Application Internationalization: Commonality Analysis Using a Family-Based Approach

To facilitate the use of a software application internationally, organizations must view it as part of a systems suite and dissect constituent components to enable or enhance their reuse.

Executive Summary

This white paper provides an analysis framework and a methodology for designing or reengineering an application to suit international markets.

An application aimed at multiple markets typically has a set of common attributes that address a common business purpose across different regions. Each regional version of the application has its own variations to meet local market requirements. This paper proposes the analysis of a set of region-specific versions of an application by considering them as a family of software systems.

We used the principles outlined in this paper for analyzing and designing a Web application of a large and leading legal business solutions provider.

Status Check

In today's increasingly global economy, a software product is often made available for international markets. In order to suit local or regional markets, the application will need to be customized in various ways. This concept of enabling an application to address different cultures/regions is referred to as globalization. Customization of an application to suit a specific locale (region/culture) is referred to as localization.[1]

Perspectives of Software Reuse

One of the key necessities after enabling an application or a software product for international markets is to enable reuse among the entire suite of enterprise software. Reuse of a software element can either be tactical or systematic.[2] Tactical reuse serves ad hoc needs and often leads to fragmentation of the reused element. A systematic approach is typically accomplished through the reuse of the structure or the framework of the application. The framework could be application building block elements such as application bootstrapping, initialization, configuration, caching, exception handling, state management and data fetch/access or user interface building blocks such as navigation, redirection or workflow handling logic.

This paper presents the analysis to enable systematic reuse of components.

The goals of application reuse may be to address business goals, or architectural goals, or both.

Business Goals

The development of an application usually begins with a need for serving a local geography. Eventually, the need to leverage and use the extant application for new markets emerges. The reasons typically include:

- Reduce "time to market" of the product or a feature for a new region by leveraging code that is already developed for another region.
- Enable accelerated maturity of the product in newer markets.
- Cut overall product development and maintenance costs.

Architectural goals

The following architectural goals must be achieved to create a reusable application design. Some of the aforementioned business goals could also be translated into architectural goals:

- The application code base should be available for use in multiple regions and to serve as a foundation for region-specific product development.
- Components developed in one region should be available for use in another region, as need arises.
- Disparate changes performed on a feature or application to meet distinct regional requirements should be isolatable from each other.
- Extensibility points should be available in an application for its reuse and enhancement by different regions.
- The application should support region-specific configurability. The configuration could be usage of a specific application skin, welcome banner, user-interface attributes, accessibility-related changes, etc.
- Application structure should support concurrent development of the product in different regions.
- Application structure should support periodic or frequent source code synchronization.

Applications built to serve different markets that help achieve the same set of business functionalities will share common features and traits. This commonality of features and traits forms the "business domain" of the application that it addresses. These applications fit well into the pattern of a "family of software products." The commonality aspects observed in the family of products is the motivating factor for reuse-oriented design.

Commonality and variability analysis (CVA) is a well-researched methodology for analyzing a family of products.[3] This paper applies these principles and presents a methodology for identification and assessment of an application to support globalization.[4]

Background and Context

Family of Products

Consider an application that needs to be designed in such a way that it can be used in multiple markets, with each market having its own region-specific localizations. The fact that applications in different regions have (significant) aspects of similarity is one of the motivations for applying a common model on which each of the applications is built upon. Applications that are built on this standard model (functional/architectural/design) can then be considered to form a "family of software products." As with any standard model of software system families, the principles of CVA can be used to arrive at a structure to represent the application set.

Commonality and Variability Analysis

The commonality among the family constituents is comprised of the feature set that is present in all or most enterprise applications in the set; variability would be the features that are present only in one or a few of them. There could also be cases where features are disparate, but can be parameterized. A parameterized type of variation is one that follows a similar structure or functionality, but input conditions or data vary.

The aspect in which one product or a component differs from another in the family is called the variation point. The variation point is the "socket" at which different variants plug in and cause particular variations. An objective of CVA is identifying the variation points.[5]

The Methodology Explained

Overview

Consider a product that is to be developed for different geographical markets. At a broad level, the similar business need that application addresses in different regions forms the underlying commonality. The region-specific customizations needed to address local market requirements form the variable aspect.

In addition to the high-level business purpose that would be similar in applications for different regions, there could be finer levels of commonality between them. Similarity could also be based on the following accounts:

- Similarity of features or a granule of business functionality.
- Similarity of user workflows or presentation aspects; a need such as to have a consistent look, feel and navigational aspects of the user interface elements could be considered commonality required across members of a software family.
- Similarity of nonfunctional aspects of the application:
  - Security standards.
  - Performance levels that are to be met.
  - Quality levels.
  - Standards to be followed in development and other stages of the product lifecycle.
- A desired similarity of the design, or structure, of the application. (In a way, this is a goal.)

Considerations that vary in an application from one region to another include:

- Content displayed to the user (region-specific content varies from one application to another).
- Content available to the application; the data source and systems providing the data.
- External systems that the application interacts with could vary from one region to another.
- User interface design.
- Display of data in different formats:
  - Date/time.
  - Currencies.
  - Units of measurements.
  - Address, phone number formats, area codes, etc.
- Application skin: color, banner, contact information, etc.

The commonality analysis helps in setting up the application's generic structure and behavior pattern. The identified commonalities form the application's building blocks. The building blocks could relate to functional or nonfunctional aspects. The identified variations are again abstracted by identification of the variation point.

The commonality in the application across different regional markets forms the basis of the application framework. The CVA methodology recommends one governing issue or concern per commonality identified. This aids in higher cohesion of the identified abstraction.

Anticipated future variations should also be taken into account. Each class of variation represents an axis of change along which the application can change over time.

Three broad kinds of abstractions can be derived from the analysis:

- **Those which represent business functionalities.** Implementations of these abstractions could vary from one region to another, but will follow a pattern. For instance, a template method pattern might describe the behavior. There is a moderate level of implementation reuse. The common implementations are those that address similar functional use cases.
- **Those which represent the essential application building blocks or the framework.** Implementations of these abstractions could yield libraries or reusable algorithms that are common across regions. These would support a high level of implementation reuse. These components would allow a possible extensibility through a "strategy pattern."
- **Those which represent the region-specific variation points.** Implementations of these abstractions are likely to be widely different from one region to another. Data format transformations, user interfaces, etc. could be examples of this. There is little implementation-level reuse. These abstractions serve as hinges or hooks on which region-specific implementations are attached to the application.

In object-oriented development terms, the commonalities would form the base of classes/abstractions, which the region-specific classes would implement. The commonalities represent the abstract entities that each region- or market-specific variation would extend. The observed commonalities and their interactions sequences can be laid out to form a template method or a strategy pattern.[6] The region-specific derived classes would then fit into the sequence as different specializations of the base abstractions. Region-specific elements would be the variations of the abstractions identified through CVA.

Diagramming Search – Variability and Commonality

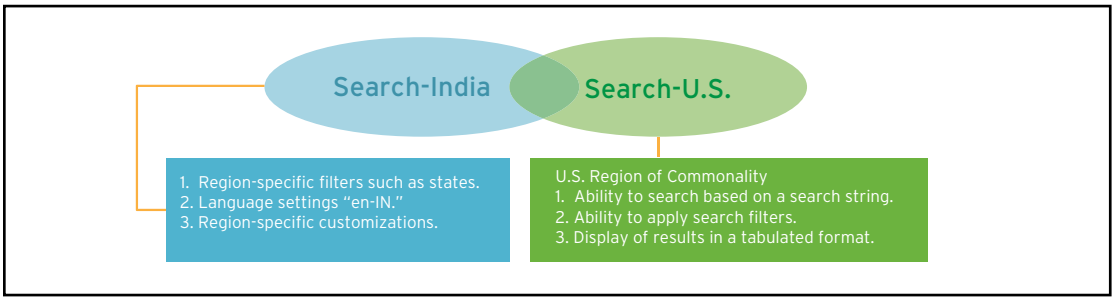

Figure 1

Illustrative Sample Case

An example will be useful in this context:

Consider a search application that is targeted for different regions or markets. The search application has the following features:

- Search based on a search string: Filters could be applied to the search to narrow down results to a specific type.
- View a document from the search result.
- Save a document to a docket.
- View contents of a docket.

Let us consider that this application needs to be made available to markets in different geographies and the application for a market should display search results relevant to that market in a specific locale. In other words, the content and format of the content displayed by the search application shall vary from one market to another.

Consider that the application is to be launched in two markets – India and the U.S. At an abstract level, the search feature in different regions would have a set of core functionalities that do not vary from one region to another. For instance, the ability to enter a search string and perform search based on that keyword, application of filters (states, locality or other regional criteria) to narrow search results, etc. are the commonalities of this capability in the two regions. These commonalities help establish a common domain model for search and interaction/sequence diagrams at a high level.

The commonality and variability of applications or components of an application can be represented with a Venn diagram (see Figure 1).

It is possible that disparate business functionalities that do not have any apparent similarity from a user perspective still have elements of commonality between them. The commonality could be along user interface/presentation layer similarities or in nonfunctional aspects such as data access patterns, algorithms or data processing logic, etc. It is these similarities that help in identifying common libraries and a runtime framework for applications developed for different regions.

It can be expected that commonalities exist among the same feature or functionality in different regions. It is also possible that commonalities might exist across different functionalities across regions (see Figure 2).

Diagramming Search – Commonalities Among Functionalities

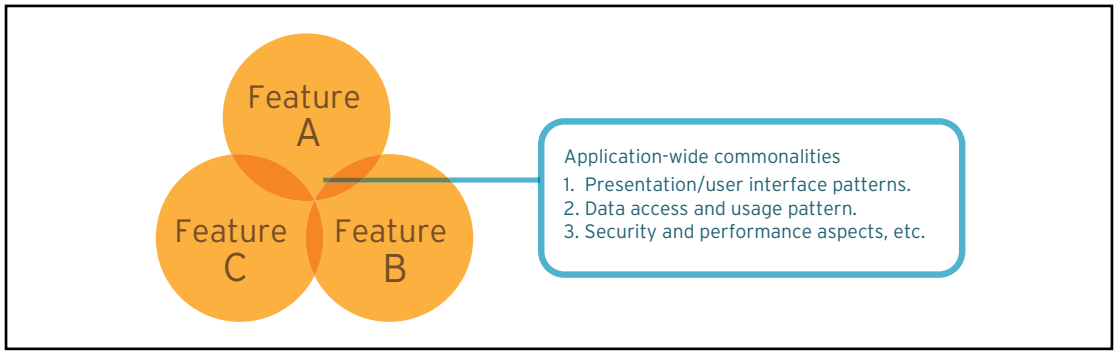

Figure 2

The commonalities observed within a feature in different areas help in establishing a feature-specific abstraction (for instance, the template method for performing a search operation), while the commonalities observed across different features aid in building the application framework and the libraries. The latter establishes the structure or the framework of the application and creates building blocks over which a feature-specific abstraction fits in (see Figure 3). Elements that can potentially be a part of a framework include:

- Components with common implementation across all markets/regions.
- Components that are likely to be stable and suffer little variation over time.

Feature modeling is a useful technique in this analysis.[7] Feature modeling helps in categorizing unique and salient capabilities of a software product. The members of a software product family share, to a large extent, common characteristics. There will, however, be variations or differences that distinguish one product of the family from another. It is to be noted that there will be elements of variability between two distinct products in a product family – it is the variability that makes one product of the family different from the other.

An Analysis Framework

Any application (product) can be considered to be comprised of a set of features, put together using building blocks, targeting a particular user base. The features, building blocks and the user base, in a way, form the key dimensions or axis of change for an application.

The following parameters are defined to measure the degree to which a use case could include feature-specific aspects, building block aspects and user base-specific aspects.

- **Feature specificity:** The degree to which the application layer or component is specific to one or more features (features are business functionalities).
- **Genericity/component reusability:** This specifies building block code. It measures the extent to which a specific component can be parameterized to suit different inputs. That is, the degree to which a layer or a component includes reusable code with or without parameterization. (Framework elements such as caching, exception handling, logging, security.) This could include utility computation, algorithmic routines or code such as application bootstrapping, initialization, configuration, etc.
- **Region specificity:** This specifies the user base or the locale-specific implementation. The degree to which a layer or component contains culture-specific code (resources/embedded strings, skins, date time or currency formatting, locale information, etc.).

Plug-in Model of Application Framework

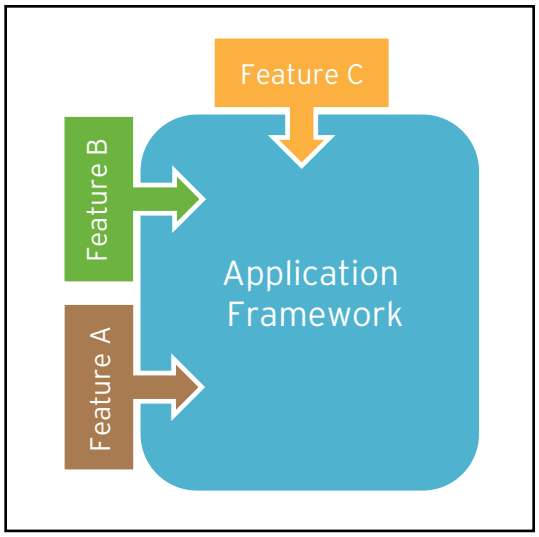

Figure 3

There will be elements of variability between two distinct products in a product family – it is the variability that makes one product of the family different from the other.

The degree of mutual isolation between feature-specific code, reusable code and region-specific code measures the extent of support for application internationalization. These parameters can be used in conjunction with the standard parameters – cohesion, coupling and abstraction – of a modular design.[8]

It is to be noted that feature specificity is different from region specificity. For example, a feature could be "display of temperature of a city" (such as in weather applications). Region specificity deals with whether the temperature needs to be displayed in Celsius degrees (such as for users in India) or in Fahrenheit (for the U.S.). The logic needed to assemble temperature data from the data store, computation of monthly averages or moving averages, etc. would be the feature-specific code. Region-specific code should be isolated

A Module or Use Case Collection

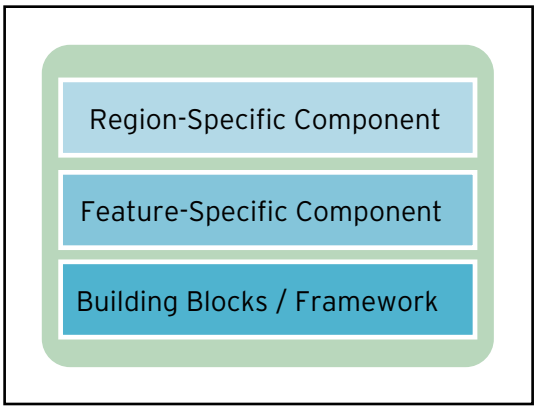

Figure 4

from feature-specific code. The code for a specific feature should have high cohesion (within its feature area) and a low degree of coupling with other feature areas.

An application can be considered to comprise a set of packages or modules (that contain implementations of related use cases) with each module comprising a set of components. Each module is most likely to have implementations of all the three types listed above. The idea of commonality and variability analysis and deriving abstractions is to establish separation of concerns (see Figure 4).

The Likert Scale[9] can be defined to measure the degree or extent to which a module (implementation of related use cases) has elements of feature specificity, region specificity and potential building block elements. Here, three primary dimensions of variations are considered: feature variance, region variance and the application building blocks. Additional parameters such as variations in offered quality of service levels, accessibility or other application- or domain-specific considerations can also be included in this analysis.

The following approach can be used to assign scales/weights for different use cases.

The application or a product is first decomposed into high-level use cases. The use cases at this level are very coarse and represent business features at a broader level. Likert Scale weights are assigned to one of the three axes of change based on the relative degree to which this use case leans toward.

For instance, the scale shown in Figure 5 can be applied for assigning weightages to different parameters.

Sample Scale of Parameter Weightage

Scale	Degree of Measurement
0	No Relevance
2	Low Relevance
5	Mostly Relevant
7	Fully Relevant

Figure 5

Figure 6 lists a set of simple use cases and a representative scaling of use cases based on three criteria.

Often, implementations to address nonfunctional needs of an application are cross-cutting

Representative Use Case Scaling

Use Case	Type	Feature Specificity	Region Specificity	Genericity
User must be able to search based on a search string.	Functional	5	5	2
User must be able to narrow down search results based on filter criteria.	Functional	5	5	2
All user actions must be captured for tracking user analytics.	Nonfunctional	0	0	7
The page must contain local contact information.	Functional	2	5	0
All errors from application must be logged in a repository.	Nonfunctional	0	2	7

Figure 6

in nature and emerge as candidates for building blocks, libraries or the framework layer.

Consider use case 2 in Figure 6: "User must be able to narrow down search results based on filter criteria." Here, the values that constitute the filter could vary by region. The data or content that constitutes the filter forms the region-specific attribute. The ability to apply filter and narrow down search results is independent of "filter value." This ability is feature specific rather than region specific. Abstracting the ability to narrow down search results based on filter could be implemented in a feature-specific component.

This scaling methodology helps in identifying and segregating variations and commonality in different use cases. The analysis helps in identifying interfaces and their implementations, and also in determining the constituents of a component (packaging of classes).

Related Work and Areas of Interest

This paper does not provide notes on implementation practices. However, a few salient and important considerations are covered below.

Versioning and Source Code Management

Versioning of artifacts is a key consideration when a component is designed for reuse. Versioning is applicable to the binary output (which is the reused "component" or package) as well as the underlying source code. A region-specific variation (localized artifact) should be isolatable from the common elements. It is also likely that each regional market has different product release criteria. The development teams building and testing the software could be different in each region. All these factors necessitate the establishment of suitable source code management policies (branching and merging).[10]

Implementation Considerations

The following reflect effective coding practices for building a world-ready application:[11]

- **Use unicode representations for strings.** Also, support the possibility of encountering data in different encoded formats (at least, the application should degrade gracefully with appropriate error messages).
- **Package the resource-only data** separate from the core logic of the application.
- **String length (byte size) for a particular user message can vary from one language to another.** Allocate enough buffer capacity to prevent memory overruns.

Application Re-factoring Techniques

As a part of implementation, it often becomes essential to harvest existing functionalities or re-factor existing code to align it with the reference architecture that supports globalization.

An existing application can be re-factored and reengineered using:

- Re-factoring through dependency inversion.[12]
- Improve feature cohesion by repackaging related assemblies.[13]

Implications for the Future of App Dev

This paper provides an analysis methodology for building an application aimed at international markets. These principles can also be applied to assess the international readiness of an application or to design future applications for globally minded companies

The guiding principle for an application to be reusable in multiple markets is to have a core executable framework code that is built on abstractions and is culture neutral. The region-specific variations plug into the common application framework. The application design must also support ease of localization.

Footnotes

[1] Concepts of globalization and localization can be found in this site: http://msdn.microsoft.com/en-us/library/aa292205(v=vs.71).aspx. .NET application specific perspectives are explained at http://msdn.microsoft.com/en-us/library/h6270d0z.aspx.

[2] Allenby et al. explain the concept of reuse in this paper: Karen Allenby, Mike Bardill, Simon Burton, Darren Buttle, Stuart Hutchesson, John McDermid, John Murdoch and Alan Stephenson, "A Family-Oriented Software Development Process for Engine Controllers," Workshop on Domain-Specific Visual Languages at OOPSLA 2001, Tampa Bay, FL.

[3] David M. Weiss discusses the idea of commonality analysis in the paper, "Commonality Analysis: A Systematic Process for Defining Families," Lucent Technologies, Bell Laboratories.

[4] J. Coplien, D. Hoffman and David M. Weiss, "Commonality and Variability in Software Engineering," IEEE Software, Volume 15, No. 6., Nov/Dec 1998, pp. 37-45.

[5] M. Clauβ, "Generic Modeling Using UML Extensions for Variability," Proceedings of the Third International Conference on Product-Focused Software Process Improvement, September 2001.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[7] Mohsen Asadi, Bardia Mohabbati et al., "Model Driven Development of Families of Service Oriented Architectures," and Jaejoon Lee, D.Muthiag and M. Naab, "An Approach for Developing Service-Oriented Product Lines," Proceedings of the First International Workshop on Feature-Oriented Software Development, discuss feature modeling in detail.

[8] R.C. Martin, Articles for Design Principles, Object Mentor, http://www.objectmentor.com/resources/publishedArticles.html.

[9] Rensis Likert, "A Technique for the Measurement of Attitudes," *Archives of Psychology*, 1932.

[10] http://vsarbranchingguide.codeplex.com/ provides detailed notes on branching and merging guidelines to address different development scenarios.

[11] MSDN Library's Globalization Step-by-Step (http://msdn.microsoft.com/en-us/goglobal/bb688110), and MSDN Library's Designing a World-Ready Program (http://msdn.microsoft.com/en-us/goglobal/bb978437) provide .NET platform specific details.

[12] R.C. Martin, "The Dependency Inversion Principle," Object Mentor (http://www.objectmentor.com/resources/articles/dip.pdf).

[13] R.C. Martin, Articles for Design Principles, Object Mentor (http://www.objectmentor.com/resources/publishedArticles.html).

About the Author

Sudarsan Srinivasan is an Architect with Cognizant's Advanced Solutions Group. He has more than 10 years of experience in solution architecture, design and development and has worked as technical lead for various project teams. Sudarsan is currently responsible for solution delivery, performance engineering and application modularization of a research application developed by a large and leading U.S.-based legal services provider. Prior to this project, Sudarsan led the development of Distributed Connectivity Services, a messaging-based communication infrastructure platform provided by Microsoft (http://technet.microsoft.com/en-us/library/dd632797.aspx). Sudarsan holds a post-graduate degree in computer applications from College of Engineering, Anna University, Chennai. He can be reached at Sudarsan.Srinivasan2@cognizant.com.

About Cognizant

Cognizant (NASDAQ: CTSH) is a leading provider of information technology, consulting, and business process outsourcing services, dedicated to helping the world's leading companies build stronger businesses. Headquartered in Teaneck, New Jersey (U.S.), Cognizant combines a passion for client satisfaction, technology innovation, deep industry and business process expertise, and a global, collaborative workforce that embodies the future of work. With over 50 delivery centers worldwide and approximately 150,400 employees as of September 30, 2012, Cognizant is a member of the NASDAQ-100, the S&P 500, the Forbes Global 2000, and the Fortune 500 and is ranked among the top performing and fastest growing companies in the world. Visit us online at www.cognizant.com or follow us on Twitter: Cognizant.

World Headquarters
500 Frank W. Burr Blvd.
Teaneck, NJ 07666 USA
Phone: +1 201 801 0233
Fax: +1 201 801 0243
Toll Free: +1 888 937 3277
Email: inquiry@cognizant.com

European Headquarters
1 Kingdom Street
Paddington Central
London W2 6BD
Phone: +44 (0) 20 7297 7600
Fax: +44 (0) 20 7121 0102
Email: infouk@cognizant.com

India Operations Headquarters
#5/535, Old Mahabalipuram Road
Okkiyam Pettai, Thoraipakkam
Chennai, 600 096 India
Phone: +91 (0) 44 4209 6000
Fax: +91 (0) 44 4209 6060
Email: inquiryindia@cognizant.com