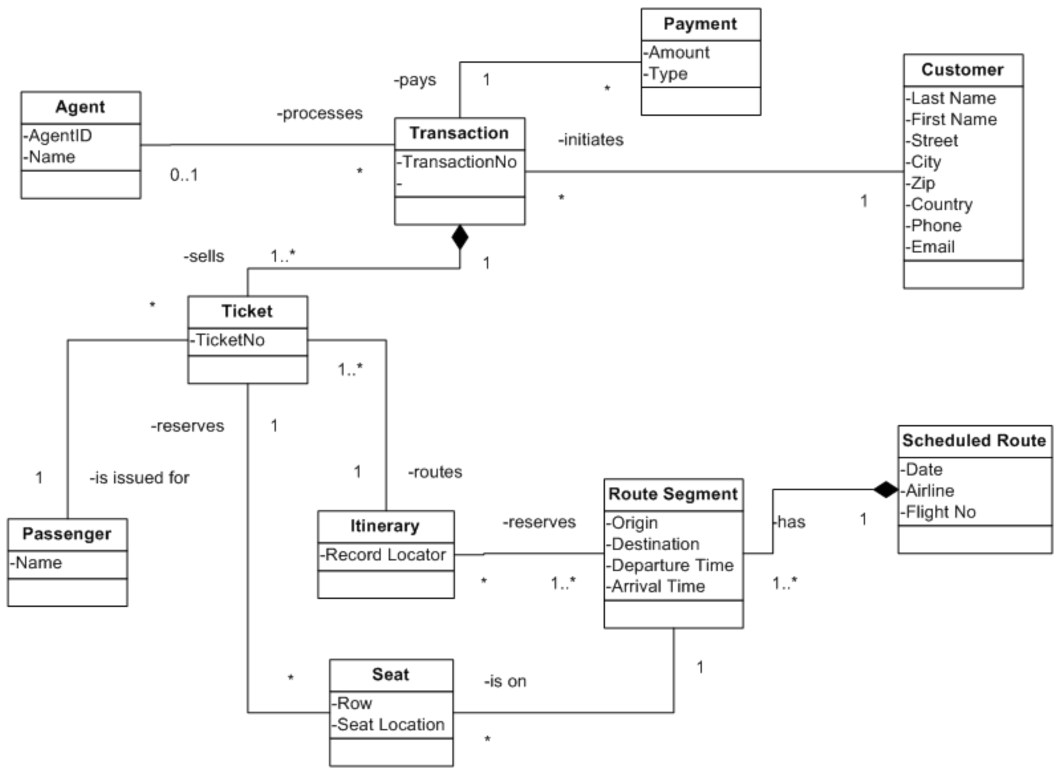

Figure 8. Decomposed Airline Reservation Component (Step 2)

4.2 Higher Flexibility of Platform Component

Second, we evaluate whether the platform component design that results from the method we introduce in this paper is more flexible than the original individual component designs. In the first evaluation step, we demonstrate that the platform component addresses the requirements of all individual components that we created it from. However, this feature alone only represents a limited advantage over individual components because platform components may be cheaper to develop and maintain (compared to developing four separate components in the scenario) and easier to organize and retrieve from a component repository. However, the flexibility advantage of a platform component design concerns whether one can reuse it in another (not initially planned for) domain by extending it via plug-ins that do not require the developer to understand the implementation details of the core component platform and existing plug-ins. Not needing to understand these details represents a substantial advantage over developing the component from scratch or extending a domain component because it encourages one to reuse the core component platform and plug-ins. Additionally, the core component platform clearly defines plug-in interfaces, which enables developers to meet new functional requirements by adding new ones to the core component platform without needing to understand design and implementation details of the reused parts of the core component platform. Thus, one can make functional additions to a component that one otherwise reuses in a black-box fashion. In contrast, extending a domain component design with functionality that exceeds the original design would require developers to understand the component in detail so they could assess which classes of the original design to reuse and how to integrate the design extension into the existing design. To support our claim of higher flexibility, we demonstrate the extensibility of the platform component design using the reservation scenario. One can make two types of extensions: one can extend a platform-based component along an existing functional dimension, which means that one or more new plug-ins are added to an existing plug-in hierarchy, or one can add a new functional dimension by creating new plug-ins to form an additional plug-in hierarchy that is rooted in the same core component platform as the original plug-in hierarchy. We applied both types of extensions to our scenario. Additionally, we used five experts to evaluate the flexibility of component hierarchy we designed using the proposed method. They evaluated flexibility by extending the component design to meet new requirements: one involved extending an existing plug-in hierarchy of the flexible component, and the other involved adding a new plug-in hierarchy to an existing core component platform (the two possible dimensions we describe above).

4.2.1 Extension Along an Existing Functional Dimension

As we discuss in Section 4.2, one can extend a platform component along an existing functional dimension (e.g., in our example, by adding another mode to the transportation function). Figure 9 illustrates how one can add a new plug-in to an existing plug-in hierarchy. With respect to the reservation platform example (Figure 6), it means that one can expand the set of supported industries by adding a new industry plug-in. To support this claim, we demonstrate that we can add a new plug-in to the reservation platform component to support a new domain (cruise reservation) that differs from the original four domains we used to design the platform component. The new cruise reservation plug-in provides the support for cruise reservation by adding functionality required specifically for cruise reservations. The cruise and the cabin classes of the new plug-in provide this functionality. Appendix C provides the requirements specification for the cruise reservation. The new cruise reservation plug-in extends the plug-in hierarchy below the lowest plug-in level that fits the domain of the extension. In the case of the cruise reservation, we extended the transport reservation with a level 2 plug-in because a cruise ship is a means of transportation that shares functionality with the design modeled in the transport reservation plug-in and, hence, can leverage this design (Figure 10).

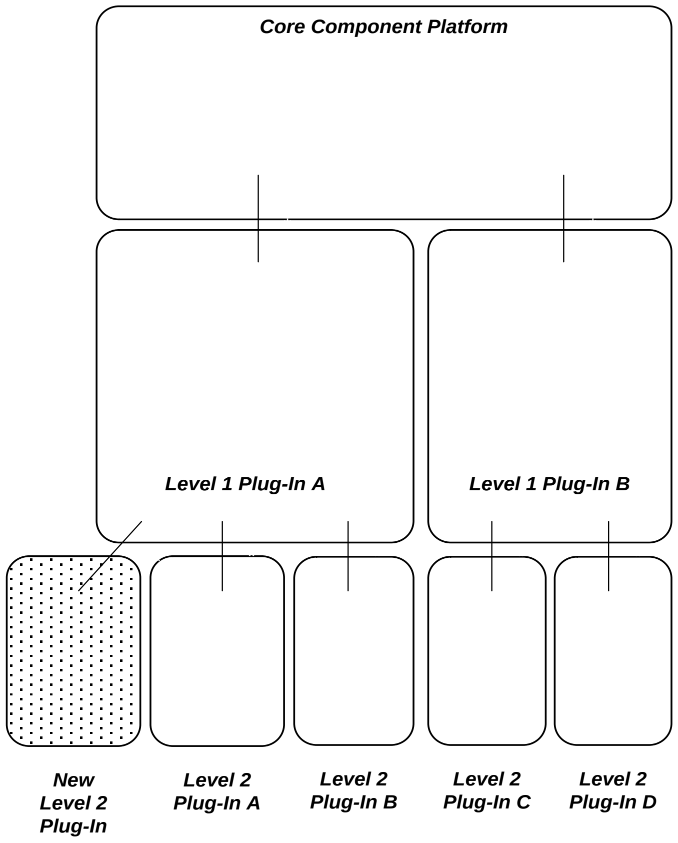

Figure 9. New plug-in Added to an Existing Plug-in Hierarchy

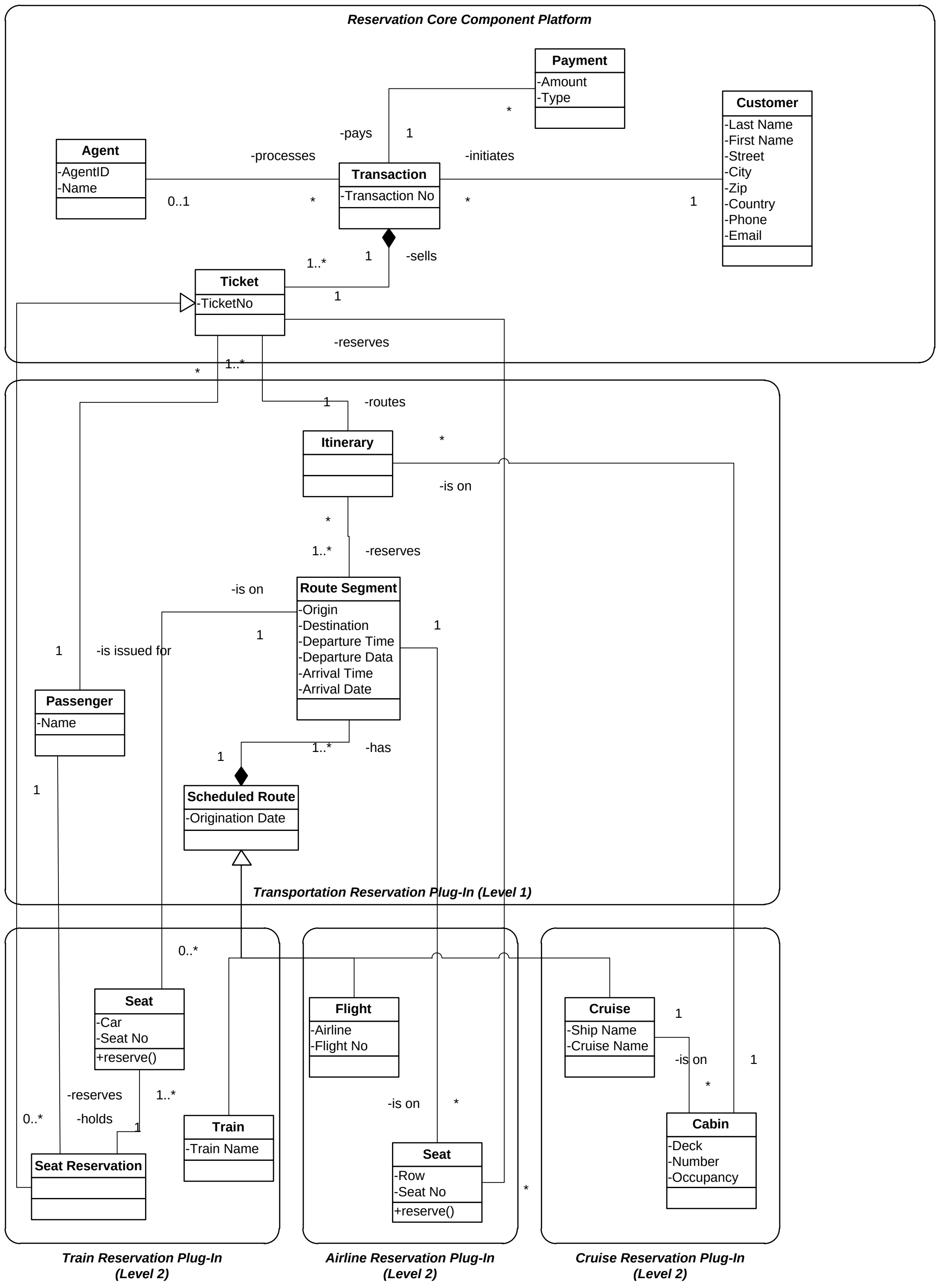

Figure 10. Platform-based Reservation Component with Cruise Reservation Extension

To further evaluate how easy it is to extend a platform component, we asked five software development experts to assess how one would need to extend the platform-based component (Figure 6) to meet the requirements of the cruise reservation. We purposefully selected the experts based on their relevant experience (Miles & Hubermann, 1994). They had worked for four to 20 years in software development and on between two and 30 software development projects each; all had experience with object-oriented modeling. Further, two experts worked in academia and three worked in industry in senior development positions. We provided each expert the four sets of scenario requirements that we used to develop the domain model and the individual component designs: the train reservation, the airline reservation, the show reservation, and the sporting event reservation system (Appendices C and B provide the requirements and the individual component designs, respectively). Further, we gave them a version of the flexible platform component (Figure 6), and we removed the plug-ins' names so they did not direct respondents to the applicable plug-ins. We gave each expert the option to either add a new plug-in hierarchy or to add a new plug-in to an existing hierarchy to support cruise reservation functionality (i.e., the two ways in which one can extend a platform component). All experts independently and correctly identified that one must add an additional plug-in on level 2 below the transport reservation level 1 plug-in. All participants confirmed that it is feasible to make such extensions in the framework of the platform component. As such, since the formal demonstration and the expert assignment support the extensibility along an existing functional dimension, we conclude that the platform component design provides such flexibility.

4.2.2 Extension Using a New Functional Dimension

If the existing platform hierarchy does not include one's desired new functionality, then one can add a new functional dimension as an additional plug-in hierarchy. Thus, one can add functionality to a core component platform that its designer did not anticipate without changing it. Figure 11 illustrates how a new plug-in hierarchy ties into an existing platform component.

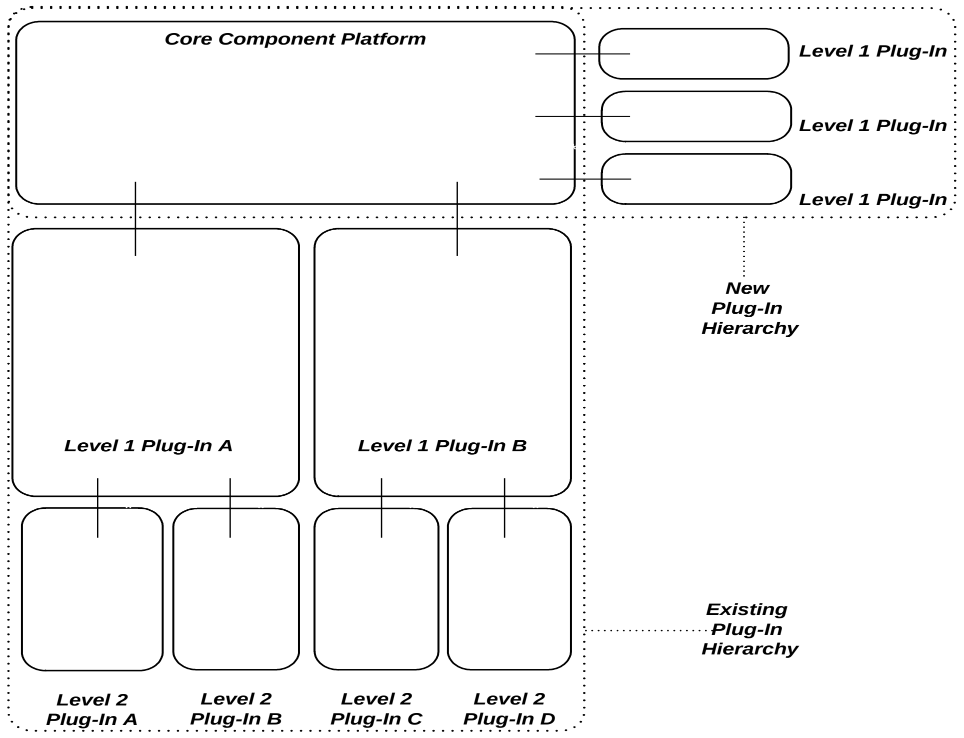

Figure 21. New Hierarchy Added to the Core Component Platform

In the context of the reservation platform component, one can add different types of booking agents as an afterthought if this addition does not affect the original core component platform. Figure 12 shows this addition, which is possible because the different types of booking agents specializes an existing platform class without modifying the original platform component design.

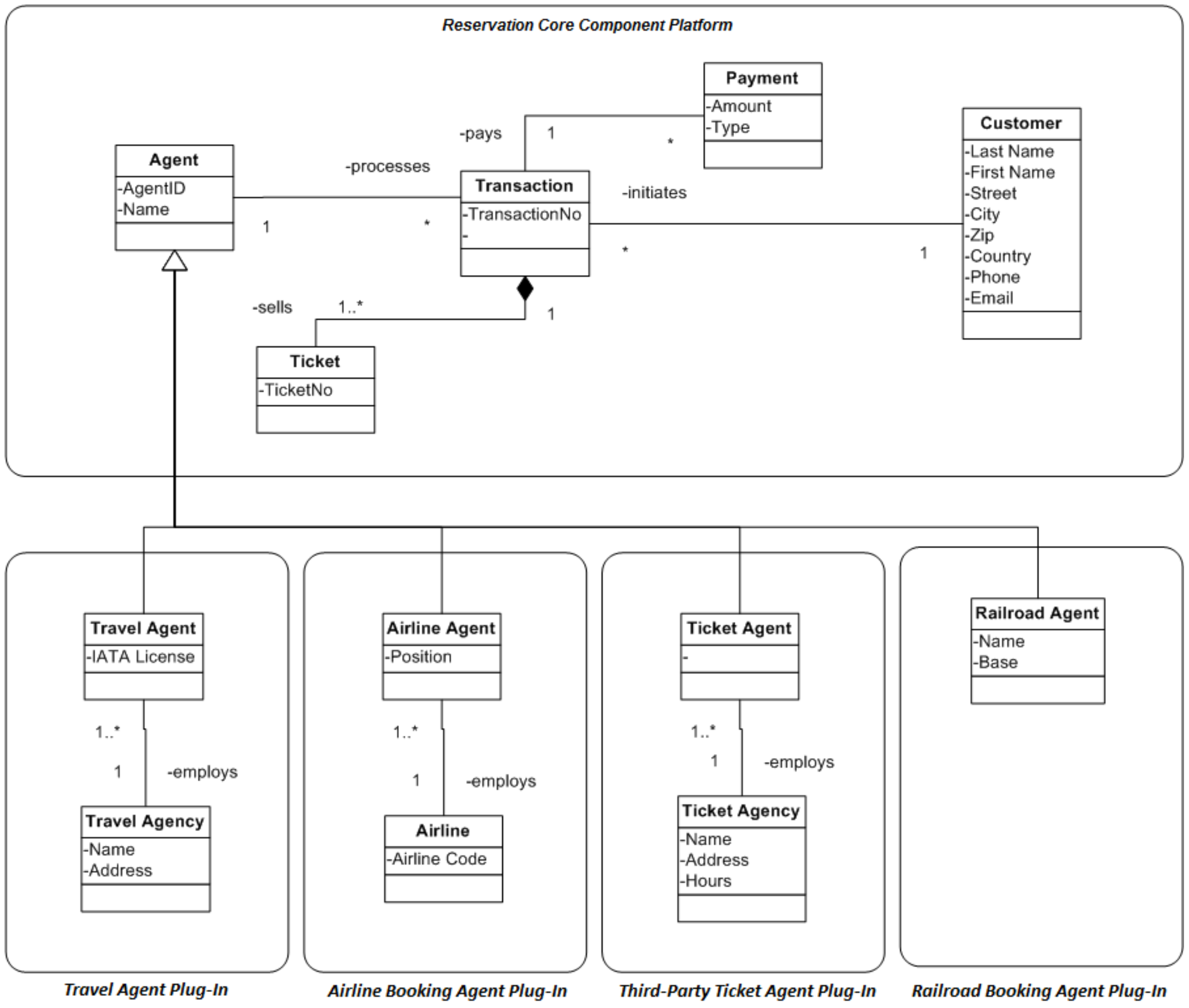

Figure 32. Reservation Core Component Platform with Booking Agent plug-in Extension

To evaluate the ease of extending the platform component in this dimension, we asked the same experts to provide ways of extending the reservation platform component (Figure 6) to meet the requirements of supporting multiple distribution channels in the form of airline agents and travel agents (compared to the demonstration above, we limited the scope in which the experts could extend the component to airline reservation and two booking agents to help them focus on the task). They had the option to either add another plug-in hierarchy or to add a plug-in to an existing hierarchy (the two ways one can extend a platform component). Four out of the five participants correctly identified that one must add a new plug-in hierarchy to the existing platform component; they also confirmed that it is feasible to meet the new requirements by extending the platform component with a new plug-in hierarchy. One participant did not respond to the questions pertaining to this set of requirements. Since both the formal assessment and the results of the expert assignment support the extensibility with a new functional dimension, we conclude that the platform component design provides such flexibility.

4.2.3 Participant Feedback

In addition to demonstrating the flexibility by using the earlier example and having the experts interact with different tasks to extend the functionality of a specific component hierarchy, we also solicited feedback from the experts on the platform component design after they interacted with it. The experts provided answers on a Likert scale (1: strongly disagree, 2: disagree, 3: neutral, 4: agree, and 5: strongly agree) to questions relating to the efficacy of the approach (Table 2). As Table 2 shows, the expert panel found that the platform component design and the plug-ins are “easy to understand”, that one can easily select appropriate plug-ins to reuse or modify, and that the platform-based component design is useful. While the sample size is small and the feedback is subjective, the results are encouraging.

Table 2. Expert Panel Evaluation Scores

Question	Average score
I found the platform component design and the plug-ins easy to understand.	4.4
The design of four components to satisfy given requirement makes sense.	4
It was easy to select the appropriate plug-in(s) to reuse or modify.	4.4
It was easy to design plug-in(s) to support unfilled requirements.	3.8
I believe that the platform-based component design is useful.	4

4.3 Evaluation Summary

We show that the design that a platform component hierarchy represents supports the same functionality as the domain models one develops it from and that one can extend a platform component in ways that individual component designs cannot. Thus, we conclude that the method that creates platform component with these properties, which is the artifact of this design research, meets the objectives that motivated our research.

5 Conclusions and Future Work

We develop an innovative method for component design that allows one to develop more flexible reusable components. We evaluated the method using several scenarios, demonstrations, and expert assessment. We found that method works and that it improves the flexibility of reusable components without requiring developers to invest time in thoroughly understanding the implementation details of the existing components. Thus, we believe that our approach can move reuse forward from an infrastructure-centered reuse paradigm to domain-specific reuse. Through the availability of more flexible components, software developers can derive increased benefits from component-based software development.

As we mention in Section 1, reusing services in the context of a service-oriented architecture (e.g., Web services) is conceptually similar to component-based reuse. Although one does not reuse software artifacts in a service-oriented architecture by calling up locally hosted compiled code but by sending messages to remotely hosted services, the challenge of maximizing reuse opportunities applies to both paradigms. Thus, services can also benefit from increased design flexibility. Service providers may be able to apply the platform design concepts we present in this study to hosted services if they can account for the differences in interfacing between a service platform and service plug-ins. The availability of platform-based services may also change how service consumers obtain, retrieve, and possibly extend services. Thus, future research could apply our method to service-oriented architecture. Such work will need to develop methods and standards for interfacing related services in a platform and investigate how to integrate such platform-based services into a service composition.

References

Baresi, L., Guinea, S., & Pasquale, L. (2012). Service-oriented dynamic software product lines. *IEEE Computer*, *45*(10), 42-48.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, *39*(10), 104-116.

Bell, P. (2007). A practical high volume software product line. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications Companion* (pp. 994-1003).

Bosch, J. (2000a). *Design and use of software architectures: Adopting and evolving a product-line approach*. Boston: Addison-Wesley.

Bosch, J. (2000b). Organizing for software product lines. In *Proceedings of the 3rd International Workshop on Software Architectures for Product Families* (pp. 117-134).

Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, Antonio, & Hinchey, M. (2014). An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software*, *91*, 3-23.

Chastek, G., Donohoe, P., & McGregor, J. D. (2002). *Product line production planning for the home integration system example* (Technical Note CMU/SEI-2002-TN-029). Pittsburgh, PA: Software Engineering Institute.

Chengjun, W. (2008). Pattern oriented service development for coarse-grained service reuse. In *Proceedings of the International Symposium on Knowledge Acquisition and Modeling* (pp. 832-836).

Chu, W., & Qian, D. (2009). Design web services: Towards service reuse at the design level. *Journal of Computers*, *4*(3), 193-200.

Clements, P. C., Jones, L. G., McGregor, J. D., & Northrop, L. M. (2006). Getting there from here: a roadmap for software product line adoption. *Commun. ACM*, *49*(12), 33-36.

Crnkovic, I., Stafford, J., & Szyperski, C. (2011). Software components beyond programming: From routines to services. *IEEE Software*, *28*(3), 22–26.

Czarnecki, K. (2007). Software reuse and evolution with generative techniques. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*.

Dhungana, D., Rabiser, R., Grnbacher, P., & Neumayer, T. (2007). Integrated tool support for software product line engineering. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*.

Donohoe, P. (2000). *Software product lines: Experience and research directions*. Boston: Kluwer Academic Publishers.

Due, R. (2000). The economics of component-based development. *Information Systems Management*, *17*(1), 92-95.

Erwig, M., & Fu, Z. (2005). Software reuse for scientific computing through program generation. *ACM Transactions on Software Engineering and Methodology*, *14*(2), 168-198.

Favaro, J. M., Favaro, K. R., & Favaro, P. F. (1998). Value based software reuse investment. *Annals of Software Engineering*, *5*, 5-52.

Fey, D., Fajta, R., & Boros, A. (2002). Feature modeling: A meta-model to enhance usability and usefulness. In G. Chastek (Ed.), *Second software product line conference* (pp. 198-216). Berlin: Springer.

Forte, M., Claudino, R. A. T., de Souza, W. L., do Prado, A. F., & Santana, L. H. Z. (2007). A component-based framework for the internet content adaptation domain. In *Proceedings of the 2007 ACM Symposium on Applied Computing*.

Gill, N. S. (2003). Reusability issues in component-based development. *SIGSOFT Software Engineering Notes*, *28*(4), 4-4.

Gill, N. S. (2006). Importance of software component characterization for better software reusability. *SIGSOFT Software Engineering Notes*, *31*(1), 1-3.

Girbea, A., Suciu, C., Nechifor, S., & Sisakm, F. (2013). Design and implementation of a service-oriented architecture for the optimization of industrial applications. *IEEE Transactions on Industrial Computing*, *10*(1), 185-196.

Griss, M. L., Favaro, J., & d'Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse* (pp. 76-85).

Heinecke, H., Damm, W., Josko, B., Metzner, A., Kopetz, H., Sangiovanni-Vincentelli, A., & Di Natale, M. (2008). Software components for reliable automotive systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*.

Heineman, G. T. (2000). A model for designing adaptable software components. *SIGSOFT Software Engineering Notes*, *25*(1), 55-56.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, *28*(1), 75-105.

Hissam, S. A., Seacord, R. C., & Lewis, G. A. (2002). Building systems from commercial components. In *Proceedings of the 24th International Conference on Software Engineering*.

Holmes, R., & Walker, R. J. (2012). Systematizing pragmatic software reuse. *ACM Transactions on Software Engineering and Methodology*, *21*(4), 1-44.

Hong, S.-J., & Lerch, F. J. (2002). A laboratory study of consumers' preference and purchasing behavioe with regards to software components. *The Data Base for Advances in Information Systems*, *33*(3), 23-37.

Hopkins, J. (2000). Component primer. *Communications of the ACM*, *43*(10), 27-30.

Jacobson, I., Griss, M. L., & Jonsson, P. (1997). *Software Reuse: Architecture, process and organization for business success*. New York, NY: Addison-Wesley Publishing Company.

Jain, H. K., & Vitharana, P. (2000). Research issues in testing business components. *Information & Management*, *37*(5), 297-309.

Jain, H., Rothenberger, M., & Sugumaran, V. (2006). Flexible software component design using a product platform approach. In *Proceedings of the International Conference on Information Systems*.

Jalender, B., Govardhan, A., & Premchand, P. (2010). A pragmatic approach to software reuse. *Journal of Theoretical and Applied Information Technology*, *14*(2), 87-96.

Kang, K. C., Lee, J., & Donohoe, P. (2002). Feature-oriented product line engineering. *IEEE Software*, *9*(4), 58-65.

Kang, K. C., Lee, K., & Lee, J. (2003). Feature oriented product line software engineering: Principles and guidelines. In *Domain oriented systems development: Practices and perspectives* (pp. 19-36). London: Taylor & Francis.

Kharb, L., & Singh, R. (2008). Complexity metrics for component-oriented software systems. *SIGSOFT Software Engineering Notes*, *33*(2), 1-3.

Krueger, C. W. (2006). New methods in software product line practice. *Communications of the ACM*, *49*(12), 37-40.

Laguna, M. A., Gonzlez-Baixauli, B., & Marques, J. M. (2007). Seamless development of software product lines. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*.

LaMantia, M. J. (2006). *Dependency models as a basis for analyzing software product platform modularity: A case study in strategic software design rationalization* (master's thesis). Massachusetts Institute of Technology, Boston.

Lau, K.-K. (2006). Software component models. In *Proceedings of the 28th International Conference on Software Engineering*.

Li, G., Muthusamy, V., & Jacobson, H. A. (2010). A distributed service-oriented architecture for business process execution. *ACM Transactions on the Web*, *4*(1), 1-33.

Manolios, P., Vroon, D., & Subramanian, G. (2007). Automating component-based system assembly. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*.

Metzger, A., & Pohl, K. (2007). Variability management in software product line engineering. *In Proceedings of the 29th International Conference on Software Engineering*.

Meyer, M. H., & Lehnerd, A. P. (1997). *The power of product platforms: Building value and cost leadership*. New York: Free Press.

Meyer, M. H., & Seliger, R. (1998). Product platforms in software development. *Sloan Management Review*, *40*(1), 61-74.

Miles, M. B., & Hubermann, A. M. (1994). *Qualitative analysis: A sourcebook of new methods*. Newbury Park, CA: Sage.

Mohagheghi, P., & Conradi, R. (2008). An empirical investigation of software reuse benefits in a large telecom product. *ACM Transactions on Software Engineering and Methodology*, *17*(3), 1-31.

Nazareth, D., & Rothenberger, M. A. (2004). Assessing the cost-effectiveness of software reuse: A model for systematic Reuse. *Journal for Systems and Software*, *73*(2), 245-255.

Neighbors, J. M. (1984). The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, *SE10*(5), 564-574.

Orriens, B., & Yang, J. (2008). Service componentization: Toward service reuse and specialization. In D. Georgakopoulos & M. Papazoglou (Eds.), *Service-oriented computing* (pp. 295-330). Cambridge, MA: MIT Press.

Park, S., Park, S., & Sugumaran, V. (2007). Extending reusable asset specification to improve software reuse. In *Proceedings of the 2007 ACM Symposium on Applied Computing*. Seoul, Korea: ACM.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2008). A design science research methodology for information systems research. *Journal of Management Information Systems*, *24*(3), 45-77.

Poulin, J. S., Caruso, J. M., & Hancock, D. R. (1993). The business case for software reuse. *IBM Systems Journal*, *32*(4), 567-585.

Ramachandran, M. (2005). Software reuse guidelines. *SIGSOFT Software Engineering Notes*, *30*(3), 1-8.

Rodriguez, J. M., Crasso, M., Mateos, C., Zunino, A., & Campo, M. (2013). Bottom-up and top-down cobol systems migration to Web services. *IEEE Internet Computing*, *17*(2), 44-51.

Rosenmller, M., Siegmund, N., Saake, G., & Apel, S. (2008). Code generation to support static and dynamic composition of software product lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*.

Sääksjärvi, M. (2002). Software application platforms: From product architecture to integrated application strategy. In *Proceedings of the 26th Annual International Computer Software and Applications Conference*.

Salonen, P. I., & Sääksjärvi, M. (2004). *Evaluation of a product platform strategy for analytical application software*. Helsinki School of Economics, Helsinki, Finland.

Salvador, F., Forza, C., & Rungtusanatham, M. (2002). Modularity, product variety, production volume, and component sourcing: Theorizing beyond generic prescriptions. *Journal of Operations Management*, *20*(5), 549-575.

Scott, A. H., Robert, C. S., & Grace, A. L. (2002). Building systems from commercial components. In *Proceedings of the 24th International Conference on Software Engineering*.

Sharp, J. H., & Ryan, S. D. (2010). A theoretical framework of component-based software development phases. *The Data Base for Advances in Information Systems*, *41*(1), 56-75.

Silic, M., Delac, G., Krka, I., & Srbljic, S. (2013). Scalable and accurate prediction of availability of aromic Web services. *IEEE Transactions on Services Computing*.

Sinha, A., & Jain, H. (2013). Ease of reuse: An empirical comparison of components and objects. *IEEE Software*, *30*(5), 70-75.

Slyngstad, O. P. N., Gupta, A., Conradi, R., Mohagheghi, P., Ronneberg, H., & Landre, E. (2006). An empirical study of developers views on software reuse in statoil ASA. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*.

Sprott, D. (2000). Componentizing the enterprise application packages. *Communications of the ACM*, *43*(4), 63-69.

Sugumaran, V., Park, S., & Kang, K. C. (2006). Software product line engineering. *Communications of the ACM*, *49*(12), 28-32.

Sugumaran, V., Tanniru, M., & Storey, V. C. (2008). A knowledge-based framework for extracting components in agile systems development. *Information Technology & Management*, *9*(1), 37-53.

Szyperski, C. A. (1998). Emerging component software technologies—a strategic comparison. *Software—Concepts and Tools*, *19*(1), 2-10.

Tolvanen, J.-P., Gray, J., Rossi, M., & Sprinkle, J. (2008). The 8th OOPSLA workshop on domain-specific modeling. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. New York, NY: ACM Press.

Vici, A. D., Argentieri, N., Mansour, A., d'Alessandro, M., & Favaro, J. (1998). FODAcom: An experience with domain analysis in the Italian telecom industry. In *Proceedings of the 5th International Conference on Software Reuse* pp. 166-175).

Vitharana, P., Jain, H., & Zahedi, F. M. (2004). Strategy-based design of reusable business components. *IEEE Transactions on Systems, Man and Cybernetics—Part C, Applications and Reviews*, *34*(4), 460-475.

Vitharana, P., Zahedi, F. M., & Jain, H. (2003a). Knowledge-based repository scheme for storing and retireving businesss components: A theoretical design and an empirical analysis. *IEEE Transactions on Software Engineering*, *29*(7), 649-664.

Vitharana, P., Zahedi, F. M., & Jain, H. K. (2003b). Design retrieval and assembly in component based software development. *Communications of the ACM*, *46*(11), 97-102.

Welke, R., Hirschheim, R., & Schwarz, A. (2011). Service-oriented architectur maturity. *IEEE Computer*, *44*(2), 61-67.

White, J., Galindo, J. A., Saxena, T., Dougherty, B., Benavides, D., & Schmidt, D. C. (2014). Evolving feature model configurations in software product lines. *Journal of Systems and Software*, *87*, 119-136.

Volkswagen group A platform. (n.d.). In *Wikipedia.* Retrieved from http://en.wikipedia.org/wiki/PQ34

Yau, S. S., Ye, N., Sarjoughian, H. S., Dazhi, H., Roontiva, A., Baydogan, M., & Muqsith, M. A. (2009). Towards development of adaptive service-based software systems. *IEEE Transactions on Services Computing*, *2*(3), 247-260.

Appendix A: CreatePlugIn() Specifications

Table A2. Formal Specification of CreatePlugIn(SD)

1. $S_C = \cup C_{ji} \mid \forall C_{ji} \in D_i \mid \forall D_i \in S_D$

2. $\forall C_j' \subseteq C_j \mid C_j \in S_C \quad : \left(\forall D_i \in S_D : C_j' \in D_i \right)$

   a. For $C_j' \subseteq C_j \mid C_j \in S_C$ contained in the most $D_i \in S_D$

   where $\neg\exists \left(C_j'' \subseteq C_j \mid C_j'' \supset C_j' \right)$ that is contained in the same *D*

$$\Rightarrow S_D' \subset S_D : \left(\forall D_i \in S_D' : C_j' \in D_i \right) \wedge S_D' \notin S_D$$

   b. The next PLUG-IN will be a child of P_k

   c. Execute CreatePlugIn(S_D')

where:	S_D	Set of Individual Component Designs
	D_i	An Individual Component Design $\left(D_i \in S_D \right)$
	S_C	Set of Classes
	C_j	Classes $\left(C_j \in S_C \right)$
	C_{ji}	Classes $\left(C_{ji} \in D_i \right)$
	P_k	Platform-Based Component Plug-In (Highest Level Plug-In is the Component Platform)

Appendix B: Individual Component Designs

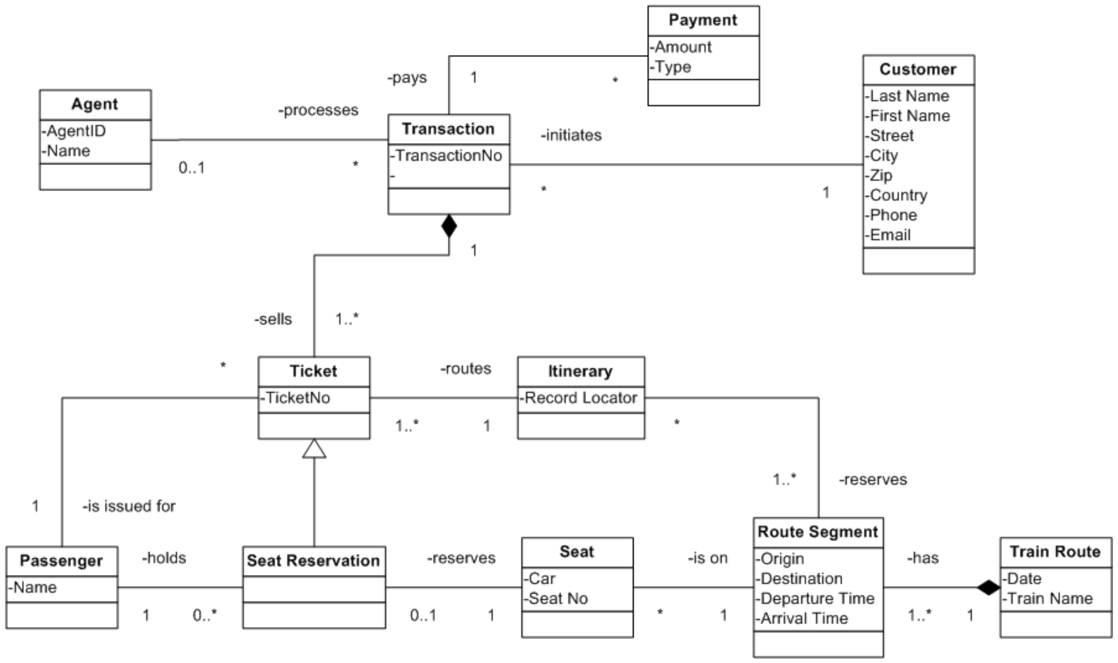

Figure B1. Component Design for Train Reservation

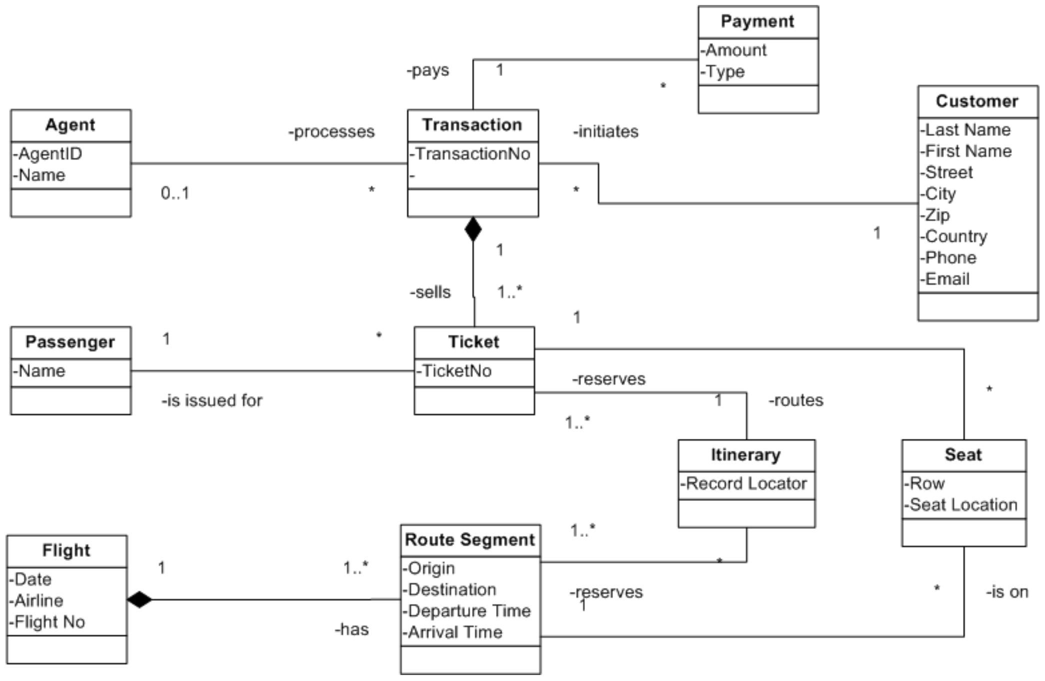

Figure B2. Component Design for Airline Reservation

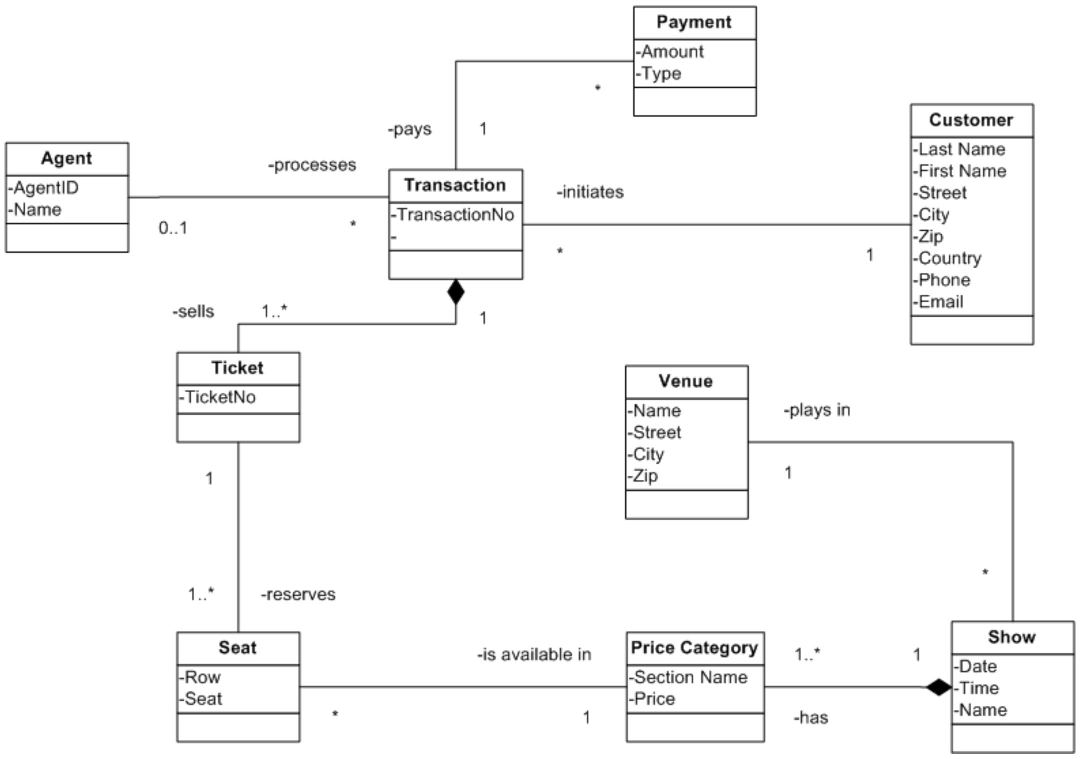

Figure B3. Component Design for Show Reservation

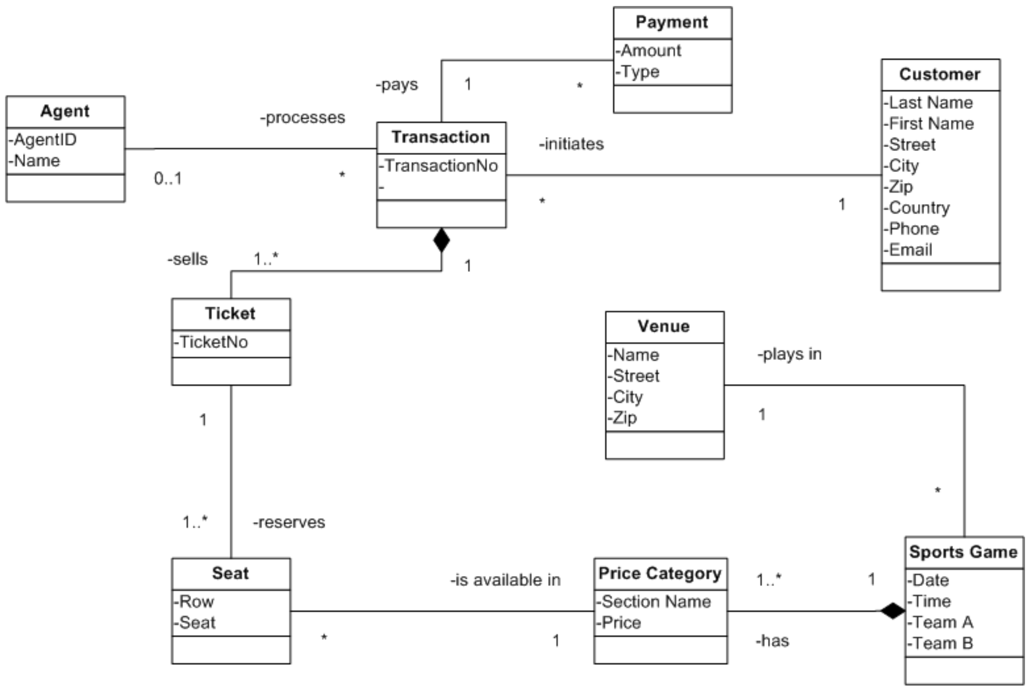

Figure B3. Component Design for Sporting Event Reservation

Appendix C: Requirements Specifications of the Individual Domain Models

Train Reservation

- Customer initiates transaction for train ticket reservations.
- A booking agent can process a transaction.
- Payments are recorded for each transaction.
- A transaction facilitates the sale of one or many tickets.
- Each ticket authorizes one passenger to travel.
- Itineraries specify the routing for tickets. A routing consists of multiple route segments. Multiple tickets (for different passengers) can be on one itinerary as long as they have the same routing.
- A train route consists of multiple route segments on a specific day. It is uniquely identified by the train name and the origination date.
- Seat reservations are optional and can be made on individual route segments. Seat reservations **do not require a ticket purchase and can be made independently from any ticket purchase for transportation (**e.g., if a passenger is not sure which train he/she will take, he/she can make reservations on multiple trains without having to purchase a ticket for each train). Each seat reservation is for a specific passenger.

Airline Reservation

- Customer initiates transaction for airplane ticket reservations.
- A booking agent can processes a transaction.
- Payments are recorded for each transaction.
- A transaction facilitates the sale of one or many tickets.
- Each ticket authorizes one passenger to travel.
- Itineraries specify the routing for tickets. A routing consists of multiple route segments. Multiple tickets (for different passengers) can be on one itinerary as long as they have the same routing.
- A flight consists of multiple route segments on a specific day. It is uniquely identified by the airline, through the flight number, and the origination date.
- Seat reservations are optional and can be made for each route segment. Seat reservations require a ticket and are for the passenger on that ticket.

Show Reservation

- Customer initiates transaction for show ticket reservations.
- A booking agent can processes a transaction.
- Payments are recorded for each transaction.
- A transaction facilitates the sale of one or many tickets.
- A show plays in a venue.
- A show is uniquely identified by the name, the date, and the time.
- A show has multiple price categories that may vary from show to show.
- Seats are available in specific price categories.

- Each ticket reserves one seat in a specific price category of the show. Tickets are not issued for specific guests, but are good for anyone who holds them.
- Seat reservations are mandatory and are part of a ticket reservation. The price category of the seat determines the ticket price.

Sports Game Reservation

- Customer initiates transaction for sports game ticket reservations.
- A booking agent can processes a transaction.
- Payments are recorded for each transaction.
- A transaction facilitates the sale of one or many tickets.
- A sports game takes place in a venue.
- A spots game is uniquely identified by the two team names, the date, and the time.
- A sports game has multiple price categories that may vary from game to game.
- Seats are available in specific price categories.
- Each ticket reserves one seat in a specific price category of the sports game. Tickets are not issued for specific guests, but are good for anyone who holds them.
- Seat reservations are mandatory and are part of a ticket reservation. The price category of the seat determines the ticket price.

Cruise Ship Reservation (for the Evaluation)

- Customer initiates transaction for cruise ship reservations.
- A booking agent can processes a transaction.
- Payments are recorded for each transaction.
- A transaction facilitates the sale of one or many tickets.
- Each ticket authorizes one passenger to take a cruise.
- Multiple tickets for different passengers that share a cabin for the cruise will be on one itinerary.
- Itineraries specify the routing for cruise tickets. A routing consists of multiple route segments.
- A cruise consists of multiple route segments. It is uniquely identified by the ship name, cruise name, and origination date.

Cabin reservations are mandatory and are made for the entire cruise (all segments). One cabin is being reserved for multiple passengers on one itinerary. (Reservations of multiple cabins would require multiple itineraries.)

About the Authors

Marcus A. Rothenberger is Professor of MIS in the Department of Management, Entrepreneurship, and Technology of the Lee Business School at the University of Nevada Las Vegas. Previously, he was faculty at the University of Wisconsin-Milwaukee. He holds a PhD in Information Systems from Arizona State University. His work includes theory testing, theory development, and design science research in the areas of software process improvement, software reusability, performance measurement, and the adoption of Enterprise Resource Planning systems.

Hemant Jain is W. Max Finley Chair in Business, Free Enterprise and Capitalism and Professor of Business Analytics, in College of Business at University of Tennessee Chattanooga. Before this, he was Professor of Information Technology Management at University of Wisconsin Milwaukee. His research interests are in data analytics, real time organizations, component based development, service-oriented architecture and healthcare informatics. He received his PhD in information system from Lehigh University, a Master of Technology from the Indian Institute of Technology, Kharagpur, India, and Bachelor of Engineering from University of Indore, India.

Vijayan Sugumaran is Professor of Management Information Systems and Chair of the department of Decision and Information Sciences at Oakland University, Rochester, Michigan, USA. His research interests are in the areas of Component Based Software Development, Ontologies and Semantic Web, Intelligent Agent and Multi-Agent Systems, and Data & Information Modeling. He is the editor-in-chief of the *International Journal of Intelligent Information Technologies* and serves on the editorial board of seven other journals. He is the Chair of the Intelligent Systems Track for Americas Conference on Information Systems (AMCIS 1999-2017). He has served as the program co-chair for the International Conference on Applications of Natural Language to Information Systems (NLDB 2008, NLDB 2013 and NLDB 2016).

JITTA

JOURNAL OF INFORMATION TECHNOLOGY THEORY AND APPLICATION

JITTA is a Publication of the Association for Information Systems
ISSN: 1532-3416