Strategy Pattern as a Variability Enabling Mechanism in Product Line Architecture

Zdravko Roško
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
zdravko@rosko.hr

Author(s) Name(s)
Author Affiliation(s)
Department/Institute
Full Address(es)
E-mail(s)

Abstract. *Business applications that share a common architecture and a set of reusable components, implemented by the Software Product Line (SPL) approach to software reuse, can benefit from handling the variability with extensive use of architectural design patterns. Use of the patterns within the Product Line Architecture (PLA) frameworks, yields a number of benefits toward the improvement of maintainability of the applications which are part of a SPL family. This paper presents use of the Strategy pattern within PLA as a preplanned variability enabling mechanism for SPL.*

Keywords. Strategy, Pattern, Software Product Lines, Variability, Framework, Reuse, Architecture

1 Introduction

Software reuse is the process of creating software applications from existing artifacts rather than building them from the scratch. Effective reuse requires a strategic vision that reflects the unique power and requirements of this technique [1]. There are many software engineering technologies that involve some form of software reuse. For example, software components, design patterns, application frameworks, application generators, etc. In the field of software reuse, many organizations employ these technologies, and many are ready to take the next step towards more effective reuse of software. Software product lines (SPL), in which; requirements, architecture, modeling and analysis, components , test cases, test data, test plans, documentation templates, and other software engineering artifacts can be reused over a number of applications, is at the moment the most promising form of the software reuse. SPL is defined as a set of software-intensive systems, sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [2]. SPL development process consists of domain engineering, (core assets development *for reuse*) and application engineering (product development *with reuse*) that builds the final products, where construction of the reusable assets and their variability is separated from production of the product-line applications. Successful product lines have enabled organizations to capitalize on systematic reuse to achieve business goals and desired software benefits such as productivity gains, decreased development costs, improved time to market, higher reliability, and competitive advantage [3].

A key asset of a product-line is the Product Line Architecture (PLA) – the shared architecture of the product family [4]. The central role of a common architecture is a major ingredient of the success of product line engineering compared to other reuse approaches [5].

Although the SPL approach has been widely adopted, the architectural design for a SPL has proved to be hard. Variety of existing techniques such as: architectural styles, frameworks, design patterns, existing components, make it difficult for software architect to design a well-structured solution for product line architecture.

A software asset may depend on architectural aspects at different levels of abstraction and generality, from external components at the low level through product line *Platform Framework* and domain components to a particular product, shown in Figure 1.

The SPL *Platform Framework* and domain specific components are the base on top of which products are created by using variability techniques. The external components layer (Figure 1), provides an Application Programming Interface (API) for accessing the basic

resources and services such as operating system, data base, graphical user interface, network, and etc.

Prod A1	Prod A2	Prod B1	Prod B2
Domain A Components		Domain B Components	
SPL Platform Framework			
External Components			

Figure 1. SPL layers

These external components are part of inter-organizational reuse, which has been highly successful in the organizations. Layers above the *External Components* are part of so called intra-organizational reuse which has been much less successful within the software development organizations [7].

A variability mechanism is a way of implementing varying characteristics of a component in software product lines. Goals of variability mechanisms are to minimize code duplication, reuse effort, maintenance cost, and to improve intra-organizational reuse.

In this paper we propose the use of *Strategy* design pattern as a variability enabling mechanism at the **SPL Platform Framework** level of abstraction. In this approach the application developers are required to implement the variation points of the SPL through the class inheritance based on the structure of Strategy design pattern.

2 Software Reuse and Abstraction

All approaches to software reuse use some form of abstraction for artifacts. Abstraction is the essential feature in any reuse technique. Without abstractions, software developers would be forced to shift through a collection of reusable artifacts trying to figure out what each artifact did, when it could be reused, and how to reuse it [8].

An abstraction for a software artifact is a description that suppresses the details that are unimportant to a software developer and emphasizes the information that is important. Since raising abstraction levels for software engineering technologies has proven to be quite difficult, the relation between abstraction and reuse provides us with the first clue to why there are so few successful reuse systems [8]. Many have noted the relationship between software reuse and abstraction. According to [9], for example, abstraction and re usability are "two sides of the same coin."

Software application typically consists of several layers of abstraction built on top of raw hardware. Looking from the abstraction levels perspective, the lowest-level software abstraction is the object code. Assembly language is a layer of abstraction above object code. A programming language (e.g., Java) is a layer of abstraction above the assembly language. In object-oriented languages such as Java, the class specification can serve as a layer of abstraction above the implementation details in the class body. These examples show that all software abstractions have two levels. The higher is referred to as the abstraction *specification*. The lower, more detailed level is called the abstraction *realization*. When abstractions are layered, the abstraction specification at one layer is the abstraction realization at the next higher layer. The abstraction specification typically describes "what" the abstraction does, whereas the abstraction realization describes "how" it is done [8].

Figure 2 shows an abstraction having a hidden part, a variable part, and a fixed part [8]. The hidden part contains the details in the abstraction realization that are not visible in the abstraction specification, while the variable and fixed parts are visible in the specification. Fixed part represents invariant characteristics in the abstraction realization and the variable part represents the variant characteristics in the abstraction realization.

Variable Part	Fixed Part
Hidden Part	

Figure 2. Abstraction parts

For example, in an abstraction for *Data Access Object* (DAO) to access relational database, the fixed part of the abstraction expresses the invariant characteristics for all types of realizations, such as the transaction control or execution of the SQL commands. The invariant behavior does not depend on the type of SQL commands (e.g. Oracle PL/SQL) executed, so the SQL type can be in the variable part of the abstraction. Each different SQL type corresponds to a different realization. The partitioning of an abstraction into hidden, variable, and fixed parts is not a natural property of the abstraction but rather an arbitrary decision made by the creator of the abstraction.

The abstraction creator decides what information will be presented to users and puts it in the abstraction specification. The same apply to which properties of the abstraction a user might want to vary and put them in the variable part of the specification. Having the DAO as an example, the value for the "*maximum rows returned*" from relational database can be placed in either the variable, fixed, or hidden part of the abstraction. In case it is placed in the variable part, the user has an option to choose the "*maximum rows to be returned*" (e.g., 10, 1000, unbounded). If the "*maximum rows returned*" is placed in the fixed part, the user knows the predefined value of maximum rows to be returned but does not have an option to change it. In case it is placed in the hidden part, the rows to be returned are completely removed from the concerns of the user.

3 Software Product Line

The basic idea of software product line engineering is to develop a reusable set of assets that support the development of a family of software products. Each

product in the product line may have a slightly different architecture; these architectures are instances of the product line architecture [10]. The design of the core assets for the product line is heavily influenced by the product line's scope, which defines what all of the product in the product lines will have in common and the specifics that they will not share with other products.
The goal of a software product line is to minimize the cost of developing and evolving software products that are part of a product family. A software product line captures commonalities between software products for the product family. By using a software product line, product developers are able to focus on product specific issues rather than issues that are common to all products.

3.1 Variability in Software Product Lines

Variability is the ability to change or customize a system. Re usability and flexibility have been the driving forces behind the development of such techniques as; object orientation, object oriented frameworks and software product lines.
Consequently these techniques allow us to delay certain design decisions to a later point in the development. With software product lines, the architecture of a system is fixed early but the details of an actual product implementation are delayed until product implementation. We refer to these delayed design decisions as variation points [11].
Variation points are places in the architecture where specific instances of flexibility have been built in. The flexibility is achieved by intentionally leaving specific architectural decisions open, but in a way so that they can be easily bound later, almost always by someone other than the architect [11]. Besides documenting variation points in the places where they occur: diagrams, design document, interface descriptions, example usage, and so forth, a single place where all the variability and its effects can be fully described is called a *Variability guide*. From the productivity point of view, documenting variation points at the places where they occur has the advantage that, the description is available where it is needed. But having a catalog of variation points in the form of *Variability guide* in one place serves as a complete overview of which variation points exists in the system. Figure 3 shows the different transformations a system goes through during the development. Variability can be applied on the representation subject during each of these transformations. A common goal in software engineering is to prepare software for change, especially when architecture for a family of products is designed [12].

Transformation process	**Representation**
Requirement Collection	Requirement Specification
Architecture Design	Architecture Description
Detailed Design	Design Documentation
Implementation	Source Code
Compilation	Compiled Code
Linking	Linked Code
Execution	Running Code

Figure 3. Representation & transformation processes

Variability points can be introduced at various levels of abstraction:

- Architecture description. A system is documented by using a high level design and architecture description documents.
- Detailed documentation. A system can be described by using notations and design documents.
- Source code. This level assumes creation of easy readable source code with comments to document the program behaviour and usage.
- Compiled code. Used by programming languages which support pre-processor directives.
- Linked code. Results of compilation are combined to form a specific product based on selected variation points.
- Running code. At the time of product execution, the linked system is started and dynamically configured.

Points where the architecture can vary from product (family member) to product are explicitly defined as part of SPL *Platform Framework*. The *Platform Frameworks* differ in how they express the commonalities and the points of variation in a population or product line. Reference architecture implemented in the form of the *Platform Framework* is significantly different then a single-product architecture, since it must serve as the basis for many different products simultaneously. Some of the architectural design decisions will be common among all the products, some will be unique to individual products, and some will be common among a subset of the products. Ordinary software architecture extending into product line architecture can be accomplished through the addition of *variation points* to create variant architectures. A member of the product family, a different product, is then represented by a variant of the architecture.

3.2 Product Line Architecture

Product Line Architecture (PLA) defines the overall software structure of the entire product line. It is the first point where the products' variation is represented in design. The specific mechanisms by which the PLA addresses the variation is somewhat dependent on the

architectural style and approach used [13].
The architecture of a system is "the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them" [14]. Unlike a traditional system's architecture, it relates to the entire product line.
In contrast to the PLA that spans the entire product line, a *Product Architecture* (PA) describes the architecture of an individual product in the product line. A product's architecture differs from its PLA by making variant-specific decisions based on variation points specified by the PLA [13]. In other words, the PLA must address how the variability in the software requirements (functional and non-functional) is used to derive the various product architectures.
Figure 4 illustrates the relationship between the PLA and the individual PA [13].

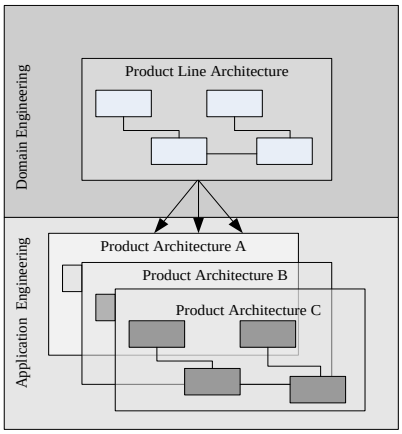

Figure 4. Product Architecture Derivation

The SPL technical architecture describes the structure of the products and provides the development infrastructure to support the product development. The infrastructure is focused on making the product developer more productive by providing the tools to support product development, including the following [22]:

- **Frameworks**, which are customizable generic solutions to specific product problems
- **Services**, which provide an infrastructure that allows products to use common functions
- **Patterns**, which are solution templates for commonly encountered problems

3.2.1 Frameworks

Software Product Lines are collections of frameworks and other reusable assets that can be tailored to create concrete software products relatively fast compared to developing from scratch [15].
A framework is a reusable design expressed as a set of software artifacts such as: classes, properties, resources, persistence objects, documents, reference application, and the way their instances collaborate. A framework can be defined as "a partial design and implementation for an application in a given domain"[24]. A developer customizes a framework to a particular software product by sub classing and using instances of framework artifacts. Framework dictates the architecture of your application. The framework captures the design decisions that are common to its application domain [16]. Framework doesn't have to be implemented in an object-oriented language, even though it usually is. Design patterns may be used in the design of a framework. A single framework may be using several design patterns. The [16] book describes the major differences between design patterns and frameworks as follows:

- Design patterns are more abstract than frameworks.
- Design patterns are smaller architectural elements than frameworks.
- Design patterns are less specialized than frameworks.

SPL embodies the processes, tools, and software assets that can be used to derive applications sharing similar structure and functionality [8]. All software assets related to a family of products are consolidated within a reusable framework, instead of being organized and reused in an ad hoc manner. The framework is used to construct a product of a family and to provide specific product-variants as needed.
The central artifact in a SPL is the framework capable of being applied to multiple applications.

3.2.2 Services

Platform Framework for product line has build in many infrastructure services. Services are designed to be shared by multiple products. Some typical product services for business applications are: *Transaction, Security, Logging, Rules, Workflow, Data cache, Data access, Data validation, Resource externalization, Exception handling, Session management.*

3.2.3 Patterns

In recent years, patterns have been used extensively in software design efforts as a way to decrease design and development time and increase robustness and quality [22]. A pattern describes a specific design problem and an abstract solution to that problem. The authors of Patterns of Software Architecture [23] define these three types of patterns as follows:

- Architectural Patterns: an architectural pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes

rules and guidelines for organizing the relationships between them.

- Design Patterns: a design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- Idioms: an idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

The difference between these three groups of patterns are in their levels of abstraction. Architectural patterns are high-level strategies that have wide implications on the overall structure and organization of a software system. Design patterns are medium-level tactics that define some of the structure and behavior of entities and their relationships.
Idioms are paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior [23].

4 Use of Strategy Pattern

This section gives an overview of our approach to variability management based on *Strategy* design pattern. Subsections 4.1, 4.2 and 4.3 introduce the model for implementation of architectural variation points based on *Strategy* design pattern.
The intent of the *Strategy* pattern is to „define a family of algorithms, encapsulate each one, and make them interchangeable“[16]. Strategy lets the algorithm vary independently from clients that use it. It is useful when many related classes differ only in behavior, because it makes it possible to configure a class with one of many behaviors. An algorithm uses data that clients shouldn't know about. The classes implementing each strategy inherit a common abstract class or interface and implement specific methods to handle each strategy. The structure of the Strategy Design Pattern is shown in Figure 5. *Context* and *Strategy* interact to implement chosen algorithm. A context forwards requests from its clients to its strategy.
In this paper we present the three locations within a PLA where the strategy design pattern can be applied to handle the variability. These locations are places in the typical business application architecture where specific instances of flexibility could be built in to support the variability of needed algorithms.

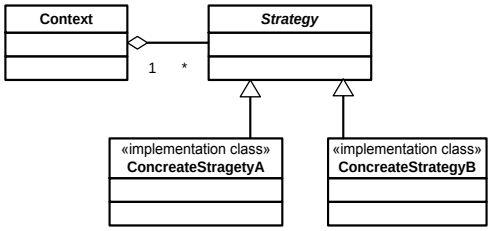

Figure 5. Strategy Pattern Structure

We propose the locations at the *Execution or Linking* level (Figure 3) of abstraction where the strategy pattern can be applied:

- the location where the client communicate with the server
- the location where the business objects uses data access object
- the location where the transaction or connection pool uses a connection to the data source

4.1 Client/Server transport

The client/server model is a computing model that acts as distributed application which partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients [17]. Clients and servers communicate over a computer network on a separate hardware, but both client and server may reside in the same system and also within the same system process.
Client/server model where the applications are split into layered such as: *presentation logic*, *business logic* and *data access logic* layer, represents rather *logical* then physical client/server model, since an application is not necessary ditributed on client and server but it could reside within one or more system processes. The choice of communication mechanism to use between the client and server, depends on the context the application is being used.
Suppose we started to develop a business application from reusable assets of the software product line. At the development time we often need to execute or access the application logic developed or residing within all of the application layers. Communication mechanisms between client and server, such as HTTP, RMI, DCOM, typically, are not available or not convenient to use while we are within an development environment. But, having available the variation point to bridge the missing communication infrastructure needed to simulate the application environment, helps us to program our application and to execute unit and integration test within our development environment.
On the other side, applications used by the end users, often require the support for different communication mechanism. Existing design patterns in this field, such as „*Protocol Plug-In*“ [18] and “*Business Delegate*”

[19], abstract application developers from communication protocol details and allow for flexible support of several communication protocols but do not address the „*Local*“ communication transport.
Most of the applications today do not support more then one communication mechanism between client and server. The lack of support causes low integration quality of an application and inconvenient application development within an IDE. Also, there is the limitation by existing patterns in case when the „*Local*“ communication needs to be implemented at the run time of an application in the form of so called „*fat client*“ where all application layers reside within one system process rather then in the form of distributed systems. These limitations lead us to seek an alternative solution for flexible communication mechanism composition.

4.1.1 Variation for Client/Server Transport

Figure 6 show the variation point and some of the variations for the client/server communication. Beside the benefits from the use of *Strategy* pattern, we propose the use of „*Local*“transport variation to support the „protocol free“application development while working within an IDE. This is needed to enable application developers not to worry about complex environment setup (Web server, EJB Container, JMS Server, etc.) needed to support diverse protocols, but rather free them to concentrate on development of application business logic. Latter on, in case the application is deployed as a “Fact Client” application, where both, client and server parts reside together as one deliverable component, the “*Local*” protocol enables an application to function free from any dependent technology related to the communication protocols such as HTTP, JMS, etc. The “*Local*” communication between client and server means that the client side components communicate with the servers side business logic components directly, free from TCP/IP communication. No communication protocol is used to deliver a message from client to server, but rather the message is delivered to the server through a “Proxy” client over a variability mechanism based on Strategy design pattern.

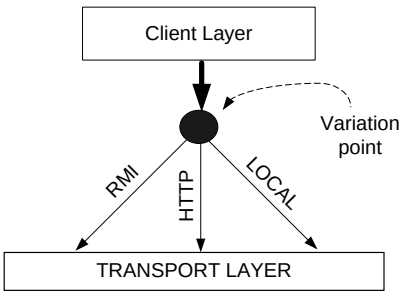

Figure 6. Communication Variation Point

Figure 7 shows the client/server communication variation *Strategy* pattern implementation where it represents the relationship between classes `ClientProxy` and `TransportClient`, using a class diagram. `TransportClient` is an abstract class to define a common behavior for the transport algorithms, while strategy classes inherit the transport client and implement an algorithm specific for the transport type such as HTTP, RMI, etc.
With this approach, it all comes down to choosing the right class for the right communication mechanism. We believe that *Strategy* and inheritance is appropriate tool to achieve this: by passing appropriate arguments to the protocol layer. Inheritance by it self, without using the *Strategy*, is not appropriate when it comes to choosing among several protocol algorithms at run time. *Strategy* `TransportClientRMI` builds message and then issues „send“communication operation to deliver message over to server layer. The fail over, asynchronous communication, timeout, exception handling, load balancing are implemented by generic transport classes independent of any particular communication mechanism.

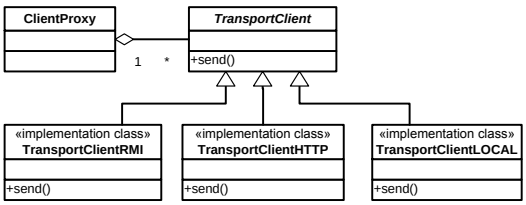

Figure 7. Structure of the transport strategy

Figure 8 sketches the way client layer objects and transport algorithm objects interact. Whenever an operation related to the transport is invoked on client layer, the execution of transport is delegated to strategy `TransportClient`. Each instance of the `TransportClient` class represents one execution of transport implemented by that class.

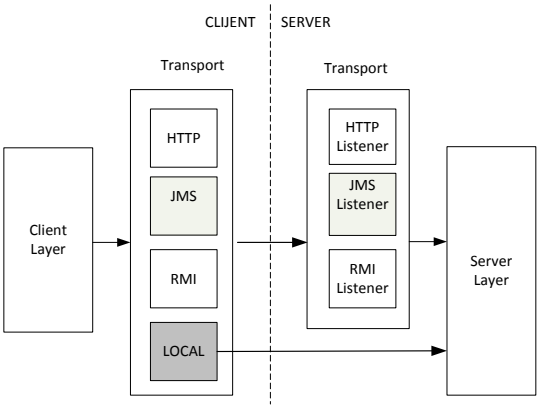

Figure 8. Components of the transport variation

4.2 Data Access Objects

Business logic application layer accessing data from any data source (databases, web services, legacy systems, flat files, and so forth) may use the *Data*

Access Object pattern which implements the Strategy [20] design pattern and hides most of the complexity away from an application programmer by encapsulating its dynamic behavior in the base data access class. Existing patterns and technologies such as *Object/Relational* mapping, EJB, etc., do not address a diversity of potential data sources and its commonality such as: connection management, transaction control, data caching, etc., for each of the specific data source types on a unique and manageable way.

4.2.1 Variation for Data Access Objects

Figure 9 show the variation point and some of the variations for the access of diverse data sources. Beside the benefits from the use of *Strategy* pattern, we propose the use of common base class for diverse type of data sources, relational or not relational. For example, the base class to implement strategy is shared between relational and non relational data sources. Having the same base class for CICS, SAP and JDBC data sources for example, makes it possible to replace the data access algorithm without making changes to the application itself. Our experience proves that the implementation of *Strategy* design pattern for the data sources access pays off, especially in the enterprise size application environment.

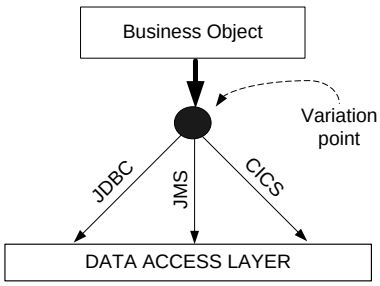

Figure 9. Data Access Object Variation Point

Figure 10 shows the *Data Access Object* variation strategy pattern implementation where it represents the relationship between classes `BusinessObject` and `DataAccessObject`, using a class diagram. `DataAccessObject` is an abstract class to define a common behavior for the data access algorithms, while strategy classes inherit it and implement an algorithm specific for the data source type such as JDBC, JMS, CICS, etc.
With this approach, it all comes down to choosing the right class for the right data access mechanism. We believe that strategy and inheritance is appropriate tool to achieve this: by setting the appropriate configuration parameters for specific data source at the business logic layer. Inheritance by it self, without using the strategy, is not appropriate when it comes to choosing among several data access algorithms for diverse data sources. Specific strategy algorithm builds a message and then issues the data source specific operation to deliver a message over the data source. The connection handling, transaction control, result set handling, timeout, exception handling, etc., are implemented by generic data access object classes independent of any particular data source access mechanism.

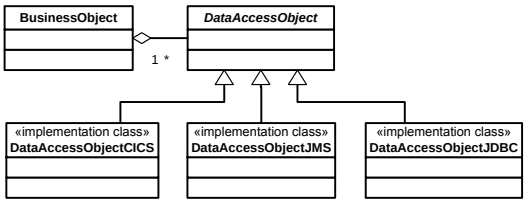

Figure 10. Structure of the Data Access Object strategy

4.3 Data Source Connections

Information of an enterprise may be in the form or relational database records, business objects in an ERP, transaction program CICS transaction processing system and etc. In order to integrate these diverse systems most vendors supported a variety of custom adapters for the integration of these systems. Basically these adapters provided complex and limited native interfaces. Because of these, application developers had to deal with too many different adapters which lacked support for connection management, handling security and transaction support.
In order to address the above problems Sun Microsystems released the J2EE Connector Architecture, JCA that provides a standard architecture for integration of J2EE Servers with heterogeneous EIS resources. It provides a common API and a common set of services within a consistent J2EE Environment [21].
JCA provides a solution for applications executing within an Application Server and accessing the supported data sources, however, in the case an application is not running within Application server or in case there is no support for JCA from a data source vendor, a custom solution is needed. We propose the use of the *Strategy* design pattern to combine JCA and other, not supported data sources. Our experience proved that applications using the proposed approach could execute within an Application Server or as a separate application but still accessing diverse data sources combined in a shared transaction context.

4.3.1 Variation for Data Source Connections

Figure 11 show the variation point and some of the variations for the data source connection. This variability combines JCA connections and product line custom connections, all within an Application Server. In case an application needs to be deployed

outside of an Application Server, the Strategy support this variations but have in mind that JCA connection are not possible to include.

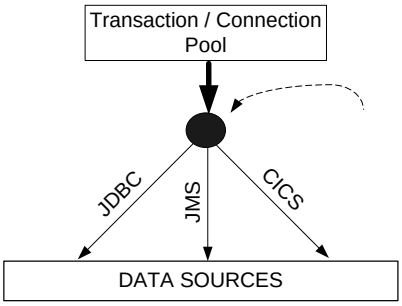

Figure 11. Data Source Connection Variation Point

Figure 12 shows two use cases of this strategy. First, the transaction support needs to reference the connections in order to execute commit or rollback. The other use case for this strategy is the custom connection pool context. In case the application developer decides to use the application outside of an Application Server and use custom connection pool, the Strategy allows managing the transaction and supporting the connection management.

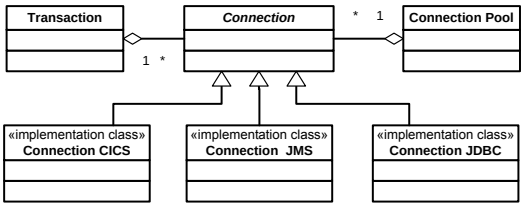

Figure 12. Structure of the Data Source Connection strategy

5 Conclusions

Most of the systems using the SPL approach to build a family of business applications will use more than one type of variation mechanism. Variation mechanisms are applied at different levels of abstraction within an SPL. The PLA is the first place where the variation is represented in design. The specific mechanisms by which the PLA addresses the variation depends on overall software structure of the entire product line. Design patterns as a proven solution for general design problems, provide a mechanism to handle variations at the PLA or at the level of product design.

This paper has presented the Strategy design pattern as an architectural variation mechanism and the three variation points inside a typical business application SPL framework. The hope is that this experience based design can be used as a guide for product line architects to make reasoned choices about which type of variation mechanism to use at the referenced variation points.

References

[1] Guide to the **Software Engineering Body of Knowledge**, The Institute of Electrical and Electronics Engineers, Inc., 2004, 8-1, pp. 120.

[2] P. Clements and L.M. Northrop: **Software Product Lines - Practices and Patterns**. Addison-Wesley. 2001, pp. 17, 29, 31.

[3] Kyo C. Kang, Vijayan Sugumaran, Sooyong Park: **Applied Software Product Line Engeenering**, Taylor and Francis Group, 2011, pp. 6.

[4] Jan Bosch: **Design and use of software architectures: adopting and evolving a product-line approach**, ACM Press/Addison-Wesley Publishing Co., 2000

[5] Frank van der Linden, Klaus Schmid and Eelco Rommes: **The Product Line Engineering Approach**, Springer, 2007, pp. 14.

[6] Zdravko Roško: **Software Product Lines: Source Code Organization for 3-tier OLTP Architecture Systems**, CECIIS, 2011.

[7] Jan Bosch: **Interview on Product Lines and Software Ecosystems**, http://www.se-radio.net/?s=bosch&submit=Find, 2009.

[8] Charles W. Krueger: **Software Reuse, School of Computer Science**, G'arnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1992.

[9] Wegner, P: **Varieties of reusability. In Workshop on Reusability in Programming** (Newport, R. I., Sept.). ITT Programming, Stratford, 1983.

[10] Magnus Eriksson: **An Approach to Software Product Line Use Case Modeling**, LICENTIATE THESIS, 2006.

[11] Mikael Svahnberg, Jilles van Gurp, Jan Bosch: **On the Notion of Variability in Software Product Lines**, Blekinge Institute of Technology, 2001.

[12] Michel Jaring and Jan Bosch: **Evolution in Software Product Families: Architecture Implementation rather than Architecture Design**, 2002.

[13] Steve Livengood: **Product Line Architecture Variability Mechanisms**, Proceedings of the Workshop held in conjunction with the 10th Software Product Line Conference, 2006.

[14] Bass, L., Clements, P., & Kazman, R: **Software Architecture in Practice (2nd edition).**, Addison-Wesley 2003.

[15] Jilles Van Gurp , Jan Bosch , Mikael Svahnberg: **On the Notion of Variability in Software Product Lines**, Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), p.45, August 28-31, 2001.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: **Design Patterns, Elements of Reusable Object-Oriented Software**, Addison-Wesley Publishing Company, Reading, USA, 1995, pp. 315.

[17] Sun Microsystem: **Distributed Application Architecture**, 2000.

[18] Markus Volter: **Remoting Patterns Foundations of Enterprise, Internet and Realtime Distributed Object Middleware**, John Wiley & Sons Ltd., 2005, pp. 135.

[19] http://www.javabeat.net/articles/22-j2ee-connector-architecturejca-an-introduction-1.html

[20] Zdravko Roško: **Dynamic Data Access Object Design Pattern**, CECIIS, 2008.

[21] http://www.javabeat.net/articles/22-j2ee-connector-architecturejca-an-introduction-1.html

[22] Paul Harmon, Michael Rosen, Michael Guttman: **Developing E-business systems and architectures**, Morgan Kaufmann, 2001. 160, 161.

[23] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Michael Stal: **Pattern-Oriented Software Architecture Volume 1: A System of Patterns**, Wiley, 1996, pp. 12,13,14.

[24] Jan Bosch: **Evolution and Composition of Reusable Assets in Product Line Architectures: A Case Study**, Proceedings of the First Working IFIP Conference on Software Architecture , 1999.