Preparing for Automated Derivation of Products in a Software Product Line

John D. McGregor

September 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-017
ESC-TR-2005-017

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Preparing for Automated Derivation of Products in a Software Product Line

CMU/SEI-2005-TR-017
ESC-TR-2005-017

John D. McGregor

September 2005

Product Line Practice Initiative

Table of Contents

List of Figures

Abstract

Organizations that adopt a software product line strategy often have business goals that concern improving their ability to produce products by lowering product development costs, by reducing the time to bring a product to market, or through other production improvements. Business goals such as these make automated product derivation an appealing strategy to a software product line organization. Automating production requires up-front investment, including the creation of both the core assets that will be assembled as products and the core assets that will perform the assembly. A software product line provides the ability to amortize the cost of the infrastructure over a set of products. This report views the process for automating the production of products in the context of a product production system. The process begins with the decision to automate, proceeds to the selection of the automation approach, and continues with the operation and management of the automated production capability. The process is illustrated by a case study automating the production process in the Carnegie Mellon® Software Engineering Institute's pedagogical product line.

1 Introduction

Software product lines have produced promising advances in lowering costs and reducing the time required to produce a set of products. They deliver these results by taking advantage of the common behaviors among the products to be produced and by managing the variations in behavior among the products. Software product lines increase productivity by exploiting both economies of scope and scale.

One of the potential benefits of software product lines is the ability to use mass customization to serve a large number of market niches at a fraction of the usual price. Automatic creation of products makes realizing those benefits more probable in many ways, including moving product development into the hands of domain experts rather than software experts.

The work in a software product line is usually divided into two separate high-level activities: (1) developing core assets (the “things” that are needed to build products) and (2) developing the products using the core assets. The core asset builders accommodate variation among the products by identifying what should be allowed to vary and designing in mechanisms that can be used to select a particular variant quickly. The product developers are sometimes required to create additional, unique software to realize the portion of the product that differentiates it from other products in the product line.

In a software product line, a product is derived from the existing core assets by exercising the variation mechanisms defined in the product line architecture. Automatic derivation refers to the use of a set of tools to both specify a product and to transform that specification into a product using the core assets. There is a spectrum of such techniques from the simple assembly of existing assets using scripts to the generation of the product components from fine-grained elements. These techniques can be classified along two dimensions: (1) completeness of the coverage of product functionality and (2) completeness of the implementation of the covered functionality. For example, some tools generate user interfaces or entire clients but do not provide support for developing the servers needed to complete the system. There are techniques for developing a single level in a multilevel architecture. This report will address each of these dimensions.

If an entire product can be derived automatically, it is because the product line’s core assets are sufficient to produce all the specified behaviors and all the variation values can be predetermined or used as input. In some cases, the derivation simply binds existing components together into predefined configurations. In other cases, the components themselves are generated from fine-grained elements. Such elements provide greater flexibility in the production capability, but more up-front investment is required to achieve acceptable results.

Automatic derivation is an old and widely used technique in software engineering. Programmers routinely derive object code automatically from source code using a compiler. Active server pages (ASPs) are derived automatically from templates and instantiation data. Both examples are transformations based on specific languages. Each transformation takes an input written in a grammar and produces a predefined output. More recent technologies do not change this basic pattern; they simply raise the level of abstraction higher so that products are developed using fewer constructs.

Automated derivation in a product line follows a similar process, as illustrated in Figure 1. The process begins with the development of a product specification. The appropriate values at each variation point are determined and included in the specification of the component that contains that point. Each specification is transformed automatically into a form that is closer to, or may be, the final executable product. There may be a series of these steps where each specification is written in a different language and each transformation is performed by a different tool. The output of each transformation moves closer to the final, executable product.

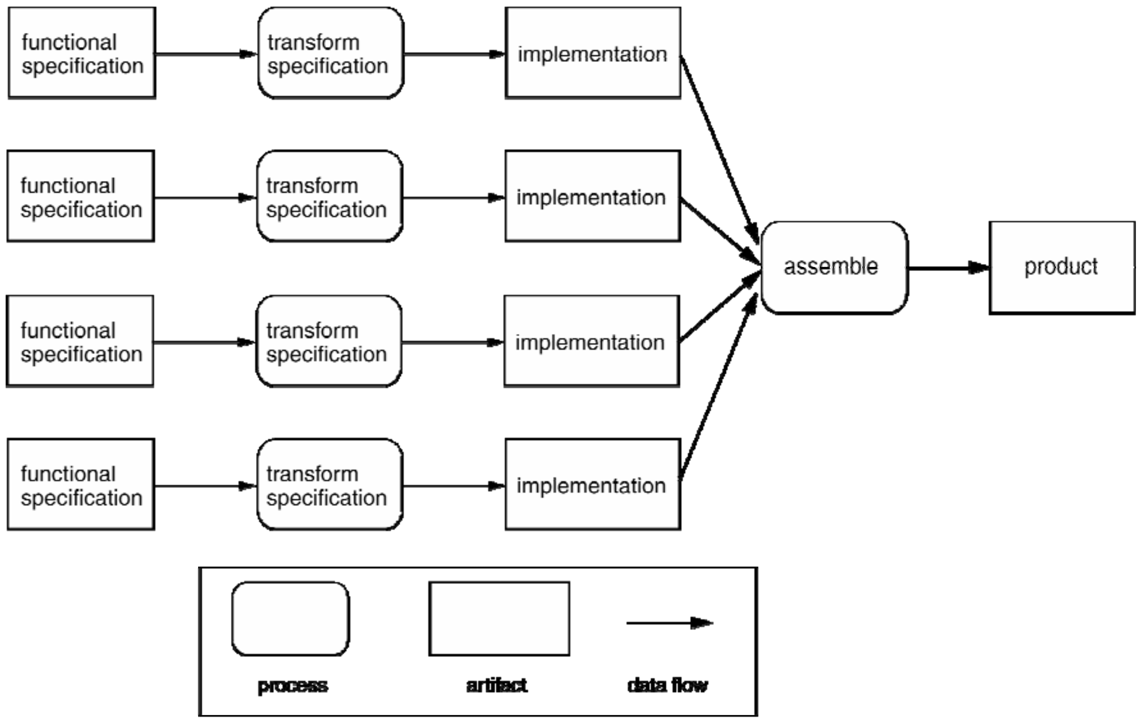


Figure 1: Chain of Derivation

This report provides an end-to-end view of the activities that are needed to support the automatic derivation of products within a software product line. The portion of the derivation process that is automated varies from one product line to another. However, in all cases, for the portion that is automated, the product implementation is generated automatically from some form of specification. The difference is the exact form of the specification and the point in the development process at which it is defined.

This report addresses *product* derivation rather than *program* derivation. Program derivation usually means the automatic creation of code from some formal specification. Automated product derivation addresses issues about the ability of existing assets to be composed and the applicability of those assets to the needed product. The assets may have been created using program derivation techniques or manually, but, in this report, they will be viewed as composable units without regard to their origins. This report goes beyond code to encompass all elements that commonly constitute a product (e.g., documentation for users and maintainers). This report will accommodate a range of ways in which assets become available for use in product production.

This report is organized to follow the steps from the earliest phases of software development to the later ones. Section 2 describes the context in which automatic derivation is appropriate. Section 3 contains a discussion of the initial decision to automate product derivation. Section 4 describes techniques for choosing the appropriate derivation technology, and Section 5 discusses carrying out the derivation process. Section 6 contains a brief case study based on the Carnegie Mellon® Software Engineering Institute's (SEI's) pedagogical product line.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

2 Context

The success of a product line depends on its ability to produce products that satisfy the organization's goals for its market or mission. These goals and how the product line organization has decided to satisfy them establish the context in which automated derivation will be applied.

2.1 Product Production Capability

This report is part of a continuing effort that is addressing issues about product production in a software product line organization. This previous work described several production-related assets: the production strategy, the production method, and the production plan, all of which are illustrated in Figure 2 [Chastek 02a, Chastek 02b, Chastek 04]:

- production strategy – The production strategy is the direct link between the business goals and product production. The strategy is a high-level statement that serves as input to the creation of the production method. It conveys the high-level specification of how core assets should be built, so products can be built that achieve the product line goals. This specification drives the creation of other production-related assets.
- production method – The production method bridges the gap between the production strategy and the production plan. The production method is engineered from the production strategy to define the processes, technologies, and models that will be used to produce products. The production method provides a high-level implementation definition of how products will be created. Defining the production method creates the information needed to produce a production plan.
- production plan – The production plan is the operational view of the production strategy. The plan for a product is created by applying the production method to the set of core assets selected for the product. The production plan identifies the techniques to be used, the schedule for using them, and the materials needed to build the product. A generic production plan is created as a core asset and supplied to product-building teams, who make the generic plan more specific to produce the product-specific production plan. The specialization is performed by composing the attached processes from each of the core assets chosen for developing the product.

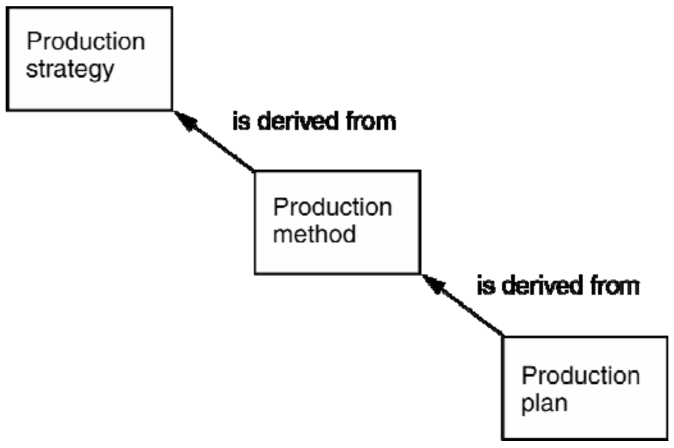


Figure 2: Relationships Among Production Elements

The decision to use automatic product derivation is a strategic decision based on those product line goals that affect product production. The decision is made very early in the product line life cycle, usually during the business case analysis, since it affects the entire product line, including the fundamental organization and cost of the project. The production strategy is the basis for creating the production method. The method defines the processes, tools, and models used to automate production. The production plan provides the basic instructions for applying the production method to develop a specific product. For a product line using automatic derivation, the production plan may be as succinct as a script that drives a set of tools leading the product builder thorough the product-specification process and that generates the product.

Figure 3 shows a basic use case diagram for a production system using automatic derivation. The uses in the diagram, which are described below, illustrate the activities that can occur in automated derivation:

- specify product – Even the most automated production system provides a means for the system's user to enter the specification of the desired product as input.
- configure product – Some automatically derived products will require additional configuration after the production process is complete. This additional configuration is usually accomplished by someone with domain knowledge (such as a domain expert or a developer with experience in the domain).
- extend product – The production system may be partially automated, and many products may require unique extensions to satisfy the complete specification.
- establish infrastructure – A number of tools are required to support automated derivation. These tools and the basic assets they will manipulate to build products must be made, commissioned, mined, or bought.

This report uses these use cases as one of its organizing elements.

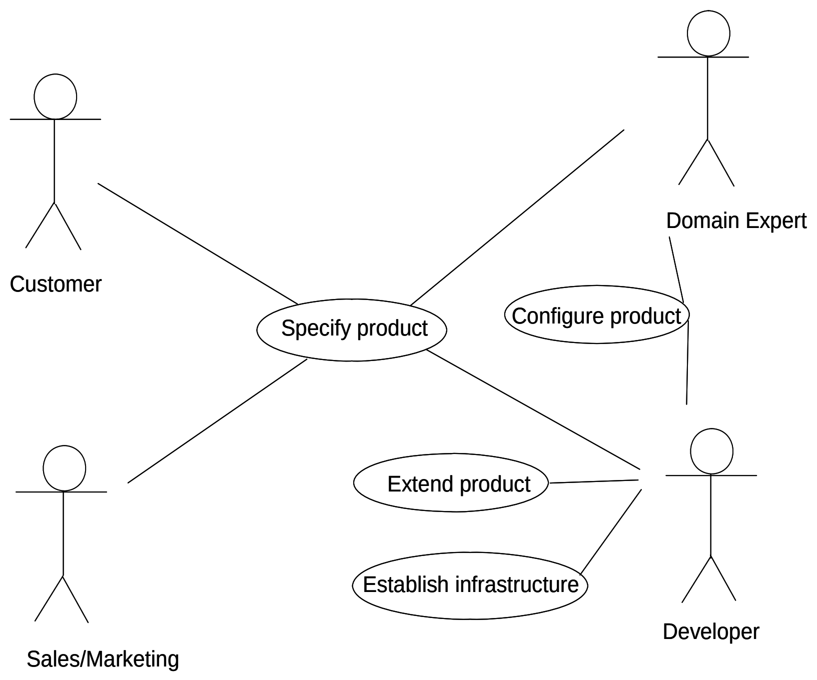


Figure 3: Use Case Diagram for Production System

2.2 Practices

Organizations that are mature in software product line practice areas, such as “Requirements Engineering,” “Understanding Relevant Domains,” “Architecture Definition,” and “Configuration Management,” will be better able to take advantage of automated derivation than those that are not.

The product line organization must be proficient at requirements engineering to use automated derivation effectively. A thorough analysis of the full scope of requirements and a recognition of the variations across products are necessary to build core assets that are sufficiently flexible. Building an infrastructure that is insufficient in scope will be both more likely and more costly if the requirements analysis is insufficient.

An organization that has a mature understanding of relevant domains within its scope will be more likely to be successful with automated product derivation. There is a tradeoff involving the additional resources required to develop the infrastructure needed for automated derivation. The relative difference in the resources needed for manual and automated derivation must be compared to the velocity at which the domain is evolving.

The amount of effort needed to create the infrastructure for automatic derivation will depend on the maturity of the organization relative to the domain. A company just beginning to develop products in a domain will take longer to establish the shared models necessary to support the needed abstractions than a company that has already developed those models.

Having a history of architecture definition contributes to success with automatic derivation. First, the practice helps establish the needed level of domain maturity. Second, it prepares personnel to recognize and be able to benefit from multiple levels of abstraction. The components that are composed to form products reflect the product line architecture and are designed to provide specific levels of quality when used correctly with the architecture.

The configuration management process in a product line using automatic derivation changes in that the elements under management are no longer programming language files, in the usual sense. That is, the components that are assembled into the product may be generated from fine-grained model pieces, as necessary. Modifications will be made to those fine-grained model pieces and not to the generated components.

2.3 Process Changes to Accommodate Automatic Derivation

Using automated means of product production affects the processes for developing both the core assets and the products.

2.3.1 Core Assets

An automated product derivation strategy puts more emphasis on core asset development than traditional product development does. The core assets must be more complete for that portion of the product that will be automatically created. The core asset developers construct the infrastructure for automated product production as well as the core assets that will be developed to create products. The infrastructure must not only assemble the components needed, but it must also ensure that inappropriate combinations of assets are prevented. When automatic derivation techniques are used, the architecture and design processes for the core assets place a higher priority on the identification of constraints among the assets.

The core asset development process encompasses meta-level constructs requiring a different set of skills. When using automated derivation, core asset developers require skill sets such as domain modeling and even building generators of generators. The set of core assets represents a more complete set when automatic derivation is the goal.

2.3.2 Products

Product building is faster and the production plan simpler in a product line that uses automated derivation. After the product specification is complete, producing the product is usually a matter of minutes depending on the size of the product. The derivation infrastructure captures the production plan in a tool set, so product building is tailored to the skill set of the personnel assigned to build products.

The production process can be operated by personnel with more domain expertise and less development skill than a more manual development process. A marketing representative can produce a product on the client's site with the client's direct input. Moving production closer to the customer changes both the marketing and production processes. The software factory approach [Greenfield 04] would even put product production into the customer's hands. With a software factory, the product line organization delivers a software tool to the client that is capable of producing the range of products within the product line scope.

3 Deciding to Automate

Deciding to use automatic product derivation is a strategic decision that has economic and technical dimensions. The production strategy imposes constraints on the production method. Deciding to automate is based on satisfying these constraints.

To ensure that the core asset developers produce core assets and the product builders build products that meet the product line's goals, a software product line organization makes plans. As described in the previous section, certain production-related assets are created early in the life of the product line as a way of planning how products will be produced. Planning for production essentially involves deciding how products should be built before creating the core assets that implement production.

As part of choosing production techniques, models are created to evaluate the feasibility of various options for producing products. In this chapter, we examine a cost model in detail because many of the other factors, such as time to delivery, can be reduced to economic arguments. Cost functions, for example, can account for the stability of the domain by returning a lower maintenance cost for a more stable domain.

3.1 Economic Model

The Structured Intuitive Model for Product Line Economics (SIMPLE) [Clements 05] provides a framework for creating cost-benefit models for decision-making support in software product line organizations. The basic SIMPLE expression is given in Equation 1.

Equation 1: Basic SIMPLE expression

$$C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n}(C_{unique}(product_i, t) + C_{reuse}(product_i, t)) + \sum_{j=1}^{numBenefits} benefit_j$$

An economic model that compares costs and benefits of an automated derivation approach to a manual assembly process can be built using SIMPLE. The basic SIMPLE expression given in Equation 1 is instantiated for each of the two approaches, and the values of the two expressions are compared to determine the approach with maximum return. The "product" parameter in the equation is the specification for an individual product. This parameter is used by the cost function to compute analyses that evaluate each asset that will be needed to satisfy the specification. The "t" parameter denotes the time at which the product is expected to be built. This parameter is used by the cost function to compute analyses that consider the time value of money, the allocation of scarce resources, and the market timing of product releases as part of the cost expression.

Each term described below is intended to be broken down into individual expressions that can be estimated more exactly for the given situation. The following assumptions and constraints apply to the cost functions when using automatic derivation:

- C_{unique} is the cost of developing the unique portion of the product. It will be very small, and in some cases zero, under the automatic-derivation approach. That is, for automatic derivation, we assume that little or no additional, unique development is needed to produce the product. There is usually a need to conduct system tests, since each product is a unique assembly of assets and has the potential to contain unique asset interactions. Even though the assets have been inspected and tested individually, unexpected behaviors may emerge when the assets are assembled. Some of the traditional cost in this category is shifted to the cost of building the core assets because the core asset base must more completely cover the behaviors of products in the product line.
- C_{reuse} is the cost of reusing the product line assets. This term should represent how long it takes the designated people to make variation decisions. For an automatic-derivation approach, this cost includes using the provided variability selection mechanisms to specify which assets are to be used for the product and determining any parameter values used for those assets. This cost is usually lower for automatic derivation than for a traditional product-building approach partially because some of the traditional reuse costs are shifted to the cost of building the core asset base. That is, product features or behaviors are selected rather than specific core assets. The infrastructure for automatic derivation, which is itself a core asset, does the mapping and selects the appropriate core assets. Having the infrastructure create a mapping and select the cores assets reduces the cost of reuse but increases the size and cost of the core asset base.

  In automatic generation, the selections may be made by domain-knowledgeable people as opposed to software development personnel, but, in either case, the selection of core assets is more indirect than for a traditional product-building approach. Depending on the domain, the domain experts may cost more or less than the development staff. Even if the experts cost more than development staff, they may be able to make selections more quickly with fewer errors.
- C_{org} is the cost of converting an organization to the product line strategy. If the organization is already using product lines, this amount will be small, covering only the cost of training personnel to use the variability selection mechanisms and tools unique to automatic derivation. If this is the organization’s first product line, this cost will be basically the same for both the manual and automated techniques. This cost represents a tradeoff between the cost of training nondevelopment personnel to use mechanisms such as Web browsers to select options and the cost of training development personnel to implement product-specific assets.

- C_{cab} includes the basic costs of producing all the core assets, including the software architecture and the software components, that are used to produce products. The core asset base becomes the cost focus when automatic derivation is used. The automatic-derivation approach includes the additional cost of building the derivation tool chain. In addition, this cost reflects the previously mentioned transfer of the cost from building the unique pieces to the cost of the original development of the core assets and the transfer of the cost from reusing the assets to making them composable.

This SIMPLE expression indicates that the automated approach will be profitable if the savings realized in producing unique pieces and having personnel make variation point selections are greater than the sum of the additional costs of the derivation infrastructure and the additional cost for identifying and developing the core assets to the level required for automatic derivation.

3.2 Technical Issues

Several technical issues must be considered as part of the decision to automate product derivation. Automated derivation encodes more of the knowledge and techniques used by product developers into a tool infrastructure than non-automated derivation. The more complete the automation is, the more investment that will be required in encoded knowledge.

3.2.1 Stability

The stability of the product definition will affect how often the infrastructure will need to be revised. Deciding to automate too early in the organization’s experience with the domain may lead to many iterations on the infrastructure.

The rate of change in the main application domains affects the frequency with which the underlying principles must be modified. The faster the changes in domain knowledge occur, the more maintenance that will be required on the automation infrastructure. These frequent changes will offset some, and perhaps all, of the potential savings from automating.

The rate of change in the scope of the product line has implications for the feature model on which the automation is based. In particular, variants at very different places in the feature tree may disappear if the scope is narrowed, or they may need to be inserted if products are added.

The maturity of the architecture affects the stability of the derivation infrastructure. Some automated techniques essentially encode the architecture in the derivation tool chain. Changes to the architecture can lead to numerous changes in the infrastructure.

3.2.2 Completeness

The derivation infrastructure is created from a set of requirements, but often the infrastructure is not intended to implement the entire production process. The product line organization may initially make the investment to automate the derivation of only a portion of the product or to automate only a portion of the production process. To ensure the correctness of the resulting product, that automation should result in a conceptual whole that can be evaluated for completeness.

One way to ensure the correctness of the product is to automate either all or none of an architectural unit. If complete architectural modules are automated, the test cases that have already been created from the architecture description can be used to test the generated assets. Most generated code is not very human readable. Planning on having product builders modify partial code is seldom productive. While it might be possible to limit automation to the production of complete modules, it is usually easier to place the requirement on the front end of the process by selecting complete architectural units to be generated.

If only a portion of the production process is automated, the automation should cover all possible cases for that portion. Each phase in the process has specific inputs and outputs. Each input provided to a process phase should be complete to ensure that the resulting output is also complete.

4 Selecting the Automation Approach

Once the decision has been made to automate product derivation, the next step is to select the technology to use. There are a number of possible choices, each possessing its own qualities. The decision as to which approach to select is made in the process of planning the overall production capability. This section follows the planning process and illustrates the sequence of decisions.

Automatic derivation requires considerable infrastructure. Typically, the infrastructure supports the specification of products and the transformation of that specification into working code. The choice of the technology for writing the product specification affects the choice of technology for transforming that specification, and vice versa.

4.1 Decision-Making Criteria

4.1.1 Product Qualities

Automatic derivation relies on a well-specified, stable software architecture. Much of the infrastructure created for automation encodes the architecture as an integral part of the artifacts that are assembled into the product. When choosing an automation technology, the architecture assumed, or created, by the technology must be evaluated to determine whether it is capable of achieving the expected product and product-production qualities.

For example, the GenVoca work of Batory and colleagues is based on a layered architecture for the generated components [Batory 00]. Each component is composed of layers of behavior that are combined in specific orders. This architecture supports the automated assembly of fine-grained pieces into components that are, in turn, assembled into products. The resulting components will have specific attributes. The core asset team must determine whether those attributes are the appropriate qualities for the products to be built in the product line.

4.1.2 Production Qualities

Different automation technologies require different skills and provide different benefits. The production qualities required in the charter of the product line are used to evaluate the automation technologies.

For example, choosing to create and use a domain-specific language results in a production environment in which product builders must understand features but few, if any, implementation details. The personnel hired to operate this production process will have training in the domain rather than software development. The products will be easy to produce as long as the entire feature set is within the product line scope but more difficult if some of the features are outside that scope.

4.2 Automation Approaches

We first discuss approaches to product specifications that are at a high level and generally applicable. We then consider approaches in which the specification technique is closely related to the generation technology.

4.2.1 Specification Approaches

A product-specification technology is selected that is compatible with the choices made during product line analysis. Chastek and Donohoe [Chastek 01] discuss two strategies for building the requirements model:

1. feature-based approach – A feature-based product specification can be used when a feature model has been created. The feature-based approach is appropriate when end users, marketing personnel, or other non-development personnel will be operating the production system because the feature level is the level most visible to users.
2. use-case-driven approach – Products can be specified by selecting the applicable requirements when a use-case-driven approach has been used to capture the product line requirements. This approach is more appropriate when development personnel will be operating the product production system because the use cases often capture information that is related to development. Use cases also tend to be more comprehensive, capturing the views of all stakeholders, not just users.

Other approaches described in the literature include the domain-based approach. This approach is similar to the feature-based approach, but it uses the more general, and usually more technical, domain model as the basis for specifying products. The domain model is usually more technical than a feature model in that it is more detailed and includes information that is not obvious to users. This approach is appropriate when domain experts will be specifying products.

The more abstracted the specification is away from the actual implementation, the more decoupled the product specification is from the product implementation. Decoupling the product specification from the product implementation usually simplifies the specification language and may reduce the skill set needed to develop a correct specification. Although high-level specification languages can be used even if implementation is manual, the infrastructure needed for automatic implementation can usually be extended easily to support checking specifications for completeness, consistency, and correctness. This type of checking is accomplished by using a grammar for the specification language and including constraints on the elements of that grammar. This checking has the effect of decreasing C_{reuse} in Equation 1 but not increasing C_{cab} by an equal amount. In addition, this checking results in a lower cost to use an asset in a given product.

4.2.2 Intelligent Build

The quickest approach to automate some of the derivation process uses an intelligent make system, such as Ant, to assemble products automatically from a manually derived specification [Apache 04]. Each product is specified as a script that assembles the required components. The scripting language is often dedicated to the build process and may be specific to the platform (e.g., UNIX make) requiring a specific skill in script writing. In the case of Ant, however, the product specification is written in Java.

Each new product requires the development of this script, similar to developing the product specification. And, just like the product specification, core assets such as a script generator are made available to facilitate this task. The script is used repeatedly during development when components are being revised rapidly and builds must be accomplished rapidly. The script is developed incrementally as new components are integrated into the build. The degree of commonality among products determines the percentage of the script that will be modified.

A product is realized from a script by executing the script. In the case of Ant, this execution involves running a Java program. Many scripting languages have platform dependencies, making them unsuitable for multiplatform development. The Java dependency of Ant makes it suitable for running on any platform that supports Java. Ant also makes managing the tool chain easy because the main driver program can invoke a variety of tools in the programmed order.

This approach is largely independent of product tailoring. Since each build script is hand built, often by cloning an existing script, new or revised components can be included in the product with almost the same effort as existing components.

This technique is suitable for individual products and platforms that are relatively stable. In a mature product line, building this script may take more effort than developing a product specification in a domain-specific language (DSL); however, developing the DSL is a major effort. The make tool may not have any checks on whether the specification is correct, complete, or consistent, whereas a DSL should have tools that make consistency checking possible. On the other hand, the generic make tool is ready-made and does not require the effort needed to build the derivation infrastructure.

Diaz and colleagues describe a detailed scheme for composing Ant build definitions to implement an automated production plan [Diaz 05].

4.2.3 DSL and Product Generation

Domain-specific languages provide a vocabulary that is accessible to someone whose expertise is in the domain of the application rather software engineering. A DSL provides deep support for automatic derivation of products. This support includes the language in which a product specification can be written and tools that support checking specifications for correctness and completeness.

DSLs support the automation of transforming the specification into executable code, but the specification process is still largely manual. The specification is a program but one that is easier to write than the complete executable program. The program is written in the DSL where fewer constructs are needed to express the solution than in a traditional programming language.

The transformation process, in this context, is usually a multipart process in which the DSL code is translated into a conventional programming language. The new program is then translated using the standard tools for that language. Often, the DSL is tied to code fragments, so the transformation process generates the components "on the fly" while producing the intermediate code.

Tailoring a product can be difficult in this environment. If the DSL is incomplete, traditional programming expertise may be required to add the necessary semantics. Additional components and modifications to the generator may also be required.

4.2.4 Metamodeling

Metamodeling is one approach to developing a DSL and providing automatic code generation. We're including it in this report because it has sufficient power and applicability to product line development.

In this approach, the product line is modeled by a metamodel, which essentially constrains derived models to the types of products described by the product line scope. The model provides a fine-grained factorization of the concepts in the domain, so they can be assembled easily and composed in a large number of ways.

Each product is specified by creating a model based on the metamodel. This process is successful if both of the following statements are true:

- The metamodel provides a sufficiently factored view of the domain to allow the modeler to state the requirements for the product accurately.
- The metamodel correctly constrains the relationships among elements in the metamodel.

The metamodel must separate common behavior from variation points so that including required behavior does not result in including optional behavior as well. Metamodels typically handle this separation by defining a platform that is a black box of common behavior.

A typical metamodeling environment provides a hierarchy of models. The hierarchy defined by the Meta-Object Facility (MOF) of the Object Management Group (OMG) is shown in Figure 4. As you move up in the figure, each level constrains the scope of the levels below it. The meta-meta level defines a modeling language that can be used to define metamodels.

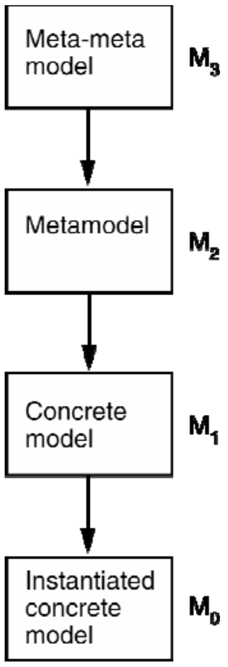


Figure 4: Meta-Tower for the MOF

The product is realized by applying a transformation to the model-based specification of the product. Some metamodeling environments then use code generators in the form of report writers to produce product code. This approach makes it easy to automate generation of documentation and other supporting materials.

The development of the complete meta-tower, the fully factored domain model, and the code generation infrastructure are the major expenses of this approach. Setting up the infrastructure can be more expensive than the usual product line asset base, but savings will be realized during production. The advantage of the tower is that each concept is pushed to the highest possible level, so changes in scope or variation points can be handled as globally as possible.

Product tailoring affects different levels of the modeling hierarchy depending on the complexity of the change. A change to the specification of a product is accomplished at level M_1. A change that requires a modification to the scope of the metamodel is accomplished at level M_2. At the least, changes at level M_2 require knowledge of the metamodel and possibly programming skills.

4.2.5 Frame Technology

A technology based on Bassett's frame technique provides another approach to a metamodel [Zhang 05]. The technique uses the ideas of Minsky, in which knowledge is represented and retrieved as a series of frames. The domain model is translated into a hierarchical network of frames. Frames at one level of the hierarchy “adapt” frames at the next level of the hierarchy to produce more complex entities by using a template-instantiation paradigm.

XML-based Variant Configuration Language (XVCL) is a technology that uses an Extensible Markup Language (XML)-type language to support the frame technique. (See Figure 5 for example syntax using the product described in the case study provided in Section 6 of this report.) Each frame is a meta-component (as shown in Figure 5) that can be specialized by combining the meta-component with a product specification (as shown in Figure 6) and a document type definition (DTD). The output is the specialized component, as shown in Figure 7. While this is a simple example, it shows the basic process.

For this technology, the product specification is an XML file that assigns specific values to a set of variables. The transformation process is the XVCL processor applying a DTD to the appropriate templates and the specification file. This technique automates the generation of components but does not directly automate the production of the final executable.

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///d:\XVCL\dtd\xvcl.dtd">
<!--
   Name:     Game.XVCL
   Title:    The Frame for Java Brickles
   Version:  01.10.02
   Category: Frame
   Project:  XVCL Example

   Copyright (C) 2005, AGM. All rights reserved.
-->

<x-frame name="Game">
package <value-of expr="?@TITLEOFGAME?"/>;

import javax.microedition.lcdui.CommandListener;

import coreAssets.Menu;

public class <value-of expr="?@TITLEOFGAME?"/> extends
Menu implements CommandListener {
    public <value-of expr="?@TITLEOFGAME?"/>() {
        super();
        this.board = new <value-of
expr="?@TITLEOFGAME?"/>Board(this.display);
        board.setApp(this);
    }
}

</x-frame>
```

Figure 5: XVCL Product Template

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///d:\xvcl\dtd\xvcl.dtd">

<x-frame name="Brickles" outfile="Brickles.java" language="java">

      <set var="TITLEOFGAME" value="Brickles"/>
      <adapt x-frame="Game.xvcl">

      </adapt>

</x-frame>
```

Figure 6: Example Product Specification for Brickles Game

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "file:///d:\xvcl\dtd\xvcl.dtd">

<x-frame name="Brickles" outfile="Brickles.java" language="java">

      <set var="TITLEOFGAME" value="Brickles"/>
      <adapt x-frame="Game.xvcl">

      </adapt>

</x-frame>
```

Figure 7: Resulting Java Code

4.3 Directing Core Asset Development

The choices made for the production method guide the core asset developers in the decisions of which assets to develop and how to develop them. The primary assets, such as the product line requirements and the software architecture, are used to derive the production assets such as specification tools and generators. In effect, the product generator is a generation technology specialized to the software architecture [Glück 96]. The product generator can generate only assemblies that are compatible with the architecture. Complex dependencies among architectural elements are encoded in the generator and automatically resolved. Doing this reduces the cost of testing the assemblies by eliminating certain types of defects, but at the cost of validating the generator.

5 Operating the Automated Production Capability

Automated product derivation techniques must support the production uses illustrated in the use case diagram shown in Figure 3. In this section, we examine each action in that figure. These actions are the responsibility of the production capability, but they are enacted as needed rather than as a fixed sequence.

5.1 Establish Infrastructure

Automatic product derivation works because it rests on an infrastructure that is created with the exact nature of the derivation in mind. That nature encompasses the scope of the product line and the production goals that the infrastructure must achieve. This infrastructure must support the two basic activities of product specification and specification transformation. Usually the infrastructure consists of models of the domain, tools that allow products to be specified using those models, and tools that map a specification onto libraries of templates or implementations.

The infrastructure can be developed in phases like any other set of core assets. However, even in a reactive approach to product line adoption, there is still an initial investment to establish the required infrastructure. Much of this investment, such as the cost of creating a domain model, is part of the cost of a product line organization regardless of the product derivation approach being used. Other costs, such as using the domain model to create a DSL, are not part of the usual cost of a product line and represent an additional investment.

The domain model is the fundamental element in the infrastructure. The model is a result of exercising the “Understanding Relevant Domains” practice area. Two different strategies are possible for developing the domain model incrementally. One is to identify some subset of the architecture and to drive it to the appropriate, and completed, level of detail. Another is to develop a model that covers the breadth of the product line’s scope but does not provide much detail. The detailed, partial model provides support for automatic derivation for that portion of the product covered by the model.

5.2 Specify Product

A product developer uses the automated production capability to specify the product to be produced. The specification can take on many different forms depending on the automation technology being used. The specification language is usually some form of a DSL that raises the level of specification above a basic programming language. MetaEdit+, for example, provides the ability to represent domain concepts as textual language primitives or as graphical icons.

The foremost issue is whether the infrastructure has the ability to specify the product completely. The Unified Modeling Language (UML) in its second full version has taken a major step in that direction. The Object Constraint Language (OCL) is a more complete language for expressing constraints than earlier versions. However, UML and other similar languages still must be supplemented to produce a complete specification.

A second issue is whether the technology matches the skill set of the product developers. The DSLs created using MetaEdit+ are more intuitive than UML for an expert in a domain other than software design modeling. The tradeoff is the amount of resources needed to create the DSL versus the extensive number of UML tools.

The result of this activity is a machine-parseable description of a product that is within the scope of the product line. A product specified using a DSL is guaranteed to be within the product line's scope because the language can not express concepts outside the scope. Other specification languages may not enforce this limit, and those specifications must be manually checked for conformance.

5.3 Configure Product

Some automatically derived products will require additional configuration after production is complete. This configuration is intended to add information specific to the environment in which the product is deployed or to address the user's preferences. The product must contain the mechanism that supports the configuration functions.

Configuration may be automated as when a plug-and-play capability is provided. For example, Eclipse is configured as it starts execution. The plug-ins directory is traversed, and each plugin.xml is used to activate a piece of Eclipse. In this case, the product has a mechanism that reads the XML files and uses the configuration information in each file to compose menus, the help system, and other product parts.

Configuration may be manual as when the user must edit a configuration file or use menu entries in the product to select options and set values. The product still includes functionality to read the file and use the values in the file. A number of programming tools use this approach.

The configuration mechanism should be selected to minimize errors and to be compatible with the skill level of the product's users. In addition, the mechanism should maintain sufficient constraint information to prevent inconsistent settings.

5.4 Extend Product

The production system may be partially automated, and many products in the product line may require unique extensions to satisfy the complete specification. The integration of the automated and manual portions of the product should occur along an architectural boundary. The extension will fit one of two scenarios:

1. The extension is a new variant value at an existing variation point. The extension adds features that are compatible with, although different from, the existing features in the architecture. The required information is provided to the infrastructure to generate the appropriate components.
2. The extension introduces a new variation point. This scenario is an unanticipated evolution of the product line [McGregor 03]. Either a modification to an existing interface or the addition of a glue-code adapter between the existing interface and the required feature is necessary.

Usually when products or features are added to the product line's scope, the infrastructure will eventually be extended. The effective way to do this is to follow the same process by which the infrastructure was built initially. The domain models are modified to accommodate the new features. These models are used to regenerate the DSL and related tools.

6 Case Study

This case study is based on the SEI's pedagogical product line. The fictional company Arcade Game Maker[1] (AGM) has a product line of three different games available in three major variations, each with a number of minor variations (see Figure 8). The games are Brickles, Pong, and Bowling. The three major variants are (1) a simple PC-based game, (2) a wireless device version, and (3) a convention giveaway version that can be customized to the company giving it away. Further, a number of platforms are supported either through different operating systems or, in the case of the wireless version, several different processors.

The AGM product line is divided into three increments of three products, each addressing specific markets. Each increment consists of one variant of all three games, such as the wireless device implementations of Brickles, Pong, and Bowling. Since a grouping of three products is often about the break-even point between product line and individual product development, this division was useful for decision making.

The organization conducted a product line analysis to develop some of the information needed to build the production strategy. The company also constructed an economic model that allowed it to analyze the costs of automating production. The model showed that the cost of automation was high in the initial increments, but, by the third increment, the team's understanding of the domain would reduce the cost of automation to the point that automation would be the most advantageous approach.

The organization developed the following production strategy: *We will produce the initial products using a traditional iterative, incremental development process using a standard programming language, Integrated Device Electronics (IDE), and available libraries. We will create domain-based assets, including a product line architecture and software components, for the initial products in a manner that will support a migration to automatic generation of products in the second and third increments.*

[1] The AGM product line is currently available at www.cs.clemson.edu/~johnmc /productLines/example/frontPage.htm, but it will eventually appear on the SEI's Web site.

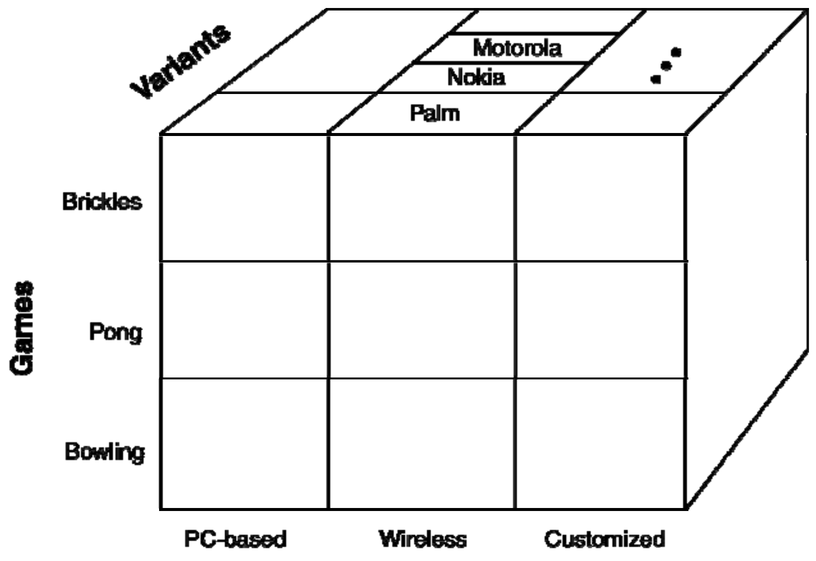
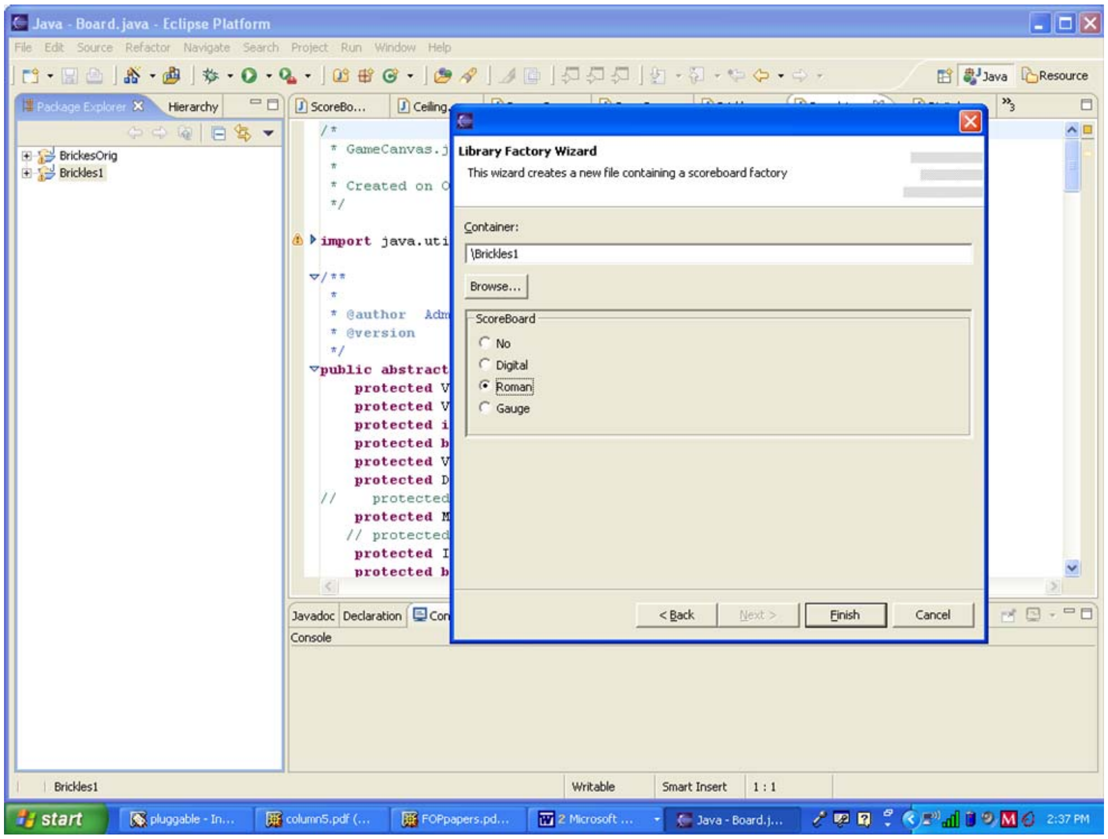


Figure 8: The AGM Product Line

As part of developing the production method specification from the production strategy, AGM identified *low cost* and *simple* as two qualities that were immediately required of the production method, with *automatic* to be added as the product line matured. The product line organization used the strategy as input to its production-planning activity. The organization used goal-driven method-engineering techniques to develop the production method from the production strategy and then developed a production plan from that method.

The AGM product line organization could use any of several automation techniques. The company's years of experience in producing game products have created many domain experts on staff, but most of them also have development skills so they can use a wide range of technologies. An approach in which a large code base is delivered and then configured is not a good choice because the wireless increment requires a small memory footprint for the products. Generation of the executable seems to be a reasonable approach because the software architecture binds all variation points during product definition.

Initially, AGM experimented with narrowly focused wizards as the means to generate portions of the products for the wireless increment based on feature selections by the product builder. Figure 9 shows one such wizard. Here the product builder can select one of several styles of scoreboards or choose to have no scoreboard at all. This technique was taking too long and was not used for the majority of the product.



Library Factory Wizard

This wizard creates a new file containing a scoreboard factory

Container: \Brickles1

ScoreBoard: No / Digital / [✓ Roman] / Gauge

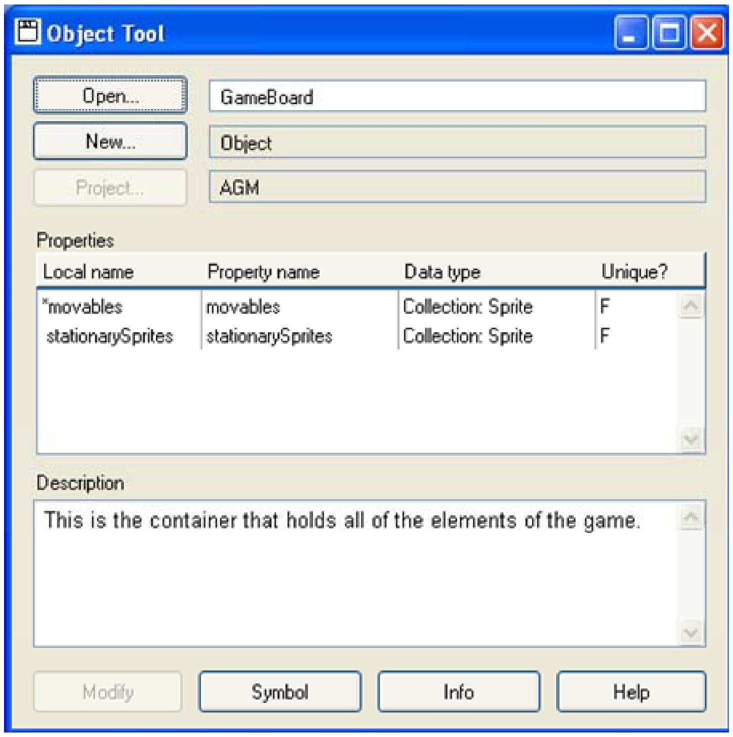
```
/*
 * GameCanvas.j
 *
 * Created on C
 */

import java.uti

/**
 *
 * @author  Adm
 * @version
 */
public abstract
    protected V
    protected V
    protected i
    protected b
    protected V
    protected D
//    protected
    protected M
  // protected
    protected I
    protected b
```

Figure 9:   Factory Wizard [2]

The team developing the production method for the customized convention giveaways that were part of Increment 3 decided to explore metamodeling, and they eventually chose it as their approach to automation. They used MetaEdit+ as the modeling tool [Tolvanen 04]. In this approach, a metamodel is developed that essentially builds a modeling tool that implements a DSL. This tool supports the development of products within the domain described in the metamodel.

The AGM domain experts developed a metamodel using their company's terminology and the basic architecture of the products. Figure 10 shows the wizard used to define a new object type. In this case, it is the GameBoard, which contains two collections: one for MovableSprites and one for StationarySprites.

[2] Thanks to John Hunt of Clemson University for the implementation and figure.



GameBoard

Object

AGM

Properties

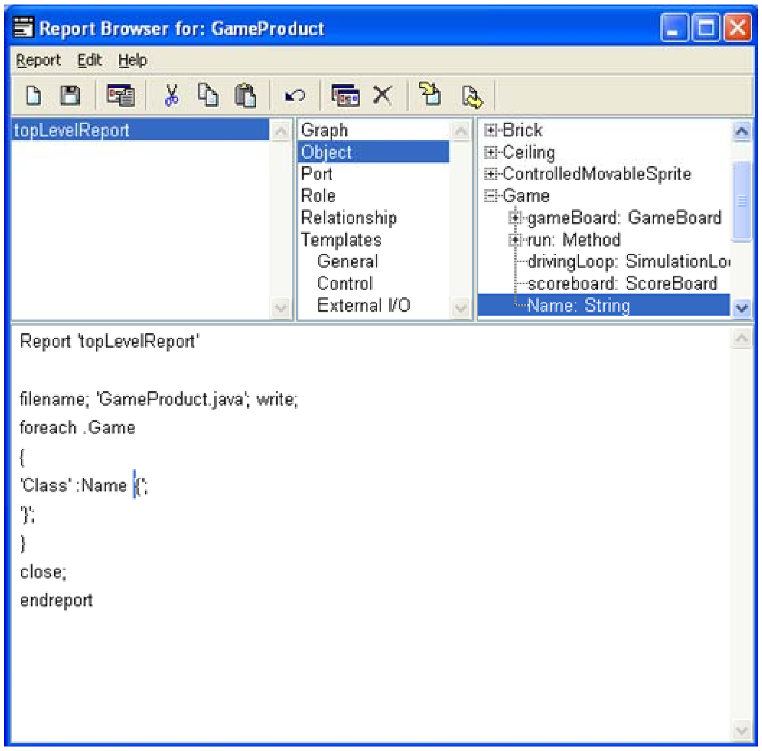
Local name	Property name	Data type	Unique?
*movables	movables	Collection: Sprite	F
stationarySprites	stationarySprites	Collection: Sprite	F

Description

This is the container that holds all of the elements of the game.

Figure 10: Object Tool

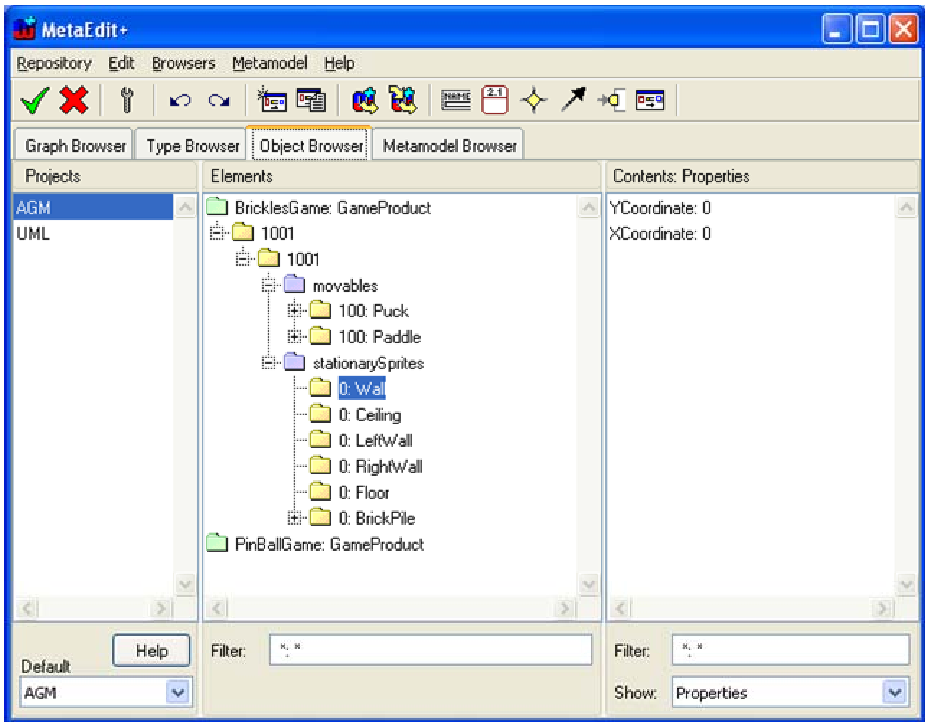
The asset developers associate each domain concept with a specific implementation. This association is made using the report browser shown in Figure 11. A series of reports are defined, and a product will be produced by a top-level report that calls the other reports that are required to produce the code needed for the product.



```
Report 'topLevelReport'

filename; 'GameProduct.java'; write;
foreach .Game
{
'Class' :Name '{';
'}';
}
close;
endreport
```

Figure 11: Report Browser

The product builders use the dialog box shown in Figure 12 to define a product. The product builder defines a new top-level graph that links together instances of object types. These links enable interactions between the associated objects. The report associated with this graph will invoke the other reports associated with the members of the graph.



AGM
UML

BricklesGame: GameProduct
1001
1001
movables
100: Puck
100: Paddle
stationarySprites
0: Wall
0: Ceiling
0: LeftWall
0: RightWall
0: Floor
0: BrickPile
PinBallGame: GameProduct

YCoordinate: 0
XCoordinate: 0

Default: AGM
Filter: *; *
Filter: *; *
Show: Properties

Figure 12: Defining a Game Product

7 Conclusions

The automatic derivation of products from a set of core assets requires more infrastructure than most manual approaches. A product line organization can still implement this capability in either the proactive (i.e., create all assets before any products) or reactive (i.e., create assets as needed for a product) modes. The percentage of the product derivation process that is automated may be 100 percent, or initially only a portion of each product may be built automatically with that portion increasing slowly over time.

A specification language and a transformation infrastructure are needed for automatic derivation. There is a tradeoff between the expressiveness of the specification language and the cost of developing the transformation infrastructure. The more fine grained and comprehensive the specification language is, the more complex and larger the infrastructure must be to implement the specifications.

Automated product derivation is attractive when time to market must be drastically reduced, but it is a viable goal only for a product line organization that is mature in the domain or that implements a standard architecture. The scope of the domain and its stability directly affect the costs of automated derivation. A sufficiently large number of products relative to the size of each product are needed to produce a positive return on investment.

A major challenge in automatic derivation is the identification of and reasoning about dependencies between variation points. While this is a problem for product lines in general, automated techniques are particularly sensitive to this problem because there may be no human oversight of what components are assembled. To avoid the inclusion of incompatible components in the same assembly, automation technology provides a means of specifying dependencies and constraints between elements.

A second challenge is anticipating the behavior that emerges when two or more components are assembled. This problem is widely recognized in component-based development, but it is true regardless of the paradigm being used. For example, this problem arises when interface methods are not properly implemented for multi-threading or when timing issues among multiple threads are not properly resolved [Wallnau 03].

This challenge can be mitigated in two ways. First, extensive integration test suites can be created to test possible integrations among components. The combinatorial explosion of component compositions makes creating such suites an arduous task. The second approach is to provide thorough specifications for components that are derived from a single software architecture.

Automatic derivation is a viable approach for a software product line organization because the scope of a product line amortizes the cost of the derivation infrastructure. Many of the practices required for product line success are also required for successful automatic derivation. In particular, the numerous planning activities in a product line organization ensure that the production capability is planned early and thoroughly. The result is an automated production capability that greatly reduces the time to market for those products that are within the scope of the product line.

References

URLs are valid as of the publication date of this document.

[Apache 04]	The Apache Software Foundation. *Axis CPP - Ant Build Guide.* http://ws.apache.org/axis/cpp/antbuild-guide.html (2004).
[Batory 00]	Batory, Don; Johnson, Clay; MacDonald, Bob; & von Heeder, Dale. “Achieving Extensibility Through Product Lines and Domain-Specific Languages: A Case Study,” 117-136. *Proceedings of the Sixth International Conference on Software Reusability*. Vienna, Austria, June 27-29, 2000. New York, NY: Springer-Verlag, 2000.
[Chastek 01]	Chastek, Gary; Donohoe, Patrick; Kang, Kyo Chul; & Thiel, Steffen. *Product Line Analysis: A Practical Introduction* (CMU/SEI-2001-TR-001, ADA396137). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents /01.reports/01tr001.html.
[Chastek 02a]	Chastek, Gary & McGregor, John D. *Guidelines for Developing a Product Line Production Plan* (CMU/SEI-2002-TR-006, ADA407772). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu /publications/documents/02.reports/02tr006.html.
[Chastek 02b]	Chastek, Gary; Donohoe, Patrick; & McGregor, John D. *Product Line Production Planning for the Home Integration System Example* (CMU/SEI-2002-TN-029, ADA405846). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports /02tn029.html.
[Chastek 04]	Chastek, Gary; Donohoe, Patrick; & McGregor, John D. *A Study of Product Production in Software Product Lines* (CMU/SEI-2004-TN-012). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications /documents/04.reports/04tn012.html.

[Clements 05]	Clements, Paul C.; McGregor, John D.; & Cohen, Sholom G. *The Structured Intuitive Model of Product Line Economics* (CMU/SEI-2005-TR-003). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu /publications/documents/05.reports/05tr003.html.
[Diaz 05]	Diaz, Oscar; Trujillo, Salvador; & Anfurrutia, Felipe I. "Supporting Production Strategies as Refinements of the Production Process," 210-221. *Proceeding of Software Product Lines: Ninth International Conference*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005.
[Glück 96]	Glück, Robert & Jones, Neil D. "Automatic Program Specialization by Partial Evaluation: An Introduction," 70-77. *Software Engineering in Scientific Computing*. Edited by W. Mackens & S. M. Rump. Braunschweig, Germany: Vieweg, 1996.
[Greenfield 04]	Greenfield, Jack; Short, Keith; Cook, Steve; Kent, Stuart; & Crupi, John. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, IN: Wiley, 2004.
[McGregor 03]	McGregor, John D. *The Evolution of Product Line Assets* (CMU/SEI-2003-TR-005, ADA418409). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports /03tr005.html.
[Tolvanen 04]	Tolvanen, Juha-Pekka. "Making Model-Based Code Generation Work." *Embedded Systems Europe 8*, 60 (August/September 2004): 36-38.
[Wallnau 03]	Wallnau, Kurt. *Volume III: A Technology for Predictable Assembly from Certifiable Components* (CMU/SEI-2003-TR-009, ADA413574). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu /publications/documents/03.reports/03tr009.html.
[Zhang 05]	Zhang, Weishan & Jarzabek, Stan. "Reuse Without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices," 57-69. *Proceedings of Software Product Lines: Ninth International Conference*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005.

REPORT DOCUMENTATION PAGE				*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE September 2005		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Preparing for Automated Derivation of Products in a Software Product Line				5. FUNDING NUMBERS FA8721-05-C-0003
6. AUTHOR(S) John D. McGregor				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213				8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2005-TR-017
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116				10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2005-017
11. SUPPLEMENTARY NOTES				
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS				12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) Organizations that adopt a software product line strategy often have business goals that concern improving their ability to produce products by lowering product development costs, by reducing the time to bring a product to market, or through other production improvements. Business goals such as these make automated product derivation an appealing strategy to a software product line organization. Automating production requires up-front investment, including the creation of both the core assets that will be assembled as products and the core assets that will perform the assembly. A software product line provides the ability to amortize the cost of the infrastructure over a set of products. This report views the process for automating the production of products in the context of a product production system. The process begins with the decision to automate, proceeds to the selection of the automation approach, and continues with the operation and management of the automated production capability. The process is illustrated by a case study automating the production process in the Carnegie Mellon® Software Engineering Institute's pedagogical product line.				
14. SUBJECT TERMS automated production, pedagogical product line, software product line				15. NUMBER OF PAGES 44
16. PRICE CODE				
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified		19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102