Microsoft

Testing Software Patterns

patterns & practices

Microsoft®

Testing Software Patterns

patterns & practices

Mohammad Al-Sabt and Matthew Evans, Microsoft Corporation

Geethika Agastya, Dayasankar Saminathan, Vijay Srinivasan,
and Larry Brader, Satyam Computer Services Ltd.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This Documentation is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2003 Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Contents

Testing Software Patterns

Introduction

Patterns have been around conceptually since the late 1970s when Christopher Alexander wrote *The Timeless Way of Building* and *A Pattern Language*, which helped spark the pattern movement that was later applied to software engineering in the 1990s. Although Alexander and the software patterns community have created quite a stir around the construction of a pattern language, *testing* such a language has been largely neglected.

Testing has a different agenda: to substantiate designs and implementations based on certain criteria and according to a methodology. Hence, following Microsoft® Solution Framework (MSF) processes, the test team has developed a formal testing methodology and testing criteria, which are distilled from best practices found in the larger patterns community.

This document describes the testing methodology that the test team developed and applied to the new field of testing software patterns.

Who Should Read This Guide

This guide is written for testers, architects, designers, and developers of patterns who want to validate the patterns that they produce in a more systematic and regimented fashion than is generally practiced in the industry today.

History

When the testing teams first examined how to test patterns, the only industry standards were peer review and a variety of pattern definitions, all of them following the Rule of Three as their base test. The AntiPatterns Web site (*http://www.antipatterns.com/whatisapattern/*) offers following definition of the Rule of Three:

> **RULE OF THREE**: A software pattern documents a recurring solution. The pattern solution abstracts three or more experiences. The solution is something which is regularly applied or practiced by some community(s) of sophisticated developers and architects. The logical basis for the rule of three is: the first occurrence shows the design can work, the second occurrence is interesting, and the third occurrence suggests that the design *might* be worthy of being a pattern because it appears to have a wider applicability. *Non-normative comment:* The informal concept behind the rule-of-three is: the first occurrence is an event, the second occurrence is a coincidence, and the third occurrence may be a pattern.

To build a methodology and criteria, the test team conducted discussions with Microsoft and industry pundits, interacted with the various patterns groups, and reviewed existing patterns literature.

Test Methodology

The following high-level overview should help you understand the overall testing approach, before you read the details about the specific tactics used and the criteria against which patterns were evaluated. Figure 1, shows the overall methodology that the test team used.

The overall testing methodology consisted of five parts:

- **Literature comparison**—compares the pattern to existing write-ups of this pattern in the greater patterns community. Is the pattern well-formed and does the guidance it provides agree with established patterns?
- **Comparison against other implementations**—compares the implementation described against past implementations discussed in previously existing patterns. This includes implementations in other technologies.
- **Implementation testing**—validates the implementation against best practices for the technologies discussed. This is close to traditional testing, except that an SDE/Test may need to create the code to be tested.
- **Technical writing clarity**—applies UI testing aims to content:
  - Are the directions clear?
  - Is the structure of the dialog boxes (content) consistent?
  - Is the flow self-evident?
  - Are the help files (3rd party books) consistent with the application?
- **Brainstorming & proof by contradiction**—applies aims of testing for exception conditions (for example, loss of network connectivity or bad data) to the content.

Although this five-part testing methodology accounts for the main areas addressed, the tactical form of the process consisted of evaluating the pattern as stand-alone guidance and as part of a cluster of patterns. For more information, see “Testing Tactics.”

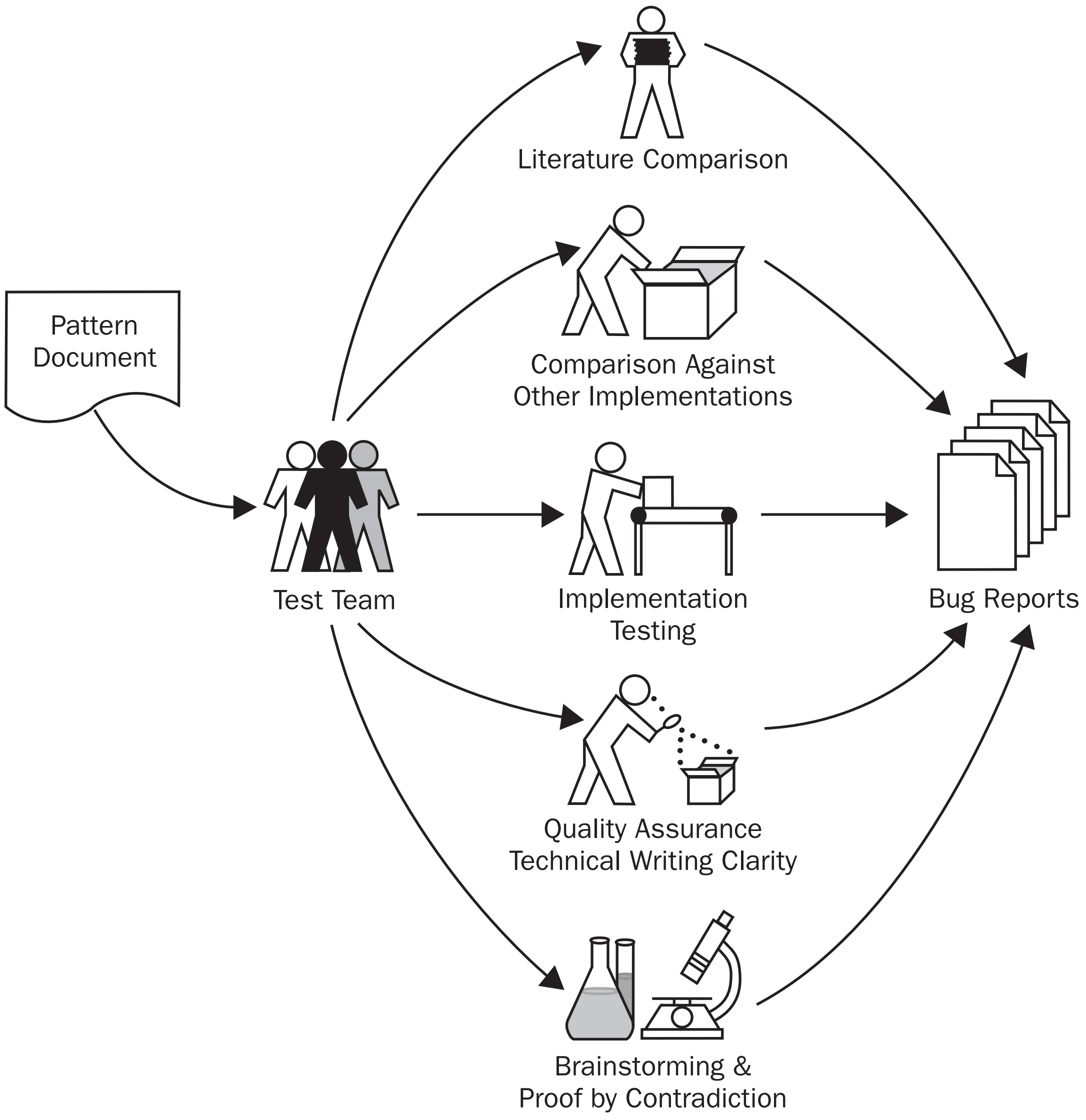
Pattern Document
Literature Comparison
Comparison Against Other Implementations
Test Team
Implementation Testing
Bug Reports
Quality Assurance Technical Writing Clarity
Brainstorming & Proof by Contradiction

Figure 1
Overall Testing Strategy

Test Criteria

The test team evaluated Microsoft patterns against the templates that the writers adopted and against quality characteristics that the larger patterns community has identified.

Microsoft Pattern Form

Although they are categorized into three levels of refinement (architecture, design, and implementation), Microsoft patterns are written in only two forms: one for architecture and design patterns and the other for implementation patterns. The two forms, like the patterns themselves, are designed to be complementary; the basic difference is in the required elements of which each consists.

Architecture and Design Pattern Form

Architecture and design patterns consist of the following **required elements** which all patterns of this form must include:

- Pattern Name
- Context
- Problem
- Forces
- Solution
- Example
- Resulting Context

Architecture and design patterns consist of the following **optional elements**, which pattern writers use only as necessary:

- Aliases
- Testing Considerations
- Security Considerations
- Operational Considerations
- Known Uses
- Variants
- Related Patterns
- Acknowledgments

Implementation Pattern Form

Implementation patterns consist of the following **required elements** which all patterns of this form must include:

- Pattern Name
- Context
- Implementation Strategy
- Resulting Context

Implementation patterns consist of the following **optional elements**, which pattern writers use only as necessary:

- Background
- Example
- Tests
- Testing Considerations
- Security Considerations
- Operational Considerations
- Known Uses
- Variants
- Related Patterns
- Acknowledgments

Through research, the test team defined several characteristics for every pattern, regardless of its form or level of refinement. These characteristics became the criteria for evaluating all architecture, design, and implementation patterns. In addition, the test team identified a few characteristics that apply only to implementation patterns. The following sections describe both types of characteristics.

Common Pattern Characteristics

The test team identified the following common characteristics of high-quality patterns, according to published research papers on the subject of patterns development and use (please refer to references section for a list of these resources). These characteristics were shared and reviewed with development and program management to drive early feedback and set expectations:

- Relevance and Completeness
- Evocative Naming
- Identified Relationships
- Composability
- Extensibility

- Absence of Pattern Overload
- Tradeoff and Alternatives
- Reconciliation of Forces

Relevance and Completeness

The pattern should be both relevant to the subject matter and to the pattern language of which it is a part. In addition, it should be a complete solution to the problem within the context that the pattern defines.Some of the questions you can ask to determine relevance and completeness include:

- Is the problem that the pattern defines relevant to the problem and forces driving the pattern? An answer of no here could mean that the pattern is not well-defined.
- Are the key decisions required for making architecture and design choices evident? An answer of no here could mean that the pattern is not complete or underdeveloped.
- Does the solution that the pattern defines only partially solve the problem? An answer of no could mean that either the problem is too broadly defined or that the solution is incomplete.

Evocative Naming

The pattern name should represent what the pattern intends to achieve and should be descriptive without being verbose. This name should use a metaphor or terminology that is meaningful to the audience and that quickly encapsulates the problem-solution pair in the minds of readers. The pattern may have an alias by which the pattern is commonly known. Any alias names for the pattern should pass the same tests as the pattern name itself.

Identified Relationships

Defining relationships, roles, and responsibilities for the elements (such as subsystems and components) is a key aspect when building the solution for the problem in a pattern. Any well-conceived pattern has clear relationships captured through formal engineering models, with the roles and responsibilities unambiguously described without compromising the intent of the pattern. This characteristic maps to solution section.

Composability

As you might expect, the term *composability* derives from music. The composability characteristic treats a pattern as an atomic unit that is analogous to a musical note. A composable pattern must be formulated in such a way that it can be combined with other related patterns to solve larger problems. This is similar to the quality of a musical note that allows you to combine it in different ways to form numerous compositions. Like many analogies, it has limitations.

Maximizing composability means minimizing restrictions and maximizing flexibility to improve the ratio between functionality on the one hand and simplicity, ease of use, and ease of learning on the other. It is completely possible to compose a set of patterns to engineer a total software solution. This characteristic maps to the pattern as a whole. Answering some of the following questions can help you determine whether a pattern is composable:

- Does this pattern, or its context, provide a solution that could form a part of the solution to a larger problem?
- Does this pattern, while helping to solve a larger problem, exclude the usage of any other patterns?
- Does this pattern work with other patterns to provide a solution to a larger problem, apart from solving the problem it is intended for?

Extensibility

Any well-conceived and well-documented pattern is extensible and also lends itself to parameterization so that it can be used along with other related patterns to solve a larger problem. Extensibility is critical because patterns are rarely implemented in isolation. An extensible pattern often provides a solution in a problem domain that is considered to be a superset or extension to the current problem. This characteristic maps to the pattern as a whole. Some of the questions you can ask to determine pattern openness include:

- Is the solution generic enough to enable it to be applied to specific business application scenarios having the relevant context?
- Is the pattern solution open to enable parameterization and hence generate variants of itself to apply to specific business application scenarios with the relevant context?

Pattern Overload

Patterns can be easily overloaded because anything and everything that attempts to solve a recurring problem can be termed as a pattern. In other words, the pattern experts have clearly mandated that patterns should attempt to solve potential technical/business problems that impact architecture and design decisions. This characteristic maps to the context, problem, and solution sections. Some of the questions you can ask to identify pattern overload include:

- Is the recurring problem inherent to the application, platform, or programming language? An answer of yes here could mean that the pattern contains too much implementation-specific information.
- Does the solution that the pattern provides consist of elements that can be used in the architecture or design for the problem being addressed? If the answer is no, then implementing the pattern may require that you use more architecture and design patterns. This would extend the architecture or design pattern beyond its intended scope.

- Does the solution consist of implementation (programmatic) sequences to use for addressing the problem? If the answer is yes, then the solution is tied to a specific technology or language. Another problem could be that the solution is overloaded with other patterns.

Tradeoffs and Alternatives

This criterion is very important because it enables the pattern tester to clearly validate whether there is a compromise for forces that are in potential conflict. This would mean that the pattern has to provide alternative solutions for each of those forces depending on the compromise.

Reconciliation of Forces

Forces which are not in conflict are reconciled in a very clear manner when providing the solution to a problem. Because forces are the direct fallout from the problem at hand, the solution would reconcile these forces without compromising the pattern's intent. This characteristic maps to the solution section.

Characteristics Specific to Implementation Patterns

In addition to the common pattern characteristics just discussed, a few additional characteristics apply only to implementation patterns.

Applicability to Target Technologies

It is very important to ensure that the pattern can be applied in the target technologies without compromising the pattern's intent. Well-written implementation patterns achieve this characteristic by applying one or more technology features to provide alternate solutions to tradeoff forces, reconciling the forces that are not in conflict with the technologies used, and providing guidelines and best practices. This characteristic maps to the solution section.

Code Review

If any code snippets are included in the pattern, the code must be tested against the following criteria:

- Microsoft naming conventions and best practices coding standards as stated in *Coding Techniques and Programming Practices (Visual Studio .NET Design Considerations)*, which is on the MSDN Web site at *http://msdn.microsoft.com/library/en-us/vsent7/html/vxconcodingtechniquesprogrammingpractices.asp*
- Independent program block with zero external dependencies

Scenarios

Application scenarios are defined. The scenarios depict the problem and the forces that drive the creation or need for a pattern. The scenario creation helps the reader relate to the context of the pattern in a tangible manner. This also helps the reader to understand the approach that the pattern explains and to validate whether the forces have been reconciled. Some of the questions you can ask to evaluate the scenarios in the pattern include:

- Is it clearly a situation that is common?
- Is it a mundane application problem, programming, or implementation detail?

Testing Tactics

Applying the testing methodology to patterns involved testing from two different viewpoints. The first viewpoint was to test each pattern independently. The second viewpoint was testing the pattern within the context of other patterns, or according to pattern clusters.

Stand-alone: Focus on the Pattern by Itself

Stand-alone testing is the place to start in testing any pattern. The stand-alone test focuses on the pattern itself and tests each of the characteristics identified earlier (see Figure 2).

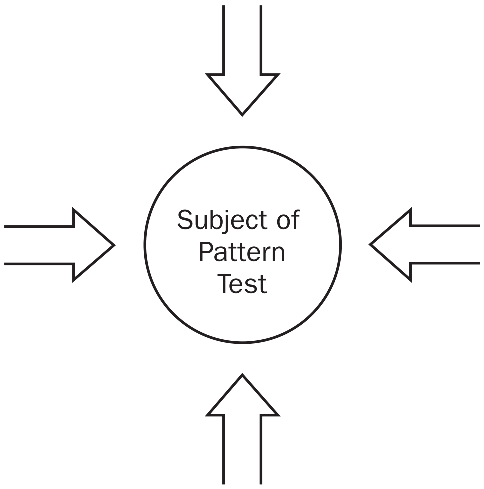

Figure 2
Stand-alone pattern testing focuses inward

Pattern catalogs often contain pairs of design and implementation patterns. In such cases, the stand-alone testing tactic calls for thoroughly evaluating the design pattern first, before moving on to any related implementation patterns.

Design-Level Testing

If a pattern cannot pass the stand-alone design-level test, it is highly unlikely that it will pass any subsequent tests. The stand-alone design-level testing consists of the following tests described earlier under "Testing Methodology":

- Literature comparison
- Comparison against other implementations
- Technical writing clarity

From this examination, the tester derives the main testing points for pattern implementation testing.

Implementation-Level Testing

Implementation-level testing examines one or more implementations for a given design pattern. One of the first items to examine is the process that the designer followed to create an implementation based on that design pattern. This instantiation of a pattern is based on the problem domain chosen to demonstrate the pattern usage. Does the process and problem domain make sense for this pattern? Are there better implementations? Or should there be more than one implementation to demonstrate the particulars for the design pattern? The process that the designer follows is a demonstration of refractoring the pattern from an abstract level to a particular problem domain. This process gives insights into how to use the pattern in actual operation and validate the correct usage.

At this level of stand-alone testing, a generic widget testing application is created per pattern. The whole purpose is to code and exercise the basic qualities that form this pattern. Figure 3 shows a pattern view of a widget testing application for a Singleton. The widget test framework can be replaced with a generic Test Harness.

Simple Widget Factory Tester

Figure 3 shows an example Widget Factory Tester that tests singularities for nonthreaded and threaded implementation. This basic approach can be applied to other patterns.

Basically, the Widget Model-View-Controller (MVC) creates the main factory which then creates the various entities as it goes through each test. The Watchdog is normally found in real-time system to handle error conditions for threads and processes. Using it here provides a way to build more complex testing systems.

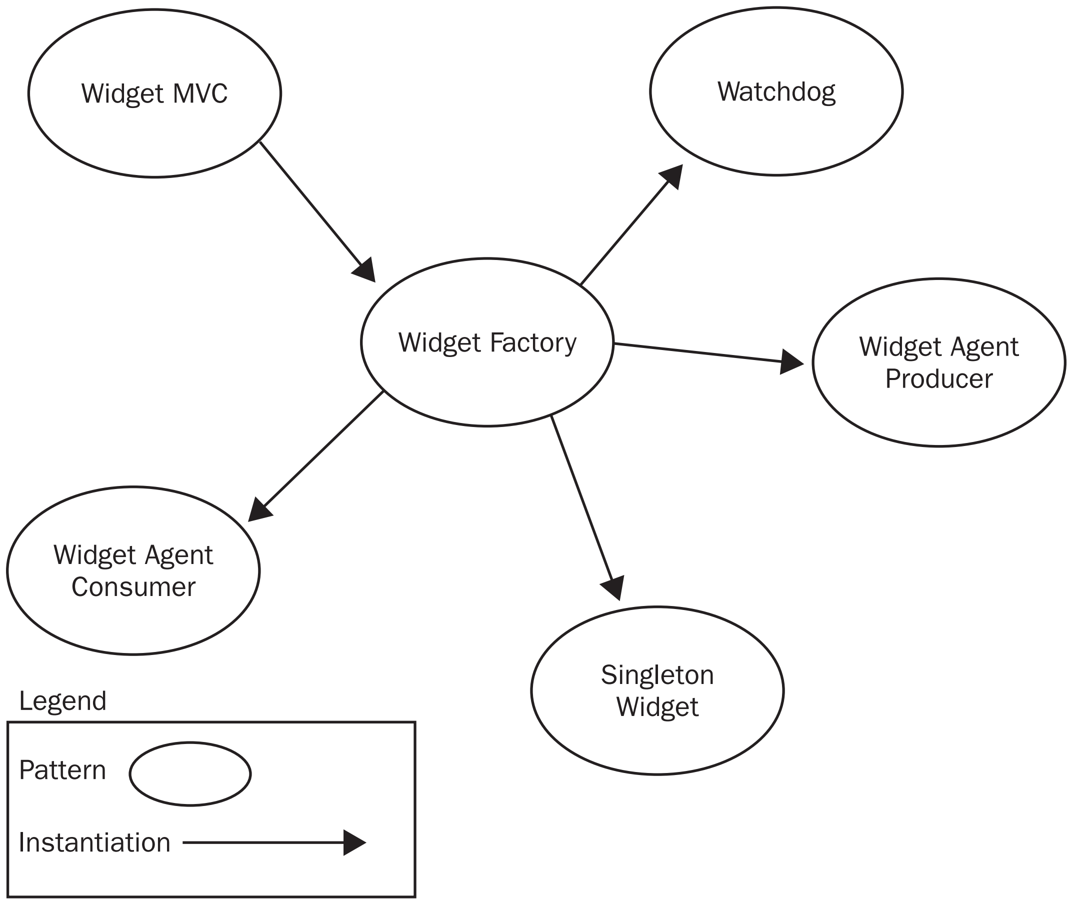

Figure 3
Simple widget test harness used for testing Singleton

The following example test runs reflect the process followed to test the *Singleton* pattern by using the widget test harness:

► **To ensure that there is only one instance in a single thread**

1. Instantiate a Singleton.
2. Reference it from the number of stub objects.
3. Create another Singleton and then determine that it is unique.

► **To ensure there is only one instance with multiple threads**

1. Instantiate a Singleton Threaded Object.
2. Create consumer and producers threads of the Singleton Object. For example, the consumers take the number off the Singleton stack, while the producers load the stack.
3. Create another Singleton and then determine it is unique.

► **To test what dependencies occur with usage of this pattern in a single thread**

1. Instantiate a Singleton.
2. Reference it from the number of stub objects.
3. Destroy the Singleton and then analyze the results.

► **To test what dependencies occur with usage of this pattern in a multiple thread**

1. Instantiate a Singleton Threaded Object.
2. Create consumer and producers threads of the Singleton Object. For example, the consumers take the number off the Singleton stack, while the producers load the stack.
3. Destroy the Singleton and then examine what happens to the dependencies. What needs to be considered in implementing this pattern in a threaded environment?

Cluster Testing: Focus on the Pattern Within a Cluster or Framework

As Figure 4 shows, cluster testing focuses on how the pattern works with other patterns, also known as the composability of the pattern.

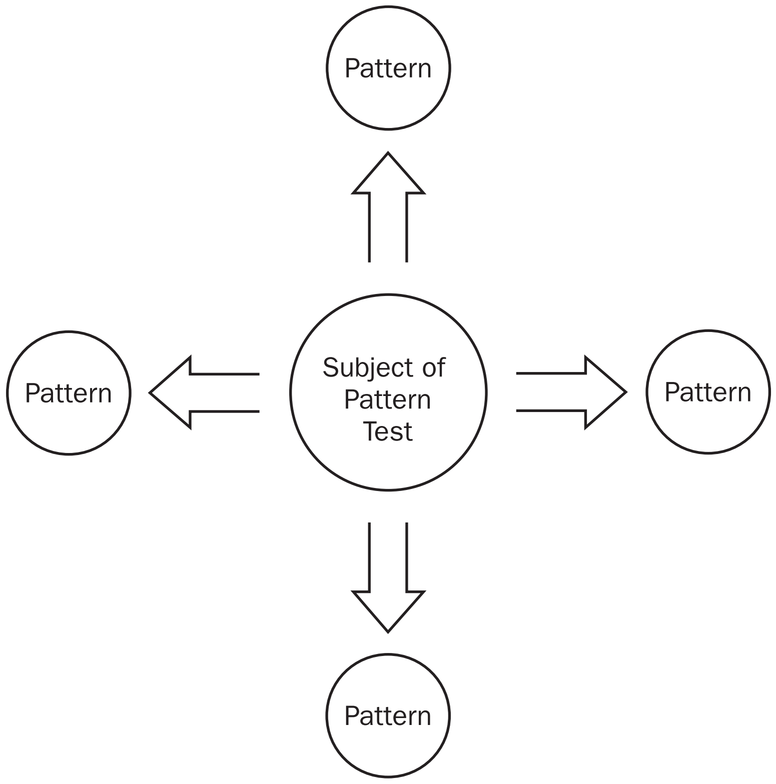

Figure 4
Cluster testing focuses outward on the architectural context

Like the stand-alone testing, cluster testing consists of design and implementation stages.

Design-Level Testing

Where the stand-alone emphasizes the core attributes of the pattern, the system-level design tests emphasize its relationship with other patterns and pattern clusters.

Implementation-Level Testing

The simple widget factory used in stand-alone testing also exposes how patterns work within clusters of other patterns as well as other patterns concepts. Again a generic testing framework was used, which can be applied to any general testing harness (see Figure 5).

To test a pattern in a cluster or in a framework, consider the following points:

- Know how extensive the testing framework needs to be. Is it a single system or a distributed system?
- Testing against multiple patterns means that some of the testing on the other patterns will not be complete yet, especially it they form part of the cluster being tested. Hence, cluster testing may span several patterns.
- The preceding testing widget framework is a node and can form the basis for a complex distributed processing system.
- The widget test harness enables you to add performance analyzer to this system; hence this simple distributed system now can be used to test load balancing, throughput, fault-tolerance, or other metrics.
- As more of these testing architectures are built. third parties could download them and use them as basic frameworks to build upon.

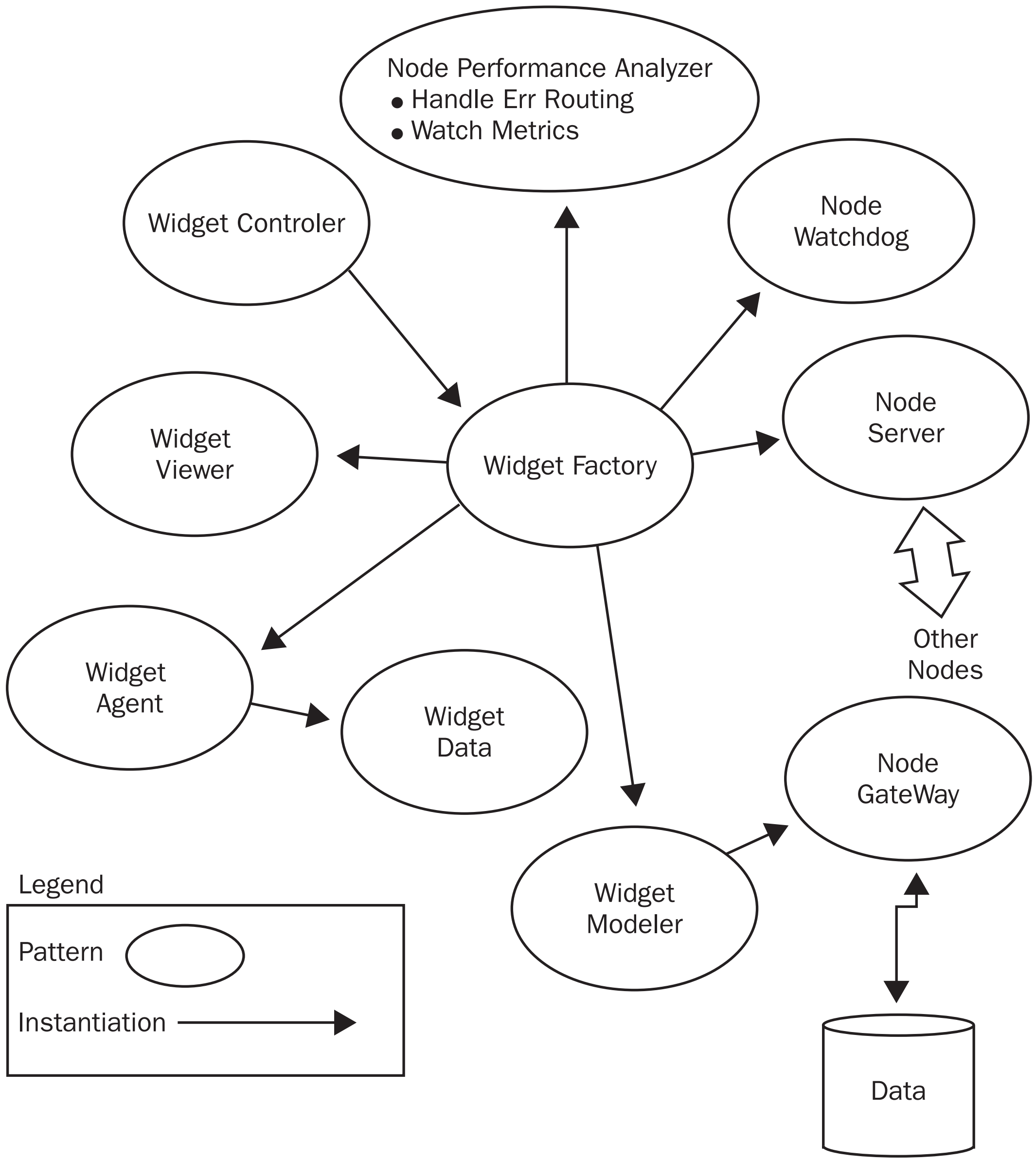

Figure 5
Widget test harness for testing a pattern for a distributed environment

Assigning Bug Severity

To determine the severity to assign a bug to a pattern, use Table 1 as a guide.

Table 1: Guide to Assigning Bug Severity

Sev.*	Cases/situations	Conforms to all guidelines
1	Pattern do not completely resolve the forces. Pattern fails to pass the test when validated against the test criteria. If the pattern does not give the best solution to handle the context/problem. Section incomplete or steps missing in a flow. Section is not technically correct. Information within a section is not relevant in the context. When results or behavior of a process flow contradicts with logically expected results.	User has found that it does not meet the criteria. There is no workaround. **If any of cases is true, then it's a severity 1** **Otherwise, it's a severity 2**
2	Steps within a section or solution, are not in the correct order. No uniformity between different sections. A logical step combines two or more major steps in one single step, unless applicable. Two differently named bookmarks/hyperlinks point to the same section. Hyperlinks/bookmarks of the same name in two different locations point to two different links. Hyperlinks/bookmarks do not load properly (this is not a defect of the guide, but the defect of the links). Hyperlinks in the guide are incorrect or unclear or not relevant. Hyperlinks do not contain the said information. Antagonistic language.	User has no simple workaround to mend situation. User cannot easily figure out workaround. **If any of these conditions is true then it's a severity 2 defect, else it's a severity 3**
3	Minor documentation errors and inaccuracies. Grammatical errors, Text misspellings, and so on. Document formatting is inconsistent. Acronyms not explained. Information is out-dated. Duplicate links provided to the same section.	User has a simple workaround to mend situation. User can easily figure out Workaround. This does not cause a bad user experience/interface. Testing of other scenarios is not affected here. **If these are true then it is a severity 3, or it is a severity 2 defects.**
4	Suggestions Future Enhancements	Clearly not a product defect.

* Sev. is an abbreviation for severity.

Pattern Test Trace Matrix

The pattern test team used a trace matrix to organize and conduct systematic tests on all patterns. The basic form of this trace matrix is shown in Table 2.

Table 2: Pattern Test Trace Matrix

Test Criteria	Methodology	Common Checklist Questions	[Pattern Name]
Relevance and Completeness			
Evocative Naming			
Identified Relationships			
Composibility			
Extensibility			
Pattern Overview			
Tradeoffs and Alternatives			
Reconciliation of Forces			
Applicability to .NET			
Scenarios			
Code Reviews			
Pattern-Specific Considerations			
General Observations			

For readability, Table 2 shows the overall layout of the Test Trace Matrix without showing the contents of each cell. Table 2 shows the test criteria identified earlier in this document as headings to the rows of the table. As you look horizontally across the matrix for each test criteria, you see general methodologies and common test questions, as well as test questions that are specific to each pattern. The contents of the Methodology and Common Checklist Questions columns are omitted here, because the contents would repeat the information discussed in the "Test Methodology" and "Test Criteria" sections of this guide. The contents of the other columns of the matrix would be specific to the actual patterns you are testing.

Contents of the Test Trace Matrix for the Observer Pattern

The following list shows the actual contents (organized by row) of the [Pattern Name] cell of Table 2 during testing of the *Observer* pattern. The data was reformatted to fit within this document. The contents of the Methodology and Common Checklist Questions Columns would remain unchanged as the test team added more columns for each individual pattern being tested. As might expect, the complete version of the matrix became very large.

Relevance and Completeness

Problem: The content in the problem section should be relevant to what the pattern is attempting to solve.

Test Case: Does the literature address how to notify the state change of one object (source) to all its dependents in one to many relationships ensuring maximum decoupling?

Result: Addressed.

Forces: The content in the forces section should be relevant and should include all possible forces that influence the given pattern.

Test Cases:

Does the section address:

1. Maximum amount of decoupling between the source and dependent objects in one to many relationships to ensure reusability of the objects?
2. Easy plug-in and plug-out of the dependent objects without affecting the source object?
3. Minimal changes to the dependent objects if the source object is changed and minimal changes to the source object if the dependent objects are changed?

Results:

1. Addressed.
2. Addressed.
3. Addressed.

Pattern Overload

Context: The context section should determine key decisions that have to be made when evolving architecture or design models for a business problem. Context should not be defined around recurring problems in environments such as technologies, programming languages, and so on.

Problem: The problem section should describe potential technical or business problems that impact your architecture and design decisions. Any attempt to describe a mundane platform or programming related problem result in pattern overload. Examples are attempting to build a linked list, which is a programmatic problem that does not influence the architecture or design.

Solution: Solution should be aligned with providing architectural or design elements (such as interaction diagrams or sequence diagrams) that lead to architecture and design models, instead of solutions to programming or environment-related problems.

Test Cases:

1. Is the business problem that the pattern defines relevant to the problem and forces driving the pattern?
2. Is the recurring problem inherent to the application, technology, or programming language?
3. Are the key decisions required for making architecture and design choices evident?
4. Does the solution consist of elements that can be used in the architecture or design for the problem being addressed?
5. Does the solution consist of implementation (programmatic) sequences to use for addressing the problem being addressed?

Results:

1. Problem is related to the pattern.
2. It is a recurring problem related to an application.
3. They are evident.
4. Yes.
5. Sequence diagrams are provided.

Evocative Naming

Name and Alias: The pattern name should represent what the pattern intends to achieve and should be descriptive without being verbose. This name should use a metaphor or terminology that is meaningful to the audience and that quickly encapsulates the problem-solution pair in the minds of readers. The pattern may have an alias by which the pattern is commonly known. Any alias names for the pattern should pass the same tests as the pattern name itself.

Test Cases:

1. Is the pattern name representative of what the pattern intends to achieve?
2. Is the pattern name ambiguous or verbose (longer than a few words)?
3. Does the pattern have an alias?
4. Is the alias name descriptive without being verbose?

Results:

1. Pattern name is relevant.
2. Not ambiguous.
3. No.
4. Not applicable.

Tradeoffs and Alternatives

Solution: The solution should clearly discuss the tradeoffs when reconciling the forces that have been described in the pattern. Tradeoffs refer to the choice of reconciling one force over another if they are in conflict with each other.

Test Case: Does the pattern address the tradeoffs between the Push and Pull model of notification?

Result: Addressed.

Extensibility

Overall Content: The pattern should be open for extension or parameterization by other patterns so that they can work together to solve a larger problem.

Test Cases:

1. Is the solution generic enough to enable it to be applied to specific business application scenarios having the relevant context?
2. Is the solution open to enable parameterization and hence generate variants of itself to apply to specific business application scenarios with the relevant context?

Results:

1. Pattern is generic.
2. Pattern is open to extension and parameterization.

Composability

Overall Content: The pattern must be formulated in such a way that it can be used with other patterns to compose an overall solution to a larger problem. The pattern itself should help to build a model architecture or design for a specific problem related to a specific context.

Test Cases:

1. Does this pattern, or its context, provide a solution that could form a part of the solution to a larger problem?
2. Does this pattern, while helping to solve a larger problem, exclude the usage of any other patterns?
3. Does this pattern work with other patterns to provide a solution to a larger problem, apart from solving the problem it is intended for?

Results:

1. Yes.
2. No.
3. Yes. It can work with other patterns, based on the application requirements.

Reconciliation of Forces

Solution: The solution should clearly explain how the forces are reconciled.

Test Case: Does the solution address each force that the pattern mentions?

Result: All forces are addressed.

Identified Relationships

Solution: The solution should describe subsystems, components, their responsibilities and relationships. Relationships should be captured through architecture or design elements in the form of interaction and sequence diagrams.

Test Cases:

1. Does the solution provide clear relationship between Subject, Concrete Subject, Observer and Concrete Observer?
2. Is a visual representation of the interaction amongst the classes and interfaces provided?

Results:

1. Addressed.
2. Addressed.

Guidelines and Best Practices

Solution: Solution should clearly provide guidelines and best practices from external sources such as comments from experts in the pattern, URL links to patterns, which will enable audience to understand and apply the patterns to build architecture/design models in the best manner

Test Cases:

Does the pattern provide guidelines for:

1. Performance optimization of the update operation? (For example, reducing unnecessary calls to the Observer.) This can be done if the Observer subscribes to the specific events (interests) of the Subject.)
2. Avoiding exceptions or undesirable conditions such as deadlocks, or dangling references of the Subject or Observer, when the observer is located remotely?
3. Adopting the Push and Pull model of notification?

Results:

1. They are provided, but are not very evident. *[Bug 01]*
2. Addressed.
3. Addressed.

Contents of the Test Trace Matrix for the Observer Implementation Pattern

The following list shows the actual contents (organized by row) of the [Pattern Name] cell of Table 2 during testing of the *Implementing Observer in .NET* pattern. The data was reformatted to fit within this document. The contents of the Methodology and Common Checklist Questions Columns would remain unchanged as the test team added more columns for each individual pattern being tested. As might expect, the complete version of the matrix became very large.

Applicability to Target Technologies

Solution: The solution can provide key information points in terms of how a given pattern can be applied to the target technologies (in this case, Microsoft .NET).

Test Case: Does the solution provide details on how this pattern can be applied in .NET?

Result: This is a design pattern. This can be applied to .NET.

Scenarios

Scenario: An application is monitoring the stock changes throughout the day. It needs to display the stock's current price and a graph for the fluctuation.

Problem: Illustrates the problem section of the scenario.

The current implementation enforces a tight coupling between the presentation object and the data object. This leads to the following issues:

1. Implementation changes in data objects result in changes in the presentation object and changes in the presentation object result in changes in the data objects.
2. It is not possible to reuse the presentation object or data object in any other part of the application because they are too tightly coupled.
3. Addition or deletion of a presentation object enforces changes in the data object.

Solution: Illustrates how the given pattern attempts to solve the problem depicted in scenario.

With the new solution in place we have a Stock class and StockDisplay and StockGraph classes. The instance of the Stock class is the data object that contains the current price of the stock. This instance acts as the Subject and with this the instance of the StockDisplay and the StockGraph subscribe as Observers. Now, when the Stock object receives any change in current stock price it notifies the instances of the StockDisplay and StockGraph to update themselves.

Reconciliation of Forces: Illustrates how the specified forces are reconciled in the solution using the pattern. With the implementation of the *Observer* pattern:

1. The StockDisplay and StockGraph classes can be reused if needed in the application because the are loosely coupled.
2. Addition or removal of any other kind of presentation object can be performed very easily without affecting the Stock.
3. If the implementation of the Stock changes to cope with the changes in the data source, the presentation objects (StockDisplay and StockGraph) are not affected.

Code Review

Test Case: Does the code adhere to Microsoft naming conventions?

Result: The variables do not have meaningful names. But it is a partial code snippet.

Pattern-Specific Considerations

Solution: Solution should bring out pattern specific-concepts that influence the overall composition of the pattern itself

Test Cases:

1. Does the literature explain the dependency, degree of coupling or decoupling between Subject and Observer?
2. Does the literature indicate the relationships and responsibilities of Subject and Observer clearly?
3. Does the literature address complex cases such as multiple (and complex) interim updates before a consistent state is reached in the Subject?
4. Is the usage and applicability of the pattern explained clearly?
5. Are the benefits of using Observer explained clearly?
6. Are the key issues addressed, such as:
    a. Who initiates the "Update"
    b. When to initiate the Notify"
    c. Subject-Observers mapping
7. Is the push/ pull model of notification addressed?
8. Does it address the case if multiple inheritance is not supported?

Results:

1. Addressed.
2. Addressed.

3. Addressed.
4. Addressed.
5. Addressed.
6. Addressed.
7. Addressed.
8. Addressed.

General Observations

The content is well-presented.

Additional Testing Tactics Specific to Data Patterns

Testing for Data Patterns, included the following additional tactics:

- Confirm that the logical model is not United States based only.
- Confirm that the logical model is normalized.
- Confirm that a physical model is presented:
  - Check the issues affecting the physical model are adequately discussed.
  - Check that appropriate indices, foreign keys, and so on are defined in the physical model
  - Check that appropriate triggers are implemented in the physical model

Patterns Tested

The following patterns currently published on MSDN were tested according to the patterns testing methodology, criteria, and tactics described in this document:

Enterprise Solution Patterns Using Microsoft .NET

The following patterns are included in *Enterprise Solution Patterns Using Microsoft .NET*:

- Model-View-Controller
- Implementing Model-View-Controller in ASP.NET
- Page Controller
- Implementing Page Controller in ASP.NET
- Front Controller
- Implementing Front Controller in ASP.NET Using HTTP Handler
- Intercepting Filter
- Implementing Intercepting Filter in ASP.NET Using HTTP Module

- Page Cache
- Implementing Page Cache in ASP.NET Using Absolute Expiration
- Observer
- Implementing Observer in .NET
- Layered Application
- Three-Layered Services Application
- Tiered Distribution
- Three-Tiered Distribution
- Deployment Plan
- Broker
- Implementing Broker with .NET Remoting Using Server-Activated Objects
- Implementing Broker with .NET Remoting Using Client-Activated Objects
- Data Transfer Object
- Implementing Data Transfer Object in .NET with a DataSet
- Implementing Data Transfer Object in .NET with a Typed DataSet
- Singleton
- Implementing Singleton in C#
- Service Interface
- Implementing Service Interface in .NET with an ASP.NET Web Service
- Service Gateway
- Implementing Service Gateway in .NET
- Server Clustering
- Load-Balanced Cluster
- Failover Cluster

Data Patterns

The following patterns are included in *Data Patterns*:

- Move Copy of Data
- Data Replication
- Master-Master Replication
- Master-Slave Replication
- Master-Master Row-Level Synchronization
- Master-Slave Snapshot Replication
- Capture Transaction Details
- Master-Slave Transactional Incremental Replication

- Master-Slave Cascading Replication
- Implementing Master-Master Row-Level Synchronization Using SQL Server
- Implementing Master-Slave Snapshot Replication Using SQL Server
- Implementing Master-Slave Transactional Incremental Replication Using SQL Server

Summary

The testing of software patterns is still a developing engineering field. This is certainly evident from the amount of formalization that this methodology represents compared to the original rule of three. By going utilizing this testing approach with the stated criteria brought the developers and testers to another level of understanding about patterns.

Now it is time to review and extend the work done. To evolve the work further, deeper understandings of the mathematics that form patterns will help mature the formalization. One of the items that stand out is to incorporate the pattern developer grid into forming a more rigorous pattern language. Another item is to incorporate some repository tools to help organize patterns and assist engineers to find the best solution.

References

This document references the following sources:

- *http://asusrl.eas.asu.edu/papers/Design-Pattern-Testing.htm*
- *http://www.enteract.com/~bradapp/docs/patterns-intro.html*
- *http://www.jguru.com/faq/Patterns*
- *http://www.hillside.net*
- *http://patterndigest.com*
- *http://hillside.net/patterns*
- *http://www.cs.wustl.edu/~schmidt/patterns.html*
- *http://www.martinfowler.com/eaaCatalog/*
- Alur, Crupi, and Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.
- Burbeck, Steve. "Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC)." *University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive*. Available at: *http://st-www.cs.uiuc.edu/users/smarch/st-docs /mvc.html.*

- Buschmann, Frank, et al. *Pattern-Oriented Software Architecture.* John Wiley & Sons Ltd, 1996.
- Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Fowler, Martin. "To Be Explicit." *IEEE Software*, November/December 2001.
- Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- Fowler, Martin; Rice, David; Foemmel, Matthew; Hieatt, Edward; Mee, Robert; Stafford, Randy. *Catalog of Patterns of Enterprise Application Architecture*. Available at: *http://martinfowler.com/eaaCatalog/.*
- Hay, David. *Data Model Patterns: Conventions of Thought*. Dorset House, 1996.
- Herzum, Peter and Sims, Oliver. *Business Component Factory*. John Wiley & Sons, Inc., 2000.
- Larman, Craig. *Applying UML and Patterns.* Prentice-Hall PTR, 2002.
- Meyer, Bertrand. *Object-Oriented Software Construction*, 2nd Edition. Prentice-Hall, 2000.
- Nock, Clifton. *Data Access Patterns*. Addison Wesley Professional, 1st edition, 2003.
- *patterns & practices*, Microsoft Corporation. "Application Architecture for .NET: Designing Applications and Services." *MSDN Library*. Available at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp.*
- Pree, Wolfgang. *Design Patterns for Object Oriented Software Development*. Wokingham: Addison-Wesley/ACM Press, 1995.
- Purdy, Doug; Richter, Jeffrey. "Exploring the Observer Design Pattern." *MSDN Library*, January 2002. Available at: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/observerpattern.asp.*
- Reilly, Douglas J. *Designing Microsoft ASP.NET Applications.* Microsoft Press, 2002.
- Reingruber, Michael C.; Gregory, William W. *The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models*. John Wiley & Sons, 1994.
- Schmidt, et al. *Pattern-Oriented Software Architecture, Vol 2*. John Wiley & Sons, 2000.
- Silverston, Len. *The Data Model Resource Book*, Vol. 1 and 2. John Wiley & Sons, 2001.

Community

This guide and patterns it describes are part of a new Patterns community on GotDotNet. GotDotNet is a Microsoft .NET Framework Community Web site that uses workspaces in an online collaborative development environment where .NET developers can create, host, and manage projects throughout the project life cycle. You can also use this Patterns community to post questions, provide feedback, or connect with other users for sharing ideas.

Access to the Patterns community is available from the following Web site:

http://gotdotnet.com/team/architecture/patterns

Feedback and Support

Questions? Comments? Suggestions? For feedback on this guide, please send e-mail to pnppatfb@microsoft.com.

The patterns discussed here are designed to jump-start the architecture and design of enterprise applications. Patterns are simple mechanisms that are meant to be applied to the problem at hand and are usually combined with other patterns. They are not meant to be plugged into an application. Example code is provided "as is" and is not intended for production use. It is only intended to illustrate the pattern, and therefore does not include extra code such as exception handling, logging, security, and validation. This deliverable, however, has undergone testing and review by industry luminaries and support is available through Microsoft Product Support for a fee.

Collaborators

Many thanks to the following advisors who provided invaluable assistance:
Rick Maguire, Shaun Hayes, Mike Kropp, and Ken Perilman; Microsoft Platform Architecture Guidance.