JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 5, No. 6, July - August 2006

A Classification Framework for Software Reuse

Vitaly Khusidman and **David M. Bridgeland,** Unisys Corporation

Abstract

Software reuse is commonly used to leverage existing assets and to reduce development cost and time. Reuse can be accomplished by several different mechanisms. This paper describes these mechanisms and proposes a classification framework for them. The framework has two dimensions: retest scope—how the reuse impacts the need for testing—and binding time—when the reuse is realized. By examining these two dimensions, we define a matrix of reuse scenarios. The reuse scenarios in this matrix show different characteristics of flexibility and ease of maintenance. Based on this classification the paper recommends using different mechanisms to accomplish reuse for short-lifecycle single solutions, typical business applications and productized (COTS) solutions.

1 INTRODUCTION

Software is often reused. This section describes why software is reused, and describes some commonalities and variations in the way reuse is practiced.

Why is software reused?

Different applications often have similar requirements. Rather than create entirely separate solutions for each application, software companies (and other software development organizations) use the similarity of requirements to save money. A single base solution is created to support multiple applications. Less code is created and less code is maintained. Similarity of requirements makes possible a software product line [Clements 02], where the majority of the requirements remain invariant across different products in the product line.

Our notion of reuse is broad. It includes both the formal reuse of object code that does not require any customization, the opportunistic cut-and-paste reuse achieved by using and modifying fragments of existing solutions, and everything between these two extreme cases.

Cite this article as follows: Vitaly Khusidman, David M. Bridgeland: “A Classification Framework for Software Reuse”, in *Journal of Object Technology*, vol. 5, no. 6, July -August 2006, pp. 43-61 http://www.jot.fm/issues/issue_2006_07/article1

Opportunistic cut-and-paste reuse is very flexible and inexpensive in design but extremely costly in later maintenance, as the different codebases are maintained separately (see Figure 1). Formal reuse of object code is inexpensive to maintain, but expensive in design and (often prohibitively) inflexible, as shown in Figure 1. Between these two extremes are a range of reuse scenarios, addressed in this paper. We present an organizing framework for the solutions between these two extremes, and develop some criteria for choosing a solution in the framework, to provide the best compromise between initial flexibility and the ease of maintenance.

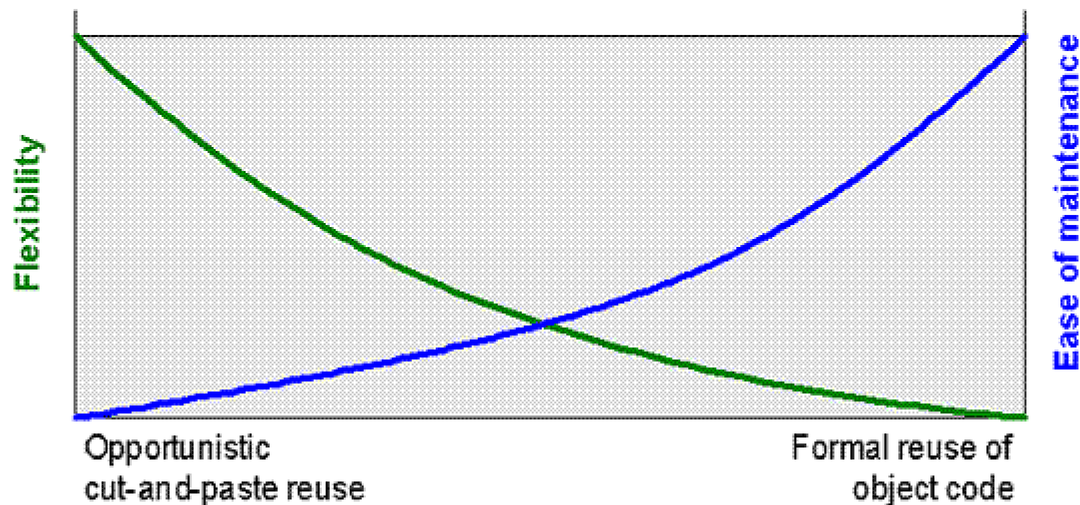


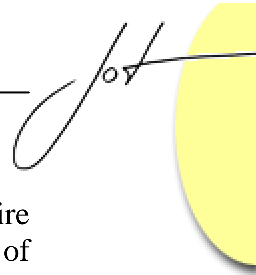
Figure 1 - A spectrum of reuse

Cloning

Cut-and-paste reuse is sometimes called software cloning. Cloning is the technique of copying logic in the existing solution, and modifying to suit the needs of the new application. When software is cloned, there are two copies, the unmodified original, and the modified clone. Both copies must be maintained, as if they were completely different rather than largely the same. This increases the cost of testing, increases the cost of maintenance, and complicates product tracking and management. Since most of the benefits of reuse are in the lower cost of testing and maintenance, cloning sacrifices most of the economic benefits of reuse [Dikel 97].

Why is software cloned? Kane identifies several forces that lead to cloning [Kane 97]:

- Developers have access to the source code. Cloning is only possible when developers can access and modify source code.
- Cloning is expedient. It is often easier to change source by "copy and modify" (i.e. by creating a clone) rather than by extending or generalizing the original source.



- Cloning requires less coordination. Generalizing existing components may require coordination not just with the component supplier, but also with other users of that component.

In an extended case against cloning, Chilton [Chilton 03] explains that cloning is both a natural developer reaction to business needs, and a productivity drain on development organizations.

Cloning has a serious impact on the economics of reuse by significantly increasing maintenance cost. This paper proposes a reuse classification framework that helps predict when software risks cloning. Our framework can also help an architect select an appropriate alternative mechanism with less risk of cloning.

2 A CLASSIFICATION FRAMEWORK

Software reuse is achieved by applying variability mechanisms. David M. Weiss and Chi Lai define variability in product family development as the ways in which members of a product family differ from each other [Weiss 99]. Jacobson [Jacobson 97] and Clements and Northrop [Clements 02] discuss variability mechanisms and their role in software reuse.

Our framework classifies variability mechanisms according to their potential to cause software cloning. This classification framework is two-dimensional. The first dimension is testing scope: what needs to be tested—or tested again—when the reuse is practiced? Does the new application need to be tested? What about the base solution on which the new application is based; does it need to be retested?

The second dimension is binding time: when is the variability realized? Is a new application generated at compile time, or does reuse customization happen via runtime configuration settings?

We claim that all reuse fits in this framework, that every reuse customization done since software started as a commercial endeavor fits somewhere in the 2-space of our framework. In particular, all the variability mechanisms identified in [Jacobson97] and [Clements 02] map to the cells of this classification framework.

But our intent in this paper is not to provide an exhaustive analysis of all possible variability mechanisms and map them to cells. Instead our intent is to explain our framework. To that end, we have mapped some of most significant variability mechanisms to illustrate the framework and show how the mapping can be done. The framework will help to perform predictive analysis and to select appropriate variability mechanisms to achieve the desired business results.

Classifications are only useful when they further and simplify analysis. Our reuse classification supports an analysis of cloning. Cloning can only be practiced on some cells of this framework. Other cells are immune to cloning.

Testing scope

How can a single base solution support a variety of different application requirements? This question is one of the central software engineering productivity themes of the last fifty years. Each application should not need to reimplement all the functionality it needs, and instead should be able to leverage the work of others. But what is the mechanism of that leverage?

Software engineering practitioners have developed many variability mechanisms to answer this question. Shared libraries are one mechanism, with different FORTRAN programs (for example) all using a common library for solving simultaneous linear equations. The shared library solves a common problem, and thousands of independently developed applications link to that library to access its base solution functionality.

Class inheritance is another very different answer to the same productivity question of supporting different applications from a common base solution. A base solution is implemented as a class, and individual applications use subclasses that inherit the base solution's functionality, using some of the methods as they are, and overriding others.

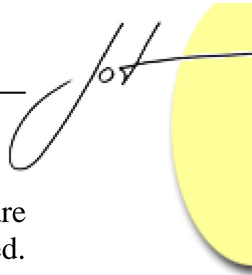
Configurable components are yet another answer to the same question. A general purpose component has predefined parameters allowing it to be configured to meet the anticipated needs of its use. When creating an application, a software developer (or alternatively, a functional expert) configures the parameters. SAP, Siebel, and other large enterprise applications take this approach for solving the software productivity problem.

There are other variability mechanisms as well, other ways of providing base solution functionality for an application: application frameworks, use case inheritance, generation of code for a new platform from a common base solution specification, and so on. We expect more variability mechanisms to be developed in the future, as software engineering continues to improve.

A key question for any of these mechanisms involves testing: what needs to be tested? If a developer creates new application `A` using base solution `B`, does she need to test just `A`, both `A` and `B`, or neither `A` nor `B`? For example, if she creates an application that links to a shared library, the application must be tested, but the library will not be broken by this linking, and need not be retested.

On the other hand, class inheritance often breaks the base solution. As noted by many, and explained best by Robert C. Martin [Martin02], inheritance is dangerous. If new application class `A` inherits from an existing base solution class `B`, `B`'s methods may not work when instances of `A` are passed instead as if they are `B`s. `A` must be tested, and `B` must be retested.

Runtime configuration provides examples of the third testing situation. A properly tested base solution `B` can be configured to create an application `A` without retesting `B` or even testing `A`. The testing has already been done, and no configuration of the predefined configuration parameters will lead to an application outside the safely tested scope.



Shared libraries, class inheritance, and runtime configurable components are examples of the only three possible answers to the question of what needs to be tested. Either

a) Both the new application and the base solution need to be tested, or
b) Only the new application needs to be tested, or
c) Neither the new application nor the base solution need testing.

A medical analogy provides handy names for these three situations. If both the new application and the base solution need to be tested, we call the relationship between them *infectious*: the new application can infect the base solution with bugs. If only the new application need be tested, we call the relationship between them *quarantined*: any problems are confined to the new application. And if neither need be tested, we call the relationship between them *immune*.

Note that we are not necessarily deriding the infectious relationship, or advocating the immune relationship. After all, infectiousness can be a good thing: a smile can be infectious, as can a laugh. Rather we are classifying existing and future reuse mechanisms by their implication for functional testing.

Every reuse mechanism fits one of these three testing scopes: it is either infectious, quarantined, or immune. The fourth possible relationship – that the new application need not be tested but the base solution must be – does not make sense.

Figure 2 shows two examples of infectious relationships. On the left is traditional OO class inheritance. In general when `Derived` class is created as a subclass of `Base`, it must be tested, and `Base` must be retested, to be sure that it works given the existence of `Derived`.

There is a special case in which `Base` need not be retested when `Derived` is subclassed from it. If the relationship between them carefully adheres to the Liskov Substitution Principle—if `Derived` is not just a subclass but a subtype of `Base`—then it need not be tested. In fact the relationship between them in that situation is immune, not infectious. Discussion of the Liskov Substitution Principle is beyond the scope of this paper; interested readers should refer to [Liskov 88] or [Martin 96].

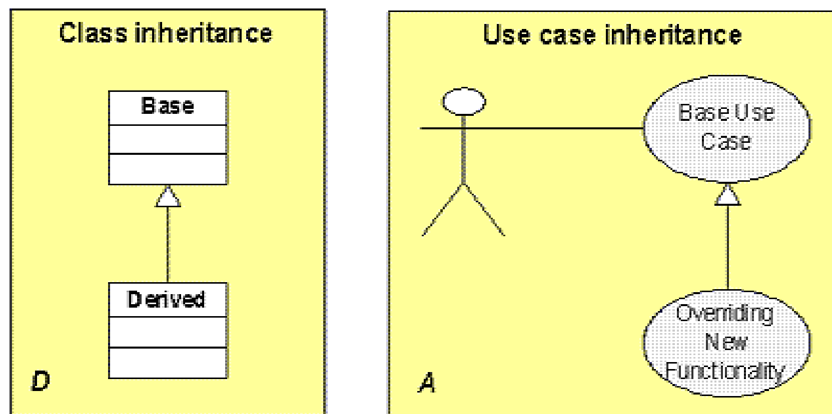


Figure 2 - Two infectious relationships

On the right of Figure 2 is another example of an infectious relationship, of use cases (requirements) instead of classes. The new use case `Overriding New Functionality` has a generalization, the existing `Base Use Case`. (This relationship was once called the uses relationship in the UML literature.) In both examples the existing functionality can be overridden by new functionality and, therefore, must be retested.

Figure 3 shows an example of a quarantined relationship between base solution components and a new component. One base solution component provides a realization of `Interface 1`. The new component `New Integrating Functionality` depends on that interface, and uses the base component. It also uses a second base solution component `Base Realization 2` in the same manner. The presence of the new component does not invalidate the correctness of the base solution: any possible errors or modified behavior introduced by the new component are quarantined there. The original base solution is not modified, overridden, or in any way changed.

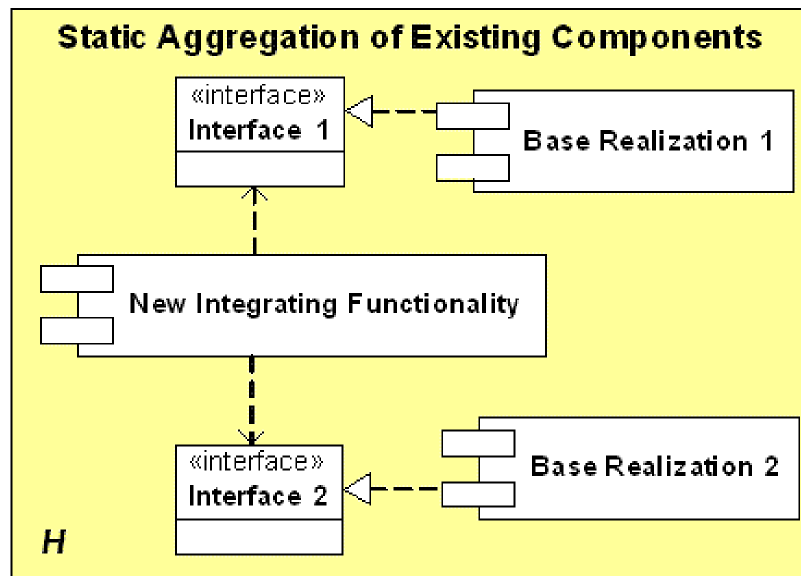
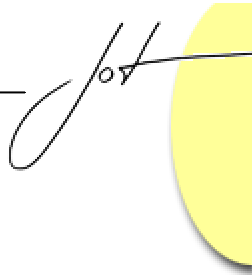


Figure 3 - A quarantined relationship

Figure 4 shows an example of the immune relationship. A base solution aggregation of classes is parameterized to create a new application. Since this aggregation (`Predefined Integration Functionality`) has already been tested— presumably with a range of possible parameters—new testing is not required. The entire configured solution is immune to errors.



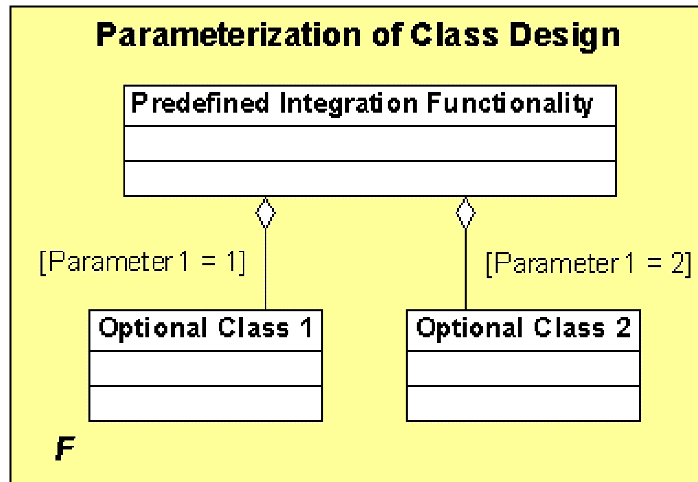


Figure 4 - An immune relationship

In practice, combinations of these three relationships are used to create a particular application. So a new application may parameterize a base solution in some places, use a base solution interface in others, and subclass from base solution classes in yet other places.

Binding time

The testing scope defines what needs to be tested when a new application is derived from a base solution. But when does that derivation happen? At what stage in the software development lifecycle does the new application start to emerge? Creating a new application at requirements time means that there is a common requirements model, a single use case model (for example) that includes both the common requirements and the space for the requirements for each of the individual applications. The design or source code that implements the individual applications may or may not be shared. Applying a variability mechanism at runtime means that a single running executable is customized to different requirements, for example by setting parameters.

We see four distinct binding time options:

- **Requirements time.** The reuse customization takes place during development of the use case model and specifications.
- **Design time.** The reuse customization takes place during development of the design model and the code. At this point source code can be added or modified.
- **Implementation time**. The reuse customization takes place during source compilation or assembly of the system from implementation components. At that point source code is not changed and all customization is limited to selection and integration of pre-designed object code components. That includes any static binding performed during application deployment.
- **Run time.** The reuse customization happens at runtime, while the system is executing. Customizations either take effect immediately—e.g. on the next

transaction—or after system reboot or component reload. Any dynamic binding performed during application deployment is also categorized as runtime customization.

Note that the binding time is when the actual customization takes place, not when the decision to customize happens. A software architect may decide to apply a runtime customization while the software is designed. She may even develop the design (at design time) in ways that make that runtime customization effective. It is still a runtime customization. What matters is when the actual binding takes place.

We already examined one example of requirements time reuse, the infectious use case inheritance on the right side of Figure 2. Figure 5 shows another example of requirements time customization. In Figure 5 a use case from the base solution has parameters, and these parameters are configured to create three new use cases.

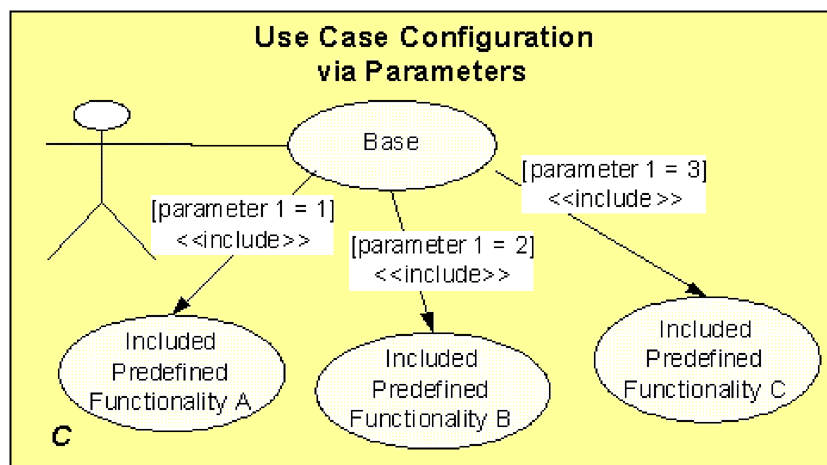
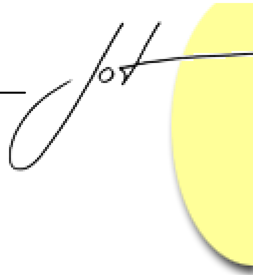


Figure 5 – Immune relationship at requirements time

We have already seen two examples of design time reuse. On the left side of Figure 2, we saw infectious class inheritance, at design time. And in Figure 4, we saw immune class parameterization, also at design time. Figure 6 shows another example. The `New Integrating Functionality` class aggregates two classes from the base solution. This `New Integrating Functionality` class is glue code, an example of the Façade pattern.



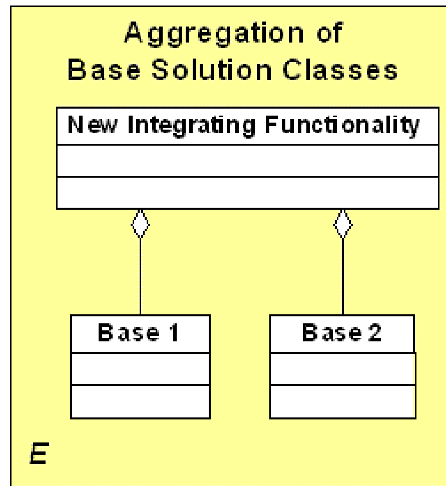


Figure 6 - Quarantined relationship at design time

In Figure 3, we explained an implementation time reuse example, interface realization by components that served to quarantine any errors to the new component being implemented. Figure 7 shows another implementation time reuse example. The `Base Realization` component provides one realization of the interface. The `New Realization` component provides a second, alternative realization. If at least one of `Base Client`'s operations uses the result of execution of the interface realization operation then that `Base Client` requires retesting.

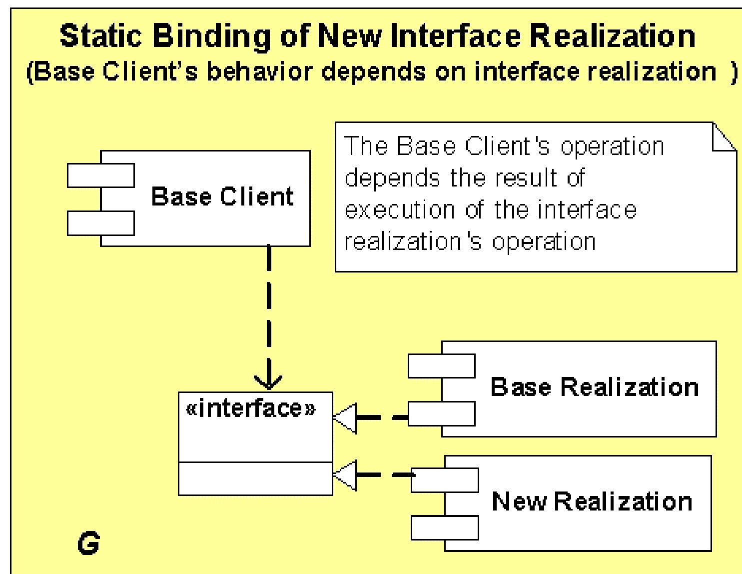


Figure 7 - Infectious relationship at implementation time

The fourth binding time is runtime. Figure 8 shows an example of runtime binding, a runtime reuse customization. A new application incorporates two existing components—`Base 1` and `Base 2`—by the means of the `New Integrating Functionality` component, a glue component realizing the Façade pattern. This example also uses the Service Locator pattern described by Martin Fowler [Fowler 04].

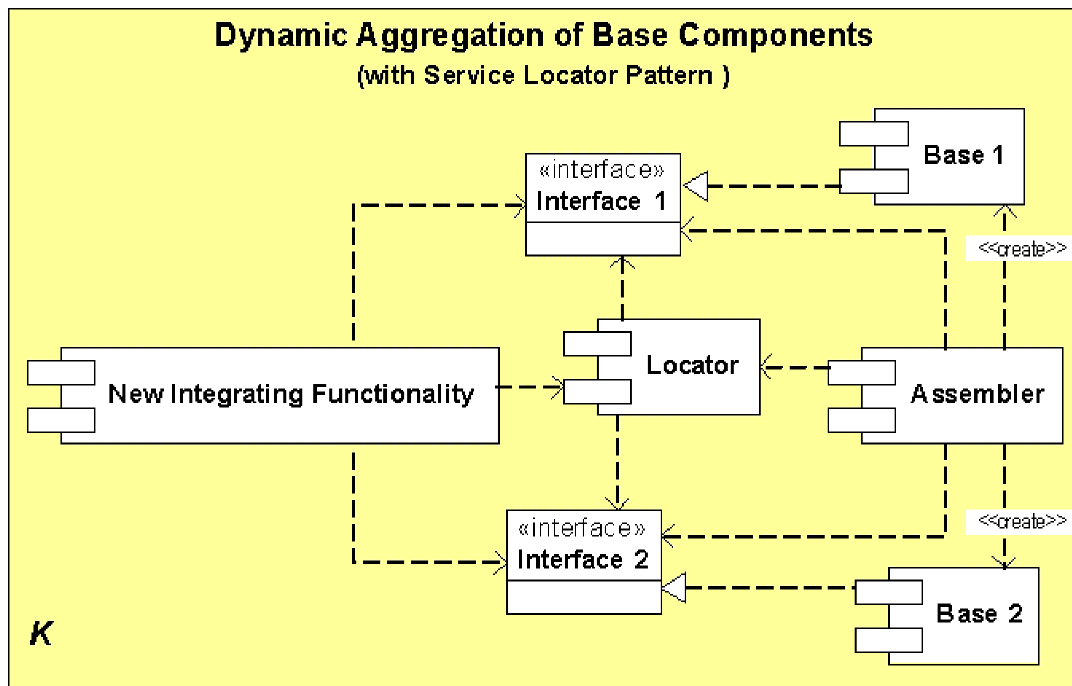


Figure 8 - Quarantined relationship at runtime

Figure 9 shows a different situation: a `New Realization` component replaces the `Base Realization` component. The Base Client need not be re-tested because it does not depend on the interface realization. This example also uses the Service Locator pattern.

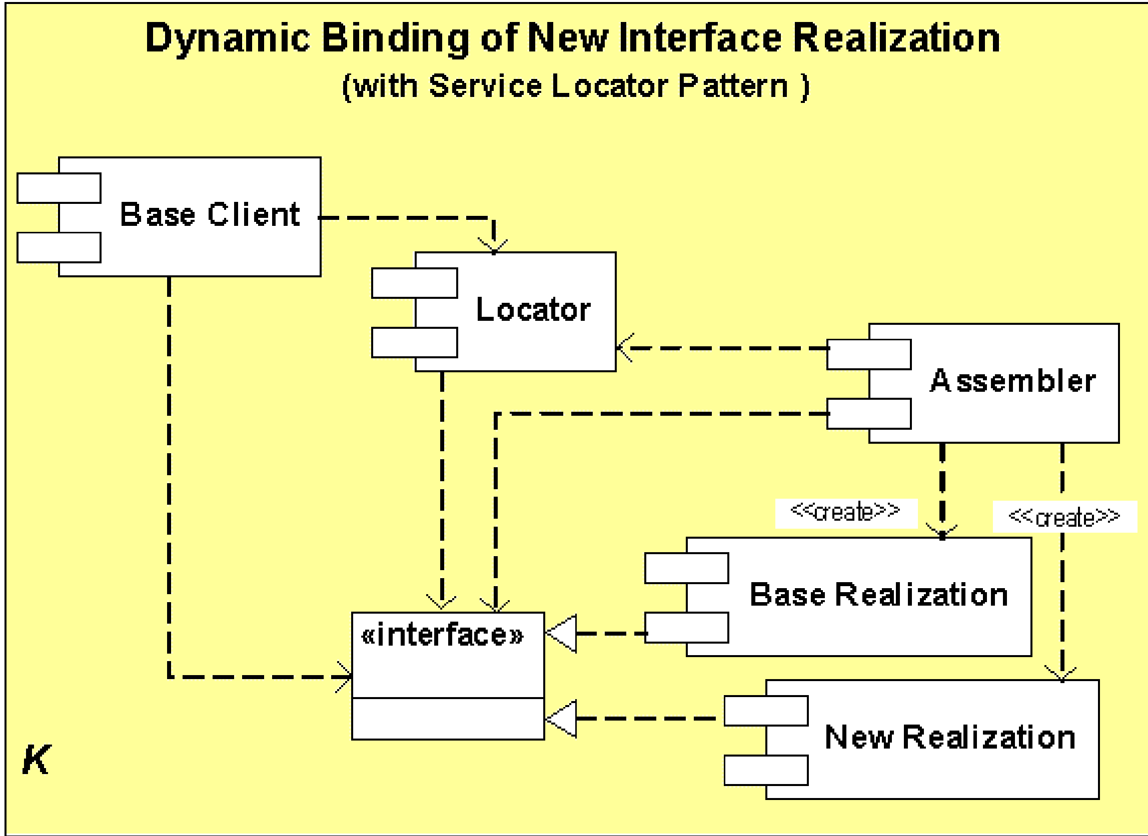


Figure 9 - More quarantine relationships at runtime

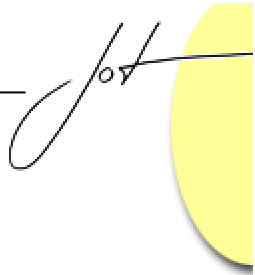
A framework of reuse scenarios

Our two dimensions – testing scope and binding time – are truly independent. All twelve combinations are possible, and in fact are practiced every day by software professionals. Table 1 describes all twelve combinations.

	Infectious **(Both base and new must be tested)**	**Quarantined** **(New must be tested. Base remains correct)**	**Immune** **(Neither base nor new need be tested)**
Requirements time	**A**. Either specialization or include (with dependency on an included use case) in use case models	**B**. Either include (without dependency on an included use case) or extends in use case models	**C**. Parameterized use case models, or the application of an analysis pattern
Design time	**D**. Either specialization or extension (with dependency on an extending class) of existing design classes	**E**. Either aggregation or extension (without dependency on an extending class) of existing design classes	**F**. Parameterized design models, or applying a design pattern
Implementation time	**G**. Static binding of a new interface realization, e.g. new operation implementation (which affects behavior of other operation of the base component)	**H**. Static aggregation of existing components, or static extension of component object code by adding new logic (which does not affect behavior of any operation of the base component)	**I**. Static binding based on parameter value, or based on template instance
Run time	**J**. Dynamic binding of a new interface realization, e.g. new EJB implementation	**K**. Dynamic aggregation of components using runtime component management patterns, e.g. Service Locator pattern, Web Services, ORB, etc.	**L**. Dynamic binding based on the parameter value, or based on template instance

Table 1 - Two dimensional classification of reuse

Examples have already been given of eight of the 12 combinations as part of the descriptions of the testing scope and the binding times. Combinations A, C, D, E, F, G, H, and K were shown in Figure 2 through Figure 9. Examples of the other four are shown below. Figure 10 shows an example of infectiousness at runtime, in which a `Base Client` component creates and uses the `Base Realization` component. Then the `Base Realization` component is replaced with the New Realization component.



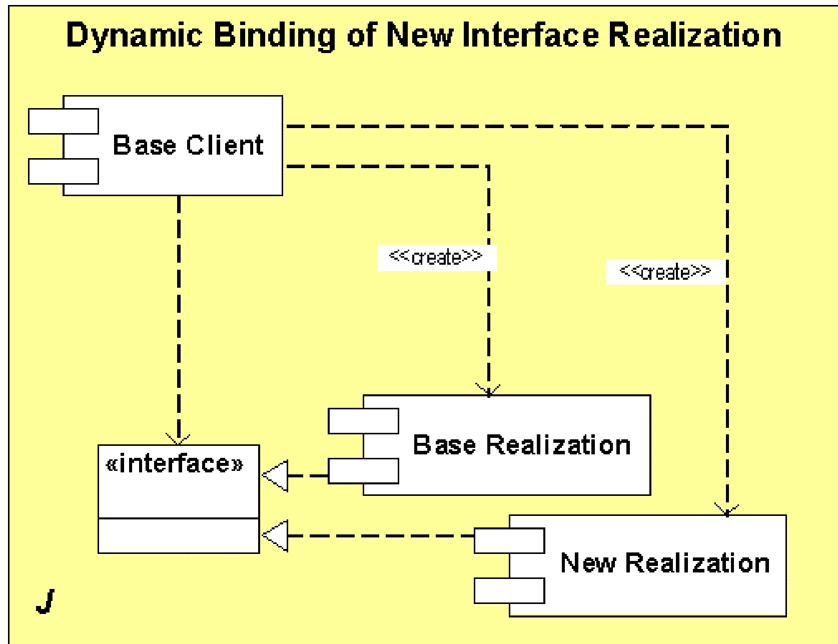


Figure 10 - Infectiousness at runtime

Figure 11 shows two examples of quarantined requirements time reuse. On the left is a simple use case extension. The new functionality adds new behavior to the existing use case. On the right is a use case inclusion, where the base use case includes behavior already isolated out in a separate use case.

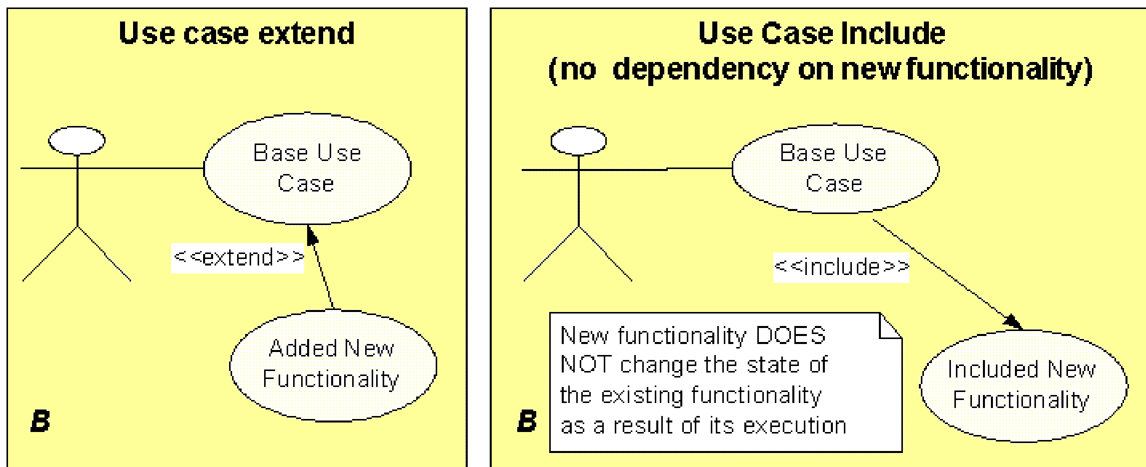


Figure 11 - Quarantined at requirements time

Figure 12 shows two final combinations. On the left is immunity at implementation time via parameterizing of existing components. On the right is immunity at runtime

configuration, also via component parameterization. The example on the right side uses the Service Locator pattern.

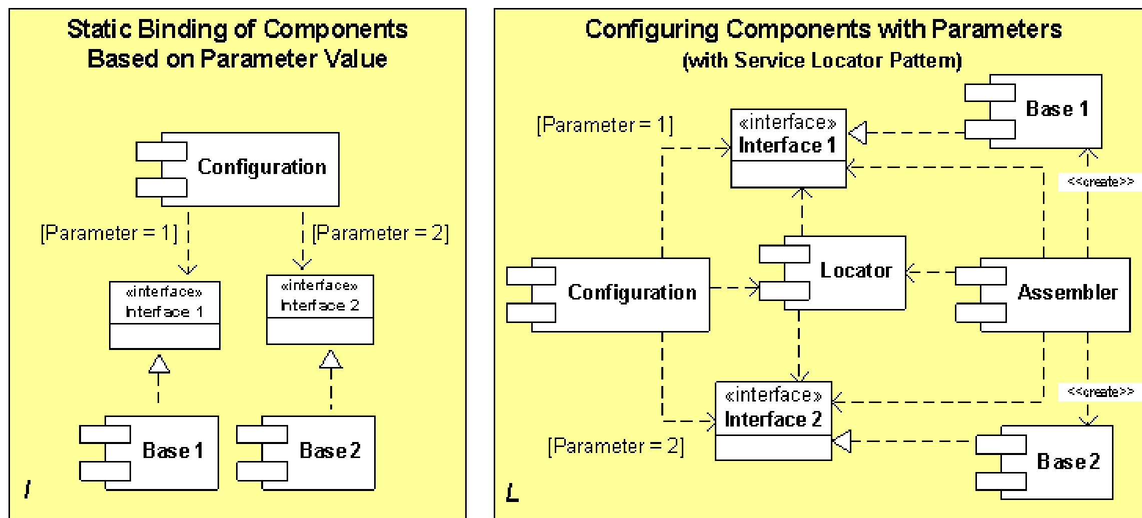
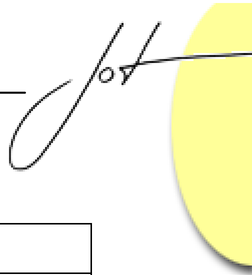


Figure 12 - Immunity at implementation time and at runtime

Reuse classification and variability mechanisms

Our classification of reuse scenarios by testing scope and binding time is a bit novel. Traditionally reuse has instead been classified by variability mechanism: how is the variability between applications accomplished? In other words: what does a software professional have to do to perform reuse? For example, David M. Weiss and Chi Lai define variability in product family development as the ways in which members of a product family differ from each other [Weiss 99]. And I. Jacobson, at al [Jacobson 97] offer a classification of variability mechanisms, as do P. Clements and L. Northrop [Clements 02].

Table 2 shows how some of the most common variability mechanisms—inheritance, extension, aggregation, inheritance realization—fit within our reuse classification framework. Some mechanisms like use case inheritance fit cleanly in a single cell. Others like interface inheritance spread across multiple cells.



	Infectious	Quarantined	Immune
Requirements time	Use Case Inheritance	Use Case Include	Use Case Extend
Design Time	Class Inheritance	Class aggregation/delegation	Parameterization Template Instantiation Generation
Implementation time		Interface Inheritance /Realization	
Run Time		Runtime Component Management Patterns	

Table 2 - Variability mechanisms within the classification framework

How customization affects cloning and flexibility

All infectious customization—everything in the first column in Table 1 or Table 2—is prone to cloning. Infectious reuse scenarios always cause tight coupling between the base and the new code. Furthermore infectious reuse scenarios assume that the object code for software artifact is modified. That causes branching of the software versions. As a result, the maintenance team deals with multiple baselines of software artifacts. Even worse: different artifacts branch at different time so maintaining traceability becomes very complex and costly. It is extremely difficult to customize via inheritance and maintain a single codebase.

By contrast, all types of immune customization—the third column in Table 2—are resistant to cloning because all customization decision points are predefined and the same baseline of software artifacts is used without regard to what configuration option is chosen.

But that resistance comes at a cost. The cost of the resistance to cloning that immunity provides is a lack of flexibility. A configurable package only supports the behaviors that are possible with the configuration parameters. Inheritance provides great flexibility; if you do not like the base solution behavior, create a subclass with the behavior you want. But with configuration, all decision points are predefined.

The middle column—the quarantined solutions—present a happy medium between the two extremes without the flaws of either.

Techniques for implementing quarantined solutions

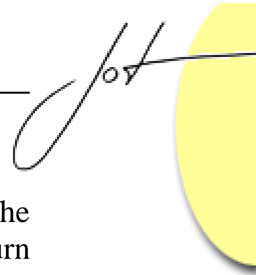
There are several techniques for implementing the quarantine testing scope. The choice of technique depends on the binding time when the extension is realized.

A requirements time quarantine—cell B in Table 1—is typically implemented with the use case `<<extend>>` association, as shown on the left side of Figure 11. One use case adds some additional functionality to an existing use case in the base solution. The `<<include>>` association among use cases can be an alternative if the include association is used in a quarantined manner. A quarantined include does not change the state of the base use case, and so the behavior of the base is unchanged. When control is returned to the base use case, the same behavior occurs in the rest of the base as if the included use case had never been executed.

Design time quarantine—cell E in Table 1—is often desirable, but traditionally has been difficult to achieve. Until recently there was no easy way to accomplish that as an extension, other than refactoring the whole against the new needs. But aspect-oriented approach promises to solve that problem. Jacobson, in his visionary paper [Jacobson 03] cites aspect-oriented software development as the missing link in modern software engineering, achieving true use case traceable modularity of the implementation components. A complimentary paper in the same journal [Pawlak 04] further elaborated on AO, especially on the *pointcut construct*, a key mechanism to allow extension without any provision in the existing solution.

Implementation time quarantine—cell H in Table 1—can be achieved with an interface in the base solution that is implemented in the extending component (see Figure 3). The `Base Client` component is dependent only on the extending component interface (not on interface implementation) in a quarantined manner.

Runtime quarantine—cell K in Table 1—can be achieved by using runtime component management patterns. These patterns eliminate dependency of the base calling component on the implementation of the called components. These patterns include Fowler's Inversion of Control or Service Locator patterns [Fowler 04], Web-Services, Object Request Brokers, and others.



All the above techniques are useful techniques for supporting a quarantining of the base solution against the danger of a new extension. Quarantine test scope in turn provides a reasonable compromise between flexibility and maintenance.

3 CONCLUSION

All software reuse can be classified into twelve distinct scenarios using a 2-dimensional matrix where one dimension is the testing scope (e.g. infectious, quarantined, and immune) and the other is binding time (e.g. requirements time, design time, implementation time and runtime)

The infectious scenarios provide the most flexibility in creating new solutions; however they are prone to produce tight coupling between base and new solution and creating cloned software. Therefore these scenarios are not recommended for customization, especially for the large-scale initiatives, like software product lines. Infectious scenarios can be beneficial for a short lifecycle single solution that does not require maintenance.

The immune scenarios are clone proof, however, they are expensive to build and limit a great deal of flexibility. Therefore, these scenarios are recommended only for solutions that need to be customized in large quantities, like COTS products.

The quarantine scenarios represent a compromise between flexibility and protection from cloning. Therefore, these scenarios are recommended for most cases of non-COTS software customization. Depending on the binding time the use case extend and quarantined include relationships, aspect-oriented methods and languages, Façade, Inversion of Control and Service Locator patterns are the major techniques to support quarantine scenarios.

REFERENCES

[Chilton 03] J. Chilton. The case against human cloning (humans cloning software). *Domino Power Magazine*. September 2003.

[Clements 02] P. Clements, L. Northrop. *Software Product Lines. Practices and Patterns*. Addison-Wesley, 2002.

[Dikel 97] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson. Applying Product-Line Architecture, *Computer*, vol. 30 no. 8, August 1997, pp 49-55.

[Fowler 04] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html.

[Jacobson 97] I. Jacobson, M. Griss, P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success.* Addison Wesley. 1997.

[Jacobson 03] I. Jacobson. Use Cases and Aspects – Working Seamlessly Together. *Journal of Object Technology*. Vol. 2, No. 4, July-August 2003. http://www.jot.fm/issues/issue_2003_07/column1

[Liskov 88] B. Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*. Vol. 23, No. 5, May 1988.

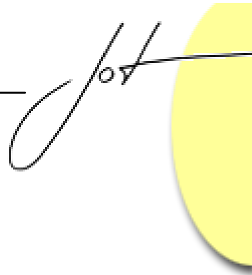
[Martin 96] R. Martin. Engineering Notebook. *C++ Report*. Nov-Dec, 1996.

[Martin 02] R. Martin. *Agile Software Development, Principles, Practices, and Patterns*. Prentice-Hall, 2002.

[Kane 97] D. Kane, W. Opdyke, D. Dikel. Managing Change to Reusable Software. *Pattern Languages of Programs (PloP) Conference* 1997.

[Pawlak 04] R. Pawlak, H. Younessi. On Getting Use Cases and Aspects to Work Together. *Journal of Object Technology*. Vol. 3, No. 1, January-February 2004. http://www.jot.fm/issues/issue_2004_01/column2

[Weiss 99] D. Weiss and Chi Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.



About the authors

Vitaly Khusidman is an Architecture Director at Unisys Corporation. Starting from 1977 he was an architect and/or development manager for numerous real-time control, health care, telecommunications, publishing, financial and insurance mission critical projects. He earned his Ph.D. in Computer Science from Institute for Control Problems of Russian Academy of Science in 1987. Email: vitaly.khusidman@unisys.com

David M. Bridgeland is Chief Technology Officer, Global Business Transformation at Unisys Corporation. He has created and led technology teams at system integrators and at venture-backed startups. He earned his M.A. in Computer Science from the University of Texas at Austin in 1989. Email: david.bridgeland@unisys.com.