JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 5, July-August 2009

Goal-driven Product Derivation

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

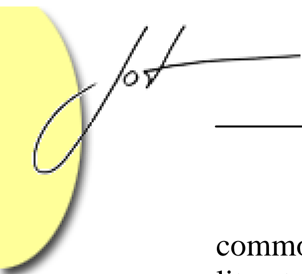
The purpose of a software product line organization is to produce products. Some organizations adopt the software product line approach because they want better quality products, some want the products faster, and some want to produce the products using fewer resources. There are many different ways to produce products and which way is the best depends on which of many goals are the ultimate objective. In this issue of Strategic Software Engineering I will discuss how the specifications of goals for the production system during strategic planning affect how products are derived in a product line organization. (By the way, remember to visit www.splc.net to see what is happening at SPLC 2009)

1 CONTEXT

As this column is published I will have just finished presenting a keynote address on product derivation at WIRE 2009. I am using the same topic for both this column and the address to provide myself the opportunity to examine derivation from the twin perspectives of strategy and implementation. In this issue of Strategic Software Engineering I will investigate the implication of "goal-driven" for how products are derived and I will bring several threads, which I have written about previously, together into a coherent whole.

Goal-driven derivation seeks to optimize the set of specified goals of the derivation process and the resulting products. The production goals are derived from the goals stated in the business case by thinking of the qualities the production capability of the product line should possess. For example, "easily extended" is a generally desirable quality for the products derived from a product line. This is often addressed by a very modular design. The production capability must then have sufficient automation to manage a large number of modules that will be mixed and matched to derive a specific product.

Product derivation is the focus of a software product line organization and its exact form contributes heavily to the achievement of targeted goals. Deelstra, Sinnema, and Bosch define product derivation by, "A product is said to be derived from a product family if it is developed using shared product family artifacts. The term product derivation therefore refers to the complete process of constructing a product from product family software assets. [Deelstra 05]" Interestingly, this definition emphasizes the



commonality, the shared assets, among the products where much of the product line literature seems to focus on the variability among products.

The typical goals of a software development organization can be summarized as faster, cheaper, better. Software product line organizations usually have more focused goals such as reducing the cost of entry into a new market. The product line's business case defines exactly what the goal means and the business justification for allocating resources specifically for this goal.

The business case for a software product line uses a cost/benefit calculation to justify the investment in assets for the software product line. These assets include asset development and product building tools required to derive a product. For example, the business case for the organization might justify the development of a plug-in for the OSATE AADL toolkit to compute their specific measure of extensibility.

The costs for product derivation can be allocated to the major cost divisions in the SIMPLE economic model, shown in Figure 1 [Böckle 04]: per-product line (fixed) costs, and per-product (variable) costs.

The per-product line costs include organizational and core asset costs. The organizational costs include training costs for the derivation tools. The core asset costs include acquisition of derivation tools. Most of the derivation tools are per-product line costs since they will be used for all products.

The product-specific costs include the costs of selecting the appropriate variants.

The benefits include the value of a bigger market share but it should also include the value of the option presented by increased productivity of the development teams. Often the achievement of the strategic business goals brings specific benefits.

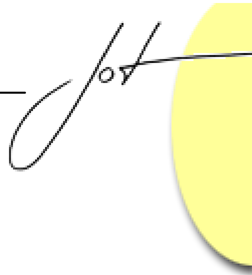
$$\sum_{j=1}^{nbrBen} B_{ben_j}(t) - C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n} (C_{unique}(product_i, t) + C_{reuse}(product_i, t))$$

benefits | per product line | per product

Figure 1 - SIMPLE

There are usually many ways in which the assets can be developed and the products built. Often the alternatives to the current way of doing business are not explored before a new effort is started. Planning for the production capability presents the opportunity to examine possibilities. Some derivation tools cost more than others and some tools will require more skilled personnel. During production planning, scenarios can be developed that are used to compare the costs and benefits of these different methods. The SIMPLE model supports comparing these scenarios and selecting the optimal one.

Figure 2 provides a mindmap that I have been working on. I make no claims for completeness, but it has been a help to me as I have thought about product derivation. The major nodes form the major divisions in this article. Product derivation actions in a product line organization begin with production planning and populate the twin threads of domain engineering and application engineering.



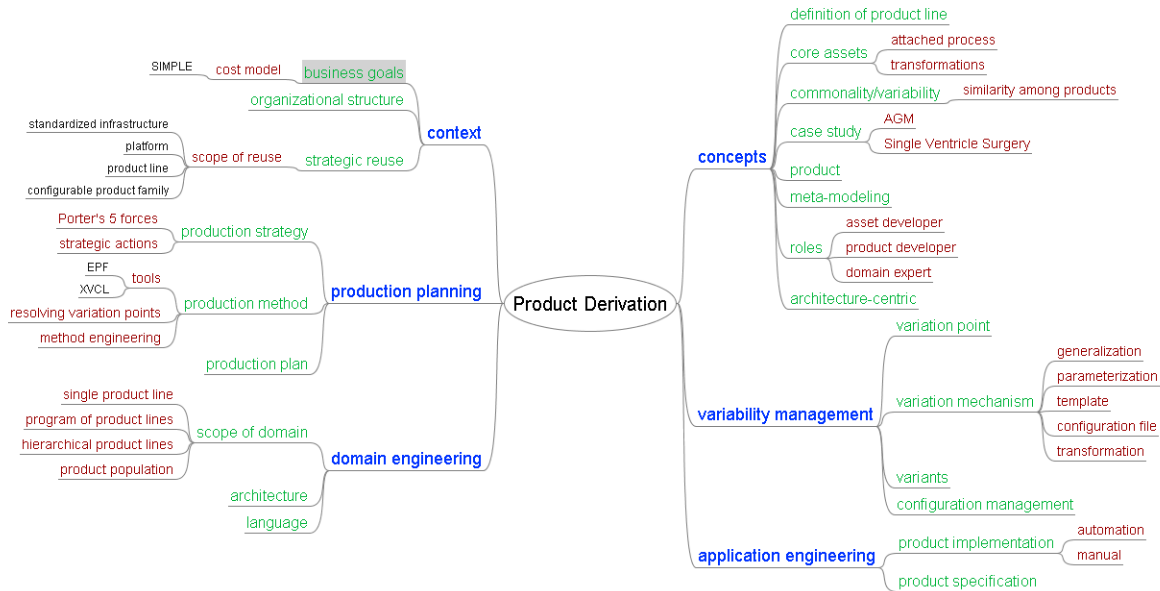


Figure 2 MindMap for Product Derivation

2 CONCEPTS

Our discussion of product derivation takes place in the context of a software product line being created and maintained by a software product line organization. I have quoted the definition for a software product line [McGregor 04] before and will present it without discussing it:

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 and that are developed from a common set of core assets in a prescribed way. [Clements 01]

There are several implications of that definition for product derivation. The "managed set of features that satisfy the specific needs of a particular market" focuses the domain sufficiently to allow for improvement in the quality of the reusable assets. The "common set of core assets" increases productivity dramatically, allowing cheaper products, because the assets have been designed with specific uses in mind. The "prescribed way" is embodied in a production plan that gives a detailed process for producing products from the core assets. Tailoring the production process gives the opportunity to select technologies, models, and processes that support the long-term goals.

Figure 3 shows that each core asset has an "attached process" that is the user's manual for the asset. The attached process for any asset that will be used to build a product is added to the production plan to help form the product-specific production plan. The attached process is the secret weapon in trying to reduce the costs associated with

reusing an asset. The attached process reduces the learning curve and speeds up the production process.

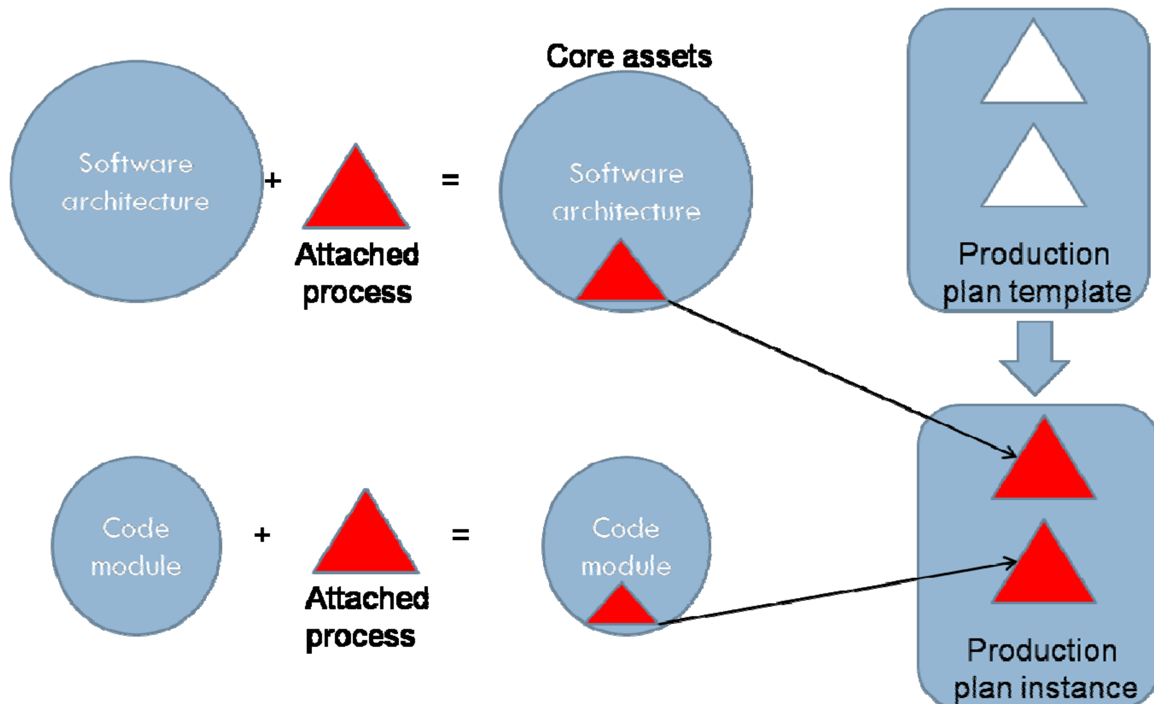
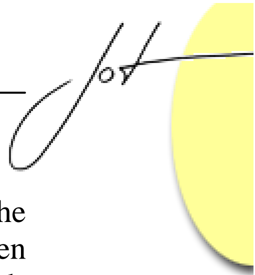


Figure 3 - Core assets and attached processes

3 PRODUCTION PLANNING

Production planning ensures that the product derivation process achieves the goals of the organization. Production planning coordinates the activities of the builders of reusable assets and the builders of products to reduce the risk associated with building these products. Production planning, which I discussed in [McGregor 06] [Chastek 09], treats the production capability as a system. Figure 4 shows a general use case diagram for such a production system.

Figure 5 shows the sequence of steps in production planning: production strategy, production method, and production plan. The production strategy is formed using Porter's Five Forces model for strategy development [Porter 98]. Each force is balanced by strategic actions. These actions are chosen to satisfy the production goals for the specific product line. For example, the plug-in architecture style makes it possible for users of a product to extend the base product themselves by downloading and installing plug-ins. This late binding approach adds a meta-feature, the ability to add features, to the product making it more attractive to users and making it more difficult for potential entrants to develop products that are competitive. This style was chosen by the Eclipse Foundation for their IDE and is a major factor in achieving the goals for that organization.



The second step is to use the strategic actions from the production strategy as the foundation for the production method by which products will be derived. I have written about method engineering previously and shown how it can be used to support a goal-driven approach [McGregor 04b]. How we want products to be assembled – who, how, where - will determine how assets should be implemented. Later binding of variants will require different mechanisms from earlier bindings. These different mechanisms will also have different properties. For example, runtime binding will usually degrade performance of the system and increase the risk of a product failure.

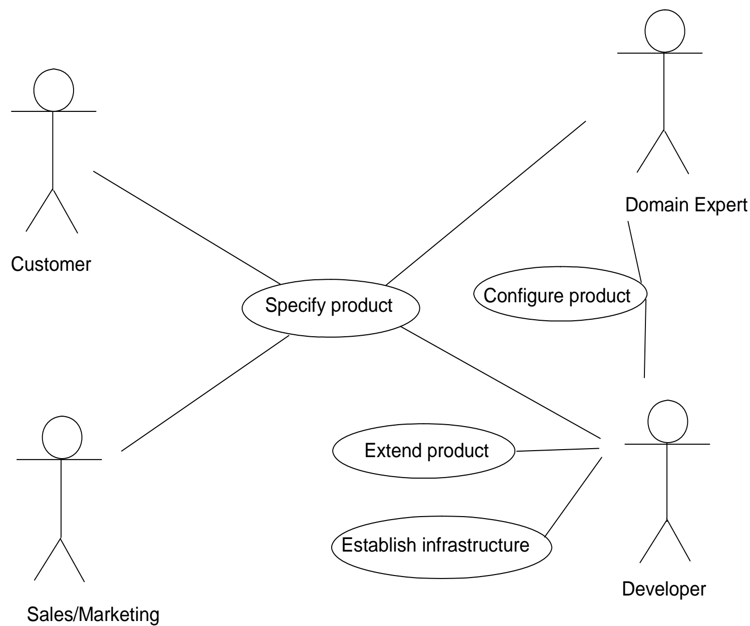


Figure 4 Use case diagram for production system

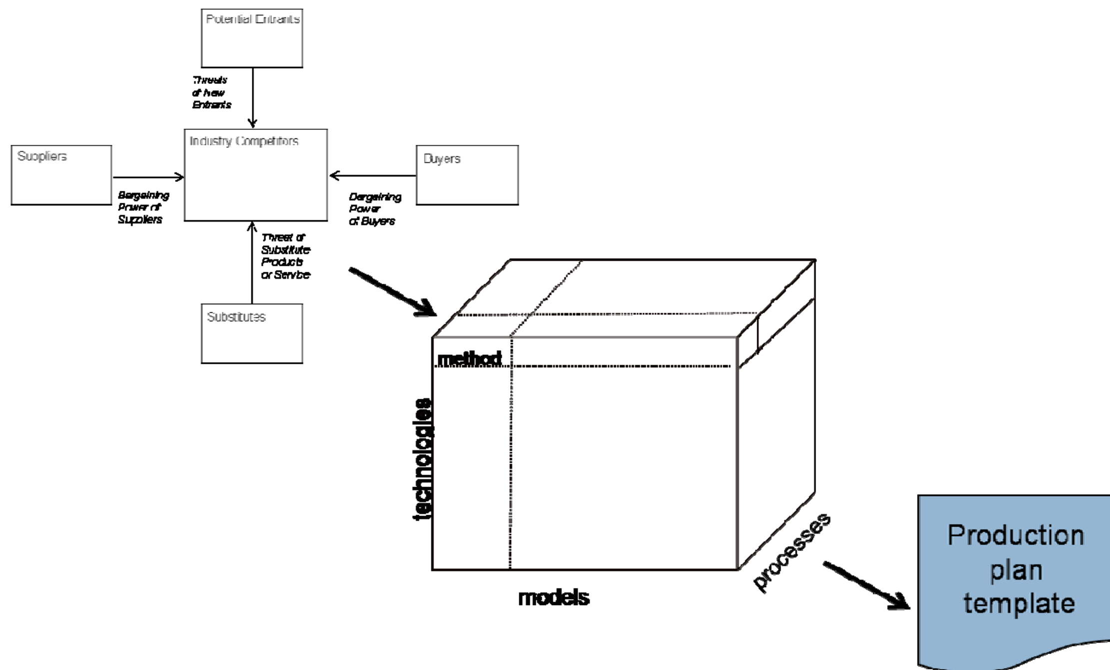


Figure 5 - Production planning

Deelstra, Sinnema, and Bosch identified several problems with product derivation by studying several industry efforts [Deelstra 05]:

Experts are overloaded

False positives on component compatibility

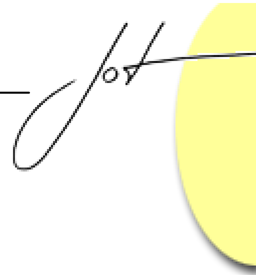
Configuration parameter errors due to large number of implicit dependencies

Redundant errors across projects due to failure to resolve dependencies explicitly

Over explicit documentation leads to traceability errors

The production method should address these problems and others. Method engineering provides techniques for tailoring a development method to fit the exact needs of the development organization [McGregor 04]. For example, the problem of experts being overloaded can be partially mitigated by capturing domain knowledge in a domain specific language (DSL). By associating pieces of semantics with the domain vocabulary, we tradeoff early heavier involvement of domain experts for future reductions in the use of domain experts.

The final step is to develop a template for the product production plan. The template is filled in with the attached processes from the assets that are selected to be a part of the product. The template can be implemented in a number of ways. The JET language [Eclipse 09b] provides a means of manipulating text, but the XML-based Variant Control Language (XVCL) [Jarzabek 03] provides a more powerful mechanism for combining a wider variety of types of information. There are many different types of information because the variations appear in the product assets as well as the supporting assets such as plans, tools, and process definitions.



4 VARIABILITY MANAGEMENT

Variability is easy, commonality is hard. It is easy to see the differences between two modules. It is much harder to talk about the parts of those modules that are the same or could be the same with refactoring. Even though we talk about commonality and variability analysis as though it is a classification scheme that divides the world into two parts, many assets have a part that is common and some part that can vary. In other words there are assets that are included in every product but that have significant variation within that asset.

Maybe variability isn't so easy. To derive a product includes selecting the variants that are bound prior to the product being deployed to customers and providing mechanisms by which the variants that are bound later can be selected. If the feature model has not been fully formed and validated, dependencies may have been missed. The resulting configuration may not be a valid product specification. As features are selected for a product, the software modules that are selected as a result determine the variation points. Feature modeling tools support the selection of features and create a model which captures the instantiation of the feature model. This is shown in the bottom half of the feature model in Figure 6.

In most cases just selecting a feature is not sufficient. Even manadatory features have variations in the details. Many of these are resolved very late in the product life cycle, maybe even during product execution at a deployment site. One useful way to think of this binding is in the context of partial evaluation [Consel 98].

The term "platform," as used by many software product line organizations, refers to the common portion but this gives the impression that the platform is unchanging from product to product. In most cases the platform is a portion of the product whose variability is controlled by parameters that are altered during initial configuration or between executions. The platform is an asset that has many variation points but that is thought of as a common element across products.

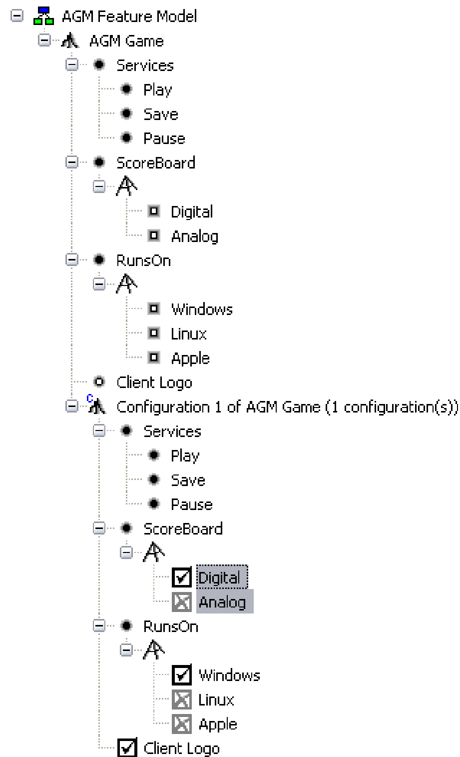


Figure 6 - Feature model

5 DOMAIN ENGINEERING

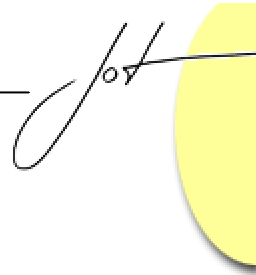
Domain engineering is the activity in which the product production capability of the product line organization is realized. This facet of the product line organization produces the reusable assets that will be used to build products. Whether or not the people who fulfill this role are integrated into the teams that actually build products or whether they are gathered into a group dedicated to building pieces, their output populates the product line architecture with software core assets.

Domain engineering is of strategic importance to the organization because it produces a set of core assets that are of sufficient scope and quality to handle all of the products envisioned for the product line. Product derivation will not be successful if the assets are incompatible with each other or require extensive rework due to shortsightedness. The strategic goals for the product line will almost always be stated in terms of product production but domain engineering will be critical to meeting those goals.

There are several road blocks to making product building efficient that can be addressed in the design and implementation of the core assets:

Dependencies among features or modules

Decisions among variants



Stability of the domain

Diversity and complexity of domain

These issues are partially addressed during the definition of the production capability. The production method will define techniques for capturing and representing dependencies among features and for relating decisions among variants to those features. Direct dependencies among features are usually captured during feature modeling. Selecting one feature may require the inclusion of another feature because of a "uses" relation and selecting one feature may require that another feature be excluded. These relationships capture some of the semantics of the domain. If there is a large number of dependencies, it may be possible to refactor the feature model, reducing the scope of some of the features, to eliminate some of the dependencies and to localize others.

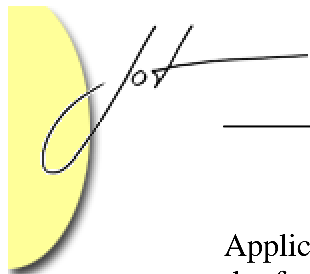
The assets related to derivation include the software core assets, the product line architecture, the production plan, and derivation tools. The derivation tools needed include tools associated with the variation mechanisms that are selected for use, software development tools, and supporting tools such as configuration management tools.

Due to the ability to amortize effort over multiple products, domain specific languages (DSLs) are attractive core assets. Once developed, a DSL can be used by persons with much less domain knowledge than the language developers. This supports a product derivation process implemented by less expensive people, often taking less labor time total than comparable products. Eclipse provides support for creating DSLs [Eclipse 09]. In a previous paper we provided a detailed example that not only used a domain specific language, it was used for both requirements and test cases [Chastek 05]. DSLs make automation of a number of engineering tasks much easier because the language is more limited than the general modeling language from which it is derived and is based on a formal meta-model that makes transformations more straightforward.

Topcased, built on top of Eclipse, provides very accurate implementations of the SysML, UML, and AADL standards [Topcased 09]. (They all continue to evolve.) These languages and supporting tools facilitate producing a sequence of models of the system from use cases for system engineers to detailed architectures for architects from which code can be generated. This very detailed modeling provides support for quality control from the beginning to the end of a project.

6 APPLICATION ENGINEERING

Application engineering is the activity of operating the product production capability. Regardless of the flavor of software product line strategy the organization adopts, the goal is to maximize the efficiency of product derivation. Rather than begin with a blank slate the team is operating within the environment provided by the domain engineering function. That environment contains assets and methods that ensure the products meet the goals of the product line organization.



Application engineering is obviously of strategic importance since product production is the focus of the organization. The production capability has been shaped to support the creation of products in ways that meet the goals of the organization. Operating that capability provides the product line organization with the anticpated benefits.

The production plan provides the application engineer with the instruction manual for building a new product. The attached processes of the selected core assets become part of the production plan and specialize the plan based on these selections. In the best case each attached process contributes a portion of an automated script that builds the product. At the least the attached process reduces the learning curve for a product builder who must manually integrate the asset into a product.

Application engineering is a straight-forward activity that can not help but achieve the goals. The production method, and the core assets, is imbued with the qualities required to achieve the business goals due to the production planning phase. Assembling the product may be a time consuming process in some cases, but the outcome is made to order.

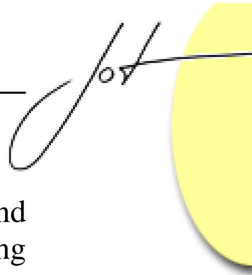
Realistically there is almost always a portion of the product that is new and different and cannot be realized directly from existing assets. The production capability has to be sufficiently powerful to support evolving an asset to meet this new need and sufficiently flexible to integrate newly developed software with the pre-existing assets.

7 SUMMARY

Product derivation is the focus of a software product line organization. "Goal-driven product derivation" adds the intent to design a derivation process, which has specific qualities, that aids the organization in achieving specific goals. In several columns and other papers I have described pieces of product derivation. In this column I have tried to tie these together into a process for preparing for product derivation. The process begins early in the life of the product line with production planning - strategy, method, and plan. The intent of this process is to define a production capability that facilitates product derivation while achieving the goals attributed to product production.

Once a plan for the production capability is defined, the organization establishes that capability by implementing the core assets referred to as product parts. The completeness of that base, before the first product is derived, depends on the adoption strategy chosen by the organization. Most often the core asset base will evolve toward completeness as more products are built.

Deriving a product, given the production capability, is simply operating that capability. This sequence of events is not random, it is a strategy that takes advantage of designing "overhead" actions like planning, which will occur infrequently, to be sufficiently heavyweight to allow derivation of individual products, which will occur very frequently, to be very lightweight.



Products are of strategic importance to a software product line organization and systematically planning, creating, and operating a production capability faciliates meeting the strategic goals of the organization.

REFERENCES

[Böckle 04] Guenter Böckle, Paul Clements, John D. McGregor, Dirk Muthig and Klaus Schmid, “Computing Return on Investment for Software Product Lines”, IEEE Software, July/August 2004.

[Chastek 09] Gary J. Chastek and John D. McGregor. Modeling Variation in Production Planning Artifacts, VaMoS 2009.

[Clements 01] Paul Clements and Linda Northrop: *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[Clements 05] Paul Clements, John D. McGregor, and Sholom G. Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE), Software Engineering Institute, CMU/SEI-2005-TR-003.

[Consel 98] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E.-N. Volanschi, J. Lawall, and J. Noyé. *Partial evaluation for software engineering, ACM Computing Surveys (CSUR)* Volume 30 , Issue 3 (September 1998).

[Deelstra 05] Sybren Deelstra , Marco Sinnema and Jan Bosch. *Product derivation in software product families: a case study, Journal of Systems and Software,* Volume 74 , Issue 2 (January 2005).

[Eclipse 09] Eclipse Foundation, www.eclipse.org.

[Eclipse 09b] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/, 2009.

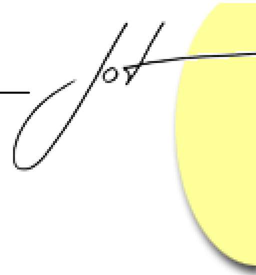
[Jarzabek 03] S. Jarzabek, P. Basset, H. Zhang, and W. Zhang. XVCL: XML-based variant configuration language. In Proc. Int. Conf. on Software Engineering, ICSE03, pages 810–811, Portland, OR, May 2003.

[Jensen 09] Paul Jensen. Experiences With Software Product Line Development, Crosstalk, January 2009.

[McGregor 07] John D. McGregor "Openness", in Journal of Object Technology, vol. 6, no. 6, July - August 2007, pp. 7-14. http://www.jot.fm/issues/issue_2007_07/column1/

[McGregor 06] John McGregor: “Planning before plans”, in Journal of Object Technology, vol. 5, no. 2, March-April 2006, pp. 27-34. http://www.jot.fm/issues/issue_2006_03/column3/

[McGregor 04] John D. McGregor: “Software Product Lines”, in *Journal of Object Technology*, vol. 3, no. 3, March-April 2004, pp. 65-74. http://www.jot.fm/issues/issue_2004_03/column6/



[McGregor 04b] John D. McGregor. Factors in Engineering Strategically Significant Software Development Methods, OOPSLA Workshop on Method Engineering, 2004.

[McGregor 04c] John D. McGregor. Product Production, http://www.jot.fm/issues/issue_2004_11/column7/, 2004.

[Porter 98] Michael E. Porter. Competitive Strategy, Free Press, 1998.

[Topcased 09] Topcased, www.topcased.org.

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University, a visiting scientist at the Software Engineering Institute, and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.