Association for Information Systems
AIS Electronic Library (AISeL)

ICIS 2006 Proceedings

International Conference on Information Systems (ICIS)

December 2006

Flexible Software Component Design Using A Product Platform Approach

Hemant Jain
University of Wisconsin, Milwaukee

Marcus Rothenberger
University of Nevada Las Vegas

Vijayan Sugumaran
Oakland University

Follow this and additional works at: http://aisel.aisnet.org/icis2006

Recommended Citation

Jain, Hemant; Rothenberger, Marcus; and Sugumaran, Vijayan, "Flexible Software Component Design Using A Product Platform Approach" (2006). *ICIS 2006 Proceedings*. 15.
http://aisel.aisnet.org/icis2006/15

This material is brought to you by the International Conference on Information Systems (ICIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in ICIS 2006 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

FLEXIBLE SOFTWARE COMPONENT DESIGN USING A PRODUCT PLATFORM APPROACH*

Design Science

Hemant Jain
Sheldon B. Lubar School of Business
University of Wisconsin - Milwaukee
Milwaukee, WI
jain@uwm.edu

Marcus A. Rothenberger
Department of MIS
University of Nevada Las Vegas
Las Vegas, NV
marcus.rothenberger@unlv.edu

Vijayan Sugumaran
Department of Decision and Information Sciences
Oakland University
Rochester, MI
sugumara@oakland.edu

Abstract

The concept of reusing software artifacts to improve development efficiency and software quality has been around for quite some time, yet it has focused mostly on in-house reuse with limited success. Recently developments in component-based software development, Web services, and service-oriented architecture are targeting inter-organizational reuse by promoting black-box reuse facilitated by standard Web service-based interface definitions. This reuse paradigm reduces dramatically the cost of component integration and maintenance, as it is no longer necessary to understand implementation details. However, this requires a very high level of standardization and clear functional definitions to facilitate retrieval through search engines. Because of this, its application has been limited to relatively small infrastructure items, such as user interfaces, printing components, and data access modules. Further, the reuse potential of such components is high, as infrastructure functionality is needed across domains. However, without the ability to build the core of an application around reusable software components, the true value of component-based software development that can offer lower cost, high quality, agility, and responsiveness cannot be realized. Thus, in order to move this adoption philosophy forward, domain-specific components must be available. This research develops a method that promotes flexible component design based on a common product platform with derivative products. Following the design research method, the methodology artifact will be evaluated through an experimental evaluation and a formal assessment of its value to component-based development.

Keywords: Component design, software reuse, component-based development, design science

* Authors contributed equally. Authorship is in alphabetical order.

Introduction

From the early years of information technology, software has predominantly been made to order to fit specific organizational needs. Custom-developed software is made to the requirements of the client organization. This option usually ensures the best fit of the software to the existing organizational processes; thus, a company is not required to change the way it does business. On the down side, custom software is characterized by high cost, low quality, longer development time, and low agility.

Reuse of software artifacts has been frequently discussed over the past decade as a means of improving development efficiency and software quality, yet such discussion emphasized mostly in-house reuse with the limitation of a comparably small repository (Basili et al. 1996; Poulin et al. 1993). It is only recently that the development of component standards, such as CORBA and JavaBeans, has promoted the emergence of component markets that support the notion of inter-organizational reuse. This movement has been further enhanced by wide adoption of Web services standards and movement toward service-oriented architecture. As opposed to the traditional reuse notion, which requires developers to modify and customize retrieved components to fit the requirements of new software projects, component-based software development limits the customization of modules to the selection of appropriate parameters. This black-box reuse approach reduces the cost of component integration and maintenance, as it is no longer necessary to understand implementation details to reuse a component (Ravichandran and Rothenberger 2003).

However, because of the need for high functionality standardization, the availability and use of black-box components have mostly been limited to infrastructure items, such as user interfaces, printing components, and data access modules (Taft 2000). The functionality of infrastructure components is highly standardized and can be clearly defined for retrieval through search engines. Further, the reuse potential of such components is very high, as infrastructure functionality is needed across domains. Nevertheless, without the ability to build the core of an application around reusable software components, the true value of component-based software development cannot be obtained. Thus, in order to move this adoption philosophy forward, domain-specific components must be available for various domains.

This research investigates how domain-specific components can be structured to maximize their reuse potential. Product platforms have achieved in the physical world what we propose in this research for software components. Platform-based components are domain-specific components that are customizable (without code changes) to various needs. Thus, this concept increases the requirements flexibility of component-based development while retaining the benefit of black-box component design. Motivated by the product platform literature, we have developed a component design methodology that draws from the areas of domain analysis, domain modeling, component-based software development, and software reuse. The design approach will be formalized and evaluated according to the design science paradigm (Hevner et al. 2004; Peffers et al. 2006).

Background and Conceptual Development

Software Reuse

Although software productivity has steadily increased, the demand for software development is still very high. To address these demands, research has been conducted to build reusable artifacts such as data objects, design patterns, and pseudo code. Reuse of such artifacts has been suggested as the only way to achieve the low-cost, high- quality, and low-development time objectives of application development (Krueger 1992; Mili et al. 1995). Reuse involves generating new designs by combining high-level specifications and existing component artifacts (Setliff et al. 1993). Tools and techniques are also being developed to provide support for reuse-based software design (Nierstrasz and Meijler 1995; Purao and Storey 1997). A number of reusable artifacts have been developed, such as class libraries, components (Szyperski 1998), frameworks, and patterns. Research has also been undertaken that attempts to realize the benefits of reuse for object-oriented conceptual design through the creation of tools to facilitate design and construction of new systems with reuse (Purao and Storey 1997; Sugumaran et al. 2000). Higher-level design fragments and models are being developed (Han et al. 1999).

Domain Analysis and Modeling

Domain analysis is an activity similar to systems analysis, performed over an application domain. It involves analyzing existing systems in the application domain and creating a domain model that characterizes that application domain. An early definition of domain analysis and modeling provided by Neighbors is: "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" (Neighbors 1984). The primary objective of the domain modeling approach to software construction is to increase reuse, i.e., reuse not only of code modules but also of domain knowledge such as domain requirements, specifications, and designs. From the domain model, target systems can be generated by either tailoring the domain model, or by a combination of evolving the domain model and then tailoring it.

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. A computer-based domain model captures both the static and dynamic aspects of the application domain. The static aspects include object types of the domain, attributes of those object types (Meyer 1988), and relationships among them. The dynamic properties include the operations associated with object types, the state changes of object types, and the messages passed between object types. The domain model may also include integrity constraints that govern the behavior of object types in the domain. Thus, it is a key ingredient for the creation of reusable core assets (Kang et al. 2002).

Recently domain analysis methods such as feature-oriented approach (Kang et al. 2002), Reuse-Driven Software Engineering Business (RSEB) (Jacobson et al. 1997), FODAcom (Vici et al. 1998), FeatuRSEB (Griss et al. 1998), and Product Line Analysis (PLA) (Chastek 2002) have been used to analyze commonality and variability among products within a product line. In particular, the feature-oriented approach has been used extensively in industry and academia since the FODA method was introduced in 1990 by the Software Engineering Institute (Clements et al. 2005). The feature-oriented approach attempts to analyze commonality and variability in terms of features, and thus a feature-based model provides a basis for developing, parameterizing, and configuring reusable assets (Kang et al. 2002).

In the feature-oriented approach, 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 features associated with each individual product must be identified and defined (Bosch 2000). In mature and stable domains, the approach identifies the features of a given domain by analyzing the domain terminology and provides feature categories as a feature identification framework (Kang et al. 1998). Also, product features identified in MPP (marketing and product plan) are organized into an initial feature model (Kang et al. 2002). Although the approach suggests standardizing domain terminology and clarifying domain scope before feature identification, it is difficult to accomplish in immature or emergent domains because of the lack of availability of domain experts and supporting material. Further, in a very new or an envisioned market like an emerging market (Li 2002), it is practically impossible to identify any reference product that would match exactly the envisioned product line. Hence, the informational input such as domain knowledge will be very scarce for such domains. In addition, the approach is not appropriate when domain experts are not available, and features are not useful for exploring new or poorly understood system characteristics (Chastek 2002).

Both FODAcom (Vici and Argentieri, 1998) and FeatuRSEB (Griss et al. 1998) have used use-case models with feature models for C&V analysis. PLA (Chastek 2002) combines traditional object-based analysis with FODA for a product line analysis. However, these methods do not provide a systematic and concrete mechanism for feature identification and the rationale for features like the feature-oriented approach does.

Component-Based Software Development

There is general agreement in the software industry that component-based software development (CBD) technology will bring profound changes in systems development and delivery. Numerous advantages of CBD are touted. An information system developed from reusable components is argued to be more reliable (Vitharana and Jain 2000), to increase developer productivity (Lim 1994), to reduce skills requirement (Kiely 1998), to shorten the development life cycle (Due 2000), to reduce time-to-market (Lim 1994),to increase the quality of the developed system (Lim 1994; Sprott 2000), and to cut down the development cost (Due 2000). Beyond these operational benefits, CBSD also has been found 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 et al. 1998). 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 replace components with newer ones, containing additional features, in order to respond to competitive forces and changing market conditions.

Component-based software development is changing the way applications are being developed and delivered to the end user. It is causing a shift in software development paradigms, particularly with the development of several component architecture standards such as CORBA, COM, and EJB (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 (Hopkins 2000). For example, in an auction application domain, one component captures the characteristics of a bid and its associated processes. Another component deals with transaction processing. These can be combined to form a larger component that would be a reusable artifact.

Software Product Line

The software product line approach is recognized as a successful method for improving reuse in software development. The idea behind the product line concept is for organizations to develop a product family from reusable core assets rather than from scratch. It is different from the platform-based component design that we are proposing in this research, as it does not address the nature of the reusable component itself. Instead, the software product line approach provides a means to select and integrate individual components to build a product line-based application. The product line approach is receiving increased attention, specifically as software engineers or developers are faced with increasing pressure to produce software more quickly and economically. To identify and describe the right functionality to be encapsulated as reusable artifacts is a key challenge within the product line approach. In order to address this challenge, the key requirements for developing future products that drive the design of a product line need to be identified. Thus, a thorough requirements analysis for the product line, where particular common and variant requirements are systematically identified and described, must be performed. Furthermore, the requirements and commonality and variability (C&V) identified must satisfy an organization's high-level business goals, and thus, requirements analysis and C&V analysis must be carried out to satisfy these goals and provide the rationale for them.

Application Frameworks

An application framework is a reusable design for implementing a software system that can be expressed as a set of abstract classes (Johnson and Foote 1998). These classes can be specialized to produce custom applications. An application framework captures the standard structure of an application by bundling large amounts of reusable code. Over the years, several frameworks have been developed in areas such as user interface design, graphical editors, networks, financial applications, etc. (Fayad et al. 1999). Some examples of frameworks are MacApp, Model View Controller (MVC), Apache Struts, Java Server Faces (JSF), Django, Symfony, Java Native Interface (JNI), Microsoft's Distributed Common Object Model (DCOM), Common Object Request Broker Architecture (CORBA) etc.

Although many object-oriented application frameworks have been developed, they have had limited success for a number of reasons. One of the main ones is that frameworks have a steep learning curve. Developers have to understand the abstract designs of classes and how they interact with each other for different purposes. Class complexity and object collaboration complexity are often cited as the major obstacles for using application frameworks. In addition, large-scale reusable frameworks also fail due to lack of integratability, maintainability, efficiency, and standards (Fayad and Schmidt 1997). Frameworks are generally tied to a particular language or technology, and hence components from multiple frameworks cannot be combined easily (Johnson 1997). Demeyer et al. (1997) specify the following design guidelines for "tailorable" frameworks: interoperability, distribution, and extensibility. Our product platform-based approach for component design is based on these principles and accommodates multiple levels of granularity compared to other frameworks. Also, our proposed approach is independent of any application framework and can be used to create a new framework in a particular application domain. We also provide a formal specification for the platform design and plug-ins, which promotes interoperability and extensibility. Thus, our proposed methodology is flexible and agile.

Domain-Specific Component Reuse

In order to successfully sell domain-specific components, providers must ensure that the software components allow for sufficient flexibility to be integrated in a maximum number of applications. The most common approach to achieving this is the parameterization of components. This means that components are developed with a set of optional functionalities and choices that can be triggered by their parameters. The component user can select the options by setting the parameters accordingly (Barnes and Bollinger 1991). This method limits possible applications of the component to what the component developers anticipate, as only those applications will be supported with appropriate parameters.

An alternative method may address the lack of extensibility. Reusing smaller components (with less functionality per component) can allow software developers to more exactly choose components based on the requirements (Apte et al. 1990). If a component representing a particular aspect of the desired functionality is not available, it can be developed with little effort and integrated with the remaining components that meet the requirements. Nevertheless, this approach is problematic: since the components each represent a low development effort, the leverage of each reuse instance is small. Activities such as retrieval and parameterization will use a proportionally larger share of development time, thus increasing development cost.

Component flexibility can be increased over traditional parameterized components by building a platform for each component that allows the creation of derivative components by 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). Complex application can then be built from platform-based components. This approach is an alternative to traditional component design combining the retrievability and integration ease of parameterized components with extensibility of very small components. Figure 1 contrasts the benefits of the three component design approaches.

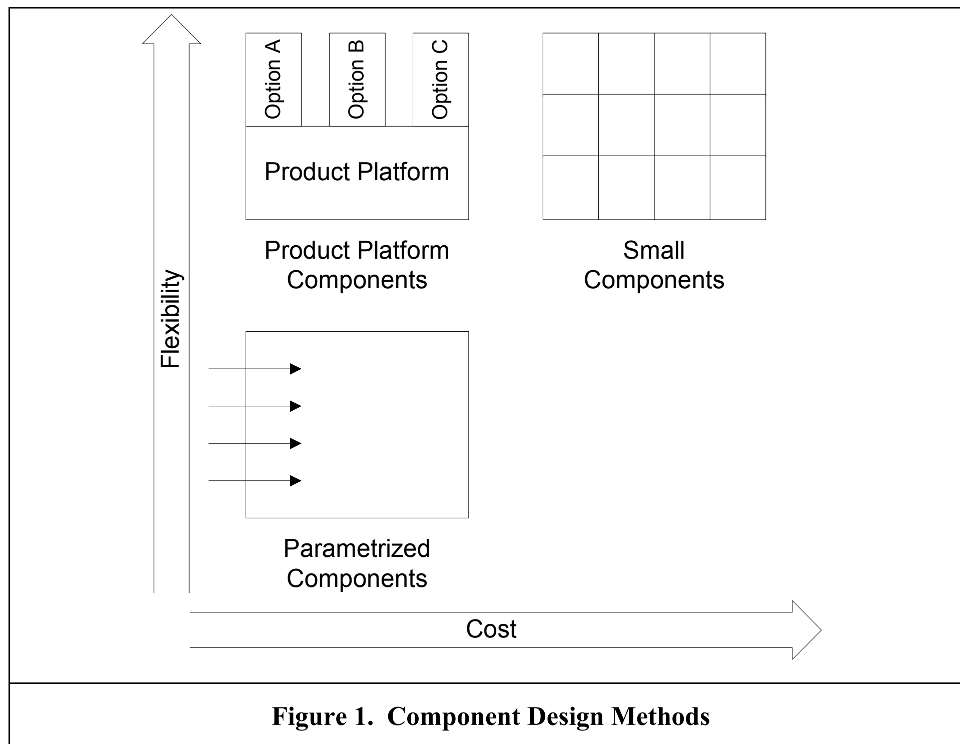


Figure 1. Component Design Methods

Product Platforms

In the physical world, the modular design of product platforms has been a means to increase product variety to better meet varying customer requirements (Salvador et al. 2002). For example, Hewlett-Packard's (HP) OfficeJet

platform combined the functions of previously distinct products, such as computer printer, fax machine, scanner, and photo copier, thus meeting customer demand in a flexible manner (Meyer and Lehnerd 1997). The lessons learned from the modular development of physical products may also apply to the development of software components; as in manufacturing, modularity promotes an increased fit between software and customer specifications by enabling the developers to combine and customize elements of the application, thus reducing the need to custom develop based on each client's specifications. In this research, we are developing a platform-based component design method as an alternative to existing component design options.

The Artifact

Platform Design through Functional Decomposition

The major challenge in platform-based component development is to design a component platform that unifies the functional requirements of the common component user base. Based on platform-development in manufacturing, the design of a platform must be guided by the customer demand (Meyer and Seliger 1998). In terms of component-platform development, this means that the projected market for the component must drive its design. We are using the Unified Modeling Language (UML) notation of the object-oriented design model to illustrate the development of a component platform. The proposed methodology consists of two phases: a) platform design and b) post-implementation modifications. The following subsections describe these phases in more detail.

Platform Design

For the design of a platform-based component, individual component models must be created to meet the requirements of each application domain in which the platform component that is to be developed will be used. As a next step, commonalities between the individual designs must be transferred into the platform and plug-in design hierarchy of the platform-based component. Finally, the design hierarchy may be evaluated and fine-tuned if necessary. The overall process diagram depicting this methodology is shown in Figure 2.

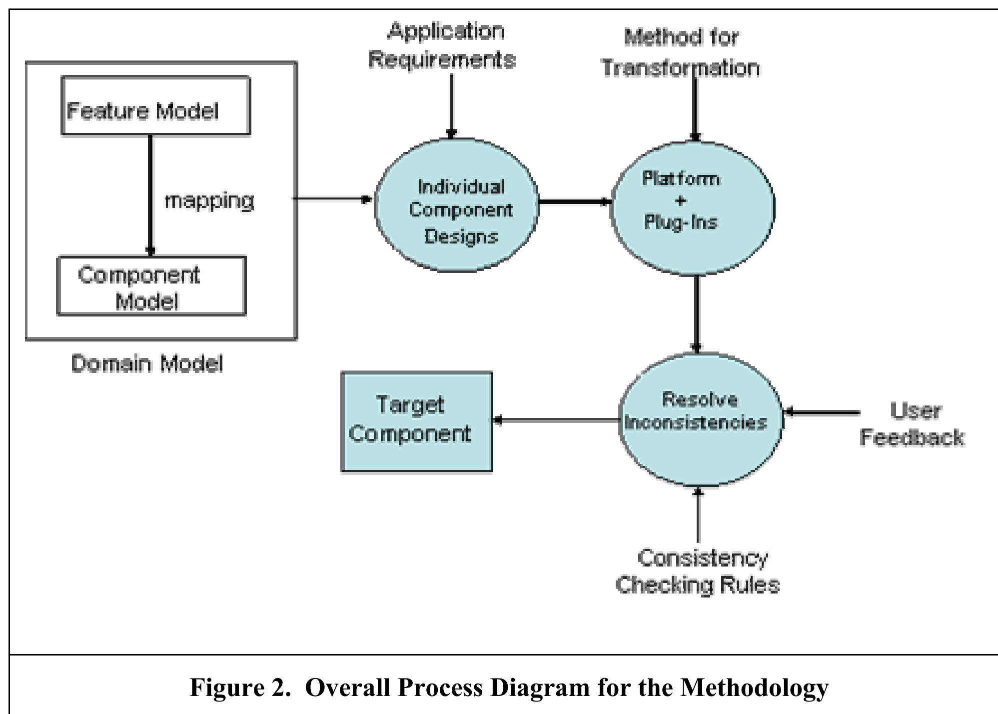


Figure 2. Overall Process Diagram for the Methodology

The paper presents a scenario of a platform-based reservation component that demonstrates the transformation process from individual components to a component platform design hierarchy. Figure 3 is the integrated reservation design hierarchy providing the functionality for reservations in different industries developed from a set of individual component designs, covering the domains of train, airlines, show, and sports game reservations. A component platform can be associated with multiple levels of plug-ins; the actual number of levels for a specific design is determined by the transformation methodology and the nature of the individual designs it converts. In this example, plug-ins are provided on two levels; the first level incorporates industry segment-specific functionality for transportation and event reservations, the second level provides the lowest level of industry functionality for train, airline, show, and sports game reservations.

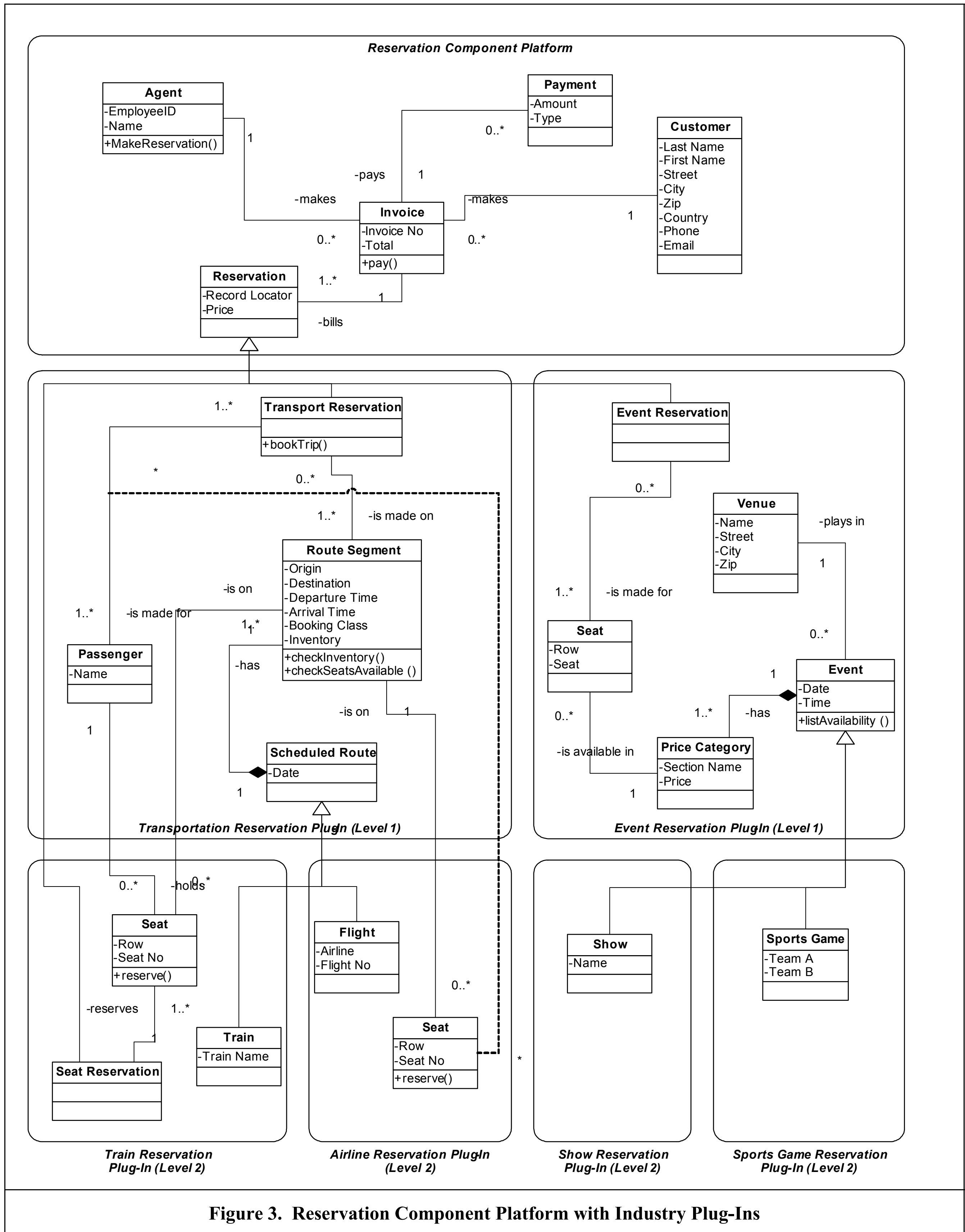
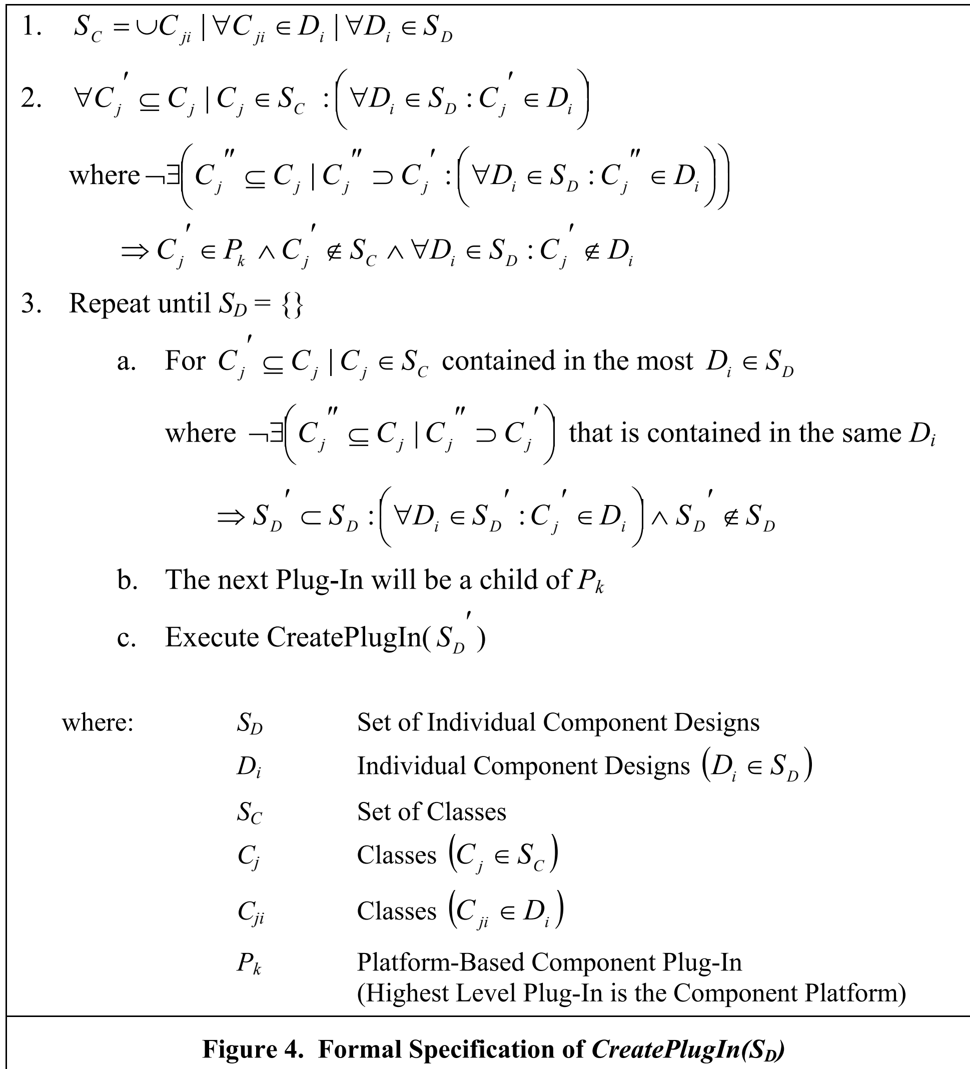


Figure 3. Reservation Component Platform with Industry Plug-Ins

When adding functionality by incorporating plug-ins into the overall platform design, the methodology differentiates between distinct functionalities and partially common functionalities: Distinct functionalities have no common elements across the derivative components; their implementation will result in separate classes for each plug-in. Thus, the functionality must be excluded from the platform; only the plug-ins will contain the class that relates to the desired functionality in each derivative (e.g. Seat class in Figure 3). Common functionalities also vary across the derivative components; however, some aspects are common to all derivatives. The commonalities manifest themselves in the implementation of common attributes or common methods in the platform's parent class that represents the dimension. The plug-ins inherit the class properties from the platform and extend the class to match the functionalities required for each derivative component (e.g. Transport Reservation class in Figure 3). Plug-ins connect to higher levels of the platform hierarchy using either inheritance (if the distinct class is a specialization of a common class) or association (if no commonalities with the distinct class exist).

The transformation process from the individual component designs to the platform design hierarchy is driven by the identification and integration of commonalities that exist between the individual designs. The conversion follows a formal process that has been refined through multiple iterations of applying the methodology to different scenarios and evaluating the outcome. Refinements were made after each iteration; Figure 4 depicts the formal specification of the resulting methodology. The development of a component platform and its plug-ins is a recursive process that we named "*CreatePlugIn*" (Figure 4). For better readability, we also use an English language description to present these steps (Figure 5).



1. $S_C = \cup C_{ji} \mid \forall C_{ji} \in D_i \mid \forall D_i \in S_D$
2. $\forall C_j' \subseteq C_j \mid C_j \in S_C : \left(\forall D_i \in S_D : C_j' \in D_i\right)$

   where $\neg\exists\left(C_j'' \subseteq C_j \mid C_j'' \supset C_j' : \left(\forall D_i \in S_D : C_j'' \in D_i\right)\right)$

   $\Rightarrow C_j' \in P_k \wedge C_j' \notin S_C \wedge \forall D_i \in S_D : C_j' \notin D_i$
3. Repeat until $S_D = \{\}$
   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_i

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

Figure 4. Formal Specification of *CreatePlugIn(S_D)*

1. S_C is the union of all classes contained in the individual 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 all individual designs that incorporates more functionality of the original class than C_j' does,
   add C_j' to the current Plug-In (or Platform if this is the highest level), remove C_j' from the set of classes S_C, and remove C_j' from all individual 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 designs D_i
      where no other class generalization exists in the same set of individual 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.
   c. Execute the process recursively with S_D' as the parameter

Figure 5. English Language Description of *CreatePlugIn(S_D)*

Post-implementation Modifications

Future derivatives of a platform can be added along the functional dimensions that were anticipated when the platform was designed. With regards to the Reservation platform example (Figure 3), that means that the set of supported industry can be expanded by adding a new industry plug-in. If the class that implements a new dimension is not part of the existing platform, functionality can also change along this dimension by adding the new class entirely to the plug-in that represents the new derivative, thus not changing the platform. Additions are only viable if the new derivative component utilizes most of the platform's functionality (in the printer context that means that it makes economic sense to add fax capabilities to a printer; however, it does not make economic sense to add fax capabilities to a dishwasher).

Further, additions should be relatively small in size compared to the existing platform (e.g. it does not make economic sense to add fax capabilities to a speaker). Although the fax function could utilize a speaker, the relative size of the fax functionality compared to the speaker functionality justifies the development of a separate fax platform. In the context of the Reservation platform example, that means that different types of booking agents can be added as an afterthought if this addition does not affect the original component platform. Figure 6 shows this addition, which is possible because the different types of booking agents specialize an existing platform class without modifying the original platform component design.

Evaluation of the Artifact

In design research, the utility, quality, and efficacy of the artifact must be evaluated (Hevner et al. 2004). Hereby the researcher can select from a variety of evaluation methods. It is important to match the evaluation methods appropriately with the artifact that is to be evaluated (Hevner et al. 2004). The design artifact in this research is the methodology facilitating a component design that we claim results in a higher component flexibility and therefore agility.

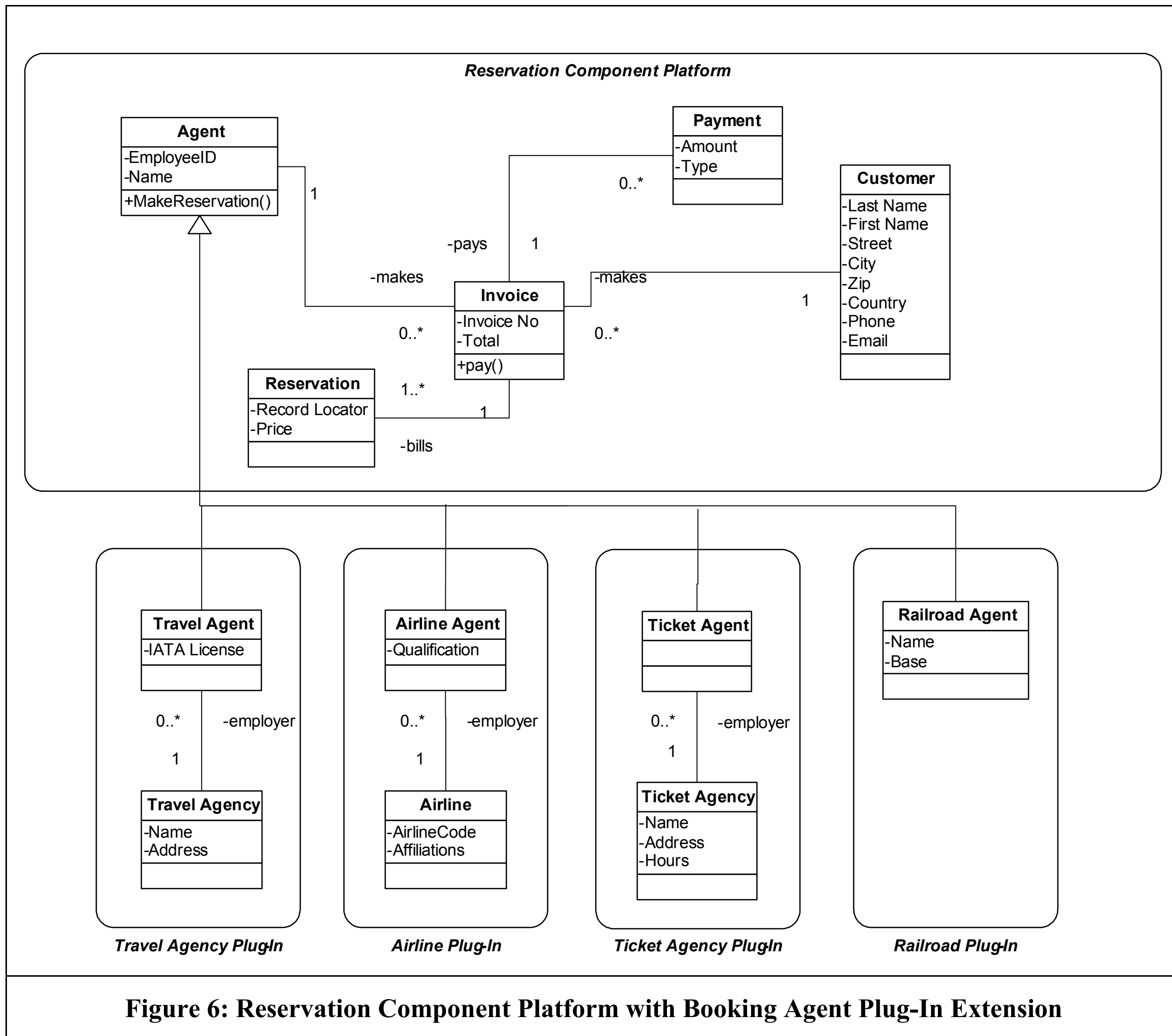


Figure 6: Reservation Component Platform with Booking Agent Plug-In Extension

A formal analysis of the reservation platform scenario demonstrates the utility of the artifact. We formally evaluated whether the platform/plug-in component hierarchy that resulted by applying the proposed methodology to the individual component designs is equivalent to the individual component designs. Only if the resulting platform component with all its plug-ins can be successfully decomposed into the original individual designs, the methodology can be deemed lossless and correct. This formal evaluation was conducted as part of the iterative process that lead to the final version of the methodology. For space considerations, the decomposition is not presented in detail; however, they are available from the authors upon request. As this functional equivalency can be formally demonstrated, the utility of the methodology is supported.

The objective of the platform component method was to improve the flexibility and agility of components. Since one platform component can incorporate a number of different domains (four domains in the scenario presented), we demonstrated that the platform component is more flexible than each of the individual components. Further, the platform component is extensible without requiring the developer to understand the implementation details of large parts of the platform. The study discussed extensions in the context of the agent scenario (Figure 6) and hereby supported the notion of agility. Thus, the scenarios presented in the study support the efficacy of the design approach.

While the utility evaluation is based on a formal assessment that is rigorous, the evaluation of efficacy is descriptive, which is a valid, yet not the strongest, evaluation method (Hevner et al. 2004). Thus, we will evaluate the methodology's efficacy and quality in an experiment involving a group of experts completing a task with and

without the platform-based approach. Subsequently, the experts will evaluate their experiences. By the time of the conference, we will have completed the research and we will present the completed evaluations.

Conclusion

This study developed an innovative method for component design that promotes the development of more flexible and more agile reusable components. This approach may be able to move black-box component reuse forward from an infrastructure-centered reuse paradigm to a method for large-scale domain-specific reuse. Thus, through the availability of more flexible and more agile components, the benefits that software developers can obtain from component-based software development may increase.

References

Apte, U.M., Sankar, C.S., Thakur, M., and Turner, J.E. "Reusability-Based Strategy for Development of Information Systems: Implementation Experience of a Bank," in: *MIS Quarterly*, 1990, p. 421.

Barnes, B.H., and Bollinger, T.B. "Making Reuse Cost-Effective," *IEEE Software* (8:1) 1991, pp 13-24.

Basili, V.R., Briand, L.C., and Melo, W.L. "How reuse influences productivity in object-oriented systems," in: *Communications of the ACM*, 1996, p. 104.

Bosch, J. "Organizing for software product lines," Software Architectures for Product Families. International Workshop IW-SAPF-3, 2000, pp. 117-134.

Chastek, G.J. "Software Product Lines," SPLC 2, 2002, p. 399 pp.

Clements, P.C., Jones, L.G., Northrop, L.M., and McGregor, J.D. "Project management in a software product line organization," *IEEE Software* (22:5), 09 2005 2005, pp 54-62.

Demeyer, S., Meijler, T.D., Nierstrasz, O., and Steyaert, P. "Design Guidelines for Tailorable Frameworks," *Communications of the ACM* (40:10) 1997, pp 60-64.

Due, R.T. "The wisdom of patterns," *Cutter IT Journal* (13:8), 08 2000 2000, pp 6-9.

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

Fayad, M., and Schmidt, D. "Object-Oriented Application Frameworks," *Communications of the ACM* (40:10) 1997, pp 32-38.

Fayad, M., Schmidt, D., and R., J. (eds.) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley, New York, 1999.

Griss, M.L., Favaro, J., and d'Alessandro, M. "Integrating feature modeling with the RSEB," *Fifth International Conference on Software Reuse*), 1998 1998, pp 76-85.

Han, T., Purao, S., and Storey, V. "A Methodology for Building a Repository of Objected-Oriented Design Fragments," 18th International Conference on Conceptual Modeling (ER'99), Paris, France, 1999, pp. 203-217.

Hevner, A.R., March, S.T., Park, J., and Ram, S. "Design Science In Information Systems Research," in: *MIS Quarterly*, 2004, pp. 75-105.

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

Jacobson, I., Griss, M., and Jonsson, P. "Making the reuse business work," *IEEE Computer* (30:10), 10 1997 1997, pp 36-42.

Johnson, R.E. "Frameworks = (Components + Patterns)," *Communications of the ACM* (40:10) 1997, pp 39-42.

Johnson, R.E., and Foote, B. "Designing Reusable Classes," *Journal of Object-Oriented Programming* (1:2) 1998, pp 22-35.

Kang, K.C., Lee, J., and Donohoe, P. "Feature-oriented product line engineering," in: *IEEE Software*, 2002, pp. 58-65.

Kang, K.C., Sajoong, K., Jaejoon, L., Kijoo, K., Euiseob, S., and Moonhang, H. "FORM: a feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering* (5), 1998 1998, pp 143-168.

Kiely, D. "Sharing data between VBA apps," *Visual Basic Programmer's Journal* (8:10), 09 1998 1998, pp 94-94-94-96, 99.

Krueger, C.W. "Software Reuse," in: *ACM Computing Surveys*, 1992, pp. 131-184.

Li, J. "The application of industrial automation technology in material management of the product line," *Journal of Shenzhen Polytechnic* (1:1), 06 2002 2002, pp 7-13.

Lim, W.C. "Effects of reuse on quality, productivity, and economics," in: *IEEE Software*, 1994, p. 23.

Meyer, B. *Object-oriented software construction* Prentice-Hall, New York, 1988, pp. xviii, 534 p.
Meyer, M.H., and Lehnerd, A.P. *The power of product platforms: building value and cost leadership* Free Press, New York, 1997, pp. xiv, 267 p.
Meyer, M.H., and Seliger, R. "Product Platforms in Software Development," in: *Sloan Management Review*, 1998, pp. 61-74.
Mili, H., Mili, F., and Mili, A. "Reusing software: Issues and research directions," in: *IEEE Transactions on Software Engineering*, 1995, pp. 528-563.
Neighbors, J.M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering* (SE10:5) 1984, p 564.
Nierstrasz, O., and Meijler, T.D. "Research directions in software composition," in: *ACM Computing Surveys*, 1995, p. 262.
Peffers, K., Tuunanen, T., Gengler, C.E., Rossi, M., Hui, W., Virtanen, V., and Bragge, J. "The Design Science Research Process: A Model for Producing and Presenting Information Systems Research," DESRIST, CGU, Claremont, CA, 2006.
Poulin, J.S., Caruso, J.M., and Hancock, D.R. "The business case for software reuse," in: *IBM Systems Journal*, 1993, p. 567 528 pages.
Purao, S., and Storey, V.C. "Intelligent Support for Retrieval and Synthesis of Patterns for Object-Oriented Design," ER, 1997, pp. 30-42.
Ravichandran, T., and Rothenberger, M.A. "Software reuse strategies and component markets," in: *Communications of the ACM*, 2003, pp. 109-114.
Salvador, F., Forza, C., and Rungtusanatham, M. "Modularity, product variety, production volume, and component sourcing: Theorizing beyond generic prescriptions," in: *Journal of Operations Management*, 2002, pp. 549-575.
Setliff, D.E., Kant, E., and Cain, T. "Practical Software Synthesis - Introduction," *IEEE Software* (10:3) 1993, pp 6-9.
Sprott, D. "Componentizing the enterprise application packages," in: *Communications of the ACM*, 2000, pp. 63-69.
Sugumaran, V., Tanniru, M., and Storey, V.C. "Supporting reuse in systems analysis," *Communications of the ACM* (43:11) 2000.
Szyperski, C.A. "Emerging component software technologies - a strategic comparison," *Software - Concepts and Tools* (19:1) 1998, pp 2-10.
Taft, D.K. "Flashline.com Profits From Software Component Market," in: *Computer Reseller News*, 2000, p. 58.
Vici, A.D., Argentieri, N., Mansour, A., d'Alessandro, M., and Favaro, J. "FODAcom: an experience with domain analysis in the Italian telecom industry," Fifth International Conference on Software Reuse (Cat. No.98TB100203), 1998, pp. 166-175.
Vitharana, P., and Jain, H. "Research issues in testing business components," in: *Information & Management*, 2000, pp. 297-309.