An Approach to Assemble Software Products using a Product Line Approach

Dr. Jay van Zyl

SystemicLogic (Pty) Ltd, South Africa

jay@systemiclogic.com

Abstract: *Software product lines present many benefits over the traditional methods of building systems. This paper discusses two primary concepts, namely, the separation continuum and application assembly using product lines. It starts by presenting a separation continuum that shows how vertical and horizontal layering can assist with separating user interface from business logic and data at an implementation level, and the separation of customer facing processes from infrastructure facing processes at a business or abstract level. An application assembly approach is discussed whereby a product line architecture is tied to the separation continuum showing how high levels of productivity can be achieved when realizing product lines. The approach presented in this paper is still under development with implementation on a limited number of product lines only. It is intended that the experiences presented will provoke and stimulate further experimentation needed to deal with large scale application assembly capabilities.*

1 Introduction

Re-usability is a topic that is constantly under discussion. The reality is that there are no "silver bullets", no easy process, no easy software solution and no bullet-proof architecture to assist with building and using re-usable software assets. In this era of objects, components and service based architectures, how do architects and software practitioners implement large-scale usable, re-usable software products that are fit for purpose? How are the benefits of product lines fully realized once implemented? These kinds of questions are answered in this paper.

This paper ties together two essential concepts, namely, the separation continuum, and product line practices. The two concepts "morph" into one, yielding benefits that all revolve around the ability to: "assemble" software systems safely, quickly and easily into product lines. The complex nature of large-scale software systems requires a clear separation of the concerns that is represented by a *separation continuum*.

Emerging approaches, such as the Model Driven Architecture (MDA) [1] and the Reusable Asset Specification (RAS) from Rational Corporation, shows that the need to have a common method for understanding how requirements are realized, independently of the implementation, is receiving a fair deal of attention. The MDA framework is based on modelling different levels and abstractions of systems, and exploiting the relationships amongst these models. The RAS provides guidelines for description, development and application of different kinds of reusable software assets. Neither of these initiatives focuses on producing product lines.

The Product Line Management v2.0 [2, pg 3] presents the following definition: "A software product line is 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". Common software assets are invented, designed and built in such a way that they can be used in a multitude of product lines. The ability to build and evolve these separately has major benefits as architectural, resource and other dependencies can be balanced.

In order to have a clearly separated technology and platform architecture [3] suggests that certain elements of flow control must be removed from the underlying architectures. Components might be independently defined and built, but the flow of processes and data presents many platform specific challenges. The core software assets must include tools to perform these tasks without having a specific platform feature implemented.

This paper contributes to the research already underway in the field of assembling systems from previously constructed software assets.

Section 2 introduces the separation continuum and the way in which the various dimensions are identified.

Section 3 introduces the concept of assembling new product lines from pre-defined software assets. It also shows how product lines are realized.

Section 4 closes with some specific comments on the implementation of the principles described in this paper.

2 Separation Continuum

The principle of "the separation of concerns" has been used by various architectural styles. Essentially, it is used to deal with the complexities that exist in the definition and use of software systems. The continuum described here attempts to extend the thinking already prevalent in layered architectures. Architectures always describe a system that is made up of smaller parts [5]. Complexity is managed by understanding the continuums that exist when describing, defining and assembling product lines using these parts.

2.1 Separation of Concerns

The Separation Continuum can be defined as: "*a systemic view of a system to understand its parts and their relationship, through understanding vertical and horizontal continuums needed where abstraction and implementation are on the vertical continuum, and human and machine facing aspects to have attributes of adaptability, context and relevance are on the horizontal continuum.*"

Model Driven Architecture [4] has a vision that overlaps with the separation continuum: "An integrated enterprise is one whose business areas, strategies, architectures, organizational units, processes, machines, technology platforms, data and information – and the organization's understanding of these – form a consistent, cohesive, and adaptive whole".

It is reasoned that the separation continuum will assist in the implementation of product line practices. The ability to understand interrelated layers of a system and the levels of re-usability required in producing software intensive product lines, have many benefits that include:

Agility – the ability of the business to respond to market demand by having highly adaptive software systems.

Time-to-market – the ability to assemble new software products by using software assets in a highly productive and easy to use manner.

Operational optimization – the ability to have focused teams produce relevant assets based on functional and non-functional requirements, and

Innovation ability – new products can be produced as prototypes by using different combinations of assets.

When architectures are designed for new product lines, there needs to be one main objective [6], namely, "avoiding a monolithic design by extreme de-coupling of components and localization of functionality so that every component can be replaced or upgraded in isolation". The separation continuum takes this concept one step further in that other dimensions in systems delivery need to be considered.

"Horizontal separation" is concerned with the separation of how the user interface is independent of the connectivity and, in turn, separate from the business logic and data access. "Vertical separation" is the layering needed to implement platform elements separately from the higher levels of abstraction needed at application levels typically represented in business requirements.

Vertical separation of contexts in the organization or system considers the following levels, moving from abstract layering to platform implementation:

Business Model – This layer represents the basic concept of "doing business".

Systems Model – The systems model deals with the systems required to satisfy a specific business model.

Applications Portfolio – Systems are required to automate required systems by means of producing software assets, product lines and product families.

Services Architecture – This architecture is concerned with the breadth of functionality needed to implement the business systems via the applications portfolio. A generic term that could be used here is "Web service" or "service based architecture".

Component Architecture - The component architecture view presents a complete view of the business components that can be exposed as services. Components are typically coarse-grained software elements that satisfy a particular requirement, and

Object Architecture – Components are created from object or other technologies and exposed as business components. Objects are the finest granularity of functionality in a typical system.

This continuum also assists with application context realization [7] and refers to this as "Zoom-in" and "Zoom-out". These "zooming" concepts can be used in order to move up or down the vertical continuum, for example, a particular business model is realized by a number of systems models. It also means that there is traceability between the various layers. This ability allows for the direct mapping onto the Model Driven Architecture through platform independent and platform dependent models [1].

The applications portfolio lives on the abstract continuum and forms the bridge between the *layering abstract* and *platform implementation* continuums. Application engineering [7], as a concept of producing applications, uses all the concepts as articulated at the various levels in the continuum. Even though the applications portfolio might seem more "Platform Implementation" specific, it is not. Overlaps into the "Layering Abstract" become more apparent when systems can be constructed quickly, and the stakeholders decide to accept changes in this level rather than the technology [8]. This means that deployable software will include the definitions of the application portfolio in some platform independent fashion such as an Extensible Markup Language (XML) definition.

3 Assembly Using a Product Line Approach

Product lines are defined in order to realize a specific customer requirement, market need or market niche. Product line features, or requirements, are determined from the *systems models* that require software *applications* to function. "Assembly" is the process of exploring and re-using existing software assets. In the case where a requirement cannot be satisfied, a development life cycle will be initiated to construct the relevant asset.

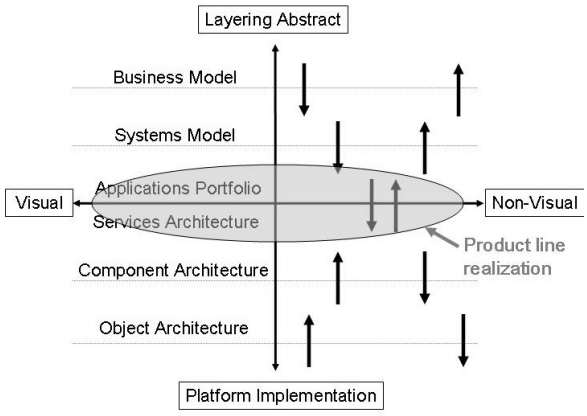

Fig. 1. Separation continuum and product line realization.

Principles of altering behaviour and information requirements of more dynamic assets can be applied in assembly processes. In "Software Product Lines", Clements and Northrop [9] state that: "In a product line, the generic form of the component is evolved and maintained in the asset base. In component-based development, if any variation is involved, it is usually accomplished by writing code, and the variations are most likely maintained separately". During system assembly the focus should be on the tying together of a predefined set of assets that comes in the form of web services or services, components or objects. It must be considered that these assets could have been acquired from different sources.

Assembly using software as assets, is needed to realize the benefits of agility, time-to-market, operational optimization and increased innovation ability.

Component and object architectures form the nucleus of software execution. Product lines are realized where the abstract layers join the implementation layers. The diagram above shows that there are two ways to realize product lines:

Top-down: The business model drives strategic initiatives from the top down to the implementation. All the interpretations of how the business systems must function are realized in the application portfolio specification. The platform implementation continuum needs to be flexible enough to respond to changes with quick turn-around times. *Business consultants* or *systems analysts* typically function in this domain, and

Bottom-up: The architectures already implemented either present enablers or inhibitors to the business model. Web services is a good example of how business is affected and how a new means of realizing functionality

quickly can facilitate new business products to be taken to market. *Developer, architect* and *assembler* are roles typically seen in this domain.

Product lines are produced from software assets to satisfy a specific product space. This production exercise should be executed with rigour, based on processes that reflect mature capabilities. Scoping of product lines has posed challenges to the production of product lines [10]. Producing the product, by using the "independent synchronised product delivery" [11] concept, poses risks to the team if the scope of the asset and overall product line is not clearly defined and communicated.

Part of the message here is that the separation continuum is used to layer elements of a software system – *layering abstract* into *platform implementation*, and that assembly by means of using software assets, forms the basis of productivity and time-to-market gains.

3.1 Separation Continuum Implementation

The approach described next relies on the understanding that systems have elements that are stable and others that are volatile in the context of on-going systemic evolution. Basically, one needs to understand "the things that will stay the same" versus "the things that might or will change" over a particular time period. "The things that stay the same" are developed in the underlying software technology platform and represent stability. "The things that change" are used to tell the software how to behave and represent volatility. This is done by means of using patterns when assembling systems. Essentially one needs to separate the commonalities from the variabilities [12].

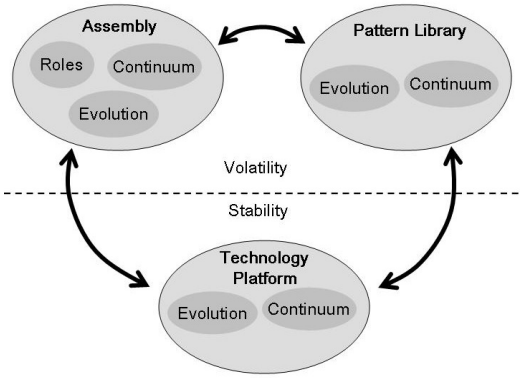

Fig. 2. Software asset creation using a pattern-technology-assembly method.

Patterns are used as a means to facilitate re-use of requirements, software assets, designs, and other artefacts needed to construct a system. Each pattern describes the solution to a known problem and is stored in library. It also describes the structural and behavioural solutions. Patterns are classified as per the separation continuum. It means that business model patterns have certain system model patterns linked to it. Certain system model patterns have application portfolio elements linked to it, and so on.

Business needs, once analysed, can be mapped onto business patterns relevant to a particular domain. If new patterns are identified, a process is initiated whereby a new pattern is created. These patterns form the basis on which the final software system will be constructed. The availability of patterns that show how specific problems have been solved presents many opportunities for business analysts to apply defined knowledge. When used in productivity tools, patterns realized previously give the development teams comfort as re-use is implemented by means of pattern recognition and application.

Looking at the separation continuum it was seen that the abstract layers describe only the patterns and requirements needed by a business audience. The *platform implementation* revealed software assets that are used to construct the various product lines using services and components. Once services are put to use, the resultant product line reveals a "look and feel" that are specific to the intended user. The user interface and other user-facing aspects are typically part of the "things that change" between two product line implementations.

The next section will explore how assets are built and acquired through using the pattern-technology-assembly method.

3.2 The Product Space

The previous section provided an introduction into the concepts separating volatility from stability when looking at software product lines. This section will show how a final product is constructed using the *pattern library, assembly* and *technology platform* areas.

Pattern library. The pattern library provides descriptions of the potential solutions that a client might require in a specific domain. The separation continuum is used to ensure the structure of the library is such that new patterns can be added over time. The horizontal continuum ensures the completeness and effective separation of the various architectural elements, that is, by having the following catered for: correct user interface, user interface controller, connectivity, business logic, and data.

The pattern library is the central location and method where process and technology knowledge are captured. All aspects of a solution can be "patternized", even if there is one pattern to match a particular software component, COTS based component, or even a Web service. For the sake of focus, this discussion will focus only on patterns for the *platform implementation* continuum.

Application patterns focus on coarse-grained software systems that implement specific business systems. These patterns need to cover all the aspects of an application that make up the deployable product line. They typically cover the specifics of how services and components will be used in a final application. Example application patterns are: electronic banking and self-service banking terminal. Each of these will have specific characteristics when used that might include the way a user identifies himself/herself, how the user interface is represented and used, and how a legacy system is accessed.

Services patterns can be classified in the areas of user interface, user interface controller, connectivity, business logic and data. These classifications apply to all the layers on the *platform* implementation continuum. This is the layer where these patterns are realized through physical software – a point where a one-on-one relationship exists between a pattern and its implementation by referencing a software asset. Services can be constructed by using components or might be used from an external source (for example, Web services). Example services patterns are ManageCustomer, TransferFunds, etc. There is a mapping between services patterns and application patterns.

Component patterns are classified the same way as the services patterns. Since most software development will take place at this layer and the object layer, a vast number of patterns are available. Some specific ones are: ManageHierarchy, StoreTransaction, RetrieveTransaction, etc. There could also have been components called ManageCustomer and TransferFunds – the services layer becomes the "externalization" of these components.

Each pattern should have behavioural and structural definitions. Pattern behaviour is the common set of stimuli, actions and responses that the pattern implements. The pattern structure is the more static definitions that the pattern needs, for example, data. Each of these definitions can be made up of other related behavioural and structural definitions.

The population of the pattern library should be done only once a clear understanding is formed of how a product can be realized in the context of an application. Analysis techniques need to be applied in order to have the appropriate levels of abstraction adhered to. The population and usage of patterns involve various roles throughout the life cycle.

Assembly. Assembly of software assets takes place in the context of available patterns. Patterns are "instantiated" by being given values to a set of properties – making its execution ready for a technology platform. Specific roles throughout the process of constructing software assets need to apply pattern usage characteristics to ensure secure implementations.

The process of assembling systems starts with a product definition or business need and follows through to the final deployment of the executable software. Roles in the assembly process might change depending on the organizational requirements. Roles like *business analysts*, *component builder* and *assembler* need productive tools that assist with re-using pre-built assets. The business analyst defines the requirement; the component builder develops components, and the assembler ties the various components together.

When using patterns at the application layer, the assembler role focuses on producing a product line that can be deployed with all its related assets. The application will contain all the services, components and objects in order to execute on a selected technology platform. The result is a definition of an application package that can be deployed.

Constructing or using services can be done via using the patterns at the services layer. Either new services are constructed or existing ones used, when a product line is realized. The assembly process facilitates this process of creating new, using existing, and deploying services for a product line.

Components are used or constructed in the assembly process depending on application and service requirements. Typically there are two paths to follow: firstly – using existing components, either built previously by using an Assembler tool or acquired from an external party, or secondly – creating new components by using an Assembler tool or industry standard development environment. Each component fits into user-interface, user-interface-controller, connectivity, business-logic, or data, classifications.

When software assets are created in the assembler from a pattern library, they can be referred to as "top-down development". If existing assets are explored (for example, legacy system components, custom developed components and COTS based systems) and are registered in the assembly environment, they can be referred to as "bottom-up development". The process of "wiring" components and services together to form a product line is largely simplified due to the ability of separating out the various continuums.

Technical. The technology platform is used to realize the pattern and its associated assembly attributes. Together these three areas make up the finally deployable component and/or service as part of a software product. Patterns implemented in particular technology platforms (for example, Java versus C# based), might require specific environmental implementation variations where security, data access and component invocation might be different.

Certain patterns are used as software requirements to develop software assets. It must be taken into account that the assembly process will define the ways in which the pattern, once turned into a software asset, can be executed. Behavioural and structural requirements are specified by the assembler and should not be developed in the underlying software code. This means that the developer of a core software asset, using the pattern as the specification, needs to design and develop code that is flexible enough to execute within the constraints of the pattern's specification. The combination of pattern, assembly properties and technical platform code form a deployable component.

Once a product line is deployed onto a particular platform, it must be possible to view the installed software assets. Services (as software assets) represent implementations of components from a multitude of unrelated platforms. They also provide the entry point into registry standards like Universal Description, Discovery and Integration (UDDI). The interface specification, that is, the contract, does not reveal the "real" implementation but is a façade to the component that contains the functionality. The Web Services Description Language (WSDL) can be used to describe software assets.

4 Discussion

Software product lines have emerged as a very important paradigm in delivering systems. Major benefits are derived from the practice of using common assets to build systems. Remarkable increases in time-to-market, product quality, and customer satisfaction are realised through this framework. With the "component" paradigm maturing and the emergence of "Web services", companies can construct systems from commonly available software assets, through a process of application assembly. The separation continuum must be seen as a means by which developers, architects, design specialists, and business analysts can participate when creating large and complex systems in harmony. The continuum facilitates the use of architectural principles that need to be applied when using Web services. It also assists to provide a means to re-use code written by others, elsewhere, using their own set of tools.

The separation of business and technology has never been more relevant than now. Assembling systems from software assets (including Web services) that have been implemented in a variety of languages and architectures is the vision for the provision of application software provision. It opens an entirely new thread of discussion as to what software is available on an organization's infrastructure versus that of its software-as-service supplier. The separation continuum and product line architecture described in this paper formed the basis of work performed between 2000 and 2002. Examples included were taken from actual software products, either available or currently under development. Strategic product development [13] and product line practices provided the process and product-line thinking required in order to produce these products.

5 References

[1] Architecture board MDA drafting team. (2001). Model driven architecture: A technical perspective. Document Number ab/2001-02-01.

[2] Clements P. and Northrop L.M. (1999). A framework for software product line practice. Version 2.0, SEI.

[3] Doerr B.S. and Sharp D.C. (2000). Freeing product line architectures from execution dependencies. Software product lines: Experience and research directions. Edited by Patrick Donohoe. Kluwer Press. ISBN 0-7923-7940-3.

[4] Dsouza D. (2001). Model-driven architecture opportunities and challenges. Kinetium. http://www.kinetium.com/. Website accessed: 15 September 2002

[5] Bachman F., Bass L., Carriere J. and Clements P. (2000). Software architecture documentation in practice: Documenting architectural layers. CMU/SEI Special Report CMU/SEI-2000-SR-004.

[6] Pronk B.J. (2000). An interface-based platform approach. Software product lines: Experience and research directions. Edited by Patrick Donohoe. Kluwer Press. ISBN 0-7923-7940-3.

[7] Atkinson C., Bayer J. and Muthig D. (2000). Component-based product line development. Software product lines: Experience and research directions. Edited by Patrick Donohoe. Kluwer Press. ISBN 0-7923-7940-3.

[8] van Zyl J.A. (2001). Class of solution dilemma. IEEE Engineering Management Conference proceedings. IEEE Electronic Library.

[9] Clements P. and Northrop L. (2001). Software product lines, practices and patterns. Addison Wesley Press. ISBN 0-201-70332-7.

[10] Schmid K. (2000). Scoping software product lines.

[11] van Zyl J.A. (2001). Product line balancing. IEEE Engineering Management Conference proceedings. IEEE Electronic Library.

[12] van Zyl J.A. (2002). Product Line Architecture and the Separation of Concerns, Second Software Product Line Conference – SPLC 2, San Diego, Kluver Publication.

[13] van Zyl J.A. (1999). Strategic product development. First Software Product Line Conference - SPLC1. Denver, USA, 28-31 August 2000.