Domain Engineering And Variability In The Reuse-Driven Software Engineering Business.

Martin L. Griss, Laboratory Scientist, Hewlett-Packard Laboratories, Palo Alto, CA.

Effective systematic software reuse requires a coherent approach to structuring architecture, process and organization. In previous columns, Ivar Jacobson and I summarize[1,2,3] our approach to extending Ivar's Object-oriented Software Engineering process (OOSE)[4] for large-scale, systematic reuse. We call our approach the "Reuse-Driven Software Engineering Business" (RSEB), described in greater detail in our forthcoming book [5]. An RSEB is a software development organization which has been explicitly structured to employ systematic reuse as a key business strategy.

The Essence Of The RSEB: Architecture, Process and Organization

The RSEB extends OOSE with architectural constructs for families of related applications built from reusable components. **Components** are public elements from several OOSE models, including use case, analysis, design and implementation classes (code). Components are not used alone, but in groups called **component systems**, which interact in a layered, modular architecture. Components within a component system are connected by relationships and interfaces, and often grouped into subsystems which behave as frameworks. Those model elements chosen to be reusable are exported from the component system via one or more **facades**. Each facade defines a distinct, coherent public interface to the component system. Non-public model elements implement and describe the components.

The RSEB defines several incremental, iterative software reuse processes, corresponding to reusable asset creation, utilization, management and support[6]. The most important RSEB processes are **Application Family Engineering, Component System Engineering**, and **Application System Engineering**. Application family engineering starts from use cases defining a set of related applications. These yield an overall layered architecture, a set of component systems, and the interfaces and relationships between them. Component system engineering designs and implements each component system, while application system engineering builds applications from these component systems.

The RSEB provides an organization model for the creation of the architecture and component systems by one or more teams, and their reuse by other teams. Instead of building each related application independently, from scratch, some parts of the organization purposely and proactively create reusable component systems, while other parts utilize these component systems to build applications more rapidly and cost effectively.

These processes and organization are modeled using OO business engineering[7].

Domain Engineering Structures Components For An Application Family

Effective component system creation relies on an activity historically called “domain engineering”[8]. The RSEB does not have an explicit domain engineering process, but distributes key domain engineering activities between the Application Family Engineering and Component System Engineering processes.

These activities (Figure 1) identify an application or subsystem “domain,” a design space for a family of related systems. The process identifies commonality and variability in the chosen problem domain, defines an architecture for applications and components, and develops a set of appropriately generalized components.

Application engineering then specializes and assembles these components into applications. These applications are largely constrained to “fit” the architecture and the components. Typical applications usually consist of components from several different domains.

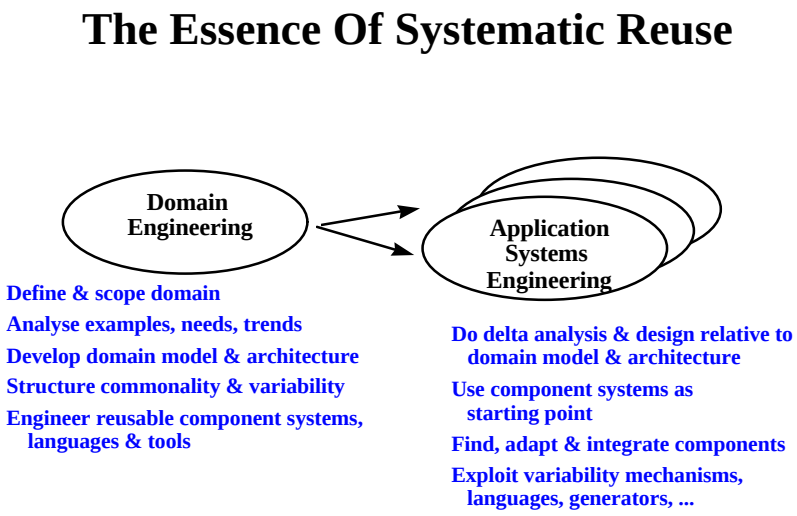

Figure 1 Domain engineering creates components for subsequent specialization and assembly into applications.

Since the creation of reusable components is more expensive than “ordinary” system development, we only develop components when an economic or business reason for reuse exists, and when the resulting components will be reused multiple times. We should not simply guess at supposedly reusable components Guesses are usually quite poor, leading to components that will not work well together, nor support effective applications development.

The RSEB identifies the high-payoff domains and components systematically. The RSEB first uses OO business engineering to analyze business needs and processes, from which it designs a suite of related information system applications for the business[7]. Business use cases defining the processes provide a unifying framework to integrate the various applications into a coherent business system. We analyze the business criteria, and capture supporting rationale, models and architectures to enable better decisions to be made, recorded and revisited for

future improvement. Example systems, user needs, domain expertise and technology trends are analyzed to identify and characterize features that are common to all family members, and other features that vary between family members.

The RSEB extends OOSE to support this model-driven approach to reuse with several constructs and activities, similar to those in many existing domain engineering methods[8]. These differ in how they identify the domain, how they match to the target software processes and technology, and how they collect, represent and cluster features. These activities involve iterative cycles over steps such as:

1. **Domain identification and scoping** - most applications consist of several recognizable or distinct subsystems or sub-problems, only some of which are worth addressing from a reuse perspective.
2. **Selection and analysis of features, examples, needs and trends -** there is a delicate balance between reactive and proactive reuse. A set of reusable components must anticipate future needs, but since this is difficult and expensive to do reliably, the process must identify and prioritize the most important features to find essential commonality and variability.
3. **Identification, factoring and clustering of feature sets -** extract and analyze the key features using extended OOSE modeling constructs, grouping requirements or implementation features into common and variant clusters.
4. **Development of 'domain' or 'generic' model and architecture -** from these feature clusters and use cases, we develop a robust architecture relating mechanisms, features, subsystems and variants. This architecture shapes all the resulting applications.
5. **Expression of usable commonality and variability -** generalized subsystems, modules and functions are identified, and related to each other as generalizations or alternatives, using appropriate variability notations and mechanisms.
6. **Implementation, certification, and packaging of reusable assets** - the 'most important' subset of the candidate assets are implemented and released as certified, reusable component systems.

The goal of application system engineering is then to rapidly and/or cost effectively develop conforming applications, constrained to the domain architecture, and taking maximum advantage of the available component systems. The appropriate steps are summarized in Figure 1.

In future columns we will discuss in more detail the exact processes we use in the RSEB. OO business engineering and systems of interacting systems play a key role in moving from target business goals and business processes to the application family layered architecture, and then to the set of component systems.

Domain Engineering Needs Richer Variability Mechanisms

At the heart of effective domain engineering is the identification and expression of commonality and variability. In addition to new constructs for component systems and facades, the RSEB extends OOSE with a richer set of mechanisms to express variability. Sub-classing (inheritance) is simply not powerful enough to produce compact and flexible components.

In implementation languages such as C++, JAVA or Visual Basic, inheritance can be augmented or replaced by combinations of parameters, aggregation, delegation, structured exception trapping, macros, generators and templates to express the needed variability. For example, Microsoft Wizards are simple template-driven generators, while NETRON/CAP[9] provides a powerful frame editing language which executes editing commands in hierarchically linked templates to produced highly specialized solutions.

We have added analogous variability constructs to our RSEB modeling language. In the Unified Modeling Language (UML) terminology, we have extended the set of *«generalization»* stereotypes available[10] in OOSE. We have defined a **variation point** as a location at which a specializing **variant** can be applied using some mechanism. Classes containing variation points are usually incomplete and require an attached variant before being usable. Variants are usually fragments of classes, text strings or parameters, used to complete the class.

Figure 2 shows a use case component for part of a simple banking example with two variation points, and several specific variants. In these diagrams we show variation points as solid dots labeled with markers **{Account overdraft}** or **{Dispense money}.** These markers can appear in any class, such as use cases, actors, analysis objects and design objects. Variants are shown as smaller use case or analysis objects, such as ***deduct fee***, attached by a generalization arrow. Figure 2 also shows several corresponding analysis components with related variation points and variants.

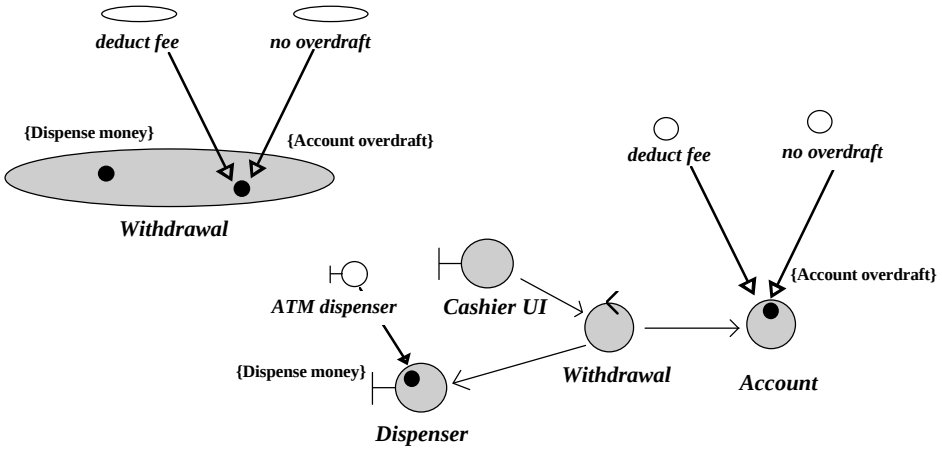

Figure 2 Use case and analysis components showing several variation points and available variants.

We define several useful variability mechanisms, and express each in UML as a particular stereotype of generalization. Mechanisms include inheritance, parameters, uses, extends, templates and generation. . Each has its own way of applying or attaching the associated variant. For example, in a use case description corresponding to Figure 2, the variation point might be a distinguished text, such as **{Account overdraft}**. A variant using an *«inherits»*, *«extends»*, *«uses»*, or *«parameterizes»* generalization will replace this variation point text.

In general, this behaves as if a copy is made of the class containing the variation point, and then replacing the variation point definition text or structure by some other text or structure. The replacement is obtained either by directly inserting a parameter or reference to the variant, or by computing some transformation using the variation point definition and supplied variant. For example, inheritance behaves as if the subclass was a complete copy of the superclass with the virtual operations overridden by the supplied operations.

This is more than a just a generic name for the distinct variability mechanisms. Each variation point is *explicitly* defined, and carefully *documented with constraints* on the type of variants, how these are to be used, and how to write new variants. Variation points and variants in one model will be related to other variation points and variants in this model, and have a defined mapping to corresponding variation points in other models.

In our next column, we will continue with a discussion of the mechanisms and processes that support systematic reuse in the RSEB. We will see how reusable components and variability in one model are traceable to others, and how frameworks and patterns are related to abstract subsystems and use cases.

References

[1] ML Griss, Systematic OO software reuse - a year of progress, Object Magazine, February 1996.

[2] ML Griss and RR Kessler, Building object-oriented instrument kits, Object Magazine, April 1996.

[3] I Jacobson,, Succeeding with Objects: Reuse in Reality, Object Magazine, July 1996.

[4] I Jacobson et al, Object-oriented software engineering: A use case driven approach, Addison-Wesley, 1992.

[5] I Jacobson, ML. Griss, P Jonsson, Software Reuse: Architecture, Process and Organization for Business Success", Addison-Wesley (to be published, 1996).

[6] ML Griss, Software reuse - a process of getting organized, Object Magazine, May 1995.

[7] I Jacobson, M Ericsson and A Jacobson, The Object Advantage: Business process reengineering with object technology, Addison-Wesley 1994.

[8] G Arango, Domain Analysis Methods, in W Schäfer et al., Software Reusability, Ellis Horwood, 1994.

[9] PG Bassett, Framing Software Reuse: Lessons from the Real World, Prentice Hall 1996.

[10] G Booch, J Rumbaugh and I Jacobson, "The Unified Modeling Language," Technical Report, Rational Software Corporation, October 1996.