Testing a Software Product Line

John D. McGregor

Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu

Abstract. The software product line approach to the development of software intensive systems has been used by organizations to improve quality, increase productivity, and reduce cycle time. These gains require different approaches to a number of the practices in the development organization including testing. The planned variability that facilitates some of the benefits of the product line approach poses a challenge for test-related activities. This chapter provides a comprehensive view of testing at various points in the software development process and describes specific techniques for carrying out the various test-related tasks. These techniques are illustrated using a pedagogical product line developed by the Software Engineering Institute (SEI).

1 Introduction

Organizations are making the strategic decision to adopt a product line approach to the production of software-intensive systems. This decision is often in response to initiatives within the organization to achieve competitive advantage within their markets. The product line strategy has proven successful at helping organizations achieve aggressive goals for increasing quality and productivity and reducing cycle time. The strategy is successful, at least in part, due to its comprehensive framework that touches all aspects of product development.

Testing plays an important role in this strategic effort. In order to achieve overall goals of increased productivity and reduced cycle time, there need to be improvements to traditional testing activities. These improvements include:

- Closer cooperation between development and test personnel.
- Increased throughput in the testing process.
- Reduced resource consumption.
- Additional types of testing that address product line specific faults.

There are several challenges for testing in an organization that realizes seventy-five to ninety-five percent of each product from reusable assets. These include:

- variability - The breadth of the variability that must be accommodated in the assets, including test assets, directly impacts the resources needed for adequate testing.

– emergent behavior - As assets are selected from inventory and combined in ways not anticipated, the result can be an interaction that is a behavior not present in any one of the assets being combined. This makes it difficult to have a reusable test case that covers the interaction.
– creation of reusable assets - Test cases and test data are obvious candidates to be reused and when used as is they are easy to manage. The amount of reuse achieved in a project can be greatly increased by decomposing the test assets into finer grained pieces that are combined in a variety of ways to produce many different assets. The price of this increased reuse is increased effort for planning the creation of the assets and management of the increased number of artifacts.
– management of reusable assets - Reuse requires traceability among all of the pieces related to an asset to understand what an asset is, where it is stored, and when it is appropriate for use. A configuration management system provides the traceability by explicit artifacts.

I will discuss several activities that contribute to the quality of the software products that comprise the product line. I will think of these activities as forming a *chain of quality* in which quality assuring activities are applied in concert with each production step in the software development process. In addition to discussing the changes in traditional testing processes needed to accommodate the product line approach, I will present a modified inspection process that greatly increases the defect finding power of traditional inspection processes. This approach to inspection applies testing techniques to the conduct of an inspection.

I will use a continuing example throughout the chapter to illustrate the topics and then I will summarize the example near the end of the chapter. The Arcade Game Maker product line is an example product line developed for pedagogical purposes[1]. A complete set of product line assets are available for this example product line. The product line consists of three games: Brickles, Pong, and Bowling. The variation points in the product line include the operating system on which the games run, a choice of an analog, digital, or no scoreboard, and whether the product has a practice mode.

The software product line strategy is a business strategy that uses a specific method to achieve its goals. The material in this chapter will reflect this orientation by combining technical and managerial issues. I will briefly introduce a comprehensive approach to software product line development and I will provide a state-of-the-practice summary. Then I will describe the current issues, detail some experiences, and outline research questions regarding the test-related activities in a software product line organization.

2 Software Product Lines

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

[1] The complete example is available at http://www.sei.cmu.edu/productlines/ppl

segment or mission and that are developed from a common set of core assets in a prescribed manner[8]. This definition has a number of implications for test strategies. Consider these key phrases from the definition:

- *set of software-intensive systems* - The product line is the set of products. The product line organization is the set of people, business processes, and other resources used to build the product line. The commonalities among the products will translate into opportunities for reuse in the test artifacts. The variabilities among products will determine how much testing will be needed.
- *common, managed set of features* - Test artifacts should be tied to significant reusable chunks of the products, such as features. These artifacts are managed as assets in parallel to the production assets to which they correspond. This will reduce the effort needed to trace assets for reuse and maintenance purposes. A test asset is used whenever the production asset to which it is associated is used.
- *specific needs of a particular market segment or mission* - There is a specified domain of interest. The culture of this domain will influence the priorities of product qualities and ultimately the levels of test coverage. For example, a medical device that integrates hardware and software requires far more evidence of the absence of defects than the latest video game. Over time those who work in the medical device industry develop a different view of testing and other quality assurance activities from workers in the video game domain.
- *common set of core assets* - The test core assets include test plans, test infrastructures, test cases, and test data. These assets are developed to accommodate the range of variability in the product line. For example, a test suite constructed for an abstract class is a core asset that is used to quickly create tests for concrete classes derived from the abstract class.
- *in a prescribed manner* - There is a production strategy and production method that define how products are built. The test strategy and test infrastructure must be compatible with the production strategy. A production strategy that calls for dynamic binding imposes similar constraints on the testing technology.

The software product line approach to development affects how many development tasks are carried out. Adopting the product line strategy has implications for the software engineering activities as well as the technical and organizational management activities. The Software Engineering Institute has developed a Framework for Software Product Line Practice^{SM}[2] which defines 29 practice areas that affect the success of the product line. A list of these practices is included in the Appendix. The practices are grouped into Software Engineering, Technical Management, and Organizational Management categories. These categories reflect the dual technical and business perspectives of a product line

[2] ^{SM} service mark of Carnegie Mellon University

organization. A very brief description of the relationship between the Testing practice area and the other 28 practices is included in the list in the appendix.

The software product line approach seeks to achieve strategic levels of reuse. Organizations have been able to achieve these strategic levels using the product line approach, while other approaches have failed, because of the comprehensive nature of the product line approach. For example, consider the problem of creating a "reusable" implementation of a component. The developer is left with no design context, just the general notion of the behavior the component should have and the manager is not certain how to pay for the 50% - 100% additional cost of making the component reusable instead of purpose built. In a software product line, the qualities and properties, required for a product to be a member, provide the context. The reusable component only has to work within that context. The manager knows that the product line organization that owns all of the products is ultimately responsible for funding the development. This chapter will present testing in the context of such a comprehensive approach.

2.1 Commonality and Variability

The products in a product line are very similar but differ from each other (otherwise they would be the same product). The points at which the products differ are referred to as *variation points*. Each possible implementation of a variation point is referred to as a *variant*.

The set of products possible by taking various combinations of the variants defines the *scope* of the product line. Determining the appropriate scope of the product line is critical to the success of the product line in general and the testing practice specifically. The scope constrains the possible variations that must be accounted for in testing. Too broad a scope will waste testing resources if some of the products are never actually produced. Too vague a scope will make test requirements either vague or impossible to write.

Variation in a product is nothing new. Control structures allow several execution paths to be specified but only one to be taken at a time. By changing the original inputs, a different execution path is selected from the *existing* paths. The product line approach adds a new dimension. From one product to another, the paths that are *possible* change. In the first type of variation, the path taken during a specific execution changes from one execution to the next but the control flow graph does not change just because different inputs are chosen. In the second type of variation different control flow graphs are created when the a different variant is chosen.

This adds the need for an extra step in the testing process, sampling across the product space, in addition to sampling across the data space. This is manageable only because of the use of explicit, pre-determined variation points and automation. Simply taking an asset and modifying it in any way necessary to fit a product will quickly introduce the level of chaos that has caused most reuse efforts to fail.

The *commonality* among the products in a product line represents a very large portion of the functionality of these products and the largest opportu-

nity to reduce the resources required. In some product lines this commonality is delivered in the form of a platform which may provide virtually identical functionality to every product in the product line. Then a select set of features are added to the platform to define each product. In other cases, each product is a unique assembly of assets, some of which are shared with other products. These different approaches will affect our choice of test strategy.

The products that comprise a product line have much in common beyond the functional features provided to users. They typically attack similar problems, require similar technologies, and are used by similar markets. The product line approach facilitates the exploitation of the identified commonality even to the level of service manuals and training materials[9].

The commonality/variability analysis of the product line method produces a model that identifies the variation points needed in the product line architecture to support the range of requirements for the products in the product line. Failure to recognize the need for variation in an asset will require custom development and management of a separate branch of code that must be maintained until a future major release. Variability has several implications for testing:

- *Variation is identified at explicit, documented variation points* - Each of these points will impose a test obligation in terms either of selecting a test configuration or test data. Analysis is required at each point to fully understand the range of variation possible and the implications for testing.
- *Variation among products means variation among tests* - The test software will typically have at least the same variation points as the product software. Constraints are needed to associate test variants with product variants. One solution to this is to have automated build scripts that build both the product and the tests at the same time.
- *Differences in behavior between the variants and the tests should be minimal* - I will show later that using the same mechanisms to design the test infrastructure as are used to design the product software is usually an effective technique. Assuming the product mechanisms are chosen to achieve certain qualities, the test software should seek to achieve the same qualities.
- *The specific variant to be used at a variation point is bound at a specific time.* - The binding time for a variation point is one specific attribute that should be carefully matched to the binding of tests. Binding the tests later than the product assets are bound is usually acceptable but not the reverse.
- *The test infrastructure must support all the binding times used in the product line.* - Dynamically bound variants are of particular concern. For example, binding aspects to the product code as it executes in the virtual machine may require special techniques to instrument test cases.
- *Managing the range of variation in a product line is essential* - Every variant possibility added to a variation point potentially has a combinatorial impact on the test effort. Candidate variants should be analyzed carefully to determine the value added by that variant. Test techniques that reduce the impact of added variants will be sought as well.

Commonality also has implications for testing:

– Techniques that avoid retesting of the common portions can greatly reduce the test effort.
– The common behavior is reused in several products making an investment in test automation viable.

2.2 Planning and structuring

Planning and structuring are two activities that are important to the success of the product line. A software product line is chartered with a specific set of goals. Test planning takes these high level goals into account when defining the goals for the testing activities. The test assets will be structured to enhance their reusability. Techniques such as inheritance hierarchies, aspect-oriented programming, and template programming provide a basis for defining assets that possess specific attributes including reusability.

Planning and structuring must be carried out incrementally. To optimize reuse, assets must be decomposed and structured to facilitate assembly by a product team. One goal is to have *incremental* algorithms that can be completed on individual modules and then be more rapidly completed for assembled subsystems and products using the partial results. Work in areas such as incremental model checking and component certification may provide techniques for incremental testing[32].

A product line organization defines two types of roles: *core asset builder* and *product builder*. The core asset builders create assets that span a sufficient range of variability to be usable across a number of products. The core assets include early assets such as the business case, requirements, and architecture and later assets such as the code. Test assets include plans, frameworks, and code.

The core asset builder creates an *attached process* for each core asset. The attached process describes how to use the core asset in building a product. The attached process may be a written description, provided as a cheatsheet in Eclipse or as a guidance element in the .NET environment, or it may be a script that will drive an automated tool. The attached process adds value by reducing the time required for a product builder to learn how to use the core asset.

The core asset builder's perspective is: create assets that are usable by product builders. A core asset builder's primary trade-off is between sufficient variability to maximize the reuse potential of an asset and sufficient commonality to provide substantial value in product building. The core asset developer is typically responsible for creating all parts of an asset. This may include test cases that are used to test the asset during development and can be used by product developers to sanity test the assets once they are integrated into a product.

The product builders construct products using core assets and any additional assets they must create for the unique portion of product. The product builder provides feedback to the core asset builders about the usefulness of the core assets. This feedback includes whether the variation points were sufficient for the product.

The product builder's perspective is: achieve the required qualities for their specific product as rapidly as possible. The product builder's primary trade-off is

between maximizing the use of core assets in building the product and achieving the precise requirements of their specific product. In particular, product builders may need a way to select test data to focus on those ranges critical to the success of their product.

To coordinate the work of the two groups, a production strategy is created that provides a strategic overview of how products will be created from the core assets. A production method is defined to implement the strategy. The method details how the core asset developers should build and test the core assets so that product builders can achieve their objectives.

A more in-depth treatment of software product lines can be found in [8].

3 Testing Overview

Testing is *the detailed examination of an artifact guided by specific information.* The examination is a search for defects. Program failures signal that the search has been successful. How hard we search depends on the consequences if a defect is released in a product. Software for life support systems will require a more thorough search than word processing software.

The purpose of this section is to give a particular perspective on testing at a high level before discussing the specifics of testing in a software product line. I will discuss some of the artifacts needed to operate a test process. After that I will present a perspective on the testing role and then briefly describe fault models.

3.1 Testing Artifacts

This definition of testing encompasses a number of testing activities that are dispersed along the software development life cycle. I will refer to the places where these activities are located as *test points.* Figure 1 shows the set of test points for a high-level view of a development process. This sequence is intended to establish the chain of quality.

The IEEE 829 standard defines a number of testing artifacts, which will be used at each of the test points in the development process [19]. Several of these test assets are modified from their traditional form to support testing in a product line:

- test plan - A description of what testing will be done, the resources needed, and a schedule for when activities will occur. Any software development project should have a high level end-to-end plan that coordinates the various specific types of tests that are applied by developers and system testers. Then individual test plans are constructed for each test that will be conducted. In a software product line organization a distinction is made between those plans developed by core asset builders that will be delivered to the product builders of every product and the plans that the product builder derives from the product line plan and uses to develop their product specific plan.

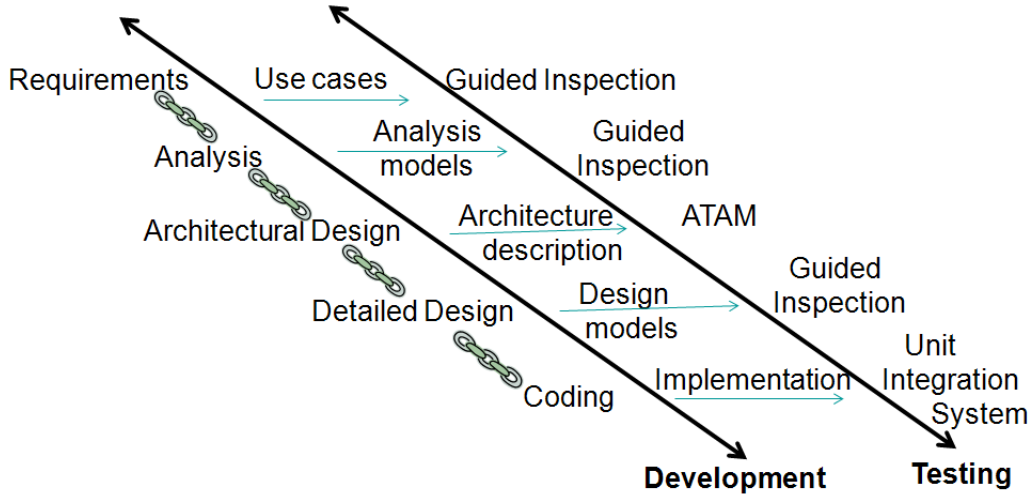


Fig. 1. Test points

The core asset builders might provide a template document or a tool that collects the needed data and then generates the required plan.

- test case - A single configuration of test elements that embodies a use scenario for the artifact under test. In a software product line, a test case may have variation points that allow it to be configured for use with multiple products.
- test data - All of the data needed to fully describe a scenario. The data is linked to specific test cases so that it can be reused when the test case is. The test data may have variation points that allow some portion of the data to be included or excluded for a particular use.
- test report - A summary of the information resulting from the test process. The report is used to communicate to the original developers, management, and potentially to customers. The test report may also be used as evidence of the quantity and quality of testing in the event of litigation related to a product failure.

The primary task in testing is defining effective test cases. To a tester, that means finding a set of stimuli that, when applied to the artifact under test, exposes a defect. I will discuss several techniques for test case creation but all of them rely on a fault model. There will be a fault model for each test point. I will focus on faults related to being in a product line in section 1.

3.2 Testing Perspective

The test activities, particularly the review of non-software assets, are often carried out by people without a traditional testing background. Unit tests are usually carried out by developers who pay more attention to creating than critiquing.

A product line organization will provide some fundamental training in testing techniques and procedures but this is no substitute for the perspective of an experienced tester. In addition to training in techniques, the people who take on a testing role at any point in any process should adopt the "testing perspective". This perspective guides how they view their assigned testing activities. Each person with some responsibility for a type of testing should consider how these qualities should affect their actions.

- Systematic - Testing is a search for defects and an effective search must be systematic about where it looks. The tester must follow a well-defined process when they are selecting test cases so that it is clear what has been tested and what has not. For example, coverage criteria are usually stated in a manner that describes the "system." All branches level of test coverage means that test cases have been created for each path out of each decision point in the control flow graph.
- Objective - The tester should not make assumptions about the work to be tested. Following specific algorithms for test case selection removes any of the tester's personal feelings about what is likely to be correct or incorrect. "Bob always does good work, I don't need to test his work as thoroughly as John's," is a sure path to failure.
- Thorough - The tests should reach some level of coverage of the work being examined that is "complete" by some definition. Essentially, for some classes of defects, tests should look everywhere those defects could be located. For example, test every error path if the system must be fault tolerant.
- Skeptical - The tester should not accept any claim of correctness until it has been verified by an acceptable technique. Testing boundary conditions every time eliminates the assumption that "it is bound to work for zero."

3.3 Fault models

A fault model is the set of known defects that can result from the development activities leading to the test point. Faults can be related to several different aspects of the development environment. For example, programs written in C and C++ are well known for null pointer errors. Object-oriented design techniques introduce the possibility of certain types of defects such as invoking the incorrect virtual method [29, 1]. Faults also are the result of the development process and even the organization. For example, interface errors are more likely to occur between modules written by teams that are non-co-located. The development organization can develop a set of fault models that reflect their unique blend of process, domain, and development environment. The organization can incorporate some existing models such as Chillaredge's Orthogonal Defect Classification into the fault models developed for the various test points [7].

Testers use fault models to design effective and efficient test cases since the test cases are specifically designed to search for defects that are likely to be present. Developers of safety critical systems construct fault trees as part of a failure analysis. A failure analysis of this type usually starts at the point of

failure and works back to find the cause. Another type of analysis is conducted as a forward search. These types of models capture specific faults. We want to capture *fault types* or categories of faults. The models are further divided so that each model corresponds to specific test points in the test process.

A product line organization develops fault models by tracking defects and classifying these defects to provide a definition of the defect and the frequency with which it occurs. Table 1 shows some possible faults per test point related to a product line strategy. Others can be identified from the fault trees produced from safety analyses conducted on product families [11, 10, 23].

Test point	Example faults
Requirements	incomplete list of variations
Analysis	missing constraints on variations
Architecture Design	contradictions between variation constraints
Detailed design	failure to propagate variations from subsystem interface
Unit testing	failure to implement expected variations
Integration testing	mismatched binding times between modules
System testing	inability to achieve required configuration

Table 1. Faults by test point

3.4 Summary

Nothing that has been said so far requires that the asset under test be program source code. The traditional test points are the units of code produced by an individual developer or team, the point at which a team's work is integrated with that of other teams, and the completely assembled system. In section 4 I present a review/inspection technique, termed Guided Inspection, that applies the testing perspective to the review of non-software assets. This technique can be applied at several of the test points shown in Figure 1. This adds a number of test points to what would usually be referred to as "testing" but it completes the chain of quality. In section 5 I will present techniques for the other test points. In an iterative, incremental development process the end of each phase may be encountered many times so the test points may be exercised many times during a development effort. Test implementations must be created in anticipation of this repetition.

4 Guided Inspection

Guided Inspection is a technique that applies the discipline of testing to the review of non-software assets. The review process is *guided* by scenarios that

are, in effect, test cases. This technique is based on Active Reviews by Parnas and Weiss [30]. An inspection technique is appropriate for a chapter about testing in a product line organization because the reviews and inspections are integral to the chain of quality and because the test professionals should play a key role in ensuring that these techniques are applied effectively.

4.1 The Process

Consider a detailed design document for the computation engine in the arcade game product line. The document contains a UML model complete with OCL constraints as well as text tying the model to portions of the use case model that defines the product line.

A Guided Inspection follows the steps shown in Figure 2, a screenshot from the Eclipse Process Framework Composer. The scenarios are selected from the use case model shown in Figure 3. For this example, I picked

> "The Player has begun playing Brickles. The puck is in play and the Player is moving the paddle to reflect the puck. No pucks have been lost and no bricks hit so far. The puck and paddle have just come together on this tick of the clock."

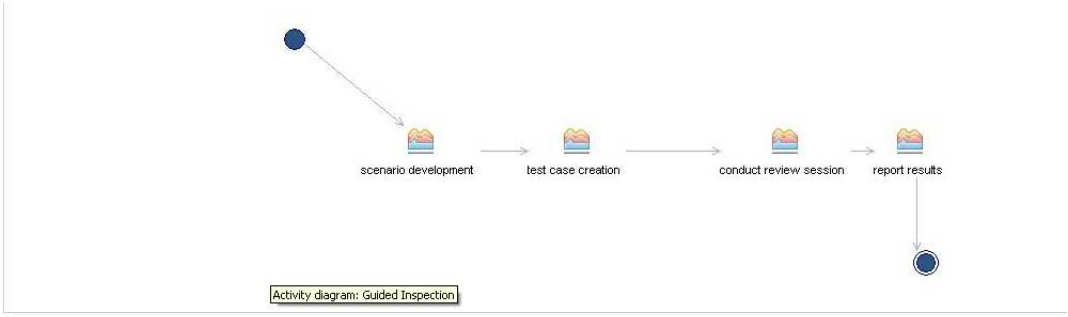
as a very simple scenario that corresponds to a test pattern discussed later.

The GameBoard shown in the class diagram in Figure 4 is reused, as is, in each game. It serves as a container for GamePieces. According to the scenario, the game is in motion so the game instance is in the moving state, one of the states shown in Figure 8. The sequence diagram shown in Figure 5 shows the action as the clock sends a tick to the gameboard which sends the tick on to the MovableSprites on the gameboard. After each tick the gameboard invokes the "check for collision" algorithm shown in Figure 6. The collision detection algorithm detects that the puck and paddle have collided and invokes the collision handling algorithm shown in Figure 7.

In the inspection session, the team reads the scenario while tracing through the diagrams to be certain that the situation described in the scenario is accurately represented in the design model. Looking for problems such as missing associations among classes and missing messages between objects. The defects found are noted and in some development organizations would be written up as problem reports.

Sufficient scenarios are created and traced to give evidence that the design model is complete, correct, and consistent. Coverage is measured by the portions of diagrams, such as specific classes in a class diagram, that are examined as part of a scenario. One possible set of coverage criteria, listed in order of increasing coverage, includes:

- a scenario for each end-to-end use case, including "extends" use cases
- a scenario that touches each "includes" use case
- a scenario that touches each variation point
- a scenario that uses each variant of each variation point



Work Breakdown

Breakdown Element	Steps	Index	Predecessors	Model Info	Type	Planned	Repeatable	Multiple Occurrences	Ongoing	Event-Driven	Op
scenario development		1			Phase	✔		✔			
create scenarios		2			Activity	✔					
Create scenarios	•	3			Task Descriptor	✔					
test case creation		4	1FF		Phase	✔	✔	✔	✔		
variation binding		5			Activity	✔	✔	✔			
instantiate scenario		6			Activity	✔		✔	✔		
conduct review session		7	4FF		Phase	✔		✔		✔	
release inspection package		8			Activity	✔					
hold inspection meeting		9			Activity	✔					
report results		11	7FF		Phase	✔					
inspection report					Work Product Descriptor						

Fig. 2. Guided Inspection description

Guided Inspection is not the only scenario based evaluation tool that can be employed. The Architecture Tradeoff Analysis Method (ATAM) developed by the SEI also uses scenarios to evaluate the architecture of a product line[2]. Their technique looks specifically at the quality attributes, i.e., non-functional requirements, the architecture is attempting to achieve.

The benefit of a Guided Inspection session does not stop with the many defects found during the inspections. The scenarios created during the architecture evaluation and detailed design will provide an evolution path for the scenarios to be used to create executable integration and system test cases.

4.2 Non-functional requirements

Software product line organizations are concerned about more than just the functional requirements for products. These organizations want to address the non-functional requirements as early as possible. Non-functional requirements, sometimes called quality attributes, include characteristics such as performance, modifiability, and dependability. Reis et al describe a technique for using scenarios to begin testing for performance early in the life of the product line[31].

Both Guided Inspection and the ATAM provide a means for investigating quality attributes. During the “create scenarios” activity, scenarios, elicited from stakeholders, describe desirable product behavior in terms of user-visible actions.

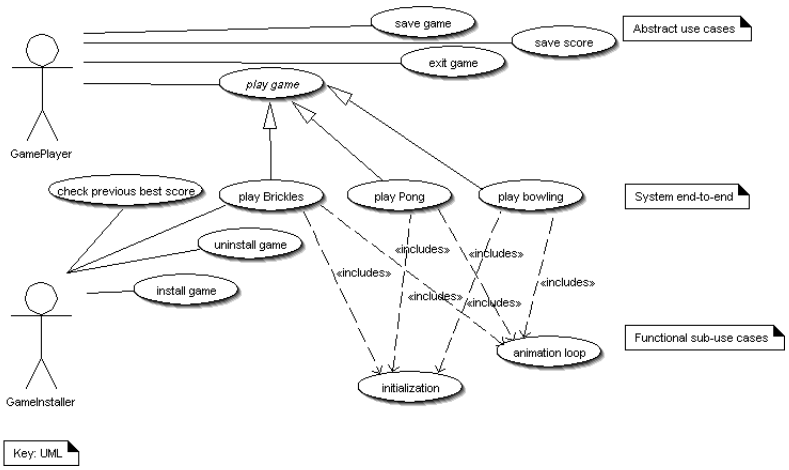


Fig. 3. Use Case diagram

The ArchE tool, developed at the SEI, aids in reasoning about these attributes [3]. It provides assistance to the architect in trading off among multiple, conflicting attributes.

5 Testing Techniques for a Product Line

The *test practice* in a product line organization encompasses all of the knowledge about testing necessary to operate the test activities in the organization. This includes the knowledge about the processes, technologies, and models needed to define the *test method*. I will first discuss testing as a practice since this provides a more comprehensive approach than just discussing all of the individual processes needed at the individual test points[20]. Then I will use the three phase view of testing developed by Hetzel[18]- planning, construction, and execution - to structure the details of the rest of the discussion.

In his keynote to the Software Product Line Testing Workshop (SPLiT), Grütter listed four challenges to product line testing[17]:

- Meeting the product developer's quality expectations for core assets.
- Establishing testing as a discipline that is well-regarded by managers and developers.
- Controlling the growth of variability.
- Making design decisions for testability.

I will incorporate a number of techniques that address these challenges.

[3] ArchE is available for download at http://www.sei.cmu.edu/architecture/arche.html

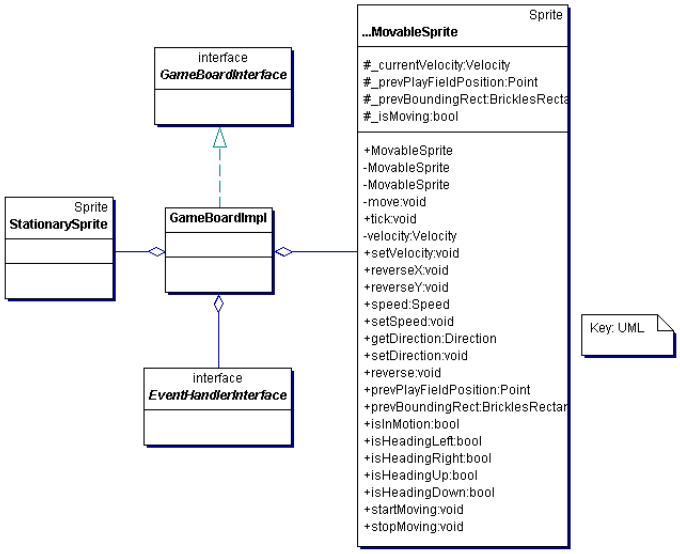


Fig. 4. Central class diagram

5.1 Test method overview

The test practice in an organization encompasses the test methods, coordinated sets of processes, tools, and models, that are used at all test points. For example, "test first development" is an integrated development and testing method that follows an agile development process model and is supported by tools such as JUnit.

The test method for a product line organization defines a number of test processes that operate independently of each other. This is true for any project but in a product line organization these processes are distributed among the core asset and product teams and must be coordinated. Often the teams are non-co-located and communicate via a variety of mechanisms.

The test method for a project must be compatible with the development method. The test method for an agile development process is very different from the test method for a certifiable software (FAA, FDA, etc.) process. The rhythm of activity must be synchronized between the test and development methods. Tasks must be clearly assigned to one method or the other. Some testing tasks such as unit testing will often be assigned to development staff. These tasks are still defined in the testing method so that expectations for defect search, including levels of coverage and fault models, can be coordinated.

There is a process defined for the conduct of testing at each of the test points. This process defines procedures for constructing test cases for the artifacts that are under test at that test point. The test method defines a comprehensive fault

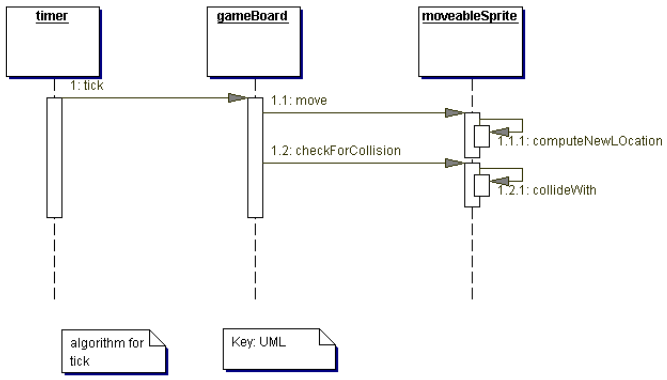


Fig. 5. Basic running algorithm

model and assigns responsibility for specific defect types to each test point. The product line test plan should assign responsibility for operating each of these test processes.

An example model of the testing practice for a software product line organization is provided at http://www.cs.clemson.edu/∼johnmc/PublishTest/index.htm. The model conforms to the SEPM and was developed using EPF. The software process engineering meta-model (SPEM)[16] provides a basis for modeling methods. SEPM is a standard from the Object Modeling Group (OMG). The Eclipse Process Framework (EPF) Composer provides an implementation of this meta-model in the form of a IDE for method modeling.

5.2 Test Planning

Adopting the software product line approach is a strategic decision. The product line organization develops strategies for producing products and testing them. The strategies are coordinated to ensure an efficient, effective operation. One of the challenges to testing in a product line organization is achieving the same gains in productivity as product production so that testing does not prevent the organization from realizing the desired improvements in throughput. Table 2 shows frequently desired production and product qualities and corresponding testing actions.

The faster time to market is achieved by eliminating the human tester in the loop as much as possible. Short iterations from test to debug/fix will keep cycle time shorter. Production is cheaper and faster when the human is out of the loop. The product line strategy also supports having a cheaper, more domain-oriented set of product builders. These product builders could be less technical and paid less if the core assets are designed to support a more automated style

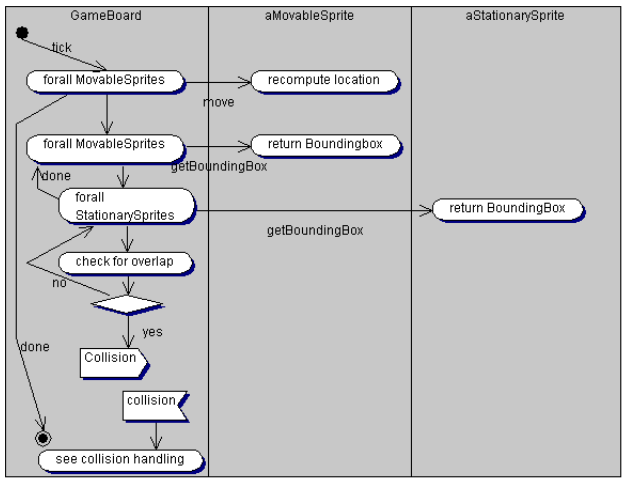


Fig. 6. Collision detection algorithm

To achieve	use
faster	automation and iteration
cheaper	automation and differentiated work force
better	more thorough coverage
mass customization	combinatorial testing

Table 2. Product qualities and testing actions

of product building and testing. The quality of the products can be made better because more thorough coverage can be achieved over time. Finally, the product line core asset base supports mass customization but requires combinatorial test techniques to cover the range of variability without having the test space explode.

The production planning activity begins with the business goals of the product line and produces a production plan that describes the production strategy and the production method[6]. The testing activities in the product line organization are coordinated with the product development activities by addressing both production and testing activities during production planning. The plan talks about how core assets will be created to support strategic reuse and how products will be assembled to meet the business goals of the organization.

The production method is a realization of the production strategy and defines how products are created from the core assets. The core asset development method is structured to produce core assets that facilitate product building. The production method specifies the variation mechanisms to be used to implement the variation points. These mechanisms will determine some of the variation mechanisms used in the test software.

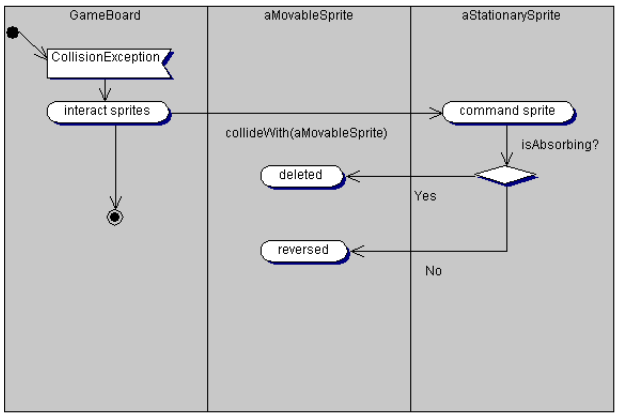


Fig. 7. Collision handling algorithm

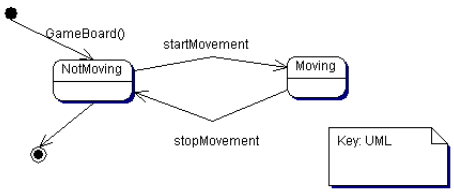


Fig. 8. Simple state diagram

Table 3 shows the division of testing responsibilities between the core asset developers and the product builders for code-based assets. In the earlier test points, shown in Figure 1 the core asset developers have the major responsibility while the product builders do incremental reviews for the portions that are added.

The product line test plan maps the information in Table 3 into a sequence of testing activities. The plan illustrates the concurrency possible in the test activities where the core asset team is testing new core assets while products built from previous assets are being tested by the product teams.

The IEEE standard outline for a test plan provides the entries in the left-most column in Table 4. I have added comments for some of the items that are particularly important in a product line environment. These comments apply to the product line wide test plan that sets the context for the individual plans for each core asset and product.

	Core asset builders	Product developers
Unit testing	Major responsibility Test to reusability level	Minor responsibility Test for specific functionality
Integration testing	Shared responsibility n-way interactions tested	Shared responsibility existing interactions tested
System testing	Minor responsibility test example products	Major responsibility test the one product

Table 3. Organization by test point

Defining the test method Once the production strategy has been defined, the product building and test methods can be developed. The method engineer works with testers and developers to specify the processes, models, and technologies that will be used during the various test activities. For example, if a state-based language is chosen as the programming language in the production method, then testing focuses on states and the notion of a "switch cover" [4] becomes the unit of test coverage.

In a software product line organization all of the other 28 practices impact testing. Some of the testing-related implications of the three categories of practices are shown in Table 5. The appendix contains a more detailed analysis of the relationship of testing to each of the other practice areas.

The test method describes the test tools that will be used. I will cover these more thoroughly in section 5.3 but I will give one example here. If development will be done in Java, the testers will use JUnit and its infrastructure. The test method in a data intensive application development effort might specify that unit testers will create datapools [5] to automate the handling of large amounts of data. The method defines techniques and provides examples such as the one below. This is the setUp method in a JUnit test class that uses datapools to apply test data to configuration cases.

```
protected void setUp() throws Exception {

// Initialize the datapool factory
IDatapoolFactory dpFactory = new Common_DatapoolFactoryImpl();

// Load the shoppingCartDatapool datapool
IDatapool datapool = dpFactory.load(
```

[4] A switch cover is a set of tests that trace all the transitions in the design state machine. A one-way switch cover makes one circuit of the transitions, a two-way switch cover makes two passes through the transitions, etc.

[5] A datapool is a specially formatted file of data for test cases. The test case contains commands that retrieve data as a test case is ready to be run. This facilitates sharing test data across different test cases and different software development efforts.

Introduction	This is the overall plan for product line testing.
Test Items	All products possible from the core assets.
Tested Features	Product features are introduced incrementally. As a product uses a feature it is tested and included in the core asset base.
Features Not Tested (per cycle)	
Testing Strategy and Approach	Separate strategies are needed for core assets and products.
Syntax	
Description of Functionality	
Arguments for tests	
Expected Output	
Specific Exclusions	
Dependencies	The product line test report should detail dependencies outside the organization. Are special arrangements necessary to ensure availability over the life of the product line?
Test Case Success/Failure Criteria	Every type of test should have explicit instructions as to what must be checked to determine pass/fail.
Pass/Fail Criteria for the Complete Test Cycle	
Entrance Criteria/Exit Criteria	
Test Suspension Criteria and Resumption Requirements	In a product line organization a fault found in a core asset should suspend product testing and send the problem to the core asset team.
Test Deliverables/Status Communications Vehicles	Test reports are core assets that may be used as part of safety cases or other certification procedures
Testing Tasks	Test Planning Test Construction Test Execution and Evaluation
Hardware and Software Requirements	
Problem Determination and Correction Responsibilities	Important that this reflect the structure defined in the CONOPS
Staffing and Training Needs/Assignments	
Test Schedules	
Risks and Contingencies	The resources required for testing may increase if the testability of the asset specifications is low. The test coverage may be lower than is acceptable if the test case selection strategy is not adequate. The test results may not be useful if the correct answers are not clearly specified.
Approvals	

Table 4. IEEE Test Plan Outline

	Core asset development	Product development
Organizational management	Shift traditional emphasis from backend to frontend testing.	Consider the impact of product sequence on development of assets including tests
Technical management	Coordinate development of core assets with product development. What testing tools are delivered with the core assets?	Provide configuration support for tests as well as development artifacts
Software engineering	Design for testability	Use testing to guide integration

Table 5. Implications of practice area categories for personnel roles

```
new java.io.File("c:\\courses\\cpsc372\\velocity\\velocity
\\velocityPool.datapool"), false);

// Create an iterator to traverse the datapool
dpIterator = dpFactory.open (datapool,"org.eclipse.hyades.datapool.
iterator.DatapoolIteratorSequentialPrivate");

// Initialize the datapool to traverse the first equivalence class.
dpIterator.dpInitialize(datapool,0);
}
```

This is a reusable chunk that could be packaged and used by any JUnit test class.

Design for Testability For a given component, if it is implemented in a single-product, custom development effort, we will assume that a component C is executed x times for a specified time period. In development where multiple copies of the product are deployed the same component will now be executed

$$nc * x$$

times in a specified time period, nc is the number of copies. In product line development, the same component will now be executed

$$\sum_{i=1}^{np}(nc_i * x_i)$$

times in the same time period, where np is the number of products, nc_i is the number of copies of a given product, and x_i is the number of executions for a given product.

In the above scenario, assume that the probability of a defect in the component causing a failure is $P(d)$. Obviously the number of failures observed in the

product line scenario will likely be greater than the other two scenarios as long as $P(d)$ remains constant. The expected number of failures can be stated as:

$$expectedNumFailures = P(d) * \sum_{i=1}^{np} (nc_i * x_i)$$

In product line development, a component is used in multiple products. These products may have different levels of certain quality attributes and different types of users. We expect that the range of input data presented to a component will vary from one product context to another. Therefore, $P(d)$ does not remain constant, it varies from product to product. If we assume that when a failure occurs in one product, its failure is known to the organization, the number of failures can now be stated as:

$$expectedNumFailures = \sum_{i=1}^{np} P_i(d) * (nc_i * x_i))$$

I will discuss later that this aggregate decrease in testability may require additional testing.

Part of developing the production method is specifying the use of techniques to ensure that assets are testable. There are two design qualities that guide these definitions:

- Observability - provide interfaces that allow the test software to observe the internal state of the artifact under test. Some languages provide a means of limiting access to a particular interface so that encapsulation and information hiding are not sacrificed. For example, declaring the observation interface to have package visibility in Java and then defining the product and test assets in the same package achieves the objective. A similar arrangement is possible with the C++ *friend* mechanism.
- Controllability - provide interfaces that allow the artifact under test to be placed in a particular state. Use the same techniques as for observability to preserve the integrity of the artifact.

Testability is important in a software product line organization because the presence of variability mechanisms makes it more difficult to search through an asset for defects. A level of indirection is introduced by many of the mechanisms.

The product line infrastructure may provide tools that provide a *test view* of the assets. This test view allows test objects to observe the state of the object under test and to set the state to any desired value. A module can provide application behavior through one interface that hides its implementation while providing access to its internals through another, transparent interface. The transparent test interface may be hidden from application modules but visible to test modules. One approach is to use package level visibility and include only the module and its test module. Or, the test interface may be protected by a tool that checks, prior to compilation, for any reference to the test interface other than from a test module.

Test Coverage Test coverage is a measure of how thorough the search has been with the test cases executed so far. The test plan defines test coverage goals for each test point. Setting test coverage levels is a strategic decision that affects the reputation of the organization in the long term and the quality of an individual product in the short term. It is also strategic because the coverage goals directly influence the the costs of testing and the resources needed to achieve those goals.

"Better" is often one of the goals for products in a product line where better refers to some notion of improved quality. One approach to achieving this quality is to allow the many users of the products find defects and report them, so they can be repaired. However, in many markets letting defects reach the customer is unacceptable.

An alternative is to require more thorough test coverage of the core assets compared to coverage for typical traditional system modules. The earlier discussion on testability leads to the conclusion that traditional rules of thumb used by testers and developers about the levels of testing to which a component should be subjected will not be adequate for a software product line environment. The increased number of executions raises the likelihood that defects will be exposed in the field unless the test coverage levels for in-house testing are raised correspondingly. This is not to say that individual users will see a decline in reliability. Rather, the increased failures will be experienced as an aggregate over the product line. Help desks and bug reporting facilities will feel the effects. If the software is part of a warranted product, the cost of repairs will be higher than anticipated. The weight of this increase in total failures may result in pressures, if not orders, to recall and fix products[26].

The extra level of complexity in the product line - the instantiation of individual products - should be systematically explored just as values are systematically chosen for parameter values. I will refer to these as *configuration cases* because they are test cases at one level but are different from the dynamic data used during product execution. A product line coverage hierarchy might look something like this:

- select configurations so that each variant at each variation point is included in some test configuration, see section 5.3; obey any constraints that link variants;
- select variants that appear together through some type of pattern even though there is no constraint linking them;
- select variant values pair-wise so that all possible pairs of variant values are tested together;
- select higher order combinations.

Other coverage definitions can be given for specific types of artifacts. Kauppinen et al define coverage levels for frameworks. The coverage levels are defined in terms of the hook and template implementations in a framework[21]. The coverage is discussed for feature-level tests in terms of an aspect-oriented development approach.

5.3 Test Construction

In a product line organization, tests are executed many times:

- as the artifact it tests is iteratively refined, and
- as the artifact it tests is reused across products.

The test assets must be constructed to meet these requirements. This includes:

- technologies to automatically execute test cases, and
- technologies to automatically build configuration cases.

The technologies and models used to construct test assets should be closely related to the technologies and models used to construct the product assets. This coordination occurs during production planning. As test software is constructed, the design of the products is considered to determine the design of the test artifacts.

There are several types of variation mechanisms. Here is one high level classification of mechanism types [12]:

- Parameterization - Mechanisms here range from simply sending different primitive data values as parameters to sending objects as parameters to setting values in configuration files.
- Refinement - Mechanisms include inheritance in object-oriented languages and specializers that use partial evaluation.
- Composition - Container architectures in which components "live" in a container that provides services.
- Arbitrary transformation - Transformations based on underlying meta-models that take a generic artifact and produce an asset.

Binding time The variation mechanisms differ in many ways, one of which is the time at which definitions are bound. There is a dynamic tension between earlier binding that is easier to instrument and verify and later binding which provides greater flexibility in product structure.

The choice of binding time affects how we choose to test. Mechanisms that are bound during design or compile time can often be statically checked. Mechanisms that bind later must be checked at runtime and perhaps even over multiple runs since their characteristics may change.

In an object-oriented environment, some mechanisms are bound to classes while others are bound to objects. This generally fits the static/dynamic split but not always. It is possible to verify the presence of statically defined objects in languages such as Java. This is a particularly important issue since these static objects are often responsible for instantiating the remainder of the system and their failure will prevent there from being a system instance to test.

Test Architecture A software product line organization is usually sufficiently large and long lived that the tools and processes need to be coordinated through an architecture. The test architecture controls the overall design of the test environment.

Several research and experience reports point to the fact that the design of the test environment should parallel the design of the product. Cummins Engines, a Software Product Line Hall of Fame member, has experience that reinforces the notion of the product architecture and the test architecture having a similar shape. "Tests must be designed for portability by leveraging the points of variation in the software as well as in the System Architecture"[33]. In fact at a top level they view the test environment architecture as a natural part of the product line architecture.

A product line test architecture must address some specific range of variability and the accompanying variety of binding times. If the range in the product line is very large, it may be reasonable to have multiple architectures and this usually happens between test points. The architecture for unit testing usually needs to have access to the internals of components and will tie into the security model of the programming language. The architecture for a GUI tester will likewise tie into the windowing model. For the very latest binding mechanisms it is necessary to tie into the execution environment such as the JVM for a Java program.

Aspect-oriented Techniques Aspect-oriented programming is one technology used to implement variation in core assets. An aspect is a representation of a cross cutting concern that is not the primary decomposition. Research has shown that for code that is defined in terms of aspects, the test software should also be defined in terms of assets [22]. In that way, the test software is bound at the same time as the product software.

I will not give a tutorial on aspect-oriented programming here but I will talk about some characteristics of aspects that make them a good variation mechanism.

An aspect is a design concern that cuts across the primary decomposition of a design. Canonical examples include how output or security are handled across a product. Aspect-oriented techniques do not require any hooks to exist in the non-aspect code before the aspects are added. The aspect definition contains semantic information about where it should be inserted into the code. An aspect weaver provides that insertion either statically or dynamically depending upon the design.

Shown below is the *paint* method for a DigitalScoreBoard that is being used in products that use the Java Microedition. The microdedition is for small devices such as cellphones. A different *paint* method is used when the base code for the scoreboard is used in the Java Standard Edition. The rest of the scoreboard code is common.

```
package coreAssets;

import javax.microedition.lcdui.Graphics;

public aspect DigitalScoreBoardA {
private static final int color = 255 << 16 | 255 << 8 | 0;
void around(ScoreBoard gsb, Graphics g) :
call(void ScoreBoard.paint(Graphics))
&& target(gsb)
&& args(g)
{

g.setColor(color);
String scoreStr = gsb.score();
int offset = (g.getFont().stringWidth(scoreStr)) / 2;
g.drawString(scoreStr, gsb.getLocation().getRealX() - offset,
gsb.getLocation().getRealY(), Graphics.TOP | Graphics.LEFT);
}
}
```

Aspect-orientation is a useful variation mechanism, particularly for unanticipated variation since no hooks are needed. Implementations of aspect-oriented programming such as AspectJ provide a variety of design constructs including inheritance and composition.

An aspect adds a fragment of behavior to the existing behavior of an object. This can be exploited for testing. In the figure below classes are defined that implement two types of scoreboards for the product line. Each class defines all of the behavior needed to keep score but does not define the method need to present it on the display. Two types of display are supported via aspects. One is for use with the Java Micro Edition and the other is for use with Java Standard Edition.

A test class is defined for each scoreboard class. Then a fragment of behavior is added to the test class that examines the graphics portion of the class behavior. There may be multiple test cases defined in the test aspect.

Assembly Core assets may be assembled from smaller pieces to incorporate the selected variants. These variants are bound at the time of this assembly. Assets, such as unit tests, may be composed to provide an increase in reuse opportunities[13].

One technology used for this purpose is the XML-based Variant Configuration Language (XVCL). The language provides a set of XML tags that are processed by an engine. Non-control language XML tags and all other content is simply streamed out. XVCL can be used to produce files that obey any format. Table 6 shows a small section of XVCL code. The *var* statements set variables to hold the names of two output files. The *adapt* statements feed the processor with XVCL code fragments.

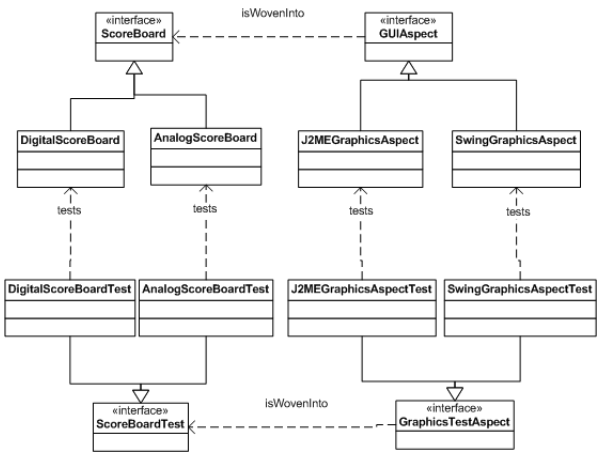


Fig. 9. Aspect design including tests

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="main.xvcl">
    <!–add variation points that are selected–>
    <set-multi var="variationPoints" value="vp1.1,vp1.2"/>
    <!–set actual file names for variables–>
    <set var="codefile" value="c:/out.txt"/>
    <set var="testfile" value="c:/testout.txt"/>
    <adapt x-frame="baseSystem.xvcl"></adapt>
    <adapt x-frame="registerUseCase.xvcl"></adapt>
    <!–add other use cases–>
</x-frame>
```

Table 6. XVCL segment

The one XVCL program will output both the asset, to out.txt, and the test asset, to testout.txt. This facilitates traceability by closely associating the product and test code.

The assembly approach requires that a complete implementation be developed and then decomposed into the appropriate chunks, breaking apart into regions of commonality. Then a set of pieces for each variant. The variant pieces can be produced as needed or produced up front for later use.

One advantage of XVCL is that it can be used with technologies that have no concept of variation. For example, a user's manual can be split at points which correspond to variation points in the product. Each piece is captured in an XVCL frame. Then control logic is added to select the correct sub-frames based on variant choices. The hierarchy of frames is resolved based on the variant choices. Each frame that is accepted is written to the output.

This approach can handle RTF files, XML files, text files, or any other format. From the example in Table 6 you can see that multiple outputs can result from the specification of one product.

More is needed. XVCL will traverse a frame hierarchy but the resulting output needs to be compiled, linked or jarred, and perhaps incorporated into an installer package. The Buckminster project of the Eclipse Foundation provides a realization service that will use tools such as XVCL to extract files from a repository and then apply a variety of tools.

This tool chain allows us to blend assets to reduce traceability problems between product and test assets, and other assets such as service manuals and other supporting information.

Generation Automatic generation is the most widely used technique. Object code is generated from source code written in a "higher-level" language. Model driven development (MDD) moves this up a level of abstraction by using design models and templates as the "source code." The model provides the situation-specific information. The templates embody patterns. The generator replaces the variability in a template with information from the model. The output is an instantiation of the pattern in an output format chosen when the template was designed.

The advantage of this approach is in the generality of the templates. A template that can be used to generate the shell for any Java class is much more useful than a template that can only be used to generate an algorithm designed to solve a problem in one product. Templates are useful for situations where the time required to abstract out the variabilities is more than compensated for by the time that would be required to complete each individual occurence of the code pattern. For example, intricate algorithms, like calculating the Doppler effect between a satellite and its ground station can be captured once in template. A second advantage is that a review of the template provides a partial review of the code that results from instantiating the template.

The primary disadvantage of generation is the effort required to achieve the appropriate level of abstraction. Failure to attain that level results in a generator that produces very incomplete code that needs much manual work. The template instantiation process produces output that may be source code in a programming, or scripting, language. This must still be converted into an executable form by compiling the output. A second disadvantage is the difficulty of coordinating the application of multiple templates.

Test patterns, see section 5.3, are used as the basis for templates used to generate test software. One pattern is the "Test the product from the perspective of the user." This can be instantiated in a script for a GUI-based test tool such as Abott. This activity follows a process like the following:

- The initial script is generated by operating the product by hand, in record mode.
- The resulting script is captured, converted into a template by replacing variation points with variables, and adding control tags.

– The template can be instantiated multiple times with specific values chosen from a test case model.

Generation requires more planning and initial effort than simply writing a JUnit test case. This effort is rewarded across the product line by greatly reduced effort for subsequent products.

Combinatorial Testing The variability in a product line leads to an explosion of interactions that would quickly require billions and trillions of test cases to test exhaustively. Combinatorial test design techniques can just as quickly reduce the number of tests needed to achieve effective levels of test coverage to a manageable number, say about 30 to 60.

The test designs from a combinatorial design technique are based on defining combinations of parameter values. At the simplest level all pairs of parameter values are selected. Most combinatorial techniques can be adjusted to generate 3-way, 4-way up to n-way possibilities. Most reports have found that pair-wise testing will result in 90%+ test coverage.

The usual use of combinatorial tests is in the selection of multiple parameters for unit testing of methods. In a product line, there is a second place where combinatorial testing is useful - configuration cases in which products are instantiated for testing. Combinatorial techniques are used to select the configuration for a product. Then each instantiated product is executed using an instantiated set of data values chosen using combinatorial design.

This approach is referred to as Design of Experiments in some literature. The orthogonal array approach of Taguchi has been automated in several tools[25]. Minitab, and other statistical packages, can be used to generate the combinatorial designs. I will illustrate with a short example from the AGM case study.

– State the test scenario. - *Each AGM game is to be able to run on multiple platforms and the optional scoreboard is one of the display items that may expose problems between platforms.*
– Choose the number of factors you want. - *This will be a 3 factor experiment.*
– Identify and list the factors that are involved - *The Game, the operating system, and the type of scoreboard.*
– Identify and list the levels for each factor - *For Game the levels are Brickles, Pong, and Bowling. For operating system: windows, apple, and linux. For the scoreboard: digital scoreboard, analog scoreboard, and no scoreboard.*
– Choose the appropriate design in minitab. - See Figure 10
– Map the design onto the problem data. - See Figure 11
– Each row in the orthogonal array is a test vector that defines the configuration of a product. Each configuration is then subjected to a set of test data.

This table can be outputed as a data file that JUnit uses as a *datapool*. The Junit test cases:

– read the configuration cases,

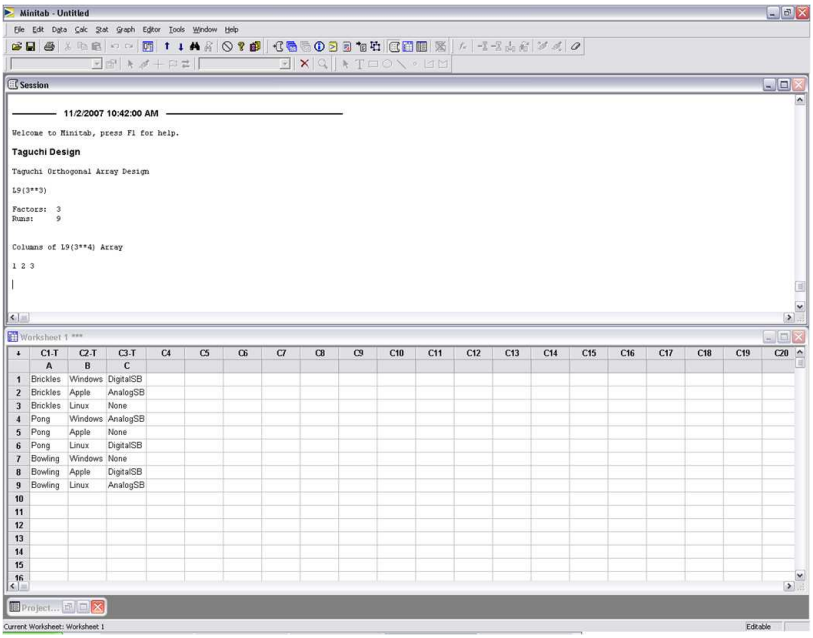


Fig. 10. Minitab setup of Taguchi experiment

- create an object under test (OUT) for each configuration,
- execute all the test cases on each OUT, and
- publish the results.

Test patterns Test patterns represent a useful mid-level test coverage criteria. Typically a test pattern represents a portion of the design that cuts across architectural boundaries and is large enough to have interesting behavior and therefore interesting defects. Running test cases that are based on the applicable test patterns uncovers interactions among a set of units.

There are numerous sources of test patterns. Every design pattern provides the opportunity to define a corresponding test pattern. A number of test patterns have been discovered[27, 3]. The Mock object test pattern has been implemented in several frameworks that generate the mock objects automatically[24].

The pattern approach is also useful for automating test generation. Generation usually calls for a template that is instantiated with variant choices. The template is nothing more than a pattern captured in a templating language such as JSP or the Java Emitting Templates (JET) from the Eclipse project.

For example, the *Test Observer* pattern is associated with the *Observer* design pattern. In many instantiations, the Observer is supposed to initiate some action upon receipt of an event of a specific type. The Observer is attached to what it is observing. The TestObserver pattern is shown in Figure 12

	A	B	C
1	Brickles	Windows	DigitalSB
2	Brickles	Apple	AnalogSB
3	Brickles	Linux	None
4	Pong	Windows	AnalogSB
5	Pong	Apple	None
6	Pong	Linux	DigitalSB
7	Bowling	Windows	None
8	Bowling	Apple	DigitalSB
9	Bowling	Linux	AnalogSB

Fig. 11. Orthogonal array for top level AGM example

The Test Observer registers for the same event as the Observer it is testing. When it receives the expected event it queries the Observer to determine that it has reacted as expected. It may also interrogate the object being observed. The Test Observer tests the interaction of the Observer, its observed object and any objects that the Observer is supposed to notify.

A product line organization will identify some patterns that it is using to implement specific behavior. Developing test cases from these patterns will ensure that the patterns are being implemented correctly. The organization may decide to automate the generation of code from some of the more widely used patterns.

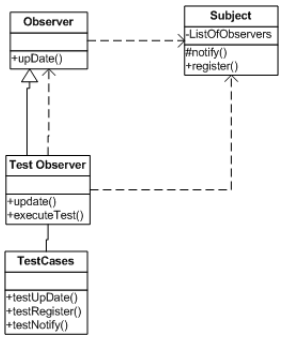


Fig. 12. TestObserver pattern

There are many techniques for constructing test artifacts. I have only presented a few because most of these techniques are not specific to product lines. With the wide range of test points in a product line effort most existing test techniques are applicable at some points.

5.4 Test Execution and Evaluation

Tests should be executed early and often. The results of executing the tests should be accumulated so that a history of data used and pass/fail rates can be used to support the maintenance of the individual core assets. This data can also be used as a measure of the health of the core asset base.

Depending on the development method used, the unit tests may be executed beginning on the first day of code development or they may not even be created until the unit is largely implemented. Writing the tests can be a way of testing one's understanding of the requirements on the unit and can raise questions early that will reduce the need to rewrite code later.

The development environment should support the rapid execution of tests and concise reporting of results. For example, Eclipse has integrated JUnit into its Java Development Tools.

Advances in testing graphical user interfaces allow increased automation. Figure 13 shows a test run with the Brickles product. Much of the implementation of this test suite can be automatically generated or the record and playback mechanism can be used.

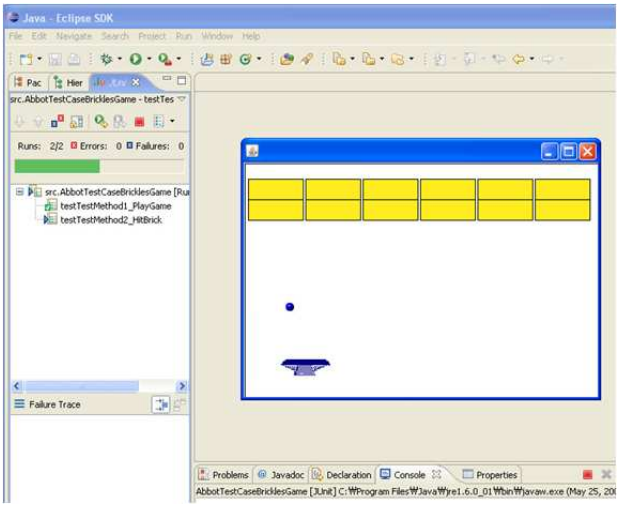


Fig. 13. Abbot tester

Below is a fragment of the Abbot test script.

```
import junit.extensions.abbot.*;
import abbot.tester.ComponentTester;

public class AbbotTestCaseBricklesGame extends ComponentTestFixture {
```

```
public void testTestMethod1_PlayGame() {

// brickles.TODO();
assertTrue(true);
// assertTrue(brickles.TODO == true);
}

public void testTestMethod2_HitBrick() {

brickles = new BricklesView();
brickles.preinit(brickles);
System.out.println("Start Test 1");

int counter = 0;
int brickGone = -1;
while (brickles.getPuck()._kaput == false) {
System.out.print(" ");
brickles.moveMouse(brickles.getPuck().getPosition());
BrickPile pile = brickles.getBrickPile();
counter = 0;
for (int i = 0; i < pile.getSize(); i++) {
Brick b = pile.getBrickAt(i);
if (b.isHit()) {
brickGone = i;
}
counter++;
}
if (brickGone != -1)
break;
}

assertTrue("Brick is gone", brickles.getBrickPile().getBrickAt(
brickGone).isHit() == true);
System.out.println(counter);
assertTrue(counter - 1 == 11);
}
```

This script was partially generated using the JET template fragment given below the generated code.

```
<c:iterate select="/agm/usecase" var="currUseCase">
public void testTestMethod<c:get select="$currUseCase/@name"/>() {

<c:iterate select="$currUseCase/stimulus/userActions/actionPerformed" var="currAction">
<c:choose select="$currAction/@action">
```

```
<c:when test="'PlayGame'"> //brickles.TODO();</c:when>
<c:when test="'HitBrick'">
brickles = new BricklesView();
brickles.preinit(brickles);
System.out.println("Start Test 1");

int counter = 0;
int brickGone = -1;
while (brickles.getPuck()._kaput == false) {
System.out.print(" ");
brickles.moveMouse(brickles.getPuck().getPosition());
BrickPile pile = brickles.getBrickPile();
counter=0;
for (int i = 0; i < pile.getSize(); i++) {
Brick b = pile.getBrickAt(i);
if (b.isHit()) {
brickGone = i;
}
counter++;
}
if (brickGone != -1) break;
}
</c:when>
</c:choose>
</c:iterate>

<c:iterate select="$currUseCase/response/reactions" var="reaction">
<c:choose select="$reaction/@reactingObject">
<c:when test="'PuckMoving'">
assertTrue(true);
//assertTrue(brickles.TODO == <c:get select="$reaction/@valueExpected"/>);
</c:when>
<c:when test="'BrickCount'">
assertTrue("Brick is gone", brickles.getBrickPile().getBrickAt(brickGone).isHit() == tru
System.out.println(counter);
assertTrue( counter - 1 == <c:get select="$reaction/@valueExpected"/>);
</c:when>
</c:choose>
</c:iterate>
}

</c:iterate>
```

6 The Business of Testing

A software product line organization is founded on a sound business case and any major activity is usually justified in the same manner. The Structured, Intuitive Model of Product Line Economics (SIMPLE) is one technique for quantifying the justification[5, 4]. The model defines four cost functions, two of which are fixed costs and two are per product costs. The model also defines a benefits function. The basic formula is:

$$\sum_{j=1}^{m} Benefits - (C_{org} + C_{cab} + \sum_{i=1}^{NumProducts} (C_{unique_i} + C_{reuse_i}))$$

Each of these functions is applied to all of the product line assets. In the discussion that follows I will focus on the testing assets.

The SIMPLE approach supports thought experiments about alternative techniques. In the following I will give the meaning of each function with respect to testing assets.

C_{org} - The cost of resolving organizational issues related to testing. All personnel who assume a testing role must be trained in that role. Organizational management will participate in developing the production strategy and the organization's test strategy.

C_{cab} - The cost of the core assets for testing. The test infrastructure and other testing tools must be acquired and perhaps adapted. The TPTP project of the Eclipse Foundation provides testing tools as open source software. This test infrastructure supports all code-based test points. The core asset team will extend the basic infrastructure to handle specific test issues. There is on-going expense as requirements evolve and defects are reported. Test case software is written as new classes are defined in response to new features and bug fixes and as new combinations of product assets are aggregated into new subsystems.

C_{unique_i} - The cost of the unique test software that must be built for a specific product or other single use test situation. Some portion of every product is unique from the other products. Test cases, and perhaps additional test drivers, must be created and must be executed.

C_{reuse_i} - The cost of getting ready to reuse the testing core assets. Despite efforts to build good assets, there will be some learning curve before the testers can effectively use the core assets. The attached processes are intended to reduce this learning curve.

$Benefits$ - The benefits realized as a result of incurring the costs. In the case of testing, these benefits are not easy to quantify. Customers experiencing higher quality, retaining reputation, elimination of litigation, and other risks reduced can be estimated.

Obviously reducing the per product costs is the best way to reduce overall costs. Automation of test creation and execution will reduce C_{reuse}. A thorough commonality and variability analysis will reduce the variability to the minimum and will identify many of the variation points. This will reduce C_{unique}.

Ganesan et al used a variation on the basic SIMPLE formula to compare testing strategies[14]. They compared a strategy in which each product was tested individually from scratch to a strategy in which an infrastructure was tested once and the product testing was limited to the product specific behavior. Their study showed a 13% savings in test costs. They did not consider a number of known reduction techniques discussed in this paper.

They did use Monte Carlo simulations, which have been used in a number of economic studies related to product lines to compensate for the uncertainty of the estimates of model parameters. The longevity of a product line would magnify inaccuracies. They ran thousands of trials varying the input parameters. The 13% savings forecasted is an estimate based on those simulation runs.

Another SIMPLE-based study that has implications for testing showed that over time the core asset base needs to be refreshed which requires an infusion of funding. Ganesan et al considered the notion of a product line generation[15]. When a new generation is created, the same test obligation will exist as during the initial creation. Through the use of incremental techniques, the effort can be reduced but must be planned as part of the overall product line planning.

7 Case Study

The Arcade Game Maker product line from AGM is a set of 3 nostalgia video games: Brickles, Pong, and Bowling. AGM wants to use an iterative, incremental approach in which the three products are released to three different markets over a relatively short period of time.

Besides the variation in platform and the obvious variation in types of game there is also a variation in platform. The first increment was a .NET implementation, the second and third were Java-based. The second was for cellphones and PDAs and was based on the J2ME. The third was a more generic version and was based on the J2SE.

During production planning AGM decided they wanted to set a goal of finding 90% of the defects no later than unit test, 9% at integration test, and the remaining 1% at system test time. As their approach matures, they would expect to find an increasing portion of requirements defects during the review process prior to any testing.

7.1 Guided Inspection

AGM has used a review process for many years but a detailed study revealed that too many design defects were being coded because they escaped detection in design reviews. AGM decided that along with the new product line strategy they would implement the more disciplined review process. Their informal review process was not finding things that were not there. That is, the review process did not define sufficient context to allow for identifying missing pieces.

The inspection team was trained to first develop a set of scenarios that capture the essential features of the artifact to be reviewed. In many cases the

scenarios could be derived from the use cases being developed as part of the requirements effort. The scenarios are prioritized based on risk and as many as possible are applied.

7.2 Core asset development

During core asset development, guided inspection was used extensively to evaluate the requirements, architecture, and detailed design models. The team found that the guided inspection process was very effective and eliminated a large number of trivial defects as well as several classes of major ones. They had greater confidence that the models used to guide the development of code were accurate and would result in good code.

Unit tests In order to achieve their goal of 90% of defects found, the test plan specified a coverage level termed "every branch" coverage for structural tests and an "every variant" level of coverage for parameters.

Integration tests AGM used a number of design patterns in defining subsystems. The integration test team used test patterns that correspond to the design patterns used in the design. These tests exercised variant combinations among the pieces being integrated. A number of defects were found that resulted from a failure to make the architecture sufficiently specific.

Sample system tests AGM used a combinatorial test design to selecte specific test configurations for testing at the core asset level. Not every product had all the features it would have once the product team added the unique portion, but they had the core functionality. This testing gave the core asset team valuable feedback about their assets and generated a To-Do list of enhancements for the second release.

7.3 Product development

The product development team benefited from all the testing done by the core asset team. The product team used guided inspection where the unique features made up a significant portion of the product.

The product team found only a few defects in the core assets as they integrated their product-unique parts. Most of the defects were within the unique code. In a few cases there was a mismatch between the core assets and the product-unique piece. These were mostly due to conflicting assumptions that did not show up in testing since the same assumptions were applied there as well.

8 Issues

In this section I will present more questions than answers but will illustrate some of the issues on the cutting edge of research.

8.1 Core assets

What is a core asset? Traditional product line approaches think of a core asset as discipline related such as a requirements document. But what if the core assets are functionally divided? A core asset is the set of hardware, software, test documentation and results, and other items related to a functional subsystem that will be reused as a unit? In the first case the set of all test suites for a test point in the product line is a core asset. In this scheme, traceability must be maintained between a specific test suite and the software it is used to test. In the second case a set of test suites is grouped with what it tests. In this scheme, traceability must be maintained between a test suite and any other test suites it uses or is derived from.

A related issue is what constitutes a configuration item and a configuration. In some product line organizations, the core asset base as a whole is one configuration item. Each release is a release of all the core assets. This usually results in a lengthy wait for the next release and a long wait for fixes to identified problems. A finer grained approach can be used. Some natural breakpoint is chosen, such as the architectural subsystem level or module level. A release is a new version of a subsystem or module. This approach results in a faster cycle time with releases of bug fixes happening more often.

The implication for testing is the faster the cycle time, the faster the testing process must react. Testing must be as agile as the development work. How do we do agile testing?

8.2 Software certification

In a related issue, how do government regulators regard reuse of certified assets? What has to be done to ensure that the regulations are met in an acceptable way? Certification of software products by regulatory agencies usually require specific evidence, including test results.

The familiar phenomenon of *emergent behavior* confounds the easy answer of testing each subsystem to a high degree of coverage and then assuming that a product assembled from these subsystems is of equally high quality. This complicates the product line approach of maintaining an inventory of core assets that are assembled to form products.

One possible approach is to store a fragment of a verification model with each asset. When the assets are assembled, the fragments of the verification model are assembled. Some form of incremental model checking algorithm might be possible.

Another approach is to structure the product line as a completed supersystem from which optional pieces are removed when not needed. Is it possible to have emergent behavior in the remaining system when pieces are taken away from that system?

8.3 Meta-data

"Meta-data" is data about data. This data can be used to reason about the data described by the meta-data. Our research is considering what meta-data should be collected about tests. Current examples include only keeping the usual data such as the name of the person who constructed the test case and when [28]. We believe that using an approach similar to the Java annotations, in which the meta-data are directly associated with the source code segments, can be used by tools to manage variability and automate processes.

Meta-data conforms to a meta-model. The meta-model defines the allowable structure of the meta-data. Our current research is focusing on defining the appropriate meta-model for test data. Current efforts at automation, discussed below, are based on existing meta-models.

8.4 Automation

Although I have talked about a number of ways to automate, this is still an issue with much work left to be done. Our research group is currently investigating a tool chain that would provide a more comprehensive approach to automation **??**.

Our premise is that the constrained scope of a product line provides a sufficiently narrow context that a domain specific language (DSL) can be profitably used. We take practices from the framework and exploit their relationships with each other.

We begin by Understanding Relevant Domains through the development of an ontology. We use the Web Ontology Language (OWL) to capture the concepts in the domain. The ontology is transformed into a DSL by adding constraints, cardinality, and other semantic information. We use the Ecore meta-model from the Eclipse Foundation to represent the language.

The Atlas Transformation Language (ATL) is used to transform the output of one tool into the input suitable for another tool. Meta-models are defined for both the source and target of the transformation. Then functions, or rules as shown in the code snippet below, map an element from the source to elements in the target.

```
module AGM2EMF;
create OUT : GameElements  from IN : MOF;


-- get BarrierBahavior from original ecore file in a Sequence
helper context MOF!EPackage def: getBB() : Sequence(MOF!EClass) =
 MOF!EClass.allInstances()
        ->asSequence()
        ->iterate(e; acc: Sequence(MOF!EClass) = Sequence{} |
        if e.eSuperTypes.toString() = 'Sequence {IN!BarrierBehavior}'
        then
                acc.append(e)
```

```
    else
                acc
        endif
);

-- get Rule from original ecore file in a Sequence
helper context MOF!EPackage def: getRule() : Sequence(MOF!EClass) =
 MOF!EClass.allInstances()
->asSequence()
->iterate(e; acc: Sequence(MOF!EClass) = Sequence{} |
if e.eSuperTypes.toString() = 'Sequence {IN!Rule}'
then
acc.append(e)
    else
  acc
  endif
);
```

The Ecore meta-model is accompanied by a set of tools that apply general templates to generate utility applications to work with the model. We can automatically build editors from the Ecore model that will support writing use cases for the product line or specific products. The use cases follow the stimulus and response pattern. A similar editor can be generated that supports the development of test cases.

A mapping is established between the concepts in the DSL and the methods in the implementations of the concepts. This mapping is exploited both for developing the product functionality and the test case semantics. The mapping is supplemented with patterns, both standard design patterns and domain specific patterns, which are captured in templates.

Figure 14 shows the comprehensive flow.

Yet to be determined is exactly how much of the process can be automated.

9 Conclusion

The software product line development strategy provides the opportunity for significant advances in product production. The strategic levels of reuse make it profitable to spend more time on assets. We can (and in some cases must) reach for more thorough levels of test coverage to compensate for the wider range of inputs that a module will experience across all the products.

The range of inputs depends on the variation among the products. The scope of the product line provides a constraint on the variability. This constraint allows us to make assumptions that facilitate automation.

To achieve some of the aggressive goals of a product line organization, more of the development steps must be automated. The initial cost of this increased automation is amortized over the multiple products. The automation comes from specific models and tools that are used.

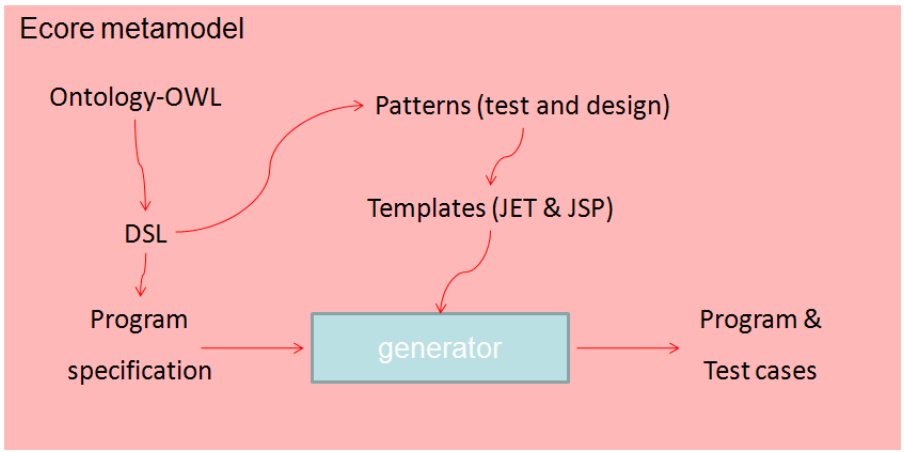


Fig. 14. Toolchain for development/test

Using meta-models to capture abstractions that define the product line provides a basis for creating specific product models rapidly. From those models much of the product can be generated. Tests as well as product behavior can be generated, sometimes from the same models.

Our current research is taking us deeper into representations and restrictions. We continue to get experience with product line creation and sustainment. In particular we are looking at the relationships between development and testing and how representations can unify the two.

10 Acknowledgements

I want to thank Kyungsoo Im and Tacksoo Im for their work on implementations and John Hunt for the implementations from his dissertation.

A Appendix - Practice Areas

In this appendix I briefly relate testing to the other 28 practice areas in the SEI's framework.

Software Engineering

Architecture Definition	The architecture is defined to a specific level of detail. The test cases for the architecture must be defined to the same level of detail as the architecture. The architecture is "tested" by comparing its definition to the requirements by the techniques defined in the Architecture Evaluation practice.
Architecture Evaluation	This area includes the ATAM technique. Architecture reviewers adopt the testing perspective to help them do an in-depth evaluation. It uses scenarios as does the Guided Inspection technique. These scenarios play the role of test cases. The scenarios are chosen to sample over the set of requirements. The sampling can be biased toward those portions of th erequirements that are a higher priority than some other area. The output of these techniques is not a "correct" or "incorrect" decision. It produces a set of risks that identify places in the architecture where it may be difficult to maintain correctness over the life of the architecture.
Component Development	The explicit interface of a component is used as the basis for functional unit test cases. In a product line it is likely that the interface will include some type of variability mechanism. This will increase the number of test cases needed to thoroughly test the component. Likewise the implementation may be more complex to support the variations required of the component. A test infrastructure that is compatible with the development method will be needed. For example, JUnit works well in a development method that is iterative and incremental. JUnit supports the evolution of test cases as the component matures.

Using Externally Available Software	Software imported into the core asset base must be tested to the same levels of coverage as in-house built code. Due diligence may find that the supplier of the software has already achieved this level of quality or the in-house team may have to develop addiitonal test cases. The test cases should be automated so that they can be reapplied when a new release of the external software is released to the organization.
Mining Existing Assets	When a fragment is mined from existing assets it needs to be tested in its new, more isolated context to be certain that the new asset is appropriately complete with no hidden dependencies.
Requirements Engineering	The requirements are one of the primary sources of input to the system test process. The requirements are also subject to verification by a Guided Inspection. The production method should prescribe the manner in which the requirements will be represented and how they can be linked to system test cases.
Software System Integration	Each level of integration is accompanied by exercising a test suite. It is at this level that previously unanticipated behaviors emerge. Test cases should cover significant interactions between the pieces being integrated to determine that each piece fulfills its responsibilities.
Understanding Relevant Domains	The domains about which our programs compute are an important source of information for testing the requirements, architecture, and assembled system. In MDD a domain specific language forms the basis for automating development. In a software product line domain models, from which domain specific langauges are created, should be tested before the language is created. A Guided Inspection of the model using the product line requirements as the context can identify incomplete, incorrect, or inconsistent models.
Technical Management	

Configuration Management	The test assets must be controlled and linked so that there is traceability to the sources of the test cases and to the asset they are intended to test. The configuration management system in a software product line uses a meta-data approach for linking since each asset will be used in many builds of many different products. A reusable component is stored in a core asset base repository. Then any use of the component in a product is expressed, in the product's CM tree as a configuration item that refers to the actual component's location. The tests for a product are managed in a similar manner. A product's CM tree contains a reference to the actual test cases that are stored in the core asset storage area. Both core asset and product developer's guide must provide nstructions on how to access needed assets and how to enter new assets into the storage area.
Measurement and Tracking	The test results should be tracked to understand how the defect profile is changing. This data should be used to manage the test process and to provide information for improving the development process. When a threshold value for the number of hours needed to find another defect is crossed, the test activity is halted.
Make/Buy/Mine/Commission Analysis	During the Buy or Commission analyses, there should be due diligence activities that investigate the quality culture of the potential supplier. The costs of testing, based on the supplier's culture, must be factored into these decisions.

Process Discipline	The numerous test processes must be coordinated with each other and with other processes that impact the same personnel and the same assets. The main threads in a software product line are the core asset development and product development efforts. As can be seen in the chain of quality in 1, there are numerous places where testig is conducted. There are also prerequsites in which one type of tesing should be conducted before another. The "discipline" in process discipline means that personnel who plan testing activities will provide a realistic schedule for them and the personnel assigned to carry out these activities will perform them according to the established process definitions.
Scoping	A well-defined, constrained scope limits the range of values that have to be considered during testing. It also limits the interactions and therefore the interaction tests that must be provided. A change in scope should trigger a review of various test plans to determine where additional tests are needed.
Technical Planning	Technical planning coordinates the various development and testing activities within the core asset and product building teams and between them as well. As periodic releases of the core asset base are scheduled the development schedules must include the appropriate time for test activities.
Technical Risk Management	Technical risk management will be informed by the results of testing. For example, if the defects found per hour of testing is not decreasing sufficiently, a risk should be raised. Testing will also identify areas of the architecture or specific types of code assets that are most problematic.

Tool Support	The development tools and the test tools need to be compatible. They both must support the range of binding times required by the selected variation points. The production method defines all of the technologies and related tools that are used for transformations, generation, and other construction techniques. It will also specify testing tools that complement those construction techniques.
Organizational Management	
Building a Business Case	The business case includes an analysis of the costs and time required for testing to the level expected in the culture and domain of the products being developed. The business case relies on a specific version of the product line scope to determine the levels of effort required for testing.
Customer Interface Management	Test results provide useful data for technical customers. The test results should be linked to the corresponding test cases and assets under test using the CM system. Customers can be engaged as testers early in the development cycle so that their suggestions can be incorporated during later iterations.
Developing an Acquisition Strategy	Test tools and assets may be acquired using this strategy.
Funding	Testing is expensive but high quality is no accident. Funding is necessary to support the planning, construction, and execution of test cases.
Launching and Institutionalizing	Although most organizations test during traditional development, adopting the product line approach will call for a shift with a heavier testing effort earlier in the cycle to ensure that the reusable assets are sufficiently robust.
Market Analysis	This analysis may influence the levels of coverage that are set for various types of testing. If customers are dissatisfied or if competitors are delivering a higher level of quality for a comparable price this may require an increase in test coverages.

Operations	Each specific test point should be assigned to a team with the appropriate knowledge and responsibility. For example, system test should be done by a team that has product line wide scope and responsibility.
Organizational Planning	During organizational planning issues such as desired levels of quality and product differentiation drive the specification of a production strategy that influences how the testing method is defined. Their efforts are coordinated through an organizational test plan that defines the overall test strategy.
Organizational Risk Management	Testing reduces the risk of litigation by identifying defects so they can be removed and not reach the customer. Test artifacts such as test results and reports are evidence that customers can review to understand the levels of quality being produced. In a product line organization, different products will present different levels of risk and will require different levels of testing.
Structuring the Organization	Personnel who are assigned to testing fulltime may form a functional team which is then matrixed to specific core asset and product teams as needed. Personnel with part-time responsibility for testing should be trained and placed in the appropriate places in the organization to have the authority necessary to be effective. This authority should include veto over passing defective assets or products on to the next step in development.
Technology Forecasting	The development technologies that will be used in the future may require different testing technology. Planners use the forecast to determine future needs for training, process definition, and tool support.
Training	Most personnel will be trained in the testing perspective and Guided Inspection techniques. Developers will be trained to effectively test their units. Personnel assigned responsibility for other types of testing may need training in the tools used to generate test cases.

References

1. Roger T. Alexander, Jeff Offutt, and James M. Bieman. Syntactic fault patterns in oo programs. In *Eighth International Conference on Engineering of Complex Computer Software (ICECCS '02)*, 2002.
2. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Professional. Addison-Wesley, 2003.
3. Robert V. Binder. *Testing Object-oriented Systems: Models, Tools, and Patterns*. Addison-Wesley, 1999.
4. Günter Böckle, Paul Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. Calculating roi for software product lines. *IEEE Software*, 21(3):23–31.
5. Günter Böckle, Paul Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. A cost model for software product lines. In *Proceedings of the 5th International Workshop on Product Family Engineering (PFE 2003)*, pages 310–316, 2003.
6. Gary J. Chastek, Patrick Donohoe, and John D. McGregor. A production system for software product lines. In *Proceedings of the Software Product Line Conference 2007*, 2007.
7. Ram Chillarege. Odc crystallizes test effectiveness. In *Proceedings of SPLiT - The Third Annual International Workshop on Testing Software Product Lines*, pages 31–32, 2006.
8. Paul Clements and Linda M. Northrop. *Software Product Lines : Practices and Patterns*. Professional. Addison-Wesley, 2002.
9. Scott G. Decker and Jim Dager. Software product lines beyond software development. In *Proceedings of the 11th International Software Product Line Conference*, 2007.
10. Josh Dehlinger, Meredith Humphrey, Lada Suvorov1, Prasanna Padmanabhan, and Robyn Lutz1. Decimal and plfaultcat: From product-line requirements to product-line member software fault trees. 2007.
11. Josh Dehlinger and Robyn Lutz. Plfaultcat: A product-line software fault tree analysis tool. In *Proceedings of Automated Software Engineering*, 2006.
12. Christoph Pohl et al. Survey of existing implementation techniques with respect to their support for the requirements identified in m3.2. Technical Report AMPLE D3.1, AMPLE consortium, 2007.
13. Markus Gälli, Orla Greevy, and Oscar Nierstrasz. Composing unit tests. In *Proceedings of SPLiT - The Second Annual International Workshop on Testing Software Product Lines*, pages 16–22, 2005.
14. Dharmalingam Gamesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. Comparing costs and benefits of different test strategies for a software product line: A study from testo ag. In *Proceedings of the Software Product Line Conference 2007*, pages 74 – 83, 2007.
15. Dharmalingam Ganesan, Dirk Muthig, and Kentaro Yoshimura. Predicting return-on-investment for product line generations. In *Proceedings of the Software Product Line Conference 2006*, pages 13 – 22, 2006.
16. Object Management Group. Software process engineering metamodel specification, 2005.
17. Georg Grütter. Challenges for testing in software product lines. In *Proceedings of SPLiT - The Third Annual International Workshop on Testing Software Product Lines*, pages 1–4, 2006.

18. William C. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Wellesly, MA, 1984.
19. IEEE. Ansi/ieee 829-1983 ieee standard for software test documentation. Technical report, 1998.
20. Pan Wei Ng Ivar Jacobson and Ian Spence. Enough process - let's do practices. *Journal of Object Technology*, 6(6):41–66, 2007.
21. Raine Kauppinen, Juha Taina, and Antti Tevanlinna. Hook and template coverage criteria for testing framework-based software product families. In *Proceedings of SPLiT - International Workshop on Testing Software Product Lines*, pages 7–12, 2004.
22. Peter Knauber and Johannes Schneider. Tracing variability from implementation to test using aspect-oriented programming. In *Proceedings of the Software Product Line Conference 2004*, 2004.
23. Dingding Lu and Robyn R. Lutz. Fault contribution trees for product families. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002.
24. Tim Mackinnon, Steve Freeman, and Philip Craig. Unit testing with mock objects. In *Proceedings of XP 2000*, 2000.
25. Paul Mathews. *Design of Experiements with Minitaqb*. ASQ Quality Press, 2004.
26. John D. McGregor. Reasoning about the testability of product line components. In *Proceedings of SPLiT - The Second Annual International Workshop on Testing Software Product Lines*, pages 1–7, 2005.
27. John D. McGregor and David A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.
28. Sun MicroSystems. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, 2004.
29. Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, pages 84–95, 2001.
30. David L. Parnas and David M. Weiss. Active design reviews: principles and practices. *Journal of Systems and Software*, 7(4), 1987.
31. Sacha Reis, Andreas Metzger, and Klaus Pohl. A reuse technique for performance testing of software product lines. In *Proceedings of SPLiT - The Third Annual International Workshop on Testing Software Product Lines*, pages 5 – 10, 2006.
32. Kurt C. Wallnau. A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, 2003.
33. Jamie J. Williams. Data driven test environment considerations when developing controls product line test architecture. In *Proceedings of SPLiT - The Third Annual International Workshop on Testing Software Product Lines*, pages 11–16, 2006.