

Verification and Validation Issues and Implications for Reuse

John D. McGregor
Dept. of Computer Science
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu

Abstract

Reuse has proved to be an elusive goal in software development organizations. Many language constructs and design principles have been introduced with the hope of creating just the appropriate abstraction that will support the creation of reusable software. None of these efforts have achieved sufficient magnitude to contribute significant cost savings. Recent industrial experience and research has shown that previous efforts have been too narrowly focused. This has led to techniques that take a comprehensive organizational approach to achieve strategic levels of reuse. These techniques can and are being applied to the verification and validation in modeling and simulation organizations.

1 Introduction

Verification and validation of modeling and simulation systems involves two separate and distinctly different levels of activity [Sargent 84]. One level addresses the model that forms the basis of a simulation and whether it is a faithful representation of reality. The second addresses the computer programs that implement the model and whether they are an accurate implementation of the specified model. This paper will focus on the second level.

Verification and validation activities are present in the product development processes of modeling and simulation systems because the construction processes are imperfect. The complexity of commercial strength systems makes it unlikely that the construction processes will be perfected anytime soon. The alternative is to make the verification and validation activities as efficient and effective as possible. There are several methods by which this can be accomplished. This paper will focus on the design and development of verification and validation assets that can be used multiple times on multiple products.

Applying reuse techniques in verification and validation activities without also applying the same techniques in the software construction activities is possible but much less effective. Coordinating reuse across both types of activities is much more effective and efficient. Likewise, once techniques are in place to support use across multiple products, it is easier to apply these techniques to a wide range of product artifacts that become assets used in future products. This paper will focus on a comprehensive and integrated approach to asset development.

“Reuse” is almost an expletive to many working to construct software products. Many efforts to achieve reuse have failed to realize even minimal benefits. Those that have, have typically only reused libraries of code including those provided by the compiler vendor or libraries of utilities created by another project in the same company, the benefit of which may not extend beyond the second product. The techniques that will be reviewed in this paper have produced orders of magnitude improvements in return on investment from the assets.

Making an asset usable in multiple products, or even across multiple versions of the same product, requires more effort than building the asset for a single specific use. This has led to investigations into how to fund the extra effort that will not be a direct benefit to the initial project. In essence, the initial project is being asked to invest resources to benefit future products. The techniques described in this paper will propose a solution to this problem.

In this paper we will use the term “test asset” to refer to any artifact that is used in the test process. This encompasses test plans, test cases, test scripts, test data, test reports and, any other document or code that supports testing. Test asset will be taken to include assets related to review and inspection processes as well as assets related to the dynamic execution of code. We will use the term “product asset” to refer to artifacts that are directly related to the system being developed. This includes the software architecture, requirements, code for the system, and any other item that is associated with the product.

Test assets will be used multiple times on the same product asset and multiple times on related, but different product assets. A test asset will be applied multiple times to the same product asset while the product asset is being developed, see Figure ITERATE. That same test asset can, in many cases, be used with the succeeding versions of the product asset, see Figure VERSIONS. In a design with varying levels of abstraction, a test asset can be used with multiple product assets. The product assets will be at a lower level of abstraction than the original target. Figure LEVELS shows a test asset that was originally targeted for Product asset A can be used with product assets B, C, and D. A test asset does not happen to be useful multiple times by accident. It requires a comprehensive strategy that is adopted at all levels of the enterprise.

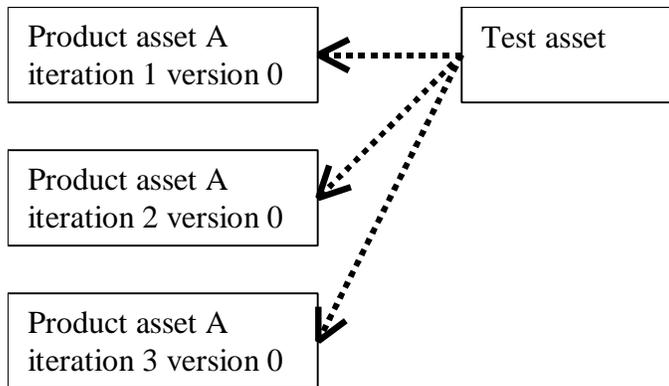


Figure ITERATE

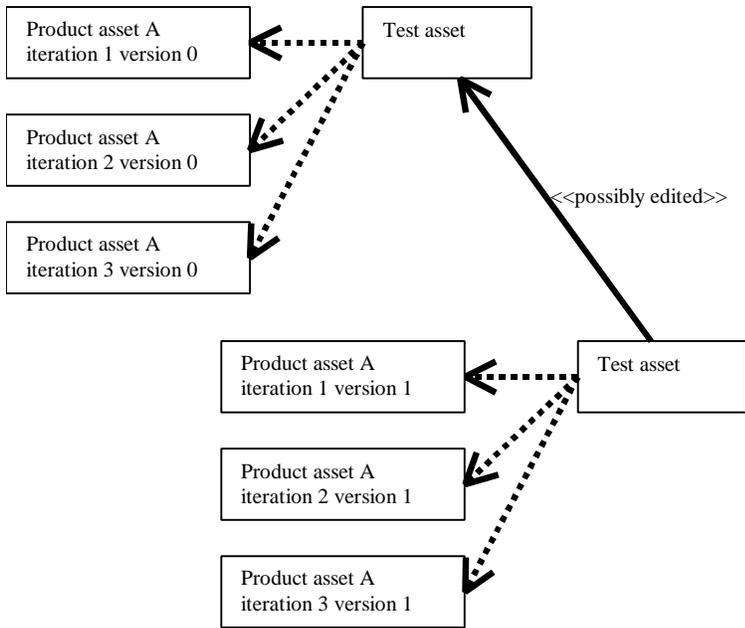


Figure VERSIONS

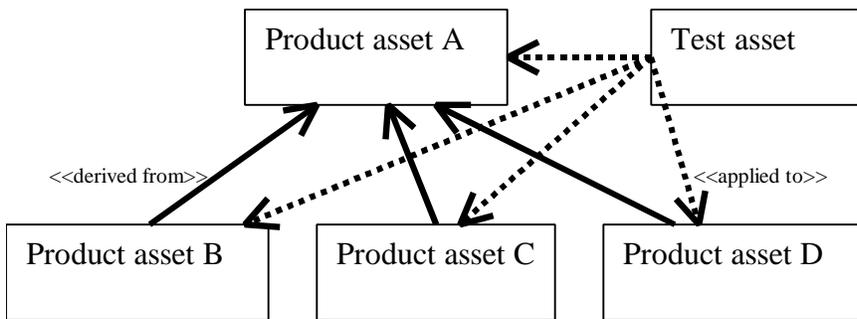


Figure LEVELS

The remainder of this paper presents a brief review of the general reuse literature in the next section followed by sections on reuse principles and their application to verification and validation of modeling and simulation systems. A more detailed history of software reuse can be found in [Biggerstaff 89].

2 History

The early history of reuse focused on the reuse of low-level technical assets, code. In the 1960s and 1970s, much reuse effort was directed at the development of code libraries. For example, LINPACK [Bunch 79] is a FORTRAN-based subroutine library used for solving systems of linear equations. The library was used and continues to be used by a number of clients. Other libraries came and went when their user base did not support the maintenance costs. Libraries illustrated the utility of encapsulating functionality in units that could easily be added to different programs. They also illustrated the problems brought on by separating functionality from data definitions.

In the 1980's and early 1990's, the object-oriented approach to software design and implementation moved the focus [McGregor 92]. Effort shifted from building libraries of isolated functions to designing structures that blended function and data into an operational unit. The definition of these units was separated from their realization by differentiating between classes and objects.

Object-oriented analysis and design methods include the concept of hierarchy by defining a generalization/specialization relationship between definitions of two design units – classes - and by defining a containment relationship between two operational units - objects. Figure HIERARCHY illustrates the Unified Modeling Language (UML) notation for denoting that two class definitions are defined by modifying a more general definition. Since test assets are usually related to product asset definitions, the containment relationship is of less interest in building test assets that can be used multiple times.

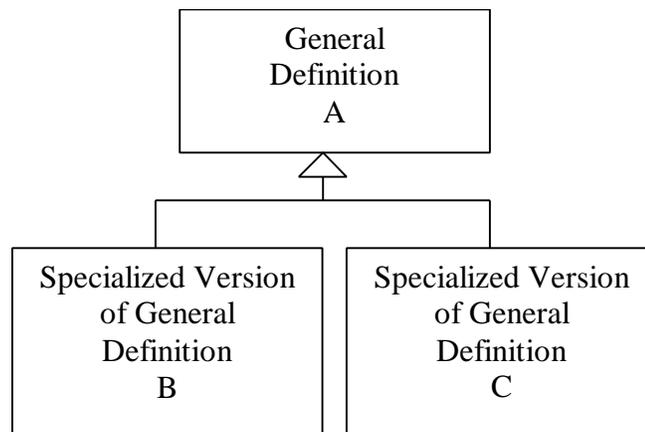


Figure HIERARCHY

During this period many [Jacobson 97, McClure 95] were pointing out that using assets in multiple products was more than just a technical issue. In fact the emphasis shifted to proposing various low-level management schemes such as repositories [Ye 01]. Repositories were intended to solve the problem of being unable to locate an existing asset but did nothing to address the issue of creating designs that facilitated the creation of test assets that were applicable in more than one place in product development. This emphasis on organizational issues became so strong that Schmidt [Schmidt 99] felt the need to point out that there were still technical issues to be addressed.

Most of the history of “reuse” has been aimed at tactical level reuse. That is the focus has been on technical and low-level management issues in the immediate project under the control of a small group. The emerging area of software product lines is focusing on strategic reuse. A software product line is “a group of products sharing a common, managed set of features that satisfy the needs of a selected market or mission area [Northrop 02].” A software product line provides a context within which multiple products can be planned simultaneously. The obvious advantage of this approach is the ability to know about the common parts of several products before the first one is built. The assets can be scoped to provide the grain size that results in the maximum applicability of the assets across the products. This will be discussed in greater detail later in this paper.

3 Assumptions

In this section we present the assumptions that constrain the work that is presented in the rest of the paper.

The greatest impact of “multiple use” of assets comes at the definition level rather than the operation level. The reuse of analysis and design-level information provides a strategic advantage by reducing the resources needed to develop the assets. In verification and validation, the significant costs are in identifying effective test scenarios or cases not in implementing those tests or applying them. Using test scripts to execute tests automatically is useful and using a ‘capture and playback’ tool is helpful as well. However, neither provides the benefit realized by using the conceptual definition of tests in multiple situations. This is especially true since the tools used from one project or product to another are often different.

Traceability between product assets and test assets facilitate the management of changes to assets. Change is inevitable. Even a minor tweak in a simulation model can require much effort to ensure that the test assets are adjusted appropriately and then reapplied. Test assets are part of a structure that links related artifacts. The links connect test cases to the appropriate requirements or component specifications and to the test scripts that implement the test case. When a change is made to a product asset, that change is propagated to the related test assets.

The software assets being produced capture and use domain expertise that is important to the success of the organization. A company, business unit of a company or development organization develops an expertise in a certain type of system and produces similar products over time, even in a software contracting environment. This expertise is captured in several ways and reused on succeeding products. The software architecture, design patterns and component implementations all capture expertise.

The V & V activities include static techniques such as inspections and reviews as well as dynamic tests. For example, at the component level, both verification and validation are performed. The developer of the component tests that the component gives correct answers and the user of the component tests that it gives the right answers.

The various verification and validation activities form a process that is independent of, but intertwined with, the product development process. The V & V activities are related due to their similar perspective even though different people often perform the activities at the different

points. Essentially, V & V activities appear at each step in the development process. Each step in the development process specifies an output. The producers of the output validate it to be certain the output is correct. The personnel who accept the output as input to the next development step verify that the output is what they expect.

The later in the product creation process that V & V begins, the less effective it will be. Each step in the product development process adds a layer of connections that depend upon the integrity of the underlying components. V & V, at the individual component level, finds many faults that are then removed before they become visible beyond the developer creating the component. If these faults are not removed at the component level, integration tests will find them and cause a much deeper repair to be necessary.

4 Principles

In this section we discuss some basic principles of creating verification and validation assets for multiple use.

4.1 Context

An asset is only usable within a certain context. Making something usable in multiple situations is a matter of making the context as all encompassing as possible. Modern languages have added constructs, such as templates, to reduce the amount of context present in a definition. Eliminating any direct dependence on the operating system, as with a virtual machine, also expands the range in which the asset is applicable.

Context is defined by a number of sources:

- ?? **Domains define context.** A specific application domain will usually define a set of standard data types. The domain will require certain levels of accuracy in answers. A simulation of a wireless telecommunications domain would define several specific protocols and other structures such as a data packet. In the wireless domain there is an important relationship between a specific protocol and a specific size of data packet. Components designed to work with one size of data packet may not work with another.
- ?? **Standards definitions define context.** A standard, much like a domain, defines standard data types, common definitions of states, and standard algorithms. The HLA defines characteristics about the Run Time Infrastructure (RTI) upon which components may be dependent.
- ?? **The operating environment defines context.** There are often system calls, link formats, and primitive data types definitions that must be compatible for use of a component to be successful. The use of a timer provided by the operating system limits the applicability of a component. There are implicit dependencies as well, such as an assumption about the scheduling policy of the operating system.

In order for a test asset to be usable across a number of systems, these systems must have sufficient context in common. This context can occur by accident or it can be planned. In section 5.1 we will discuss a planned approach.

4.2 Parallel Use

Test assets are more likely to be used for multiple systems if the product assets they are used to test are used in those same systems. Test assets such as test cases and test plans are tied to specific product assets. Every system test case is traceable to one or more system requirement. Other test assets, such as unit test plans, are traceable to assets such as components or subsystems.

Certain types of test assets, particularly test patterns, can be used across products that do not use the same requirements or code. In the case of a test pattern, the tester recognizes a test obligation that is similar to one he or she has seen before. The appropriate pattern is selected and applied, more on this in section 5.4. These assets have a broad impact that is more difficult to accurately measure; however, it is clear that the impact is far greater than that of specific test scripts. For an asset to be useable across a wide range of systems, it must be at a higher level of abstraction.

4.3 Parallel Structure

Designing tests to have the same structure as the portion of the system to be tested produces test assets with the least amount of effort possible and maintains close traceability. One particular example of this is the generalization/specialization relation between two class definitions. If class B specializes the definition of class A, the test class for B will specialize the test class of class A. This significantly reduces the effort of developing test cases and test data sets. Since the test class derived by inheritance corresponds class B, traceability is preserved and changes to B can be addressed quickly in the associated test class [McGregor TBA].

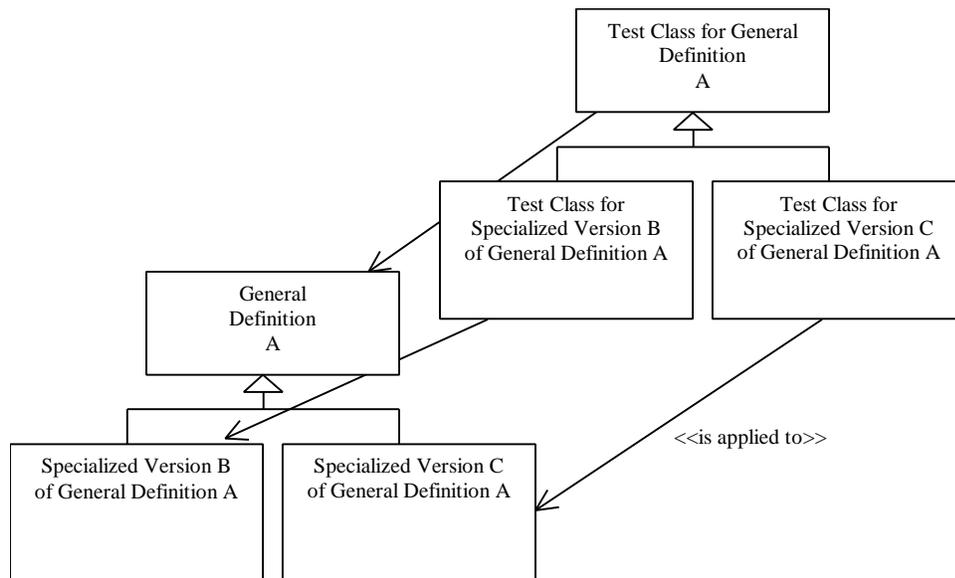


Figure PACT

A second example is the “extends” and “generalizes” relations between two use cases. When use case B extends use case A, the test plan for use case B extends the test plan for use case A. When use cases B and C specialize use case A, use cases B and C provide specific values for some attribute defined in A. Often A is not an actual requirement, it is too abstract. B and C are the actual requirements against which tests will be developed. However, test plans, cases and scripts can be defined in the same terms as use case A. The assets associated with use case A are the basis for the test assets for B and C. Tests structured this way are easier to develop and reduces the number of test that actually have to be executed [Harrold 92].

4.4 Standard Interfaces

The concept of an interface as a specification separate from implementations of that interface [Liskov 86] supports the use of the specification information across multiple implementations. Functional test cases are derived from knowledge of the specification alone. As the architecture team defines interfaces as part of the structure of the architecture, functional tests can be written against each interface. That functional test suite can be used to test every implementation of that interface.

Standards take this reuse to another level. Standards such as IEEE Standard 1516 [IEEE TBA] are often written so that they either place requirements on systems that implement the standard or they are written in terms of interface specifications. In these cases, the test assets based on the standard interfaces are reusable across all systems that adhere to the standard. Obviously there will be specific information that is more detailed than the information in the standard. The test assets for the actual system will specialize the standards-based test assets.

4.5 Abstraction, encapsulation and, information hiding

Abstraction, encapsulation and information hiding are standard devices for supporting reuse in software design.

Abstraction eliminates certain details in order to simplify an asset. For design assets it is usual to eliminate implementation details. There can be levels of abstraction in that the degree to which detail is omitted can vary. This is useful in that a person can select the definition with the appropriate amount of information, i.e. the appropriate level of abstraction. A test pattern is an abstraction of the implementation details of test information that may reside in several related components.

Information hiding reduces interaction faults and improves quality of components. It also complicates the testing process by decreasing testability. The measure of testability is the probability that tests will find faults, provided that faults exist [Voas 95]. Information hiding

makes the examination of program state during execution more difficult. This is particularly a problem for systems such as simulations where the state is an important part of correctness at any specific point in an execution. Techniques such as reflection [McGregor 01b] must be used to overcome this otherwise useful design technique.

Encapsulation provides a means of closely associating a set of assets. An object in object-oriented programming encapsulates a set of data attributes and the methods that manipulate that data. Encapsulation can enhance testability by ensuring that all the needed information is in one unit. This unit can be an object, a package, or a configuration. In section 5.2 we will discuss the use of a configuration to encapsulate the test assets that are associated with a product asset.

5 Techniques

In this section we will discuss a comprehensive approach to multiple use and a few supporting constructs and techniques. This approach defines management, development, and testing processes that support each other and that provide an environment for the development of a set of related products.

5.1 Multi-product production

A software product line is a strategy for planning and producing multiple products that share a related set of features [Clements 02]. The results of several years of industrial experience have shown product lines to be a highly effective approach that results in the use of each asset in multiple products and in order of magnitude reductions in the resources required to produce a product. A software product line organization adopts a comprehensive set of practices that span the areas of organizational management, technical management, and software engineering. These practices include review, inspection, and testing practices that emphasize the use of assets across the products in the product line.

The organization defines the scope of the product line to encompass a set of products that share a common feature set. Some of the features can be varied, for example the operating system upon which the product executes, and the sum total of the specific values chosen for each variation makes each product unique. The scoping analysis, analysis of requirements and the definition of the software architecture define the context that facilitates the creation of assets that can be used in several of the products in the product line.

Each test asset in a product line can be used on several of the products in the product line [McGregor 01a]. Examples of test assets include:

- ?? test plans and processes,
- ?? architecture evaluation scenarios,
- ?? standard component interface test assets, and
- ?? system level test assets.

Each of these assets is linked to specific product line assets using the configuration management tool. Choosing a product line asset leads directly to the appropriate test assets.

Each asset created in a product line context is guaranteed to be used in several products in the product line by virtue of the planning, scoping, and architecture processes. This does not mean that just any asset is usable across several products. Effort is still required to achieve the appropriate level of abstraction. However, the context defined by the products in the product line helps the product developer determine the correct level of abstraction.

The product line approach provides a context in which V & V assets as well as development assets are more carefully planned. By selecting this approach to product development, a company is adopting an organization-wide strategy that facilitates the use of assets in multiple products.

5.2 Reactive vs Proactive Reuse

The typical approach to reuse is reactive. An opportunity to use an asset arises when, while trying to solve a design problem, the designer remembers that a specific asset exists and selects it. When an asset is made “reusable”, the thought is that probably in the future there will be a need for that asset in some other program. However, the context of these future uses is not clear in these situations and the design is a guess on the part of the developer.

Proactive reuse is more profitable than reactive reuse and supports the approach described in section 5.1. A proactive approach includes a comprehensive planning process. Before the investment to make an asset reusable is made, there is a plan to use the asset in several programs. A closely related approach is the use of domain analysis [Prieto-Diaz 87] to identify those assets that are most likely to be used in multiple related products.

As described in section 4.2, a proactive approach, where reuse of test assets follows the use of the corresponding product assets, will result in greater reuse of the test assets associated with the reused product assets. The encapsulation principle discussed in section 4.5 provides guidance for producing test packages. These packages, like the package construct in some languages, encapsulate a product asset and the test assets that apply to that product asset.

A component test package encapsulates a product component with a test plan for the component, both functional and structural test suites, and any data sets needed to test the component. A system test package groups together a family of use cases, the test plan for that set of requirements, inspection scenarios involving those use cases, the functional tests for that aspect of the system, and test data sets. Test packages are defined in the configuration management system so that assets written in different languages as well as documents produced by a variety of tools can be encapsulated in a single unit.

5.3 Architecture

Recent developments in the representation and analysis of software architectures have led to more useful architectural models. “The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of these components, and the relationships among them” [Bass 98]. This definition takes architecture well beyond the single block-diagram style of representation.

The software architecture comprises all of the structures of the system. Even small systems will have an architectural description that is large and complex. To make the architecture a usable asset, a number of different “views” of the architecture are created [Kruchten 95]. A view of the architecture is a representation of some subset of the architecture in which some parts of the architecture are emphasized and other parts are de-emphasized or omitted. Many questions that stakeholders have about a system can be answered by creating an architectural view for that stakeholder.

The module view, which shows dependencies among interfaces, is one of the many views of the system produced in an architecture-based design process. This view provides two benefits. It defines all of the major interfaces that implement the system’s required functionality. The view also captures one type of structure for the system, the dependencies between interfaces. These dependencies represent the control and data flows of the system when the interfaces are implemented.

The High Level Architecture (HLA) is an abstract architecture for simulation reuse and interoperability [DMSO 02]. HLA has been institutionalized as IEEE Standard 1516. Beyond the development benefits of an architecture, HLA serves as a basis for establishing design and test patterns [OSC 02]. The architecture provides many of the characteristics needed to build software that can be used to test more than one system.

Product-specific architectures are derived from the HLA. Early in the development process the product-specific architecture must be validated. The Architecture Trade-off Analysis Method (ATAM) [Clements 02b] is an architecture evaluation method that utilizes a set of generic techniques that are used to specialize a set of general constructs into the specific questions to be used for evaluation. ATAM uses a standard set of quality attributes to guide the evaluation process. These qualities are prioritized for each individual process. This technique is illustrated for large-scale simulations in [Jones 01].

5.4 Design and test patterns

Gamma et al [Gamma 99] formalized the notion of reusable software design knowledge in the form of software design patterns. Their patterns were about general object-oriented design

concepts. Each pattern describes a design problem and the constraints on any solution to that problem. The pattern then presents the solution that is most often selected given those constraints.

McGregor [McGregor 01b] defined a test pattern as the canonical solution to a specific testing problem. Each test pattern is associated with a specific design pattern. When the design pattern is used in an application, the test pattern associated with it can also be applied.

Both McGregor [McGregor 01b] and Binder [Binder 99] present selected test patterns and discuss the generation and use of test patterns in general. Binder's test patterns represent testing knowledge and techniques and can be used to structure the test process. McGregor's test patterns are derived from the design patterns used in the software being tested and contribute to the identification and design of the test cases.

Experts in a domain mentally organize their knowledge into patterns. The Modeling and Simulation domain is rich in standard techniques [Asahikawa 01, Naedele 98]. Every design pattern can have one or more test patterns associated with it. The test patterns are used to structure tests anytime the corresponding design pattern has been used.

5.5 Model-based Techniques

A number of groups in various research communities have taken a "model-based" approach to system design [ECOOP 02]. The intention is to extend the pattern approach. A model is the equivalent of a coordinated set of patterns. The Object Management Group (OMG) has launched a model-driven architecture approach to software design [D'Souza 01].

6 Summary

In this paper we have presented a number of techniques that are being used in various organizations. In this section we summarize by considering how these activities can be integrated into existing test processes and describe some of the work that needs to be done to advance the state of practice.

6.1 Putting it all together

The intention is to list specific activities that are derived from the techniques previously presented. These activities do not form a comprehensive process. They should be integrated at the appropriate places in the current development process.

6.1.1 Across Products

Some of the activities are independent of a specific product but dependent on the context of development. In a software product line organization the product line team handles these tasks. Non-product line organizations will often have either a “Forward looking” technology team or a process team that can handle these tasks.

6.1.1.1 Document test patterns

Companies should document test patterns that prove useful in the company’s specific context. The set of tools, the domain, and the development process all shape these patterns. Some patterns will be more generic than others and personnel in different companies but working in a common domain are often able to share patterns.

6.1.1.2 Develop an integrated development and testing process

Personnel who have not participated in the development of a product are used for some phases of V & V of that product to ensure objectivity. This does not mean that the processes that guide the two groups have to be separate. A single integrated, or at least coordinated, process results in more opportunities for better communication and for recognition of inter-relationships between the product and V & V assets.

6.1.1.3 Develop standard test cases

The HLA can serve as the source of test cases for each of its interfaces. In fact, the Level One Test Procedures, Version 1 [DMSO 02] provide a start in this activity. Within a company, which has a specific language, tool set and development process, each of these test procedures can be made more specific to the local environment and used across multiple simulations.

6.1.2 Individual Product

These activities are specific to one product because they relate to the unique combination of requirements selected for that product. In a software product line organization the product-specific teams handle these tasks. Non-product line organizations are usually organized around projects. A project team would handle these tasks.

6.1.2.1 Develop a testing architecture view

The test view of the architecture identifies all those interfaces and services within interfaces that are particularly of use during some level of testing. For example, a component may have 30 services in its public interface but only 5 of them are used for tasks such as putting the component in a specific state before executing a test or for dumping the state of the component to evaluate the effect of executing the test case.

The creation of the test view aids in the development of testware rather than the development of test cases. It allows the V & V personnel to participate in system development at a very early stage.

6.1.2.2 Develop test packages

The integrated product development process should guide personnel to encapsulate product artifacts and V & V artifacts in packages. The package may be a language construct, as in Java, or the configuration management tool may be used to associate the members of a package. V & V personnel support their activities by using packages to trace the effect of change in product artifacts on the testing artifacts. The packages are transparent to the product developer since their build scripts select the development artifacts from the packages to assemble the complete product.

6.2 Future Work

There are many areas that can benefit from additional research and development.

?? The HLA should be exploited more fully as a source of abstractions. Patterns and generic test cases should be developed and made available to the modeling and simulation community.

?? Companies should also develop domain-specific test patterns that fit their models.

?? Companies, which expect to build more than three products before they go out of business, should investigate the applicability of software product lines to their domain and market.

References

- [Asahikawa 01]** Asahikawa, Toyoaki Tomura. Developing Simulation Models of Open Distributed Control System Using Object-Oriented Structural and Behavioral Patterns, <http://minf.coin.eng.hokudai.ac.jp/members/kanai/ISORC2001.pdf>.
- [Bass 98]** Bass, Len, Clements, Paul, and Kazman, Rick. Software Architecture evaluation [TBA]

ecture in Practice, Addison-Wesley, 1998.
- [Batory 97]** Batory, Don. Intelligent Components and Software Generators, Technical Report 97-06, Department of Computer Sciences, University of Texas at Austin, February 1997.
- [Biggerstaff 89]** Biggerstaff, Ted J. Software Reusability: Applications and Experience, ACM Press Frontier Series, 1989.
- [Binder 99]** Binder, Robert V. Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.
- [Boehm 81]** Boehm, B. W. Software Engineering Economics, Englewood Cliffs NJ, Prentice-Hall, 1981.
- [Bunch 79]** Bunch, J.; Dongarra, J.; Moler, C.; and Stewart, G.W. LINPACK User's Guide, STAM, Philadelphia, PA, 1979.
- [Clements 02a]** Clements, P. & Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
- [Clements 02b]** Clements, P. , Kazman, R., and Klein, M. Evaluating Software Architectures: Methods and Case Studies, Addison-Wesley, 2002.
- [Cohen 99]** Cohen, Sholom. Guidelines for Developing a Product Line Concept

of Operations, Software Engineering Institute, 99-TR-008, 1999.

- [DMSO 02]** Defense Modeling and Simulation Office,
<https://www.dmsomil/public/transition/hla/>
- [DeLano 97]** DeLano, D. and Rising, L. A Pattern Language for System Test in Pattern Languages of Program Design 3, Reading, MA, Addison-Wesley, 1997, R. Martin, D. Riehle, and F. Buschmann, eds, pp. 503–525.
- [D’Souza 01]** D’Souza, Desmond. Model-Driven Architecture and Integration,
www.kinetium.com. <http://www.omg.org/mda/presentations.htm>
- [ECOOP 02]** ECOOP 2002 Research Workshop, Model-based Software Reuse, 2002.
<http://research.intershop.com/workshops/ECOOP2002/papers.shtml>
- [Flater 01]** Flater, David. Impact of Model-Driven Standards, National Institute of Standards and Technology.
<http://www.omg.org/mda/presentations.htm>
- [Gamma 95]** Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. Design Patterns, Addison-Wesley, 1995.
- [Harrold 92]** Harrold, Mary Jean and McGregor, John D. Incremental Testing of Object-Oriented Class Structures, Proceedings of the Fourteenth International Conference on Software Engineering, 1992.
- [Jacobson 97]** Jacobson, Ivar, Griss, Martin, and Jonsson, P. Software Reuse: Architecture, Process, and Organization for Business Success, Addison-Wesley-Longman, 1997.
- [Jones 01]** Jones, Lawrence G. and Lattanze, Anthony J. Using the Architecture Tradeoff Analysis Method to Evaluate a Wargame Simulation System: A Case Study, CMU/SEI-2001-TN-022, Software Engineering Institute, 2001.
- [Kruchten 95]** Kruchten, Phillippe. The 4+1 Architectural View model of Architecture, IEEE Software, 12(6), November 1995, pp. 42 – 50.

- [Liskov 86]** Liskov, Barbara. Abstraction and Specification in Program Development, McGraw Hill, 1986.
- [MacKinnon 00]** MacKinnon, Tim; Freeman, Steve; and Craig, Phillip. Endo-Testing: Unit Testing with mock Objects, extreme Programming and Flexible Processes – XP2000, 2000.
- [McClure 95]** McClure, Carma. Model-Driven Software Reuse Practicing Reuse Information Engineering Style, Extended Intelligence, Inc., 1995.
<http://www.reusability.com/papers2.html>
- [McGregor 92]** McGregor, John D. and Sykes, David A. Object-Oriented Software: Engineering Software for Reuse, International Thomson Computer Press, 1992.
- [McGregor 01a]** McGregor, John D. Testing Software Product Lines, CMU/SEI-2001-TR-022, Carnegie Mellon University, 2001.
- [McGregor 01b]** McGregor, John D. and Sykes, David A. A Practical Guide to Testing Object-Oriented Software, Addison-Wesley, 2001.
- [Naedele 98]** Naedele, M. and Janneck, J. Design patterns in Petri net system modeling, Proceedings of ICECCs'98, 1998, pp. 47-54.
- [OSC 02]** Object Services and Consulting, Inc. Composition and Federation Patterns in Componentware Software Architectures,
<http://www.objs.com/ddb/federation.html>.
- [O’Ryan 01]** O’Ryan, Carlos; Schmidt, Douglas C.; and Noseworthy, J. Russell. Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations, International Journal of Systems Science and Engineering, CRL Publishing, 2001.
- [Parnas 76]** Parnas, David L., “On the Design and Development of Program Families”, IEEE Transactions on Software Engineering, v SE2, n1, March 1976, pp. 1 –9.
- [Prieto-Diaz 87]** Prieto-Diaz, Reuben. Domain Analysis for Reusability, In Proceedings of COMPSAC 87, October 1987.
- [Redondo 02]** Redondo, Rebeca P.; Arias, Jose J. Pazos; Vilas, Ana Fernandez; and Martinez, Delen Barragans. ARIFES: Reusing Formal Verification Efforts in a Requirements Specifications

Stage, Model-based Software Reuse: ECOOP 2002, 2002.
<http://research.intershop.com/workshops/ECOOP2002/papers.shtml>

[Russ 00]

Russ, Melissa L. & McGregor, John D. A Software Development Process for Small Projects, IEEE Software, Sept/Oct 2000.

[Sargent 84]

Sargent, Robert G. Simulation Model Validation in Simulation and Model-Based Methodologies: An Integrative View, edited by T. I. Oren, B.P. Zeigler and M.S. Elzas, Springer-Verlag, 1984.

[Schmidt 99]

Schmidt, Douglas C. Why Software Reuse has Failed and How to Make It Work for You, C++ Report, January 1999.

[Voas 95]

Voas, Jeffrey M. and Friedman, M. Software Assessment: Reliability, Safety, Testability, John Wiley and Sons, 1995.

[Weiss 99]

Weiss, David M.; Tau, Chi; & Lai, Robert. *Software Product-Line Engineering*. Reading, MA: Addison-Wesley, 1999.

[Ye 01]

Ye, Yunwen. An Active and Adaptive Reuse Repository System, Proceedings of the 34th Hawaii International Conference on System Sciences, IEEE Press, 2001.