2008

An Event Monitor and Response Framework Based on the WSLogA Architecture

Todd Christopher Brett
Nova Southeastern University, todd@brett-family.net

An Event Monitor and Response Framework Based on the WSLogA Architecture

by

Todd Christopher Brett

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Information Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2008

We hereby certify that this dissertation, submitted by Todd Brett, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree Doctor of Philosophy.

_____	_____
Michael Laszlo, Ph.D. Chairperson of Dissertation Committee	Date
_____	_____
Frank Mitropoulos, Ph.D. Dissertation Committee Member	Date
_____	_____
Steven Zink, Ph.D. Dissertation Committee Member	Date

Approved:

_____	_____
Edward Lieblein, Ph.D., Dean	Date

Graduate School of Computer and Information Sciences
Nova Southeastern University

2008

An abstract of a dissertation submitted to Nova Southeastern University
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

An Event Monitor and Response Framework Based on the WSLogA Architecture

by
Todd C. Brett

October 2008

Web services provide organizations with a powerful infrastructure by which information and products may be distributed, but the task of supporting Web service systems can be difficult due to the complex nature of environment configuration and operation. Tools are needed to monitor and analyze such Enterprise environments so that appropriate engineering, quality control, or business activities can be pursued.

This investigation resulted in the development of a software development kit, the WSLogA Framework, which is inspired by the vision of Cruz *et al.* (2003, 2004). The WSLogA Framework provides distributed Enterprise systems with a platform for comprehensive information capture and environment management. Five component groups are intended for employment to enable integrated workflows addressing monitoring and response activities, but these components may also be used individually to facilitate the phased integration of the WSLogA Framework into existing environments. The WSLogA Framework's design is portable across technology platforms (*e.g.*, Java and .NET) and a variety of technologies may be substituted for the provided implementations to address unique system architectures.

The WSLogA Framework supersedes existing logging and monitoring solutions in terms of both capability and intent. Applications based on the WSLogA Framework have an internal, real-time view of their operation and may adjust their environment based on the information provided by events related to their or system activities. The WSLogA Framework is intended as a software development kit around which system functionality may be organized and implemented, which makes the WSLogA Framework an architectural peer or complement to traditional application frameworks such as Spring's Web module. WSLogA Framework based systems should be envisioned as information appliance elements rather than traditionally scoped applications or services.

Dedicated to Connie and Xavier with my love, respect, and appreciation.

Thank you.

In the beginning we must simplify the subject, thus unavoidably falsifying it, and later we must sophisticate away the falsely simple beginning.

Moses Maimonides

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

John Gall

Acknowledgements

I am fortunate to live as a global citizen, and I appreciate the ongoing support of family, friends, mentors, and organizations distributed across the domains of Canada, the United States, and beyond. The effort to interweave and accept the aggregate experience of these interactions has proven to be a challenge in its own right, but the broader membership has granted insight not otherwise available within a nationalist perspective; it serves as the multifaceted and holistic context by which I explore people and their information realization and knowledge exchange within the context of technology.

My wife, Dr. Constance Brett earned her Ph.D. in 2005, and she has generously shared her insights into the frustrations, experiences, and occasional irony that define the journey of a graduate student. Connie accommodated, seemingly without complaint, my years of extended hours as both a professional and academic, and at the most appropriate times brought me back from the plethora of papers to ensure my participation as a husband, father, and friend—breaks essential to our long term sanity, I am sure. I am indebted to Connie for her unfaltering support as a loving, dependable partner and friend.

Dr. Mike Laszlo, my advisor, and the dissertation committee are appreciated members of my academic journey. Mike has appropriately balanced his expert interventions with my need for independent exploration. His timely and direct feedback has never left me guessing as to his intent or my need for additional thought. Mike's guidance has fostered my appreciation of the rewarding intricacies of scholarly research.

The faculty and staff at Nova Southeastern University have frequently gone beyond the call of duty with their interest in my efforts and success. In particular, I am grateful for Dr. Steven Zink's accommodation of my interest in software engineering and its application to information retrieval, knowledge management, and information policy systems. Dr. Marlyn Littman's interest in distributed and parallel computing often mirrors my own, and I appreciate her efforts to highlight projects from my homeland—many of which I intend to further explore and, hopefully, contribute. The University merged its Information Science and Information Systems programs part way through my studies, and the persistent assistance of Dr. Eric Ackerman was critical to ensuring my timely completion of course obligations despite the revised course plans.

I am grateful for the support of the Central Ohio Compuware branch, and in particular to Carrie Archer, Jerry Jones, Vasil Hlinka, Lori Hubbard, and Matt Smith. These people tolerated my enthusiastic discourse, ensured that I had time to attend each of the semester institutes, and otherwise provided their encouragement as I progressed through the multitude of assignments, research reports, and the dissertation investigation. Clients such as the State of Ohio and Cisco Systems have also accommodated my unusual schedule with grace, and I wish to thank Mike Morris, Florina Comanescu, and Junilu Lacar for their support.

I appreciate the support of yWorks, which produces the yDoc UML generator. yWorks donated the use of yDoc for my research as the means by which professional grade class diagrams could be rapidly generated and integrated into JavaDoc pages. This capability has saved untold hours of document generation and enabled my ability to quickly assess the framework's structure so that key efficiencies and functionality could be introduced.

My extended family and friends, inherited and adopted, have always shown their support and cheered me on to greater achievements. Many parents, uncles and aunts, siblings and cousins, as well as neighbors are accomplished in their own right, and I appreciate sharing in their experiences as motivation for my own throughout this endeavor. Of these people, I especially wish to thank my parents.

My Mom, Pamela Norgan always listens patiently when I describe discoveries or vent about frustrations. She applies humor liberally to life, and I have adopted her way as the means for blunting my native impatience with the understanding that everything worth pursuing requires dedication and time.

My Dad, Richard Brett and Mom, Kathleen Brett encourage my questioning of life and the world around us—despite our divergences of philosophy and action. From reading to multiplication tables, these two have ensured my ability to independently engage the world and savor its nuances. They remind me of my civic responsibilities and that each individual's contribution to society is significant.

My Dad, "Sean" Fitzgerald and Mom, Karen Fitzgerald welcomed me into their family and provided their support throughout the uncomfortable period of my transition to meet the demands of a new but wonderful country. Their perseverance through personal and demanding challenges inspire and demonstrate how our will and conviction are essential to success and the enjoyment of life.

My Grandmother, Amy Brett believed in fostering imagination and creativity. I was always welcome to boil mud on her stove in the attempt to simulate alien landscapes, or to use magnets and wire for the purpose of establishing electric transmissions across her living room floor. Every question from why planes fly to how dinosaurs ended up in the dirt was asked at one point or another. Perhaps more than any other person, my Grandma Brett serves as the inspiration for a lifetime of continual questioning and exploration.

The challenges brought about by graduate studies forge many friendships, such as that with James Hoban. James shares my passion for problems inherent to corporate evolution and knowledge transfer, and we spent many hours debating these and other topics within and outside of the classroom. Shari Plantz-Masters humors my ongoing musings about teaching, but more importantly ensured my understanding of project management and enabled me to balance career obligations so that I could ensure time for family, friends, and further academic pursuit.

Finally, I wish to acknowledge the efforts of pioneers such as David Thompson, William Mackenzie, Simon Fraser, Meriwether Lewis, and William Clark. These intrepid individuals set out against unforgiving odds to illuminate their unseen universe and the rich cultures of people who lived within that vast horizon, and then shared their wealth of knowledge with their respective originating societies. I spent my adolescent years exploring in the shadow of their monuments—such as the Howe Pass—and retraced many of their routes throughout Western Canada and the Oregon Territory. Their determination and service in improving our understanding of the world is truly inspiring.

Table of Contents

List of Tables

Tables

List of Figures

Figures

Chapter 1

Introduction

Problem Statement and Goal

Web services provide businesses with a powerful infrastructure by which information and products can be distributed, but the task of supporting Web service systems can be difficult due to the complex nature of environment configuration and operation. Web service and host environment monitoring and analysis tools are needed to facilitate the formation of business and development strategies. Ideally, these facilities are an integral part of the systems they support. Unfortunately, existing tools do not permit the comprehensive integration of monitoring, analysis, and response mechanisms with Web service based systems and their host environments.

Cruz *et al.* described (2004) and provided a limited demonstration (2003) of their proposed monitoring architecture for Web services, the WSLogA (Figure 1-1). The WSLogA improves upon traditional click-stream traffic analysis strategies by using Web service intermediaries to analyze SOAP messages rich in detail as they travel through a system. However, the WSLogA is not suitable for production environment management because the architecture does not provide for integrated, holistic monitoring of both the transaction components and their environment with the customizable capability for rules based interaction based on event analysis. For example, a failed network router may cause the false

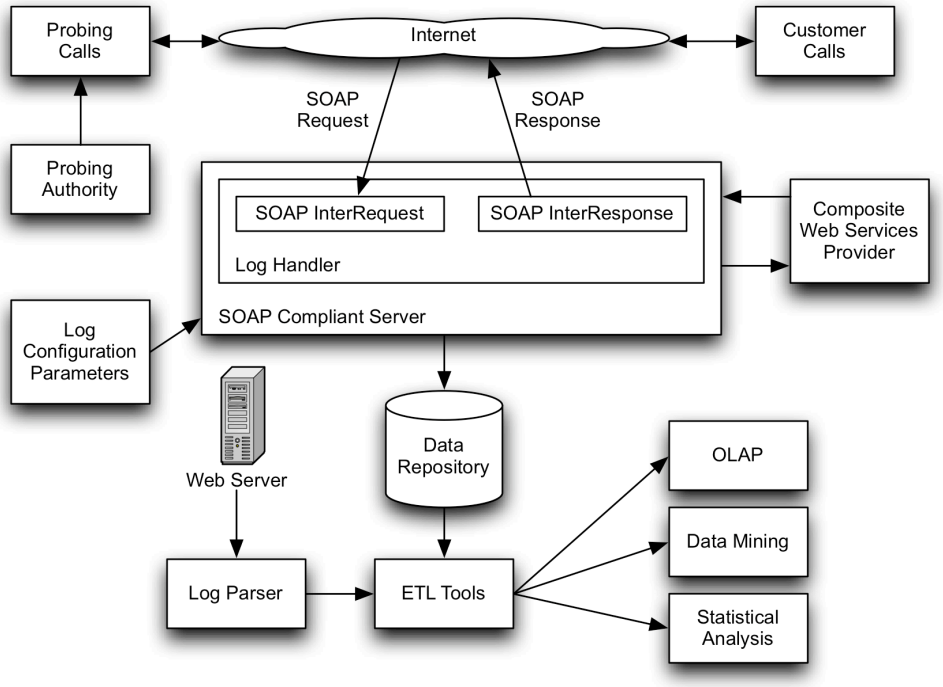


Figure 1-1. WSLogA topography.

identification of a transaction endpoint failure by a WSLogA component because the component would not understand how the environment caused the communication failure—it only knows that Web service transactions were interrupted. Administrators can chain third party tools to achieve post-mortem transaction analysis, but this approach is awkward and may fail to deliver real time response. Additionally, most tools cannot be modified so administrators and engineers cannot instruct the system to respond and correct the environment using improved techniques as knowledge of the problem domain is refined. Finally, the implementation of WSLogA components is an ad hoc effort that does not offer reuse across independent systems—new projects must recreate the architecture, which permits architectural fragmentation and the otherwise unnecessary introduction of bugs.

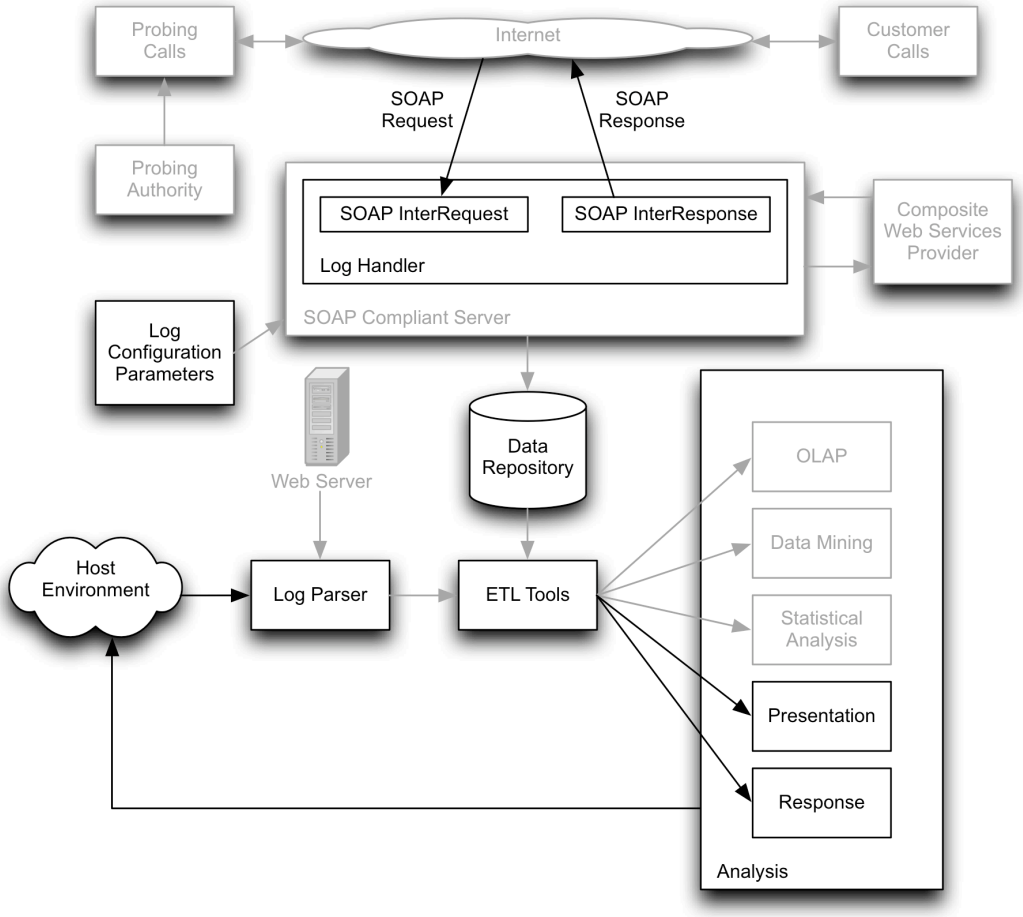


Figure 1-2. Aspects of WSLogA addressed by the WSLogA Framework.

This investigation successfully resulted in the establishment of a framework facilitating the development of WSLogA components, and significantly improves upon the WSLogA by incorporating mechanisms for policy driven information collection and normalization; transaction and environment event monitoring to facilitate holistic runtime analysis with the capability for real-time environment interaction; and the distribution of information communicating system behavior or state to people or external systems. The complex workflows inherent to monitoring, analysis, and response are predefined by the WSLogA

Framework using Template Method and Strategy patterns, which permits third parties to focus on implementing business rules rather than architecturally redundant logic. Bold elements in Figure 1-2 illustrate the investigation's focus within the WSLogA.

Investigation artifacts include the WSLogA Framework's design, its implementation using the Java platform and supporting technologies, and test systems based on Sun Microsystems' Adventure Builder system (Appendix B) that demonstrates the framework's solution for the problem domain. The data persistence layer was extended to support non-XML solutions, such as a relational database management system, and event handlers for application or operating platform concerns (*e.g.*, Application server or router logs) were defined to support comprehensive system behavior or state analysis.

Relevance and Significance

Web services provide a powerful tool for information and product distribution, yet such environments can be difficult to develop and support due to their complex nature. Transactions are event driven and rely on XML or proprietary messaging mechanisms, service components can wrap legacy systems that are difficult to integrate into dynamic workflows, and the Application server or other environment components can affect the operational behavior of system implementations in unforeseen ways. Production support, quality assurance, and engineering personnel must comprehend the operational impact and timing of these intricate interactions in order to provide a high degree of service quality.

Existing performance and event monitoring tools focus on the needs of a narrow audience, such as system administrators, or are intended for use within a single controlling organization. Few solutions use the SOAP messages from Web service components as the

principal channel for data capture, or even consider SOAP messages as a data source for analysis. This investigation is among the first to address production environment management using SOAP data as the primary vehicle for transaction characterization, and it fulfills the WSLogA vision with the establishment of a reusable framework suitable for driving holistic enterprise production management solutions. The Java based implementation ensures the WSLogA Framework's universal applicability to a variety of enterprise systems, including as those developed using competing platforms such as Microsoft's .NET.

Researchers can use the WSLogA Framework to better comprehend information exchange and generation between Web service components, and in particular for those systems comprised of components that are highly parallel or distributed across independent servers. Environment management issues pertinent to production control can be explored through extensions such as the event response engine, and policy components facilitate the exploration of information assurance within complex enterprise systems that could operate across legal or cultural boundaries. Practitioners can use the WSLogA Framework to integrate holistic transaction analysis and response into their enterprise systems with a reduced need for expert understanding in the problem domain.

Research Objectives

This investigation's intent was to establish a framework fulfilling WSLogA principles with significant improvements by means of information capture, information exchange, and environment management capabilities juxtaposing transaction and environment event analysis for distributed, service oriented systems. The completed framework's functionality must be mature enough for demonstration using the Adventure Builder application and

contrived framework extensions that exercise the WSLogA principles and improvements introduced as part of this investigation. Table 1-1 describes the envisioned subsystems comprising the improved WSLogA system and its framework implementation. These objectives are successfully realized with the establishment of the WSLogA Framework, its unit test harness, and demonstrations within the context of the Adventure Builder system.

Table 1-1. Research objectives and accomplishments.

Objective	Accomplishment
Comprehensive Project System: The establishment of a transaction monitoring engine based on WSLogA architectural principles. Integrates best practices from distributed monitoring solutions. Monitoring and event capture capabilities are extensible to incorporate event data from additional systems.	• The establishment of a framework facilitating development of WSLogA components and their integration into Web service systems and their host environments. • The use of black- and white-box tiers tiers to appropriately hide complexity while facilitating appropriate extension of the framework. • Templated workflows that organize and control the monitoring, analysis, and response processes. • Component packages addressing each of the subsystems identified for the proposal: information collection, event management, event analysis and response, and information presentation. .
Project Subsystem: SOAP Intermediaries for event data capture. Foundational classes or interfaces supply functionality required for all such entities. Specialized classes using this subsystem will be provided as examples for third party developers and will be immediately useful for most production environments as basic data capture services.	• Monitoring, reporting, and recording components with included extensions for the GenericHandler interface, which is provided by both Sun Microsystems and Apache Axis Web service development platforms. • Example extension components made available through unit tests and demonstrations.

Table 1-1. Research objectives and accomplishments.

Objective	Accomplishment
Project Subsystem: A log management framework accepting SOAP intermediary and other source data (*e.g.*, from an Application server's log file). Captured data will be organized in the data repository. Real time and controlled updates will be provided as appropriate to support other system components.	• Monitoring, reporting, and recording components with included extensions for external object observation and data stream parsing (*e.g.*, log files). • Standards based information collection with logging technologies such as Log4J, which permits the immediate integration of systems based on the framework with legacy service environments. • Policy driven information collection, routing, and normalization. For example, sensitive information can be encrypted or buffering mechanisms can be employed to throttle data capture.
Project Subsystem: A data repository supporting the storage and organization needs for the log management, response engine, and presentation frameworks. Anticipated is a multi-component system (*e.g.*, several relational databases or a database combined with XML files) to facilitate phased processing from raw data to highly structured information groups.	• JDOM and XML based information representation and data persistence. • Extensible information model permitting the use of alternate technologies, such as EJBs or JDBC driven SQL statements for high efficiency data transfer. • The use of the HyperSQL relational database (HSQLDB) for event information persistence and unit testing. • Alternate RDBMS technologies, such as Oracle and MySQL, can be substituted for HSQLDB to accommodate established environments.
Project Subsystem: An ETL support engine and framework will facilitate the transfer of data or information from the data repository for use by other system services, such as the presentation and response subsystems.	• The JDOM based information model and persistence subsystem coupled with the RDBMS permits the organized insertion of data for retrieval using perspectives. • Data and metadata associations with event information permit flexible organization of event models by perspective components. Event types can also be related across type domains for flexible integration of information generated by disparate systems.

Table 1-1. Research objectives and accomplishments.

Objective	Accomplishment
Project Subsystem: A presentation framework will support the transfer of information to external systems and users in a variety of formats. Translation and formatting functionality will be extensible to handle third party custom needs. Foundation classes or interfaces supply functionality required for all such entities. Specialized classes using this subsystem will be provided as examples for third party developers and will be immediately useful for most production environments as basic information transfer and monitoring aid services.	• A perspective subsystem accommodates single or multiple queries against the event persistence subsystem. • A daemon is provided to schedule and execute perspective components. • Perspectives can be observed by response components to permit real time or scheduled analysis and response tasks. • Perspectives can also serve information to reporting systems, such as for presentation to administrative, quality control, or engineering staff. • Perspective daemons can be operated as part of the Web service system or as a distinct process interacting with a common data persistence mechanism.
Project Subsystem: A response engine and framework will enable the processing of event data and the execution of environment interaction. Foundation classes or interfaces supply functionality required for all such entities. Specialized classes using this subsystem will be provided as examples for third party developers and will be immediately useful for most production environments as basic environment maintenance services.	• A response subsystem accommodates business logic for analyzing event information and performing environment modifications based on the analysis. • A daemon is provided to schedule and execute response components that have been updated by a perspective. • Response components are managed to prevent redundant scheduling if event information is still being analyzed when a perspective makes new information available. • Response daemons can be operated as part of the Web service system or as a distinct process interacting with a common data transfer mechanism.

Limitations and Delimitations

This investigation did not address performance issues, such as CPU or network bandwidth consumption. Framework performance is best tuned in response to several applica-

tion implementations (D'Souza & Wills, 1998; Richter, 1999), and is beyond the scope of this investigation's goal to create functionality. Information assurance concerns were not addressed, although the provided policy mechanisms can be used to facilitate information assurance or security operations. Production environments encrypt sensitive data transmitted over a public network (such as the Internet), and conceivably transport systems (such as TCP combined with SSH), monitoring filters, or analysis engines can address such concerns. All data used in the research was non-encrypted and of a non-sensitive nature. Components such as an Application server cannot be modified without the participation of product vendors or partner organizations, and such were considered out of scope.

Definition of Terms

Key terms are defined in this section to ensure an appropriate context for the subsequent discussions. Appendix I provides a general glossary.

- *Service oriented architecture (SOA).* An organization of logic or system components to accommodate a standard and modular manner of providing resources.
- *Web service (WS).* A form of SOA intended for business service implementations. Web services use a common communication standard based on SOAP, and they can be assembled from a variety of interoperable platforms (*e.g.*, Java or .NET).
- *Framework.* A partial system implementation controlling workflows within a specific problem domain. White box frameworks expose their implementation to developers for extension. Black box frameworks do not accommodate change or extension within the foundation or core components.

- *WSLogA platform.* The architecture described by Cruz *et al.* (2003, 2004) or likely implementations for the architecture.
- *WSLogA Framework.* The components implemented and bundled as part of this investigation. This term distinguishes what has been made available for use in Web service or other distributed applications from what is possible in terms of implementation strategies for the WSLogA Framework's components.

Summary

Web services provide an infrastructure by which information and products can be distributed, but these systems are complex and can require significant management. Monitoring and analysis tools are needed to facilitate the formation of business and development strategies. These facilities should be implemented as an integral part of the systems they support. Cruz *et al.* (2003, 2004) described a superior alternative to click stream analysis for describing user or system behavior with their introduction of the WSLogA architecture, but their proof of concept does not facilitate component reuse and does not define key mechanisms required for monitoring and response in production grade environments.

This investigation successfully resulted in the creation of a framework facilitating the development of WSLogA components that can be integrated with applications to provide holistic transaction and environment monitoring, management, and communication. The WSLogA Framework improves professional practice by organizing complex information management workflows and tasks for distributed and service-oriented systems in a reusable manner. Researchers may use the WSLogA Framework’s comprehensive information capture, routing, and analysis capabilities to identify data exchange or other behaviors within

distributed systems, including those that span legal or cultural boundaries. The WSLogA Framework also provides a foundation for the investigation of distributed system performance, information assurance, and transaction security.

Chapter 2

Review of the Literature

Historical Overview of the Theory and Research Literature

This investigation involved the design of a framework addressing transaction environment monitoring, response, and communication of state with WSLogA principles guiding the core architecture. The pursuit of such a framework faced several barriers and issues over the course of its design, implementation and testing. Well-balanced frameworks are inherently difficult to design, and the loosely coupled nature of services affects the approach for design aspects such as a framework's inversion of control. The highly distributed environments in which many components operate challenge efficient and effective system monitoring and analysis. The problem domain involves considerations for quality of service, object-oriented development, frameworks and design patterns, service oriented architectures, and information retrieval.

Quality of Service

System integrity and quality of service are critical issues for software development. Software systems are intangible and involve complex interactions between components and the environment. Envisioning how the system's constituent parts will interact with each other and their host environment can be quite difficult, and research continues to explore strategies for discovering system faults and producing easily maintained code in non-conflicting

manners. Quality of service, which in part arises from system integrity, is a fundamental aspect of consumer trust and business growth (Brett, 2004). Comprehending system performance involves application execution analysis and management. Execution tracing is one practical data gathering technique, and numerous trace management systems have been developed with varying capability and intent. The debuggers found in many popular IDEs provide the most common example, but logging APIs and dedicated performance monitoring solutions are increasingly popular among practitioners. Research efforts have started to blur the distinction between monitoring and business system components—the WSLogA architecture offers one example through its use of same-concept components to monitor other system components (*e.g.*, Web services to monitor Web services).

Integrated development environments (IDEs) have become an important solution for ensuring system integrity in the development phases (Boekhoudt, 2003). IDEs offer convenient access to functionality, such as component visualization and debugging, and contemporary IDEs incorporate build standardization using automation scripts provided by tools such as Apache Maven and Ant. The popularity of modeling languages such as UML encourages the development of IDEs, such as OptimalJ, which bring RAD concepts to the code level (Greenfield & Short, 2003).

Programming languages have evolved to ensure quality through the reduced potential for faults. Procedural development strategies permit the isolation of cohesive logic into single functions or themed APIs, but extending a procedural system remains difficult in part due to the ease by which data structures can be duplicated and unintentionally modified in manners that make them unsuitable for continued use in existing portions of the system during the definition of new tasks (Lafore, 2002). Methodologies based on object-oriented

analysis and design have resolved some of these issues by instead focusing on the data aspect of problem domain representation and adding tasks specific to the data's manipulation only as necessary (Richter, 1999).

Traditional debugging strategies remain essential, however, despite the convenience provided by many IDEs (Boekhoudt, 2003; Telles & Hsieh, 2001). Log messages remain a time-honored form of application behavior tracking (execution tracing). In practice, developers insert log messages throughout their source code and monitor the application's output to observe the execution progression. Execution tracing is one of the most useful methods for debugging system behavior (Telles & Hsieh, 2001)—a sentiment supported by the developer community through the creation of popular logging frameworks, such as Apache's Log4J (Gulcu, 2002, 2005), and their adoption into popular products such as the GlassFish Application server. Sun Microsystems acknowledged the usefulness of execution tracing by providing Java developers with a powerful logging API as part of the J2SE SDK (2004, 2001b). The Logger, Handler, and Formatter classes work together to accept message statements, stream message data to a specific repository, and store the data in a specific format (Banes, 2004). Several other projects, such as grid monitoring systems, have since adopted similar implementation patterns (Lee *et al.*, 2002).

Log data can describe almost anything related to a system's state of execution when stored in the proper format (Gulcu, 2002; Telles & Hsieh, 2001). Rosenstein (2000) provided a series of case studies that together demonstrate how Web Server logs can be used to determine visitors and their site navigation habits. Spiliopoulou (2000) mined Web Server logs to evaluate how clients use and perceive web sites, and observed that the server provided a trace of client browsing habits, including the length of time spent at specific pages. Adminis-

trators can use such information to optimize system performance, technical support can learn how to reproduce issues experienced by customers, and architects can evaluate how effectively the web site facilitates the client's information or product needs.

Unfortunately, log data in and of itself is not particularly useful; the volume of data can be prohibitive to analyze (Helsinger *et al.*, 2003; Telles & Hsieh, 2001) when produced by multithreaded applications or environments involving multiple application instances (*e.g.*, distributed systems). Monitoring tools can actively track generated trace data, and analysis tools can filter out irrelevant data or identify system behavior patterns. Several log analyzers specialize around certain types of logs with goals ranging from eliminating system faults to assisting with performance tuning. DevPartner (Compuware, 2005) interacts with the Java Virtual Machine during an instrumented application's execution to obtain a log of application behaviors suggestive of inefficient coding practices or memory leaks. An alternative, Analog, specializes in analyzing Web Server logs (Turner, 2004) and is freely available. Unfortunately, Analog offers only static reports best interpreted by system administrators. Real-time responses require dynamic system analysis, which makes solutions such as Analog ineffective. Barra *et al.* (2002) list several tools that perform similarly with comparable drawbacks. None of these tools use Web service intermediaries as the event-capture technology, or consider targeting SOAP as the primary event data source. Sun offers a programmatic solution through the Java Management Extensions (JMX) specification to the Java core, and its refinement for remote functionality (McManus, 2002; McManus & Vienot, 2003). JMX operates through a tiered approach involving application instrumentation, middleman agent beans, and a console or control system written by third parties to recognize the functionality and data exposed by the beans (Dutta, 2004; Sun, 1999). This strategy abstracts the applica-

tion monitoring and management task from the application suite, and could even, in theory, be used to integrate with non-Java systems using Sun's Java Native Interface (JNI) technology. McGregor (2003) demonstrated how JMX can be integrated with the JUnit testing tool to provide functional checks for a system that could, with only minor enhancements, provide a real-time monitoring and reporting system. Valetto and Kaiser (2003) used JMX to assist with the adaptation of an external monitor and analysis system, but the need to instrument the target system makes their approach impractical for those components not under the control of the interested organization, such as Web services located on third party servers.

Object-Oriented Development

Object-oriented development continues to supplant procedural development as the choice strategy for logic organization, reuse, and easier maintenance. Procedural development considers logic from the perspective of tasks and handles data as necessary to support those tasks. Object-oriented development inverts this perspective by focusing on modeling data first and then adding tasks as necessary to control or communicate data states. Apple Computer and Be found object-oriented development so efficient that their operating systems were designed to ensure all of the functionality is accessible by third parties through object frameworks or APIs (Apple Computer, 2003; Be, 1997).

The data focus and process inversion provided by object-oriented development serves as the foundation for modern frameworks and service-oriented architectures. Classes bearing attributes (instances of data with specific structure and expected behavior) and methods (functions manipulating the entity's state) form the atomic logic entities in languages such as Java and C++. The result is a component that, when instantiated into its runtime object

form, understands what it knows about itself and how its state can be altered into acceptable alternatives. Classes can be defined to hide their inner workings and force all client logic to access or modify the data through one of the class methods (encapsulation).

Classes offer many other advantages that make object-oriented development the ideal foundation for services. Classes can build on each other to provide increasingly specific functionality (inheritance). For example, a basic mammal class could describe the general characteristics of a mammal modeled in the system, and a dog class could be derived from the mammal class. The dog class would receive the mammal characteristics without further work so that it can focus on specializing on attribute, behavior, or state management specific to dogs. Perhaps more interesting is the ability for systems to instantiate objects of the derived class yet reference and operate on the new object using methods or variables of the preceding class' type (polymorphism). In addition to extending classes into more concrete types, the combination of class types as attributes for a new class can create complex components (aggregation and composition). The result is a modeling strategy able to simulate the problem domain in a manner natural for human thinking. The sum of these characteristics ensures that client components are isolated from unnecessary implementation details, and that isolation can make the service system more flexible in terms of fulfilling multiple business needs as well as maintenance adjustments (D'Souza & Wills, 1998; Lafore, 2002; Richter, 1999).

The approach to design, however, has involved a number of strategies (Monarchi & Puhr, 1992). Forerunners such as D'Souza and Wills (1998) conceived of the Catalysis method for identifying component roles and their inter-relationships with other system components. Agile practices such as test-driven development and refactoring are also finding acceptance

(Armitage, 2004). The communication of these designs appears to be solidifying into the UML modeling language, and languages such as these have in turn inspired visual or model driven engineering products such as the Sun ONE Studio (Sturm, 2002), Poseidon, OptimalJ, and Prograph (Greenfield & Short, 2003).

Frameworks and Patterns

The drive to organize object-oriented systems and foster deliberate logic reuse gave rise to the concepts of object frameworks and design patterns (Schmidt *et al.*, 2004). Frameworks are similar to procedural Application Programming Interfaces (APIs), but with the added benefit of strategies such as inversion of control without the need for unwieldy callback functions and memory addressing (Fayad & Schmidt, 1997; Schmidt & Buschmann, 2003; Schmidt *et al.*, 2004). Design patterns codify expert knowledge regarding class relationships and object interactions to permit design reuse (Biljon *et al.*, 2004; Gamma *et al.*, 1994; Shalloway & Trott, 2001).

Frameworks are similar to design patterns as both deal with well-defined roles and relationships, but frameworks provide an implementation whereas patterns only describe such systems. Further, frameworks employ patterns in their design. Developers face the challenge of ensuring that frameworks are designed to meet changing market needs through logical expansion points and careful component relationship architectures (Roberts & Johnson, n.d.). Improvements to the framework must not affect existing systems dependent on the framework's previous API or functionality (Fayad & Schmidt, 1997; Gurp & Bosch, 2001).

Framework quality has unfortunately varied greatly, with solutions of poor quality often arising from architectural oversights or technical limitations. For example, the Microsoft

Foundation Classes (MFC) framework (Microsoft, 2003a, 2003b) provided a limited solution for the reuse of basic client application tasks based on the Win32 API (such as displaying a window). MFC was difficult to extend and covered limited aspects of the Win32 API, and the behavior associated with classes or methods would change over time forcing developers to rewrite dependent logic. Better frameworks are found in examples such as the Java Logging API provided in the J2SE (Sun Microsystems, 2001). The Logging API is comprehensive within its problem domain and is easily extended to handle new scenarios without client rewrites.

Design patterns in software development were first popularized by Gamma *et al.* (1994) as part of their effort to encourage knowledge reuse in manners similar to that in traditional engineering and architecture fields. A formal pattern presents a design specification addressing a scenario (problem domain example); the articulation of component roles, their responsibilities, and their interaction; and potential consequences (positive or negative) arising from the pattern's application to a system. Patterns are available for a multitude of problem domains such as human-computer interaction and e-commerce (Alur *et al.*, 2003; Shalloway & Trott, 2001).

Service Oriented Architectures

Service oriented architectures (SOA) are the natural culmination of object-oriented architectures and distributed systems. Developers need a method by which logic can be organized in a manner that increased task coherence while remaining available across the network for use by other systems (Farrell, 2004), often unknown to the original developer of the SOA component. In this regard, SOAs are macroscopic frameworks for distributed computing.

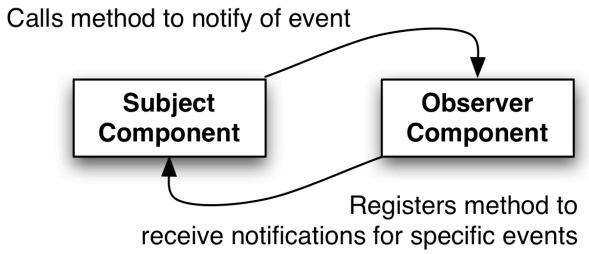


Figure 2-1. The Observer pattern relationship.

The Observer pattern defined by Gamma *et al.* provides a good example of a typical service-client relationship, as illustrated in Figure 2-1.

A client (the *observer*) registers itself with a service component (the *subject*) to receive a callback with data or action instructions. The service component executes the collection of registered callback routines whenever a relevant event occurs. Of course, services can be designed for linear access by a client and without knowledge of the client.

The key aspect of a service is its specialization and general availability to clients (Farrell, 2004; Graham *et al.*, 2005; Iltchenko, 2006). A simple example involves a tax calculator service. A single component can perform the calculation, or the service could be an entire framework with well-defined nodes (hot spots) for the client to extend. The service could be located within the same organization as its clients, or be publicly available over a network such as the Internet for general consumption. Regardless, the service's parts would culminate in the function of calculating tax. A car rental system could use the service to calculate the tax charge applicable to a potential transaction. Figure 2-2 illustrates component interaction of this nature.

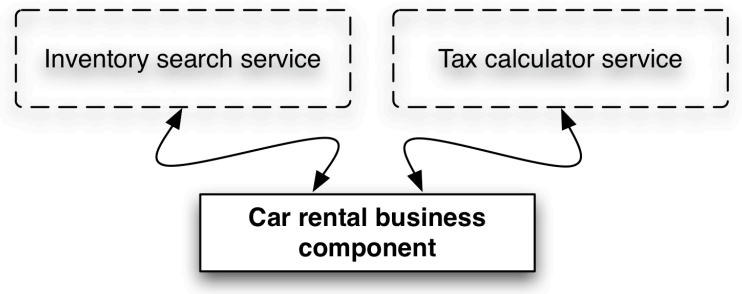


Figure 2-2. A hypothetical service and client interaction.

Although the thought of services often conjures images of business processes, such as the car rental example, many services provide simple backend access to useful resources. For example, Oracle and BEA developed service data objects (SDOs) for the purpose of providing an abstracted method for managing data access (Williams & Daniel, 2004). SDOs offer the advantage of a simple architecture over traditional frameworks such as JDBC (and can even wrap traditional access technologies).

SOAs continue to inspire changes to development methodology (Zimmerman *et al.*, 2004). Object-oriented design focuses on class roles and their relationships; component-oriented design and framework design took form by building on object-oriented methodologies; and Service-oriented design adds to the list of design considerations functional choreography (processes) and business domains.

Web Services

The problem with SOAs is that much of their design and implementation is delegated to the development community. In traditional object-oriented research such standards deficits serves as a strength, but businesses need the confidence of being able to build services that

can be sold or traded with other organizations long after the initial service architectures and platforms are decided upon. Standards ensure a loose coupling between component dependencies and interaction, and Web services provide the standards based solution that businesses can rely upon.

Web services target enterprise architecture concerns by emphasizing business logic availability and component integration across networks with minimal service redundancy (Arsanjani *et al.*, 2003; Graham *et al.*, 2005). For example, one Web service might provide credit information to other Web services specialized in financial matters such as determining mortgage or car loan eligibility. Web services are implemented using many technology platforms and a variety of data package structures and transport layers enable transactions, as illustrated in Figure 2-3. A universal registry to facilitate service interaction is often involved when coordinating Web service discovery or interaction between organizations (Graham *et al.*, 2005).

The most common form of data packaging is the Simple Object Access Protocol (SOAP) (Box *et al.*, 2000; Chavda, 2004; Graham *et al.*, 2005). XML documents specialized for inter-service communication provide the structure for SOAP data. SOAP is a key enabler in Web service technology (Chavda, 2004; Graham *et al.*, 2005; Thai & Lam, 2001) because XML is platform independent (Bray *et al.*, 2004; Stanek, 2002) and SOAP enjoys solid integration with the key e-commerce technology platforms (J2EE and .NET).

An effect of these implementation approaches is the ability for legacy systems to be encapsulated using Web services. Specific business functionality can be exposed for use elsewhere without the immediate need to rewrite the original system (Arsanjani *et al.*, 2003), although conversion work can be subsequently performed.

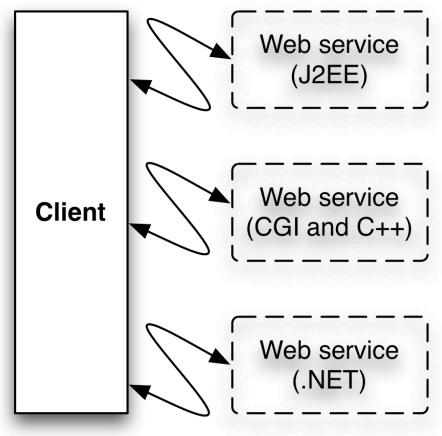


Figure 2-3. Web services are based on a variety of technology platforms.

Information Retrieval

Information management remains a key focus of business technology initiatives, and the retrieval of information is an important aspect of system monitoring, response, and communication. Information retrieval systems must be efficient in their work to locate and retrieve the information requested. A variety of strategies are available for applications to use, with brute force and cataloging techniques both providing examples of currently popular strategies (Singhal, 2001; Tague *et al.*, 1991). Domain specific information pools, such as a J2EE Application server log, can be stored in well-defined forms, with structure and content rules easily enforced by the data persistence mechanisms. Technologies such as XOM and JAXB can conveniently represent and access such data structures when the data is stored in XML form (Fordin, 2004). Web services already utilize such data encapsulation strategies for their communication and processing mechanisms (Graham *et al.*, 2005), and a similar strategy can be applied to other domain specific entities. Previous investigations into error detection and

recovery systems dependent on information retrieval principles pertinent to event analysis can be applied to the event capture and analyzer engine (Brett, 2005).

Theory and Literature Specific to the Study

The problem with traditional log analysis is that the tools are either proprietary systems that integrate poorly with in-house solutions, or that the tools only consider specific types of logs. Application based analyzers are insufficient as solutions when the organization needs to integrate their functionality into custom systems because seamless interoperation would be difficult if not impossible to achieve. Conceivably, multiple tools might be able to read each other's output files in such a manner that the tools could be choreographed for system monitoring, but this approach lacks elegance. Additionally, tools that only consider specific kinds of logs are insufficient solutions because many other environment variables, such as routers or virtual memory usage, can hinder or prevent an application's execution. Distributed systems, such as grids, complicate the situation by creating workflow segments that cannot be directly analyzed (such as systems housed by external organizations) or that result in spliced logs (such as that created by running the same application on different servers).

Service and Web Monitors

Cruz *et al.* (2004) described a Web services architecture, WSLogA, that uses Web service intermediaries to capture data from the SOAP messages between Web service components. WSLogA allows for event data storage and event processing engines, but the architecture remains mostly a definition for the kinds of tools involved in such systems (*e.g.*, Application servers). Cruz *et al.* (2003) demonstrated WSLogA in the form of a reference implementation data mart analysis system for user workflows. The advantages of the WSLogA architecture

include non-invasive monitoring of existing systems (*e.g.*, logging calls are not added to the system source code and binaries are not modified), multiple component monitoring, and monitoring configuration control.

McGregor and Schiefer (2003) recognized that administrators must have a better understanding of how an overall service architecture behaves, and within that perspective developed an approach addressing real time event analysis and performance monitoring components based on principles similar to WSLogA. Just as with WSLogA, McGreger and Schiefer focus on the workflow of the service applications (messages and Web service component activities).

Gombotz and Dustdar (2005) focus their related work on data mining considerations, which positions their work for all types of SOAs. Clickstream data generation through source code modification (*e.g.*, logging) or intermediary components (*e.g.*, servlet chains) provide a degree of transaction detail surpassing that provided exclusively by Application server logs. As with WSLogA, Gombotz and Dustdar focus on the service portion of the environment, although data from the Application server and servlet filters are the primary data sources. Clickstream research has a significant history in the service and business management research fields (Gombotz & Dustdar, 2005; Hu & Zhong, 2005; Rosenstein, 2000; Spiliopoulou, 2000), and the concepts learned for Web Server monitoring and analysis are applicable to an overall architecture such as the framework developed in result of this investigation.

WSLogA and similar architectures do not explicitly address distributed computing issues. Network bandwidth can constrain the type or quantity of data being logged so alternate event or data capture strategies might be required to ensure that the decision system can be adequately primed (Helsinger *et al.*, 2003; Lee *et al.*, 2002). WSLogA does not consider a

variety of other sources for pertinent event information. For example, a bad router or insufficient memory could cause the service system to fail, yet inspection of only the SOAP changes between service components will not reveal that issue. Quality assurance staff could misinform developers of system issues and production support staff could waste time cycling the incorrect system components were they to base their decisions exclusively on the kind of feedback available from WSLogA or similar systems.

Organizations can design their own intermediaries to fulfill architectures such as WSLogA but a common implementation foundation does not exist. A framework would permit the reuse of the WSLogA monitoring concepts (Schmidt *et al.*, 2004) and account for distributed computing issues that need to be addressed by all intermediaries. Java is nearly ubiquitous within enterprise environments, so its use as the implementation platform for such a framework should facilitate adoption and enhancement by organizations. Many of the requisite components envisioned for such a framework are also available in Java forms. The J2SE, J2EE, Log4J, and JBoss Application server are examples of such technologies.

General Monitoring, Analysis and Response

Monitoring activities are concerned with breadth of coverage, appropriateness of presentation, and data collection performance. Even if a monitoring system will not directly present system information, it must still store data representative of the situations witnessed. Analysis activities are concerned with data sources, data correlation, business logic, and performance.

System performance must be captured from end-to-end in order to effectively address the information needs of an organization (Lee *et al.*, 2002). Not all components may be

available for observation or instrumentation due to natural system boundaries, such as sessions and non-controlled third party systems. Events from multiple sources must be correlated in order to provide a holistic view of the system's state (Lee *et al.*, 2002), particularly as processes triggered by monitor data might require several disparate events to occur before execution. Log data might need to be preprocessed (Spiliopoulou, 2000) due to the varied structure of performance logs among applications.

All applications executing on the studied system's host environment consume CPU and memory resources; if not accounted for, CPU and memory usage could distort reports regarding the studied system's performance and result in misdirected maintenance and development efforts (Helsinger *et al.*, 2003; Lee *et al.*, 2002). The monitoring system itself must not excessively steal resources from the host environment. Further, data volumes must be managed without appreciably degrading system performance. Quality of data might need to degrade as volume increases, which could involve the utilization of alternate communication channels and caching techniques to ensure that an appropriate perspective of the system is provided within the capability of the host environment's resources.

Multiple views of the monitoring and analysis data must be accommodated to serve the needs of varying user roles (Barra *et al.*, 2002; Cruz *et al.*, 2004; Lee *et al.*, 2002). For example, system administrators might be interested in the CPU and memory loads for the system, whereas technical support might wish to know the state of user transactions. The framework should provide monitoring functionality accommodating tiered levels of observation trustworthy of producing an accurate, precise representation of the system's execution.

Organizations can use Web services to expose legacy systems (Arsanjani *et al.*, 2003). Source code instrumentation, such as embedding service messages or function calls, is not

possible in these situations. The solution should provide event data input mechanisms that can capture feeds from non-service repositories, and possibly even allow for notes entered by users of the system (such as a comment inserted by an administrator to be associated with an event flow range).

Frameworks

Frameworks rely on the inversion of execution control to coordinate component activities (Fayad & Schmidt, 1997; Schmidt & Buschmann, 2003), yet Web services are, by nature, loosely coupled with operations triggered by message events (Graham *et al.*, 2005). The framework design must carefully consider asynchronous event management and transaction requests, such as those established *ad hoc* via service registries, unless service composition rules such as BPEL4WS (Milanovic & Malek, 2004) are employed.

Encryption and decryption functionality might need to be provided or accommodated so Web services can deal with sensitive data (Arsanjani *et al.*, 2003). Event correlation engines must allow for integration with auditing and cryptography solutions so that analysis logic can access event data contents, otherwise in secure environments only the event type would be visible. The monitoring and analysis components themselves must also be auditable to ensure their own proper behavior (Arsanjani *et al.*, 2003).

Summary of Prior Research

Logging and associated practices remain an accepted and encouraged method for improving a system's integrity and quality of service. Logging provides insight into a system's behavior, but to be effective all logs within the environment need to be taken into account. Unfortunately, existing tools are proprietary, focus on a limited range of log structures, or do

not integrate well into custom solutions. Technologies that permit the construction of custom solutions, such as JMX, require modifications to the sources or artifacts, and that practice is not an option for external systems. Multiple staff roles might be involved in properly translating tool results into information meaningful to all interested parties, such as development teams, production support, and executives.

The acquisition of data needs to be thorough but many challenges must be overcome in distributed systems to ensure that appropriate collection occurs. Applications running on different servers, multithreaded systems or those with multiple simultaneous sessions, network performance, and systems not controlled by or visible to the interested organization are just a few factors that can impede data collection. The data store or retrieval mechanism must properly sequence collected data. Real time analysis or communication of the data must account for non-temporal data entries even after those processes begin.

Object-oriented designs and implementations provide good foundations for service-oriented architectures due to the methodology's perspective of data definition and management. Frameworks serve to organize logic into reusable solutions that can reduce knowledge requirements and workloads by third parties dependent on the functionality provided by frameworks. Web services challenge framework development because of their reliance on loose communication and interaction coupling strategies.

Information retrieval was an essential aspect of the framework. Many strategies exist for developing a backend information retrieval system, but the strategy selected must complement the data storage and data manipulation technologies adopted by the framework. For example, JAXB would be appropriate for interacting with XML based storage solutions, but a MySQL RDBMS could be more efficient for storing large volumes of data.

Web services provide businesses with a standardized means to integrate operations through technology. WSLogA and parallel architectures have demonstrated data collection for workflow and user behavior analysis, but a reusable implementation for disparate organizations is not yet available. Such architectures are ineffective in production environments without obtaining a holistic perspective of the system's behavior and state.

Contribution of this Study

The pursuit of a Web services framework for system monitoring and analysis faced several barriers and issues over the course of design, implementation, and testing. Well-balanced frameworks are inherently difficult to design, and the loosely coupled nature of Web services certainly affects how design aspects such as a framework's inversion of control must be approached. The highly distributed environments in which many functional components operate also challenge efficient and effective system monitoring and analysis. WSLogA is improved by the availability of a framework facilitating the development of components intended for WSLogA environments.

This investigation explored framework development for Web services and provides an understanding of how loose component and communication coupling can be best addressed through inversion of control strategies. Highly customizable monitoring and analysis strategies for transaction environments are documented through detailed designs, implementation, and test analysis. The expert knowledge gained regarding the problem domain and its solution is reusable by practitioners through the availability of the framework. WSLogA is improved through the availability of a software development kit based on its architectural principles and server distribution requirements.

Chapter 3

Methodology

Research Methods Employed

The information systems design science research framework proposed by Hevner *et al.* (2004) and the principles of design research as observed by the AIS (2005) guided this investigation's methodology. Artifacts were developed and analyzed for their suitability as a solution to the problem domain in a rigorous, iterative fashion organized using the spiral software development lifecycle (Schach, 2002). Systems analysis practices such as scenario mapping through use cases (Whitten *et al.*, 2001) and object-oriented techniques such as role-based design (D'Souza & Wills, 1998; Richter, 1999) served as the approach for engineering the artifact's functionality and organization. Analysis of an evolving artifact's behavior through the application of tests resulted in an improved understanding of the problem domain (Louridas & Loucopoulos, 2000), and the artifact's validated design codifies the developing theory (AIS, 2005). This active reflection drove the artifact's iterative evolution as it is formed into a suitable solution (Hevner *et al.*, 2004; Whitten *et al.*, 2001).

Specific Procedures Employed

This investigation emphasized the development of an artifact design incorporating lessons learned from the artifact's implementation and exploration within an environment representative of the problem domain (AIS, 2005; Hevner *et al.*, 2004). The design's assertions

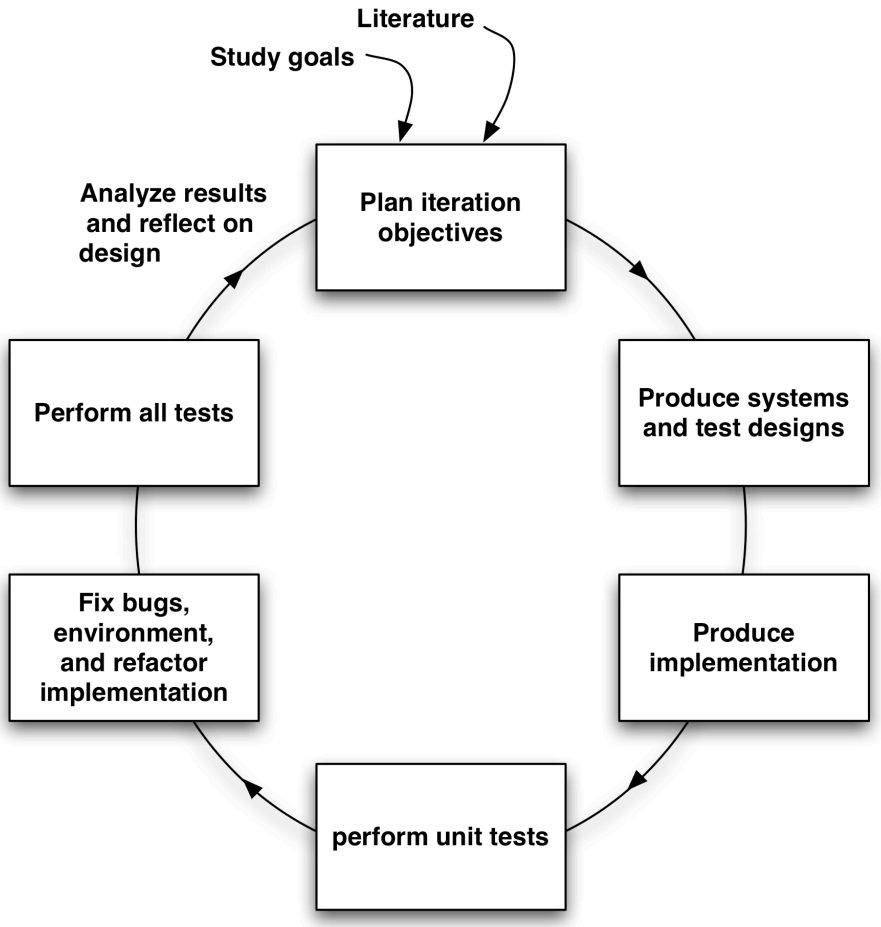


Figure 3-1. The organization of investigation activities around the spiral lifecycle.

and assumptions were validated through the creation of the artifact's implementation in the form of the WSLogA Framework, as well as its exercise using automated and manual processes (Appendix A; Appendix F). The lessons learned from the implementation and test analysis served as the basis for subsequent designs (Edwards, 2004; Hevner *et al.*, 2004). Configuration and automation strategies ensured consistency between iteration activities such as regression tests. Figure 3-1 depicts the organization of activities within the iterations.

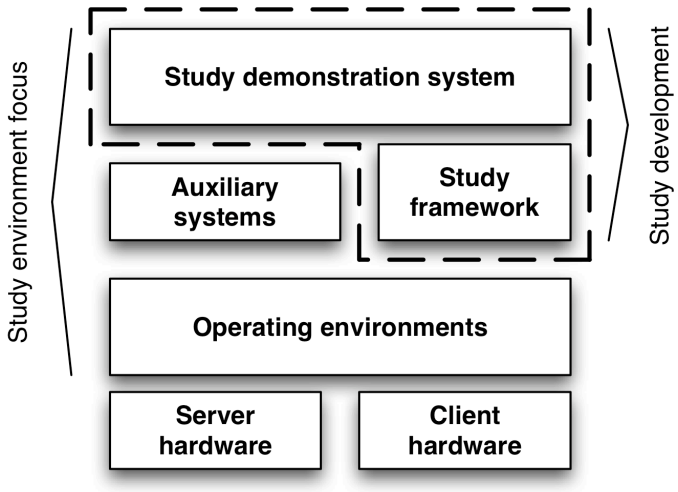


Figure 3-2. Emphasis of this investigation's work and analysis.

Iterations started with the formation of objectives divided between environment, design, and test strategies. Environment considerations included the establishment of the Application server and logging systems to simulate the problem domain by means of a simplified representation (Appendix B; Appendix C). Design considerations included the specification of the artifact's behavior as comprised by assertions and assumptions formulated using the body of literature, lessons learned from previous iterations, and the investigation's goals. Test considerations included the configuration and execution of scenarios within the prepared environment appropriate to the problem domain. The artifact's design was considered valid when its implementation successfully addressed the test objectives. Figure 3-2 depicts the emphasis of work and analysis for the proposed investigation.

Several design documents were prepared each iteration. The interaction between participants within the problem domain (a scenario) were described using requirements and use cases (D'Souza & Wills, 1998; Richter, 1999; Whitten *et al.*, 2001) (Appendix H). Each use case

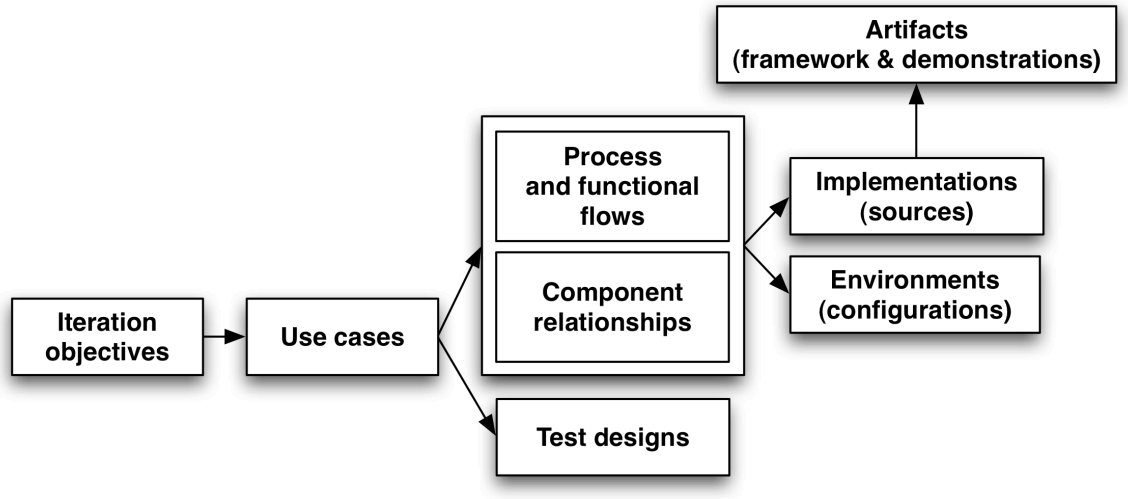


Figure 3-3. Transitioning from objectives to implementation.

contained a nominal flow (depicting ideal or likely events) and, as appropriate, alternate flows deemed significant (situations resulting in a fault). Class diagrams, object interaction diagrams, and process or functional flows specifying the organization and behavior of the artifact's components were prepared from the use cases to guide the artifact's implementation (D'Souza & Wills, 1998; Richter, 1999) (Chapter 4). The use cases also defined the scope and activities of tests (Appendix A). Figure 3-3 depicts the transition from iteration objectives to the artifact's implementation.

Both the automated and manual tests validated the implementation. Both types of tests adhered to data and event scripts based on associated use cases, and the results for both types of tests were documented using the methods specified in Appendix A. Automated tests were conducted through JUnit implementations, and as such were executed every iteration to ensure proper regression testing (Staff & Ernst, 2004a). Automated tests were executed

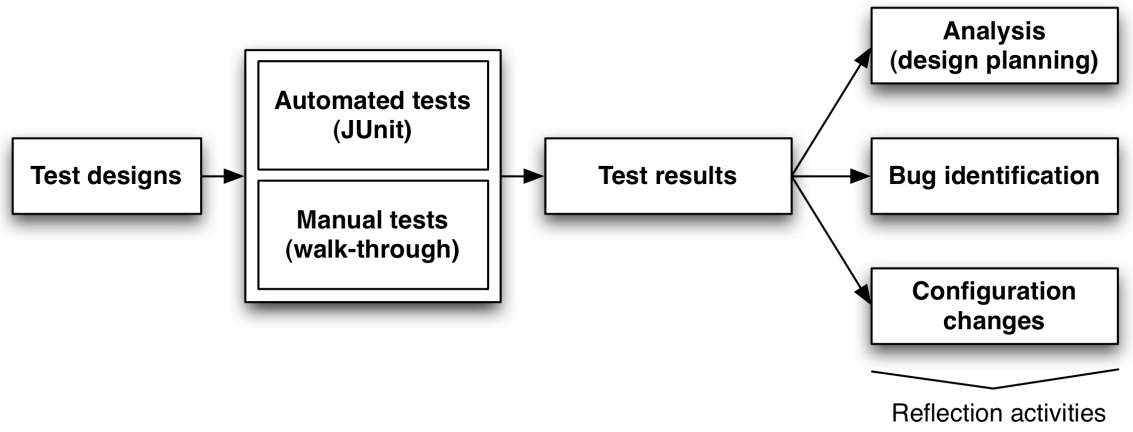


Figure 3-4. Transitioning from test designs to results analysis.

after each implementation attempt to identify bugs, and executed again in conjunction with the manual tests after bug fixes to obtain results for analysis (Figure 3-1). Analysis of the test results provided insights for subsequent iteration designs, and identified aspects of the environment requiring improvement to better simulate the problem domain (Edwards, 2004; Hevner *et al.*, 2004; Maximilien & Williams, 2003). Figure 3-4 depicts the transition from test designs to the analysis of test results.

This investigation's resultant artifacts are comprised of development and runtime frameworks. As such, object-oriented development and framework development techniques (Cortes *et al.*, 2003; D'Souza & Wills, 1998; Gamma *et al.*, 1994; Richter, 1999; Whitten *et al.*, 2001) were key considerations throughout the design process. Both white- and black-box framework architectures (Richter, 1999) were acceptable, and the WSLogA Framework is a combination of both types.

Formats for Presenting Results

The results for this investigation demonstrate the developed artifact's suitability as a solution to the problem domain. Evidence includes the design documents, implementation sources, implementation artifacts, test scripts and the result analysis, and environment configurations. Discussed in Chapters 4 and 5 are those aspects of the documents relevant to the outcome and significant issues that challenged the investigation's activities. A DVD image containing the full set of documents generated by the investigation was made available in association with the final report.

Discussions regarding scenarios and the artifact architecture incorporate the UML modeling language (OMG, 2001, 2006), with the emphasis on use case, activity, class, and sequence diagrams. The UML modeling language does not illustrate all key framework design issues, such as significant expansion points for use by third parties (hot spots), and for these situations the consistent use of alternate diagram strategies was substituted. Discussions pertaining to the artifact's design focus on class diagrams and their associated use cases, other design documents, and significant implementations.

The efficacy of the design was demonstrated using Java source code and compiled binaries. Both sources and binaries were made available (Appendix C), but only source code segments pertinent to a discussion are included in this report. In situations where the audience's comprehension could be improved a source segment was presented to provide the discussion's context.

Validation of the working model's efficacy was accomplished using test and data scripts. Each script is described using a form detailing information pertinent to the reproduction of

the test, and automated tests include JUnit source code and binaries based on the script. Appendixes A and C discuss the manner by which tests were prepared or analyzed.

Resource Requirements

Design, development, and test tools were used throughout the investigation (Appendix D), along with a demonstration environment simulating the problem domain (Appendix C). Hardware and software configurations remained consistent throughout this investigation except where changes were warranted because of bugs interfering with the research. The hardware and software utilized were selected based on their suitability to the problem domain and general availability to other researchers or practitioners.

The significant software tools can be broken into design, implementation, test, configuration, and automation categories. Table 3-1 summarizes the significant technologies utilized and Appendix D discusses their configuration. Auxiliary tools such as Microsoft Office are assumed. Platforms and operating environments typical of Web service environments were involved throughout the implementation and test activities. An Intel based Macintosh was used because Mac OS X provides a representative UNIX environment in the form of Berkley UNIX and a Windows environment by means of the VMware 80x86 virtualization software. GlassFish was used for the application server due to its role in demonstrating Java technology in a variety of Sun Microsystems certification courses, as well as its native support for key Web service technologies supporting the J2EE SDK. Table 3-2 describes the platforms and operating environments used.

Table 3-1. Significant tools, platforms, and environments.

Tool	Purpose	Source
OmniGraffle	UML and other documentation	Omni Group www.omnigroup.com
Eclipse	Source code implementation and binary generation	Eclipse Foundation www.eclipse.org
Maven 2	Binary generation, test execution, data management, and environment configuration automation	Apache Software Foundation maven.apache.org
JUnit	Test automation	Open source community junit.sourceforge.net
Subversion	Version control and configuration of investigation documents	Open source community Bundled with Mac OS X
Java 1.5 (J2SE) SDK	Source code implementation and binary generation	Sun Microsystems, Inc. java.sun.com/j2se/1.5.0/
J2SE 1.4 SDK	Source code implementation	Sun Microsystems, Inc. java.sun.com/j2ee/1.4/
Java Web services Development Pack	Source code implementation	Sun Microsystems, Inc. java.sun.com/webservices/
Log4J	Source code implementation and testing	Apache Software Foundation logging.apache.org/log4j/
Mac OS X	Design, implementation, and testing	Apple Computer, Inc. www.apple.com
VMware Fusion and WindowsXP	Testing and demonstrations	VMware, Inc. www.vmware.com Microsoft Corporation www.microsoft.com
GlassFish	Testing	Sun Microsystems, Inc. GlassFish.dev.java.net

Reliability and Validity

The consistent approach to configuration, automation, data management, and test strategies ensured this investigation's reliability and validity. The availability of the software components, including both artifacts and environments, should facilitate the reproduction of the investigation's results by other researchers and practitioners.

This investigation organized its activities within an agile (Berczuk & Appleton, 2003; Conboy & Fitzgerald, 2004; Highsmith, 2002) form of the spiral lifecycle (Schach, 2002). This strategy facilitated and encouraged the consideration of design, implementation, and quality assurance efforts or artifacts necessary for developmental research (AIS, 2005; Hazzan & Dubinsky, 2007; Hevner *et al.*, 2004), yet emphasized framework design exploration through continual testing, refactoring, and integration (Fowler, 2006; Garsombke, 2003).

Design research methodology depends on the researcher's ability to reflect on the results of each iteration's events and outputs (Hevner *et al.*, 2004). This investigation integrated testing and analysis as key activities that preceded design and followed implementation. In this manner, feedback regarding the design's efficacy through validation of the working model was obtained at regular interviews throughout the iterations. Automated testing is advocated by researchers and practitioners alike for its ability to facilitate the accurate execution and consistent reproduction of test steps, as well as the active discovery of artifact faults (Cortes *et al.*, 2003; Edwards, 2004; Maximilien & Williams, 2003; Telles & Hsieh, 2001). JUnit is an effective test tool (Gaffney *et al.*, 2004; Louridas, 2005; Olan, 2003; Wick *et al.*, 2005), and because JUnit tests are implemented using Java and related technologies many of the investigation's tests will be conveniently reproducible for either validation of the investigation or, to varying degrees, comparison against similar studies.

The Subversion version control system (Appendix E) was used to track and organize documents generated by the investigation's activities. Comparisons can be made against text document changes to facilitate analysis. Environment configurations were documented to facilitate precise recreations, and automation tools were employed to ensure the consistent execution of build, data management, and JUnit test processes. The consistency between iterations for these activities permitted appropriate comparisons during result analysis and regression testing. The strategy of using automated tests also ensured the consistency for manual evaluations of the implementation and environment.

Summary

This investigation used design research methodology combined with the spiral lifecycle to iteratively investigate the resultant artifact's design, implementation, testing, and result analysis—a technique known as reflection. Framework and object-oriented design practices formed the foundation for the preparation of the artifact's component organization and relationships. Automated and manual testing of the artifact within a carefully configured environment facilitated reflection, ensured rigor, and enables result comparisons.

Chapter 4

Results

Findings

This investigation resulted in the successful realization of all research objectives with the establishment of the WSLogA Framework. The WSLogA Framework serves as a platform for enabling the holistic monitoring, analysis, and response tasks required to ensure the robust operation of Web service based Enterprise systems. The WSLogA Framework's core functionality is based on the principles of the WSLogA system described by Cruz *et al.* (2003, 2004), and significantly extends that platform by incorporating a policy based information collection facility; an event information processing and analysis engine; and an event response system with environment management capabilities. The WSLogA Framework provides implementations for best practices addressing Web service transaction monitoring and an application's management of its environment in response to related events.

The WSLogA Framework can be extended to capture and provide information regarding the activities of Web services, their transactions, and their host environments. The information aggregated for analysis is organized around the content of SOAP messages, and supplementary information may be collected based on observations of the application's components or related resources (*e.g.*, the application server or log files). The WSLogA Framework

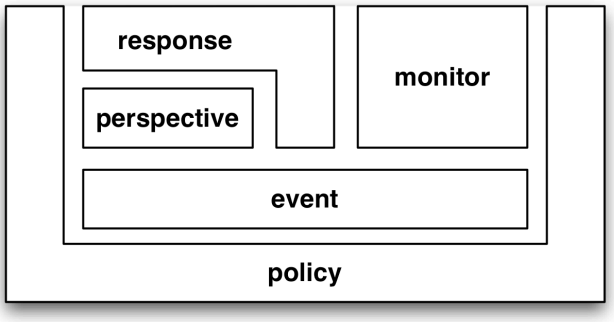


Figure 4-1. The WSLogA Framework's component groups.

makes the event information available to components that can analyze the events and in response manage the Enterprise environment to ensure its continued operation.

Practitioners can use the WSLogA Framework to implement Web service based systems integrating monitoring, analysis, and response functionality that holistically considers both transaction data, represented by SOAP messages or application objects, and environment data, such as network logs. Researchers can use the WSLogA Framework to understand the flow of information within service oriented distributed or parallel applications, as well as explore the information assurance concerns regarding processing points within the system for scenarios involving components operating across disparate hosts or processing regions.

The WSLogA Framework Platform

The WSLogA Framework is a modular software development kit in the form of a framework. Component groups within the WSLogA Framework are designed to provide integrated workflow support with functionally related groups (Figure 4-1), but most components may also be independently integrated into an application to support a phased adoption of the WSLogA Framework. For example, the Policy Group components may be of use to any

Enterprise system requiring post-deployment management of information normalization procedures (*e.g.*, the manner by which social security numbers are formatted before being committed to a log). Table 4-1 summarizes how each group corresponds with the functionality envisioned for the WSLogA Framework.

Table 4-1. Required component groups and corresponding facilities.

Functionality	Corresponding Facility	Objective
Information Capture An information capture and routing subsystem by which SOAP message, object attribute, environment logs, user input, and other information sources can be accessed, normalized, and channeled for use by framework components.	• The *Monitor Component Group* provides information capture, routing, and normalization capabilities. • Applications can be created that extend the Monitor Group's components for integrated and native information management, and legacy systems can be wrapped or observed using these components to contribute information for event realization and analysis.	Yes
Event Management An information modeling and persistence subsystem that handles the organization and transport of event information for use by framework components.	• The *Event Component Group* models event related information and manages the transport of event information between framework components and a persistent storage platform, such as a database.	Yes
Information Presentation An information normalization and distribution subsystem that facilitates the routing of event information for use by analysis or reporting systems.	• The *Perspective Component Group* provides query based access to the event information managed by the Event Group. • Information normalization may be performed by this group to maximize the framework's integration with external systems.	Yes

Table 4-1. Required component groups and corresponding facilities.

Functionality	Corresponding Facility	Objective
Event Response An information analysis and environment management subsystem that facilitates the execution of business rules intended to communicate system state or behavior, as well as to make environment adjustments to ensure the continued operation of the application.	• The *Response Component Group* facilitates event information analysis, correlation, and environment management. • Response tasks are scheduled as the result of information generated by components from the Perspective Group to ensure the organized handling of events as they are realized by the system.	Yes
Policy Management A behavior management subsystem by which business rules may influence the behavior of other framework components, such as the manner by which the Monitor Group formats data during the normalization process.	• The *Policy Component Group* facilitates event information analysis, correlation, and environment management. • Established in the recognition that information normalization by the Monitor Group, and possibly other framework or application components, required the flexibility to handle legal or cultural requirements that were not consistent across system hosts.	No

Applicability of the WSLogA Framework to Enterprise Environments

Enterprise environments can involve complex compositions of application servers, data stores, message transports, and operating hosts interacting in manners not necessarily clear in terms of significant contact points or outcomes (Telles & Hsieh, 2001; Whitten *et al.*, 2001). Web services are inherently subject to these complexities yet their quality of service is dependent on the development and support teams' comprehension of these interactions.

Cruz *et al.* (2003, 2004) described an architecture, the WSLogA, with the capability of monitoring Web service components by means of simple service probes and the capture of SOAP message information. This investigation sought to produce a design, demonstrated by a Java based implementation, which addresses the WSLogA's principal concerns and enhances the WSLogA by introducing holistic information collection and environment response capabilities. The design succeeds by dividing the responsibility of the sought functionality into modules accommodating environment management through the use of information capture, routing, persistence, retrieval, and analysis functionality (Table 4-1). The information collection capabilities augment SOAP message inspection with integration points provided for logging systems and ad hoc system elements.

The WSLogA Framework is implemented using the Java language and related technology platforms, but the design is generally compliant with the requirements for a variety of contemporary software development languages. Microsoft's .NET platform (Telles, 2001; Thai & Lam, 2001) provides the C# language, which reproduces Java functionality relevant to the WSLogA Framework—SOAP transaction management, object-relational mapping, and support for dynamic, pluggable components within runtime environments (required for select policy management strategies as discussed later in this chapter). Reporting solutions such as Crystal Reports (Business Objects, 2008) or Cognos (Cognos, 2008) can be substituted for modules such as the Response Group.

The WSLogA Framework is intended to support Enterprise systems involving SOAP transactions by adding information capture and environment response capabilities with minimal modification to logic implementing business rules. For example, a SOAP message monitor can be added to a SOAP handler chain with only configuration changes to the

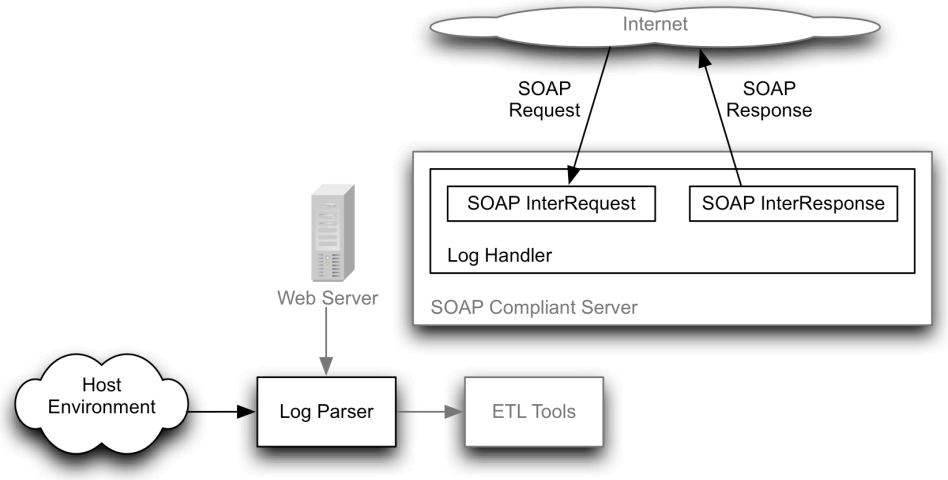


Figure 4-2. WSLogA elements addressed by the Monitor Group.

affected module's configuration file (Graham *et al.*, 2005). Log messages produced by the Log4J framework (Gulcu, 2002) and J2SE Logging API (Arnold *et al.*, 2005; Sun Microsystems, 2001) can be captured and combined with SOAP information to provide context for SOAP analysis (Telles & Hsieh, 2001). Event information correlation is delegated to components operating outside of the application (but potentially within the same JVM). Many WSLogA Framework components integrate policy managed logic to permit flexible information management and event processing within the same application architecture. A common data model was established to organize and correlate event information for application or environment sources, including sources operating across different machine or process boundaries.

Demonstrations exercising important WSLogA Framework components within the context of the Adventure Builder application (Appendix B), which uses Web service components, are provided to facilitate continued research and adoption of the WSLogA Framework

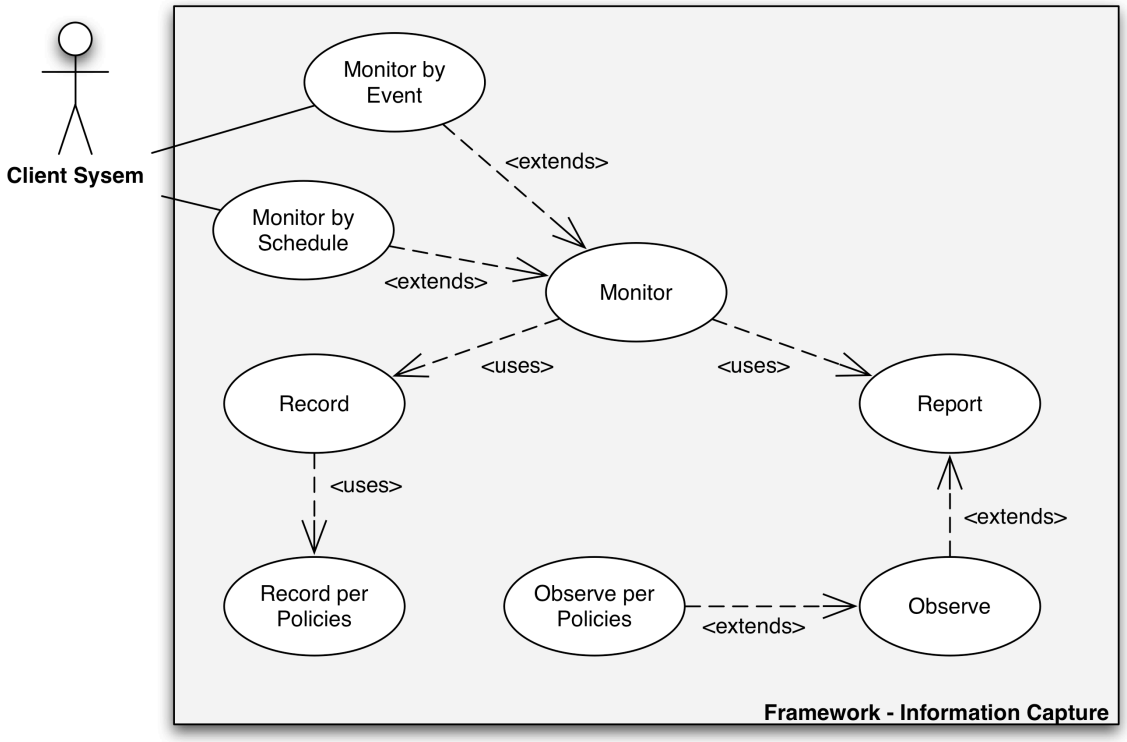


Figure 4-3. Use cases applicable to the Monitor Group.

(Appendix C). A complete implementation of the WSLogA Framework using Java and supporting technologies, JavaDoc documentation, component and system diagrams, and an extensive test suite featuring both unit and integration test contexts (Appendix A) has been made available (Chapter 3).

The Monitor Component Group

The *Monitor Group* is comprised of those components that report and record information related to the Web services, their transactions, or related environment information, as well as those components that organize information collection and routing processes. The

interfaces, classes, and resources for this group are defined within the *org.ws.loga.monitor* package. Figure 4-2 illustrates those portions of the WSLogA platform that are addressed by members of the Monitor Group with grey elements indicating boundary components. Figure 4-3 illustrates use cases embodying these workflows. Appendix H documents the activities associated with each use case.

Roles and Responsibilities

Five information collection and routing roles were envisioned for the Monitor Group. Reporter components describe events and objects, and Recorder components route the descriptions to consumers (*e.g.*, the Event Group). A Monitor component coordinates Reporters and Recorders for situations in which strong relationships exist, such as with SOAP intermediaries (Chavda, 2004; Graham *et al.*, 2005) or related runtime objects. Figure 4-4 illustrates the Monitor Group component roles and their relationships.

The Reporter describes events and objects with significant meaning to the application and its environment. Reporters may consider multiple characteristics of an event or object context before creating a report, and the manner by which information is transcribed into the report may be influenced by active policies associated with the Reporter.

The Subject represents events, runtime objects, system resources, and their contexts. The Subject is the focus of the Monitor Group but has no implementation because facilities such as Java 1.5's generics (Arnold *et al.*, 2005) are assumed to provide suitable mechanisms by which Subjects can be exposed to Monitor Group components. The Recorder routes information generated by Reporters to appropriate consumers. The WSLogA Framework manifests Event Group components as the consumer of the information, but alternate

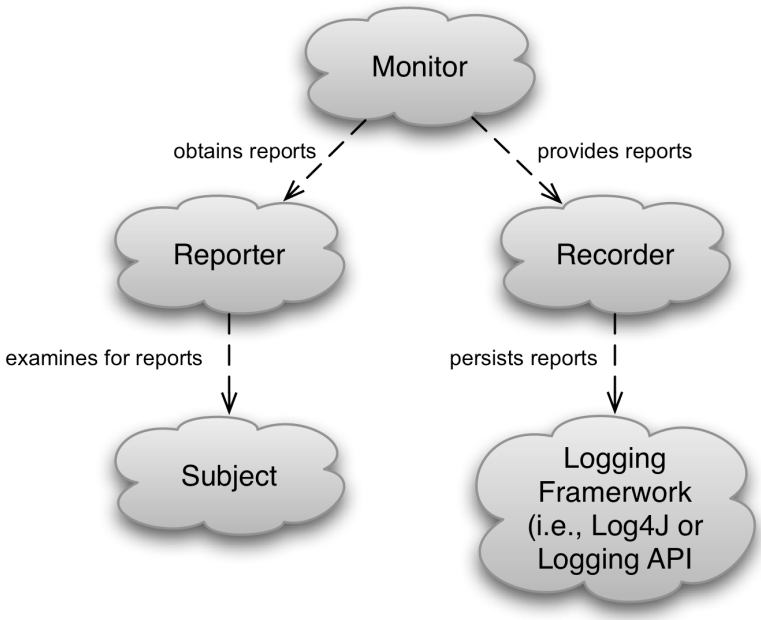


Figure 4-4. Monitor Group component roles.

consumers such as sockets to external systems can also be established as Recorders. Complementary technologies, such as the Log4J framework (Gulcu, 2002), may be used for the transport mechanisms in some Recorders to reduce the learning curve of engineers extending the monitor or Event Group services.

Structure

The Monitor Group is organized around a generic Monitor component intended to accommodate unique system requirements, as well as three platform specific Monitors that respectively address SOAP transactions, Log4J events (Gupta, 2003), and JDK Logging API events (Gupta, 2003). The Reporter component is provided for data calculation, and it is extended by an Observer component to acquire data provided by a variable Subject. The Inspector component complements the Recorder family by facilitating detailed Subject

analysis. The Recorder component is provided to route data to consuming systems, such as another Web service or a relational database management system. Figure 4-5 illustrates the structure of the Monitor Group's components.

Reporter is an interface that represents a point from which the Monitor Group may access event data regarding the monitored Enterprise system. Reporter is generic enough to represent both a calculation (*e.g.*, a summary value representing more complex relationships) and a data acquisition (*e.g.*, a file or object attribute value). The Observer abstract class implements Reporter to make explicit the task of acquiring information from a definable Subject, such as a runtime object or environment service. The Inspector interface is provided as an analytical assistant for the Reporter component family. Inspectors are expected to assess to some degree of detail, possibly using calculations, information regarding a specific Subject set and then prepare a report using that information.

Recorder is an interface that represents a data consumer, or at least the entry point for moving data to a consumer set. Recorders do not perform data analysis, but they may filter information contained within provided reports prior to submitting the report to a consumer, such as to ensure security obligations are met or to maximize bandwidth efficiency.

Implementation

The Monitor Group is implemented as three packages addressing monitoring, report generation, and report routing. All of the components have definitions that control the workflows necessary to facilitate information acquisition and management, and several specialized components are provided with integrated support for policies that control information acceptance or formatting. Applications only need to instantiate a specialized

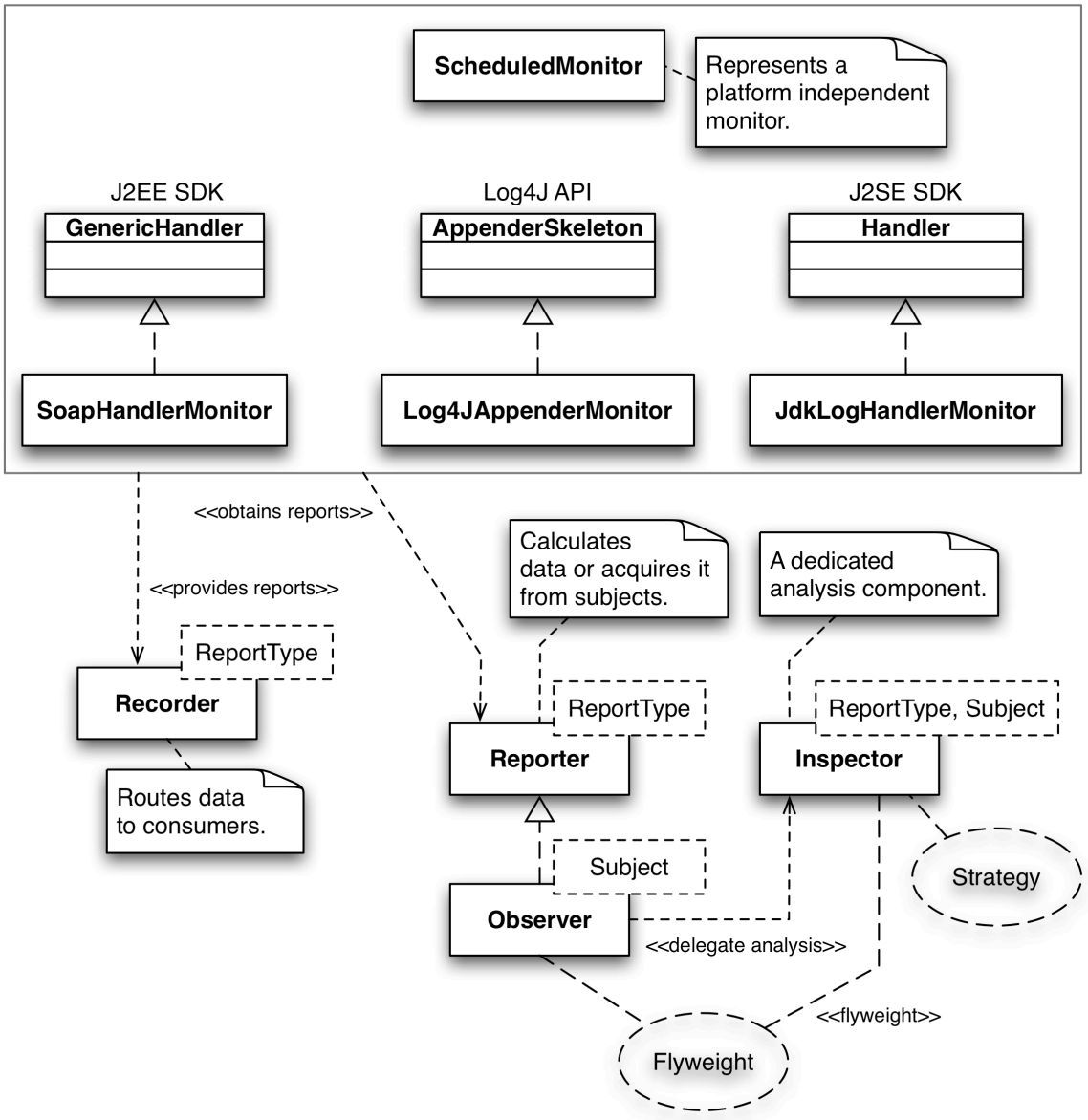


Figure 4-5. Monitor Group component structure.

Reporter to take advantage of the provided workflow, but customization for component or environment monitoring and information routing is also supported.

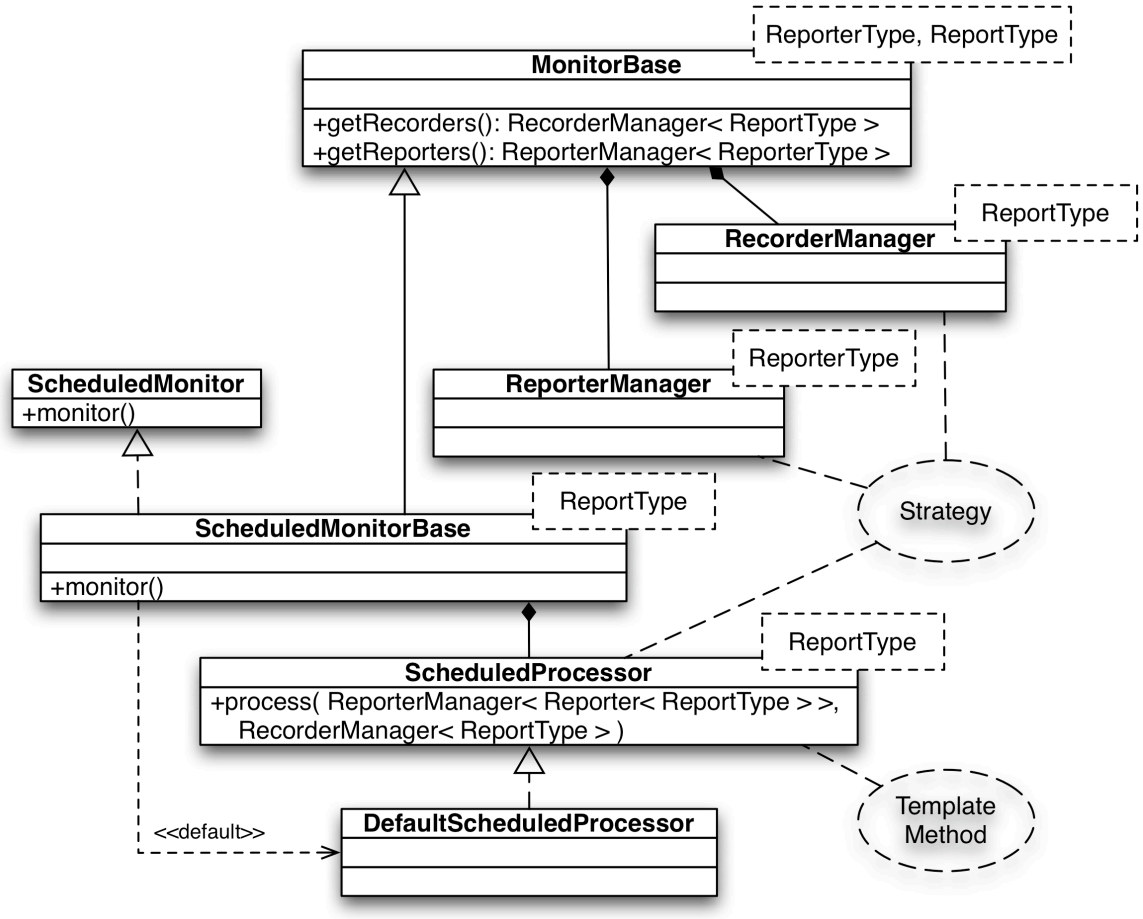


Figure 4-6. The ScheduledMonitorBase.

General monitoring functionality is provided by the ScheduledMonitorBase component family, illustrated in Figure 4-6, which is located at the root package, *org.wsloga.monitor*. ScheduledMonitorBase extends MonitorBase and implements the ScheduledMonitor interface to mark the component as being a Monitor. MonitorBase is a generic abstract class that provides management for Reporter and Recorder components addressing common report themes. ScheduledMonitorBase organizes information produced by Reporters and exchanges the information with Recorders using a ScheduledProcessor derivative. De-

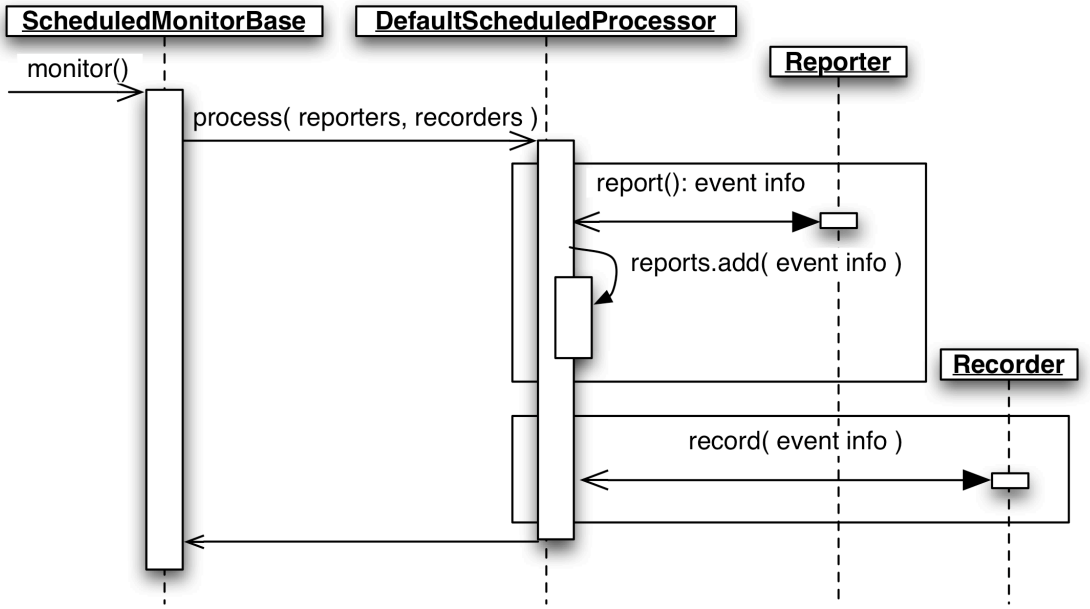


Figure 4-7. ScheduledMonitorBase delegates monitoring to ScheduledProcessor.

faultScheduledProcessor is assigned to ScheduledMonitorBase instances when the strategic delegate (Gamma *et al.*, 1994) is unspecified. DefaultScheduledProcessor iteratively obtains reports from associated Reporters and provides those reports to associated Recorders for routing. The Monitor Group does not provide scheduling capability because such functionality is addressed by external projects, such as Quartz (Cavaness, 2006), but third parties can derive monitors from ScheduledMonitor to organize monitoring activities.

Figure 4-7 illustrates the standard workflow for ScheduledMonitorBase as coupled with the DefaultScheduledProcessor. Monitor event management is delegated to the ScheduledProcessor implementation, and DefaultScheduledProcessor responds by acquiring a report from each registered Reporter as appropriate and then passes the reports to each registered Recorder for routing.

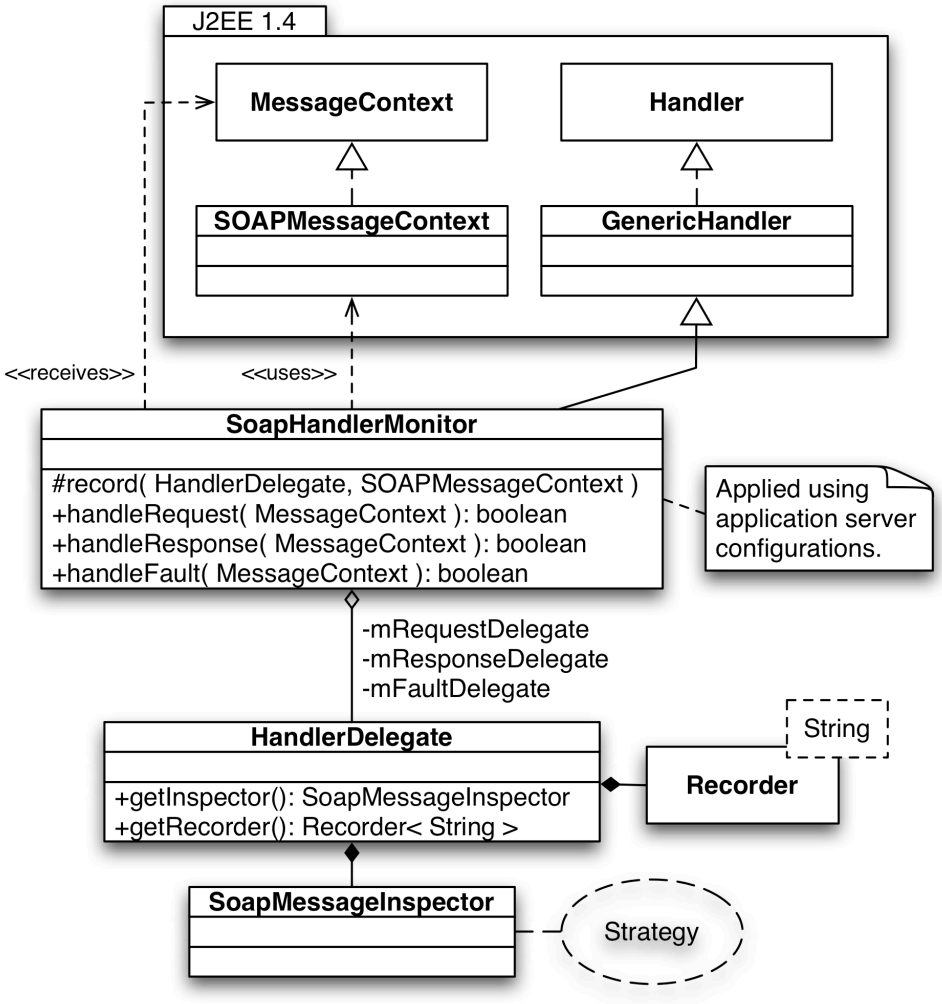


Figure 4-8. The SoapHandlerMonitor.

The SoapHandlerMonitor component is used to monitor SOAP transactions. SoapHandlerMonitor extends GenericHandler (Singh *et al.*, 2004), which provides a default implementation of the Handler (Graham *et al.*, 2005; Singh *et al.*, 2004) interface intended for intercepting and processing SOAP messages traveling through J2EE or Axis managed application servers. SOAP messages are platform independent (Bray *et al.*, 2004; Singh *et al.*, 2004; Stanek, 2002) so SoapHandlerMonitor is able to address SOAP information regardless of the

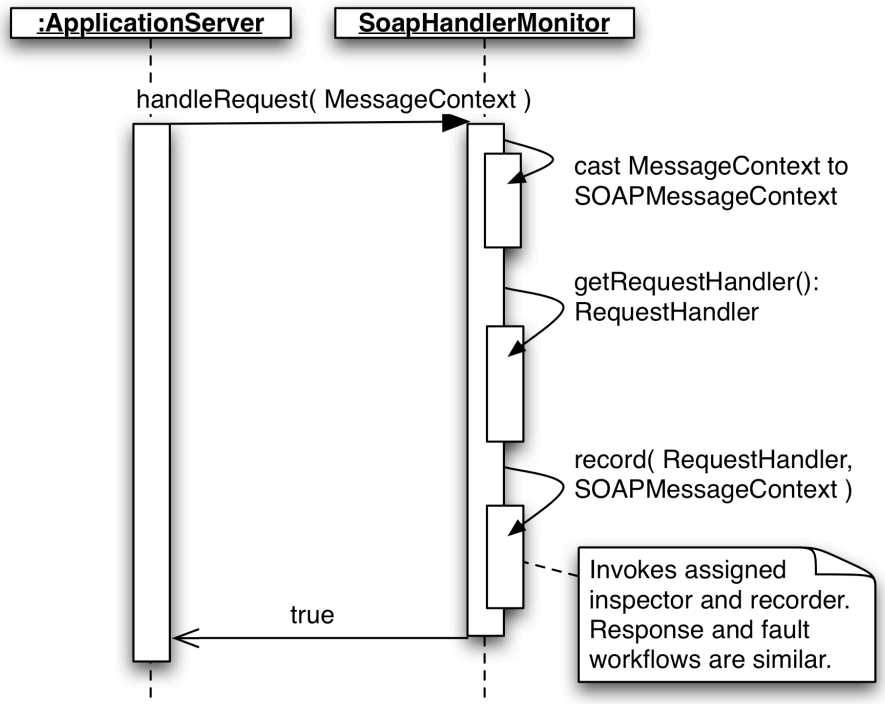


Figure 4-9. SoapHandlerMonitor is integrated into SOAP transactions.

source technology platform. Figure 4-8 illustrates the SoapHandlerMonitor and its associated components.

Handler declares three message processing methods—handleRequest, handleResponse, and handleFault—that are invoked by the application server and provided with instances of MessageContext (Graham *et al.*, 2005; Singh *et al.*, 2004). In the case of a SOAP based system, the provided MessageContext object is actually a SOAPMessageContext (Graham *et al.*, 2005; Singh *et al.*, 2004), which is extracted by SoapHandlerMonitor and delegated to instances of HandlerDelegate registered with the SoapHandlerMonitor instance. HandlerDelegate is a class internal to SoapHandlerMonitor that coordinates SOAPMessageContext processing and report recording using a combination of provided SoapMessageInspector and Recorder

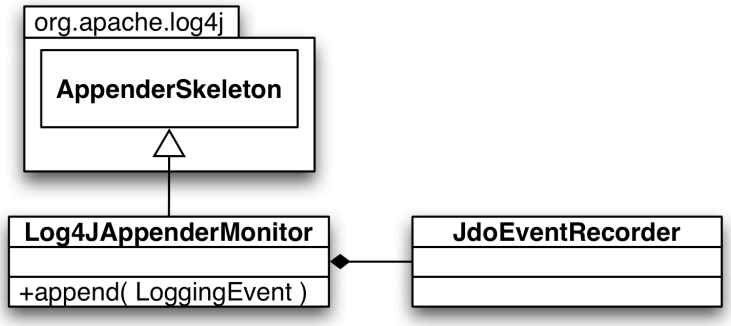


Figure 4-10. The Log4JAppenderMonitor.

objects. As such, third parties should extend HandlerDelegate or SoapMessageInspector to provide custom SOAP message analysis.

Figure 4-9 illustrates the standard workflow for SoapHandlerMonitor as invoked by an application server to process a SOAP message. The SoapHandlerMonitor extracts the SOAP-MessageContext object and provides it to SoapMessageInspectors exposed by the assigned HandlerDelegate objects.

Log event monitoring within the context of Log4J enabled systems is provided by the Log4JAppenderMonitor, illustrated in Figure 4-10. Log4JAppenderMonitor extends Log4J's AppenderSkeleton class (Gulcu, 2002; Gupta, 2003), and as such may be configured using a standard Log4J properties file to make use of filters, honor log levels, and other functional aspects of the Log4 framework, which makes Log4JAppenderMonitor a convenient vehicle by which the WSLogA Framework may be quickly integrated into legacy applications that would be difficult to update because of source code intricacy or for which source code is not available but a Log4J configuration can be adjusted (Gupta, 2003). The JdoEventRecorder component used internally by the Log4JAppenderMonitor can be replaced with a Recorder

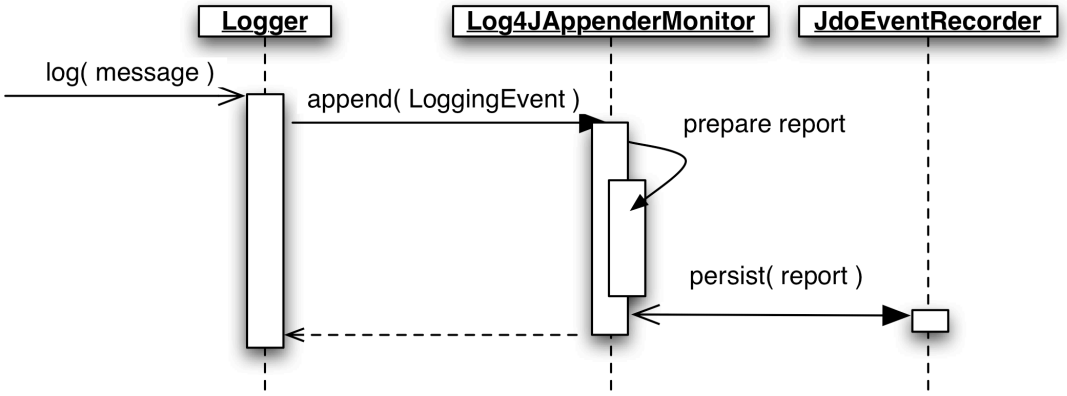


Figure 4-11. Log4JAppenderMonitor routes Log4J messages to a persistent data store.

instance appropriate for alternate information routing technologies, such as Hibernate (Bauer & King, 2006).

Figure 4-11 illustrates the standard workflow for log event processing by Log4JAppenderMonitor within the context of an application using the Log4J framework. Log messages are generated by the application and host environment and then transferred to Appender (Gupta, 2003) components by the Log4J framework. By default, the WSLogA Framework routes the received message information into a relational database management system using the Event Group.

Log event monitoring within the context of J2SE Logging API (Arnold *et al.*, 2005; Gupta, 2003) enabled systems is provided by the JdkLogHandlerMonitor, illustrated in Figure 4-12. JdkLogHandlerMonitor extends Logging API's Handler class (Gupta, 2003), which permits the monitor component to be configured within the standard J2SE JVM extension framework. This approach permits integration of the WSLogA Framework into systems for which Log4J was not an option (*e.g.*, an Apache commons logging strategy (Oak, 2004) was not em-

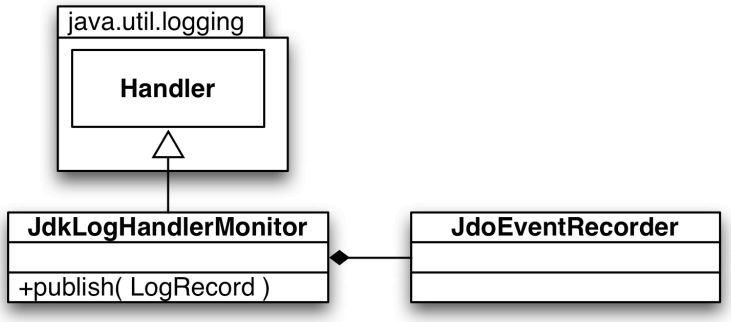


Figure 4-12. The JdkLogHandlerMonitor.

ployed). The JdkLogHandlerMonitor parallels the Log4JAppenderMonitor by employing the same policy based approach to information adjustment and by routing information to a relational database management system through the use of JdoEventRecorder; however, JdkLogHandlerMonitor must be configured as an extension to a JRE in which the application is executed (Gupta, 2003; Sun Microsystems, 2001, 2004). This configuration can be achieved by direct modification of a Sun based JVM, such as that provided for J2SE 1.5, or with the assistance of an application featuring appropriate JVM control, such as the GlassFish application server.

Figure 4-13 illustrates the standard workflow for log event processing by JdkLogHandlerMonitor within the context of a typical J2SE application using the Logging API. Log messages are generated by the application and host environment and then transferred to the Handler components by the J2SE. As with the Log4JAppenderMonitor, the default WSLogA Framework configuration routes the received message information into a relational database management system using the Event Group.

Reporter is an interface that provides the report method for use by monitors and other components interested in obtaining information regarding a Reporter object's Subject. The

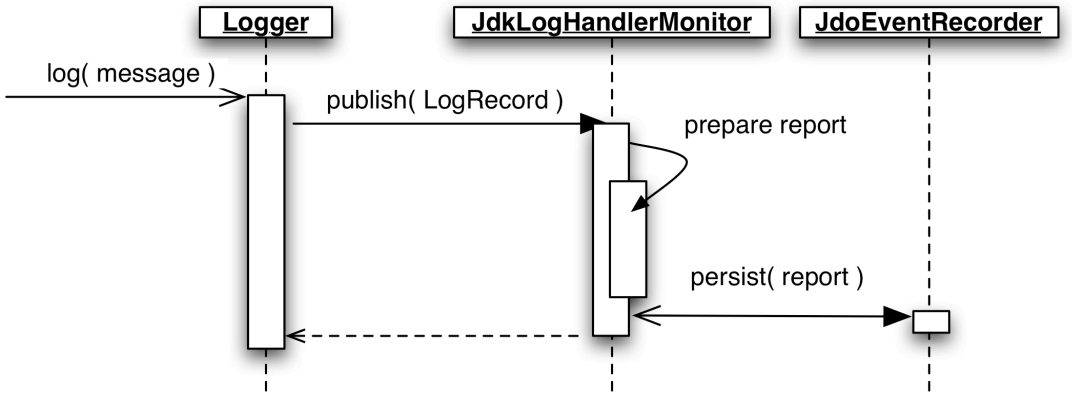


Figure 4-13. JdkLogHandlerMonitor routes J2SE messages to a persistent data store.

report method's return value is generic (Arnold *et al.*, 2005) and may represent any report structure appropriate to the system, such as a Java Object or textual XML. Reporters may serve as calculators (*e.g.*, data generators as opposed to harvesters), and in such cases a component may wish to implement both the WSLogA Framework's Reporter and the J2SE (1.5 or greater) SDK's Future interface (Goetz *et al.*, 2006) to take advantage of contemporary threading mechanisms offered by the Java platform. Figure 4-14 illustrates the Reporter and its associated components.

The Observer abstract class implements Reporter and adds functionality for tracking Subjects. Subject is generic (Arnold *et al.*, 2005), and, as such, may vary according to the system's needs. For example, an Observer might track a file within the environment or a Java Object receiving data from a SOAP transaction. The method getReportSubjects can be used to retrieve Subjects valid for report preparation, such as those that might be deemed candidates by an associated policy set. PoliciedObserver extends Observer by overriding the getReportSubjects method to honor policy filters. ObjectObserver is a concrete implementation of

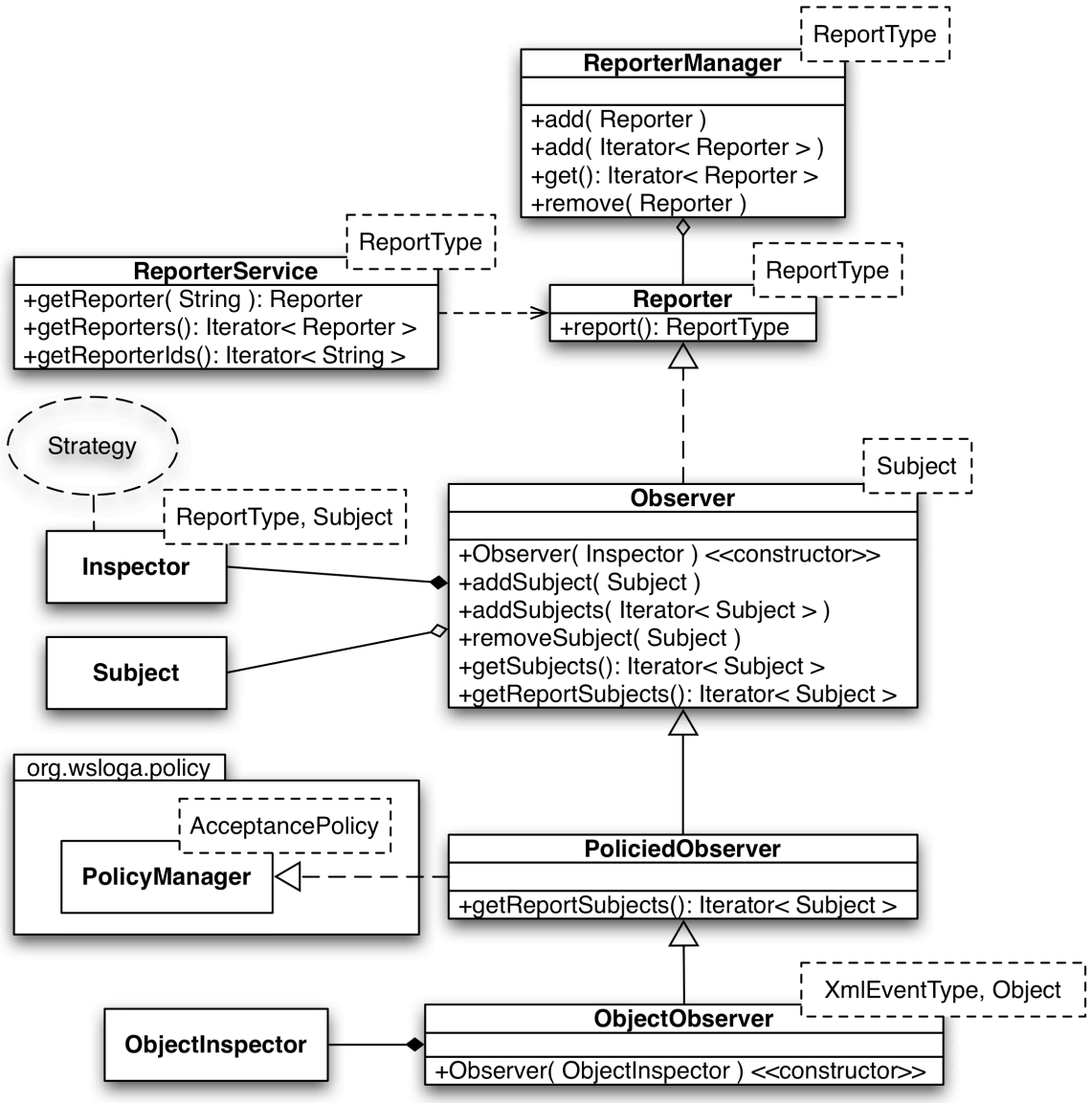


Figure 4-14. The Observer.

PoliciedObserver that uses an ObjectInspector instance to analyze Java Objects and prepare reports using the XmlEventType component provided by the Event Group.

Figure 4-15 illustrates a typical workflow for the preparation of a report by an ObjectObserver interacting with a WSLogA Framework enabled application. An ObjectObserver is

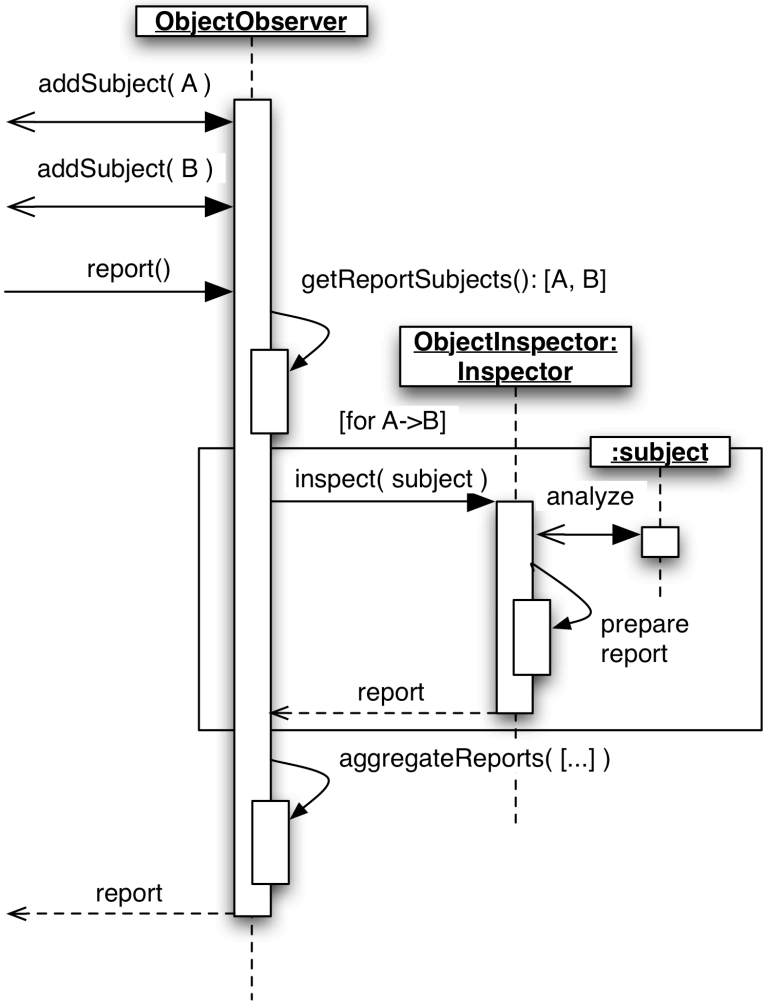


Figure 4-15. ObjectObserver reports on an Object's characteristics.

prepared and Subjects of interest are associated by the application. The ObjectObserver is then provided to a Monitor that can periodically pull reports regarding the Subjects.

Recorder is an interface that declares behavior for accepting reports generated by Reporter instances. Recorder implementations may further process or route information as appropriate to the system. The JdoEventRecorder is a concrete implementation of Recorder that accepts reports represented as textual XML and persists the information by using a

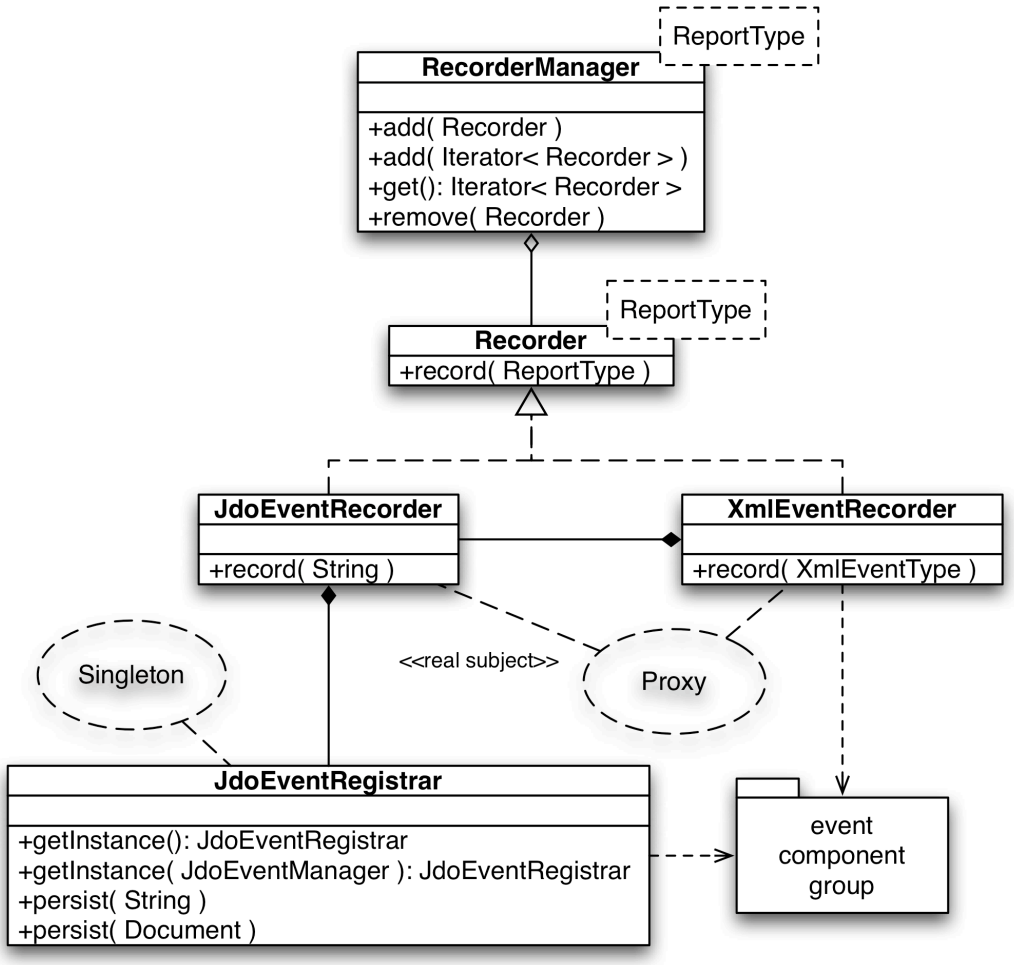


Figure 4-16. The Recorder.

JdoEventRegistrar instance. XmlEventRecorder accepts XmlEventType objects and translates the information into a textual XML report for consumption by an embedded Recorder that accepts reports represented as String instances. Figure 4-16 illustrates the Recorder interface and its associated components.

JdoEventRegistrar is a Singleton (Gamma *et al.*, 1994) class that interacts with the Event Group to appropriately generate new database entries or associate information with existing database entries in a manner that satisfies JDO's implementation constraints. The use of such

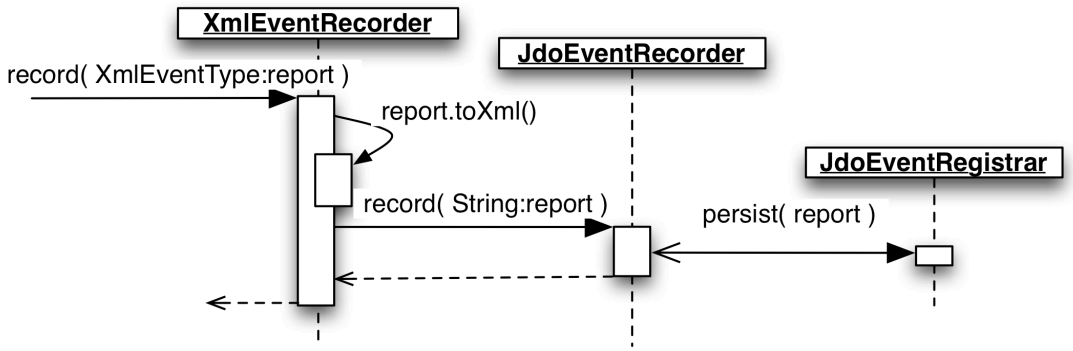


Figure 4-17. WSLogA Framework report objects are easily persisted.

a gateway into an associated data store facilitates the maintenance of data integrity within multithreaded systems.

Figure 4-17 illustrates a typical workflow for the routing of report information into an associated relational database management system by means of an XmlEventRecorder. A Monitor transfers report information from a Reporter to the XmlEventRecorder, which first transforms the report into textual XML and then provides the report to a JdoEventRegistrar so that the information may be persisted.

The Inspector component provides the Monitor Group with functionality for generating detailed report information regarding a Subject. Third parties may build domain specific Inspectors, but predefined Inspector sets are provided for SOAP and log message inspection within the context of the most popular J2EE technology platforms for those purposes. The Inspector interface contains generic references to the type of report and Subject addressed by the inspection. Policy management is provided for Inspector through the PoliciedInspectorBase abstract class, which may be extended by third parties to produce flexible inspection solutions for their applications. PoliciedInspectorBase uses AcceptancePolicy instances to

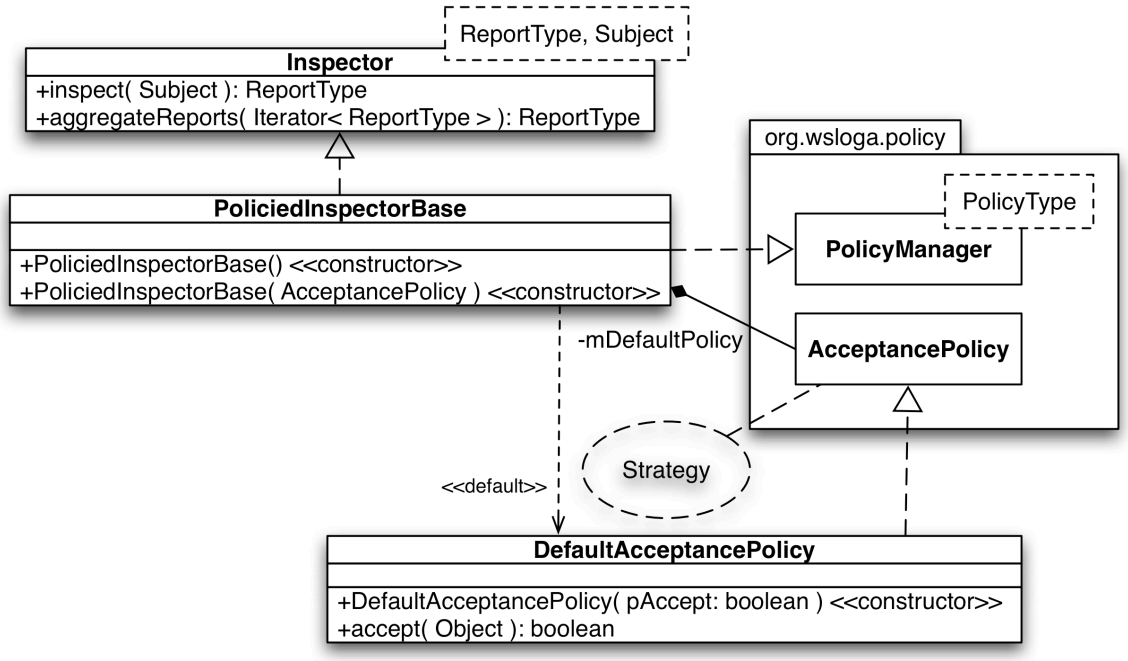


Figure 4-18. The Inspector.

filter the type and content of information obtained from a Subject. For example, a Policy could be established to ignore social security numbers within an Object representing a financial account. A default AcceptancePolicy is permitted for reference by PoliciedInspectorBase whenever a standard policy associated within a PoliciedInspectorBase instance fails to filter information. DefaultAcceptancePolicy is defined as a member class of PoliciedInspectorBase, and will accept all information provided for the report. Figure 4-18 illustrates the core Inspector components.

SoapMessageInspector (Figure 4-19) is a dedicated Inspector for SOAPMessage (Graham *et al.*, 2005) objects that are used by J2EE and Axis Web service environments, such as those provided by the GlassFish and JBoss application servers. SOAPMessage objects provided to

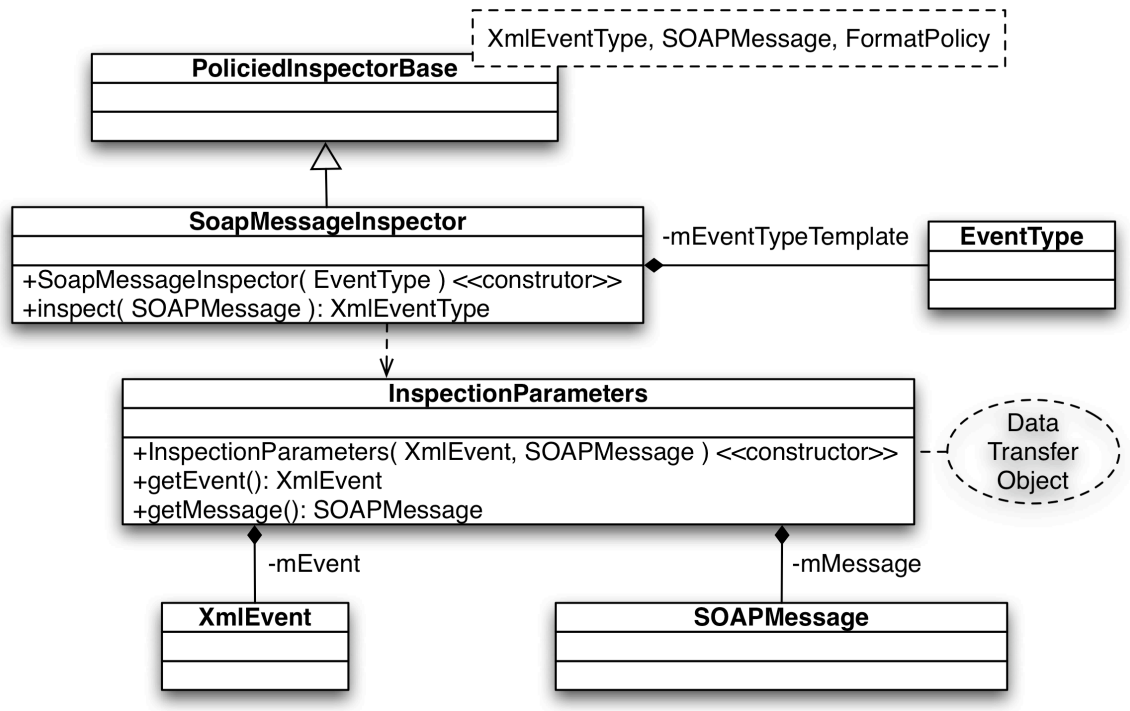


Figure 4-19. The SoapMessageInspector.

SoapMessageInspector, such as by a SoapHandlerMonitor, are properly analyzed for information contained within XML fields in addition to information stored as Java Object attributes. SoapMessageInspector implements PoliciedInspectorBase to provide Policy managed information extraction and report generation.

Integration with encoded and legacy log architectures is provided by the WSLogA Framework using a component set that shares log inspection functionality, as illustrated in Figure 4-20. Encoded log messages are those whose message payloads are structured to accommodate parsing, such as what may be provided by XML, whereas Legacy log messages are those in plain text format, such as those intended to be read by a developer debugging an application. LogInspector is an interface that declares core log message processing

functionality that accepts a generically typed message object and provides hook methods intended to extract information from the message object. Log4JInspector and SunLogInspector implement LogInspector according to log platform requirements—Log4J LoggingEvent (Gulcu, 2002; Gupta, 2003) objects in the case of Log4JInspector and J2SE LogRecord (Gupta, 2003) objects in the case of SunLogInspector. Parallel components extending log platform classes are provided to accept log messages from log systems and route messages to respective log Inspector components for processing. EventReportLayout includes Log4JInspector as a composite attribute and may be integrated into Log4J based systems. EventReportFormatter includes SunLogInspector as a composite attribute and may be integrated into J2SE Logging API based systems. The default behavior of the log inspection components is to capture log messages in a manner that permits policy based control, but third parties may provide additional logic to enable analysis of highly refined information such as that contained by XML encoded message payloads.

Employment

The Monitor Group is integrated into the WSLogA Framework as the information acquisition and routing mechanism. Functionality is provided for observing or inspecting data from a variety of sources—such as SOAP transactions, log frameworks, or Java runtime objects—and, with the assistance of Policy components, route appropriately filtered or calculated information into data stores, such as a relational database management system or another Web service. Figure 4-21 illustrates the relationship between the Monitor Group and other component groups as well as the environment.

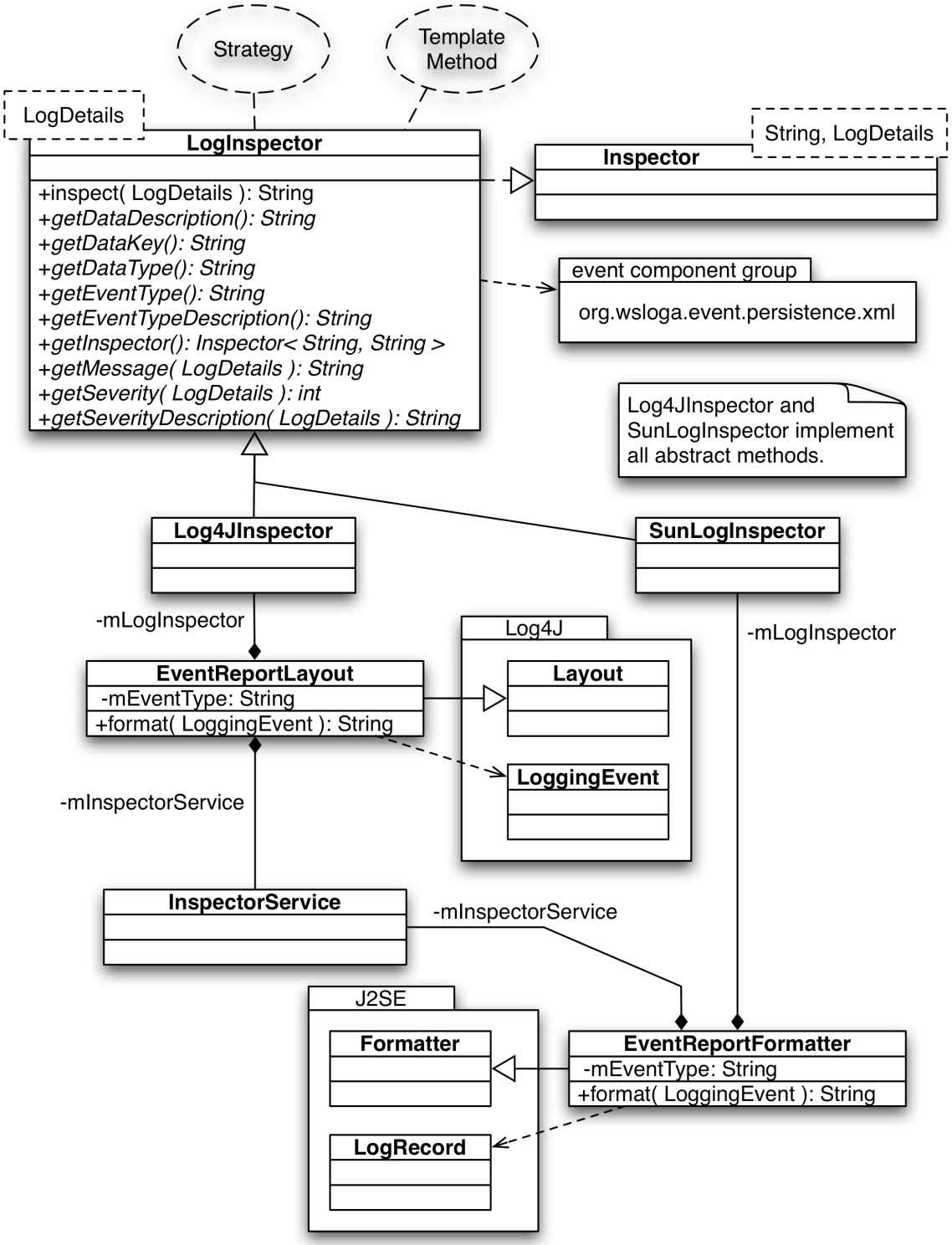


Figure 4-20. The log framework Inspectors.

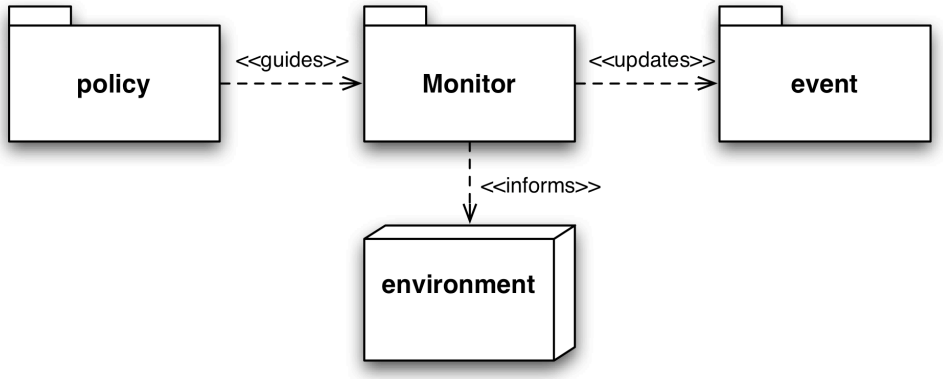


Figure 4-21. Monitor Group relationships.

The Monitor Group must be employed in conjunction with the Policy Group, but otherwise is independent of other WSLogA Framework components. Such an independent integration of the Monitor Group would facilitate robust information acquisition for Enterprise systems, and, in particular, those based on Web services that require data capture across contexts (*e.g.*, information provided by both SOAP messages and runtime objects). The Monitor components can be employed to control the timing of observations to ensure information acquisition is coordinated for sessions or transactions. Figure 4-22 illustrates the general relationships involved in the deployment of only the Monitor Group and essential associated components.

The Monitor Group is demonstrated as the information acquisition and routing mechanism for the Adventure Builder application (Appendix C). Scenarios such as the information collection example employ the Monitor Group as an integrated function of monitoring systems hosted within the GlassFish application server process. A SOAP monitor is associated with the Lodging Web service to capture lodging requests generated by the Order Processing

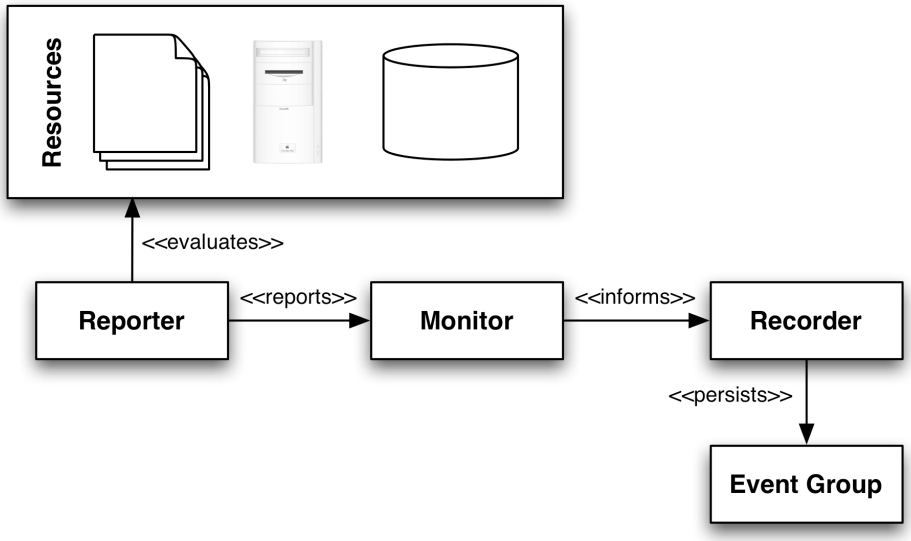


Figure 4-22. General employment of Monitor Group members.

Center (OPC) Web service. ObjectInspector components are also configured and associated with the lodging request generation and consumption components provided by Adventure Builder, and the information is later correlated to ensure transaction integrity. A key feature of the strategy employed is that the monitoring system requires only minor modifications to the Adventure Builder application and does not impact the business logic's flow. Figure 4-23 illustrates an example workflow involving the Monitor Group.

Constraints and Opportunities

The Monitor Group provides the capability to capture SOAP message information made available by request, response, and fault events; however, the WSLogA Framework improves on this capability by also enabling the acquisition of information related to the transaction's context. For example, functionality is provided for integration with Log APIs, and third parties may develop more complex acquisition components such as those that inspect

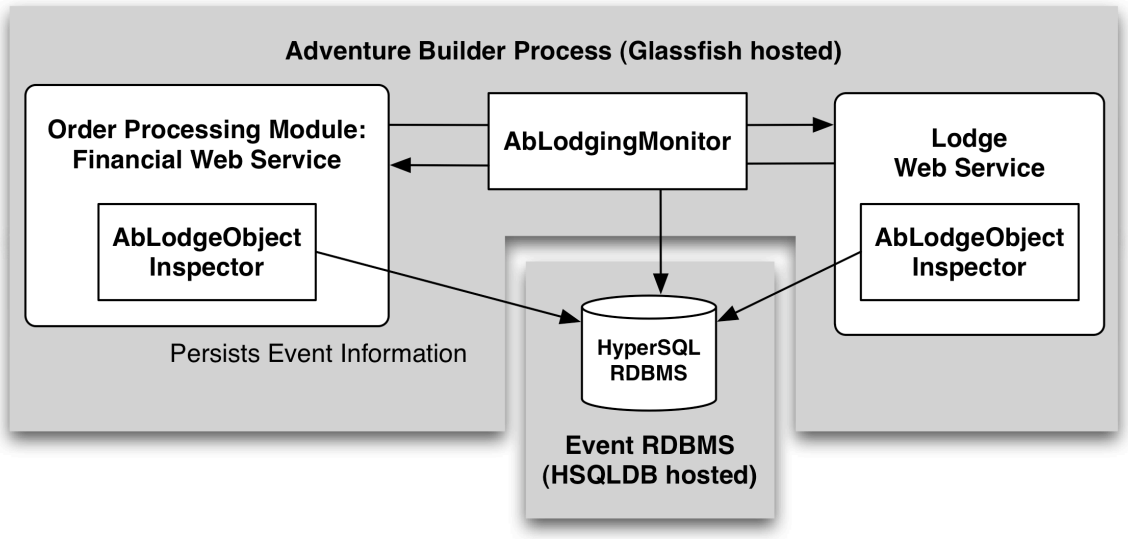


Figure 4-23. An example employment of the Monitor Group.

databases, system log files, or distinct hosts. A combination of information sources facilitates decision making and reporting that provides a holistic understanding of system behavior that cannot be obtained by SOAP analysis alone (Telles & Hsieh, 2001).

The concept of coordinating reporting and recording activities by means of a Monitor cleanly separates tasks to permit the development of specialized components (Gamma *et al.*, 1994; Greenfield & Short, 2003; Schmidt & Buschmann, 2003). Further, Reporter and Recorder components can be employed independently of a coordinating Monitor component to provide Enterprise systems with the best flexibility for establishing information flows. However, Monitors should be introduced to Enterprise systems whenever information pertaining to a set of Subjects or event milestones should be recorded as a coherent report. For example, a B2B e-commerce exchange may wish to confirm the entry and exit statuses of

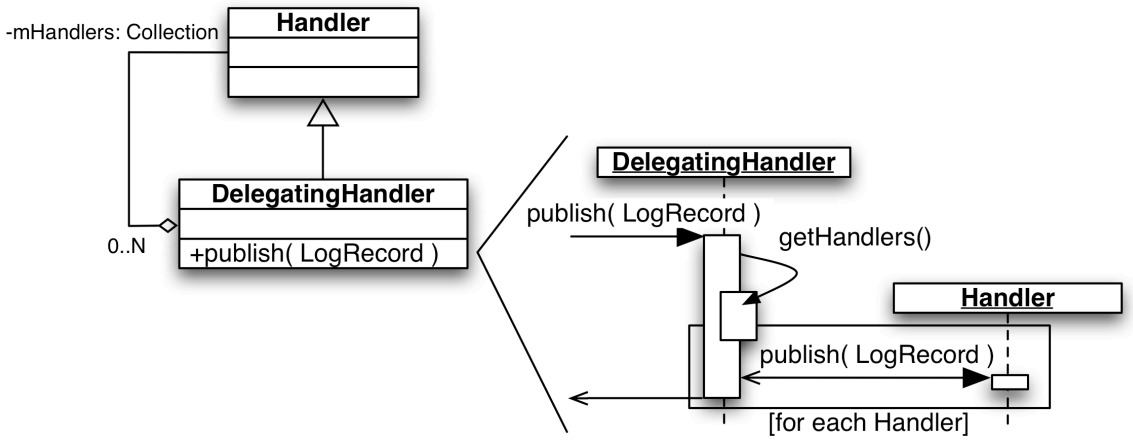


Figure 4-24. A J2SE Logging API derived Handler may delegate message management.

a Web service module, and part of that confirmation may include user account or server state calculations.

The Reporter component family makes explicit that information can be calculated or harvested. Often, as may be provided by an Inspector, the report produced will be a combination of values that could further be modified according to rules introduced by active policy sets. Third parties extending the WSLogA Framework should introduce log events (that may also be harvested by WSLogA Framework, such as by means of Log4J integration) for complex data flow and transformation relationships to ensure a complete understanding of original versus modified or calculated data during the development phase.

The logging API integration components—Log4JAppenderMonitor and JdkLogHandlerMonitor—provide convenient integration of the WSLogA Framework into established Enterprise systems, and in particular those systems that are difficult to modify (such as an application server), but convenience comes at the price of flexibility. Log messages bearing unstructured information (*e.g.*, text intended to be read by people) may not be suitable for

use in an environment analysis engine without significant preprocessing. Systems providing event information to WSLogA Framework may need to be reworked to ensure that log messages use structures such as those provided by XML. Reporting components can subsequently process the structured information to produce human friendly reports, if necessary.

The J2SE Logging API is generally configurable only through JVM properties (Gupta, 2003), which means data acquisition goals may conflict with the log routing intention of established systems. For example, the GlassFish application server uses the J2SE Logging API to manage its log records, and GlassFish's log configuration is performed through an administrative console that sets log management preferences globally for the JVM. JdkLogHandlerMonitor can only be used with GlassFish as a configurable, external entity to the system as a substitute Handler instance in lieu of GlassFish's preferred Handler (Appendix C). Derivations of the JdkLogHandlerMonitor may need to be developed that conveniently allow delegation of log information to other Handler instances by means of an external configuration mechanism to ensure that both the WSLogA Framework and the host system's information management objectives can be realized without adversely affecting the information flow. Figure 4-24 illustrates such a relationship.

The Policy Group can significantly enhance information flow and transformation by introducing business rules that may change report content without requiring adjustments to the principal acquisition or host logic. This functionality is important for those systems deployed throughout environments that may have different information management obligations. WSLogA Framework enables convenient policy integration through components such as PoliciedObserver.

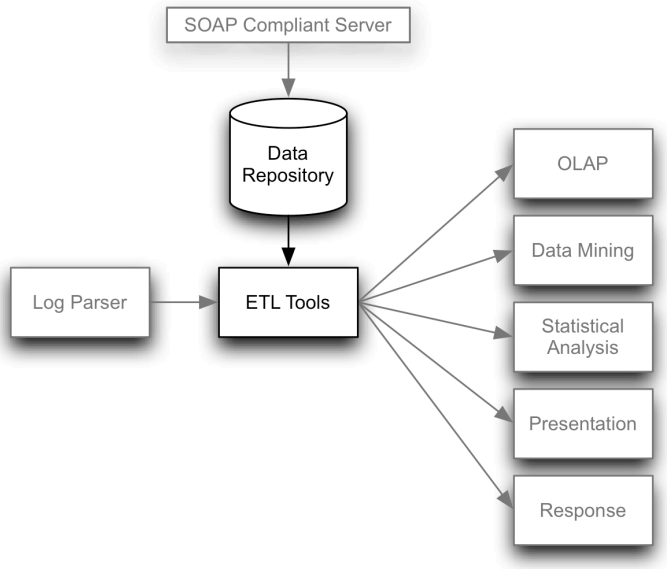


Figure 4-25. WSLogA elements addressed by the Event Group.

The Event Component Group

The *Event Group* is comprised of those components that model and persist the information captured by the WSLogA Framework. The interfaces, classes, and resources for this group are defined at the *org.ws.loga.event* package. Figure 4-25 illustrates those portions of the WSLogA platform that are addressed by members of the Event Group with grey elements indicating boundary components. Figure 4-26 illustrates the use cases embodying these workflows. Appendix H documents the activities associated with each use case.

Roles and Responsibilities

Eight roles were envisioned for the Event Group components for the purpose of modeling the information of interest to WSLogA Framework components. The focus is on the description of an event, but ancillary roles assisting event or data management are provided

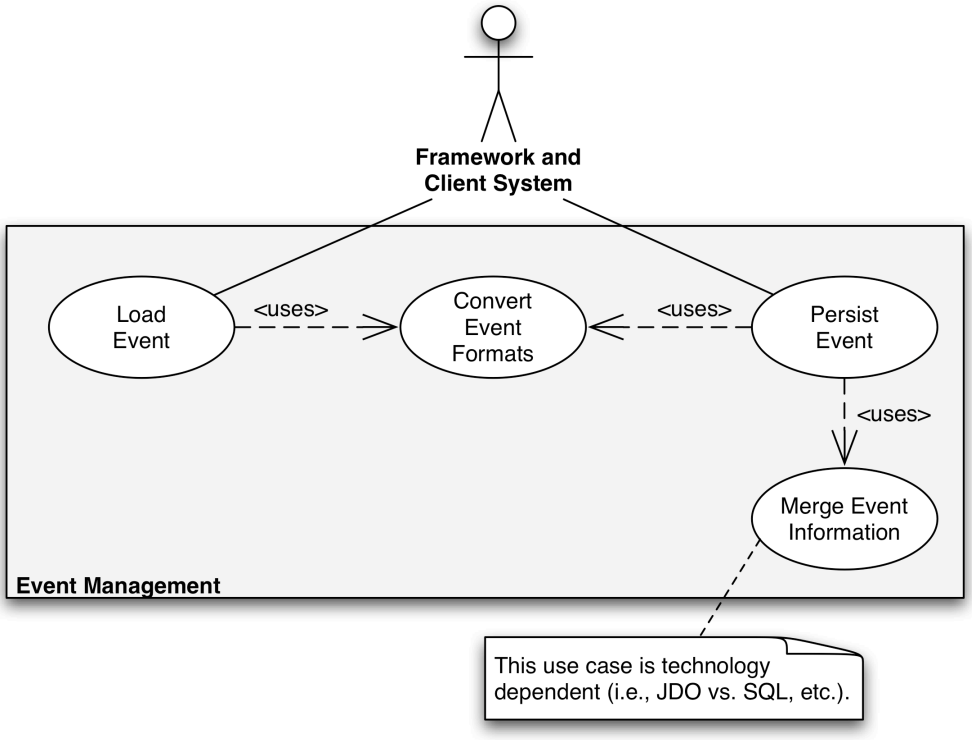


Figure 4-26. Use cases applicable to the Event Group.

for the convenience of Perspective or Response Group components. Figure 4-27 illustrates the Event Group component roles and their relationships.

The Event role represents an event occurrence within the service, transaction, or environment and is generally comparable to log messages or click stream data. An Event encapsulates data specific to its occurrence, which is represented within the system as generic Datum instances or specialized types such as Locations.

Similar Events are organized within an Event Type, which is metadata facilitating the convenient reference of Event sets. Event Types are characterized by a Severity, which should be interpreted as a degree of significance within the universe of Events as opposed to continuity of system functionality. This recognizes that issues of continuity are really a

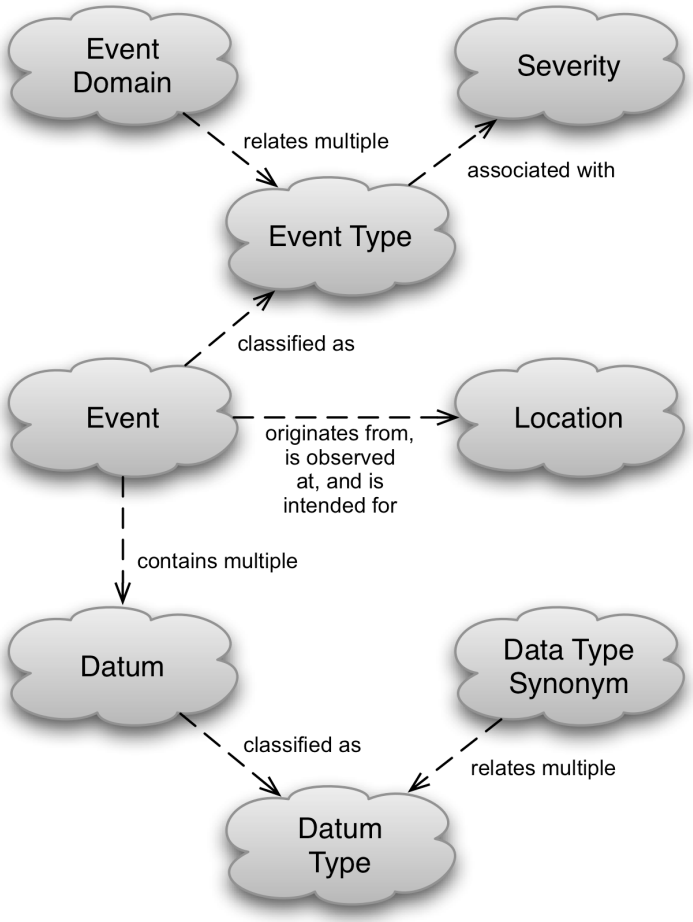


Figure 4-27. Event Group component roles.

matter of perspective and best interpreted by analyzers within the boundaries of business rules, such as those analyzers enabled by the Response Group (Lai *et al.,* 2005; Larson & Stephens, 2000).

Event Domain and Data Type Synonym roles represent additional metadata organizing Event- and Datum Types for the convenience of perspective or Response Group components. The establishment of these roles recognizes that long term organization of information

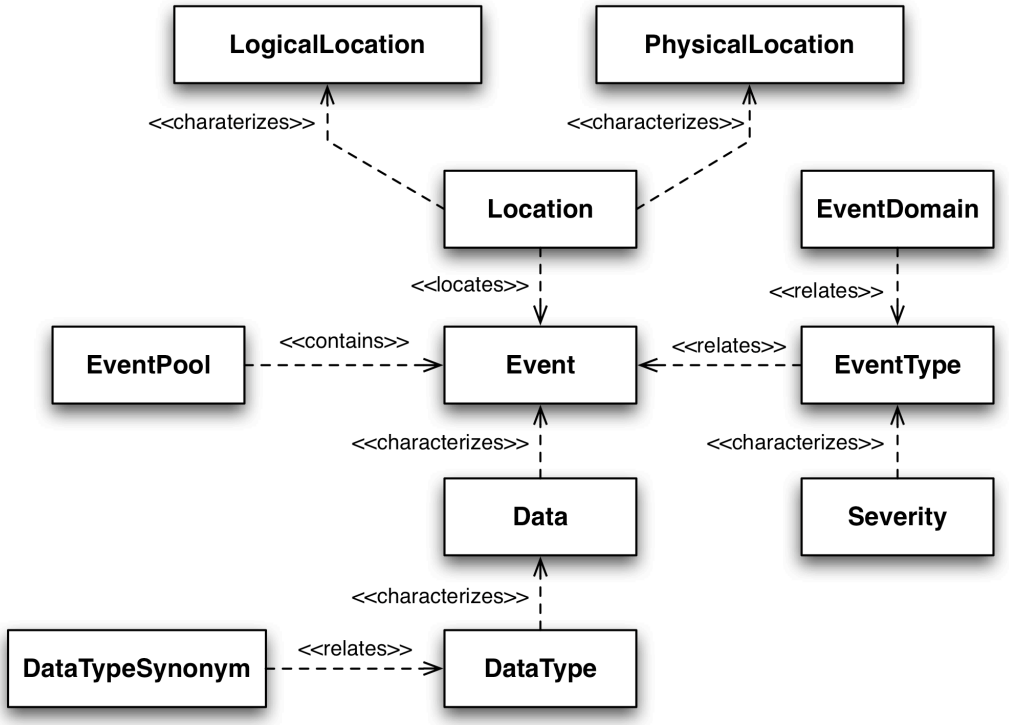


Figure 4-28. Event Group component structure.

within a system's event database is subject to change due to evolution in system enhancements or extensions.

Structure

The Event Group is structured using the Event interface and supporting components, as illustrated in Figure 4-28. The interfaces provided define the structure for event values, locations, and categories, and, as such, form the data model for the WSLogA Framework's management of acquired information. The interfaces do not make assumptions as to whether the information will be persisted, which permits the development of component

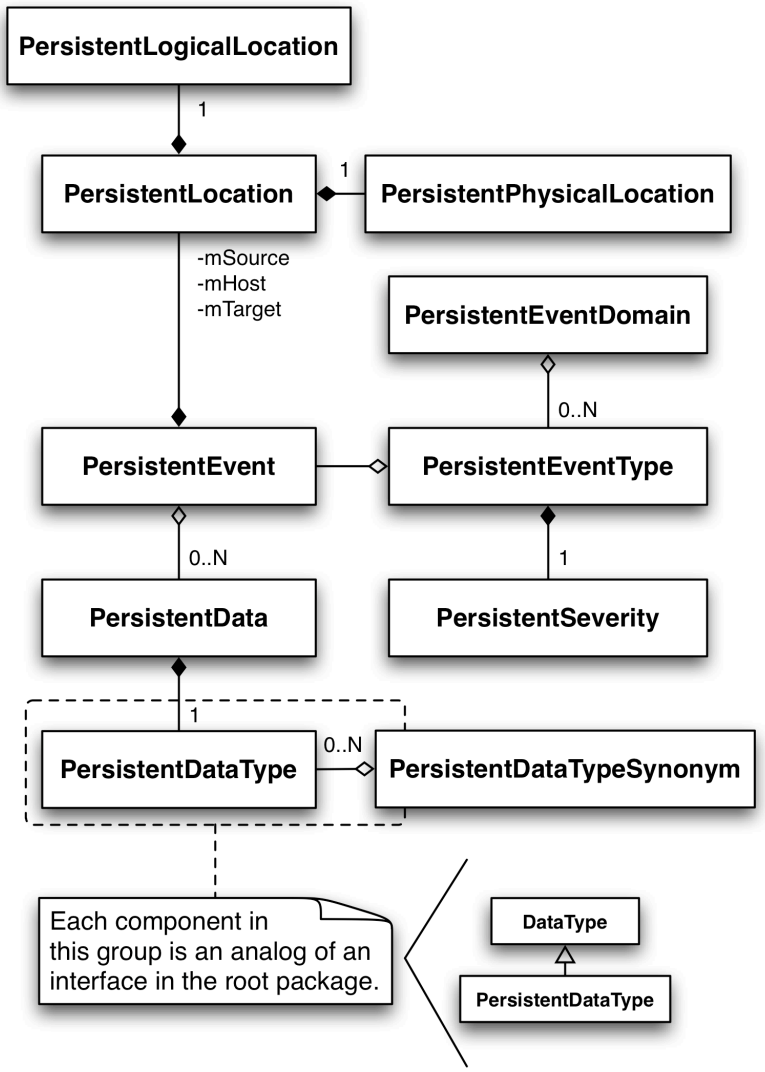


Figure 4-29. The event components define the data model.

families that distinctly address inter-component communication (*e.g.*, using XML payloads) and data store transitions (*e.g.*, using JDO or Hibernate).

The Event interface is intended to organize information related to an event occurrence, such as the Event's location, transaction attributes, and processing markers applied by managing components (*e.g.*, the Response Group's ResponseTask). All other components

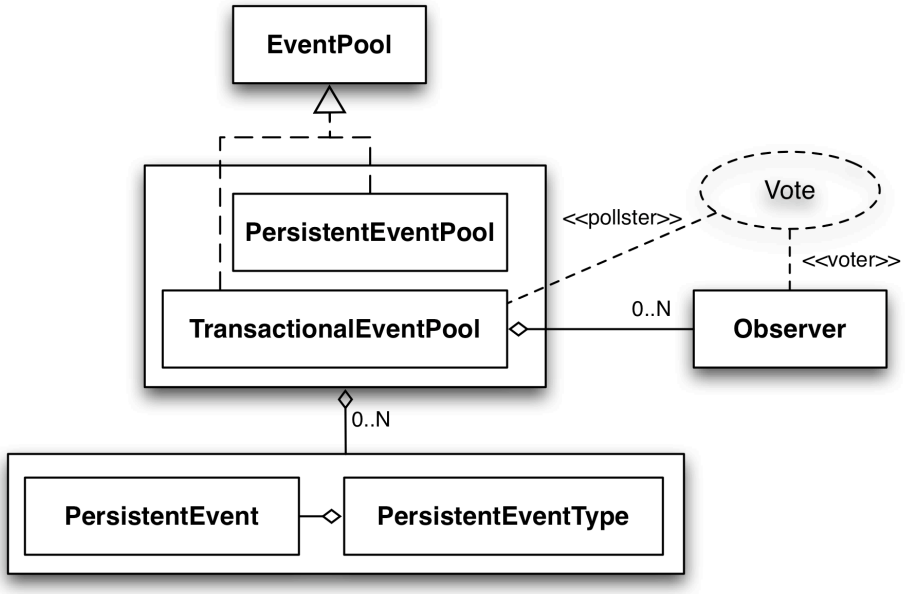


Figure 4-30. The Event Group facilitates inter-component event information transfer.

within the Event Group are or ganized around the Event interface to either define related information containers or to manage persistence.

The EventType interface is intended to organize Event objects related in terms of a logical type established by the Monitor Group. For example, Events could be organized in terms of the Web service in which the Event occurs, or Events could be organized in terms of transaction types (of which multiple types could be handled by a single service endpoint).

The EventDomain interface is provided to relate EventTypes that may be conceptually similar but identified using different labels. For example, one development department may establish an EventType known as *com.someCompany.serviceFailure* and another department could use *com.someCompany.nonResponsiveService* to capture what is effectively the same issue from the perspective of an environment management system (Meadow *et al.*, 2000).

The Location interface provides access to Physical- and LogicalLocation components, which record the coordinates for an Event's occurrence. Analysis systems can build maps of an information set's system traversal using the Location information—including for Web service nodes external to an in-house Web service system if the external nodes also implement the WSLogA Framework. Credit and financial institutions, among others, often provide services involving the operation of multi-organization Web services and can use this feature to ensure those partners with the best performing systems are rewarded with system usage during transactions (Anselmi *et al.*, 2007; Tong & Zhang, 2006).

The Data and DataType interfaces are intended to facilitate event descriptions and should be managed by Event objects. Data objects can uniquely identify information for later retrieval and analysis. DataType objects can be associated with Data objects to identify the kind of information being tracked, such as information that is part of the Event (*e.g.*, the amount of a fund transfer) or metadata provided by Event processors (*e.g.*, marking an Event as processed so that it isn't redundantly analyzed).

The Event Group provides a model by which gathered information is organized but no assumption is made about the data persistence system used to accept information. Applications can implement the Event Group's foundation interfaces to integrate most data persistence technologies if the distributed WSLogA Framework components do not meet the adopting system's requirements.

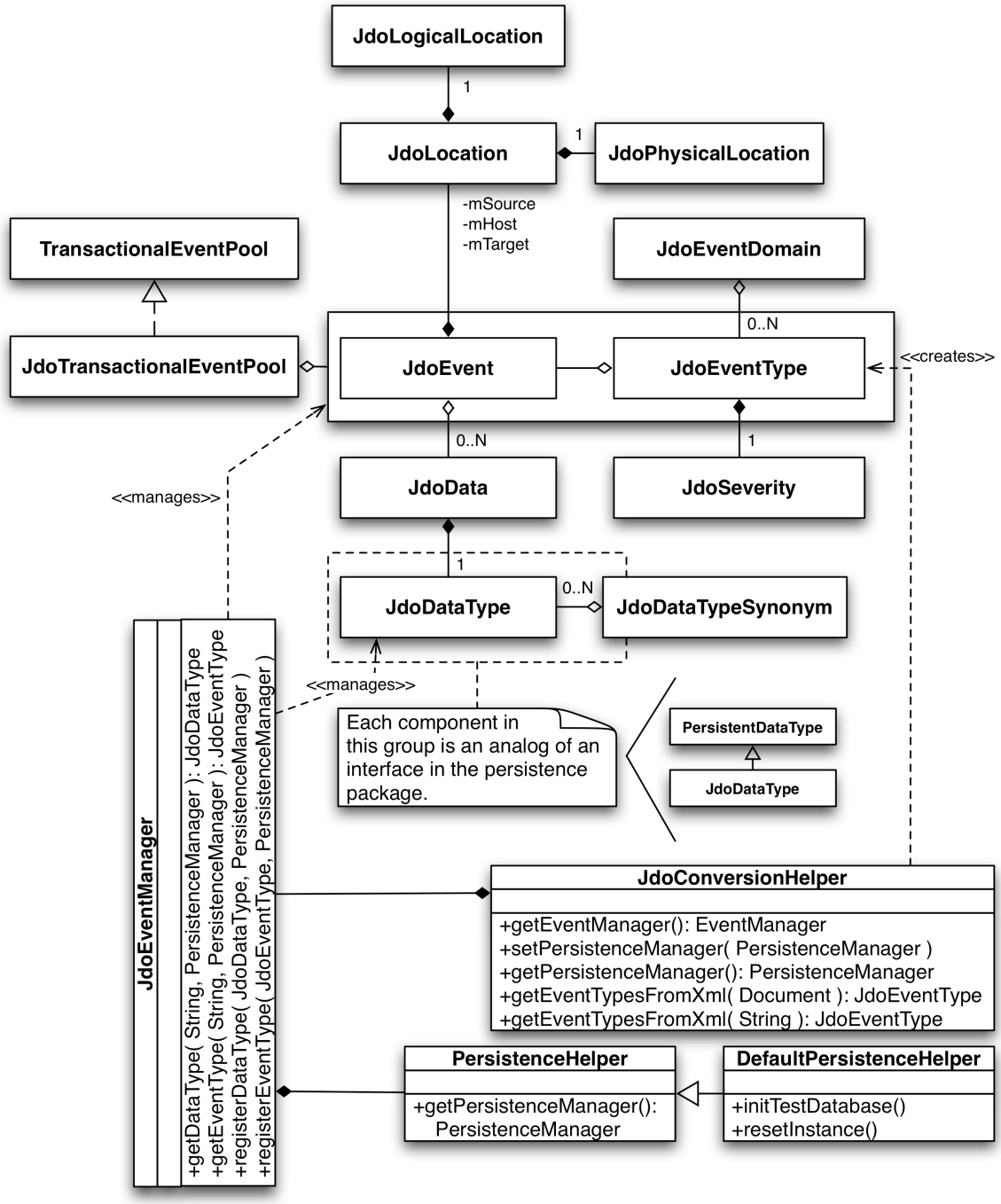


Figure 4-31. The data model as adopted for use with JDO.

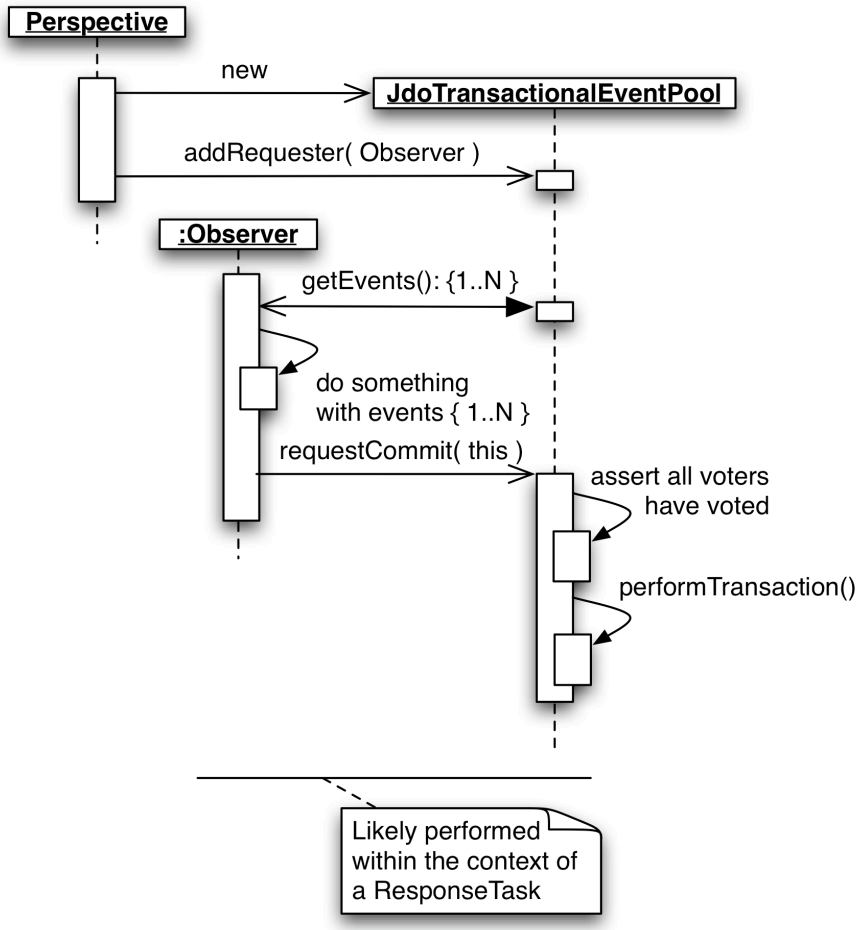


Figure 4-32. JdoTransactionalEventPool enables bidirectional information management.

Implementation

Each component role is provided with an abstract or concrete class implementation that organizes information in the form of class attributes and provides derivations with hook methods for state initialization or information management capabilities expected as a common occurrence for the data model. Third parties may use or derive the components within this package to share data among components utilizing otherwise incompatible

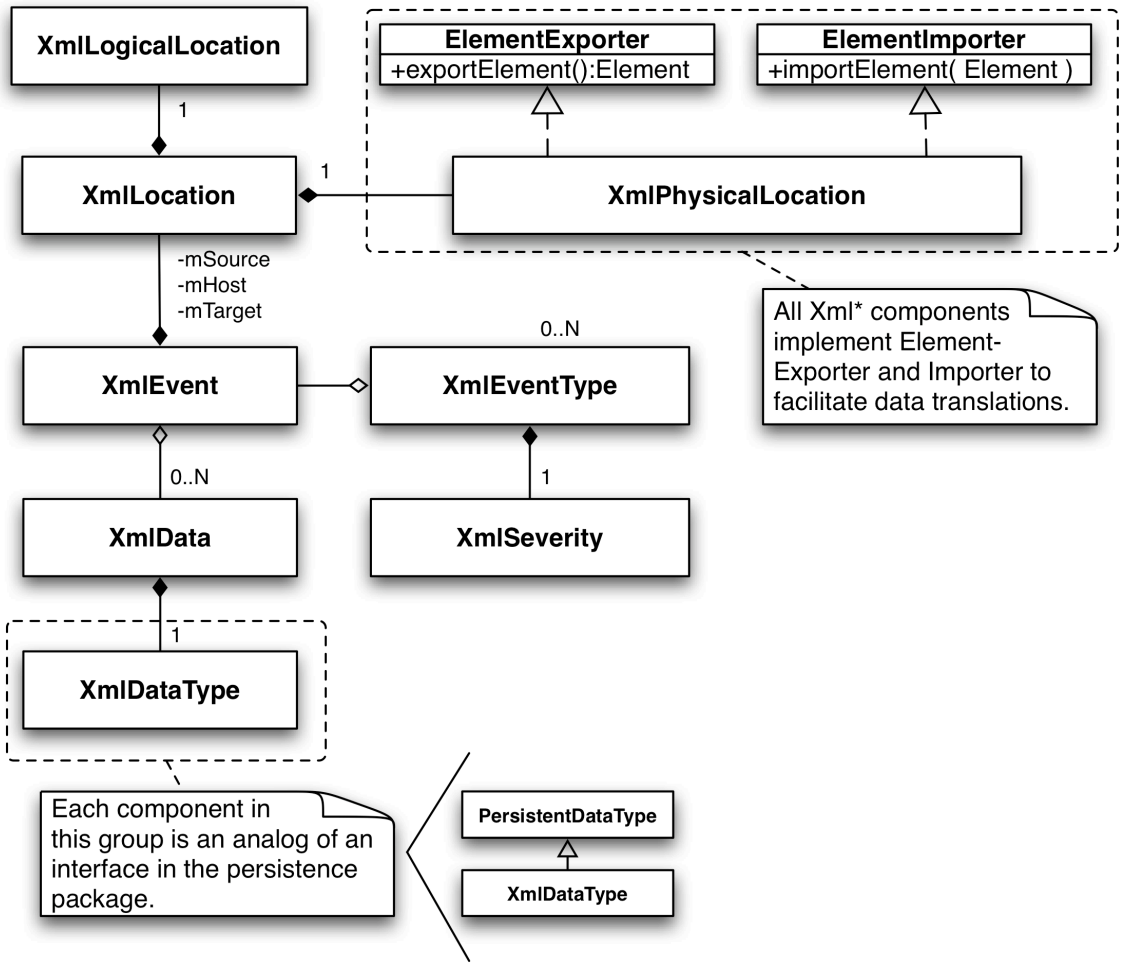


Figure 4-33. The data model as adopted for in inter-component information exchange.

technology platforms, such as Hibernate (Bauer & King, 2006). Figure 4-29 illustrates the components implementing the information data model.

Also defined within the *persist* extension package are components for transferring event information among system modules or WSLogA Framework APIs. EventPool is the base data transfer object (Alur *et al.*, 2003) within the WSLogA Framework, and should be used by

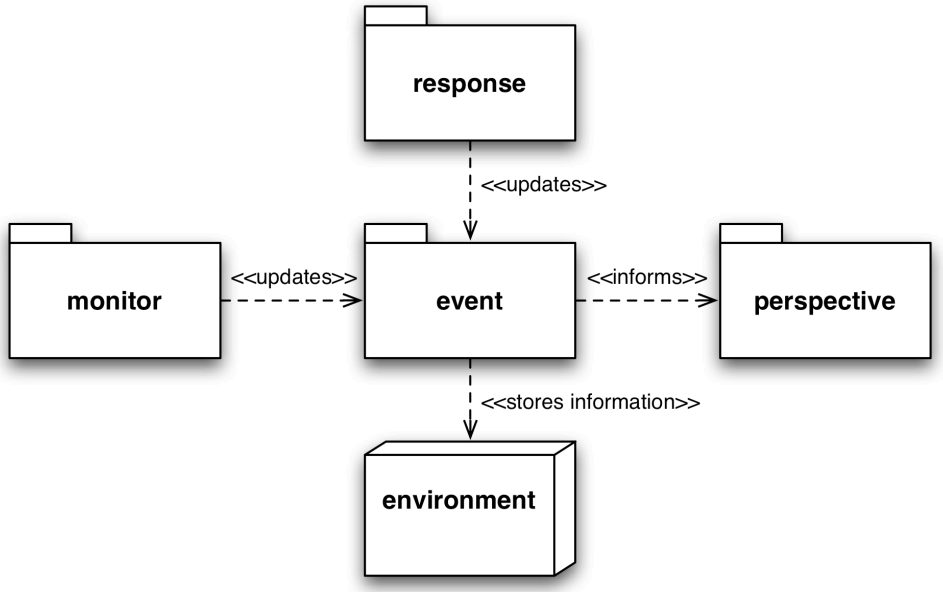


Figure 4-34. The Event Group relationships.

third parties to move information from Perspective to Response components when such information should not be modified by analysis engines. Figure 4-30 illustrates the EventPool and associated support components.

The WSLogA Framework provides data persistence capabilities that take advantage of relational database management systems, such as that provided by the HyperSQL database engine (HSQLDB Development Group, 2008) used in this investigation's demonstration (Appendix C). The JDO technology platform (Tyagi *et al.*, 2004) was adopted for this purpose because it uses an object-oriented approach to transferring data between the application and data tiers that is easily understood by Java developers from a variety of data management backgrounds (Landre *et al.*, 2007; Senthil *et al.*, 2007).

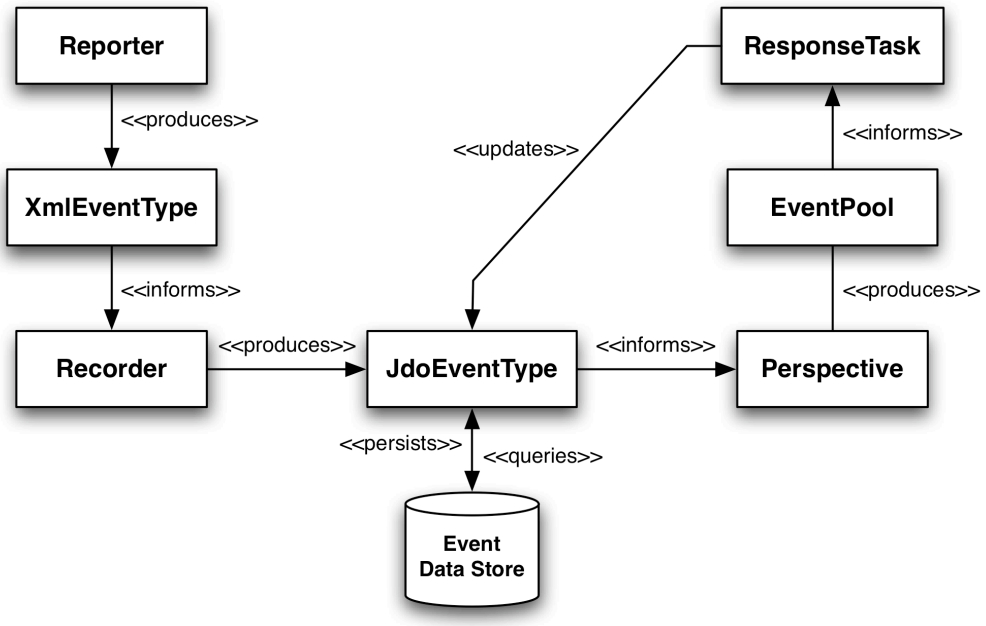


Figure 4-35. General employment of Event Group members.

Each data model class defined in the base package is extended and enhanced for use with the JPOX JDO engine (JPOX, 2008). Rules enforcing referential integrity are implemented in components as necessary, and functionality is provided within the JDO components that enables their conversion into XML to facilitate convenient transfer of their information across system boundaries for which JDO may not be an option (*e.g.*, a socket or RMI connection to a parallel processing system). Similarly, information in an appropriate XML form may be accepted by the JdoEventType component to produce an object hierarchy appropriate for persisting information into the associated database. Figure 4-31 illustrates the JDO enabled components.

The JDO component family provides enhanced capabilities for transferring Event information between Perspective and Response components. The TransactionalEventPool

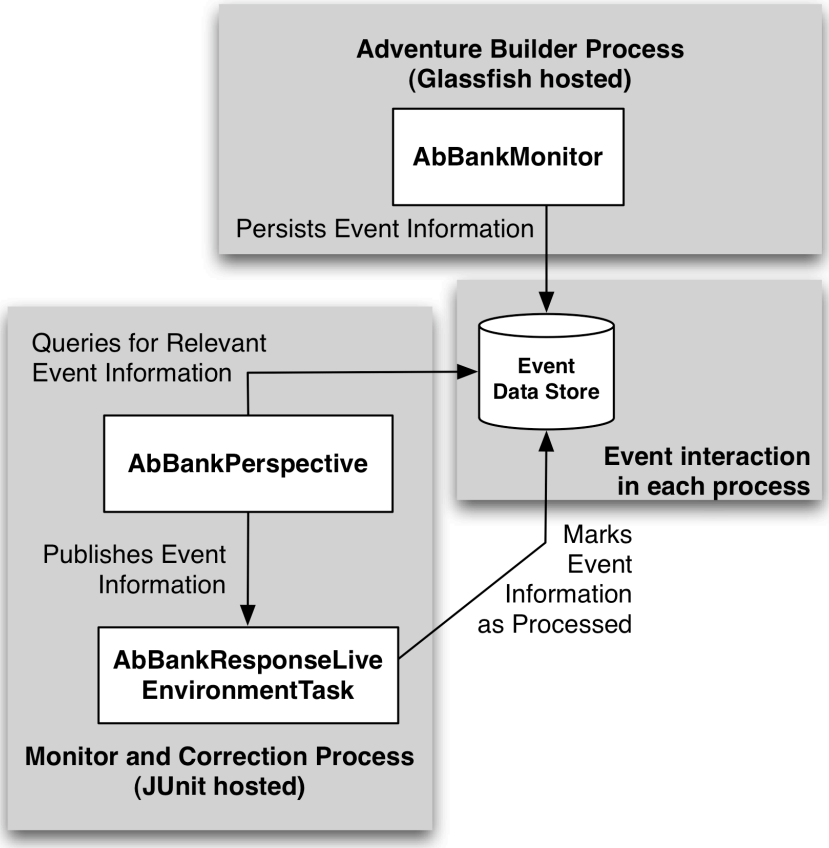


Figure 4-36. An example employment of the Event Group.

accommodates the registration of objects that may modify the associated Event information, such as to add metadata that prevents redundant processing of Event records. Registered observers can vote on how Event information provided by the TransactionalEventPool should be persisted, if at all. The change can be made effective once all of the observers have voted as well as when a timeout period set by managing logic expires. For example, a Perspective could instantiate a TransactionalEventPool object for which the data should only be considered valid for a maximum of five minutes. Figure 4-32 illustrates use of the TransactionalEventPool within a typical Event analysis module.

A parallel component family manages the transition of event information from the structure of XML to object form. The components within the *xml* package are not responsible for persisting event information so the managing logic is limited to ensuring reasonable data integrity within object models established in result of a parsed XML feed. The components within the *persist* package are used as the foundation model ensuring transparency in data conversation within the WSLogA Framework, such as for managing the transition of Event information from XML report to JDO object form, so competing technologies, such as JAXB (Graham *et al.*, 2005), were not used. Figure 4-33 illustrates the components provided by the *xml* package.

Employment

The Event Group is integrated into the WSLogA Framework by serving both as the data model for event information and the principal mechanism by which that data is transferred between the application and data tiers. The Monitor and Perspective Groups are structured around the Event Group's functionality. The Recorder component set within the Monitor Group uses the Event Group's JDO integration to persist reports about Events and their context. EventPool and TransactionalEventPool serve as Data Transfer Objects (Alur *et al.*, 2003) to move event information from the data store to event analysis and response engines, which permits Perspective components to focus on framing the ad hoc data models presented for reports and analysis instead of data loading. The common data model provided by components within the persistence package, such as PersistentEvent, permit the establishment of extension packages in which components provide specialized data management capabilities. For example, the JDO based components juxtapose the WSLogA Framework

data model with the persistence management capabilities of the JPOX framework. Likewise, a third party could create a custom data management platform with capabilities such as persistence over the wire (*e.g.*, using a Web service). Figure 4-34 illustrates the relationship between the Event Group and other component groups as well as the environment.

The Event Group is an integral part of the WSLogA Framework and is not intended to be used apart from the other component groups. Instead, third parties should concentrate on integrating either the Monitor or Perspective Groups into their system architecture to take advantage of their relationship with the Event Group. However, third parties may wish to provide data management extensions to the components within the *persistence* package to accommodate system specific technology constraints. For example, a system based on the Spring framework (Walls & Breidenbach, 2007) may use the Hibernate data management platform, in which case the developers for such a system are likely more comfortable organizing data queries using SQL instructions. Figure 4-35 illustrates the general relationships involved in the deployment of the Event Group within the context of the Monitor and Perspective Groups.

The Event Group is demonstrated as the event data model and management mechanism for the Adventure Builder application (Appendix C). All of the scenarios presented within Appendix C involve the capture of information from sessions involving the Adventure Builder application (Appendix B), for which the result is event information persisted within the associated HyperSQL database configured for use with the WSLogA Framework. As appropriate, Perspective-derived components use the Event Group to retrieve event information with specific characteristics from the database and share subsets of that information with ResponseTask derivations. Although the HyperSQL database is used in the demonstra-

tion, any relational database management system compatible with the JPOX framework can be configured for use with the WSLogA Framework's default implementation. Figure 4-36 illustrates an example workflow involving the Policy Group.

Constraints and Opportunities

The Event Group is designed to describe event information using a common denominator model easily represented within relational data systems, such as the HyperSQL relational database used for this investigation's demonstrations (Appendix C). Relational data systems are popular complements to Enterprise application environments and a variety of object-relational mapping (ORM) platforms—including Enterprise Java Beans (EJBs), Java Data Objects (JDO), and Hibernate—have been developed to integrate Java based systems with relational data systems. A feature of many ORM solutions is that they use the Java Database Connectivity (JDBC) API (Reese, 2000) to transfer data, which enhances an Enterprise system's flexibility by offering potential integration with non-traditional formats that include ad hoc file systems and XML data sets. Particularly in the case of the XML file set, these alternatives can open up opportunities for investigating WSLogA Framework integration with search platforms such as Apache Lucene (Gospodnetic & Hatcher, 2004) or Hadoop (Apache, 2007; Dean & Ghemawat, 2008) to augment the perspective or Response Group capabilities.

The JPOX framework for JDO was selected as the data persistence technology because that platform can operate outside of Application server containers and provides software developers with an object-oriented paradigm that naturally complements the Java language. Enterprise JavaBeans (EJBs), Hibernate, and JDBC/SQL access were also considered for the

implementation, but their dependency on application containers or procedural data access strategies eliminated their candidacy for the initial version of the WSLogA Framework.

Initial JDO implementations are limited in their ability to handle queries such as those using negation to shape result sets. Some Perspective components developed to demonstrate the WSLogA Framework had to use expensive query strategies to circumvent query structure limitations that would have been easily solved using SQL syntax (Appendix C). However, the intended effect of the Perspective components—the availability of specific data sets—was achieved with a moderate work around. Environments using a relational database system to persist Event information captured by the WSLogA Framework may also use custom report engines, such as Crystal Reports (Business Objects, 2008) or Cognos (Cognos, 2008), to directly access the tables and records for efficient data shaping and retrieval.

The Event Group provides Enterprise systems with the flexibility of operational continuity of monitoring and response processes despite erroneous or fatal behavior in front end systems. For example, in the failing Web service demonstration scenario (Appendix C) the Adventure Builder application suffers significant component failure, yet the monitoring and response processes located in the JUnit process driving the demonstration remained effectively operational while using Event Group components to retrieve Event information and mark processed Event records.

The Perspective Component Group

The *Perspective Group* is comprised of those components that retrieve and normalize information managed by the Event Group and distribute it to response or reporting systems. The interfaces, classes, and resources for this group are defined in the *org.ws.loga.perspective*

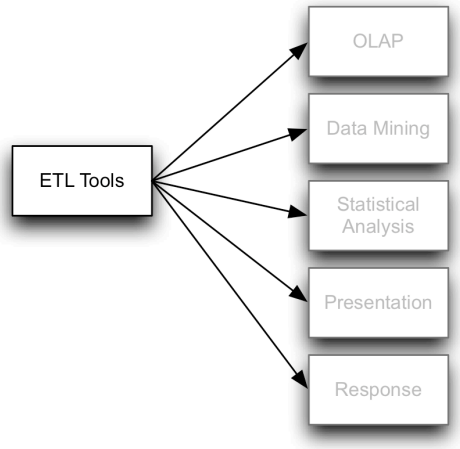


Figure 4-37. WSLogA elements addressed by the Perspective Group.

package. Figure 4-37 illustrates those portions of the WSLogA platform that are addressed by members of the Perspective Group with grey elements indicating boundary components. Figure 4-38 illustrates the use cases embodying these workflows. Appendix H documents the activities associated with each use case.

Roles and Responsibilities

Five roles were envisioned for the Perspective Group components for the purpose of co-ordinating and performing information retrieval and normalization for Response Group components or external system processes, such as reporting applications. Figure 4-39 illustrates the Perspective Group component roles and their relationships.

The Perspective performs the information retrieval and normalization tasks. It is provided with a resource reference to the Event information managed by the Event Group components and can establish queries for information retrieval. Perspective may also normalize information in terms of content or structure to ensure its suitability for consumption

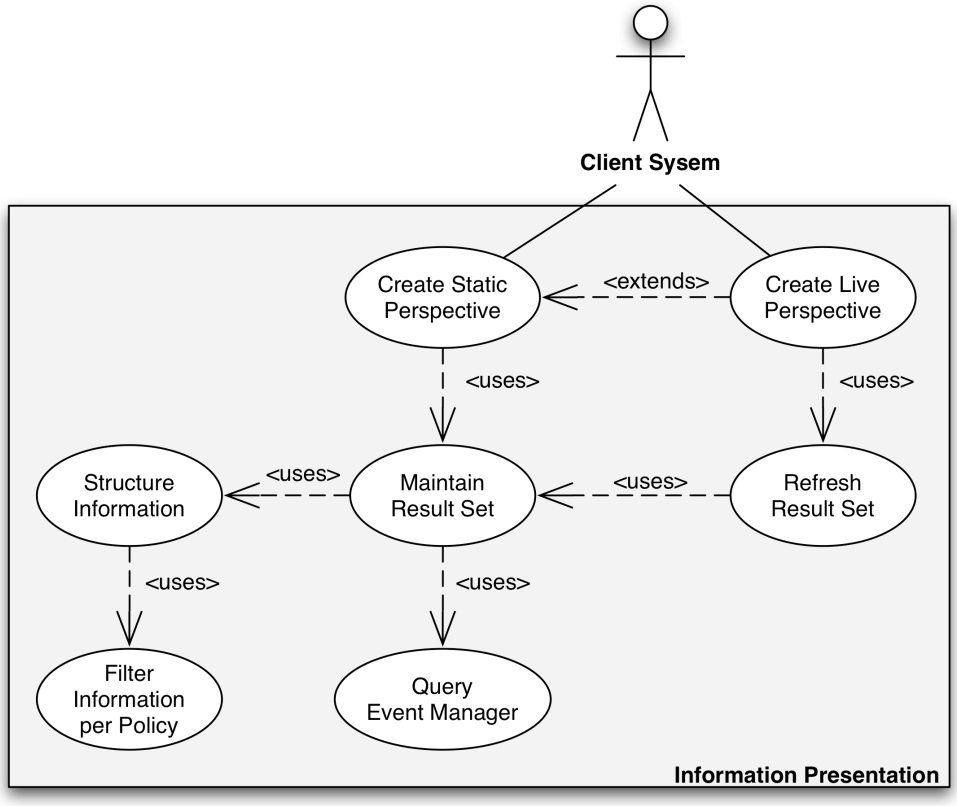


Figure 4-38. Use cases applicable to the Perspective Group.

by Response Group components or an external process such as a reporting system. A Perspective knows its preferred schedule for making Event information available to Event Processor objects, and can be dynamic (*e.g.,* multiple queries may be performed) or static (*e.g.,* only one query will be performed).

The Perspective Scheduler works with Perspectives and the Perspective Runner to ensure that Perspectives are submitted for execution at appropriate intervals. A Perspective Scheduler queries each Perspective to learn about its preferred schedule and then attempts to meet that schedule by submitting Perspectives ready for operation to a Runner. The Perspec-

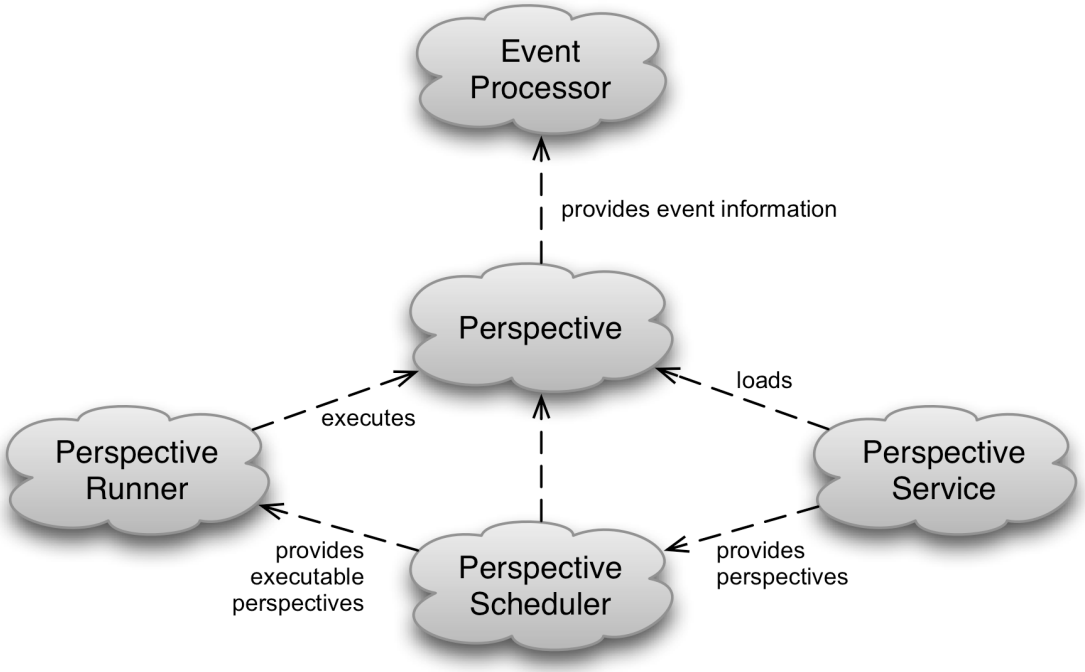


Figure 4-39. Perspective Group component roles.

tive Runner executes Perspectives in a manner suitable for the environment, and ideally in a concurrent manner.

Perspectives make their processed information available to Event Processors. The Event Processor may observe one or more Perspectives for updates to Event information, or another mechanism may be established by which the availability of information is communicated to the Event Processor.

The Perspective Service is made available for loading Perspectives, which may be useful for non-container processes, such as a daemon based on the WSLogA Framework. A configuration may be supplied to the Perspective Service, or Perspective characteristics may be predetermined by the service for specialized analysis systems.

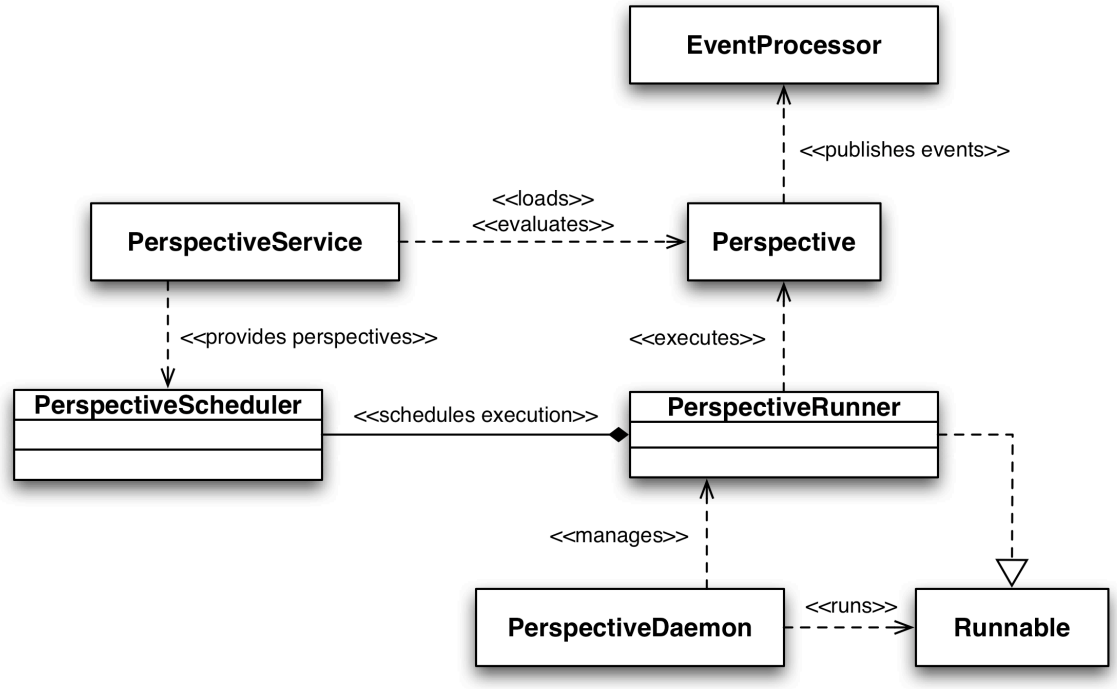


Figure 4-40. Perspective Group component structure.

Structure

The Perspective Group is principally structured using the Perspective interface and supporting components for loading, scheduling, and executing Perspective components, as illustrated in Figure 4-40.

The Perspective interface declares a Template Method (Gamma *et al.*, 1994) that implementing components define to obtain Event information, as well as method signatures for functionality required by the WSLogA Framework for managing the information retrieval and distribution workflows. This interface is appropriate for information distribution by which external systems are directly updated with Event information, although a complementary implementation of the Response Group's ResponseTask is appropriate for separat-

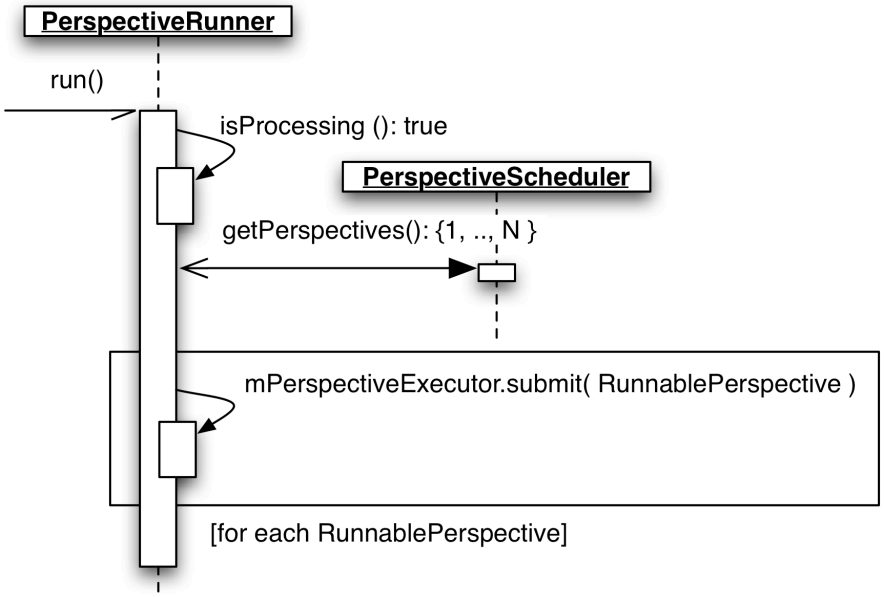


Figure 4-41. Principal perspective component interaction.

ing processing concerns. The WSLogA Framework distribution provides an enhanced Perspective that facilitates this separation of concern.

The PerspectiveRunner class manages the execution of Perspective objects, and works with the PerspectiveScheduler to identify Perspective instances that are ready to retrieve information. Both the PerspectiveRunner and PerspectiveScheduler are defined as concrete components as they are an integral bridge between the perspective and Event Groups. These components work intimately with Perspective objects to coordinate and perform information retrieval and distribution tasks. Figure 4-41 illustrates the principal sequence for the perspective components.

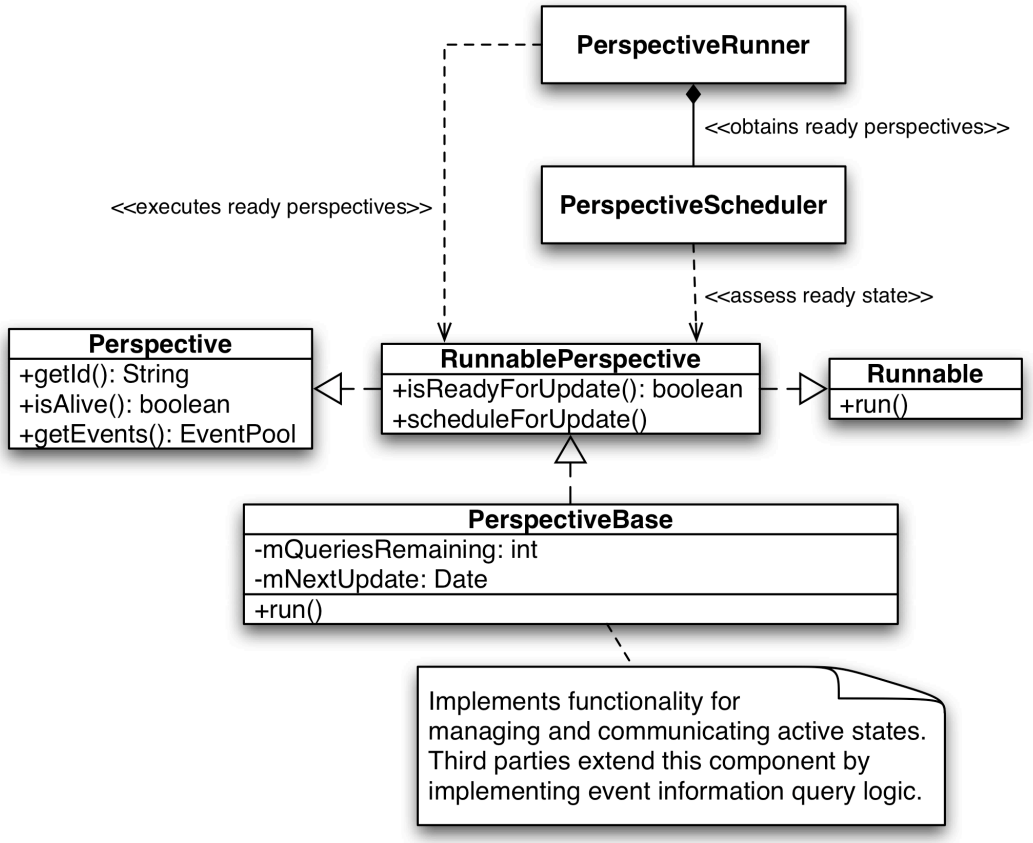


Figure 4-42. PerspectiveBase defines key behaviors for PerspectiveRunner integration.

Implementation

The Perspective interface implements the Runnable interface, which permits its threaded execution by PerspectiveRunner. The EventPool interface from the Event Group is used to track updated and normalized Event information for distribution among external systems or other WSLogA Framework components.

The PerspectiveBase abstract class implements the Perspective interface to provide the critical management functionality expected by the PerspectiveRunner and PerspectiveScheduler components. Applications creating custom information distribution workflows

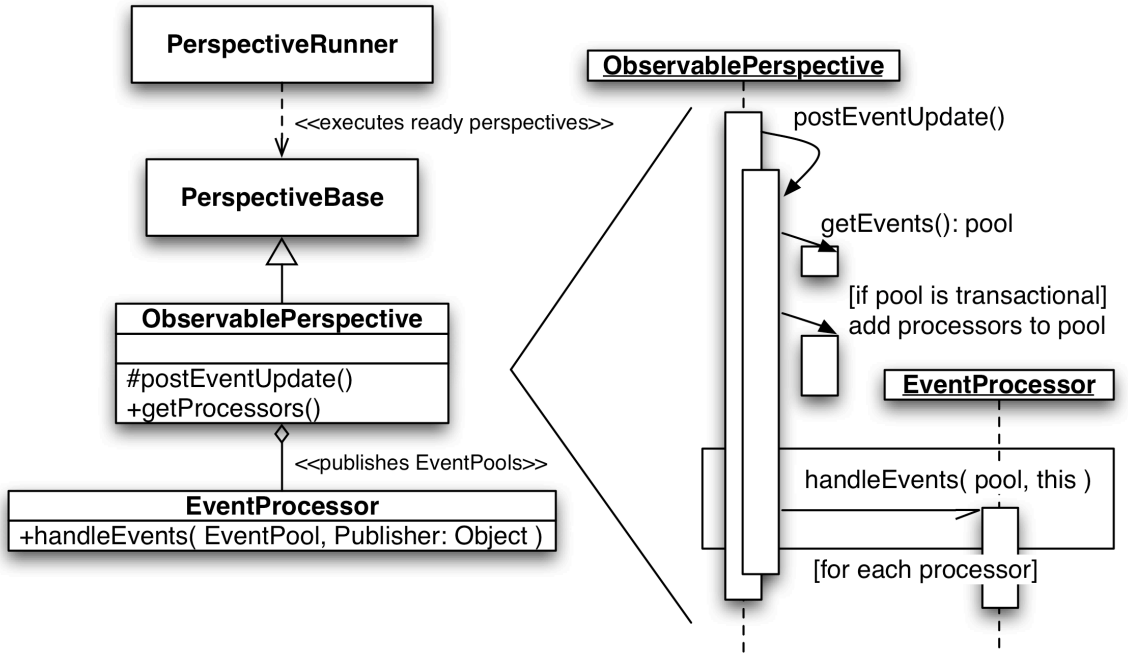


Figure 4-43. ObservablePerspective pushes event information to EventProcessors.

should extend the PerspectiveBase abstract class and implement the template information retrieval method to ensure compatibility with the WSLogA Framework workflows. Figure 4-42 illustrates the PerspectiveBase relationship with the PerspectiveRunner and PerspectiveScheduler components.

The ObservablePerspective abstract class extends PerspectiveBase and works in tandem with the EventProcessor interface to distribute normalized event information among consuming external systems or components. (The WSLogA Framework distribution implements the Response Group's ResponseTask as an EventProcessor to accommodate the standard analysis and response workflow.) Figure 4-43 illustrates the ObservablePerspective and EventProcessor relationship.

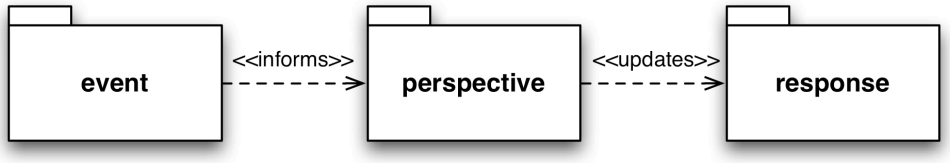


Figure 4-44. The Perspective Group relationships.

The PerspectiveRunner class executes Perspective components provided by the PerspectiveScheduler using ExecutorService (Goetz *et al.*, 2006). The PerspectiveScheduler makes use of the PerspectiveService component to identify Perspective objects ready to query the Event Group for event information updates, and makes the active Perspective objects available to the PerspectiveRunner.

Employment

The Perspective Group is integrated into the WSLogA Framework by serving to shape and make available Event information for use by reporting and analysis engines. The Response Group is structured according to the services provided by the Perspective Group, for which the ObservablePerspective and ResponseTask (an implementation of EventProcessor) component relationship is a prime example. The principal advantage of the Perspective Group is to provide Policy managed Event information shaping prior to its consumption by reporting and analysis engines, which is an important concern if sensitive information may be captured by the WSLogA Framework system (*e.g.*, social security numbers or customer habits). Figure 4-44 illustrates the relationship between the Perspective Group and other component groups as well as the environment.

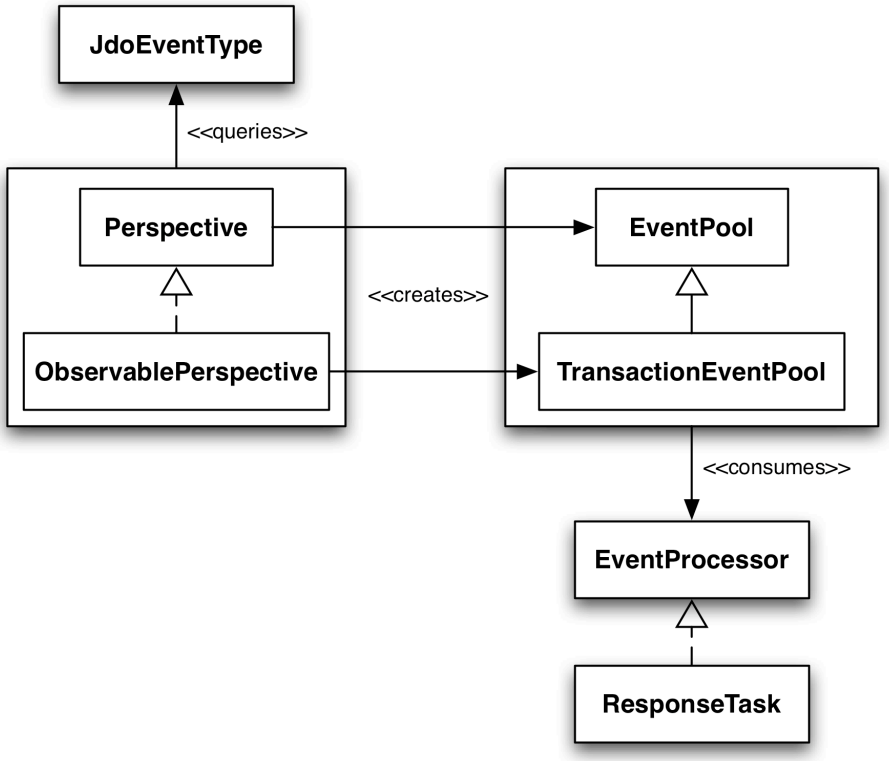


Figure 4-45. General employment of Perspective Group members.

The Perspective Group can be employed independent of all other WSLogA Framework components to provide basic information shaping and routing functionality; however, advanced features were implemented using elements of the Event Group and, as such, third parties should plan to adopt both component groups when evolving existing application architectures. Regardless of the degree of adoption, third parties must provide their own logic shaping the Event information retrieved. JDO integration provided by the Event Group is ideal for this purpose and the demonstrations provided as part of this investigation (Appendix C) use this strategy when preparing EventPool objects for use by ResponseTask

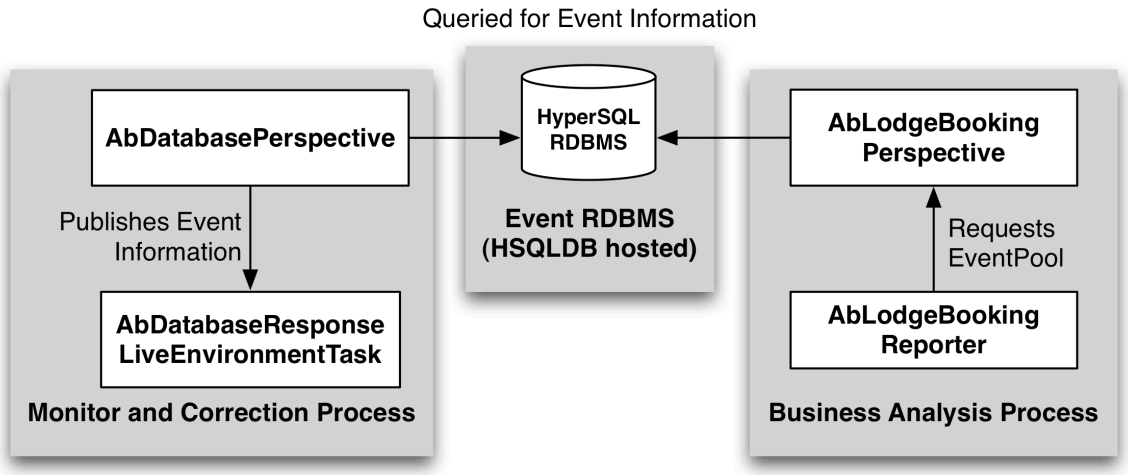


Figure 4-46. An example employment of the Perspective Group.

instances. Figure 4-45 illustrates the general relationships involved in the deployment of the Perspective Group.

The Perspective Group is demonstrated as the Event information shaping and provider mechanism for the Adventure Builder application (Appendix C). All of the scenarios involving event analysis use Perspective derivations to shape the Event information made available to Event Processors. The information capture demonstration uses a pull-based Perspective implementation in which the Perspective derivation only loads and prepares Event information upon the request of an external component. The failing Web service and failing database scenarios take advantage of push-based Perspective derivations that periodically load and prepare Event information and then push the Event information to observing Response Task based components. Figure 4-46 illustrates the usage of both push- and pull-based Perspective derivations.

Constraints and Opportunities

Information distribution is an important concern for reporting systems as well as systems responsible for ensuring proper application operation and performance across production environments. The Perspective Group provides Enterprise systems with a mechanism for retrieving the aggregated and correlated information from the persistent data store maintained by the Event Group, normalizing the information for consumption, and distributing the information to consuming processes.

The implementation strategy for the distributed WSLogA Framework perspective components provides a workflow that tightly integrates the information retrieval, normalization, and distribution tasks. Applications only need to extend PerspectiveBase with custom retrieval and normalization logic while still gaining the benefit of the controlled WSLogA Framework workflow. ObservablePerspective can also be extended to accommodate consumer registration, which minimizes the logistical tasks necessary to streamline the information distribution process.

The Perspective Group is designed with the assumption that members of the Event Group will be utilized to obtain information. As such, the mechanism for query management will depend on the technology driving the subset of persistent data classes providing information access. For example, the WSLogA Framework is distributed with JDO enabled information management, which is excellent for organic data models and linear data access but is still limited in the types of complex queries possible for retrieving specific data subsets. Perspective components for systems in which the information management technology could change should make use of Proxy and Strategy patterns (Gamma *et al.*, 1994) to

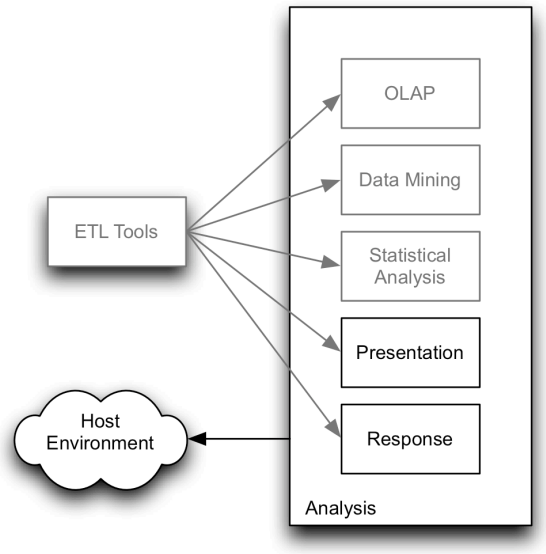


Figure 4-47. WSLogA elements addressed by the Response Group.

delegate information retrieval to components that may be easily substituted without the need for reworking the Perspective's principal logic.

The Response Component Group

The *Response Group* is comprised of those components that process information retrieved by Perspective Group components and manage the application or environment in response to the analysis results. The interfaces, classes, and resources for this group are defined at the *org.ws.loga.response* package. Figure 4-47 illustrates those portions of the WSLogA platform that are addressed by members of the Response Group with grey elements indicating boundary components. Figure 4-48 illustrates the applicable use cases. Appendix H documents the activities associated with each use case.

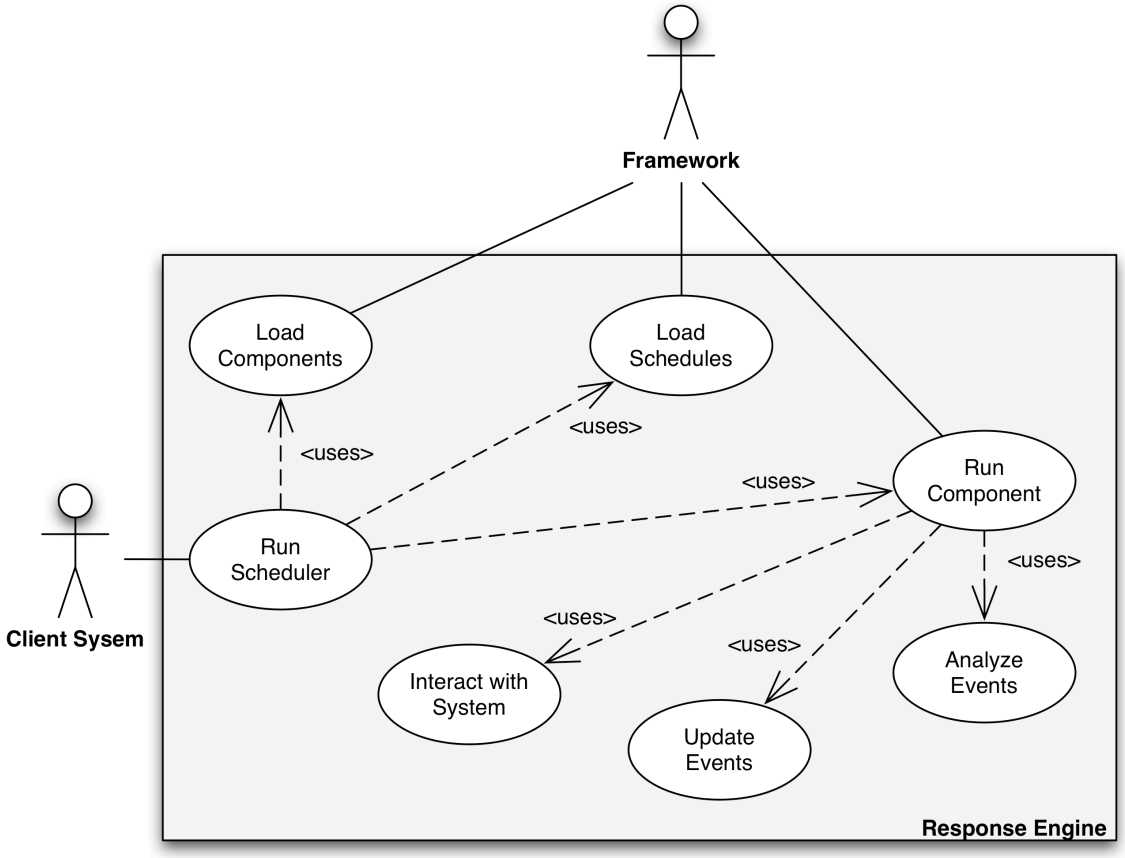


Figure 4-48. Use cases applicable to the Response Group.

Roles and Responsibilities

Four roles were envisioned for the Response Group components for the purpose of accepting Event information from Perspective Group components, analyzing the obtained information, and making environment adjustments in response to the analysis results. Figure 4-49 illustrates the Response Group component roles and their relationships.

The Response Task manages the analysis of Event information and effects change in the application or environment in response to the analysis result. The Response Task provided for distribution with the WSLogA Framework is envisioned as a form of the Event Group's

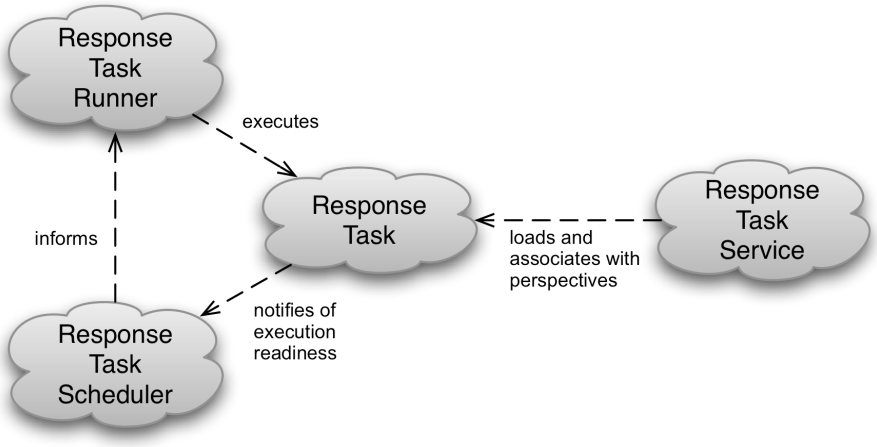


Figure 4-49. Response Group component roles.

Event Processor role, but conceivably any consumer of Event information made available by the WSLogA Framework could serve as a Response Task. The Response Task may also directly work with Event Group components to add metadata markers regarding information provided by the associated Perspective, such as to indicate that the Response Task has already processed specific Events (Brett, 2005).

Response Task components wait to receive updated event information from a Perspective when manifested as a specialized form of the Event Processor defined as part of the Perspective Group. The Response Task notifies the Response Task Scheduler upon receiving updated Event information, and the Response Task Scheduler works with the Response Task Runner to execute the Response Task at an appropriate time.

The Response Task Service is provided to facilitate Response Task loading and configuration, such as to associate Response Tasks with Perspectives. The Response Task Service may

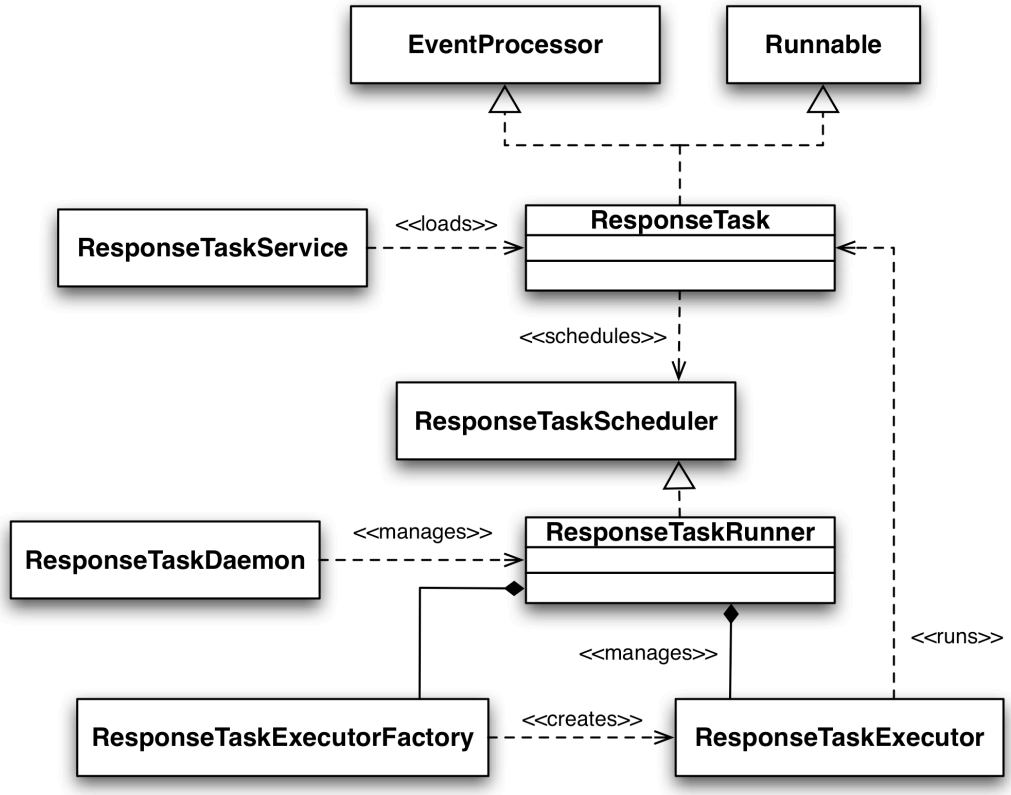


Figure 4-50. Response Group component structure.

be particularly useful for processes operating outside of an Application server, such as a production environment control system.

Structure

The Response Group is principally structured using a relationship between a component representing the Response Task and components managing the scheduling and execution of the Response Task, as illustrated in Figure 4-50.

The ResponseTask abstract class represents work to be performed in response to the application's analysis of the Event information as harvested by the WSLogA Framework.

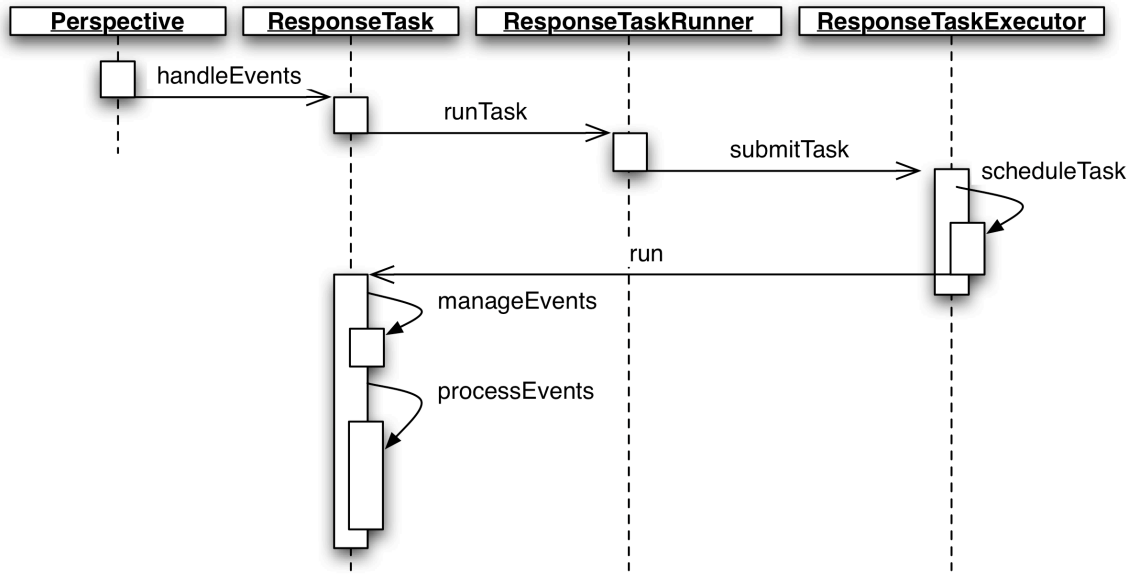


Figure 4-51. Principal response component interaction.

ResponseTask is defined as a Runnable (Arnold *et al.*, 2005) component to facilitate the simultaneous execution of multiple tasks, and the component implements the EventProcessor interface provided by the Perspective Group to enable its consumption of information provided by Perspective components. The WSLogA Framework manages ResponseTask objects after their instantiation, which permits adopting systems to focus on the business logic driving system stability and reporting.

The ResponseTaskScheduler interface is responsible for scheduling the execution of ResponseTasks upon being notified by ResponseTasks that they are ready to process Event information or perform environment management. Implementations of ResponseTaskScheduler permit flexibility in how system resources are distributed to handle responses (Helsinger *et al.*, 2003; Lee *et al.*, 2002). For example, an application could implement a Scheduler that gives priority to system maintenance tasks over tasks generating reports.

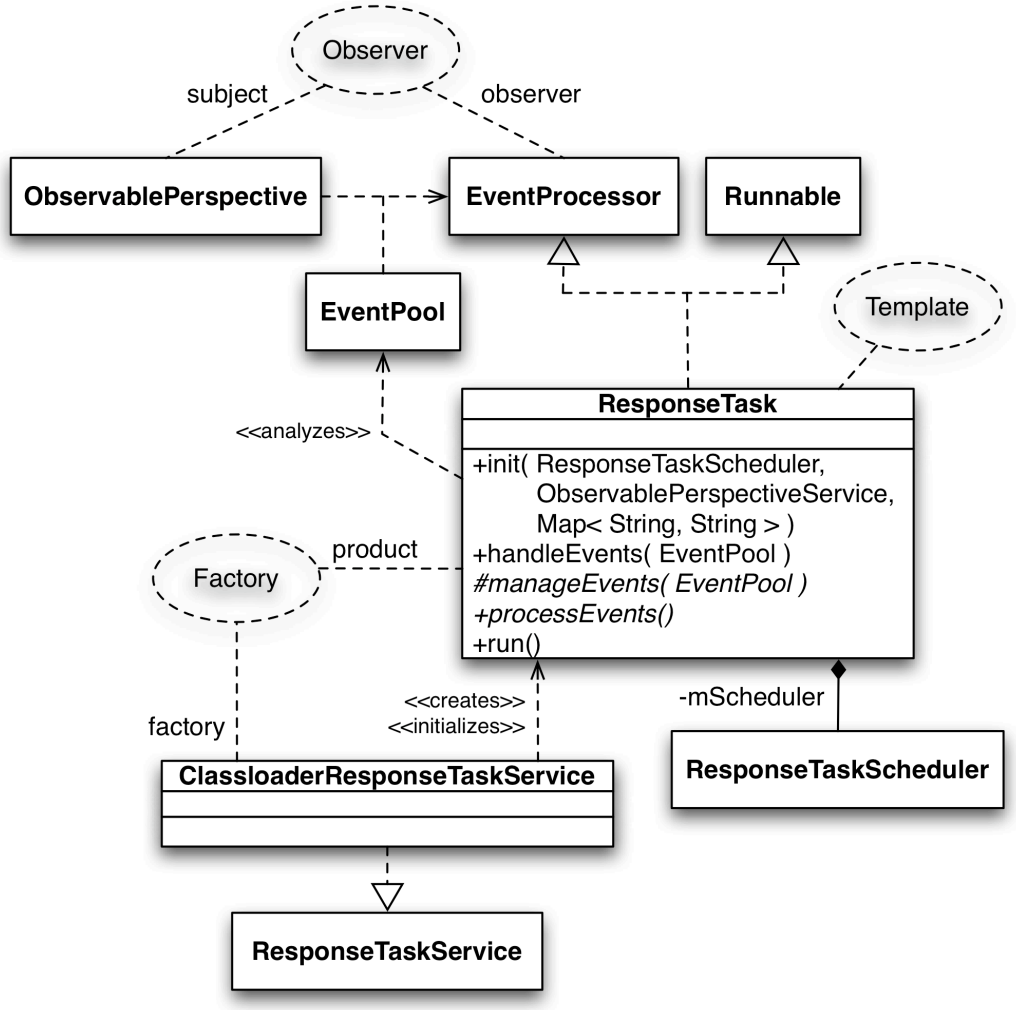


Figure 4-52. The ResponseTask organizes response behavior.

The ResponseTaskRunner class implements the ResponseTaskScheduler interface and is responsible for the execution of ResponseTask objects. ResponseTaskRunner delegates operation of ResponseTask instances to ResponseTaskExecutors, which are obtained from a ResponseTaskExecutorFactory. This delegation ensures that applications have the ability to choose a ResponseTask management strategy appropriate for the system's response and resource requirements. For example, servers with significant operating resources (*e.g.*, RAM)

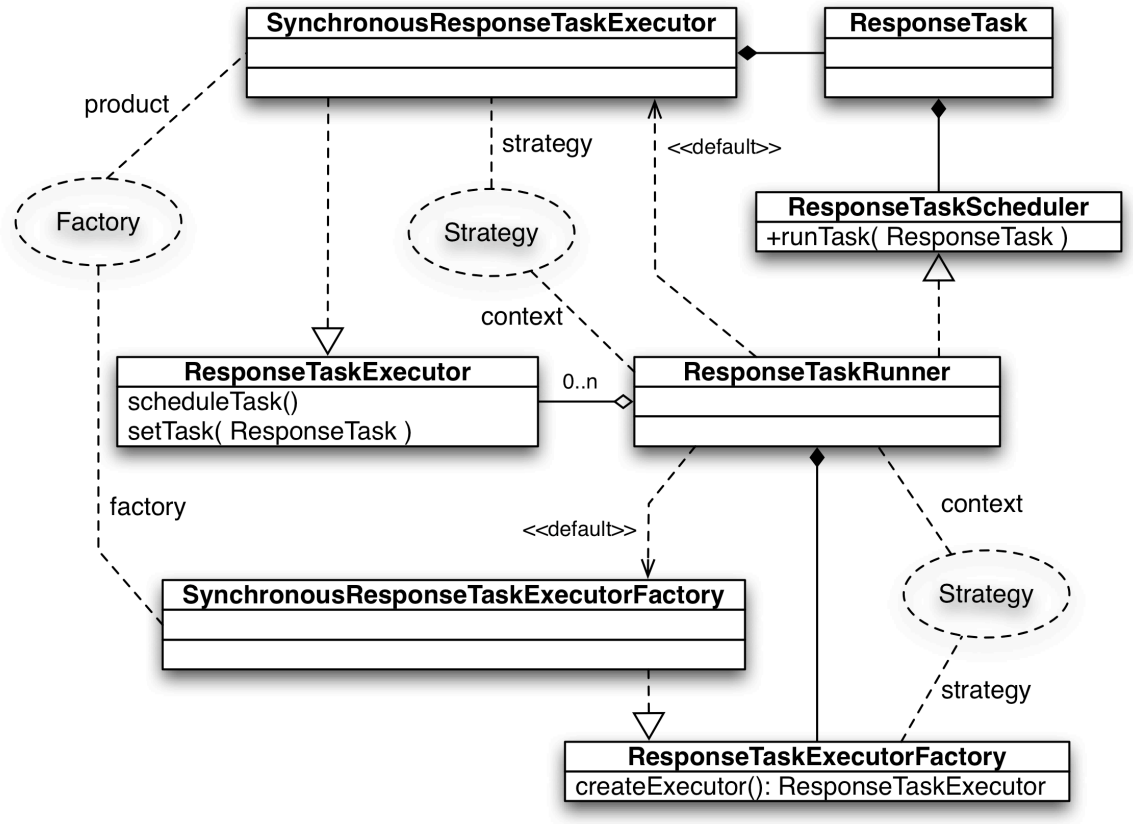


Figure 4-53. The ResponseTaskRunner drives response activities.

may be able to handle the ResponseTask instances within a single, local JVM; however, systems could instead implement a ResponseTaskExecutor that distributes ResponseTask execution among nodes within a grid (Helsinger *et al.*, 2003; Lee *et al.*, 2002).

Figure 4-51 illustrates the relationships among the ResponseTask, ResponseTaskScheduler, and ResponseTaskRunner components. Adopting systems only need to implement the ResponseTask component to take advantage of the default Event information processing and response workflow.

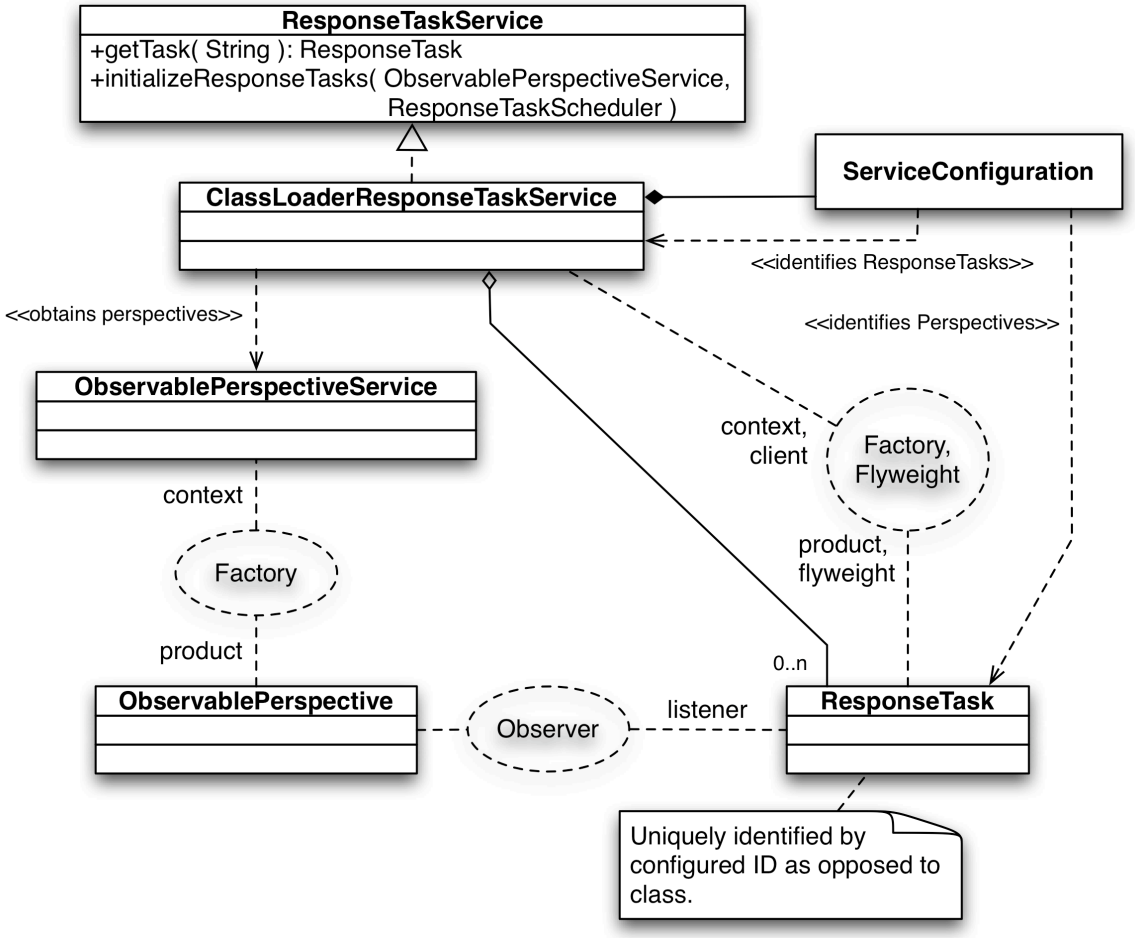


Figure 4-54. The ResponseTaskService loads ResponseTasks.

Implementation

The Response Group is implemented as three packages addressing ResponseTask, ResponseTaskService, and daemon components that include the ResponseTaskRunner. All of the components except ResponseTask have definitions that control the workflows necessary to facilitate typical analysis and response operations in Enterprise system contexts. Applications only need to implement ResponseTask and associate the derived component with a Perspective to benefit from the default workflow.

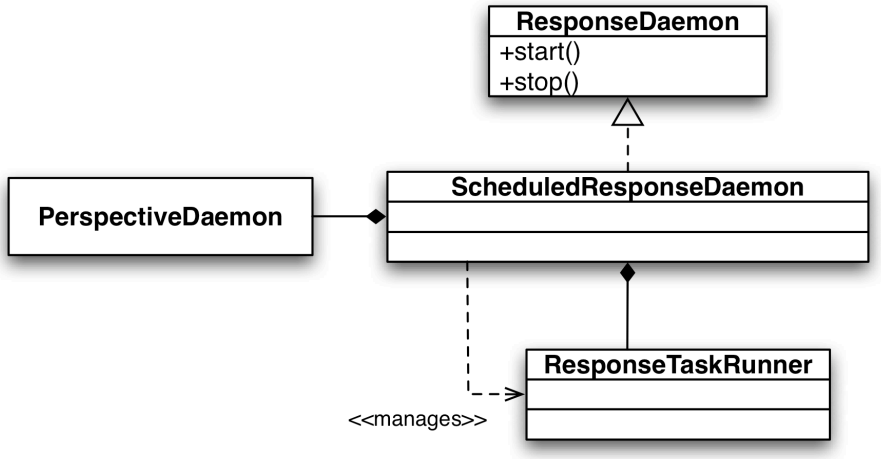


Figure 4-55. The ResponseTaskDaemon provides an operational entry point

The ResponseTask is defined as an abstract class with a Template Method (Gamma *et al.*, 1994; Shalloway & Trott, 2001) for information analysis and response logic. Common functionality for associating a Perspective and guarding against redundant execution is provided (*e.g.*, processing Event information while a previous update is still being processed). This strategy permits extending components to focus on the business logic and the management of related resources, such as a JMX component (L. McGregor, 2003; McManus, 2002; McManus & Vienot, 2003). Figure 4-52 illustrates the ResponseTask component.

The ResponseTaskRunner is defined as a class that works in conjunction with a provided Strategy component (Alur *et al.*, 2003; Gamma *et al.*, 1994; Shalloway & Trott, 2001) to manage the scheduling and execution of ResponseTask objects. ResponseTaskRunner implements the ResponseTaskScheduler interface to facilitate ResponseTask registration, and a ResponseTaskService may establish this association with ResponseTask instances during their initialization. Varied ResponseTask management behavior is enabled through the use of

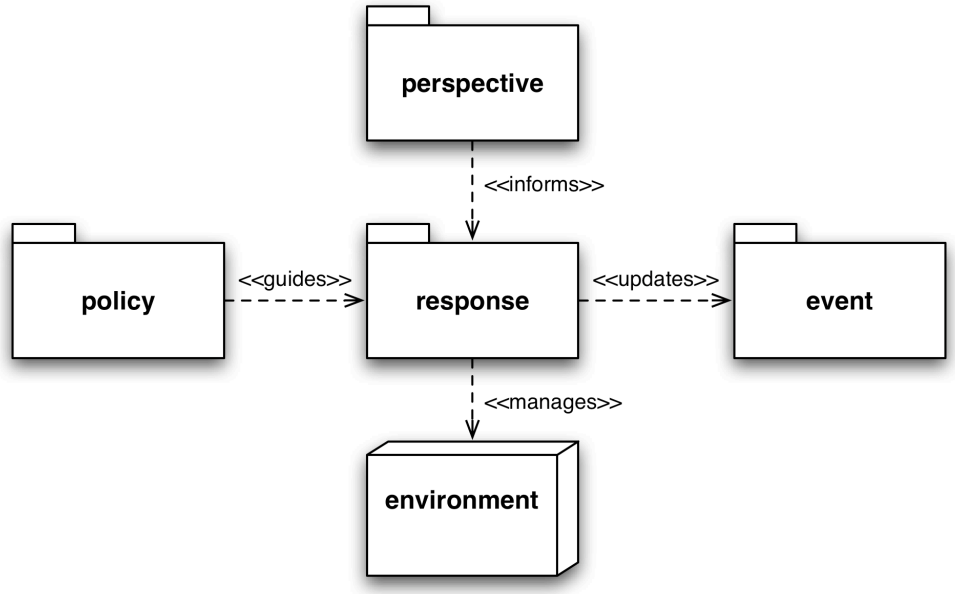


Figure 4-56. The Response Group relationships.

an associated ResponseTaskExecutorFactory component that produces ResponseTaskExecutor instances that serve as ResponseTask management proxies (Gamma *et al.*, 1994; Shalloway & Trott, 2001). The default behavior is to execute synchronously each ResponseTask against its queue of assigned Events for processing. Third parties can vary this behavior, including the execution of the ResponseTasks in foreign JVMs, by providing alternate implementations of ResponseTaskExecutorFactory and ResponseTaskExecutor. Figure 4-53 illustrates the ResponseTaskRunner component.

The ResponseTaskService is defined as an interface, and is responsible for providing ResponseTask objects for use in the Event analysis and environment management process. ClassLoaderResponseTaskService implements the WSLogA Framework's default ResponseTaskService, and may be used as a Factory (Gamma *et al.*, 1994; Shalloway & Trott, 2001) to produce ResponseTask instances from classes available to the JVM and initialize each task by

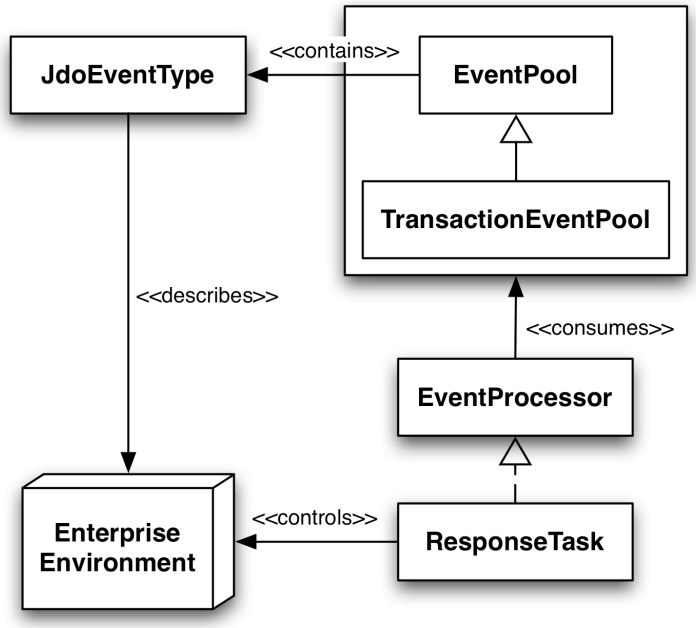


Figure 4-57. General employment of Response Group members.

providing references to the associated ObservablePerspective (Perspective Group) and ResponseTaskScheduler. Figure 4-54 illustrates the ClassLoaderResponseTaskService.

The ResponseTaskDaemon is defined as an interface, and is responsible for managing an Event analysis and environment management process based on Response Group components. ScheduledResponseDaemon implements the WSLogA Framework's default ResponseTaskDaemon, and may be used to operate a ResponseTaskRunner using ResponseTask implementations available to the host JVM. Figure 4-55 illustrates the ResponseTaskDaemon.

Employment

The Response Group is integrated into the WSLogA Framework by serving as the endpoint for Event information organized and provided by Perspective components, and by refining the Event information pool made available to WSLogA Framework components

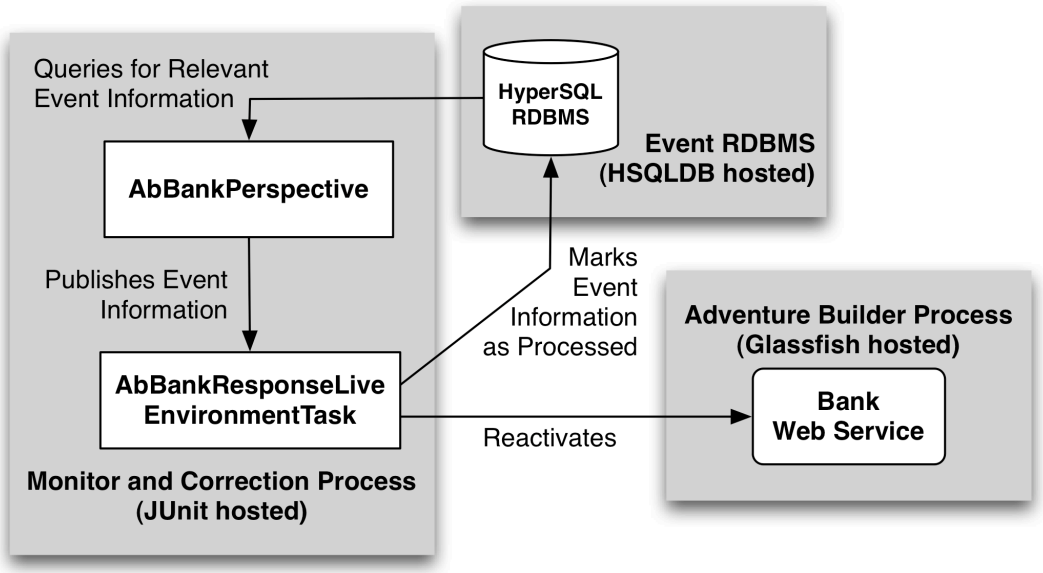


Figure 4-58. An example employment of the Response Group.

through updates to Event metadata in result of Event analysis. Policy components can be integrated into ResponseTask logic to enact behavior such as determining when environment interaction should be performed based on Event pattern observation or ensuring that resultant reports only include information for which the audience is authorized. The Response Group facilitates environment management by means of the processEvents method provided by the extensible ResponseTask component. No default implementations are defined, but extensions to the WSLogA Framework providing such implementations could be developed as common response requirements are identified for specific architectures (*e.g.,* Web services running on GlassFish application servers). Figure 4-56 illustrates the relationship between the Response Group and other component groups as well as the environment.

The Response Group can be employed independent of most other WSLogA Framework components; although, such architectures should consider also using Perspective and EventPool components to facilitate Event information transfer from the Event information pool. This strategy permits independent marshalling and exposure of Event information, which may be critical to ensuring that only authorized consumers of the Event information (represented by ResponseTask implementations) have access to information Perspectives (Lai *et al.*, 2005; Larson & Stephens, 2000; Monson-Haefel, 2004). The analysis and response system should also remain external to the J2EE application being managed to ensure the proper operation of Response Group members in the event of failure within the application (Garlan & Schmerl, 2002; Helsinger *et al.*, 2003; Lee *et al.*, 2002). Figure 4-57 illustrates the general relationships involved in the deployment of only the Response Group and essential associated components.

The Response Group is demonstrated as the Event analysis and environment management mechanism for the Adventure Builder application (Appendix C). Scenarios such as the failing Web service example employ the Response Group components by means of a process (JUnit) operating externally to the Adventure Builder application. Analysis components extending ResponseTask are provided as appropriate for each example, and a GlassFish management component is defined for environment interaction. The ResponseTask components receive Event information from corresponding Perspective components and are scheduled using the default strategies provided with the WSLogA Framework. If appropriate, environment interaction is provided in response to the analysis outcomes. Figure 4-58 illustrates an example workflow involving the Response Group.

Constraints and Opportunities

System maintenance is an important concern for any distributed system, and Web service environments such as that typified by the combination of the Adventure Builder application and the GlassFish application server are equally susceptible to Enterprise environment issues. The Response Group provides Enterprise systems with a mechanism for analyzing information aggregated from multiple sources and executing environment or application adjustments in response (Dashofy *et al.*, 2002)—in effect, the Response Group provides a suitable foundation for the development of self-healing systems. For example, in the event a database pool fails a ResponseTask component could realize the failure and restart the database pool.

The ResponseTaskDaemon and ResponseTaskRunner components permit the externalization of Response Task operations, which accommodates holistic pattern analysis using both internal (*e.g.*, application generated) and external (*e.g.*, router log file) information (Garlan & Schmerl, 2002; Wang, 2005). Further, failure within the application or its immediate host, the application server, does not prevent error recovery from initiating. For example, Handler (Graham *et al.*, 2005) components are associated with specific Web services in GlassFish, which means that an inactivated Web service prevents its corresponding Handlers from recording transaction events; however, use of the ResponseTaskDaemon and its associated components ensures that error recovery is performed (Appendix C).

The concentration of information analysis and response operation into the Response Group permits the centralization of policies and rules regarding environment management (Wang, 2005). As a result, sub-frameworks specialized for use with the WSLogA Framework can be developed to accommodate reusable self-healing system logic for similar architectures

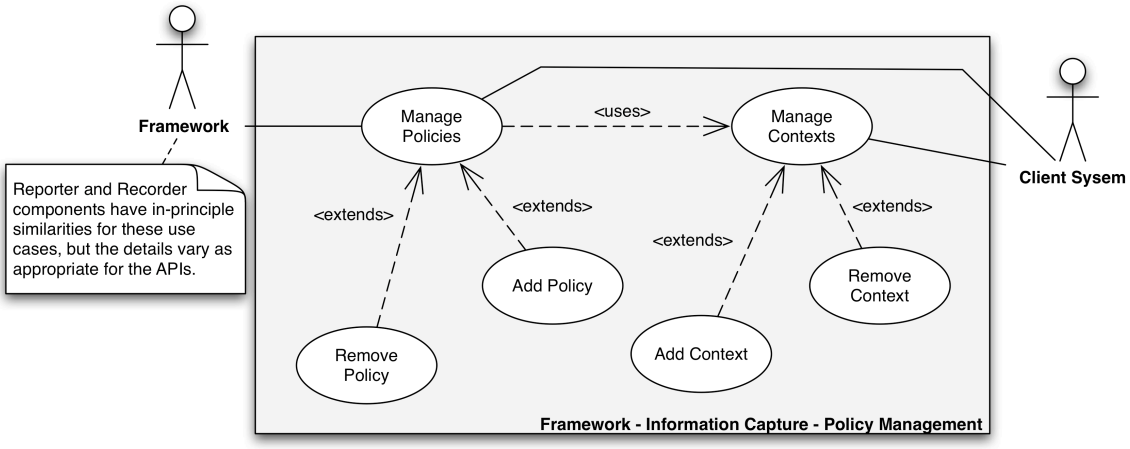


Figure 4-59. Use cases applicable to the Policy Group.

across disparate applications. For example, the application server management logic for failed components, such as Web services and databases, should work similarly for the WebLogic application server regardless of the implementation details for Web services or their Handlers (Graham *et al.*, 2005) deployed within WebLogic.

The implementation strategy for the distributed WSLogA Framework Response components provides a workflow that tightly integrates the information retrieval, analysis, and response tasks. Applications only need to implement a ResponseTask component with the appropriate business rules (and ensure a suitable Perspective component is available) to take immediate advantage of these features. However, systems can take advantage of the flexibility provided by the ResponseTaskScheduler and ResponseTaskRunner interfaces to define extraordinary resource management in resource sensitive systems, such as those that must ensure real time responses.

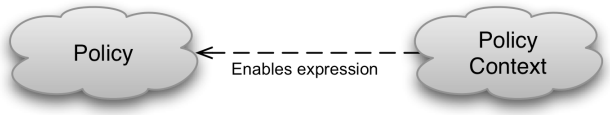


Figure 4-60. Policy Group component roles.

The Policy Component Group

The *Policy Group* is comprised of those components facilitating the expression of business rules affecting information management (Wang, 2005). Policy contexts are defined to represent behavior within the context of system, legal, or cultural boundaries that determine policy expression, which enables flexible adjustment of the system's behavior without the modification of principal workflows. The interfaces, classes, and resources for this group are defined at the *org.ws.loga.policy* package. Figure 4-59 illustrates the use cases embodying these workflows. Appendix H documents the activities associated with each use case.

Roles and Responsibilities

Two roles were identified from the use cases for the purpose of representing a framework policy and contexts in which that policy could operate. Figure 4-60 illustrates the Policy Group component roles and their relationships.

Policy components can manifest behavior that confirms acceptance of a process or task, or the objects can act upon information in manners that normalize the content or structure to make it acceptable for specific contexts. For example, an acceptance Policy could indicate whether a social security number should be recorded as part of the Event information stream; a formatting Policy could substitute the character 'x' for a social security number's

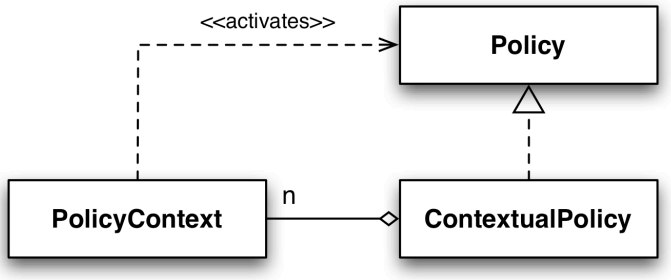


Figure 4-61. Policy Group component structure.

first five digits to mask the significant parts of the social security number enabling the unique identification of an individual.

Policy Context components confirm whether specified Policy objects should express their behavior. For example, a Policy Context could represent an account type of interest—such as that for a European customer—in which Policies enabling information capture rules adhering to strict privacy standards will be approved for expression. Policy Context components can also represent workflow phases for which general customization of information management should take place regardless of legal or cultural considerations, such as whether the business prefers to document the time required for transactions.

Structure

The Policy Group is principally structured using a simple relationship between two interfaces and a helper abstract class, as illustrated in Figure 4-61.

The Policy interface represents the Policy role, and as such serves as a proxy for the behavior or state implied by the rules defining the Policy. The Policy interface permits components—including the Policy—to assert whether the Policy's rules will be executed if the

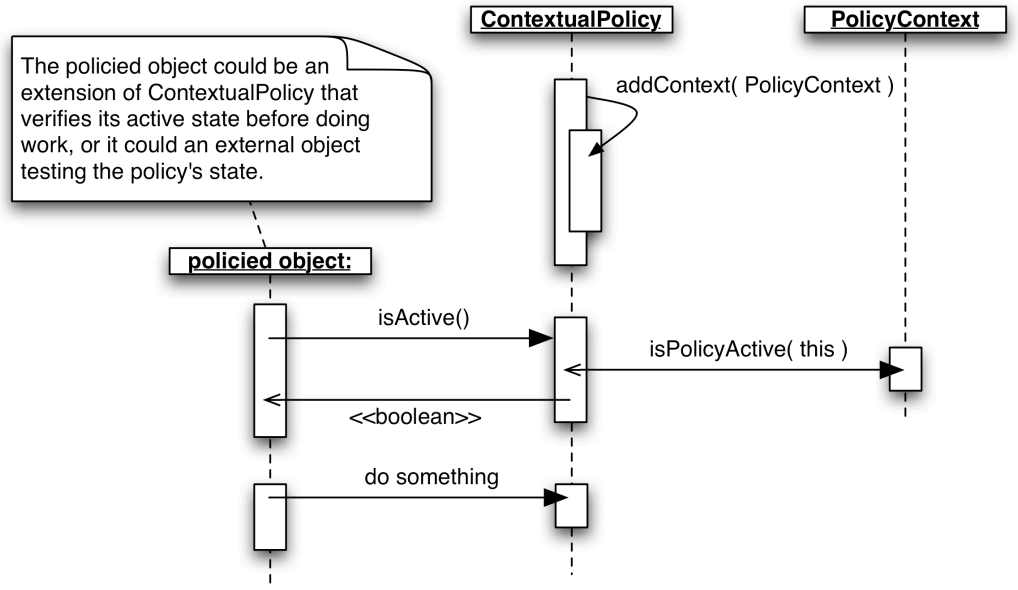


Figure 4-62. Principal policy component interaction.

Policy is invoked. Implementing components must validate the availability of resources necessary for Policy execution.

The PolicyContext interface represents the Policy Context role, and as such serves as a proxy for the evaluation affecting a Policy component's ability to operate. Implementing components have several options for evaluating a Policy's active state. The Policy object can be inspected, such as through reflection (Arnold *et al.*, 2005), to determine if its attributes warrant the Policy's activity as an instance-specific consideration; for example, a context may enforce the rule that only Policies established by the local JVM may execute and foreign Policies must not execute. The environment can also be assessed to determine activity; for example, the context could be associated with a specific language (*e.g.*, French) and only permit Policy execution if that language is active for the client.

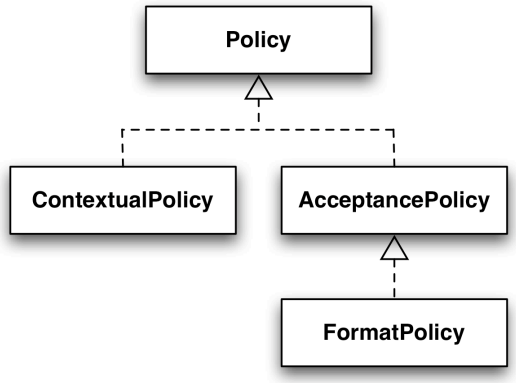


Figure 4-63. Filter, format, and contextual policy specializations.

The ContextualPolicy abstract class provides the functionality necessary to manage the association of PolicyContext objects with a Policy object. This permits PolicyContexts to be aggregated with a Policy that may traverse the system as part of a transaction, and with an appropriate remote procedure call implementation the PolicyContexts could even be transferred to remote systems. The WSLogA Framework defines the workflow by which ContextualPolicy objects will consult associated PolicyContext objects to determine whether the ContextPolicy should be active. Figure 4-62 illustrates the principal sequence for the Policy components.

Implementation

The WSLogA Framework implements the Policy Group as two distinct packages that respectively address specialized Policy or PolicyContext behavior and state as necessary for general use. Extending applications may build upon the structure defining interface components or override Template Methods (Gamma *et al.*, 1994) within the specialized classes, such as that provided by AcceptancePolicy.

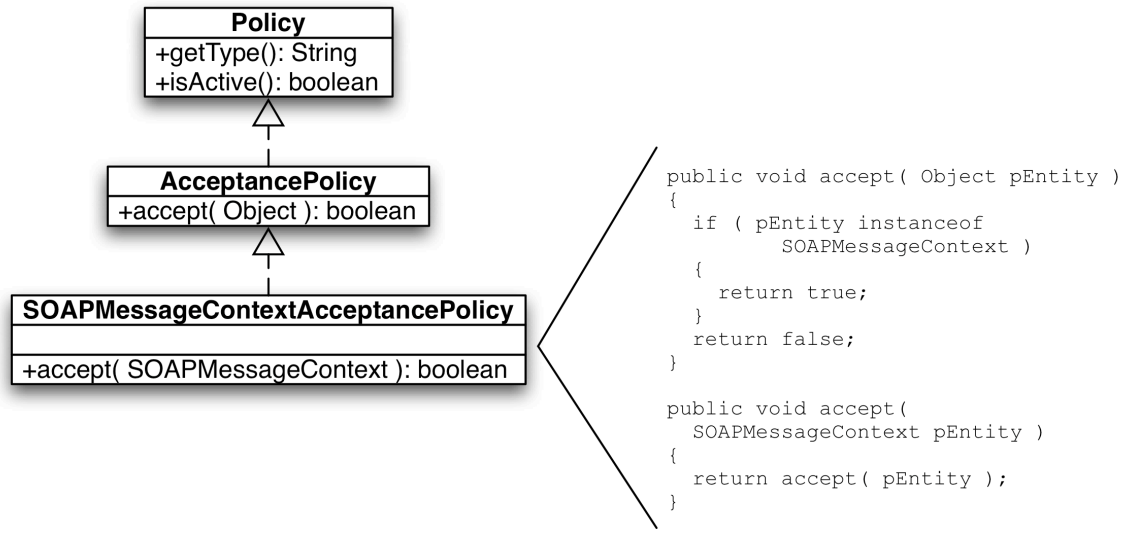


Figure 4-64. Policy roles are indicated by means of interface implementation.

The Policy interface is the archetype of components manifesting or directly supporting the Policy role. Extending implementations are accommodated through Template Methods (Gamma *et al.*, 1994) intended for the highly cohesive expression of business rules. Generics (Arnold *et al.*, 2005) are employed to provide compile time distinction between affected entities and, if appropriate, the results of Policy operations. Figure 4-63 illustrates the WSLogA Framework components derived from the Policy interface.

The preferred method for introducing a policy pattern into the WSLogA Framework is to declare the intended behavior as a Template Method (Gamma *et al.*, 1994) for an interface derived from Policy. The use of Policy as a type marker clearly communicates the derived component's intent and enables convenient organization of the derived components within the package hierarchy and collections. Implementing components can manifest the Policy rule by defining logic for the Template Method. Entities operated upon by the Policy-derived component are injected into the object as necessary to satisfy a process' choice in behavior.

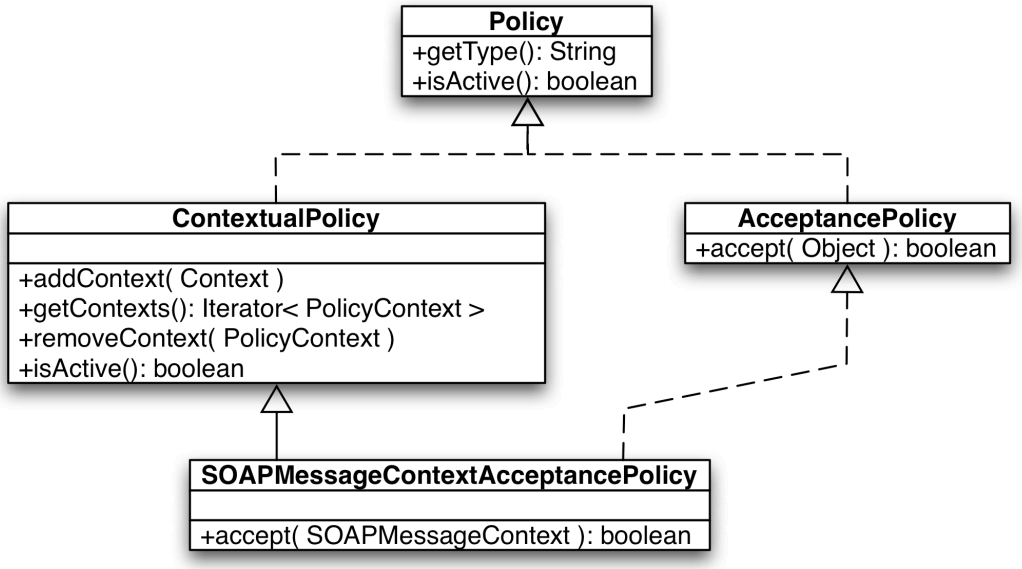


Figure 4-65. Policy context management is provided by means of class extension.

For example, a reporting component could provide a formatting Policy with a social security number so that the digits could be masked according to the Policy needs. A default Policy might be to leave the social security number in its raw form. The AcceptancePolicy and FormatPolicy components follow this strategy for Policy implementation, and serve as the foundation for members of the Monitor Group that facilitate reporting processes. Figure 4-64 illustrates this relationship using the AcceptancePolicy component.

The ContextualPolicy abstract class is derived from Policy to communicate its role as a Policy component. However, rather than provide templates for policy patterns, the ContextualPolicy class provides functionality for aggregating Policy contexts with transient Policy objects. Policy pattern components, such as AcceptancePolicy, are declared as interfaces, so the functionality provided by ContextualPolicy can be made available to WSLogA Framework or application Policy components with the creation of a new class extending Contextual-

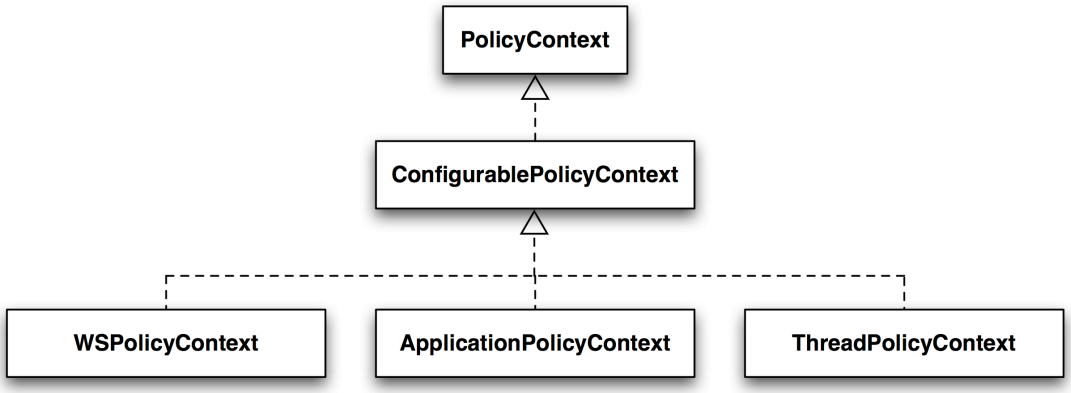
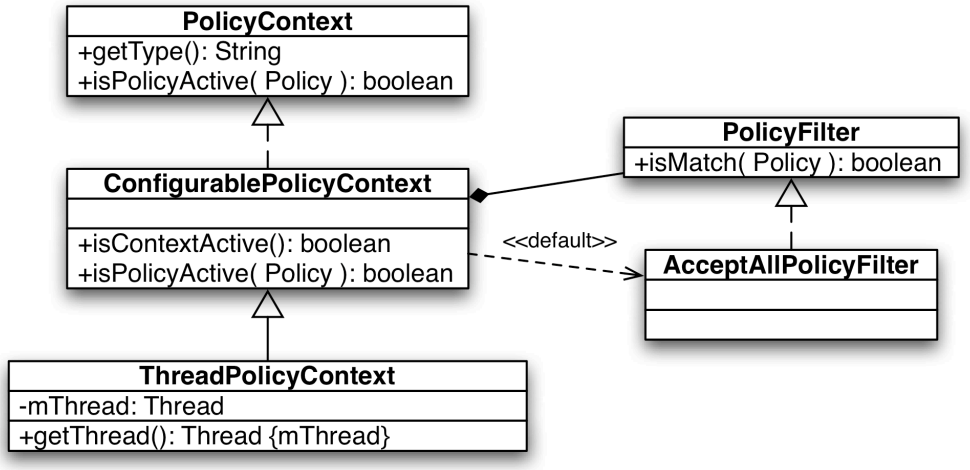


Figure 4-66. PolicyContexts provide scenario based policy activation.

Policy that also implements the desired Policy-derived pattern components. Figure 4-65 illustrates the use of ContextualPolicy to enhance Policy components distributed with the WSLogA Framework.

The PolicyContext interface is the archetype of components representing environments for which Policies may be active. Extending implementations are facilitated through the use of ConfigurablePolicyContext, which makes use of strategy components for evaluating provided Policy objects and provides a Template Method (Gamma *et al.*, 1994) for use in determining whether the context is active and can assess Policies. Figure 4-66 illustrates the WSLogA Framework components derived from the PolicyContext interface.

ConfigurablePolicyContext is provides functionality for assessing whether provided Policy objects are active, and for determining whether the context is active and can make such assessments. Policy evaluation is delegated to a PolicyFilter component, which permits specialized contexts to be developed that make similar policy evaluations within their scope. For example, the ApplicationPolicyContext considers Policies within a global system scope




```
// Member of ConfigurablePolicyContext
public boolean isPolicyActive( Policy pPolicy )
{
  return isContextActive() && getPolicyFilter().isMatch( pPolicy );
}

// Member of ThreadPolicyContext
public boolean isContextActive()
{
  return getThread() == Thread.currentThread();
}
```

Figure 4-67. ConfigurablePolicyContext facilitates ad hoc context definitions.

but ThreadPolicyContext only evaluates Policies that are within the scope of a specific thread; however, using the PolicyFilter strategy each context could be set to evaluate only those Policies for formatting credit card numbers. Extending components can add logic to assess whether the context may be considered active by overriding the isContextActive template method (Gamma *et al.*, 1994). For example, ApplicationPolicyContext is always considered to be active, but ThreadPolicyContext is only considered active when operated within a specified thread. Figure 4-67 illustrates the ConfigurablePolicyContext and how it facilitates behavior for the distributed WSLogA Framework components.

Management components are also provided by the WSLogA Framework to facilitate Policy and PolicyContext association for other component groups. For example, the Monitor

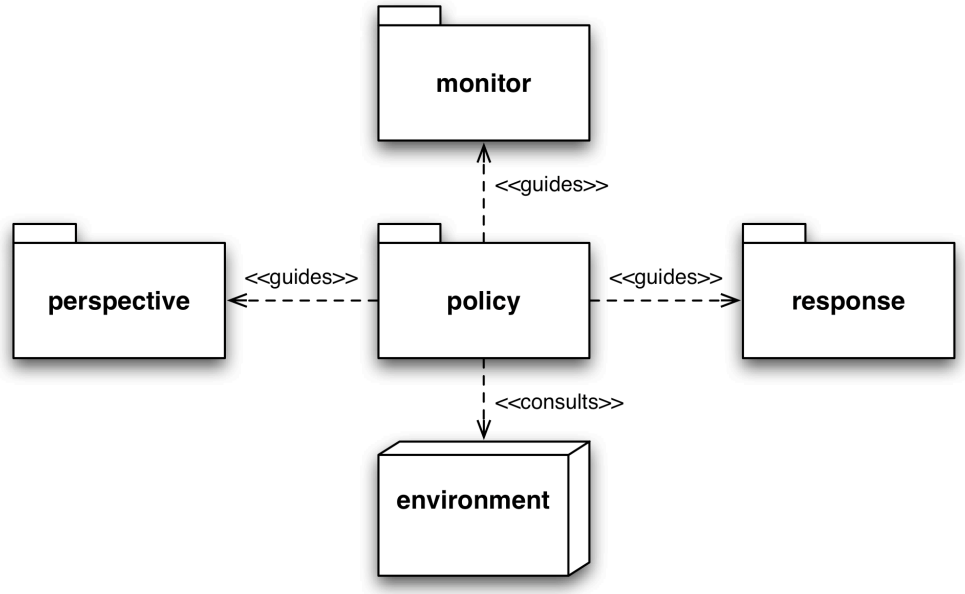


Figure 4-68. The Policy Group relationships.

Group uses PolicyManager for its reporting components to enable Policy association, and ContextualPolicy uses PolicyContextManager to enable PolicyContext association.

Employment

The Policy Group is integrated into the WSLogA Framework by serving as the gateway for information transfer among component groups, the associated persistent storage system, and client systems such as those used to prepare reports. The Monitor and Perspective Groups are structured with the Policy Group's functionality in mind, and PoliciedObserver provides an example of how information filtering and flow control has been established within the WSLogA Framework as a fundamental architectural element. The Policy Group provides the flexibility required of Enterprise applications deployed to disparate jurisdictions in that rules for information transfer and formatting can be expressed universally (*e.g.*, with

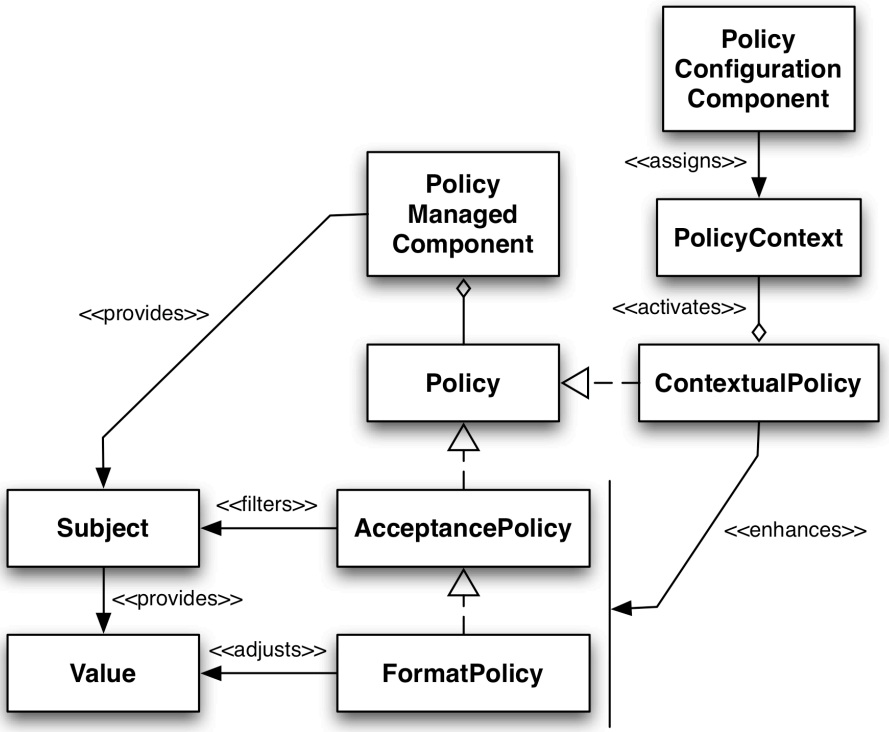


Figure 4-69. General employment of Policy Group members.

static or hard checkpoints) or per-environment through a variety of configuration options (*e.g.,* replaceable JAR libraries, calculations, and environment analysis). Figure 4-68 illustrates the relationship between the Policy Group and other component groups as well as the environment.

The Policy Group can be employed independent of all other WSLogA Framework components to provide a controlled process by which variable information transfer and formatting may occur. For example, many Web service applications make use of a logging framework, such as Log4J, which provides APIs permitting the development of custom data formatter or persistence components. A custom component could be developed by a third party that integrates the Policy Group to enable rules based processing of the log informa-

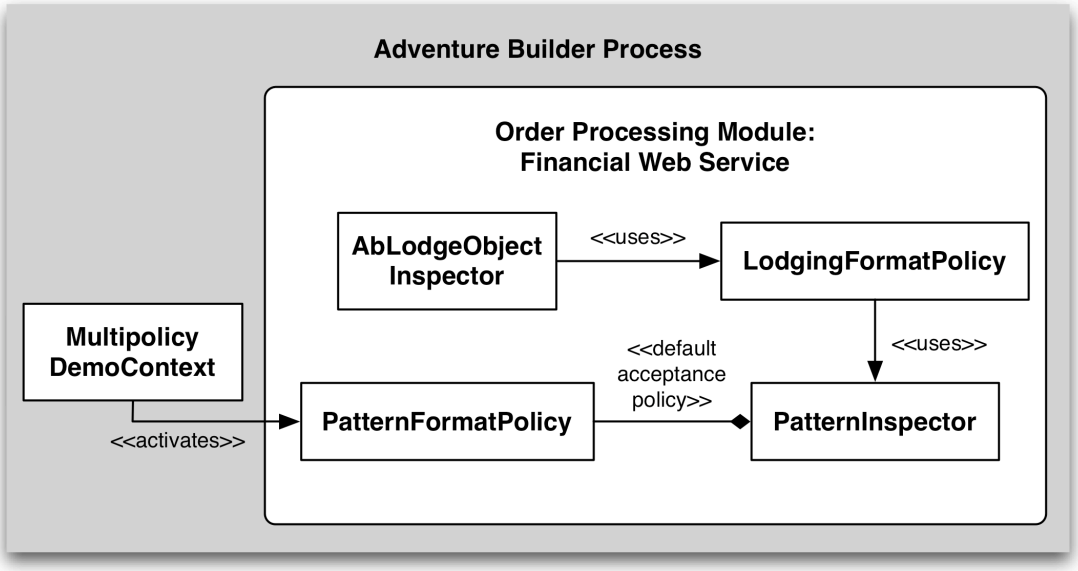


Figure 4-70. An example employment of the Policy Group.

tion, such as to ensure sensitive information (*e.g.*, a social security number) is masked before being placed into a public log. As such, the independent use of the Policy Group in an existing Enterprise environment can introduce development teams to the key information management concepts used by the WSLogA Framework, which could ease the subsequent adoption of advanced information flow component groups—such as the Monitor and Perspective Groups. Figure 4-69 illustrates the general relationships involved in the deployment of only the Policy Group.

The Policy Group is demonstrated as the information transfer and formatting mechanism for the Adventure Builder application (Appendix C). Scenarios such as the information capture and multiple policy examples employ the Policy Groups by means of Policy aware, WSLogA Framework derived components hosted within the GlassFish application server process. Figure 4-70 illustrates an example workflow involving the Policy Group.

Constraints and Opportunities

There are many opportunities for the use of the Policy Group in distributed systems, and especially for those systems that operate across legal or cultural boundaries. For example, the British Columbia government provides organizations based in the United States access to select health records for processing (Fayerman, 2008), and policies could be used to properly mask or otherwise transform information before it is provided to protect the interest of that Province's residents.

The Policy Group addresses the problem of policy expression and management using an object-oriented approach. Systems adopting this policy strategy can use the Policy components to represent and execute business rules for information management in a method natural for the Java platform. Policy updates can be performed by replacing outdated class files, and if a plug-in architecture is enabled, such as through a custom class loader, then policy enhancements can immediately take effect without the need to restart the system. However, highly distributed systems must be cautious with such approaches because policy expression should be consistent across machines within comparable policy regions, which means updates to Policy components must be properly scheduled and performed.

As Policy rules are expressed using the Java language, Policies established using the WSLogA Framework cannot be transferred to external systems or applications developed using competing languages, such as Microsoft's .NET, without the addition of a conversion framework. A future improvement to the Policy Group would be to externalize business rules using XML file sets or scripting languages, which could then be translated or executed by the WSLogA Framework to effect the desired behavior.

Summary

The WSLogA Framework fulfills the vision established by Cruz *et al.* (2003, 2004) for a flexible SOAP monitoring platform, and significantly improves upon their vision by providing end-to-end data management services with rules-driven processing. Information may be acquired from a variety of sources, such as SOAP messages, runtime objects, and environment data sources. The design transcends the Java platform and should be reproducible using comparable technology platforms, such as with Microsoft's .NET software development kits. The WSLogA Framework is optimized for Web service architectures, but the comprehensive information management approach ensures the WSLogA Framework's suitability for a multitude of Enterprise architectures. New applications can be designed as information appliances organized around the structure provided by the WSLogA Framework, and existing applications may gradually migrate to the WSLogA Framework's structure through selective implementation of the framework's component groups.

The Monitor Group provides systems based on the WSLogA Framework with a powerful mechanism for acquiring information from Enterprise systems, and, in particular, Web service based Enterprise systems such as those demonstrated by Sun Microsystems' Adventure Builder application operating within the GlassFish application server. Subsystems include monitoring, observation, inspection, and recording mechanisms to provide varying degrees of information acquisition and routing functionality. Systems building on the Monitor Group's functionality can quickly acquire and route session or transaction information without significantly affecting the established business logic. Functionality for capturing information from common Enterprise sources, such as SOAP and log messages, is predefined

by the WSLogA Framework, as is the capability to route captured information to a persistent data store with the assistance of the Event Group.

The Event Group facilitates the transfer of data from the WSLogA Framework component layer and a data store, such as a relational database management system or another Web service. A core component group, representing Event information, permits Enterprise systems to substitute technologies to satisfy engineer experience or platform limitations. For example, the Hibernate data persistence platform—which is based on configurable SQL statements—could be substituted for the JDO based data management strategy bundled with the WSLogA Framework whenever JDO cannot appropriately address complex queries. The appropriate use of common-denominator class types (*e.g.*, the preference of interfaces or abstract classes over concrete classes) when transferring Event information, such as by means of an EventPool instance, permits perspective components to make immediate use of new data store technologies.

The Perspective Group facilitates retrieval and organization of event information from the data store associated with a WSLogA Framework session. Information can be pushed (*e.g.*, the information is updated and then provided to observers) or pulled (*e.g.*, an observer instructs when new information is desired), which accommodates a variety of environment management and reporting scenarios. Policies can be introduced to components derived from the Perspective Group to enforce security or implement progressive disclosure. The workflow is designed to accommodate information made available by the Event Group, but alternate information sources could be integrated, if necessary. Information retrieval and distribution operations are best operated externally from Web service applications to permit the continuity of the WSLogA Framework's operations should an application fail.

The Response Group integrates environment management and reporting into the WSLogA Framework, which enables the Framework to handle data synchronization and task activation on behalf of third parties. The Response Group provides Enterprise systems with a mechanism for analyzing aggregated and correlated Event information, as well as for responding to the results with adjustments to the application or environment. Response operations can be externalized from the distributed system's application, which ensures that response operations can be executed even if application services fail. Alternate implementations made possible by the use of interfaces for response scheduling and execution enables systems to distribute response execution in a manner that best utilizes the system's resource constraints. The WSLogA Framework defines the workflow by which Event information is provided by a Perspective component to an observing ResponseTask component, which as implemented as an EventProcessor. Applications only need to implement the ResponseTask interface to make analysis and business logic available to the response system. ResponseTaskScheduler and ResponseTaskRunner components have been implemented to organize the simultaneous operation of ResponseTask objects ready for execution, but applications can define their own version of these interfaces to make the best use of system resources.

The Policy Group facilitates variable information management behavior without requiring architectural changes after deployment. Applications can define contexts in which Policies should operate and associate the contexts with specific Policy objects. The Policy Group is employed by the Monitor Group to guide information normalization before it is persisted by the Event Group for later analysis. Although the Policy Group members are integral participants in the WSLogA Framework, the components may also be adopted as an independent feature set by applications requiring a phased adoption of the Framework.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

This investigation established a novel design for an Enterprise system monitoring and environment management system that is portable across software development language platforms, and demonstrates that design through a Java based implementation. The artifacts were explored within the design research framework described by Hevner *et al.* (2004) and manifested by means of an iterative process for which design elements were envisioned, tests derived from the designs were prepared, and implementations were produced within the context of the tests. The result is a software development kit, the WSLogA Framework, suitable for adoption by practitioners as the basis for enabling holistic information capture within Enterprise environments and the management of environment parameters in response to analysis of the captured information. In effect, the WSLogA Framework enables Enterprise systems to be perceived as information appliances rather than traditional applications with distinct operational boundaries. Researchers may use the WSLogA Framework to explore the workflows by which information is produced and exchanged within Enterprise environments, and, in particular, those based on Web services.

The Achievement of Investigation Objectives

This investigation successfully produced a design and demonstration implementation that addresses the architectural vision of Cruz *et al.* (2003, 2004) for a SOAP monitoring system, the WSLogA, and improves upon the WSLogA by incorporating holistic information acquisition and environment response mechanisms. The WSLogA Framework establishes five significant component groups: Monitor, Event, Perspective, Response, and Policy. Each group provides predefined functionality and workflow integration that, together, enable comprehensive information management permitting an application's architecture to focus on business logic while maintaining support for operational analysis and correction. Further, WSLogA Framework's component groups are extensible, and alternative technologies may be substituted in lieu of provided components to accommodate a system's unique requirements so as to facilitate integration of the WSLogA Framework into existing environments.

The WSLogA Framework serves as a bridge between the concepts of information harvesting (*e.g.*, SOAP message capture or click stream production), operational dashboards (reporting utilities for a spectrum of services), and application development (the availability of a software development kit). The result is a platform encouraging software and system architects to envision applications as Enterprise elements supporting the overarching system as an information appliance that may exist across organizational boundaries rather than as distinct components organized around technology (*e.g.*, a "Java Web service application") or deployment (*e.g.*, "some system in California").

The Artifact Development Methodology

Test-driven development is a natural complement to iterative artifact exploration because the practice facilitates thoughtful consideration of the problem domain being modeled and how proposed solutions (*e.g.*, software components) should behave within the context of that environment. Design deficiencies and unforeseen workflows involving the components can be identified before the components are extensively implemented. Investigation efforts are therefore focused on the literature to identify potential solutions, the lessons learned from prior experiments, and the components' architecture. Researchers experimenting with architectures and component sets should adopt the principle of test-driven methodologies to gain the benefit of their efficiencies.

The Test and Demonstration Methodologies

The test-driven development strategy adopted to explore problem domain concerns, refine component functionality, and demonstrate important aspects of the WSLogA Framework was effective and likely reduced the effort necessary to produce a mature series of artifacts for the investigation. The planning and establishment of tests requires significant consideration of the components under test, which identifies problematic design elements. However, the strict form of test-driven development in which all tests are produced prior to the implementation of principal components did not work well for the problem domain's exploration. Instead, it was more effective to first identify component roles and relationships and define those using interfaces or lightweight abstract classes. Tests organized around the interfaces and their default relationships (*e.g.*, workflows made possible by WSLogA Framework component interactions) could then be established and concrete implementations

could then be defined as appropriate. The tests could also be developed to directly address the enhancements provided by a component definition within an inheritance tree, and existing behavior could be re-asserted by importing tests for the fundamental functionality.

Integration tests were useful for asserting the validity of complex component relationship implementations. Originally the Selenium test platform was used to capture workflows with the assumption that portions of the Adventure Builder would be executed for specific component sets, but the WSLogA Framework's evolution did not ultimately benefit from that approach. Instead, unit tests provided the more effective behavior validation for individual components and immediate relationships using techniques such as dependency injection. For example, mock Policy objects could be injected into a reporting component to influence its behavior across multiple Policy contexts as a contrived workflow progressed. Integration tests were better expressed as demonstrations providing examples of how the WSLogA Framework components could be used within system contexts, such as within the context of Adventure Builder operating across multiple application servers.

The Adventure Builder Context

Adventure Builder is a contrived J2EE 1.4 application used by Sun Microsystems to demonstrate the architectural principles addressed by their book, *Core J2EE Patterns* (Alur *et al.*, 2003) and various training programs supporting the Java certification tracks (Appendix B). The application is partially implemented using Web service technologies and enables a series of scenarios to be developed that illustrate the WSLogA Framework's success in achieving the investigation's objectives. Adventure Builder's implementation requires the generation of supporting Web service components by its host Application server, which means the

server selected for the demonstration must be compatible with the JAX-RPC and J2EE specifications. JBoss 4 was originally selected for use as the demonstration Application server because of its broad adoption throughout the software industry, but surprisingly it could not properly generate the supporting components despite its claim of J2EE 1.4 compatibility. Sun Microsystems' reference implementation for the J2EE standard, the GlassFish application server, was instead adopted. GlassFish correctly deployed and served the Adventure Builder application and proved to be easily managed by the build and test systems developed in support of this investigation.

Implications

Ensuring an application's operation requires a multitude of approaches that range in nature from robust design strategy to environment adjustments during runtime, and for decades researchers and practitioners have considered increasingly sophisticated mechanisms by which operational support may be provided. Enterprise environments complicate the issue of operational support beyond that for desktop application suites with the addition of concerns that include network based data exchange, clustered hosts, and extended periods of operation. Service-oriented architectures, such as those represented by Web services or computing grids, are quickly evolving to establish highly productive, multi-organizational contexts in which B2B e-commerce or research is performed. The complex and often hidden interactions between all components in these environments determine the operational health of the Enterprise environment.

Click stream monitoring of application workflows enabled administrators to understand the general manner by which users interacted with a hosted system, and the use of comple-

mentary tools facilitates the merging of logs or system performance data from a myriad of servers to provide more holistic perspectives of system utilization. Unfortunately, these tools generally remain external to the applications hosted within application servers and thus only provide indirect and inferred understandings of how monitored systems perform for some contexts while leaving many environmental factors unknown. Perhaps more important, such tools fail to provide environments with the means by which real-time corrections may be made to application behavior or environment status to ensure continual operation.

Cruz *et al.* (2003, 2004) recognized that SOAP messages contain business information and may additionally carry system information describing Web service component states (*e.g.*, operational or business rule faults) that can be harvested to enrich an administrator's understanding of the system's health. SOAP messages may traverse disparate technological platforms and organizational boundaries, and, as such, the operational information has the potential to expose quality of service issues that otherwise would remain hidden to an organization's support staff attempting to troubleshoot problems for which symptoms may not have so specifically identified misbehaving or defunct components.

The WSLogA Framework juxtaposes the architecture for SOAP message information harvesting envisioned by Cruz *et al.* (2003, 2004) with an environment response mechanism suitable for integration with application or service components hosted by Enterprise environments, including those spanning organizational boundaries. Contextual information harvesting is supported by collection mechanisms that integrate with traditional application information sources (*e.g.*, logging mechanisms), runtime object descriptions (through reflection and object state analysis), and extensible components for additional sources that may include files, sockets, and other external resources. Information structure and content

may be controlled through policies that can be context specific, which enables the implementation of a transactional architecture satisfying general business requirements but whose information management behavior may be customized after development or deployment for specific jurisdictions (*e.g.*, the European Union versus the United States).

Applications using the WSLogA Framework as a core element can make information management and self-healing operations an integral aspect of their operation. Systemic monitoring of information exchanges, transaction parameters, and operational behavior with an internal perspective of the application permits components to be designed and implemented with the convenient capability for state and behavior management. Error correction capabilities may also be implemented to accommodate issues transcending organizational boundaries, which permit the overall Web service system to activate candidate services based on a holistic and refined understanding of the quality of service offered by each. In effect, the Enterprise environment has the capacity to become its own intelligent agent capable of communicating its operational state with reliable precision and adjusting its overall behavior to ensure the continued and correct operation of business processes.

An interesting difference between traditional information routing or harvesting systems, such as the Log4J logging framework, and the WSLogA Framework is the pervasiveness and intent of information management provided by the WSLogA Framework. The WSLogA Framework is designed to support information harvesting for all aspects of the Enterprise environment with the intent that the information be used to influence operational outcomes for the implementing systems. Applications implementing Log4J or similar frameworks could largely consider information routing to be an implementation detail supporting development or debugging activities within the greater architecture dictating a Web applica-

tion nature, but the WSLogA Framework's influence on application design should be significant enough that an adopting system must necessarily exhibit a new nature—that of an information appliance. As such, the WSLogA Framework is an architectural equivalent, and possibly a complement, to existing frameworks such as Spring's Web module.

Finally, the availability of the WSLogA Framework increases the likelihood of information stealing and other abuses across cooperative Enterprise systems. The research community should examine related concerns so that future platforms appropriately guard against unintended information acquisition and misuse. The issues of identity, trust, and confidentiality must not be ignored as the WSLogA Framework evolves.

Recommendations

Enterprise environments do not operate as a collection of individual parts, but rather their behavior is the culmination of the myriad orchestrated interactions among all operational elements. Web services are further complicated in that their identity does not necessarily end at an organization's boundary and, instead, should be considered a sum of all supporting services involved in satisfying the system's workflows (albeit participants may change if services can be selected dynamically to satisfy, for example, quality of service calculations). In other words, all interacting services within a B2B relationship—and by extension, their host environments—can be considered the same application.

Practitioners must change their perspective of Enterprise environments and Web service systems from one of technology or application organized entities to that of information appliances. The business information, transaction information, operations information, and meta-information regarding these and other aspects of the conjoined parts are collectively

essential to providing the insight necessary to evolve the overall system in terms of quality of service and functionality; the WSLogA Framework is a manifestation and enabler of this recognition. The WSLogA Framework should be adopted in Java oriented environments, and the Monitor and Policy Groups should be evolved into a set of sub-frameworks addressing the information management possibilities for Enterprise systems. The WSLogA Framework should also be further developed, perhaps as a community project, for improved performance as well as security and information acquisition capabilities.

Contemporary frameworks defining application structure, such as Spring's Web module and the Apache Axis Web service framework, are conceptually compatible with the WSLogA Framework so framework researchers should explore how the WSLogA Framework may be integrated into these technologies. Candidate integration success may be measured by the degree information harvesting and environment response capabilities become natural extensions of systems based on the juxtaposed platform without the need for significant engineer familiarity regarding the component mechanics.

Information assurance researchers should investigate manners by which information may be leaked or generated for inappropriate use in result of such holistic and pervasive information management architectures. Policy definition and expression within distributed and parallel systems needs to further investigated. The ease by which distributed system components can be hosted across the world in legally or culturally disparate contexts increases the chance that inappropriate information management will occur, which could result in political, sociological, or economic problems. The concept of what constitutes a policy, the architectural hotspots for policy integration, transactional boundaries for when policy changes are realized during the course of a workflow, and other engineering concerns

must be addressed so that a universal model for policy integration into software systems may be established. It may be possible to integrate domain language support into the Policy Group, perhaps by means of the Java Virtual Machine's support for scripting languages, to permit business analysis, quality control, and information security staff to directly influence application behavior.

Monitoring and response activities are secondary objectives for most workflows. For example, a transaction involving financial deposits is first concerned with ensuring that the correct deposit is recorded and then concerned that monitoring systems are notified that the deposit occurred with specific characteristics. Aspect oriented programming (AOP) seeks to describe cross-cutting concerns, such as logging, as distinct from business logic and then correctly combine the two workflows during runtime with minimal instruction by the system's engineers. Some of this behavior can be simulated through deliberate system design (i.e., as workflow elements, such as those enabled by Template Methods) but monitoring and response operations must still be explicitly defined at appropriate points within a component set. AOP permits the definition of a logic trigger (*e.g.*, when a method exits on the stack) that is executed when implementation patterns for the principal components are executed. The implications of this approach to execution definition for the WSLogA Framework are significant and should be explored.

The exploration of the IT artifact is an iterative process in which evolutions in artifact behavior, state, and organization are deliberately investigated, consciously analyzed, and purposefully improved (Hevner *et al.*, 2004). This investigation's methodology demonstrated that adaptations of iterative, test-driven development focus the development of software artifacts. Role based design may be used in conjunction with the usual information gathering

process (literature reviews or lab results), and from the design tests should be established prior to other lab work. The test framework then guides the process of exploring implementations satisfying the design in a manner that is easily compared and documented. The tests also serve as documentation for the artifacts. Software development researchers should consider the use of test-driven development to structure investigation efforts and communicate results or intent to researchers outside of the project.

Summary

This investigation established a novel design for an Enterprise system monitoring and environment management system that is portable across software development language platforms (*e.g.*, the common Java and .NET software development kits), and demonstrates that design through the Java based WSLogA Framework. The WSLogA Framework is organized around the Web service monitoring architecture proposed by Cruz *et al.* (2003, 2004), but improves upon the architecture by incorporating holistic information capture, event analysis, and environment response capabilities. Five component families were established within the WSLogA Framework to meet these needs.

The Monitor Group enables information acquisition and routing. Components implement reporting roles that accept Subjects ranging in nature from SOAP messages to runtime objects. The Subjects are analyzed or used as the basis for calculations so that reports may be prepared and provided to components implementing recording roles. Recording components route the information to consumers, such as the relational database management system bundled with the WSLogA Framework as part of its demonstration system.

The Event Group defines the information model for the WSLogA Framework and serves as both the principal consumer and provider of event information for WSLogA Framework components. Data model implementations are provided for integration with persistent data stores (i.e., databases) as well as inter-component data transfers (i.e., as XML payloads). JDO was selected as the principal means of data transfer to data stores but alternate technologies, such as Hibernate, may be substituted.

The Perspective Group facilitates information extraction from the persistent data environment maintained by the Event Group, as well as the restructuring of extracted information to support the WSLogA Framework's integration with response or external systems. The Response Group is established on the platform provided by the Perspective Group and provides event analysis and environment response capabilities. The Response Group provides integrated workflow support with the Perspective Group but adopting systems must define their own error recovery and performance optimization implementations.

The Policy Group enables the definition of rules by which information may be filtered or otherwise transformed as it traverses workflows defined by the other WSLogA Framework component groups. Contextual behavior is defined to permit the execution of policies in specific scenarios to ensure the flexible behavior of WSLogA Framework based systems post-deployment and across operational scopes (*e.g.*, physical distribution or legal context).

This investigation was guided by a design research framework (AIS, 2005; Hevner *et al.*, 2004). The WSLogA Framework was established using an iterative approach based on the Spiral Lifecycle that facilitated the deliberate consideration of the problem domain, body of literature, lessons learned from prior experiments, implementation of components, and rigorous testing of components. Significant emphasis was placed on the use of automation

and test methodologies, such as test-driven development, to establish controlled environments in which components would function prior to component implementation. This approach ensures that component designs directly correspond to the requirements of the problem under consideration. The extensive test suite produced as part of this investigation, which includes unit and integration tests, also serves to document the problem domain addressed, the key principles behind the WSLogA Framework's design, and the manners by which third parties may adopt and extend the functionality provided by the WSLogA Framework. Consideration was given to configuration management throughout the investigation to ensure that key technology variables were tracked to ensure iteration results were comparable and to permit reproduction of the results by third parties. The efficacy of the completed WSLogA Framework was demonstrated on UNIX and Windows operating platforms, and an ISO file bearing the source code, build harness, tests, and demonstrations has been made available.

Researchers may use the WSLogA Framework to explore the complex component interactions and information workflows involved in Web service environments, and, in particular, for those that span physical machine, organizational, or legal boundaries. Practitioners may use the WSLogA Framework to establish Enterprise systems capable of communicating and reacting to their operational state or that of their environment. The WSLogA Framework facilitates the establishment of real-time, complex monitoring and management applications for Web services operating both within and external to an organization. The WSLogA Framework provides new opportunities for research into technologies addressing information policy and assurance.

AOP technologies and methodologies, such as those in the Spring Framework, provide an interesting context in which the WSLogA Framework may operate. Information acquisition can occur at transaction, component, or method boundaries, and event analysis coupled with environment response may be invoked upon exit points. The WSLogA Framework already provides low-touch system integration with its SOAP message monitoring and inspection capabilities, but in theory aspects can extend low-touch integration to most monitoring and inspection or response mechanisms within the Framework.

Search engine technology broadly relevant to Enterprise environments continues to evolve thanks to projects such as Apache's Lucene and Hadoop. Map/Reduce and related approaches to data organization may be applied to SOAP messages and associated system event records captured by the WSLogA Framework, which could permit the advancement of production monitoring and environment response systems. It may be possible to integrate Map/Reduce strategies with the Java virtual machine's support for scripting and domain languages to enable staff such as business analysts or technical support to create ad hoc rules for WSLogA Framework's analysis and response components.

Platforms such as the WSLogA Framework facilitate the acquisition and analysis of transaction information in manners that are holistic and time related. It may be possible for the WSLogA Framework to be used by middle tier organizations to collect information that can then be used in manners other than its intended purpose. For example, a government could collect information from commercial or health transaction systems that could later be analyzed to establish user profiles for security follow-up. Careful consideration should be given to the ethics of using the WSLogA Framework for such purposes, and mechanisms for

guarding information (such as with the use of WS-Security or the WSLogA Framework Policy Group) should be explored.

Appendix A

Quality Assurance

Measurement in IS Design Science

Information systems design research (AIS, 2005; Hevner *et al.*, 2004) recognizes design as a principal research artifact (guideline 1), but the artifact's quality, utility, and efficacy must be demonstrated before it can be considered valid (Hevner *et al.*, 2004). A Java based implementation of this investigation's resultant design was thoroughly tested (guideline 3) to, in part, demonstrate the problem relevance (guideline 2) and satisfaction of research contributions (guideline 4). The quality assurance framework used for the tests contributed to the research's rigor (guideline 5), facilitated the design search process (guideline 6), and facilitates the research's communication to technology oriented audiences (guideline 7).

The test-driven development (TDD) principle (Rainsberger & Stirling, 2005) was adopted to guide the formation of contexts in which implementations satisfying the design's intent and specification could be produced. Integration tests demonstrated the complex interaction of framework components with extension (third party) and system (*e.g.*, Application server and application) components. Unit tests demonstrated the implementation's design fidelity and exposed behavior or object state issues that needed to be resolved before subsequent implementation efforts could be pursued. Integration and unit tests exposed design deficiencies that were resolved with an iterative consideration of the WSLogA archi-

tecture, the literature base, and test results (Chapter 3). The tests exercise the design's intended behavior, which means the tests are extensions of the documentation base for the WSLogA Framework (Astels, 2003; Rainsberger & Stirling, 2005). Researchers may use the tests as benchmarks when exploring design modifications. Practitioners may use the tests to assert that the WSLogA Framework is mature enough for use in their systems, as well as to ensure that their extension components adhere to the intent and specification of the WSLogA Framework. This section describes the strategies and tools used to prepare measurements appropriate for facilitating this research.

The Test-driven Development Principle

The Spiral lifecycle (Schach, 2002) adapted for use in this investigation made quality assurance an integral aspect of the design's evolution and validation (Chapter 3). Designs were envisioned with input from Cruz *et al.*'s (2003, 2004) description of WSLogA, relevant literature, and insights gained from prior iterations. The designs were refined using an iterative process by which tests were prepared, implementations satisfying the tests were produced, and further need for design refinement or extension was identified (Cortes *et al.*, 2003; Hevner *et al.*, 2004; Rainsberger & Stirling, 2005). Additional tests were created as bugs in the source code were discovered to assert that subsequent development or refactoring corrected implementation behavior (Telles & Hsieh, 2001). Figure A-1 illustrates the test process adapted for use in this investigation.

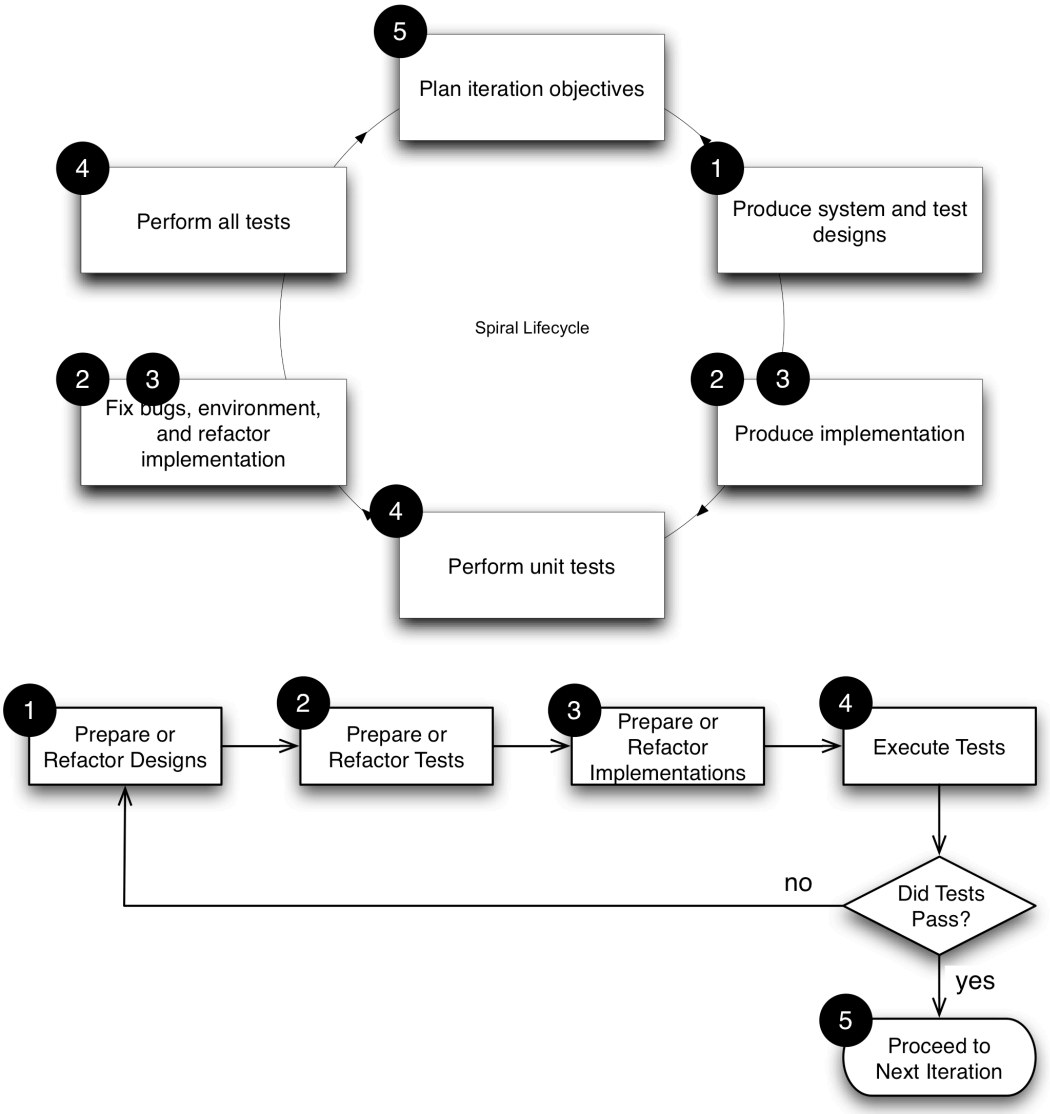


Figure A-1. Test-driven development as applied to this investigation.

This quality assurance strategy is inspired by the principle of test-driven development (Rainsberger & Stirling, 2005), which has been shown to result in higher quality implementations (Bhat & Nagappan, 2006; Maximilien & Williams, 2003) and appears to enhance learning outcomes (Bowyer & Hughes, 2006; Wick *et al.*, 2005). TDD based projects tend to progress slower than those based on a test after coding (TAC) strategy but yield higher

quality elements due to the necessarily extensive consideration of system contexts (Bhat & Nagappan, 2006; Canfora *et al.*, 2006; Maximilien & Williams, 2003).

TDD is a natural fit for framework development because such tests provide controlled contexts by which evolving component or method hot spots can be evaluated within an active test process, and the impact of framework refactoring relative to the design goals can be immediately perceived.

Assessment of the WSLogA Framework's API

Framework based APIs provide generalized solutions to problem domains common among application sets (Cortes *et al.*, 2003; D'Souza & Wills, 1998; Greenfield & Short, 2003; Schmidt *et al.*, 2004). For example, the Struts framework (Cavaness, 2004) provides a Model-View-Controller (Alur *et al.*, 2003; Gamma *et al.*, 1994) architecture for Web oriented applications. Struts' implementation manages extension components to coordinate the exchange and processing of information between presentation and business logic in a coordinated, predictable manner.

The correct behavior of the WSLogA Framework's APIs had to be verified within scenarios concerning individual method operation, component states after method invocation, and the control of extension components. Unit tests driven by the JUnit framework and test runtime engine were prepared and regularly executed to handle API tests (Appendix C). Successful test results indicated API adherence to design specifications, and the exploration of unsuccessful tests provided insight into the problem domain. Test results were a key input into the design process.

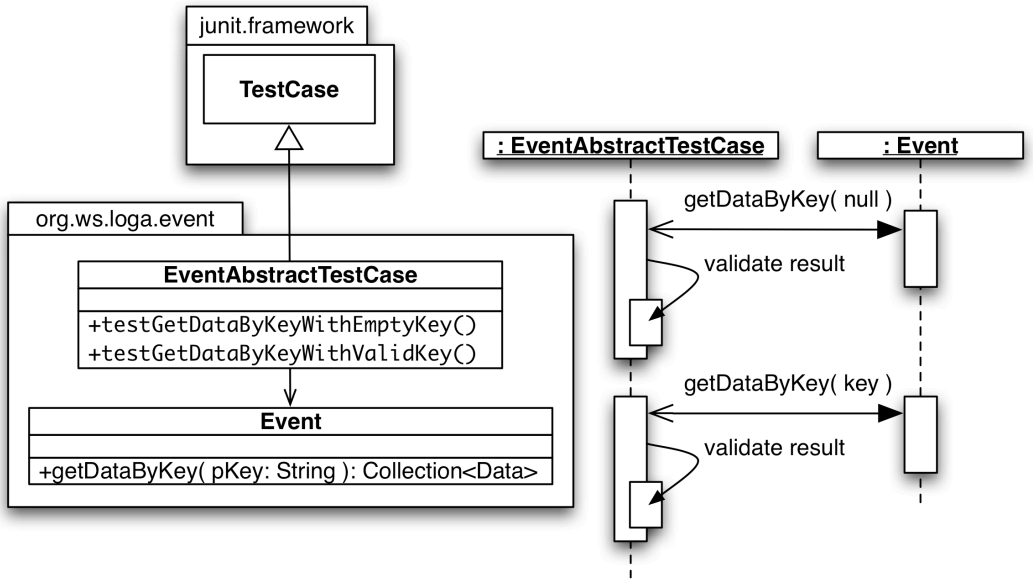


Figure A-2. Controlled exploration of method behavior through unit tests.

General Test Strategy

Initial unit tests focused on key business logic rather than attribute access methods (typically referred to as getters and setters) and similarly auxiliary or trivial operations. These initial tests permitted rapid implementation of a component to facilitate exploration of a proposed design. Unit tests providing more comprehensive coverage were implemented as part of the refactoring process for the design, implementation, and initial unit tests. The test cases adhered to the philosophy of Hunt and Thomas (2006):

- operation results must be correct;
- method boundary conditions must be appropriate and satisfied;
- inverse value and state relationships must be considered;
- operation error conditions must be forced; and,
- operation performance characteristics must be within bounds.

Boundary conditions, inverse relationships, and error conditions received specific attention during test preparation. Method inputs were considered for class types, list order and

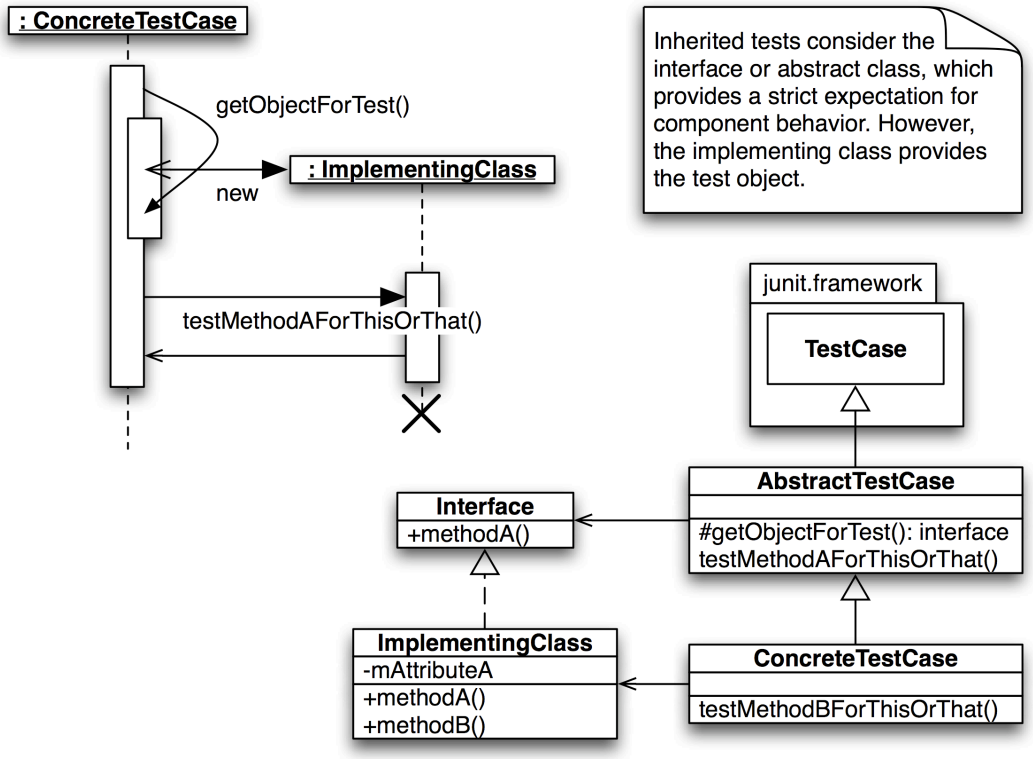


Figure A-3. Abstract test cases enforce the behavior of concrete components.

cardinality, degree of object state (*e.g.*, null, instantiated, or initialized), and minimum and maximum ranges. Malformed inputs were deliberately provided in some test scenarios to facilitate boundary, range, and error testing. Figure A-2 illustrates the controlled approach to method invocation.

Interface Component Test Strategy

WSLogA Framework design efforts focused on the production of a hybrid framework in which white and black box components (Richter, 1999) were incorporated. Role based design techniques (D'Souza & Wills, 1998; Richter, 1999) identified principal components—often those serving as framework engine templates or hotspots for framework extension by third

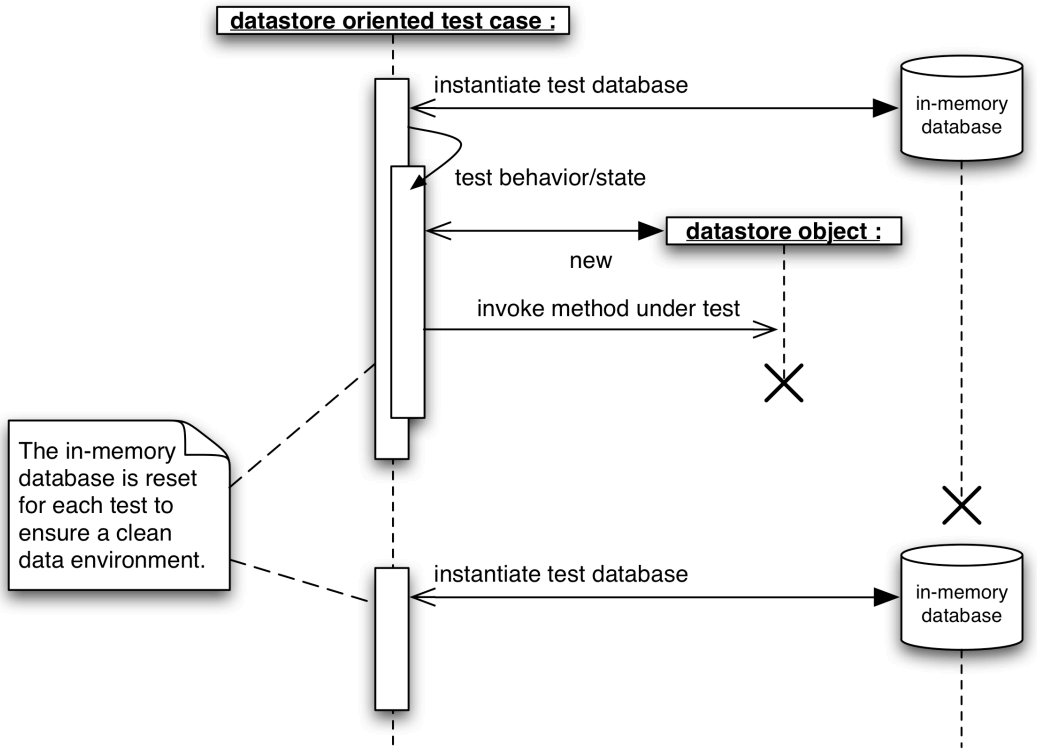


Figure A-4. In-memory databases are created and discarded for each unit test.

parties—that were implemented as interfaces. Interfaces do not provide functionality, but they do imply behavior expectations through method signatures and component documentation. Proper implementations of these interfaces were enforced with the use of abstract test cases, which document and enforce behavior expectations by providing test suites for an interface's methods.

A test case for a concrete component implementing an interface is expected to extend the abstract test case, which ensures that the interface's tests will be executed as part of the concrete component's test suite. The abstract test cases therefore guide WSLogA Framework extension and assure developers of such components that their results adhere to the

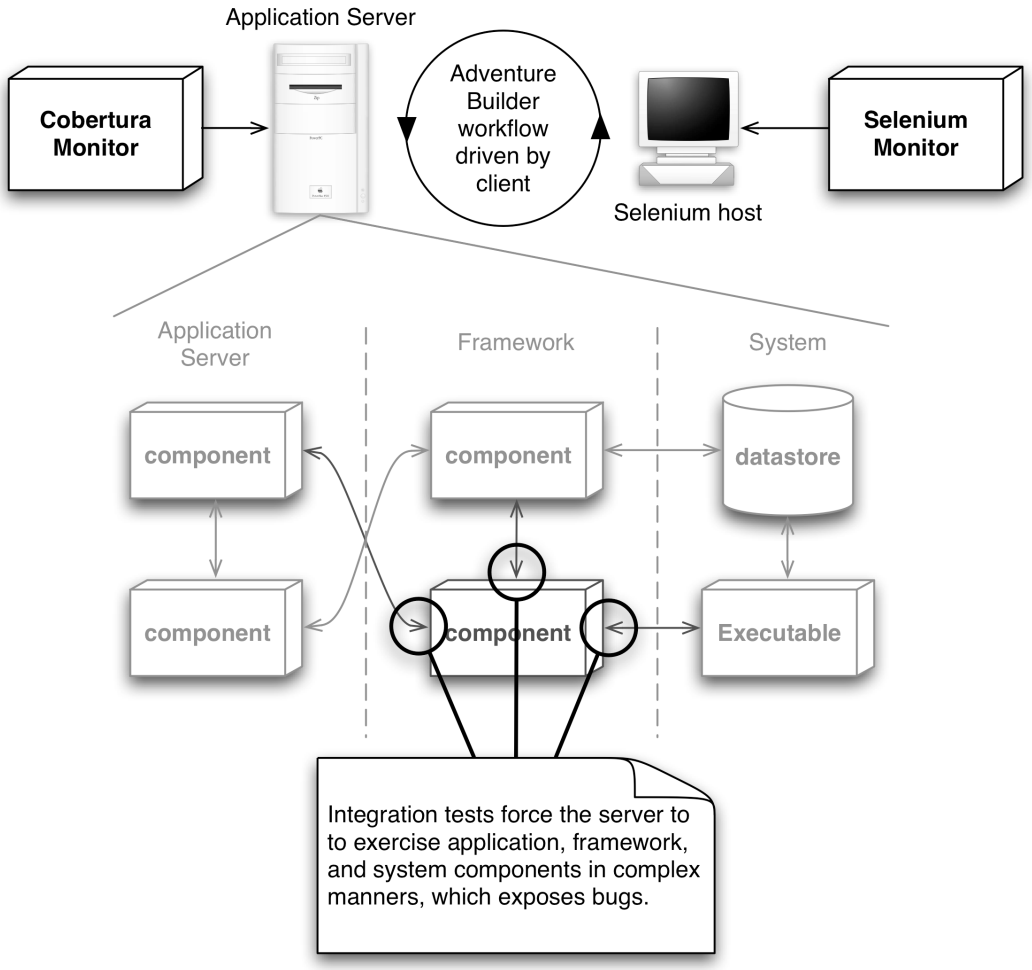


Figure A-5. Integration tests expose bugs hidden in complex relationships.

WSLogA Framework's intent. Figure A-3 illustrates how abstract test cases enforce design intent for concrete implementations of interfaces.

Data Management Test Strategy

The WSLogA Framework includes a data management layer that coordinates the exchange of data between data stores and the WSLogA Framework or application components. The provided implementation relies on JDO for transaction management, but third parties

may substitute their own data management and transaction strategies to support otherwise incompatible application architectures. The HSQLDB database engine (Simpson & Toussi, 2005) was adopted to facilitate testing of data management components as HSQLDB databases can be operated exclusively in-memory. Test cases initialize the data management components under test with an association to the HSQLDB database using scripts that provide just enough structure and content to facilitate the test. At the test's conclusion the database may be reset or discarded to ensure unit tests are always conducted with clean data stores. Figure A-4 illustrates the data management configuration process for unit testing. The DBUnit framework (Rainsberger & Stirling, 2005) was considered for data management, but concerns regarding JDO compatibility during this investigation's exploration of JDO frameworks eliminated DBUnit as a primary test management vehicle for data stores.

Assessment of the WSLogA Framework's Potential for Integration

The framework developed as part of this investigation is intended for integration into Web service oriented systems, which means that consideration must be given to the WSLogA Framework's ability for complex system integration. Sun Microsystems' Adventure Builder application (Appendix B) in combination with the GlassFish J2EE Application server and bundled Derby database were used to host the WSLogA Framework for integration testing. Integration tests were comprised of recorded workflows that could be used to explore successful and (deliberately) erroneous scenarios. The ThoughtWorks Selenium IDE (Holmes & Kellogg, 2006) integrated with the FireFox Web browser was used to capture the workflows, and the scripts were converted into JUnit tests for automatic execution as part of the project's automated build process (Appendix E). The execution of the recorded work-

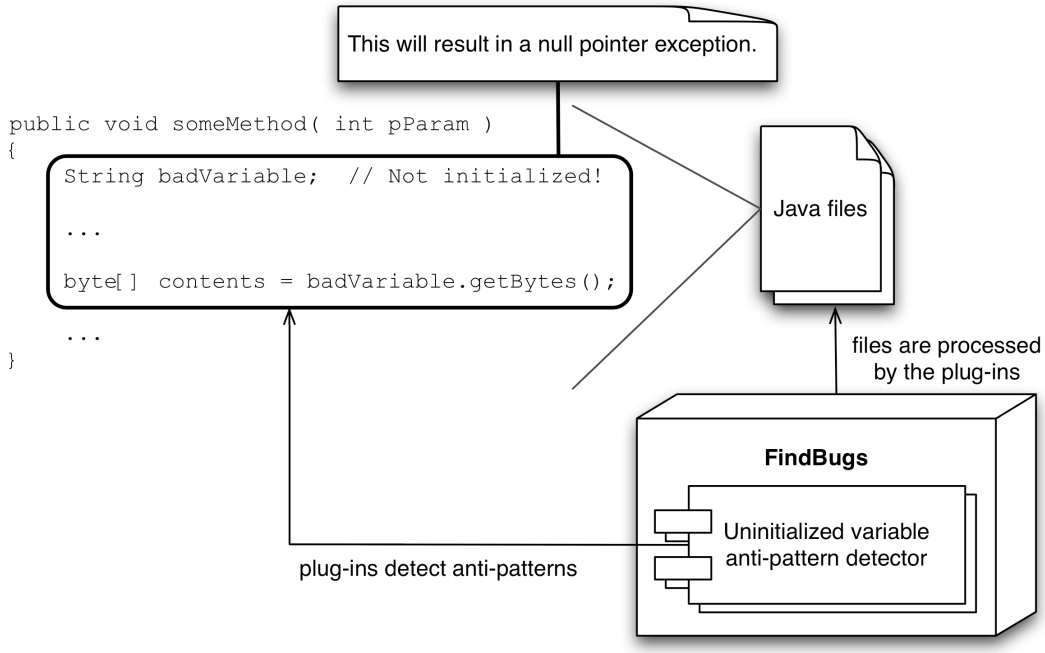


Figure A-6. Static analysis tools process source code to identify anti-patterns.

workflows exercised component instantiation, method invocation, data exchange, and thread management for the integrated Application server, Adventure Builder application, and framework systems. Figure A-5 illustrates how integration tests provide a comprehensive measure of framework and host environment interaction.

Static Source Code Analysis

Unit and integration tests detect faulty behavior and state but may not indicate why the behavior was faulty. For example, a null pointer exception may be caught by JUnit during a test—JUnit will report the exception instance and stack trace, but the source code must still be debugged or otherwise inspected before the cause of the null pointer exception can be known. Static analysis tools facilitate quality assurance by examining implementations for

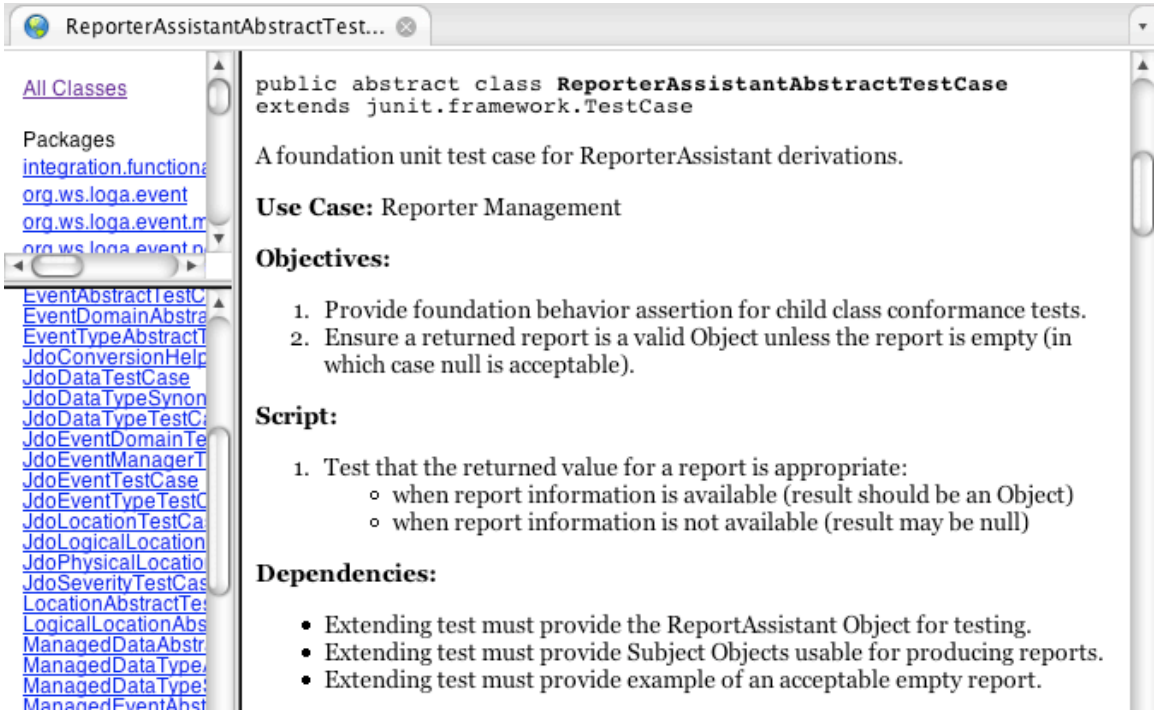


Figure A-7. Test case documentation.

patterns known to permit faulty behavior. For example, the failure to initialize a variable prior to its manipulation can result in a null pointer exception. A static analysis tool can detect and report such problems before the source code is compiled and executed (Ayewah *et al.*, 2007; Foster *et al.*, 2007). Figure A-6 illustrates the static analysis process.

This investigation made use of the FindBugs static analysis tool (Foster *et al.*, 2007; Hovermeyer & Pugh, 2007), which is associated with the University of Maryland. FindBugs integrates with Maven for automated bug reporting (CodeHAUS, 2007).

Test Documentation

Documentation was prepared to communicate unit and integration test intent and requirements. Each test case was prepared with a detailed HTML fragment so that JavaDoc

documentation prepared from the test case's comments would clearly communicate important aspects of the tests performed. Figure A-7 illustrates test documentation within a unit test case's JavaDoc page.

Surefire reports (Appendix G) generated as part of the automated build process (Appendix F) were archived during the investigation to facilitate exploration of framework implementations across iterations. Folders containing test documentation and results were numbered to ensure temporal clarity for test results.

Test Result Analysis

Successful test results indicated the likelihood that components or workflows under test adhered to the design intent and specifications. Components whose tests completed successfully were considered stable for use by third parties and for subsequent use in related experiments or project work. Unsuccessful test results indicated that a component's implementation either did not satisfy the design's intent or specification, or that the design did not adequately address the problem domain. Problematic components were first analyzed for implementation issues (*e.g.*, bugs or malformed relationships caused by the refactoring of other components) and then for design issues. Straightforward implementation issues were simply corrected and verified with additional tests. Complex implementation issues, such as malformed relationships or a failure to address a discovered scenario, resulted in the design's rescheduled work for a subsequent iteration.

Summary

A test-driven development approach was used to guide the evolution of designs and implementations. Initial tests for interesting use cases were produced in conjunction with a

design's specification. Components were implemented and explored with the initial test suites. Additional tests were developed as a design or implementation matured to ensure that bugs and new components behaviors or relationships were properly accounted for despite subsequent refactoring within the same or related packages. Unit tests ensured the validity of WSLogA Framework API implementations and integration tests ensured the WSLogA Framework's potential for integration within a complex, Web services oriented system. Tests were documented to communicate their intent and outcomes with the use of HTML headers in unit test Java files as well as Surefire success reports. Researchers can use the WSLogA Framework's test suite as a guide for further experimentation or extension. Practitioners can use the test suite as documentation and a measure of the WSLogA Framework's ability to perform in the expected manner.

Appendix B

Adventure Builder as the Test Environment

A Comprehensive J2EE Reference System

This investigation used Sun Microsystems' Adventure Builder system as the environment for testing the WSLogA Framework's efficacy. The Adventure Builder application permits users to browse and purchase a series of vacation packages supplied by vendors and service providers. The system simulates a reasonably complex J2EE system involving Web services and external transaction dependencies (such as communication with hypothetical financial entities). Several of Sun's books, websites, and certification courses use Adventure Builder to demonstrate J2EE best practices, so the application's popularity should enable software engineers quickly to comprehend the proposed WSLogA Framework's design and component distribution within a functional environment.

The Adventure Builder Architecture

A complete description of Adventure Builder and the involved design principles can be obtained by visiting Sun's Adventure Builder project online (Sun Microsystems, 2005) or by examining the associated resources (Singh *et al.*, 2004; Sun Microsystems, 2006). Figure B-1 illustrates the components and processes involved for the presentation and business tiers.

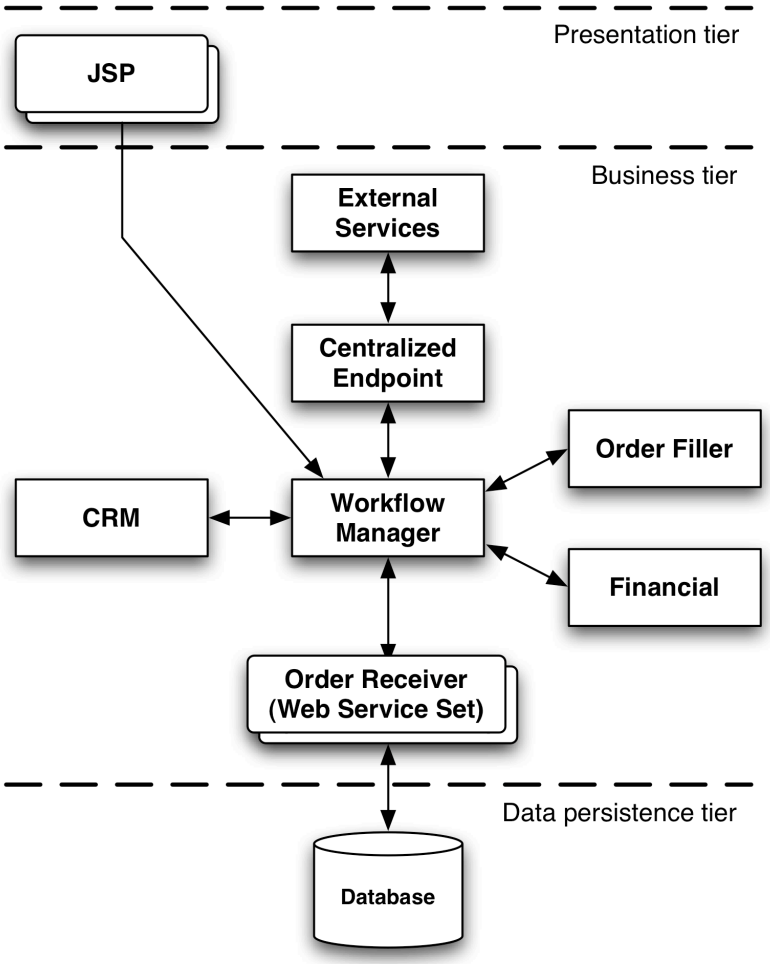


Figure B-1. The Adventure Builder architecture.

Adventure Builder is based on the model-view-controller architecture pattern (Cavaness, 2004; Gamma *et al.*, 1994), in which a controlling series of components coordinates them activities of presentation and business model. The business model is responsible for mapping the data tier into the application's components. Clients interact with Adventure Builder through a series of Java Server Pages (JSPs) presented within a web browser. A database persists information such as catalog items and user vacation package selections. The controller uses a master servlet to specify the overall workflow, but sub-sections rely on Web service

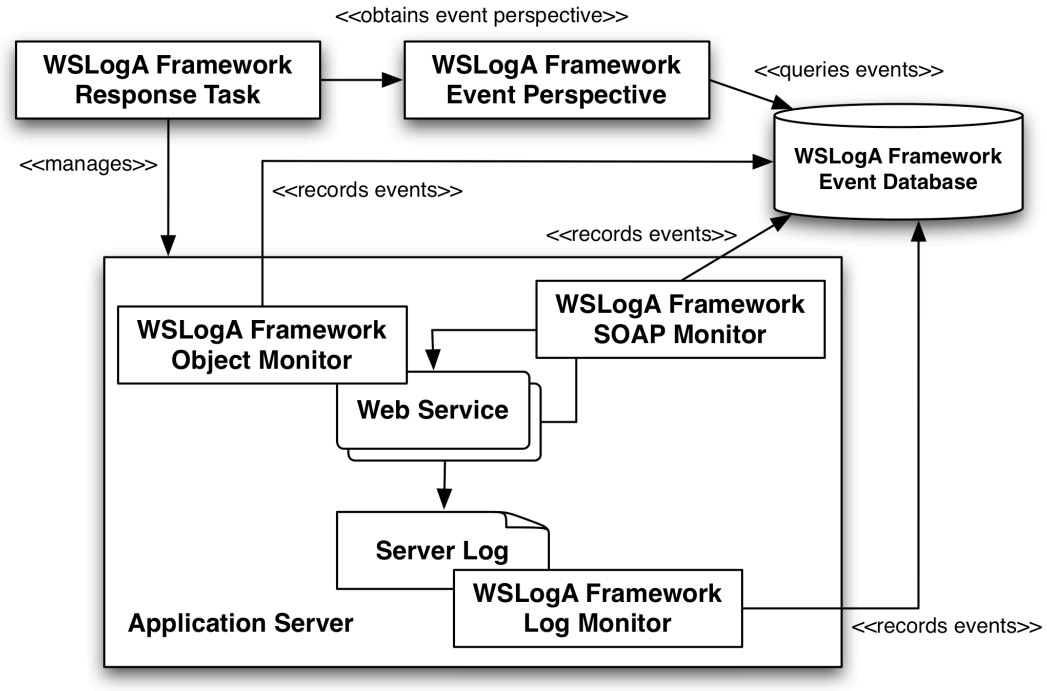


Figure B-2. WSLogA Framework components interact with Adventure Builder.

components that interact with a persistent data layer or external (to the organization) entities for detailed business rule implementation. Adventure Builder is similar to other J2EE systems in that it depends on a host of application and Web Server, data persistence, networking, log, and operating services that provide the types of maintenance challenges WSLogA derivatives are intended to learn about and manage.

Integration with WSLogA

WSLogA uses intermediary Web service components to analyze data in transit and report events or content of interest for later analysis, as illustrated by Figure 1-2. The proposed framework adds observing components for relevant system aspects such as the Application server, database, and operating system services. The information collected by these compo-

nents is placed into a persistent storage solution, such as a database, which primes the event processing engine's queue. The event-processing engine relies on pluggable components to present information or interact with the environment for corrective maintenance. Figure B-2 illustrates in simplified form the manner by which framework data capture and information processing systems interact with Adventure Builder, the Application server, and other environment components or log repositories.

Adventure Builder as Related to Cruz *et al.* Research

Cruz *et al.* (2003, 2004) demonstrated WSLogA principles by examining systems for workflow and transaction information. Adventure Builder also provides workflow and transaction scenarios that the WSLogA Framework can process in manners similar to the examples provided by Cruz *et al.* As such, the use of Adventure Builder for the demonstration system

facilitates proper evaluation of the produced WSLogA Framework in lieu of the systems used by Cruz *et al.* Additionally, the use of Adventure Builder as the demonstration system host application permits researchers to explore with consistency the WSLogA Framework's design against a known benchmark. Practitioners can use the demonstration system to test their WSLogA Framework extensions.

Appendix C

WSLogA Framework Demonstrations

Intent

The WSLogA Framework provides holistic information collection, analysis, and event response capabilities to Enterprise systems in a manner that reduces the knowledge and work necessary for the implementation of such functionality. The WSLogA Framework is demonstrated through four scenarios involving the framework, the Adventure Builder J2EE application, and the J2EE 1.4 compliant GlassFish application server. The demonstrations are configured for execution on Mac OS X and a subset of the live demonstrations may be executed on WindowsXP Professional. The combinations of these scenarios and platforms enable WSLogA Framework's strengths and weaknesses to be assessed in an objective manner within the context of environments representative of service oriented Enterprise systems. The demonstration suites provide the context by which the WSLogA Framework components are described in Chapter 4, and Chapter 5 addresses the implications of demonstration outcomes.

Organization

Demonstrations of the WSLogA Framework are organized into scenarios that have mock and live environment counterparts featuring comparable workflows. Both suites are controlled using the JUnit test harness, which permits the extension of tests through the

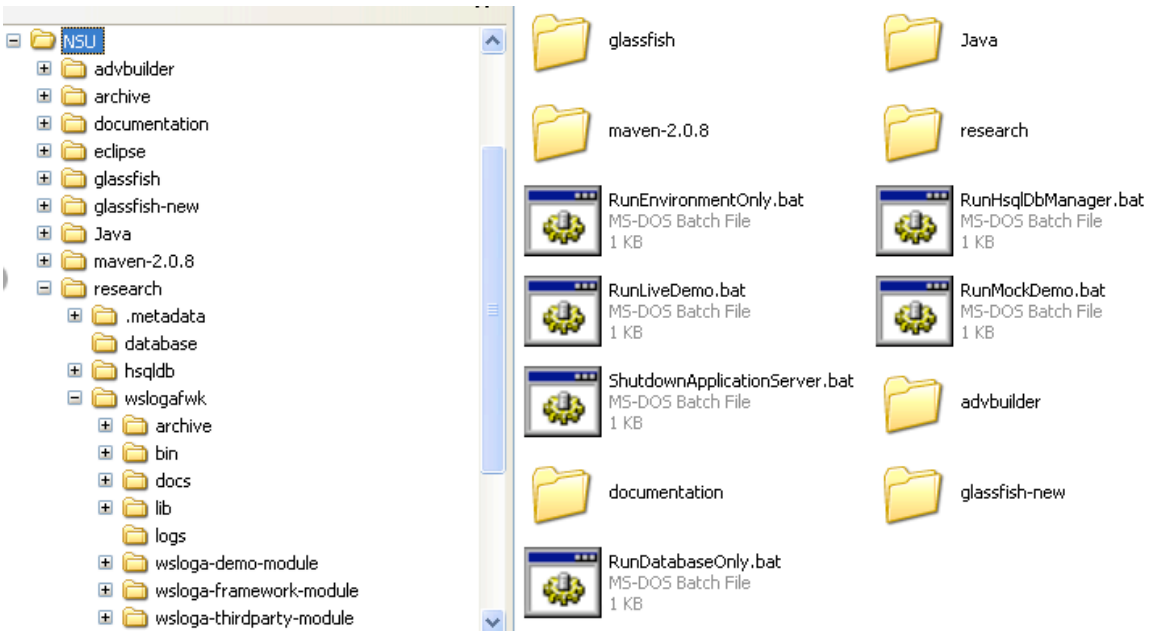


Figure C-1. Scripts are available to run the demonstrations using common options.

addition of new Java routines and facilitates analysis of the WSLogA Framework components through the use of debugging tools (Appendix A).

The mock demonstration suite simulates environment components, such as the application server, by organizing the process or data flows involving WSLogA Framework components or derivations and injecting data values or assessing results in a manner that requires minimal resources and provides maximum operational precision (Freeman *et al.*, 2004; Staff & Ernst, 2004b). As a result, studies regarding the behavior of WSLogA Framework components or derivations may focus on component mechanics without the distraction of environment availability, configuration, and operational timing.

The live demonstration suite uses external systems, such as the GlassFish application server, to exercise the WSLogA Framework components or derivations within contexts

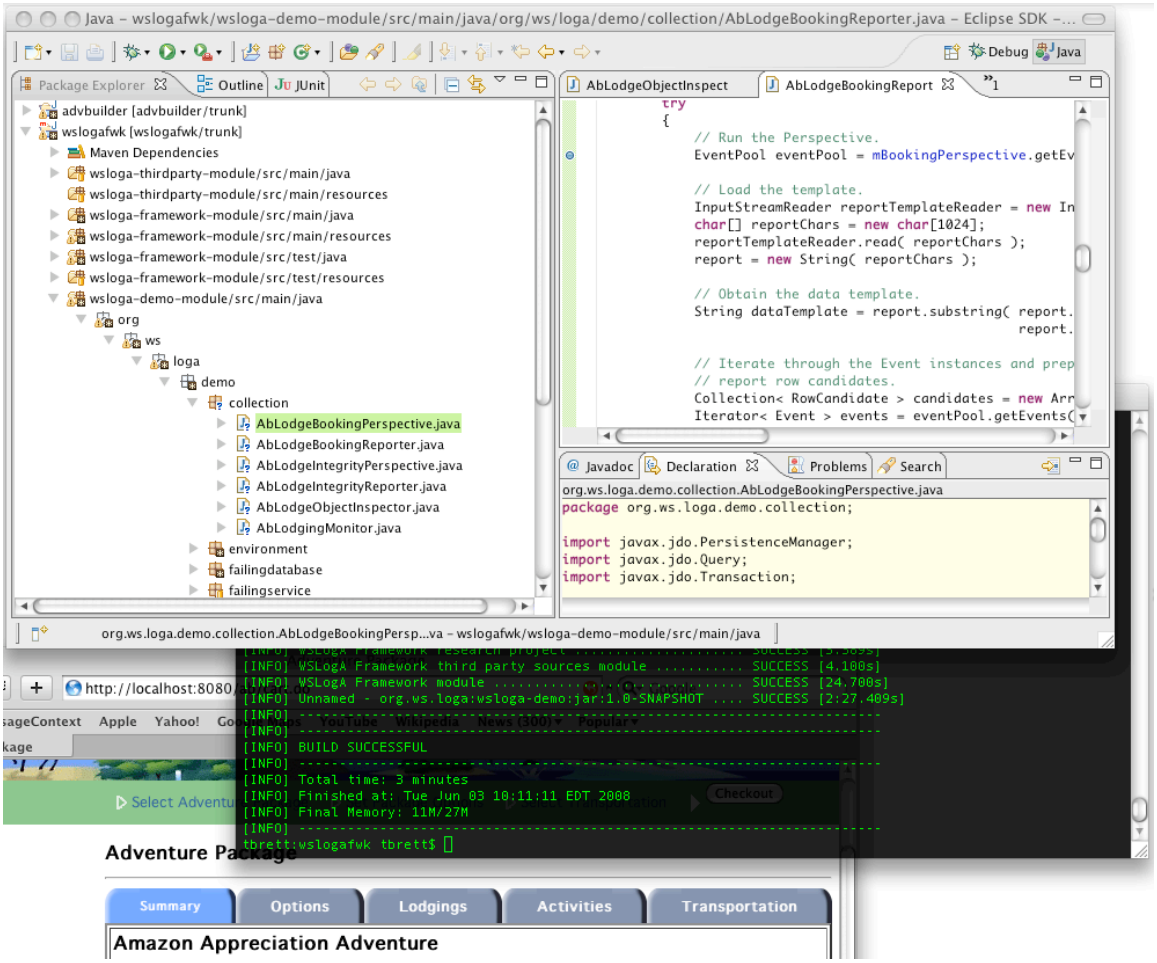


Figure C-2. Project tools are provided to facilitate WSLogA Framework analysis.

representative of Enterprise systems (Holmes & Kellogg, 2006). The Selenium extension for JUnit is used to control Web browsers hosting interactive Adventure Builder sessions. Live demonstrations are operationally less precise than their mock counterparts and may be affected by factors that include, but are not limited to, the operating system, degree of processing power, and quantity of RAM made available by the host machine.

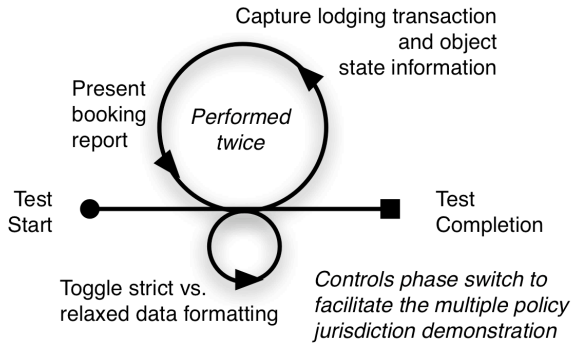


Figure C-3. Demonstration phases for information collection.

Distribution

The source code and supporting tools for the WSLogA Framework were archived onto DVD and made available with this report to facilitate third party assessment of the investigation's artifacts. Appendix D discusses the configuration used to prepare the WSLogA Framework for development and quality assurance in conjunction with the tools utilized throughout the Framework's development. UNIX is the preferred environment for operating the mock and live environment demonstrations, and specifically the Mac OS X operating system was used for principal development and quality assurance.

VMware virtualization technology was adopted to host the Microsoft Windows environment and operate the demonstrations within that context. A virtual PC provided in the form of a VMware virtual hard disk file was prepared using the WindowsXP Professional SP2 edition provided through the MSDN to NSU graduate students. The Eclipse IDE, Maven2 build engine, the J2SE 1.5 JDK, and GlassFish v1 were installed on the VMware system and preconfigured for use with the WSLogA Framework project. The execution of all mock and

```
      Observer lodgingObserver = new Observer( inspector );
      lodgingMonitor.getReporters().addReporter( lodgingObserver );
      try
      {
         // Do Interaction layer processing
         ...
      }
      finally
      {
         lodgingMonitor.monitor();
      }
   }
```

Figure C-4. Adventure Builder modifications to include inspectors are minimal.

live environment demonstration suites within Mac OS X was captured as a QuickTime video and may be considered by audiences unable to operate the WindowsXP based distribution.

Operation

The VMware virtual machine produced for the demonstrations must be used with one of VMware's virtualization hosts, such as VMware Workstation (Microsoft Windows and Linux), VMware Fusion (Mac OS X), or the VMware Player (Microsoft Windows and Linux). The free VMware Player for Microsoft Windows is provided as part of this investigation's project (Appendix D) and is recommended for configuring and hosting the VMware virtual machine.

A set of script files is made available for both the UNIX (bash shell) and Microsoft Windows platforms. These script files are suffixed with the appropriate file extension and are placed within the *src/main/script* folder for the demonstration module (Appendix D). The VMware virtual disk also makes these scripts available within the *C:\NSU* folder. Figure C-1 illustrates the location of the script files. The UNIX scripts are configured to operate in the correct folders as provided on the DVD or VMware distributions.

The complete demonstration suite (except the aforementioned live demonstrations disabled when using the VMware demo environment) may be started using the *RunLiveDemo* script. The *RunMockDemo* script may be executed to exercise only the mock demonstration suite. All demonstration suites make use of an HSQLDB server to store event information captured by the WSLogA Framework, but only the live environment demonstrations take advantage of a persistent, disk based database server facilitating post-operation evaluation of the generated event information (Appendix A). The *RunDatabaseManager* script launches a graphical database manager that may be used to inspect the event information and generate reports with the use of SQL commands. The *RunEnvironmentOnly* script starts the HSQLDB server and the application server that hosts distributed WSLogA Framework components and the Adventure Builder application. Individuals desiring to trace the framework's operation during manual execution of the Adventure Builder application should run only the environment and the Eclipse IDE to control the source code that is explored (Figure C-2).

The Eclipse IDE and Maven2 development tools are preconfigured in the VMware distribution for use with the WSLogA Framework project. A debug profile for the externally operated GlassFish Application server is made available through the Eclipse IDE's debug menu, and the command line may be used to execute Maven2 build instructions for building and packaging the WSLogA Framework artifacts (Figure C-2). The VMware virtual hard disk provides practitioners with a self-contained development and runtime environment for WSLogA Framework development, and provides researchers with a self-contained environment for artifact evaluation.

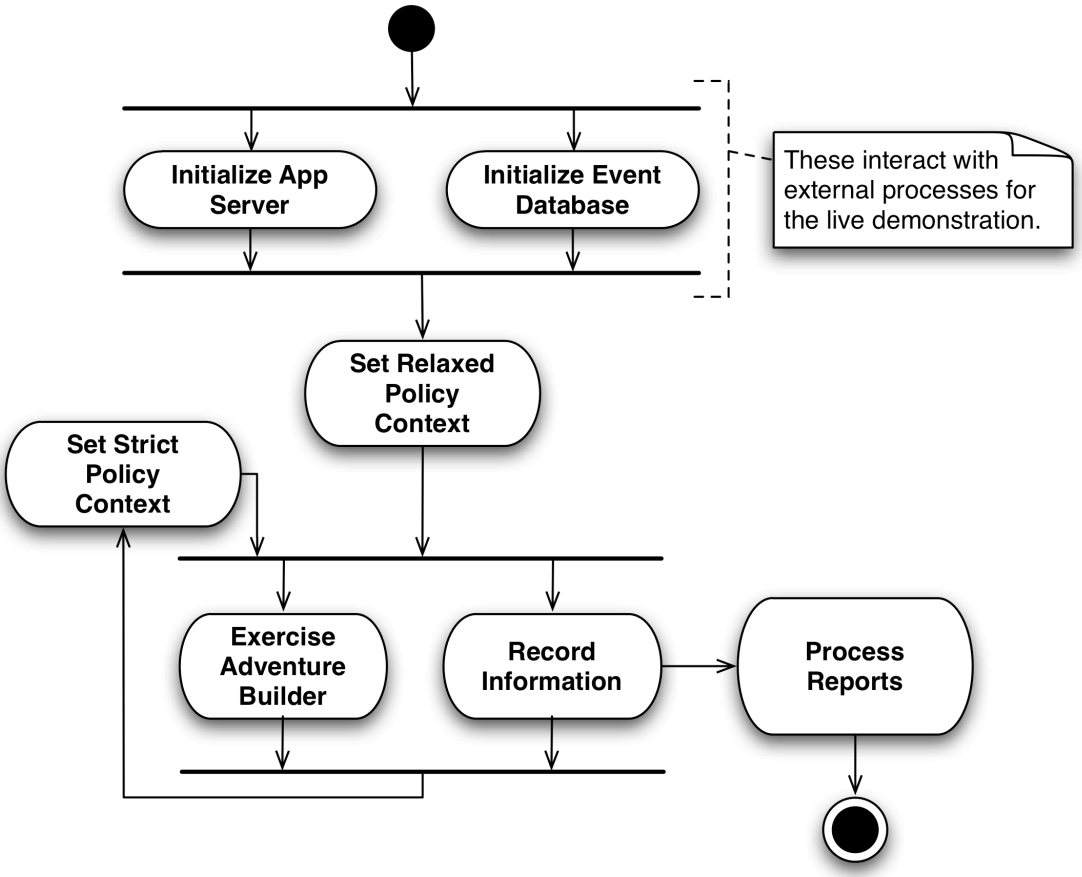


Figure C-5. Demonstration flow for information collection.

Information Acquisition and Dissemination

Cruz *et al.* (2003, 2004) introduced the WSLogA as a means by which information could be gathered from Web service based systems to facilitate business decision making and technical support, which suggests that any derived framework should focus on information collection. The WSLogA Framework facilitates information acquisition across multiple source types through SOAP and log message monitoring as well as the runtime observation or active inspection of Java Objects and system components. The combined use of these mechanisms in an enterprise system permits precise and focused information collection supporting

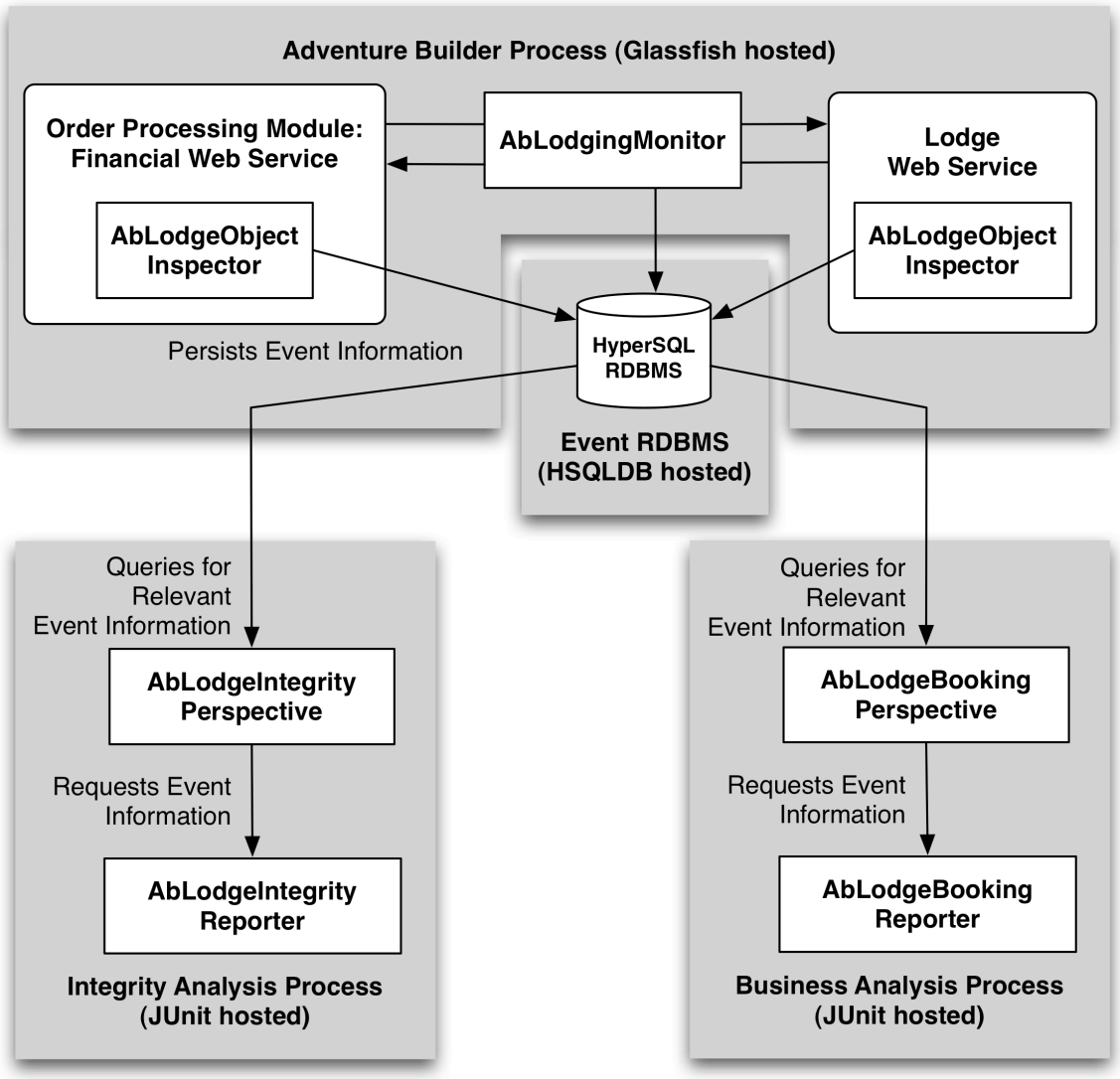


Figure C-6. Demonstration interaction for information collection.

engineering, support, and business concerns (Gulcu, 2002, 2005; Gupta, 2003; Telles & Hsieh, 2001).

This demonstration uses the communication between the order processing controller (OPC) and lodging Web services included as part of the Adventure Builder application to demonstrate capabilities of the SOAP message and Java Object inspection information

acquisition mechanisms (log interception is demonstrated as part of the failing database demonstration discussed within this appendix). The SOAP based transaction between the OPC and lodging services is inspected to identify the lodges preferred by customers and whether data transferred from the lodge purchase order, SOAP transport, and order processor maintain information integrity.

Figure C-3 illustrates the phases provided for the consideration of WSLogA Framework component behavior. The demonstration does not require an event history so the application workflow necessary to invoke the SOAP transaction is performed only once. A SOAP Handler captures the information necessary to prepare a report regarding selected lodges in a manner that does not require modification to Adventure Builder's source code. The OPC and LodgingSupplier modules are each modified to contain an Object inspector used to monitor lodging information. However, the source code modifications are superficial (Figure C-4) and the business logic driving the transaction remains unchanged.

Report components are provided for use in the demonstration's JUnit logic for the purpose of preparing an HTML based lodging report and to ensure information integrity is maintained throughout a session's data flow. Perspective components providing focused access to the WSLogA Framework's event pool are used by the report components to obtain the relevant event information. Figure C-5 illustrates the flow among the WSLogA Framework derived components and the operationally relevant Adventure Builder services.

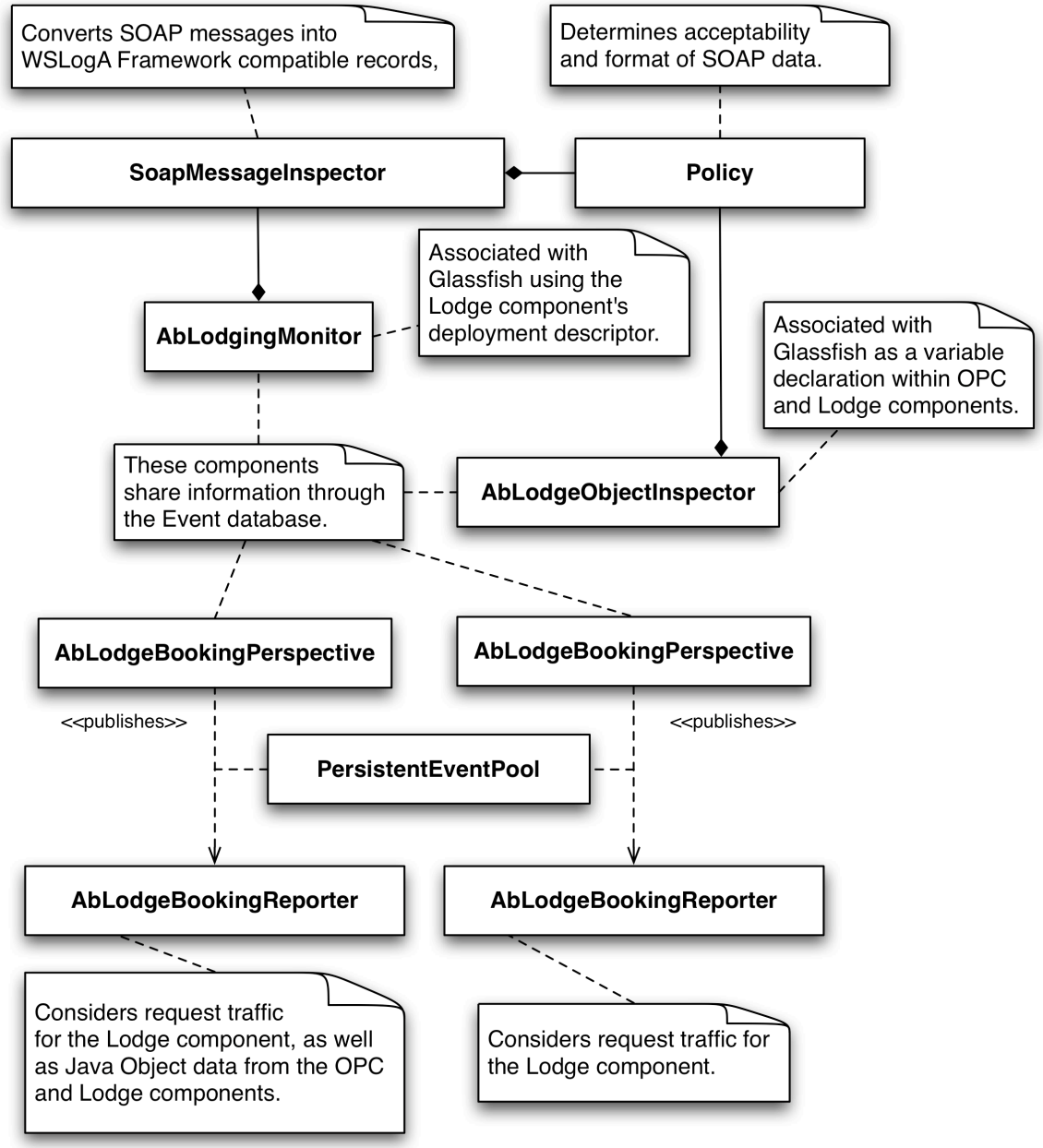


Figure C-7. Information collection demonstration components.

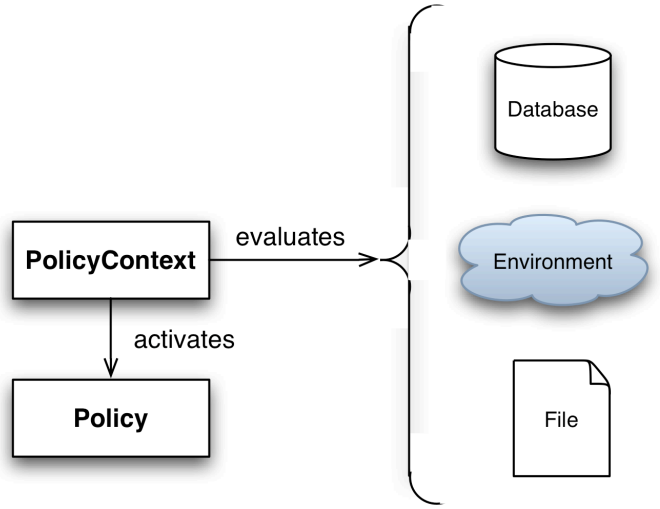


Figure C-8. Policy contexts evaluate scenarios to control policy behavior

The live environment workflow deviates from the mock workflow in that the GlassFish application server and HSQLDB process are operated in independent processes. The use of distinct system contexts asserts the runtime utility of the WSLogA Framework components for the purpose of Web service monitoring and profiling, and provides an example of component distribution across processes that are typically distinct in Enterprise systems. Figure C-6 illustrates the interaction among the WSLogA Framework derived components and the operationally relevant Adventure Builder services.

Multiple components were prepared using the WSLogA Framework to acquire and, if appropriate, modify information either processed by the Adventure Builder application or that was obtained by inspecting Java Objects operating within an Adventure Builder session. Figure C-7 illustrates the WSLogA Framework extensions for the information capture demonstration, which are provided in the demonstration module (Appendix D).

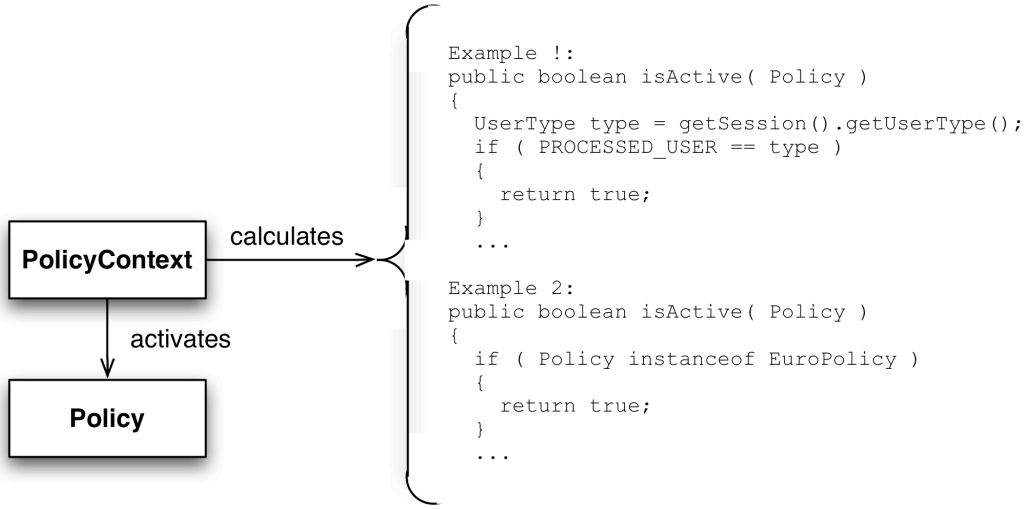


Figure C-9. Static contexts communicate the general applicability of policies.

The AbLodgingMonitor component extends the SoapHandlerMonitor provided by the WSLogA Framework, which in turn uses a policied Observer component for message analysis and a Recorder component for persisting the event information into the WSLogA Framework's database. A process external to the application server (in this scenario, the test JVM) is established to host event information perspective and response components that handle Lodging Web service management.

AbLodgeObjectInspector extends ObjectInspector by initializing a set of delegate Object field inspectors that, in turn, have an active state regulated by a PolicyContext sensitive to a demonstration-controlled flag contained within the session's WSLogA Framework database. Each delegate component, such as the NonProcessingFieldInspectorDelegate, uses a regular expression or Java based calculation to ascertain whether an inspected field contains relevant content and, if so, produces a PatternInspector capable of filtering or otherwise manipulating the field's data to suite information acquisition requirements.

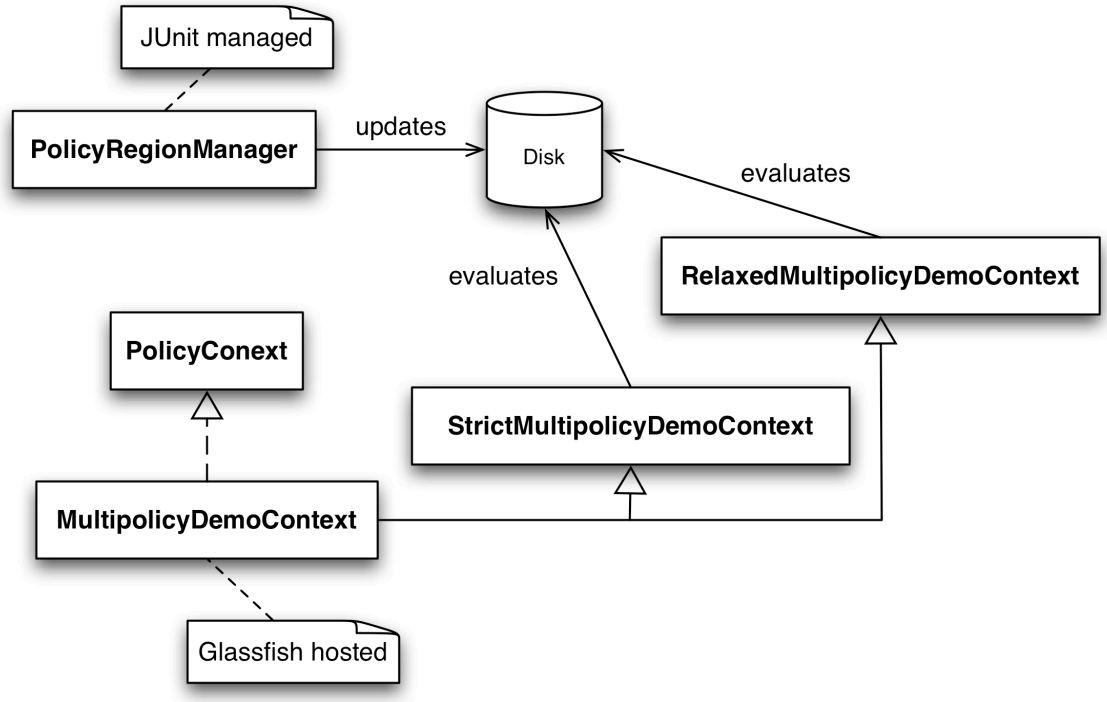


Figure C-10. Dynamic contexts ensure the selective policy activation.

AbLodgeIntegrityReporter is a Reporter component that pulls event information from AbLodgeIntegrityPerspective per reporting event to ascertain whether data was transferred from the originating Lodging component across the SOAP mechanisms and into the receipient Lodging component within Adventure Builder. The calculation is performed by matching information captured by AbLodgeObjectInspector and AbLodgingMonitor objects during an Adventure Builder session.

The Adventure Builder application was modified to support the generation and insertion of a GUID value into the Lodging data stream at the transaction's outset, and this ID is used by the AbLodgeIntegrityReporter to relate otherwise generic event records. The AbLodgeIntegrityPerspective uses JDO based components from the Event Group to retrieve the event

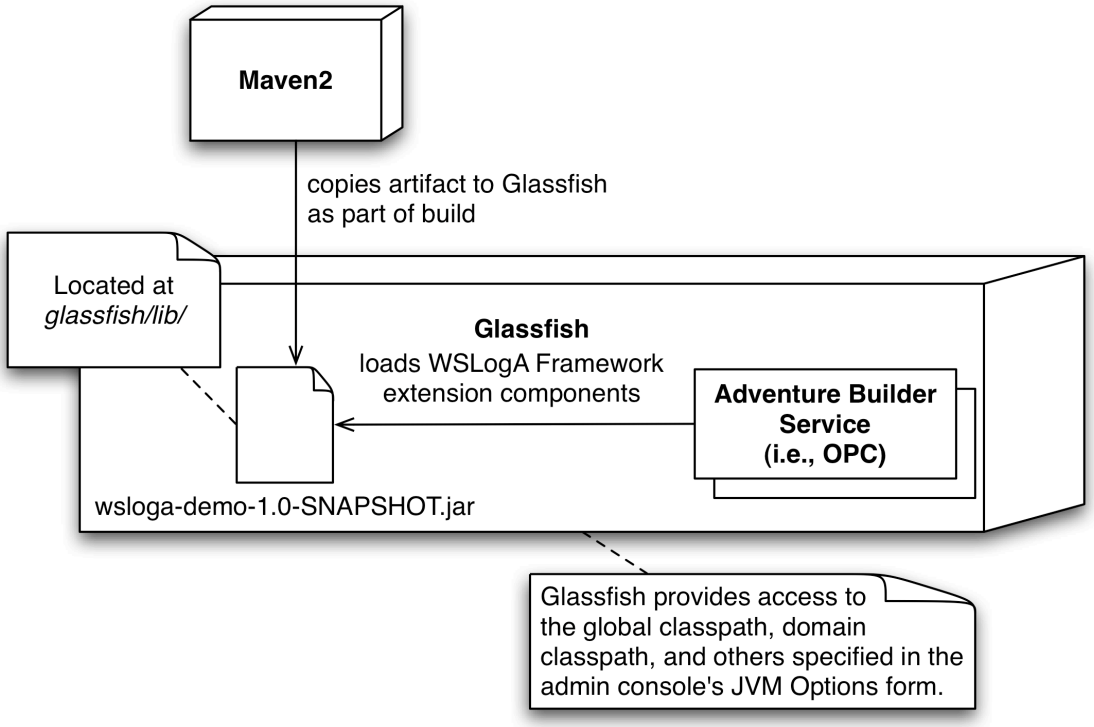


Figure C-11. Demonstration policies and contexts are embedded within project artifacts.

information and prepare a non-transactional EventPool object for consumption by AbLodge-IntegrityReporter.

AbLodgeBookingReporter is a Reporter component that pulls event information from AbLodgeBookingPerspective per reporting event to list the lodging booking requests made within the Adventure Builder application. Each booking request row entry within the report provides a quantity column indicating the number of bookings requested using the same information. This report is also used by the multiple policy jurisdiction demonstration to illustrate how changes in acquired information, such as to introduce end user anonymity, can affect reports and the ability of WSLogA Framework derived components to process captured information. The AbLodgeBookingPerspective uses JDO based components from

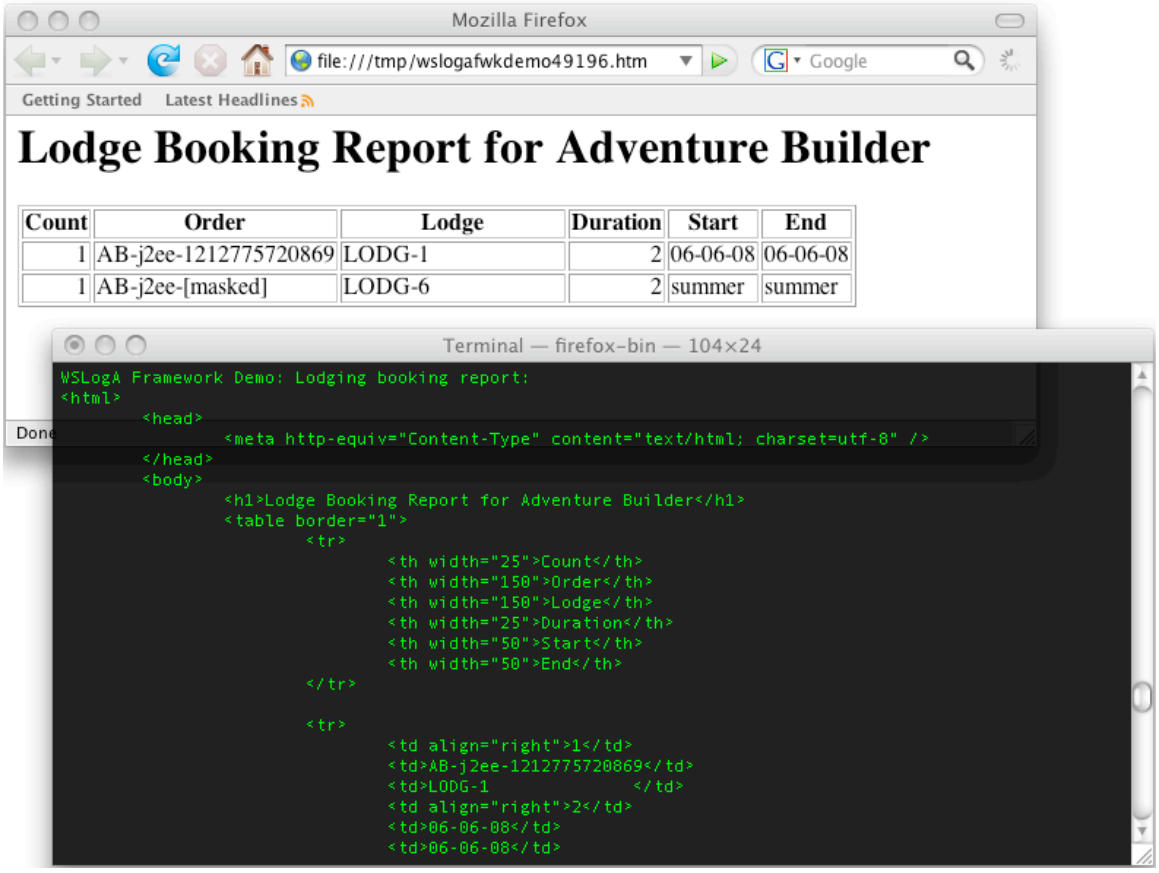


Figure C-12. The multiple policy report illustrates policy driven information formatting.

the Event Group to retrieve the event information and prepare a non-transactional EventPool object for consumption by AbLodgeBookingReporter.

Information collection is the central functionality of the WSLogA Framework. An extensive set of reporting and recording components are provided by the WSLogA Framework to facilitate a diverse range of information collection needs within Web service based systems. This demonstration makes use of the SoapHandlerMonitor and ObjectInspector components to provide an example of coordinated information collection within a transaction's workflow to accomplish multiple objectives—that of business report preparation and confirmation of

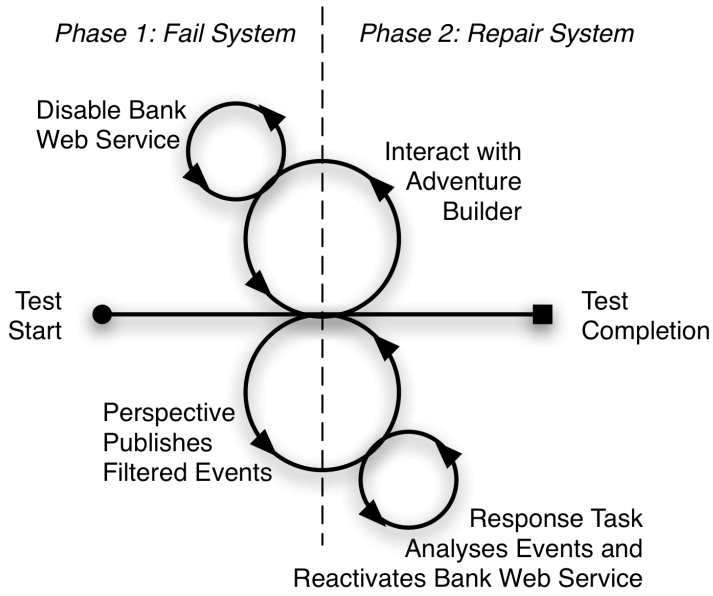


Figure C-13. Demonstration phases for failed Web service recovery.

data integrity within a complex workflow. The multiple uses for the same information pool underscore the importance of pervasive information harvesting by systems to accommodate data mining and knowledge applications.

Multiple Policy Jurisdictions

Web service applications provide the benefit of being able to operate across a diverse range of host systems, which enables organizations to form partnerships that lead to highly integrated information systems. Organizations may also benefit from being able to produce an information system in one country and host the system in multiple countries to take advantage of operational benefits specific to the available infrastructures. For example, a Web service could be developed to organize and provide access to digitized journal articles, but operate as a self-contained system on disparate university campuses. Legal jurisdictions may enforce information management policies, such as those pertaining to privacy, that are

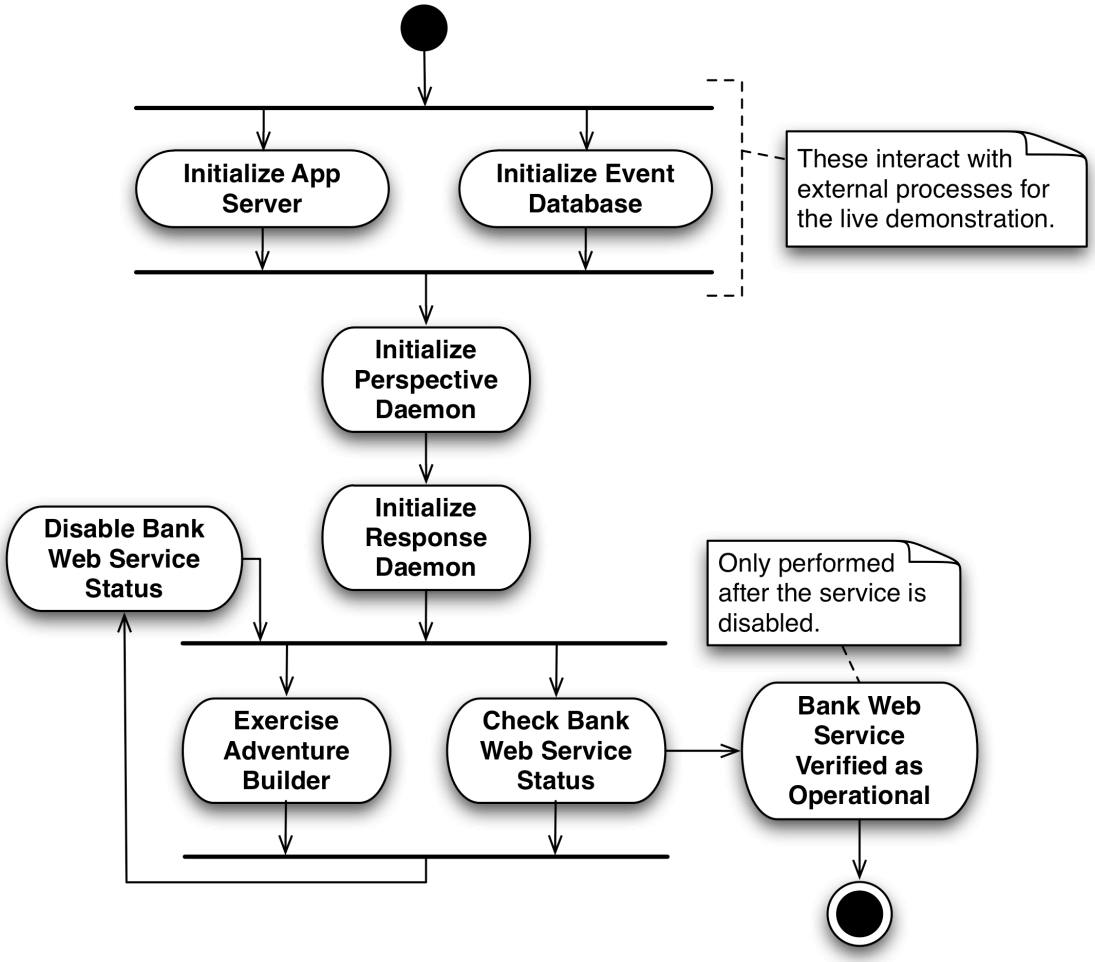


Figure C-14. Demonstration flow for failed Web service recovery.

not inter-compatible, which means for an architecture to be efficient the design needs to address both the business rules' mechanics and the varying information storage or presentation requirements. The WSLogA Framework's Policy Group enables both framework and third party components to focus on business logic by delegating information acceptance and formatting. Different information management policies can be introduced to the application by referencing the appropriate policy component library. This demonstration provides an

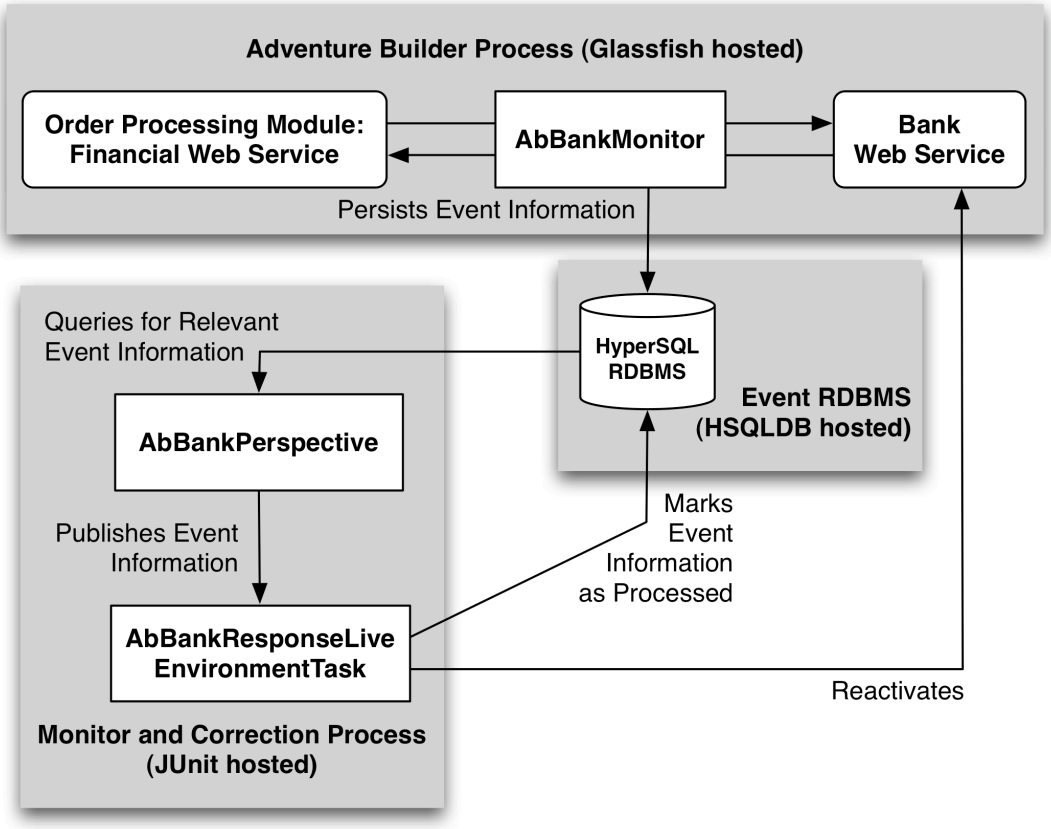


Figure C-15. Demonstration interaction for failed Web service recovery.

example of how different policy component sets can affect the information management behavior of a WSLogA Framework derived system.

The multiple policy demonstration's operation is integrated into that of the information collection demonstration because policy implementations affect the outcome for information collection. Policies can be static, such as denying redundant data, or dynamic, such as to appropriately mask sensitive information depending on a jurisdiction's requirements. Several strategies may be followed to implement policy exchange within WSLogA Framework based systems (Chapter 4), and this demonstration illustrates how contexts may be used to control individual policies. Figure C-8 illustrates context managed policy behavior.

```
<webservice-description>
   <webservice-description-name>
      CreditCardService
   </webservice-description-name>
   <wsdl-file>
      META-INF/wsdl/CreditCardService.wsdl
   </wsdl-file>
   <jaxrpc-mapping-file>
      META-INF/CreditCardServiceMap.xml
   </jaxrpc-mapping-file>
   <port-component>
      <description>port component description</description>
      <port-component-name>
         CreditCardIntfPort
      </port-component-name>
      <wsdl-port xmlns:CreditCardns="urn:CreditCardService">
         CreditCardns:CreditCardIntfPort
      </wsdl-port>
      <service-endpoint-interface>
         com.sun.j2ee.blueprints.bank.creditcardservice.CreditCardIntf
      </service-endpoint-interface>
      <service-impl-bean>
         <ejb-link>CreditCardEndpointBean</ejb-link>
      </service-impl-bean>
      <handler>
         <handler-name>AbBankMonitorHandler</handler-name>
         <handler-class>
            org.ws.loga.demo.failingservice.AbBankMonitor
         </handler-class>
      </handler>
   </port-component>
</webservice-description>
```

Figure C-16. Failed Web service recovery monitor descriptor entry.

The demonstration provides format policies that permit the capture data in its raw form as represented by a String value. These basic policies use a static context to ensure they are always available regardless of the application's operational context, which means data formatting behavior may be predicted for software development environments. Figure C-9 illustrates this relationship.

A set of strict formatting policies are also associated with each inspector, but these policies are associated with a context that only activates when a sentinel value indicates a

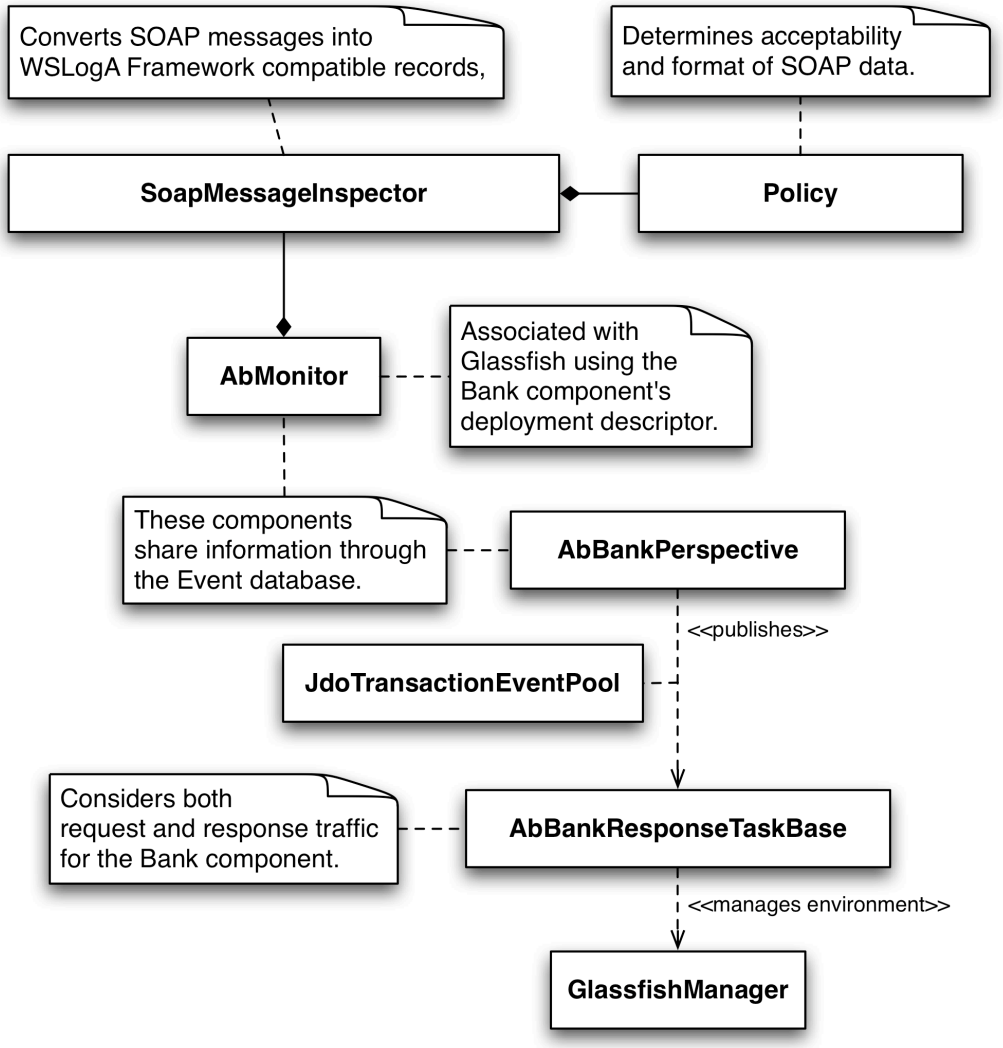


Figure C-17. Failed Web service recovery demonstration components.

strict information management jurisdiction is in effect. The sentinel value is controlled by the JUnit managed logic, which permits reports to be prepared demonstrating information collection outcomes across a variety of contexts. Figure C-10 illustrates this relationship.

The policy context and policy components can be made available to the application at any point within the system's class path, such as the application server's common library, the domain library, or the application archive. The multiple policy demonstration stores the

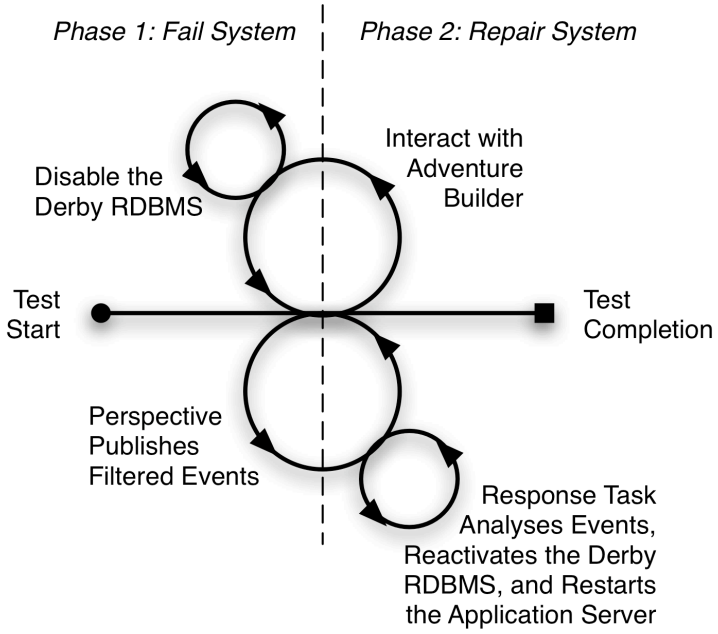


Figure C-18. Demonstration phases for failed application server database recovery.

policy and supporting components within the demonstration project's artifact, which is made available to the Adventure Builder application and WSLogA Framework extension components from within the application server's common library; however, objects instantiated as part of the policy system are active within the appropriate module class trees and thread memory assignments. Figure C-11 illustrates the interaction among the WSLogA Framework derived components and the operationally relevant Adventure Builder services.

The multiple policy demonstration concludes with the publication of an HTML based report (Figure C-12) that shows multiple lodging requests across at least one session with a relaxed requirement for information masking and a different session with strict information masking requirements.

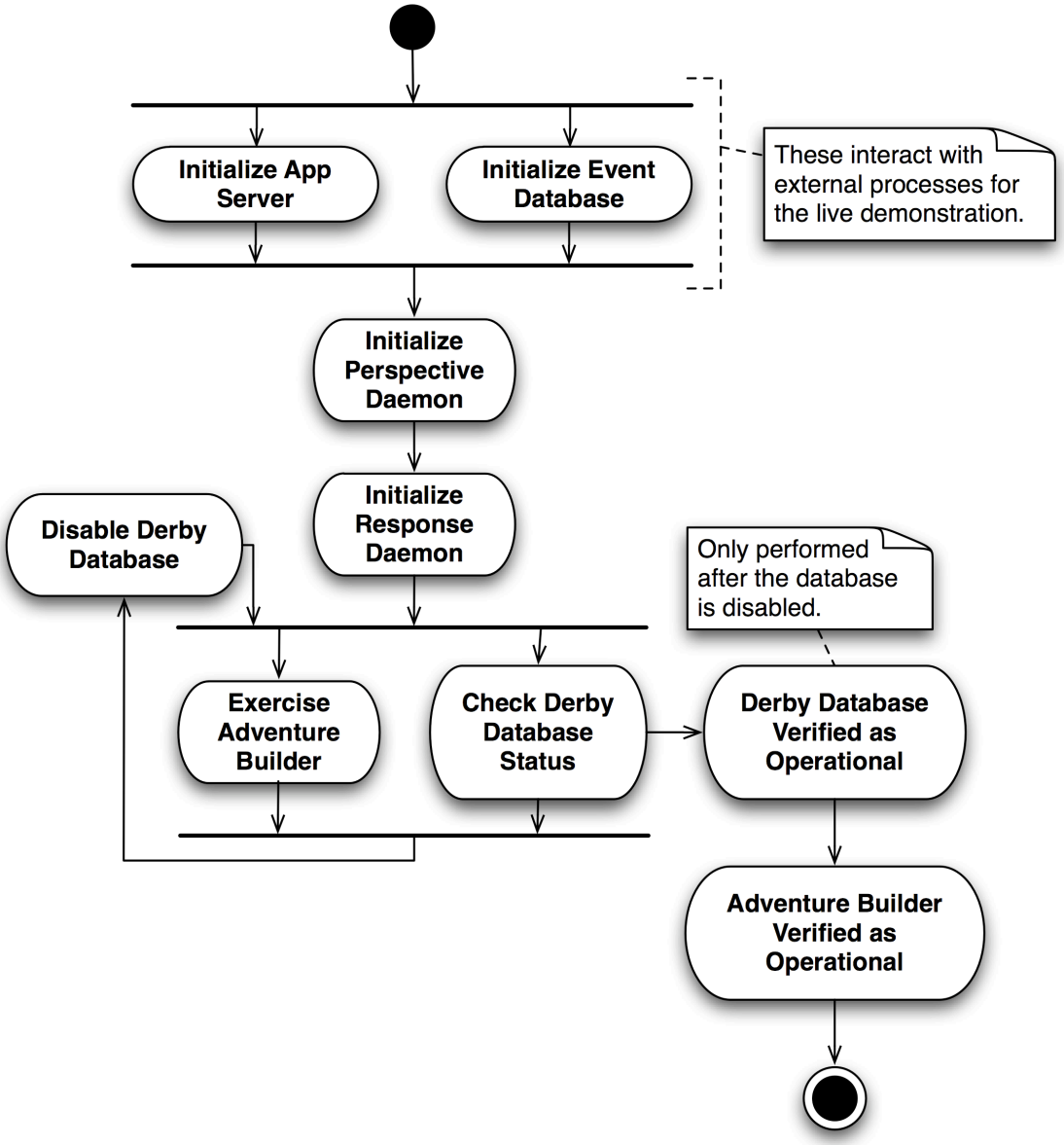


Figure C-19. Demonstration flow for failed database recovery.

Information capture performed by the same application architecture may be required across jurisdictions with varying information management requirements. The WSLogA Framework's policy support permits the development of a common architecture addressing business requirements with varied information acceptance and formatting policies according

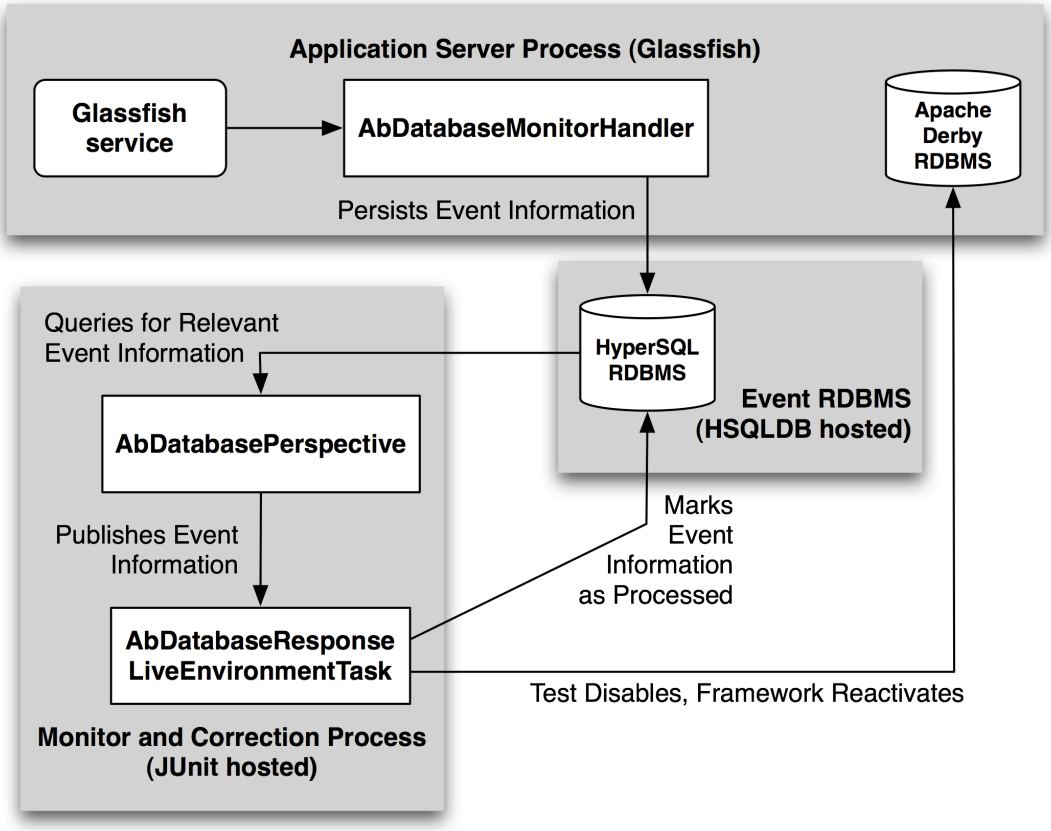


Figure C-20. Demonstration interaction for failed database recovery.

to a dynamic and interchangeable rules system. Inherent support within the WSLogA Framework for such functionality ensures that applications based on the Framework will have a growth path compatible with multinational organization concerns.

Detection and Recovery of a Failed Web Service

Web services are comprised of components that are subject to failure in result of misconfigurations, system resource limitations, and runtime exceptions (Cruz *et al.*, 2003; Cruz *et al.*, 2004; Graham *et al.*, 2005; Telles & Hsieh, 2001). Production environments must ensure the availability of their Web service components, which means these components must be

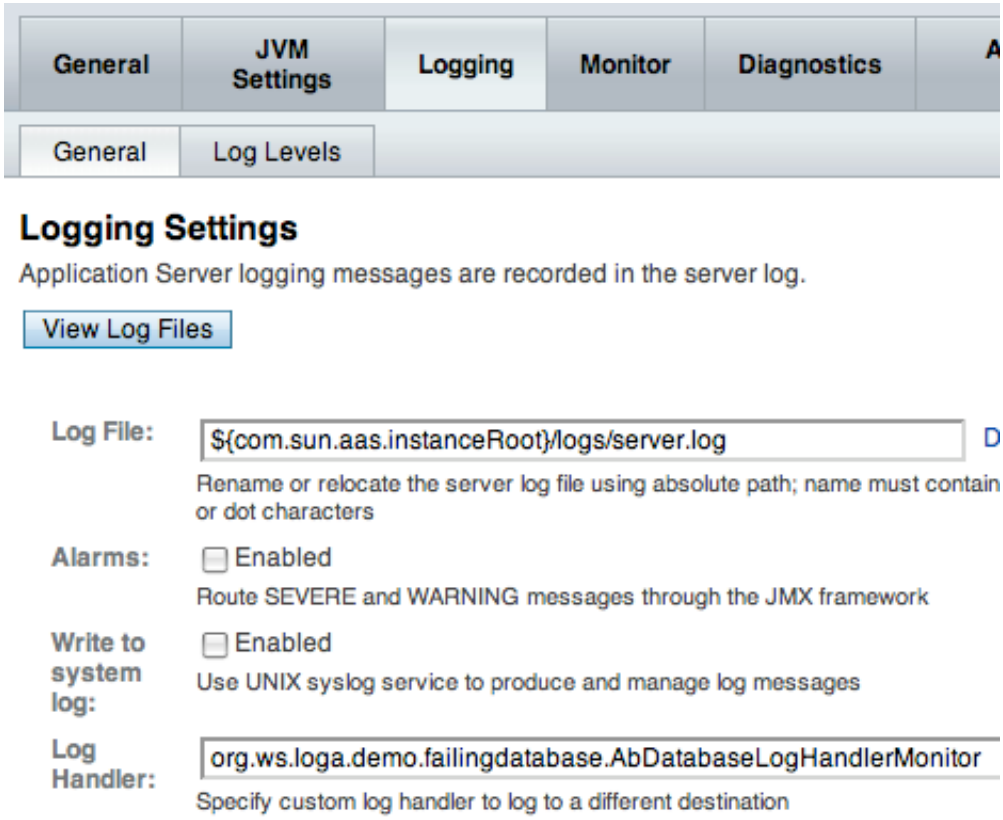


Figure C-21. The failed database recovery demonstration uses a custom Handler.

monitored for their availability and behavior (Cruz *et al.*, 2003; Cruz *et al.*, 2004; Lee *et al.*, 2002). In the event a Web service ceases to be available for application participation it needs to be restarted.

This demonstration provides an example of how a failed Web service may be detected and reactivated using components based on the WSLogA Framework. Self-healing systems can be difficult to establish (Babaoğlu, 2005; Telles & Hsieh, 2001), but use of the WSLogA Framework makes such monitoring and recovery straightforward.

Figure C-13 illustrates the phases provided for the consideration of WSLogA Framework component behavior. The demonstration begins with the standard operation of the envi-

ronment components to build an event history. The Bank Web service provided with Adventure Builder (Appendix B) is disabled after a history is established and the demonstration test logic monitors the Web service to determine when it is reactivated by a response task. A monitor observing the SOAP messages between the Bank Web service and its client Web service, Order Processing Center records SOAP events in a database managed by the WSLogA Framework. A perspective retrieves the relevant event information and publishes the information to a listening response task. The response task analyzes the event information and, if appropriate, interacts with the application server to reactivate the Bank Web service. Figure C-14 illustrates the activity sequence for the demonstration.

The Selenium oriented workflow deviates from the mock workflow in that the GlassFish application server and HSQLDB process are operated in independent processes. The use of distinct system contexts asserts the runtime utility of the WSLogA Framework components for the purpose of Web service recovery, and provides an example of component distribution across processes that are typically distinct in Enterprise systems. Figure C-15 illustrates the interaction among the WSLogA Framework derived components and the operationally relevant Adventure Builder services.

A monitoring component is associated with the Web service of interest through an entry in the service's configuration file, as illustrated in Figure C-16. The bold XML marks the entry for the WSLogA Framework component. The entry is minimal in that the monitor, which is a type of Handler (Graham *et al.*, 2005; Singh *et al.*, 2004), only requires a name and class reference. (The monitor class must be exposed to the classpath at runtime.)

The AbBankMonitor component employed in this demonstration extends the SoapHandlerMonitor provided by the WSLogA Framework, which in turn uses a policied Observer

component for message analysis and a Recorder component for persisting the event information into the WSLogA Framework's database. A process external to the application server (in this scenario, the test JVM) is established to host event information perspective and response components that handle Bank Web service management. AbBankPerspective queries the WSLogA Framework database for event information relevant to the Bank Web service and publishes the filtered information for use by listening AbBankResponseTaskBase components (mock and live extension variants are defined to enable appropriate environment interaction). The response component considers both request and response message counts in determining whether satisfactory Bank Web service operation is available, and when predefined tolerance levels are breached the response task interacts with the application server to reactive the Bank Web service. Figure C-17 illustrates the WSLogA Framework extensions for the failing Web service demonstration, which are provided in the demonstration module (Appendix D).

Self-healing systems (Dashofy *et al.*, 2002; Wang, 2005) are of interest to software engineers seeking to produce robust Enterprise solutions, such as those typically demanded of Web service systems that interact as critical components in partnerships. The Adventure Builder application provides a context suitable for simulating this relationship because the Bank Web service represents the role an independent financial institution would play for the adventuring booking organization. All Web service systems have the risk that a deployed Web service may fail, and the components derived from the WSLogA Framework for this demonstration provide a benchmark by which comparable services may be developed and compared for production environments. Third parties should also consider the related scenario of dependent resource failure, such as the loss of connectivity to a database integral

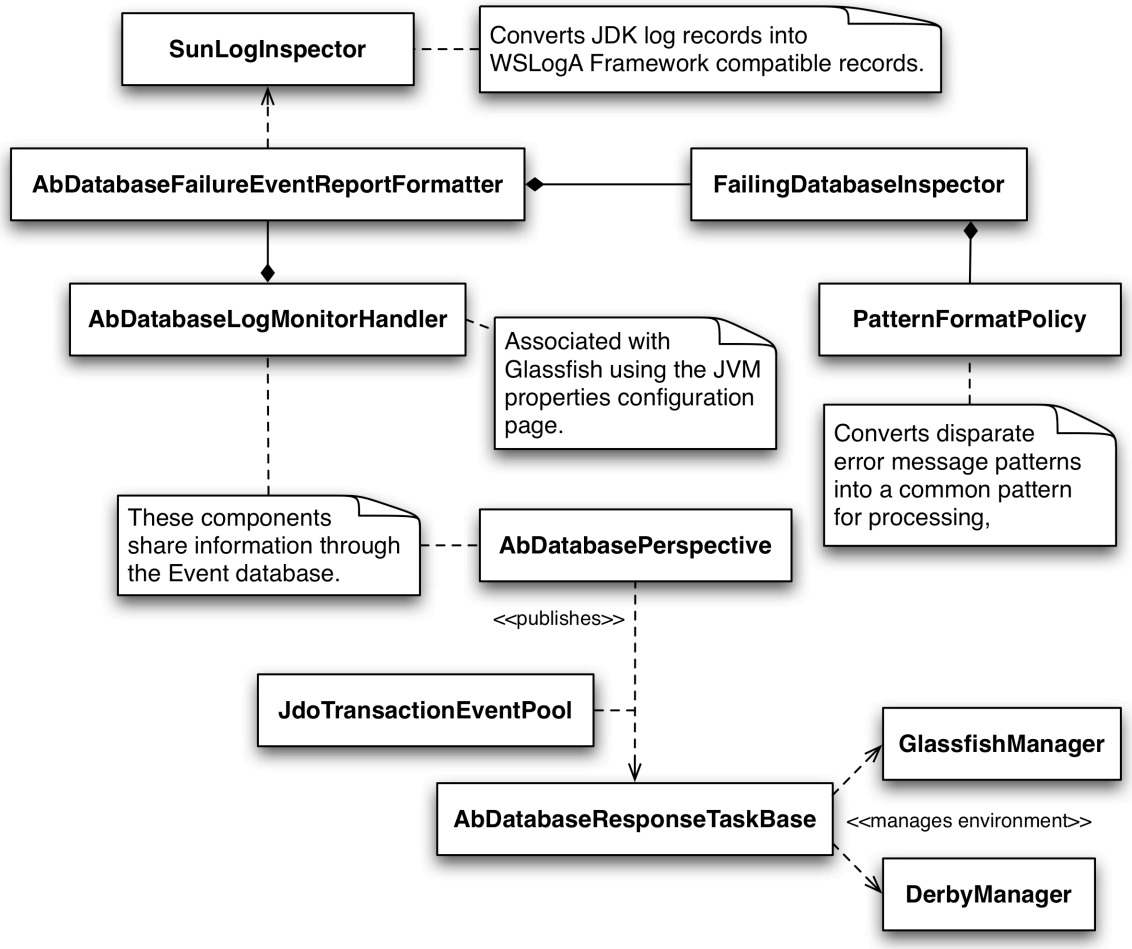


Figure C-22. Failed database recovery demonstration components.

to the Web service system. The Detection and Recovery of a Failed Database demonstration describes such a scenario.

Detection and Recovery of a Failed Database

Web services may depend on resources such as database connections for proper functionality, which means production environments must ensure that these resources remain available for use by the Enterprise system. In the event a resource dependency fails it must

be reactivated and, if appropriate, the dependent system must be recycled to ensure continued operation.

This demonstration provides an example for how a failed external resource—the Apache Derby database integrated with the GlassFish application server—may be detected and reactivated using components based on the WSLogA Framework. Figure C-18 illustrates the phases provided for the consideration of WSLogA Framework component behavior. The demonstration begins with the standard operation of environment components to build an event history. The Derby database manages application data and JMS queues for Adventure Builder Web services, which means its operation is essential to maintaining workflows. The Derby database is disabled after a history is established and the demonstration test logic monitors the database to determine when it is reactivated by a response task. A monitor observing log messages provided by GlassFish and its embedded components, such as the Derby database, records events in a distinct database managed by the WSLogA Framework. A perspective retrieves the relevant event information and publishes the information to a listening response task. The response task analyses the event information and, if appropriate, interacts with the application server to reactivate the Derby database and recycle inoperable Web services (*e.g.*, those that crashed because of database dependencies). Figure C-19 illustrates the activity sequence for the demonstration.

The Selenium oriented workflow deviates from the mock workflow in that the GlassFish application server and HSQLDB processes are operated in independent processes. The use of distinct system contexts asserts the runtime utility of the WSLogA Framework component for the purpose of database recovery, and provides an example of component distribution across processes that are typically distinct in Enterprise systems. Figure C-20 illustrates the

interaction among the WSLogA Framework derived components and the operationally relevant GlassFish resources or Adventure Builder services.

The GlassFish application server uses the J2SE Log API to record events for the server core, extension components (*e.g.*, the JMS queue), integrated systems (*e.g.*, the Apache Derby database), and hosted applications. The J2SE Log API must be configured in the JVM through system properties or a configuration file embedded in the JVM's distribution folder. The WSLogA Framework provides JdkLogHandlerMonitor as the entry point component by which event capture utilizing the J2SE log stream may be accomplished, and the demonstration component extending this component, AbDatabaseLogHandlerMonitor, is exposed for use by GlassFish through the application server's JVM options page, as illustrated in Figure C-21. Further configuration of Formatter or other components could also be provided by this configuration page but to simplify the demonstration these component relationships are established programmatically.

The AbDatabaseLogHandlerMonitor component deployed in this demonstration extends the JdkLogHandlerMonitor provided by the WSLogA Framework, which in turn manages a J2SE Log Formatter and JdoEventRecorder to prepare and persist event information provided by GlassFish log requests. A process external to the application server (in this scenario, the test JVM) is established to host event information perspective and response components that handle Apache Derby database management. AbDatabasePerspective queries the WSLogA Framework database for event information relevant to the Apache Derby database and publishes the filtered information for use by listening AbDatabaseResponseTaskBase components (mock and live extension variants are defined to enable appropriate environ ment interaction). The response component considers SQLException and related errors to deter-

mine that the Apache Derby database has been disabled. Figure C-22 illustrates the WSLogA Framework extensions for the failing database demonstration.

Self-healing systems (Dashofy *et al.*, 2002; Wang, 2005) are of interest to software engineers seeking to produce robust Enterprise solutions, such as those typically demanded of Web service systems that interact as critical components in partnerships. The GlassFish application server provides a context suitable for simulating this relationship because the Apache Derby database represents the role an RDBMS would play for a J2EE application. All Web service systems have the risk that the associated database may fail, and the components derived from the WSLogA Framework for this demonstration provide a benchmark by which comparable services may be developed and compared for production environments. Third parties should also consider the related scenario of Web service failure. The Detection and Recovery of a Failed Web Service demonstration describes such a scenario.

Summary

The WSLogA Framework is a holistic solution for information capture, analysis, and environment management. The provided information capture mechanisms target SOAP messages, runtime objects, and log messages generated by foundational systems such as the application server. Policy component influence the extent to which information is persisted and the manner of its presentation format, which permits common architectures to developed for applications that must respect information policies for diverse jurisdictions. Response formation and execution mechanisms permit environment correction and reporting that accommodates machine and human audiences. The workflows and component behav-

iors automated by the framework address essential and complex relationships necessary for advanced production support and management.

The WSLogA Framework's configurations, workflows, and components are demonstrated by means of a pre-configured VMware virtual machine operated by Windows XP. The WSLogA Framework's source code has been installed and its resultant artifacts and project reports are made available for inspection. The GlassFish application server and its associated database, Apache Derby, are configured for use with the Adventure Builder application with modifications for WSLogA Framework integration. The Eclipse IDE and Maven2 build engine are installed and configured for use with the WSLogA Framework project and GlassFish to support unit and integration tests that consistently execute and validate the WSLogA Framework. Researchers may use the virtual machine to assess the WSLogA Framework. Practitioners may use the virtual machine to better understand or enhance the WSLogA Framework. Adjustments to the WSLogA Framework can be compared against the baseline provided by the original demonstration virtual machine associated with this report.

Appendix D

Configuration Management

Introduction

The complexity of software development requires a disciplined, consistent approach to the production of the documents, artifacts, and environments necessary for the rigorous exploration of a technology (Berczuk & Appleton, 2003; Hevner *et al.*, 2004; Mason, 2006). This section discusses the configurations used for this investigation with the intent of facilitating result reproduction, and complements several other appendixes within this report: Appendixes A and C address the quality control strategies employed to ensure rigor; E and F continue this section's discussion by respectively detailing the version control and automation strategies; and Appendix G overviews the development and audit reports.

Host Environment

All project phases made use of the Apple Macintosh platform, operated by Mac OS X, for which the J2SE 1.5 SDK is a standard component (Apple Computer, 2008). Applicable service packs released during the course of research were applied to the operating system, and are reflected in the operating system's version number as provided in Table D-1. This environment adequately represents industry enterprise environments for which the WSLogA Framework would be an enhancement (Sun Microsystems, 2008c; TheServerSide.com, 2005).

Tests were also conducted using the Microsoft Windows platform (Appendix C), which suggests that the WSLogA Framework can be generally employed throughout a heterogeneous environment.

Table D-1. Hardware and software platforms.

Platform		Version
Apple Macintosh	Model	MacBook
	CPU	Dual Core 2 at 2 GHz
	RAM	3 GB
	Disks	120GB with 60 GB partition for research documents and runtime
Mac OS X	10.5.3 (9D34) integrated with Darwin 9.3.0 (Berkley UNIX variant)	
VMware Fusion	Version	1.1.1
	Model	X86 Intel compatible
	CPU	Reflects host PC configuration
	RAM	2 GB
	Disks	50GB
WindowsXP	NSU MSDN as licensed for SCIS student activities, Service Pack 2 (SP2)	

The GlassFish Application server (Sun Microsystems, 2008a) was utilized in conjunction with the Adventure Builder application to host WSLogA Framework tests. GlassFish is a J2EE 1.4 compliant system, and is representative of the J2EE Application servers used throughout the industry (Sun Microsystems, 2008c; TheServerSide.com, 2005). JBoss (2008) was proposed as the Application server, but environment configuration efforts for Adventure Builder exposed a bug by which the proposed server incorrectly handled Web service deployments requiring JAX-RPC support (JBoss failed to generate the Web service component stubs for Adventure Builder). GlassFish correctly handles all J2EE, Web service, and Adventure Builder

requirements. Table D-2 describes the library files (*e.g.,* JARs) added to GlassFish or the unit test environment (simulating system use within an application server) to support operation of the WSLogA Framework. Where possible, descriptions are from the perspective of the Maven project file. Library build scopes are defined in Maven 2 terms (Appendix F).

Table D-2. Libraries and components.

Artifact	Version	Scope	Declared
javaee (1.4 as distributed with GlassFish v1)	9.0_01	Compile	Y
hsqldb	1.8.0.7	Test	Y
easymock	2.2	Test	Y
easymockclassextension	2.2	Test	Y
selenium-java-client-driver	0.9.2	Test	Y
appserv-ws	9.0_01	Provided	Y
j2ee (1.4 as distributed with GlassFish v1)	9.0_01	Provided	Y
j2ee-svc (1.4 as distributed with GlassFish v1)	9.0_01	Provided	Y
ant	1.6	Compile	n
dom4j	1.6.1	Compile	n
geronimo-spec	1.0.1B-rc2	Compile	N
jakarta-regexp	1.4	Compile	N
jdo2-api	2.0	Compile	N
connector	1.0	Compile	N
jta	1.0.1B	Compile	N
jaxen	1.1-beta-8	Compile	N

Table D-2. Libraries and components.

Artifact	Artifact	Artifact	Artifact
jdom	1.0	Compile	N
jpox	1.1.7	Compile	Y
jpox-enhancer	1.1.7	Compile	Y
jpox-maven-plugin	1.1.7	Provided	Y
log4j	1.2.14	Compile	Y
bcel	5.2	Compile	N
xalan	2.7.0	Compile	N
xercesimpl	2.6.2	Compile	N
xmlparserapis	2.6.2	Compile	N
xml-apis	1.0.b2	Compile	N
xom	1.1	Compile	Y
cglib-nodep	2.1_3	Test	N
commons-logging	1.0.4	Test	N
servlet-api	2.4	Test	N
jetty	5.1.10	Test	N
junit	3.8.2	Test	N
selenium-core	0.8.3	Test	N
selenium-server	0.9.2	Test	N
selenium-server-coreless	0.9.1	Test	N

One system account was established on each investigation system—the Mac OS X workstation and Windows VMware virtual hard disk—and used for all project phases to ensure consistent variable configurations. Environment variables, such as CLASSPATH, were configured in the appropriate system registry (*e.g.*, the UNIX .profile file and the Windows System control panel) or in a build property file. These files are included in the research archive to facilitate result reproduction in other environments.

Table D-3. Environment variables.

Variable	Value
ANT_HOME	$J2EE_HOME/lib/ant
CLASSPATH	$CLASSPATH:$J2EE_HOME/lib/j2ee.jar:$J2EE_HOME/lib/javaee.jar:$J2EE_HOME/lib/j2ee-svc.jar:/Volumes/Media/dev/wslogafwk/lib/ydoc-2.2_03-jdk1.5/lib/ydoc.jar:/Volumes/Media/dev/wslogafwk/lib/ydoc-2.2_03-jdk1.5/lib/class2svg.jar:/Volumes/Media/dev/wslogafwk/lib/ydoc-2.2_03-jdk1.5/resources
GLASSFISH_HOME	/Applications/ appserver /GlassFish
J2EE_HOME	$J2EE_SERVER
J2EE_SERVER	/Applications/appserver/GlassFish
M2_HOME	/Applications/maven-2.0.7
PATH	$PATH:$M2_HOME/bin:$J2EE_HOME/bin:$ANT_HOME/bin:$SVN_HOME
WSLOGA_DEMO_SERVICE_PACKAGE_HOME	/Volumes/Media/dev/advbuilder

Test Environment

A test environment was used to facilitate the WSLogA Framework's evaluation. Archiving or scripting pre-configured environments and re-instating such as needed through an automated process ensured consistent test environment preparation (Berczuk & Appleton, 2003; Haftmann *et al.*, 2007; Hunt & Thomas, 2006; Rainsberger & Stirling, 2005). The consistency permits comparative consideration of WSLogA Framework or environment changes across development iterations, and the availability of the archived environments permit independent evaluation of results. Two distinct test environments were maintained: unit tests executable from within the IDE or an automated build (Appendix A), and a functional test environment involving GlassFish and Adventure Builder (Appendix C).

The unit tests use the JUnit framework, which permits an organized approach to minimal environment preparation (*e.g.*, a database pool), test execution, reporting, and environment cleanup (Appendix A). Each WSLogA Framework component was developed in parallel with a test component intended to ensure that all pre- and post-conditions mandated by component methods were satisfied. All unit tests were designed to avoid the need for an Application server to ensure maximum test efficiency during development. Unit tests were executed both from within the IDE and as part of a WSLogA Framework build using the Maven or Ant scripts developed to facilitate this investigation. The SureFire plug-in for Maven managed unit testing and produced reports describing unit test outcomes (Appendix F). All unit test sources as well as the final build's unit test report are included in the project archive associated with this report (Appendix C).

Cobertura was utilized to produce source code execution maps in conjunction with unit test activities (Appendix F). Cobertura instruments Java class files prior to unit test execution,

monitors unit test execution to identify source code executed during the tests, and reports those aspects of source code executed during the unit tests (Appendix G). Analysis of the execution maps permits the refinement of unit tests to ensure all important source code elements are addressed by one or more unit tests.

The functional tests require the execution of the test application and WSLogA Framework components (Appendix A). Limited automation of these evaluations was supported by JUnit, but most testing occurred through the manual handling of the test system as specified by scenario scripts documented using the Selenium functional test tool. SQL scripts provide data facilitating the functional tests, and are included in the project archive associated with this report.

Development Documents and Tools

The investigation utilized tools, environments, and documents commonly found within the IT industry to ensure general applicability of the research results for researchers and practitioners. Proposed tools, environments, and documents for the investigation remained static as appropriate, but products were upgraded or replaced as bugs were discovered or significant enhancements were made available by vendors (such as with the Application server and version control system). This approach provides an appropriate balance between result consistency and the availability of an appropriate lab environment for project work. Table D-4 describes the final versions of significant applications and plug-ins utilized to produce the documents and artifacts for the investigation.

Table D-4. Applications and significant plug-ins.

Application or Components	Version	Purpose
Eclipse IDE	3.3.1.1	Java and other document editor.
BBEdit	7.0.3	Multiplatform text editor for Java source code, XML documents, and related content.
Subversion	1.4.4	Version control system used to manage project source code, documents, and environment configurations.
Subclipse	1.2.4	Eclipse IDE integration with Subversion.
Maven	2.0.7	Build, test, and report engine (includes Ant).
VMware Fusion	1.1.1	Virtualizer for the WindowsXP platform used to distribute and demonstrate the WSLogA Framework.
OmniGraffle Pro	4.2.2	UML and other document preparation.
yDoc	1.1	Automated UML document preparation.
GlassFish Application server	v1 (9.0_01)	J2EE 1.4 compliant application server used to host Adventure Builder and the WSLogA Framework demonstration.
Adventure Builder	1.0.5	J2EE 1.4 Web service based system simulating a travel booking system used to facilitate WSLogA Framework quality control and demonstration.
HyperSQL Relational Database Management System (HSQLDB)	1.8.0.7	Relational Database Management System with extensions for Java application integration. Used to facilitate WSLogA Framework quality control and demonstration.

Applications and Environment

The Eclipse IDE and BBEdit text editor were utilized for Java and script implementation. The Eclipse IDE is a popular open architecture development platform with broad community support from key vendors (Eclipse Foundation, 2008). The IDE's functionality can be repaired (such as to eliminate bugs) or extended through the installation of plug-in components. The BBEdit text editor is a popular Mac OS X text editor with support for Java and XML documents produced or maintained on a variety of platforms (Bare Bones, 2007; MacWorld, 2005a).

Subversion and the Eclipse IDE Subclipse plug-in were utilized for managing versions of the investigation's documents. Subversion's role in the investigation's version control strategy is discussed in Appendix E, but in summary Subversion is a version control tool intended by its developers to replace CVS (Berczuk, 2003). CVS was proposed for the investigation and utilized until April 2007, but was replaced by Subversion after the Subversion development team addressed key bugs with version 1.4.3 (Tigris.org, 2007). Subclipse enables direct repository access and control through the Eclipse IDE interface, which makes convenient the management of research activities such as exploratory development.

The Maven and Ant build systems were utilized for producing investigation artifacts from the source documents and controlling unit tests (Appendix F). Ant is a declarative scripting language and platform for controlling the manner by which artifacts are produced; Maven incorporates and extends Ant functionality by providing a common build platform oriented around industry best practices for source and artifact document organization, build workflow, and other configuration or engineering related activities. Maven integrates with the Eclipse IDE through the use of the Maven 2 Eclipse Integration plug-in.

VMware Fusion was used to produce the virtualized distribution of the WSLogA Framework based on the WindowsXP platform (Appendix C). The VMware product family permits x86-based environments, such as a WindowsXP system, to be established using portable virtual hard disk files, which can then be operated on physical x86 systems with similar performance yet in a manner completely distinct from the host (VMware, 2008). VirtualPC was proposed for this task, but as the Mac OS X platform for VirtualPC only executes on PowerPC systems it was eliminated by Microsoft as a product line with the advent of Intel based Macintosh PCs (MacWorld, 2006). The VMware virtual hard disk can be executed to run tests within the WindowsXP context, and reverted to its original state to ensure subsequent tests can be compared with appropriately identical settings (*e.g.*, database state).

OmniGraffle Pro and yDoc were utilized to produce UML diagrams. OmniGraffle provides the Mac OS X platform with diagramming support compatible with Microsoft Visio (MacWorld, 2005b). yDoc is a JavaDoc extension library that produces UML class diagrams for incorporation into industry standard JavaDoc documentation produced by the JavaDoc tool (yWorks, 2008). Diagrams prepared using each application were used to envision and document the WSLogA Framework.

Design Documents

Design documents were produced using OmniGraffle Pro and yDoc to visualize the WSLogA Framework's architectural evolution throughout research iterations with an emphasis on class, activity, and sequence UML diagrams (Appendix G). Class diagrams describe the structural relationship between WSLogA Framework components, such as framework extension nodes (hot spots) and increasing degrees of component functionality

through inheritance. Activity diagrams describe execution- or workflows and applied to both source code and tests. Sequence diagrams describe component interaction.

Source and Supporting Documents

Java, SQL, configuration, and build files were produced as source documents for the investigation. Java produced for the WSLogA Framework is limited to J2SE 1.5 and J2EE 1.4 functionality to ensure reasonable industry applicability. SQL scripts adhere to common SQL-99 and SQL-2003 features (Toussi, 2008) to ensure compatibility with the HSQLDB system employed to facilitate quality control for the WSLogA Framework. Use of SQL standards should ensure general portability of the SQL scripts to other database systems, such as Oracle or MySQL. Configuration files provide custom session behavior for the Application server, database, Adventure Builder application, WSLogA Framework, and build system. The configuration files produced for the WSLogA Framework are structured using XML or adhere to the INI strategy common for Java oriented properties files. Maven and Ant build scripts are formatted according to XML schemas published by their respective development teams.

Test and Supporting Documents

Java and JavaScript documents were utilized for unit and functional tests. JUnit classes were developed in tandem with the tested source code as a method for assisting the discovery of architecture requirements and a mechanism for verifying implementation correctness. Selenium JavaScript scripts were developed to provide automated navigation of the Adventure Builder and demonstration system user interfaces as a means to verify functional behavior. SQL documents were used to manage the Adventure Builder and demonstration

system's database content. Initialization scripts reset the databases to ensure test results were not polluted by development activities or prior tests.

Development Artifacts

The investigation resulted in Java, database, and documentation artifacts. These artifacts represent the goal of the investigation's project and can be used to validate the WSLogA Framework's behavior or in support of new projects.

Compiled Documentation

The JavaDoc, yDoc, Cobertura, and SureFire tools were utilized to produce API and testing documentation (Appendix G). JavaDoc documentation includes UML class diagrams produced by the yDoc plug-in for JavaDoc. Cobertura reports describe the degree of source code coverage by the unit tests, and SureFire reports describe unit test outcomes. Third parties can analyze the documentation to gain an understanding of the functionality made available by the WSLogA Framework and the manner by which that functionality may be incorporated into a new project.

Binaries, JARs, WARs, and EARs

The WSLogA Framework is comprised of a JAR file set that includes the classes and other resources necessary for the WSLogA Framework's utilization by another project. Third parties can include in their projects the WSLogA Framework's JARs as dependencies to gain an implementation foundation for their custom WSLogA architecture and configuration. Third party libraries required by the WSLogA Framework JARs must be externally configured

within the host environment. For example, the XOM library was placed into GlassFish's lib folder to ensure its availability to the wsloga-framework-1.0-SNAPSHOT.jar.

The Adventure Builder application is comprised of a WAR containing the Web application as well as a JAR set containing the supporting Web services (Appendix B). These artifacts may be reproduced using the Adventure Builder project included in the VMware distribution file, and they are made available for use in integration tests within the domain1 server instance provided with the VMware distribution file.

Databases

The GlassFish application server is bundled with the Apache Derby database (Sun Microsystems, 2008b), which was used to host Adventure Builder's seed and session data. Application seed data was injected into the Derby database using SQL scripts provided with the GlassFish project sources. Session data was generated during tests, and could not be removed other than by resetting the host environment's state (*e.g.*, by using VMware's snapshot and rollback functionality).

The WSLogA Framework uses the HSQLDB relational database management application (Hsqldb.org, 2008) to manage captured information (Appendix A). The WSLogA Framework's test environment involved both disk and in-memory HSQLDB sessions, and both manifestations were reset prior to the subsequent execution of tests when using the Maven2 build instructions established for the project (Appendix F). SQL scripts are provided with the WSLogA Framework to facilitate environment configuration and maintenance, as well as to support subsequent research.

Summary

The development and demonstration of the WSLogA Framework involved a multitude of documents, tools, and environments. A strict configuration management approach to ensuring consistency across formats, versions, and utility was maintained to facilitate this investigation's iterative development and quality control practices. The tools and environments were selected for their ability to represent affordable, common technology platforms likely to be found within Enterprise development environments, which further ensures that the WSLogA Framework serves as a relevant and accessible technology.

Appendix E

Version Control

Intent

Configuration management is concerned with accuracy—specific document and environment versions provide anticipated behavior for software releases (Bar & Fogel, 2003; Berczuk & Appleton, 2003; Casey *et al.*, 2006; Enes, 2007; Estublier *et al.*, 2005; Mason, 2006). The Subversion (Mason, 2006) version control system was used to manage the evolution of design, development, and test documents. ZIP archives (PKWare, 2007) were used to organize the environment components necessary for testing and analysis, such as the Application server. Maven (Casey *et al.*, 2006) was used to manage third party explicit and transitive artifacts supporting the WSLogA Framework's functionality or build process. This section describes the processes and tools used to manage the sources and artifacts for this investigation.

Subversion and CVS

This investigation made use of the Concurrent Versions System (CVS) (Bar & Fogel, 2003) and Subversion (Mason, 2006) version control systems. CVS is bundled with many operating system distributions (Bar & Fogel, 2003), such as Mac OS X (Apple Computer, 2006), and enjoys widespread support within the software development industry (Berczuk & Appleton, 2003). Subversion is a modern version control system intended to replace CVS through the

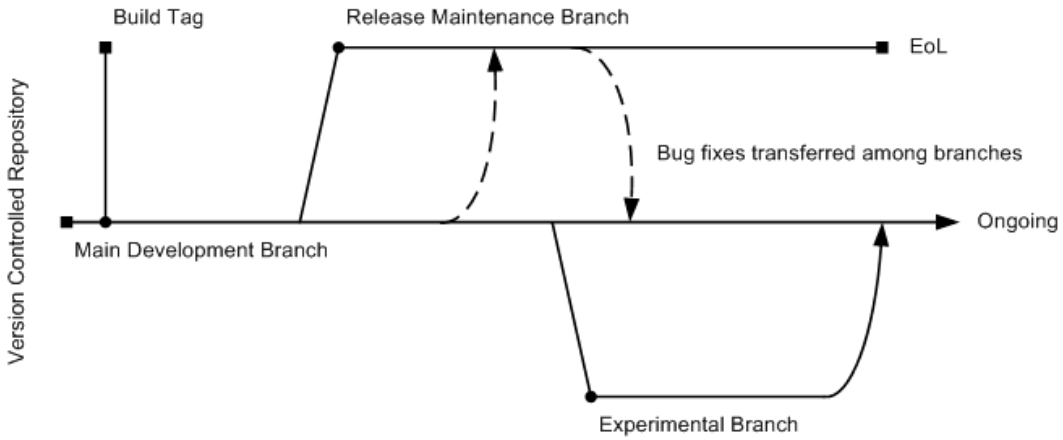


Figure E-1. The version control process.

use of a distinct source code base and the incorporation of lessons learned by the industry in its use of CVS and other version control systems (Collins-Sussman *et al.*, 2004; Mason, 2006). CVS was proposed and initially utilized for the investigation because of its maturity and availability at the time of the proposal; however, Subversion has superior version management capabilities (Mason, 2006) and was adopted for the investigation in May 2007 after its development team addressed a series of critical bugs with the release of Subversion version 4.3 (CollabNet, 2007).

Subversion's popularity among Open Source development teams has resulted in broad product support. The Eclipse IDE directly integrates with Subversion through the use of the Subclipse (CollabNet, 2006; Herborth, 2006) plug-in to support local development with seamless version control. Maven and Ant can execute command line statements for interaction with any command line based application (Casey *et al.*, 2006), but Maven also provides built-in support for Subversion that accommodates version control processes typical for Java development projects.

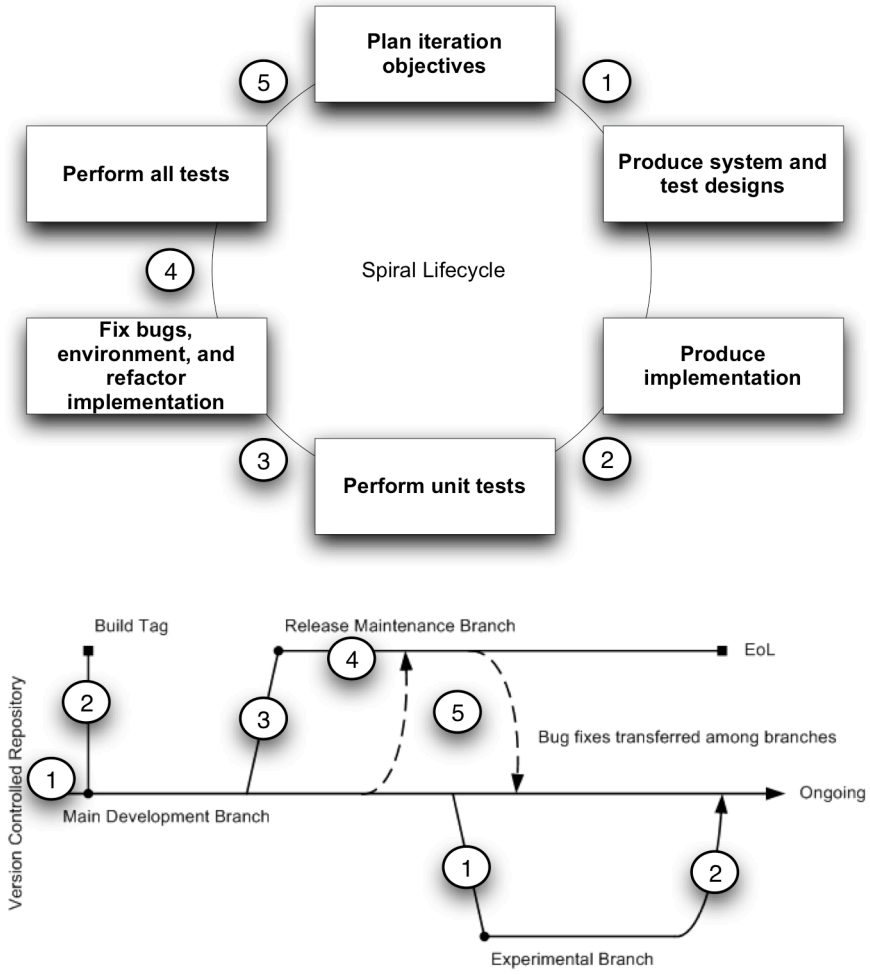


Figure E-2. The version control process as applied to this investigation.

ZIP Archives

The ZIP format enjoys widespread adoption among the significant software development and operating platforms, including the Java SDK (Arnold *et al.*, 2005; Sun Microsystems, 2003) or operating systems such as Mac OS X (Apple Computer, 2007) and Windows XP (Microsoft, 2004). The GlassFish Application server and supporting components, such as the bundled Derby database engine, were regularly adjusted to reflect the needs of testing and analysis

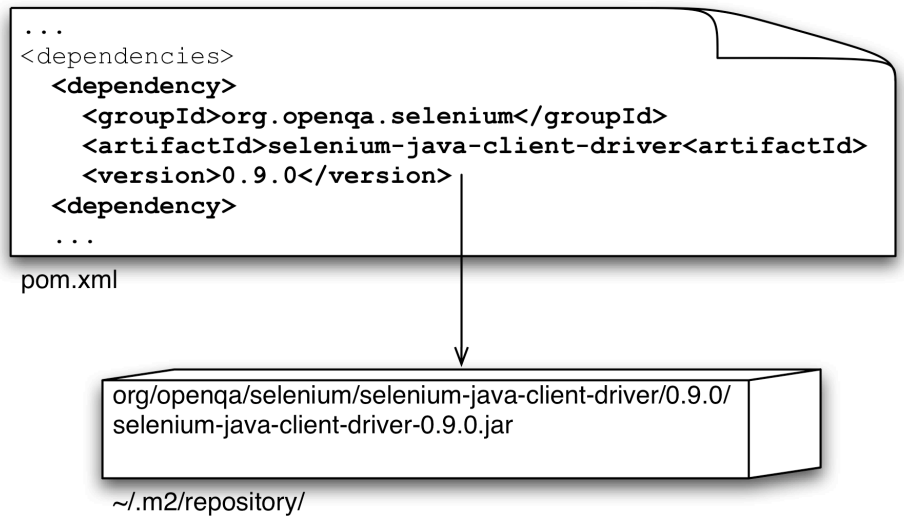


Figure E-3. Project object model file declares dependencies.

throughout the investigation. Preferred component configurations were bundled within a single archive and unarchived as necessary to provide subsequent tests with a fresh environment (Rainsberger & Stirling, 2005).

Maven

Maven is a configuration management and build tool produced by Apache to manage the complex build and release processes for the group's multitude of open source projects (Casey *et al.*, 2006; Enes, 2007). Maven was designed using lessons learned and best practices for information technology projects within the industry (Casey *et al.*, 2006), and as such is suited for the development and release of frameworks such as the one produced by this investigation. Maven incorporates Ant, provides integrated Subversion connectivity, and is supported by development tools such as Eclipse through the use of third party plug-ins (Casey *et al.*, 2006; Mergere, 2007). Maven was used to manage the WSLogA Framework's

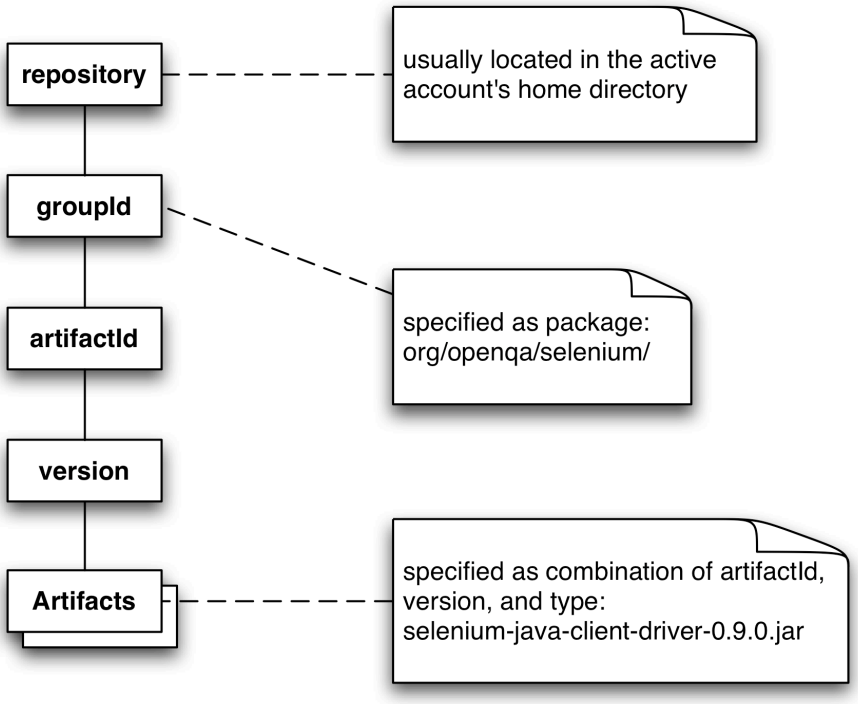


Figure E-4. Maven repository organization.

build and reporting processes as well as to organize third party artifacts required for WSLogA Framework functionality or development in fulfillment of the software configuration management third party code line pattern (Berczuk & Appleton, 2003).

Source Management

Source files were developed on a local workstation and stored within a Subversion repository. These workspaces are organized within the repository and local work environment according to design patterns obtained from agile and iterative software development best practices (Bar & Fogel, 2003; Berczuk, 2003; Berczuk & Appleton, 2003; Casey *et al.*, 2006; Mason, 2006). Figure E-1 illustrates the repository organization, which was optimized for an iterative process accommodating multiple threads of simultaneous work and limited re-

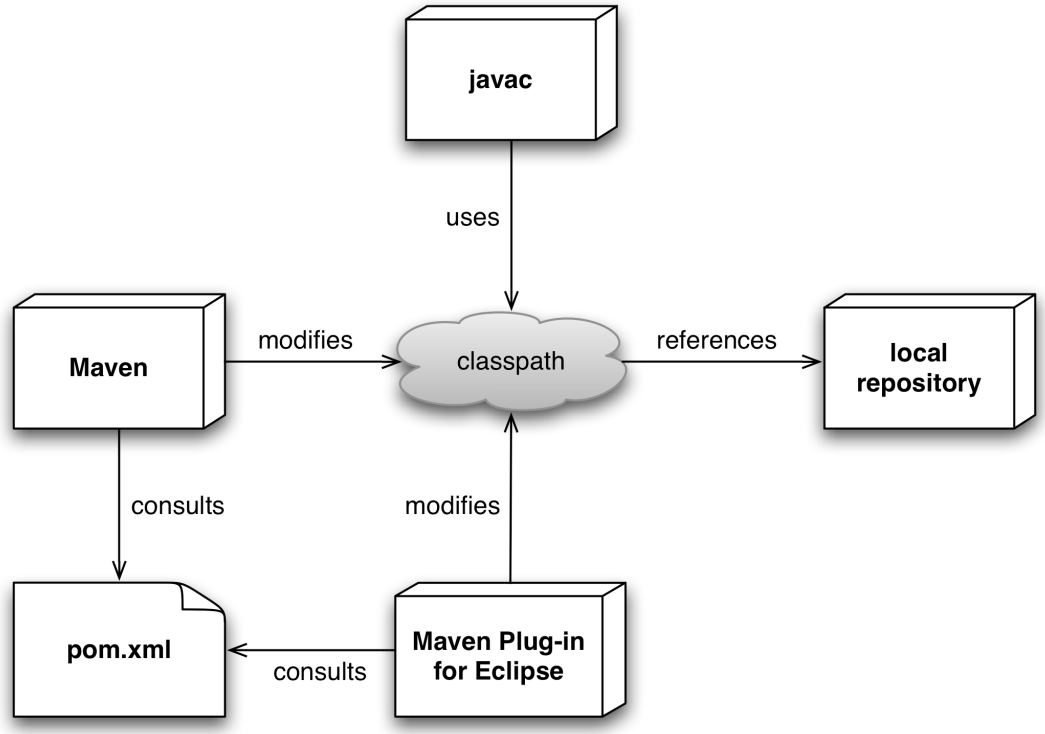


Figure E-5. Maven modifies the classpath during build operations.

leases. Changes to designs and implementations were easily tracked for comparative analysis over time and to provide rollback points for when a tentative effort did not provide the desired functionality. A single local workspace, such as an Eclipse project, only contained source for one branch.

Figure E-2 illustrates the relationship between the version control strategy implemented for this investigation and the development methodology utilized for artifact identification and creation (also see Figure 3-1). A main development branch contained primary design, implementation, and test source files for the WSLogA Framework. Iterations started with the acquisition of source files obtained from the main development branch and placed into a local workspace. Optionally, an experimental branch was used if the envisioned implementa-

tion might have resulted in a significant architectural change to the WSLogA Framework. Completed artifacts and source files were committed to the main development branch and tagged for reference. A release branch was prepared after successful unit and integration testing for the full WSLogA Framework, which ensured a stable artifact and source file set that could be used as a benchmark with subsequent work and tests. Bug fixes identified by subsequent testing were committed to the release branch and merged with the main development branch.

Dependency Management

Maven based projects declare explicit dependencies on third party components, such as JAR files, within a project object model represented by XML in a pom.xml file (Casey *et al.*, 2006). Maven understands the transitive dependency model for many third party components packaged for use with Maven, such as JUnit, and manages these ancillary artifact requirements on behalf of the project. Figure E-3 illustrates the relationship between the pom.xml and third party dependencies within the build environment.

Maven repositories are organized in general accordance with the Java package standard (Arnold *et al.*, 2005; Casey *et al.*, 2006), as illustrated in Figure E-4. A root directory, /repository, contains artifacts organized by their group ID, artifact ID, and version. Group IDs are structured as directory paths and may be specified by an organization to distinguish its artifacts from similarly named artifacts provided by other organizations. The artifact ID identifies the component in terms of a functional theme, and the version distinguishes between multiple releases of the same artifact. Components are labeled with a combination

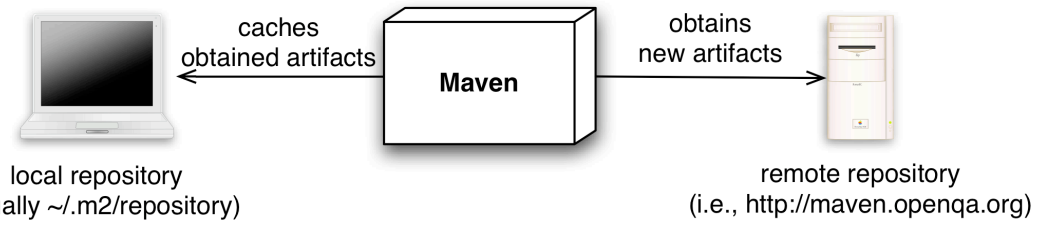


Figure E-6. The local repository is updated from remote repositories as necessary.

of the artifact ID, version, and artifact type (*e.g.*, JAR) and stored within the version directories.

For example, the Open QA group publishes a multitude of components packaged for use with Maven environments, and one such component drives Selenium integration tests. Open QA uses the group ID /org/openqa/selenium to organize its Selenium components. The artifact is identified within the selenium subgroup through the use of the /selenium-java-client-driver directory. Multiple releases of the driver have been provided, such as the 0.9.0 version utilized by this investigation. The 0.9.0 release is represented as a version directory. The version directory contains the driver JAR and a bundled pom.xml file that describes the driver's dependencies (known as transitive dependencies). The path to the artifact JAR file is specified as /org/openqa/selenium/selenium-java-client-driver/0.9.0/ selenium-java-client-driver-0.9.0.jar.

Maven and related components, such as CodeHAUS' Maven plug-in for Eclipse (Casey *et al.*, 2006; Mergere, 2007), manipulate the class path provided to the javac tool during build operations. Maven examines the project's pom.xml file and the pom.xml files for specified dependency artifacts to produce an overall dependency model for the build operation. The

classpath is then adjusted with references to dependencies stored within the build computer's Maven repository. Figure E-5 illustrates the relationship between each component.

Maven is bundled with a multitude of popular artifacts, such as JUnit, but many third party artifacts and updates to Maven's core functionality must be obtained after Maven is installed. As illustrated in Figure E-6, Maven uses the build computer's network connection (often involving but not restricted to the HTTP protocol) to contact remote Maven repositories and acquire missing or updated components. Maven then caches the components on the local Maven repository for subsequent use.

Summary

A standards based approach to configuration management was an integral part of this investigation's production and management of source files, artifacts, and environments. The Subversion version control tool, the Maven build and dependency management tool, and archives based on the ZIP format were used to manage the versions of WSLogA Framework elements so that software development, quality control, and release management could be performed quickly and efficiently. These tools enjoy prominence within the industry and are accessible for researchers or practitioners interested in reproducing or evolving the WSLogA Framework produced during this investigation.

Appendix F

Automation

Intent

Configuration management is concerned with precision—artifacts and their behaviors should be reproducible when the same environment and techniques are implemented (Bar & Fogel, 2003; Berczuk & Appleton, 2003; Casey *et al.*, 2006; Collins-Sussman *et al.*, 2004; Enes, 2007; Estublier *et al.*, 2005; Fowler, 2006; Hatcher & Loughran, 2003; Hevner *et al.*, 2004; Rainsberger & Stirling, 2005). Such reliable reproduction facilitates continued research, development, or the assessment of artifact and theory quality. Process automation is an important tool for ensuring precise reproduction of artifacts and their behaviors from a source and environment base (Berczuk & Appleton, 2003; Fowler, 2006; Rainsberger & Stirling, 2005). The Maven build management and Ant automation tools were used by this investigation to facilitate process automation for the WSLogA Framework's build, test, and packaging requirements.

Ant

Ant is a task automation tool produced by Apache to manage builds for the group's multitude of open source projects (Hatcher & Loughran, 2003). Tasks are declared using XML within a build file. Ant is bundled with tasks for common Java oriented build operations

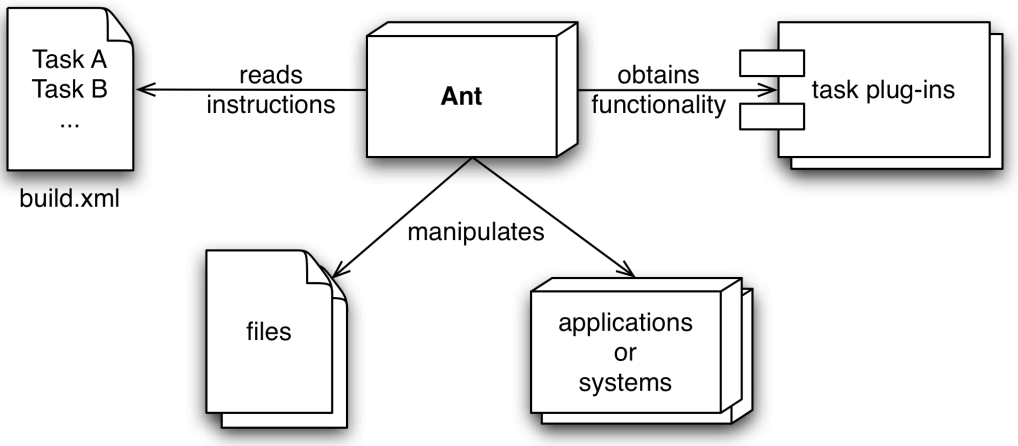


Figure F-1. Ant uses plug-ins to execute tasks manipulating the environment.

(such as the compilation of Java files into class files) but the automation engine can also be extended with Java based components (Pepperdine, 2003).

Ant can be contrasted with earlier project build tools—such as UNIX scripts or the make tool for C based applications—in that the language is declarative versus scripted and tasks are declared in terms of temporal relationships or behaviors (Hatcher & Loughran, 2003). Script oriented languages can also provide task automation, but bugs may arise due to the nature of the syntax and improper logic can be difficult to debug (Bar & Fogel, 2003; Chandra *et al.*, 2003; Hatcher & Loughran, 2003; Telles & Hsieh, 2001). An Ant foundation for Java projects provides the benefit of industry familiarity and thus reduces the time and configuration work necessary for third parties to begin producing artifacts. The Adventure Builder application from Sun Microsystems, which is used to demonstrate aspects of the WSLogA Framework developed as part of this investigation, makes use of Ant for the automated construction and packing of its components. Figure F-1 illustrates the relationship between Ant and build elements.

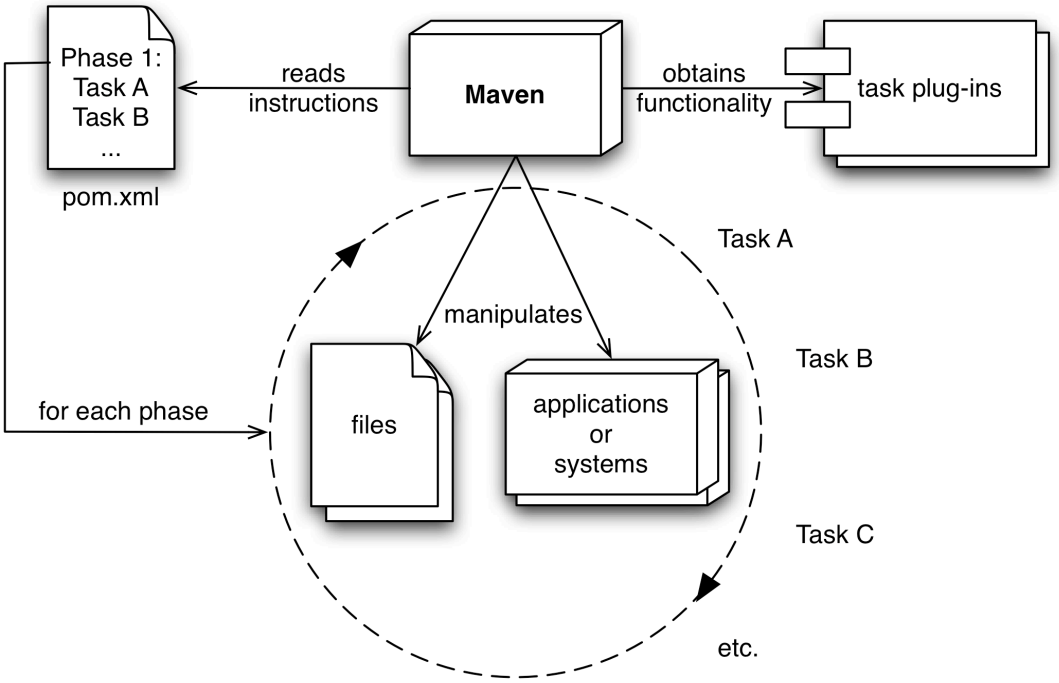


Figure F-2. Maven executes tasks within standard lifecycle phases.

Maven

Maven is a build and configuration management tool produced by Apache to manage the complex build and release processes for the group's multitude of open source projects (Casey *et al.*, 2006; Enes, 2007). Maven is often perceived as an evolution of Ant because Maven utilizes Ant's libraries and plug-ins to provide core automation functionality (Casey *et al.*, 2006). Figure F-2 illustrates Maven's build lifecycle.

Maven improves upon Ant by enforcing conventions for organizing operations, properties, and other configuration concerns (Casey *et al.*, 2006; Zyl, 2006). Ant provides tasks as elementary units—such as compiling Java files—and delegates build organization choices to implementing teams (Hatcher & Loughran, 2003), but Maven defines a series of build lifecycles (such as for JAR oriented projects) with

common phases bearing appropriate functionality that are executed in a progressive order. These phases can be customized according to project needs (Casey *et al.*, 2006), but the commands necessary to execute a lifecycle up to a particular phase always remains the same. As such, teams familiar with Maven can begin immediate reproduction of a Maven based project's artifacts without learning how the project's specific build tasks operate.

The Automated Build

This investigation used an automated build based on Maven's JAR lifecycle (Casey *et al.*, 2006), which is optimized for the compilation of Java source code and the production of a JAR file that can be distributed with applications or Application servers. The Site and Clean lifecycles (Casey *et al.*, 2006) were also used for project maintenance. The lifecycles were customized for the WSLogA Framework's specific needs through the inclusion of tasks for obtaining source code from Subversion, running unit or integration tests, managing testing environment components such as the GlassFish Application server, and generating reports. Builds were not considered successful unless all of the components could be generated and successfully tested. Reports were produced at the end of each iteration phase. Table F-1 describes build phases for the WSLogA Framework's implemented Maven lifecycles and the customized tasks associated with each build phase.

Table F-1. The Maven lifecycle as applied to the WSLogA Framework.

Build Phase	Intent of Phase	Customization
JAR lifecycle phases:		
initialize	Validate that the pom.xml and workspace are properly structured and that the necessary information is available.	• None.

Table F-1. The Maven lifecycle as applied to the WSLogA Framework.

Build Phase	Build Phase	Build Phase
generate-sources	Generate sources for inclusion in the compilation. Often used by code generators.	• None.
process-sources	Process the source code, such as to filter symbols for replacement values.	• None.
generate-resources	Generate resources for inclusion in the compilation. Often used to apply database or Application server settings.	• None.
process-resources	Process the resources, such as to filter symbols for replacement values.	• None.
compile	Compile sources into binary form.	• The Java 1.5 SDK is required for successful compilation. • The Java 1.5 language target is specified to enable annotations, enumerations, and other features utilized by the framework.
process-classes	Process binaries, such as to insert instrumentation information.	• JPOX modifies class files to enable JDO functionality.
generate-test-sources	Similar to generate-sources, but for test components.	• None.
process-test-sources	Similar to process-sources, but for test components.	• None.
generate-test-resources	Similar to generate-resources, but for test components.	• None.
process-test-resources	Similar to process-resources, but for test components.	• None.

Table F-1. The Maven lifecycle as applied to the WSLogA Framework.

Build Phase	Build Phase	Build Phase
test-compile	Compile test sources into binary form.	• None.
test	Execute unit tests.	• Integration tests controlled by JUnit are excluded from execution.
package	Create packages for artifacts, such as the JAR file.	• None.
pre-integration-test	Prepare environment and components for integration tests.	• The Selenium Remote Control server required to run Selenium based integration tests is started. • The packaged framework is copied to the GlassFish Application server. • The GlassFish Application server and database are started.
integration-test	Execute integration tests.	• Selenium based integration tests are executed.
verify	Otherwise verify the organization, structure, and suitability of artifacts.	• None.
install	Install the artifacts into the local Maven repository for use in other processes or components.	• None.
deploy	Deploys the artifacts to a remote Maven repository.	• None.
Clean lifecycle phases:		
pre-clean	Prepare the workspace for cleaning.	• None.
clean	Clean the workspace of directories and files generated during Maven builds.	• Remove Cobertura files generated in locations not recognized by Maven.

Table F-1. The Maven lifecycle as applied to the WSLogA Framework.

Build Phase	Build Phase	Build Phase
post-clean	Configure the cleaned workspace for new work.	• None.
Site lifecycle phases:		
pre-site	Prepare the workspace for a build with the intent of generating reports.	• None.
site	Perform a build and generate reports.	• Generate JavaDoc documentation with UML class diagrams using yDoc. • Generate Cobertura code coverage report for unit tests. • Generate FindBugs report to document implementation patterns known to facilitate critical bugs. • Generate unit test success report. • Generate HTML source code view with hyperlink references among components. • Generate information site with project, participant, and dependency pages.
post-site	Manage reports, such as to copy them to a server for public review.	• None.

Table F-2. Ancillary automation scripts.

Script	Purpose	Targets
GlassFish.xml	Manages the GlassFish and bundled Derby database components. The Application server environment can be populated with Adventure Builder components or resources, and the environment can be archived or unarchived to facilitate testing.	• GlassFish.archive *Archives the GlassFish and bundled Derby installation in ZIP format.* • GlassFish.unarchive *Unarchives a ZIP archive and replaces the existing GlassFish installation with the archive's contents.* • GlassFish.installWsLogAFwk *Installs the wslogafwk.jar file in the lib folder of the test application configured within GlassFish for this investigation.* • GlassFish.installAdventureBuilder *Installs the Adventure Builder components and initializes the Derby Database in the test domain created within GlassFish for this investigation.*
svn.xml	Manages the configuration of a Subversion repository that may be used to hold the source files made available by this investigation.	• svn.create.repository *Establishes a new repository for the project and imports source files.*
maven.xml	Installs non-standard dependencies into the local Maven repository.	• maven.repository.archive *Archives the local Maven repository in ZIP format.* • maven.repository.unarchive *Unarchives a ZIP archive of a Maven repository and overlays the contents onto the local Maven repository.*

Automated Environment Support

Ant scripts and associated properties files were produced to facilitate the configuration and subsequent management of the development and test environments. The GlassFish Application server and bundled Derby database can be prepared with updated Adventure Builder components and database information, and also be archived into ZIP format for later use if a configuration proves to be useful for testing or demonstrations. The Subversion repository configuration used for this investigation can be reproduced with an Ant script, and sources made available from this investigation can be imported into the new repository for continued research, development or evaluation. WSLogA Framework dependencies required for development, testing, or evaluation but that are not provided by one of the significant Maven plug-in mirror sites can be installed into a local Maven repository through the use of a provided script. Table F-2 describes these ancillary scripts.

Summary

Task automation permits repetitive tasks to be executed consistently for precise reproductions of results within the same environment. This investigation used automation to manage the consistent generation of artifacts from sources, measurement of source or artifact fitness (*e.g.*, with unit tests), and the preparation of documentation related to the sources, artifacts, or tests. Additionally, Ant scripts were prepared to manage the Maven, Subversion, and GlassFish environments to facilitate project configuration, testing, or distribution. Maven and Ant enjoy prominence within the industry and are accessible for researchers or practitioners interested in reproducing or evolving the framework produced during this investigation.

Appendix G

Reports and Documentation

Intent

Reports were generated throughout the course of this investigation to describe the quality or functionality of sources and artifacts. Report generation was driven by Maven (Appendix G) with the use of third party plug-ins integrating tools such as Sun Microsystems' JavaDoc. These report sets are incorporated into the project archive prepared as the result of this investigation and made available with this dissertation report. This section describes the types of reports and documentation produced and how the information provided assisted investigation efforts, or how it may assist third parties.

Reports Facilitating Third Party Adoption or Development

Frameworks are complex in that they represent a generalized solution to a problem domain. Application logic is introduced to a framework through extension components implementing hotspots that the framework manages through inversion of control, dependency injection, and other strategies (Arthur & Azadegan, 2005; D'Souza & Wills, 1998; Fayad & Schmidt, 1997; Fowler, 2004; Richter, 1999; Schmidt *et al.*, 2004). Documentation facilitates framework adoption (Kotula, 1998) by communicating key concepts regarding the sources and artifacts (Forward & Lethbridge, 2002) and by reducing the time required for individuals

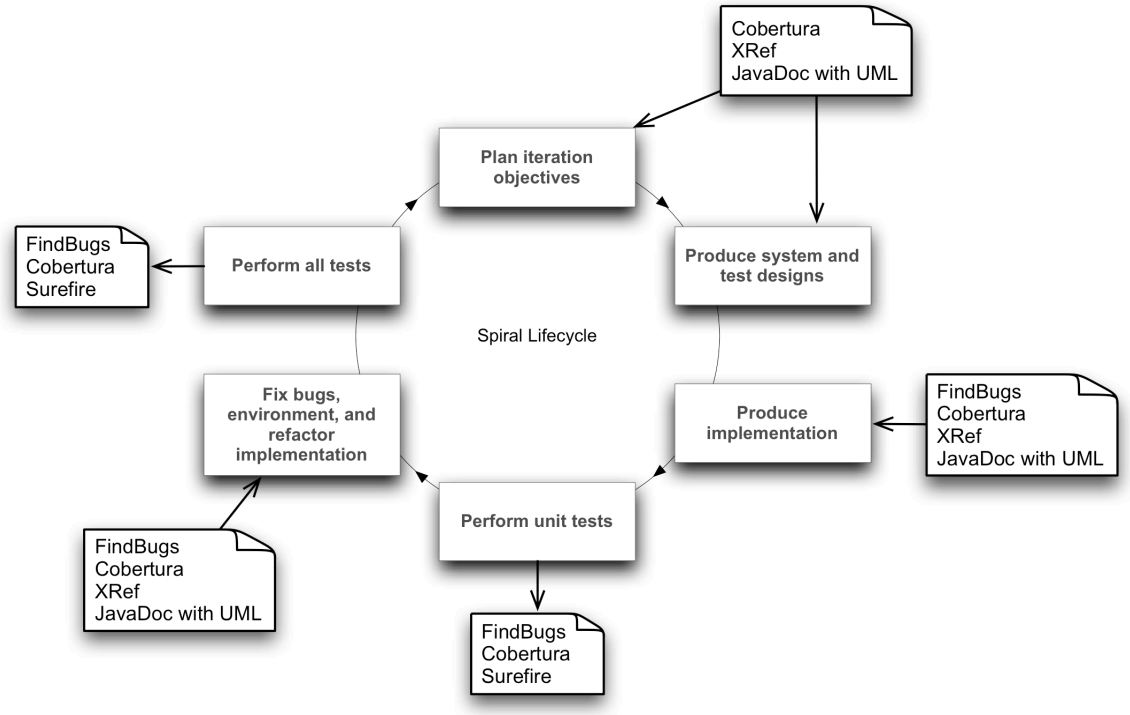


Figure G-1. Reports applied to this investigation process.

to learn about how the framework and third party extensions interact to provide the desired behavior (Sherif & Vinze, 1999).

Reports Facilitating the Investigation Process

The Spiral Lifecycle adapted for use in this investigation ensured the iterative, holistic consideration of artifacts from the perspective of design, implementation, and quality control (Chapter 3; Appendix A). Iterations were planned according to completed tasks, knowledge gained from experiments, issues identified by tests, and artifact requirements identified but not yet designed or implemented.

Reports from the automated build (Appendix F) were consulted throughout iterations to ensure development efforts built upon existing work and addressed concerns preventing

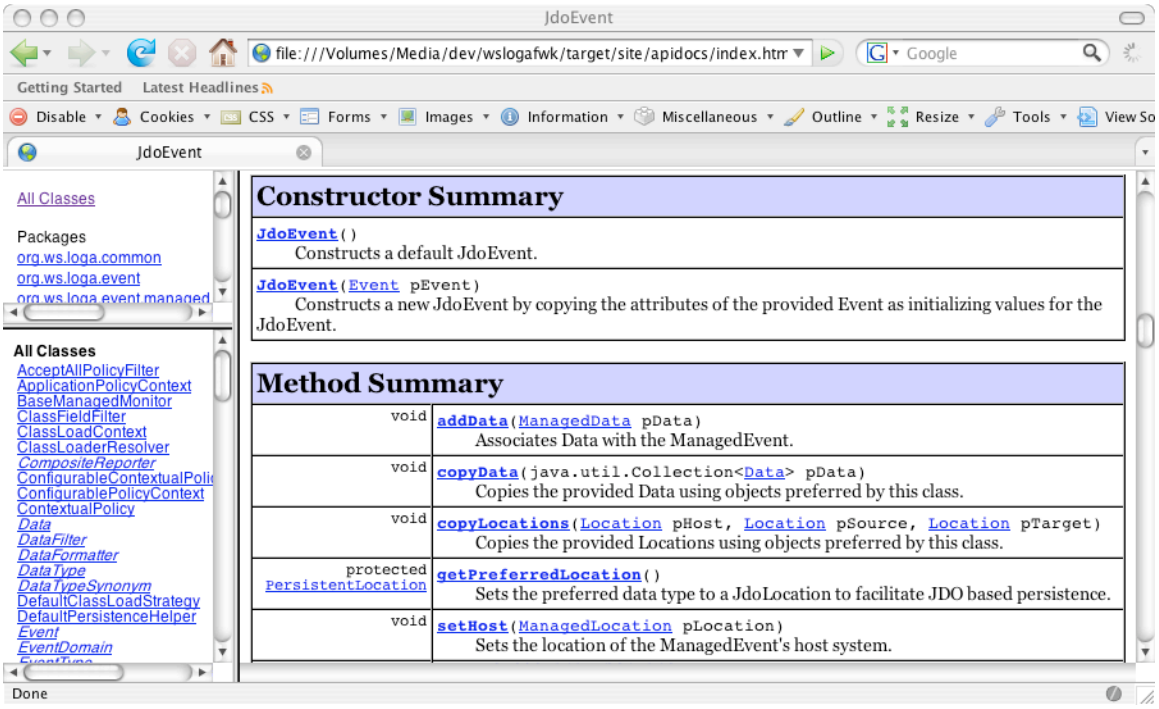


Figure G-2. JavaDoc reports provide textual information regarding components.

dependent tasks, as Figure G-1 illustrates. Quality control reports also facilitated project governance (Schwalbe, 2006; Wysocki *et al.*, 2000).

Documentation Oriented Reports

Three types of reports document the organization and functionality of the sources and artifacts produced by this investigation: textual component descriptions, graphical component descriptions, and source code cross-references. Each report provides a different perspective of the WSLogA Framework's components, methods, and strategies.

Textual component descriptions are provided using Web pages produced using Sun Microsystems' JavaDoc utility (Arnold *et al.*, 2005; Kramer). JavaDoc reports document APIs in terms of their packages, components, attributes, and methods. Software engineers may use

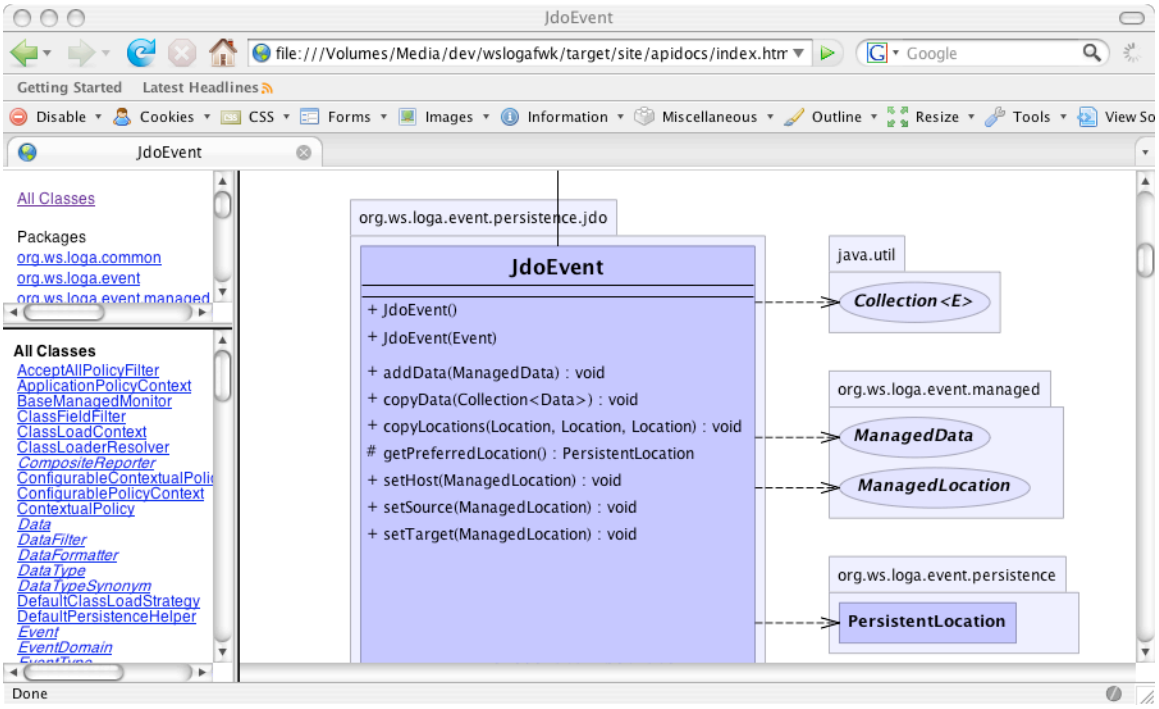
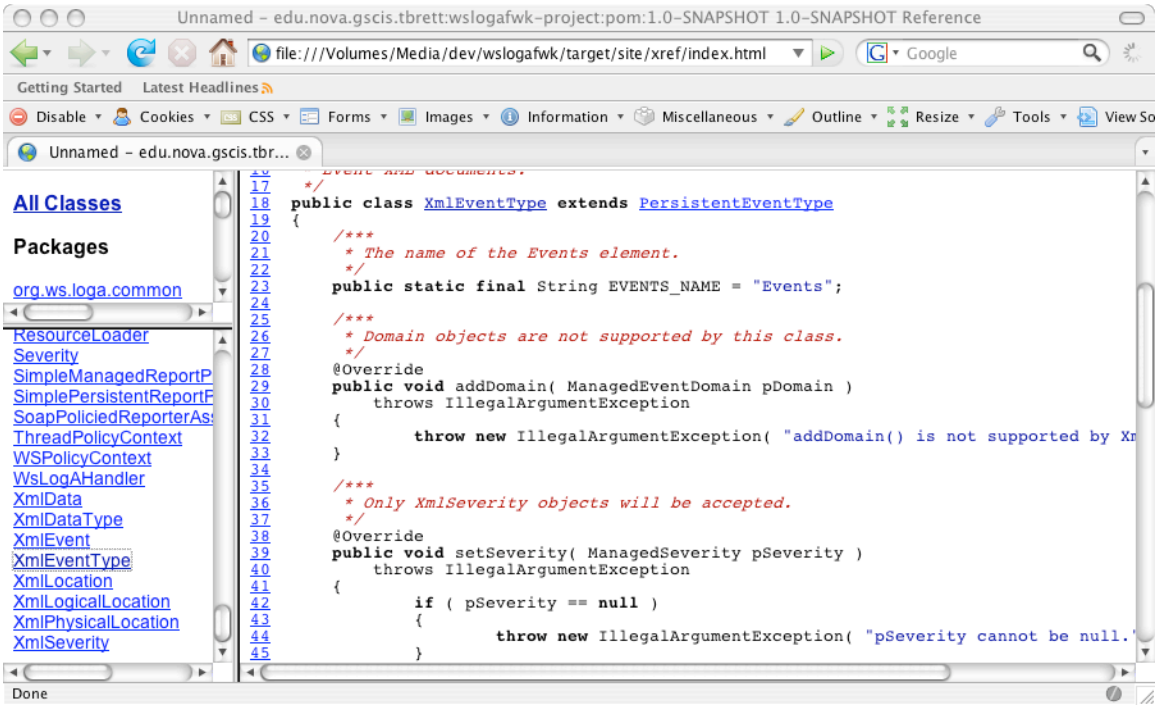


Figure G-3. yDoc UML diagrams are embedded in JavaDoc Web pages.

JavaDoc reports to understand the WSLogA Framework's capabilities in terms of their structure and behavior. Figure G-2 illustrates a partial JavaDoc Web page with constructor and method summaries.

The yDoc plug-in[1] was used to augment the JavaDoc report with embedded UML class diagrams illustrating either components within a package or a specific component and its object dependencies. yDoc integration with the JavaDoc utility permits these class diagrams to accurately reflect the available components and their significant structural relationships each time the JavaDoc report is regenerated. Figure G-3 illustrates a partial class diagram for the same component described in Figure G-2.

[1] yDoc is a product of yWorks, a company that provides documentation tools for a variety of development languages and platforms. yWorks generously donated a commercial version of yDoc for use in this investigation. More information regarding yWorks is available online at [http://www.yworks.com].




```
 */
public class XmlEventType extends PersistentEventType
{
    /***
     * The name of the Events element.
     */
    public static final String EVENTS_NAME = "Events";

    /***
     * Domain objects are not supported by this class.
     */
    @Override
    public void addDomain( ManagedEventDomain pDomain )
        throws IllegalArgumentException
    {
            throw new IllegalArgumentException( "addDomain() is not supported by Xm
    }

    /***
     * Only XmlSeverity objects will be accepted.
     */
    @Override
    public void setSeverity( ManagedSeverity pSeverity )
        throws IllegalArgumentException
    {
            if ( pSeverity == null )
            {
                throw new IllegalArgumentException( "pSeverity cannot be null."
            }
```

Figure G-4. XRef reports facilitate the quick exploration of source code.

Source code can be difficult to navigate when voluminous and an IDE supporting visual browsing, such as Eclipse, is not available. Software engineers may also wish to explore a framework's implementation without obtaining source code—such as when considering WSLogA Framework revisions. The XRef report tool generates HTML based documentation containing the source code with hyperlinks referencing related components, as illustrated in Figure G-4. For example, XRef will provide a hyperlink to an interface that a class implements. In this manner, the hyperlinks are located within contextually relevant locations. Packages and components are also presented using indexes styled after the default JavaDoc template distributed by Sun Microsystems, which makes report navigation straightforward for experienced Java developers.

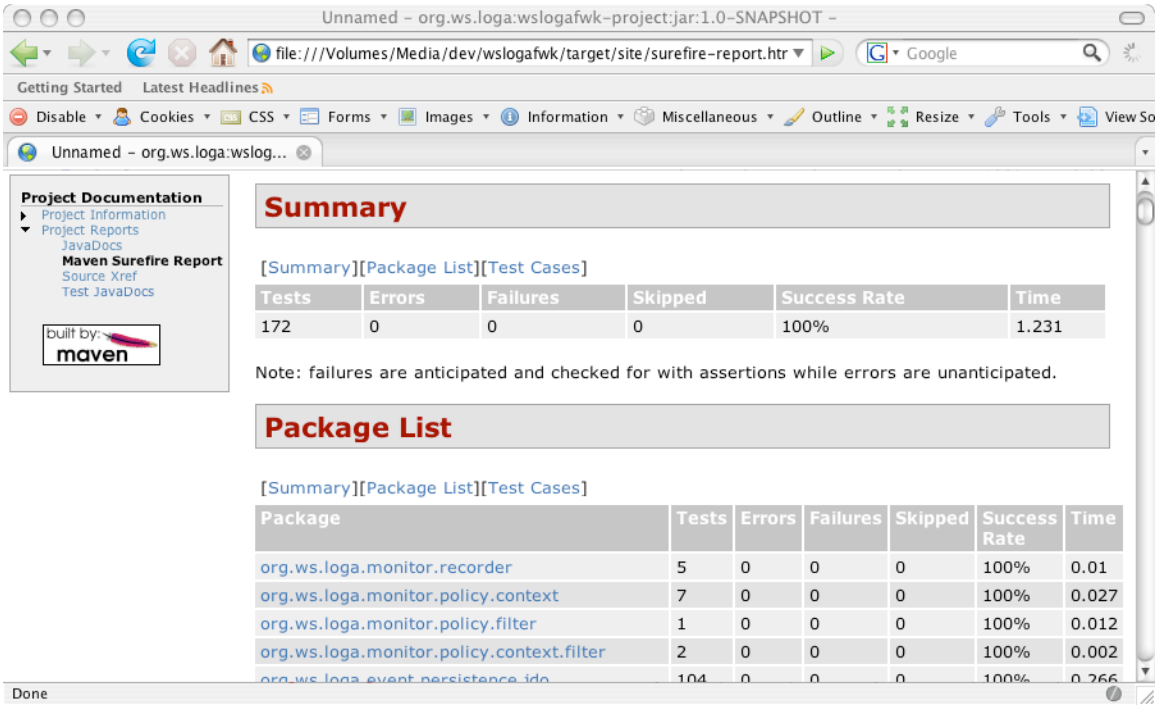


Figure G-5. Unit test success summaries and statistics are provided in Surefire reports.

Quality Control Oriented Reports

Three types of reports document the quality of sources and artifacts produced by this investigation: unit test results, unit test code coverage, and source code segments matching patterns known to permit faulty behavior. Each report provides a different perspective of the WSLogA Framework's adherence to planned functionality.

Unit tests ensure that component implementations honor intended behavior within the context of inputs and associated components, such as injected dependencies. For example, a method that adds two numbers and then returns the sum should generate the correct sum for a known set of numbers; a unit test could call the method and provide predefined numbers and then assert that the obtained sum is appropriate.

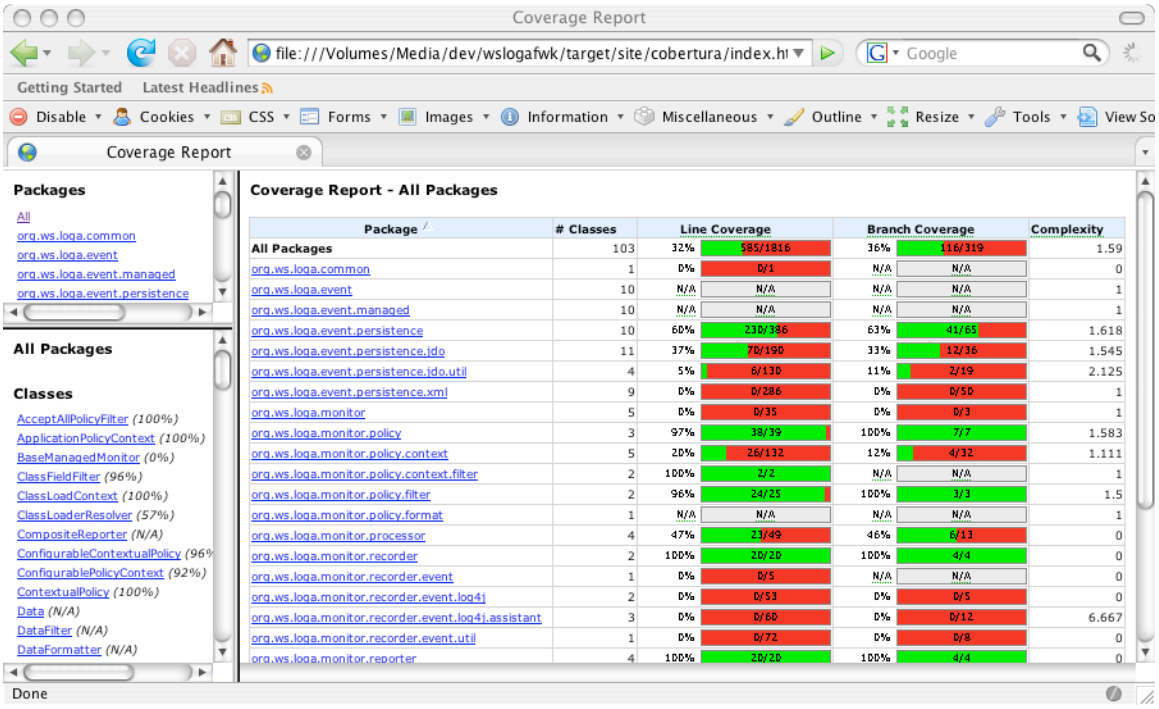


Figure G-6. Cobertura reports illustrate source code unit test coverage.

The Surefire report generator provided with the Maven automated build tool (Appendix F) generates an HTML based report, illustrated in Figure G-5, that displays test result summaries and associated statistics. Test logs and exception information can be obtained with the use of hyperlinks on the summary page. Surefire reports were archived throughout this investigation to provide benchmarks by which the results of iterative work for experiments or maintenance could be compared.

Test coverage involves source code or artifact analysis to identify structures or states that may permit unintended behavior, as well as to identify which parts of a system are not exercised by tests (Ayewah *et al.*, 2007; Berner *et al.*, 2007; Huang *et al.*, 2007). Source code coverage by the unit tests was monitored with the use of Cobertura reports. Cobertura

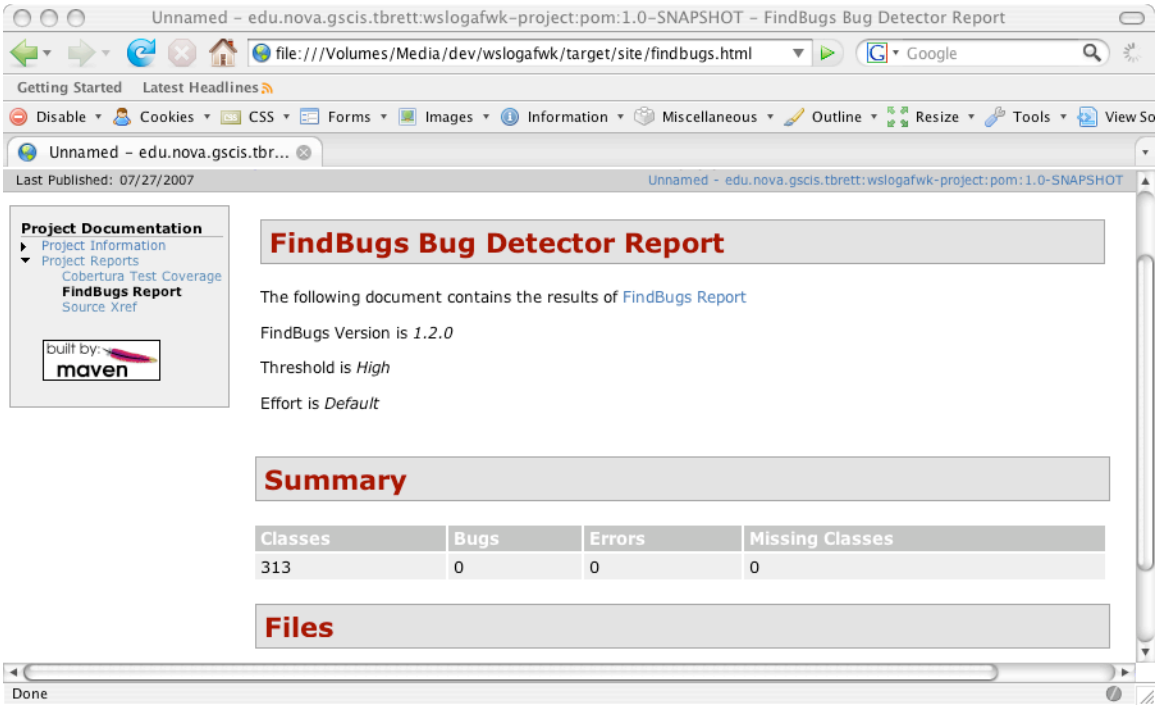


Figure G-7. FindBugs report showing categories in which bugs would appear.

(Harold, 2005; Yang *et al.*, 2006) instruments class files prior to testing to insert monitoring instructions. Cobertura generates an HTML based report after unit test execution by Surefire that can be analyzed to determine the statements that were executed during the tests and the degree by which conditional branches have been pursued. Figure G-6 illustrates a Cobertura report for a component under development. Cobertura does not determine if appropriate data endpoints were used to activate runtime logic paths, but test scenarios not envisioned during design may be identified through the exposure of untested regions in the source code (Berner *et al.*, 2007).

Bug pattern analysis is performed on implemented components with the use of FindBugs. FindBugs uses plug-ins to inspect Java source code for patterns known to permit

unintended behavior (Ayewah *et al.*, 2007; Foster *et al.*, 2007). For example, the failure to properly initialize a variable could result in a null pointer exception. The FindBugs report complements the Cobertura test coverage report by highlighting the types of risks present within the source code, and unit tests can be developed or refined based on report information to exercise potentially unsafe code.

Summary

The generated reports provided an important foundation for design, implementation, and quality control efforts. Code coverage provided by Cobertura identified source code statements not exercised by unit tests. JUnit test results reported by Surefire facilitated comparison of WSLogA Framework component behaviors over multiple iterations. FindBugs reports identified potentially unsafe implementations matching bug patterns that can be difficult for initial unit tests to discover. JavaDoc reports augmented with embedded UML class diagrams communicate the APIs available for use by third parties, and an HTML representation of the source code facilitates navigation of the component implementations. Practitioners integrating the WSLogA Framework for use in their systems can use these reports to learn about the WSLogA Framework's suitability and maturity for production systems. Researchers can use these reports to verify the WSLogA Framework's robustness and identify sections for further study.

Appendix H

Use Case Descriptions

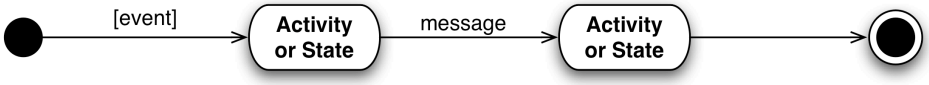
Overview

Use cases in UML form (Richter, 1999; Stevens & Pooley, 2000) were prepared as part of the design process for this investigation to identify components as well as guide the WSLogA Framework's artifact and test implementations. Use cases are a type of functional specification that provides a business or workflow perspective into the system's domain model (Richter, 1999). Examples of information that can be obtained from a use case or its description include workflows, system component roles, and the relationships between roles and the workflows in which they participate (Richter, 1999; Schach, 2002; Stevens & Pooley, 2000; Whitten *et al.*, 2001). This section describes the use cases presented in Chapter 4 using activity diagrams (Richter, 1999; Stevens & Pooley, 2000) or comments for scenarios as necessary for clarification.

Use cases were prepared according to the domain model specified or implied by the Proposal's objectives: the establishment of information capture, event management, response, and presentation systems within the WSLogA Framework. These subsystems define the functionality described by the WSLogA (Cruz *et al.*, 2003; Cruz *et al.*, 2004), but also specify additional functionality to support the WSLogA Framework's role in facilitating Web service analysis and environment manipulation. Implementations were required to provide the

Use Case(s):	Principal use case names
Comments:	A synopsis of the use case.
Trigger:	The event or message that starts the flow.
Pre-State:	Expectations of state prior to use case flow.
Post-State:	Expectations of state after the use case flow.

Normal Flow and States:



Alternate Flows and States:

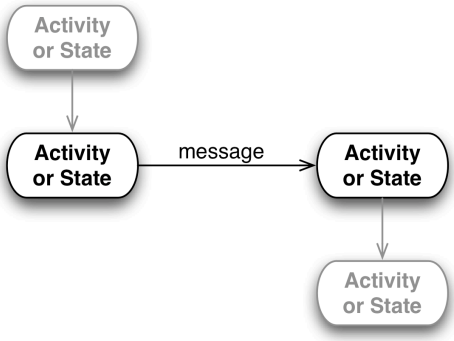


Figure H-1. Use case descriptions include an activity diagram and clarifying comments.

specified functionality but component and package boundaries were structured as appropriate for an object-oriented architecture. For example, the information capture domain defines policy interaction, but policies were implemented in a manner that permits their use throughout all WSLogA Framework subsystems or third party extensions.

The software industry has yet to adopt a universal functional specification structure, but in general functional specifications address concerns such as use case triggers, pre-conditions, post-conditions, and significant activities (Richter, 1999; Schach, 2002; Stevens & Pooley, 2000; Whitten *et al.*, 2001). This appendix describes those concerns and, where

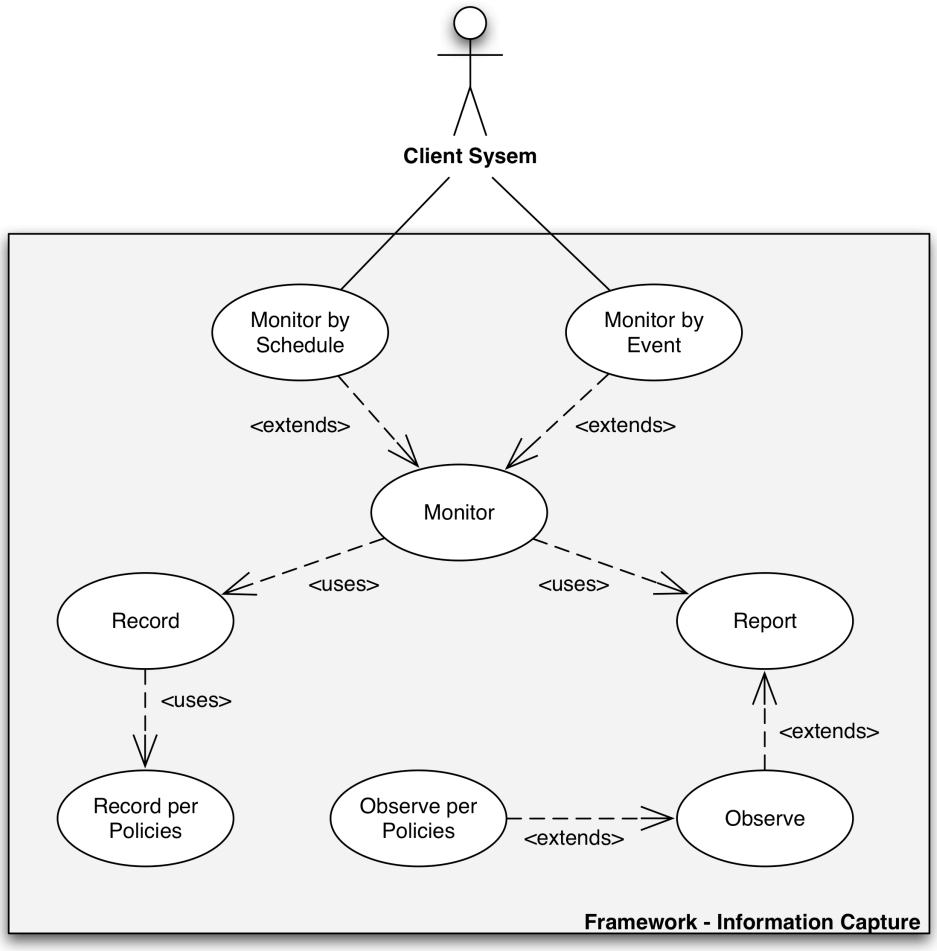


Figure H-2. Principal information capture use cases.

appropriate, provides examples of the desired functionality using references for similar technologies—such as the allusion to Quartz in select Response Engine use cases (Quartz is a thread scheduling library). Figure H-1 illustrates the use case description format used for this report, which is permitted to break across pages. Each remaining section discusses the use case domain, introduces the use cases, and provides activity diagrams and their comments to clarify the expected behavior and state of each use case.

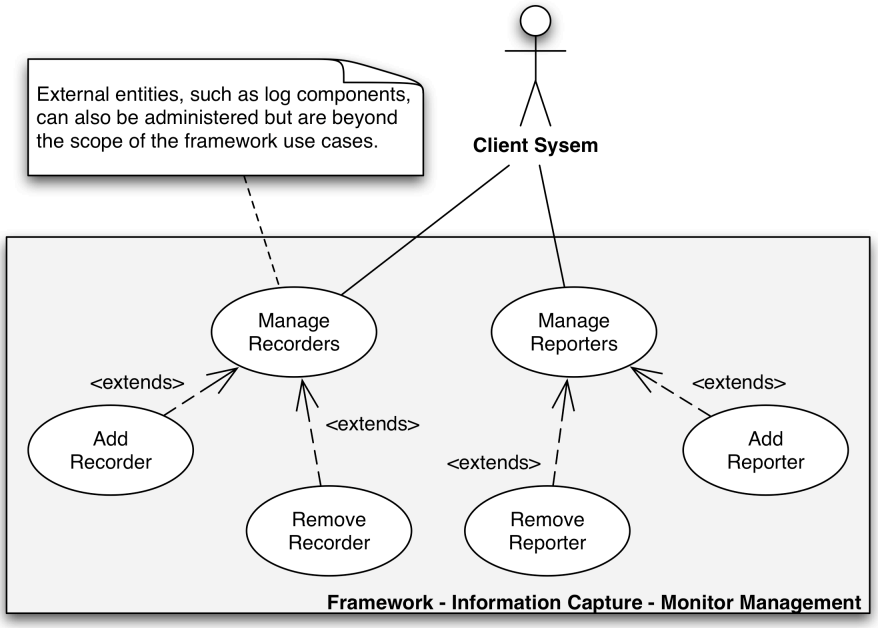


Figure H-3. Monitor management use cases.

Information Capture Use Cases

The information capture subsystem satisfies the Proposal's objectives for establishing data capture SOAP handlers and a data routing log management system. Unlike standard information capture frameworks, such as Apache's Log4J and Sun Microsystems' Logging API, information normalization and filtering through configurable policy expression is a core feature of the architecture. Applications utilizing the information capture subsystem may be ported to host environments among diverse legal jurisdictions or cultural regions with respect to information management policies (*e.g.,* privacy) because the policy management architecture reduces or eliminates the need for significant implementation changes. Figure H-2 illustrates the principal use cases address reporting and recording within a coordinated context established by monitors.

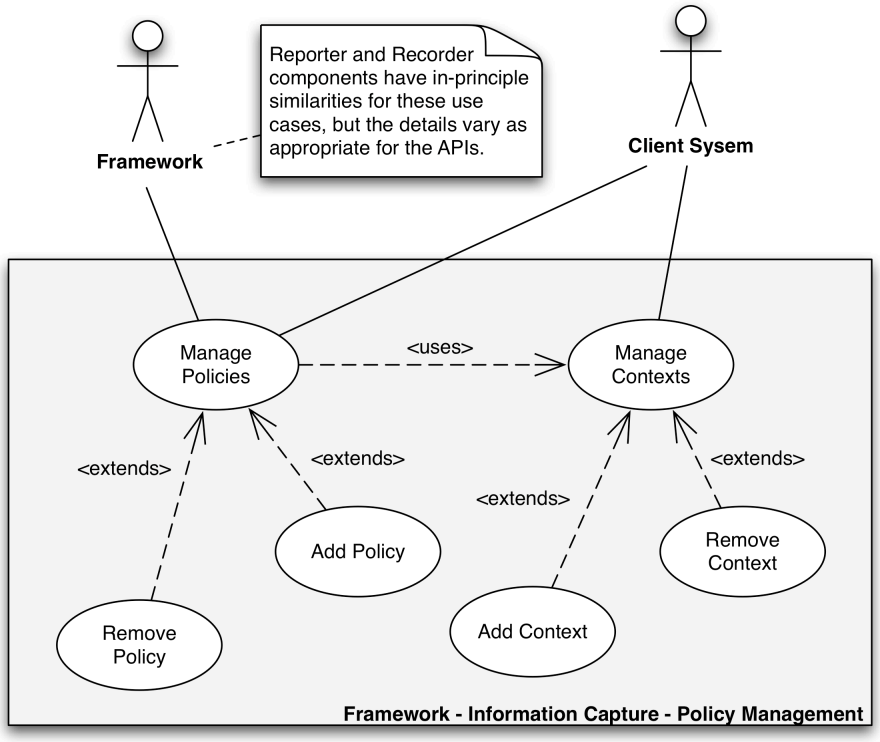


Figure H-4. Policy management use cases.

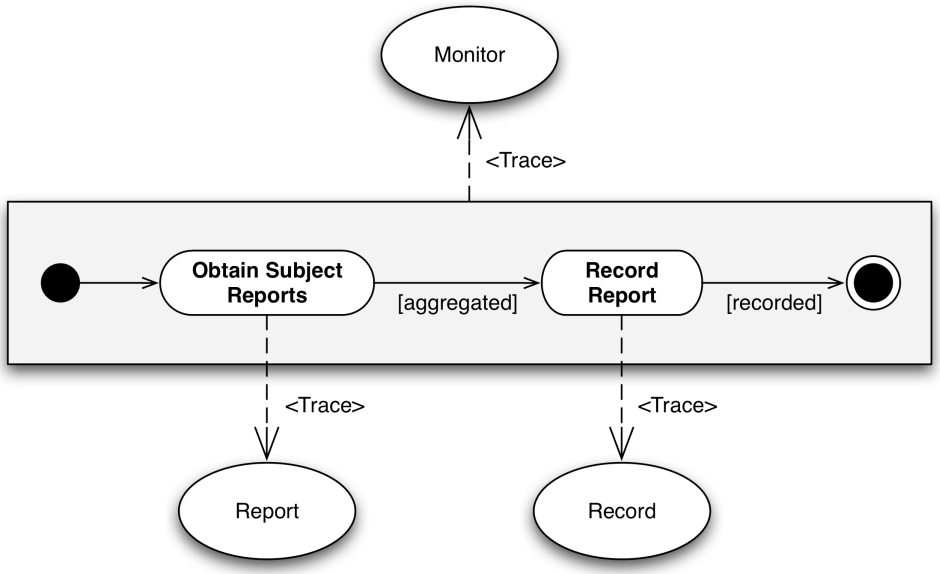
Component management is an integral part of framework configuration, and use cases were prepared to provide examples for the types of management expected for recorders, reporters, and the myriad of other principal components addressed by the WSLogA Framework. Figure H-3 illustrates the use cases prepared for information capture component management, and these use cases are referenced throughout this appendix as reference models for similar functionality.

Policy management and expression was envisioned initially for information capture as the mechanism by which sensitive information could be masked or omitted before it was committed to a permanent record. As such, the design for policy workflows is part of the information capture use case set despite their implementation as a distinct package that may

be utilized by any subsystem or third party extension. Figure H-4 illustrates the use cases addressing policy and policy context management.

Use Case(s):	Monitor
Comments:	The monitor use case represents the process by which inspectors or reporters produce information and provide such to recorders in the form of report objects. Reports may be aggregated by the coordinating components prior to submission to recorders.
Trigger:	A request to monitor, such as a SOAP handler event, begins the monitoring process.
Pre-State:	Reporters are provided with Subjects or other resources.
Post-State:	Recorders have provided registered consumers with the processed report information.

Normal Flow and States:



Use Case(s):	Monitor by Schedule, Monitor by Event
Comments:	The Monitor by Schedule and Monitor by Event use cases reflect the need to permit, respectively, persistent and managed monitoring. Event based monitors should be provided with a specific subject at the time of event, whereas scheduled monitors should only be considered with specifically assigned reporters.
Trigger:	A scheduled manager, such as a thread or other mechanism, will trigger

Monitor by Schedule. A specific event or request will trigger Monitor by Event.

Pre-State: Monitor by Event should be provided with guidance regarding the report to produce or subject to observe.

Post-State: Identical to the Monitor use case.

Normal Flow and States:

Please reference Monitor.

Use Case(s):	Monitor, Record

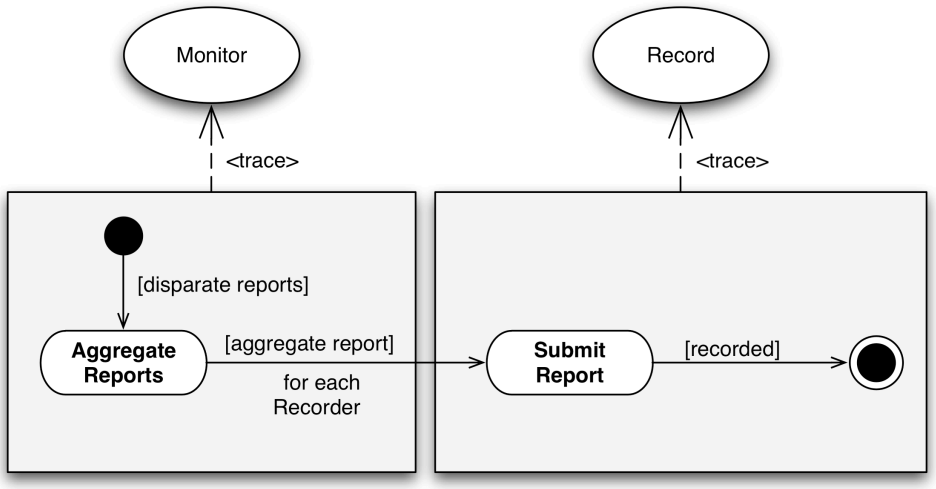
Comments: The Record use case is used by Monitor to persist or otherwise provide information to registered consumers. The report information provided by Monitor is aggregated into a single report. External consumers, such as a Log4J Appender, may apply their own filter rules.

Trigger: The submission of a report by Monitor or comparable entity.

Pre-State: The information is provided in an acceptable report format.

Post-State: The report has been provided to each registered consumer for recoding. Persistence as part of the recording process, if any, is a function of the consumer.

Normal Flow and States:



Use Case(s):	Monitor, Record per Policies

Comments: The Record per Policies use case is used by Monitor to persist or otherwise provide information to registered consumers. The report information provided by Monitor is aggregated into a single report, but consumers can filter and discard report contents according to policies established by the framework mechanisms or

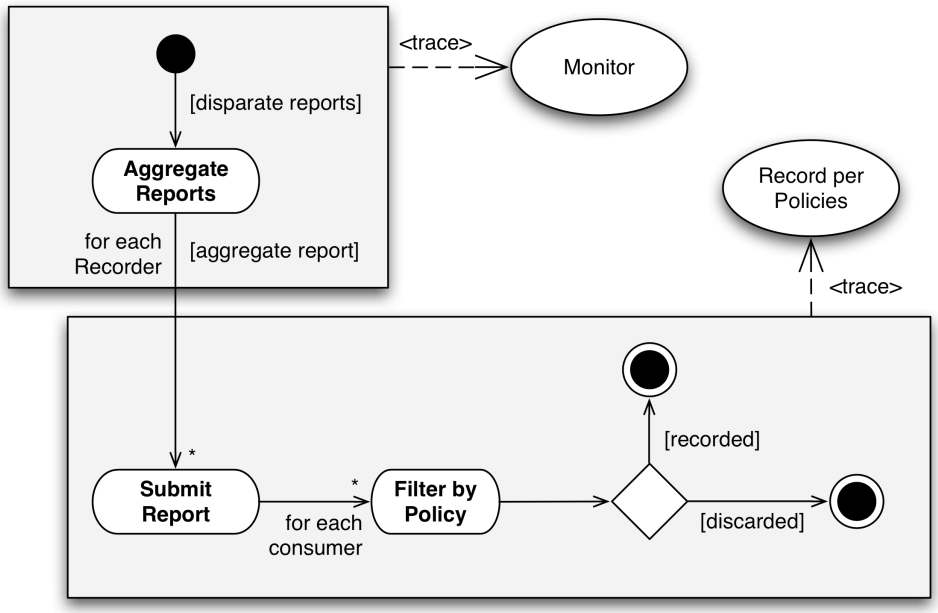
external configures (*e.g.*, Log4J).

Trigger: The submission of a report by Monitor or comparable entity.

Pre-State: The information is provided in an acceptable report format.

Post-State: The report has been provided to each registered consumer for recoding. Persistence as part of the recording process, if any, is a function of the consumer.

Normal Flow and States:



Use Case(s): Report

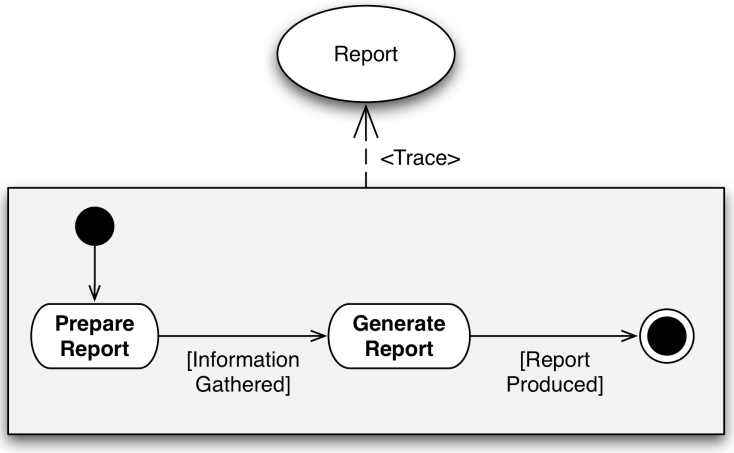
Comments: The Report use case is used by Monitor to generate a report regarding some topic of concern. Report types may vary in structure to accommodate subject or context configurations, but such structural differences must be clearly communicated to the consumer (*e.g.*, using an XML schema reference or Java class type). Empty reports are permissible and may take the form of report shells (*e.g.*, an XML wrapper) or object references (*e.g.*, null).

Trigger: The request for a report by Monitor or comparable entity.

Pre-State: None.

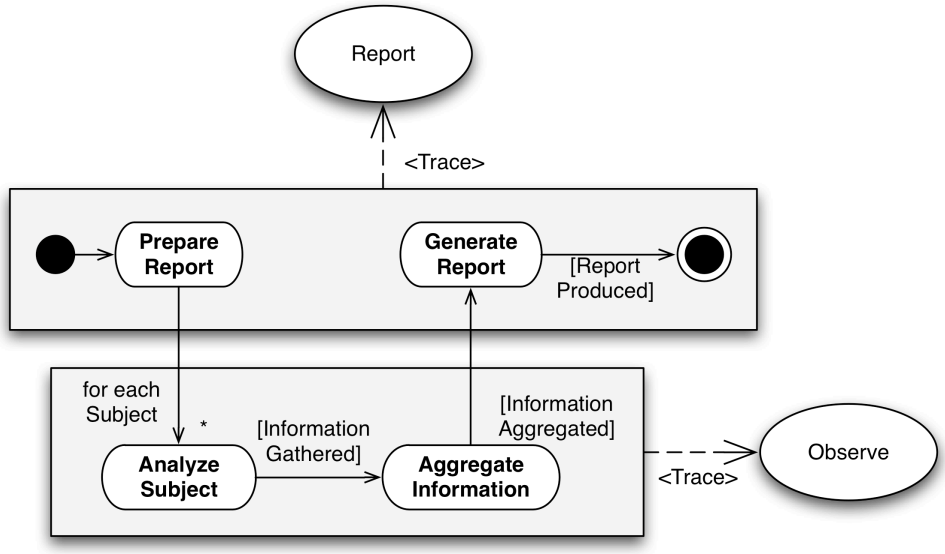
Post-State: A report of an acceptable structure and content has been generated and provided to the requesting entity or an appropriate proxy.

Normal Flow and States:



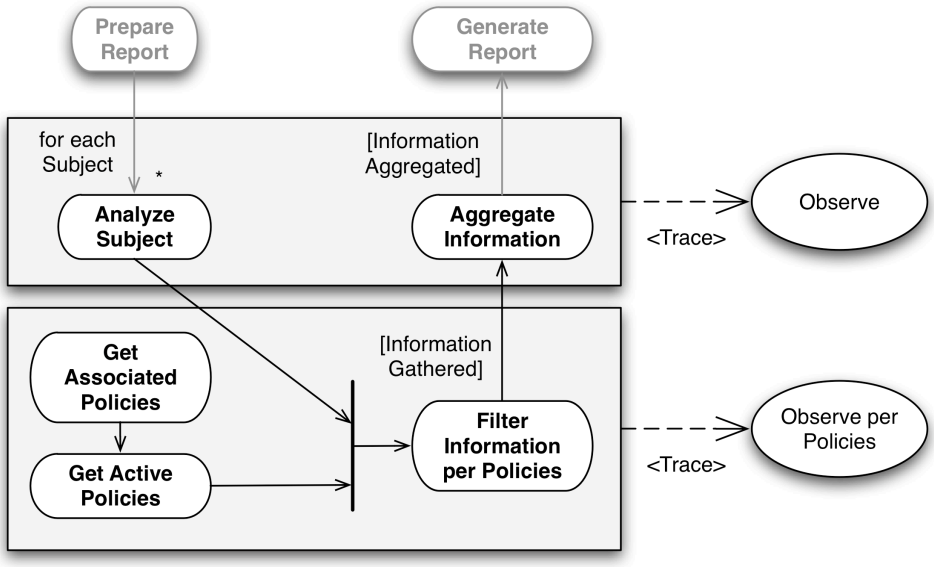
Use Case(s):	Report, Observe
Comments:	The Observe use case is used by Monitor to generate a report regarding a specific subject or sets of related subjects. The reporting components should be specialized for those subjects to enable more detailed reporting than what a generic reporter might provide. Report types may vary in structure to accommodate subject or context configurations, but such structural differences must be clearly communicated to the consumer (*e.g.*, using an XML schema reference or Java class type). Empty reports are permissible and may take the form of report shells (*e.g.*, XML wrapper) or object references (*e.g.*, null).
Trigger:	The request for a report by Monitor or comparable entity.
Pre-State:	The observing component must be primed with one or more subjects for a non-empty report to be produced.
Post-State:	A report of an acceptable structure and content has been generated and provided to the requesting entity or an appropriate proxy.

Normal Flow and States:



Use Case(s):	Observe, Observe per Policies
Comments:	The Observe per Policies use case is used by Observe to generate a report regarding a specific or a set of subjects within the constraints of active associated policies.
Trigger:	The request for a report by Monitor or comparable entity.
Pre-State:	The observing component must be primed with one or more subjects for a non-empty report to be produced.
Post-State:	A report of an acceptable structure and content has been generated and provided to the requesting entity or an appropriate proxy.

Normal Flow and States:



Use Case (s):	Add Recorder

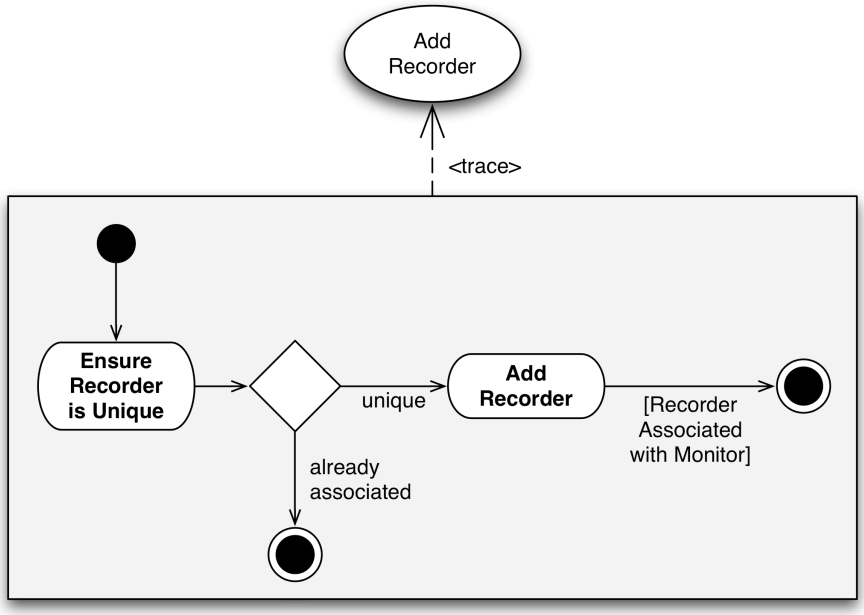
Comments: Monitors use the Add Recorder use case to associate a recorder. The recorder must be unique within the association to prevent duplicate record requests from being sent to the same recorder.

Trigger: A monitor attempts to associate a recorder.

Pre-State: None.

Post-State: A single instance of the recorder is associated with the monitor.

Normal Flow and States:



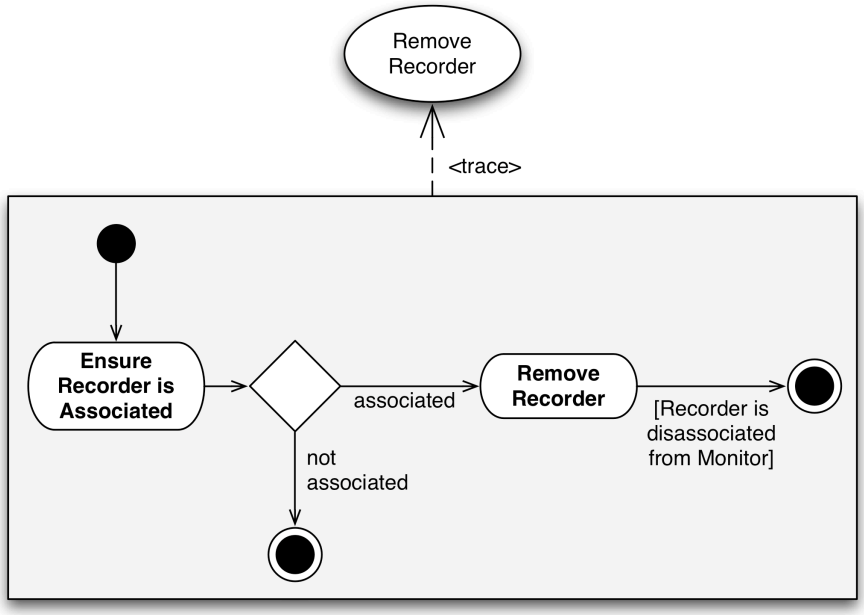
Use Case(s):	Add / Remove Reporter
Comments:	The Add / Remove Reporter use cases are identical in flow to their respective Add / Remove Recorder counterparts.
Trigger:	Similar to Add / Remove Recorder, but with reporter objects.
Pre-State:	None.
Post-State:	Similar to Add / Remove Recorder, but with reporter objects.

Normal Flow and States:

Please reference Add / Remove Recorder.

Use Case(s):	Remove Recorder
Comments:	Monitors use the Remove Recorder use case to disassociate a recorder.
Trigger:	A monitor attempts to disassociate a recorder.
Pre-State:	None.
Post-State:	The recorder is not associated with the monitor.

Normal Flow and States:



Use Case(s):	Add / Remove Policy
Comments:	The Add / Remove Policy use cases are identical in flow to their respective Add / Remove Recorder counterparts, except that the entity being modified is a recorder, a reporter, or other entity that is policy aware. For example, a policy aware observer may be modified such that it is associated with a policy, and that policy must be within a unique relationship to the observer.
Trigger:	Similar to Add / Remove Recorder, but with policy objects.
Pre-State:	None.
Post-State:	Similar to Add / Remove Recorder, but with policy objects.

Normal Flow and States:

Please reference Add / Remove Recorder.

Use Case(s):	Add / Remove Context
Comments:	The Add / Remove Context use cases are identical in flow to their respective Add / Remove Recorder counterparts, except that the entity being modified is a policy that is context aware.
Trigger:	Similar to Add / Remove Recorder, but with context objects.

Pre-State: None.

Post-State: Similar to Add / Remove Recorder, but with context objects.

Normal Flow and States:

Please reference Add / Remove Recorder.

Event Management Use Case Descriptions

The event management subsystem was defined to satisfy the Proposal's objectives for supporting event persistence and information transfer among other subsystems or third

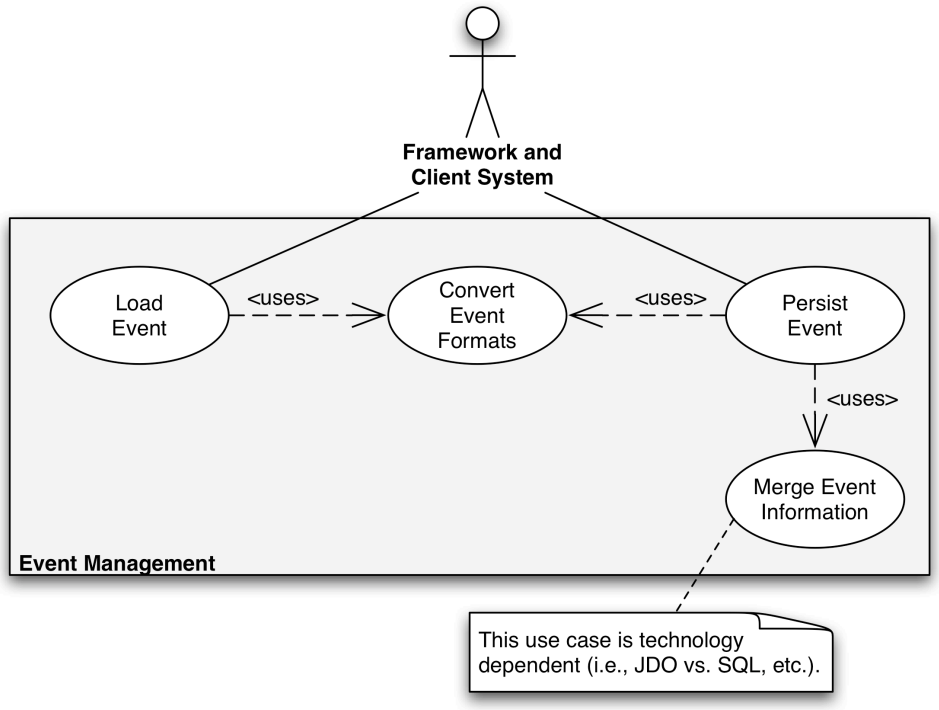


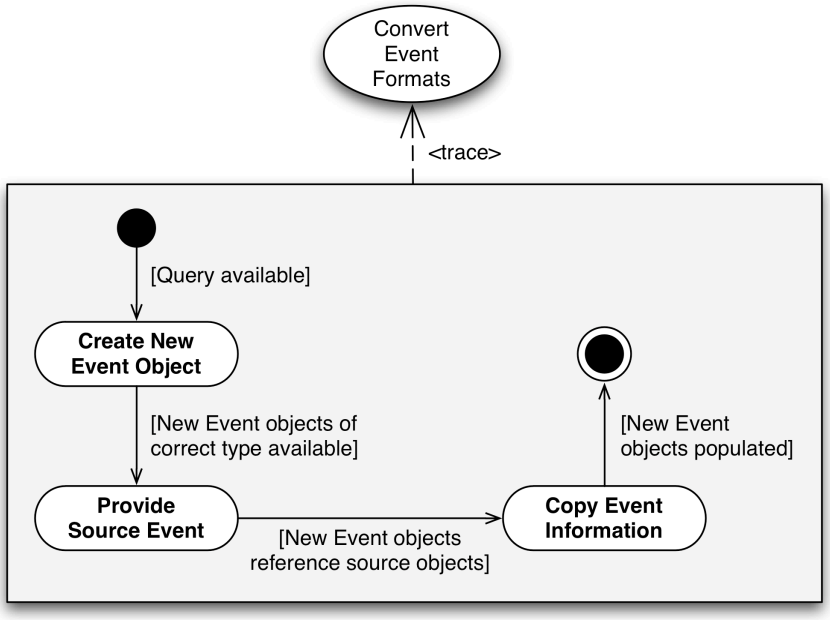
Figure H-5. Event management use cases.

party extensions. The event management subsystem establishes the mechanisms by which information may be organized within a data store, persisted by the information capture subsystems, and retrieved by processing components.

Structures are envisioned for the transfer of event information, such as from reporter to recorder, and the storage of event information, such as a database schema or Java data access object. Figure H-5 illustrates the event management use cases.

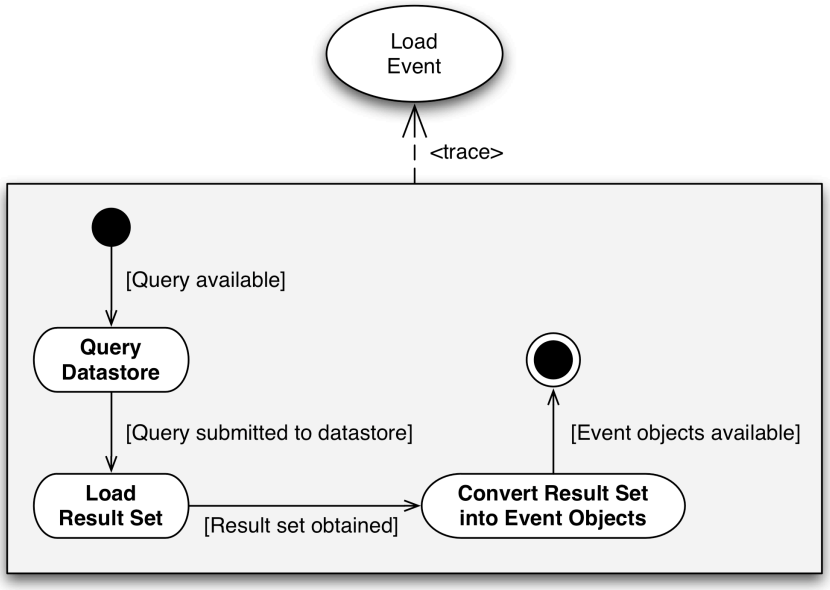
Use Case(s):	Convert Event Formats
Comments:	A generic data sharing mechanism must permit the transparent exchange of information between similar event components. In this manner, third party extensions may operate on event information without being concerned about the underlying persistence implementations.
Trigger:	An event component of one implementation is provided for information association with a comparable event component of another implementation. For example, an XML oriented event component is associated with a JDO oriented event component.
Pre-State:	None.
Post-State:	The information in the provided component is represented within the recipient component.

Normal Flow and States:



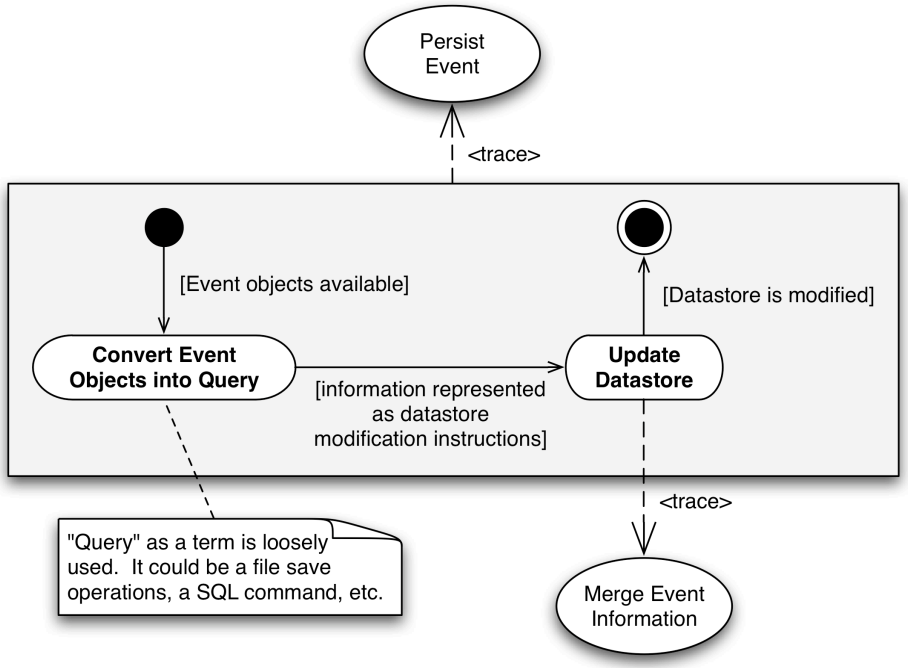
Use Case(s):	Load Event
Comments:	Subsystems and third party extensions must be able to load event information from the data store.
Trigger:	A request is made to obtain event information.
Pre-State:	No event information is loaded.
Post-State:	Event information in the data store is made available for use in an appropriate component.

Normal Flow and States:



Use Case(s):	Persist Event
Comments:	Event information obtained by the information capture subsystem or updated by processing third party extensions must be updated in the data store.
Trigger:	A request to persist edits to event information.
Pre-State:	None.
Post-State:	The event information is represented in the data store.

Normal Flow and States:



Use Case(s):	Merge Event Information
Comments:	Event information added to the data store must be merged in such a manner that it is chronologically correct when loaded for processing.
Trigger:	Event information is added to the data store.
Pre-State:	None.
Post-State:	Event information previously stored is properly organized relative to the newly added information for the purpose of processing.

Normal Flow and States:

Depends on implementation.

Response Engine Use Case Descriptions

The response engine subsystem was defined to satisfy the Proposal's objectives for supporting event processing and environment interaction by third party extensions. The response engine subsystem establishes the mechanisms by which event information appropriate for a

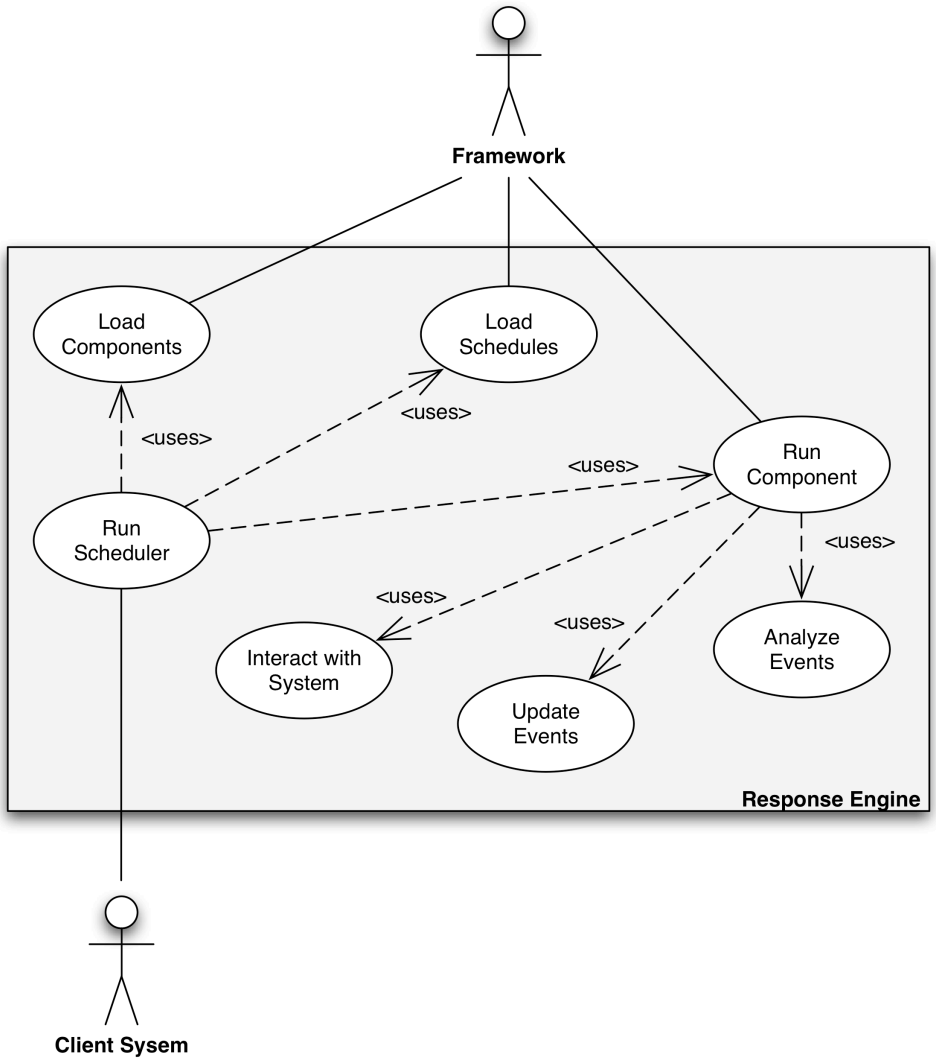
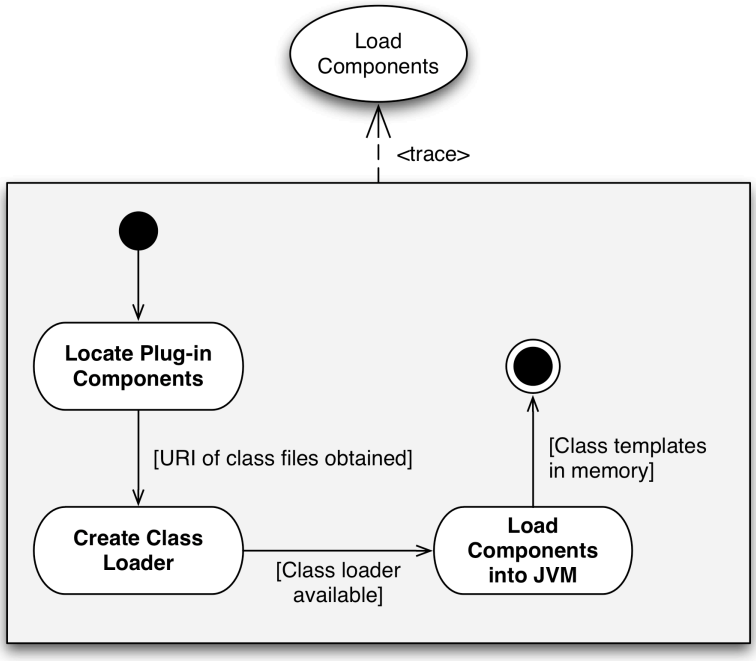


Figure H-6. Response engine use cases.

processing component's domain (*e.g.*, the analysis of failed Web services) may be provided to registered processors. Structures are envisioned for scheduling event information updates and the execution of event information processors. Figure H-6 illustrates the response engine use cases.

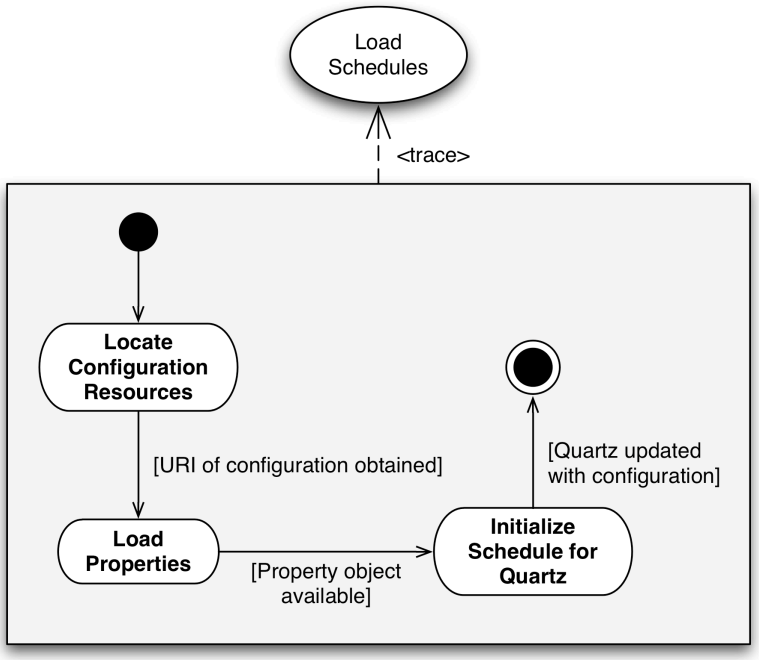
Use Case(s):	Load Components
Comments:	Third party extensions may be represented as pluggable components that are loaded for operation by the scheduler.
Trigger:	The scheduler prepares for the operation of processing components.
Pre-State:	None.
Post-State:	Processing components are available for operation.

Normal Flow and States:



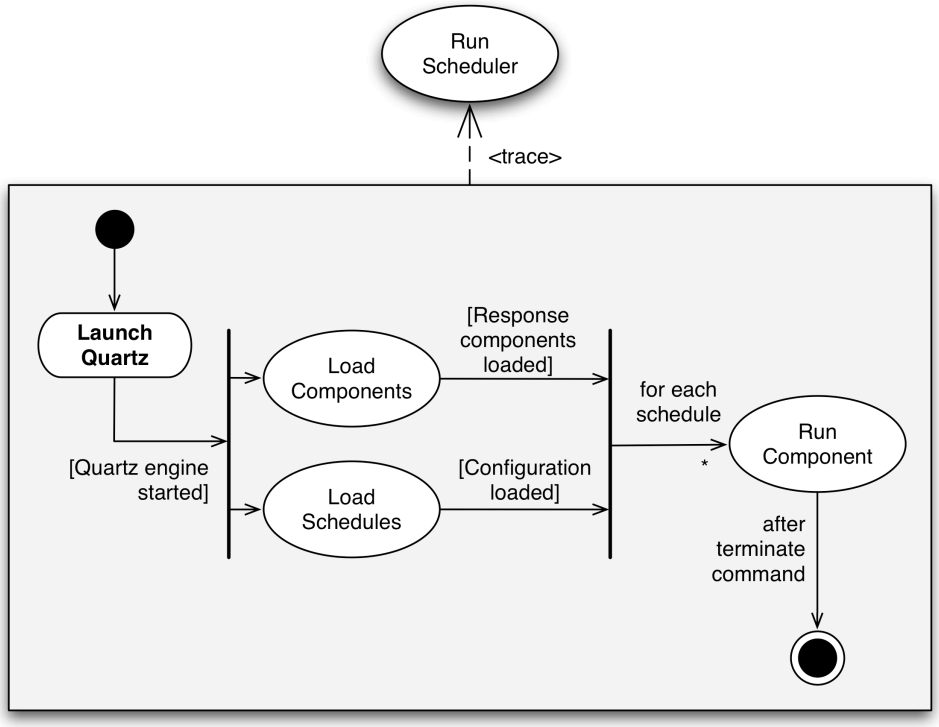
Use Case(s):	Load Schedules
Comments:	Event information processor and environment manager components may operate on varying schedulers. A mechanism must be provided for scheduling activities by these components.
Trigger:	Initialization of scheduler.
Pre-State:	None.
Post-State:	Processing components are executed at appropriate intervals.

Normal Flow and States:



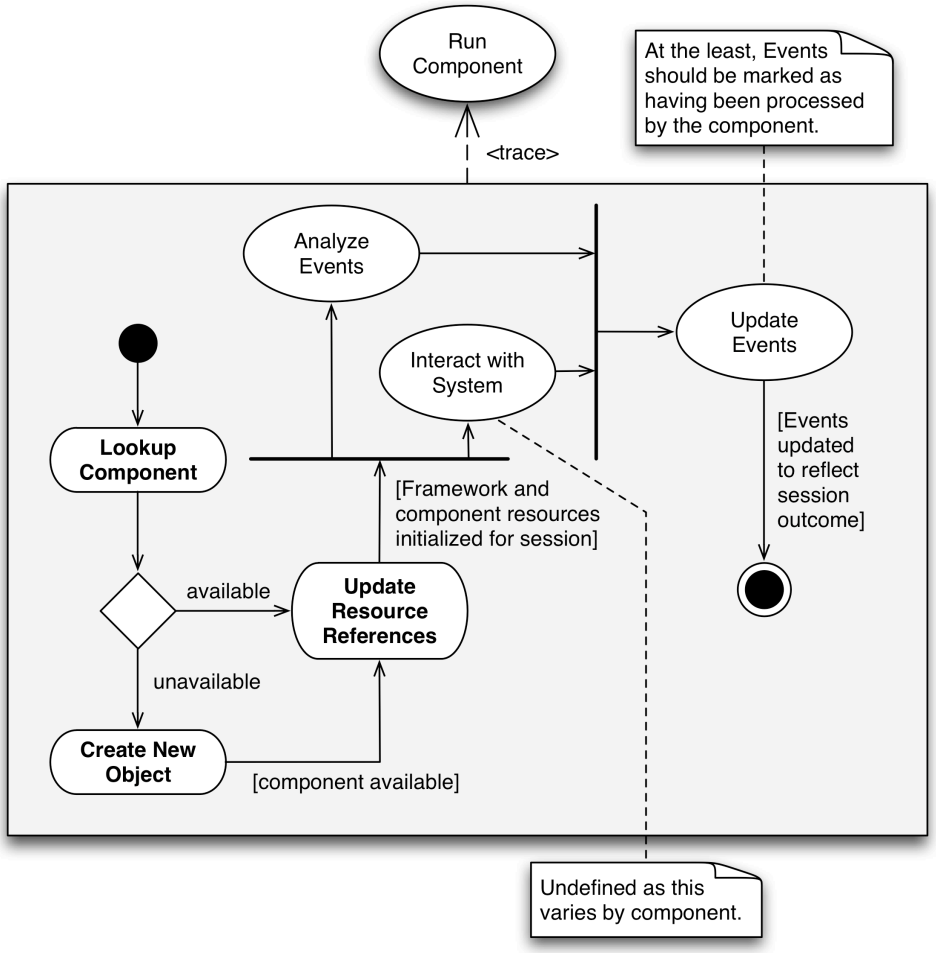
Use Case(s):	Run Scheduler
Comments:	The event information processing components are operated by a scheduler. The scheduler operates iteratively until the occurrence of an event stipulating that processing should terminate.
Trigger:	Execution of scheduling daemon.
Pre-State:	Processing components are ready for operation.
Post-State:	Processing components have completed operation.

Normal Flow and States:



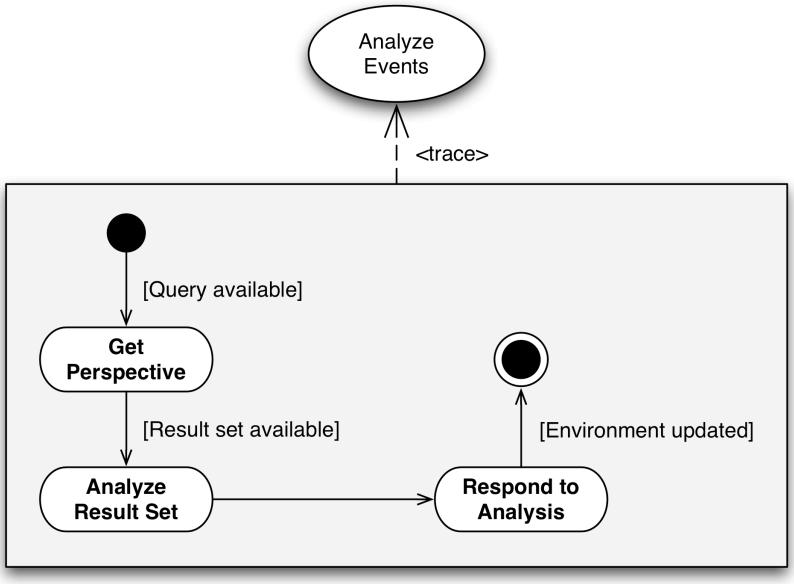
Use Case(s):	Run Component
Comments:	Each processing component is operated in a manner that ensures appropriate access to event information and other framework resources. System interaction is a definition of the implementation. Processed events may be flagged to prevent the component from re-processing handled information.
Trigger:	The processing component's schedule indicates activation.
Pre-State:	The processing component is ready for operation.
Post-State:	The processing component has processed available information.

Normal Flow and States:



Use Case(s):	Analyze Events
Comments:	Processing components are provided with event information for analysis and updates.
Trigger:	The scheduler has activated the processing component.
Pre-State:	The event information for processing has been provided to the component and the scheduler has activated the component.
Post-State:	The provided event information has been processed.

Normal Flow and States:



Use Case(s):	Update Events
Comments:	Processing components are provided with event information relevant to the component's analysis objective. These events should be flagged in some manner, such as by being associated with metadata, to prevent re-processing by the same component.
Trigger:	The scheduler has activated the processing component.
Pre-State:	The processing component has been provided with the opportunity to analyze the provided events and respond to the environment.
Post-State:	The provided event information has been updated to represent its post-processing state.

Normal Flow and States:

Depends on the component implementation.

Use Case(s):	Interact with System
Comments:	Processing components are provided with the opportunity to interact with the environment, such as in response to event analysis. In addition to system or JVM security constraints, policies can be used to enforce processing component behavior.
Trigger:	The scheduler has activated the processing component.

Pre-State: The processing component has been provided with the opportunity to analyze the provided events and respond to the environment.

Post-State: The environment is adjusted according to the processing component's instructions.

Normal Flow and States:

Depends on the component implementation

Information Presentation Use Case Descriptions

The information presentation subsystem was defined to satisfy the Proposal's objectives for supporting the distribution of raw or processed event information with other subsystems or third party extensions. The information presentation subsystem establishes the mecha-

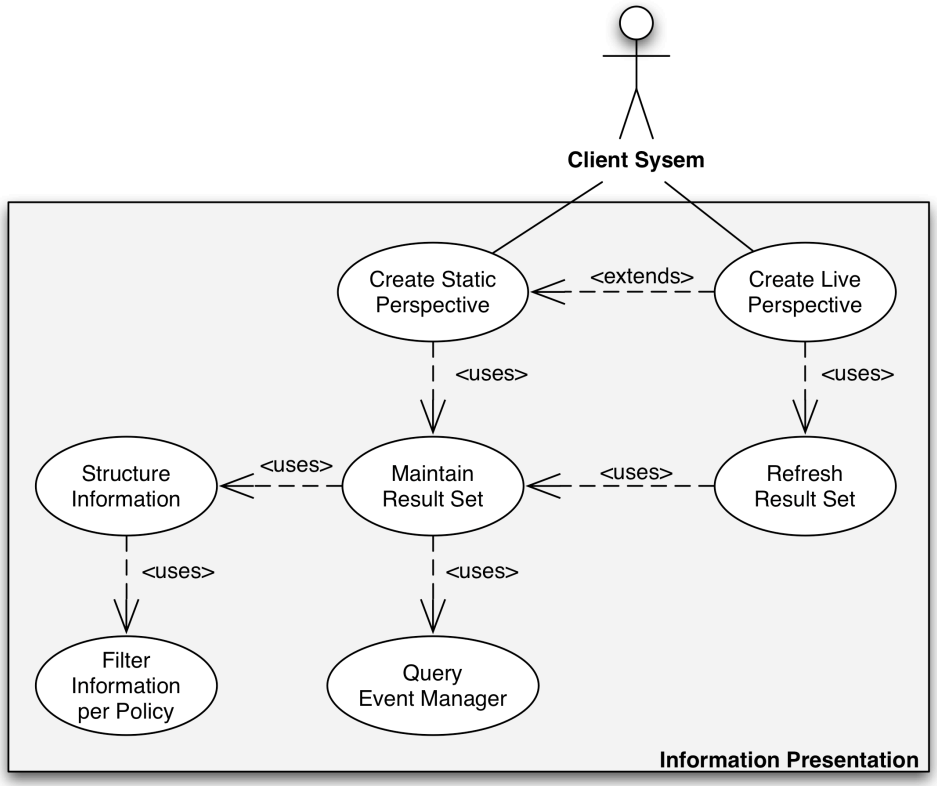
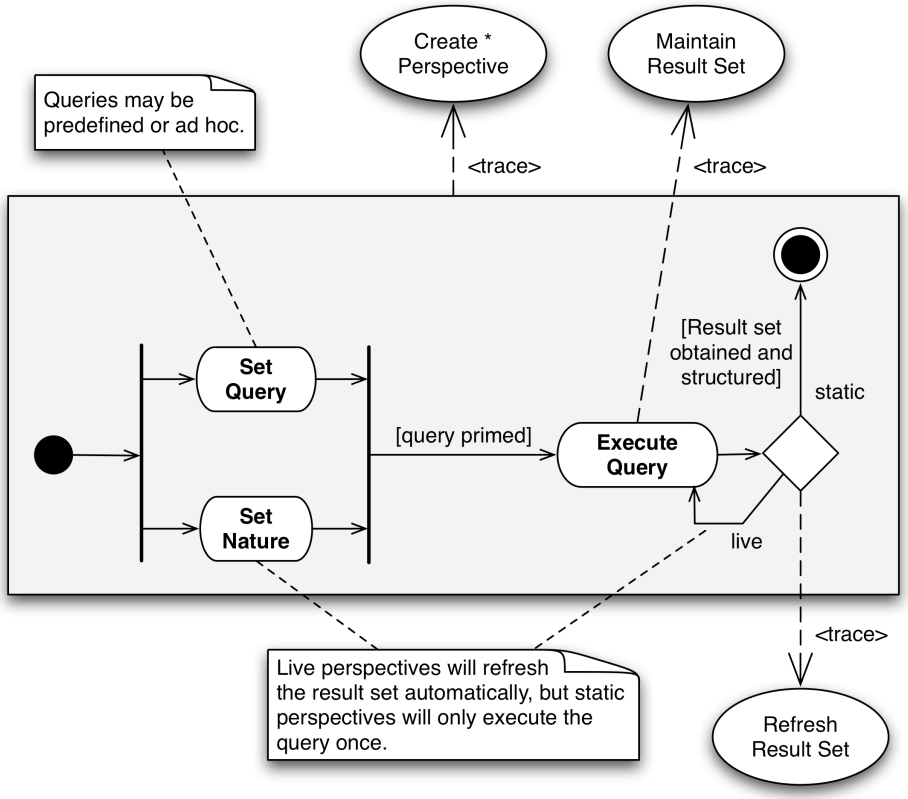


Figure H-7. Information presentation use cases.

nisms by which raw or processed event information may be provided to registered consumers. Structures are envisioned that permit third party extensions to specify the nature of event information desired as a static or dynamic perspective. Figure H-7 illustrates the information presentation use cases.

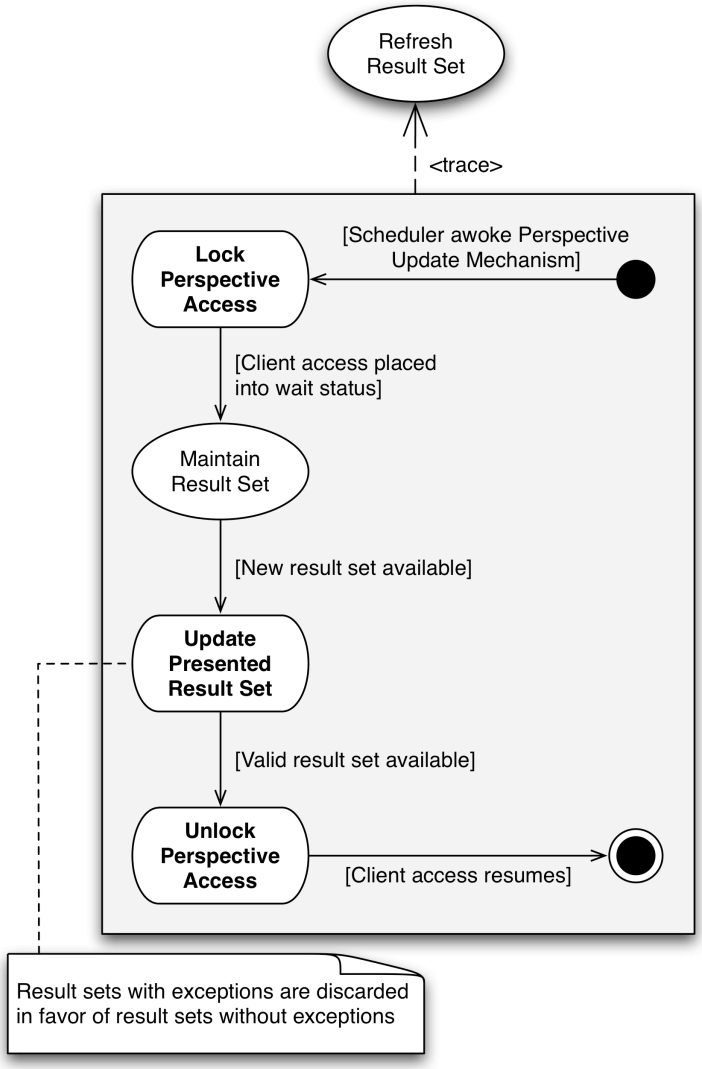
Use Case(s):	Create Static/Live Perspective
Comments:	Perspectives into the event information made available by the event management subsystem are analogous to the SQL view mechanism established by databases such as Oracle or Microsoft SQL Server. The perspectives may be static in that the populating query or calculations are only performed at the Perspective's initialization or live in that subsequent queries are performed to refresh the event information. Static perspectives may be useful for transferring event information to external systems or static display (*e.g.,* as HTML).
Trigger:	A perspective is requested.
Pre-State:	None.
Post-State:	A perspective of the appropriate static or live nature is provided with initial event information sets.

Normal Flow and States:



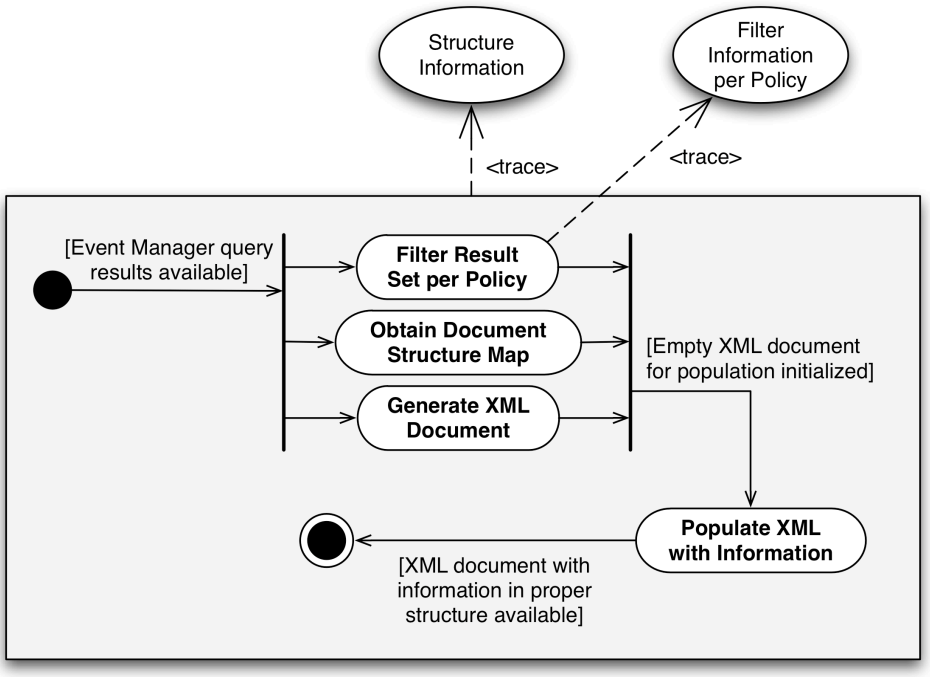
Use Case(s):	Refresh Result Set
Comments:	Live perspectives periodically update their event information. Failed updates should not eliminate a prior valid information set. Consumers should not be permitted to begin processing in the middle of an update, and updates should not be permitted during active processing.
Trigger:	The perspective is updated.
Pre-State:	Consumers are placed in waiting states.
Post-State:	The associated information set is updated and consumers are permitted to access and process the information.

Normal Flow and States:



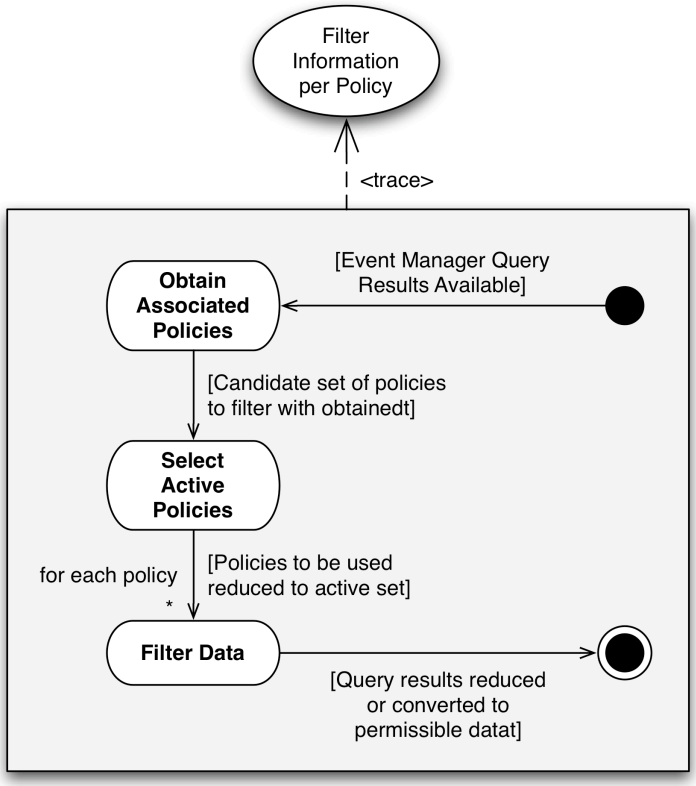
Use Case(s):	Structure Information
Comments:	Perspective information may be structured in a form different from that used by the event management subsystem's native format. Calculated information may be part of the set.
Trigger:	The perspective is updated.
Pre-State:	None.
Post-State:	Information is properly structured for consumption.

Normal Flow and States:



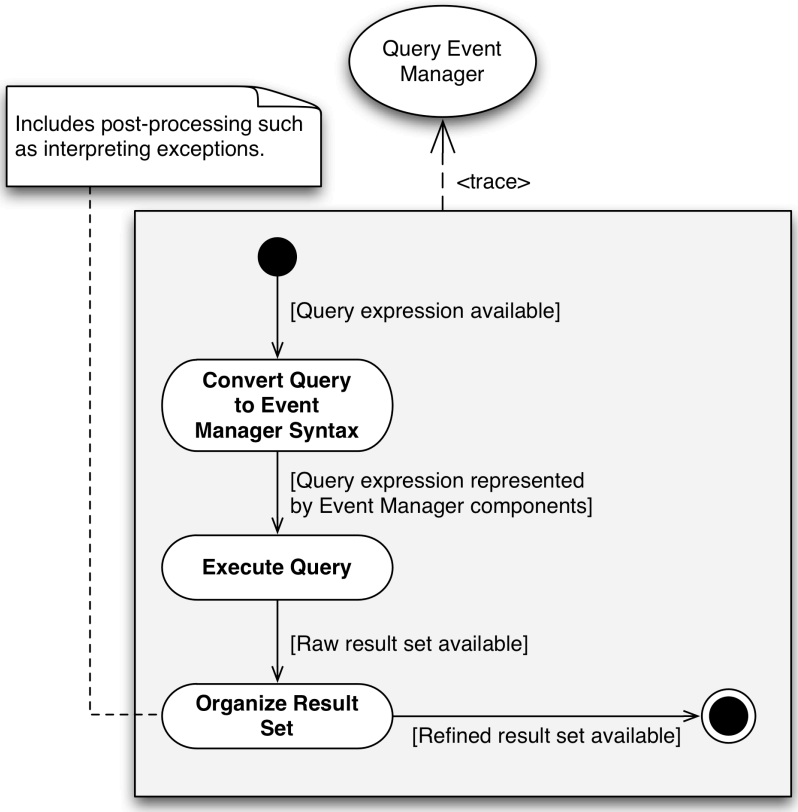
Use Case(s):	Filter Information Per Policy
Comments:	Perspectives may filter information prior to making the information available for consumption. For example, the first five digits of a social security number could be replaced by the x character.
Trigger:	Information is loaded or structured for use in a Perspective.
Pre-State:	The information is in its raw form.
Post-State:	The information is in a filtered form.

Normal Flow and States:

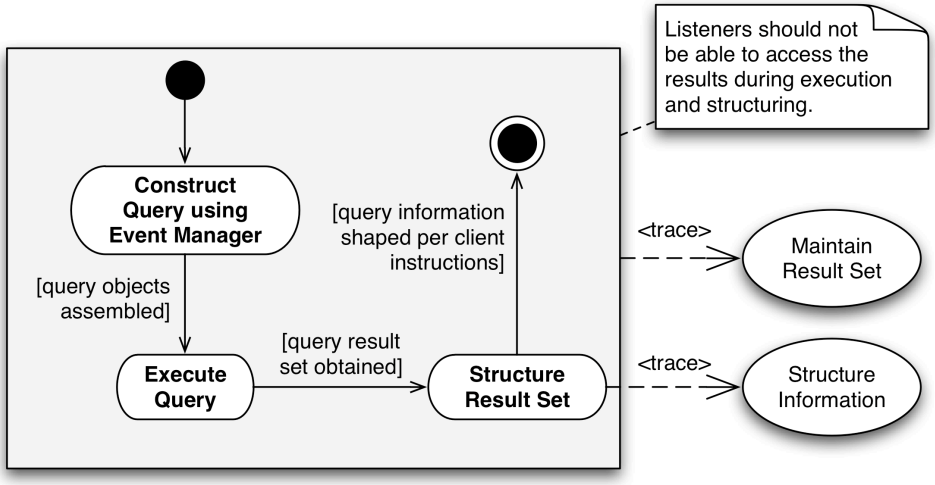


Use Case(s):	Query Event Manager
Comments:	Perspectives are populated using query mechanisms established by event management subsystem implementations. The event information may be restructured and new information may be calculated before the perspective's information set is ready for consumption.
Trigger:	The perspective is updated.
Pre-State:	None.
Post-State:	The associated information set is appropriately structured.

Normal Flow and States:



Alternate Flows and States:



Appendix I

Glossary

abstract class An incomplete class offering functionality common to all potential sub-classes.

aggregation A method for extending the functionality and state of an object. The class definition includes an attribute bearing the desired characteristics, and the class logic manipulates the attribute to achieve the desire effects. Also see composition.

API See Application Programming Interface.

Application Programming Interface A collection of components or functions that provide logic within a common theme for use by client systems.

Architecture. Refers to the structure and relationships, envisioned or actual, of a design's implementation.

black box framework A framework that attempts to hide most of the library complexity from the developer. Often characterized by the use of aggregation as the primary means of behavior extension.

ByteCode The compiled form of Java source code.

class A code template that is used to create objects. In Java, classes offer full support for popular object-oriented development features.

client-side Indicates that the activity in question occurs on a local workstation computer, such as that used in a home or office.

component A general term for a class, object or closely related collection of either.

composition A strong form of aggregation in which the included attribute requires a value to be supplied in a valid form before the object can be successfully created.

dependency A component requirement of the project or an artifact required by the project. Also see transitive- and explicit dependency.

design pattern A codification of expert architecture knowledge for reuse by software designers. Pioneered by Gamma *et al.* in the early 1990s, software engineering design patterns are loosely based on the concepts developed for civil architecture.

Enterprise Generally refers to software applications or systems intended for commercial and industrial use, typically over a network. Networks are typically involved due to remote clients. Clients and servers are often comprised of multiple subsystems of varying type and scope.

event (a) An occurrence of note. (b) An object representing a logical occurrence of note.

framework An architecture intended to organize and control the execution of a specific domain of tasks. Also see Black Box and White Box.

functional specification A specification statement that describes a process, task or behavior that a completed system will feature.

garbage collection The process of releasing memory and other resources consumed by an object that is no longer referenced by system logic.

hot spot A component or method interface that allows functionality to be added to the library.

inheritance A method for reusing functionality defined in a class to model a more accurate version of the concept represented by the class.

interface (a) A special form of class that is completely abstract in nature. Method declarations are given but functionality is not defined (child classes inherit from an interface and define the appropriate functionality for each method). Often used to model a role or viewpoint that might be performed by part of a component library. (b) A term for the a specific method identified by its parameter types.

J2EE See Java 2 Enterprise Edition.

J2SE See Java 2 Standard Edition.

Java An object-oriented programming language invented by Sun. Applications written in pure Java have the ability to be run on most platforms supporting a standard JVM.

Java 2 Enterprise Edition An API that provides functionality specialized for activities commonly performed in Enterprise systems implementing using Java.

Java 2 Standard Edition An API that provides functionality common to many systems implemented using Java.

Java Runtime Environment The collection of applications and libraries that support the execution of Java-based systems.

Java Virtual Machine The application in the JRE that mimics a hardware system in which Java ByteCode can be run.

JRE See Java Runtime Environment.

JVM See Java Virtual Machine.

Mac OS X An advanced operating system based on BSD UNIX published by Apple Computer.

MDI See Multiple Document Interface.

Multiple Document Interface A windowing system in which document windows are displayed inside of a master application window. See also SDI.

non-functional specification A specification statement that describes everything else not described by a functional specification. Often intangible system benefits will be described, including stability and performance.

explicit dependency A dependency introduced to the project through a declaration within the project's build file or project object model.

object A specific instance of a class.

POM See project object model.

Project object model A model of a project's build and configuration requirements. Maven based projects declare a project object model within an XML based pom.xml file.

role Represents a precise range of behavior that a component will exhibit.

scenario A model involving a possible flow of logic and behavior within a system.

SDI See Single Document Interface.

SDK See Software Development Kit.

Service Oriented Architecture An organization of system functionality to accommodate a standard and modular manner of processing requests. SOAs are task specific.

Single Document Interface A windowing system in which all application windows are presented to the user outside of any container window. See also MDI.

SOA See Service Oriented Architecture.

Software Development Kit A collection of APIs for a common development platform. Associated tools, utilities and documentation may also be included.

static class diagram A type of UML diagram that models classes and their relationships within a given system.

Swing The J2SE framework that handles the development of graphical user interfaces.

SWT A component library developed by Eclipse that facilitates the development of graphical user interfaces using native operating system APIs.

text specification Describes a scenario associated with a use case diagram.

transitive dependency A dependency implicitly introduced to the project by a component relationship required by an explicit project dependency. Transitive dependencies are not declared by the project, but can often be controlled by build tools such as Maven to ensure uniform dependency compatibility and version resolution within the classpath.

UML See the Unified Modeling Language.

Unified Modeling Language An object-oriented modeling language that illustrates and describes the functionality, behavior and state of a system. Several syntaxes are used to specialize communication focusing on either static entities (such as classes and their relationships) or dynamic entities (such as method calls between objects).

Unit Test A test of high granularity. In object-oriented systems, a unit test

would typically focus on a single method for an object. Unit tests are an integral part of agile development methodology. JUnit offers a unit test framework and runtime system.

use case diagram A UML diagram that illustrates general domains of functionality or behavior that a system embodies.

viewpoint A perspective of system behavior specific to a user domain. For example, a casual driver and a professional mechanic are interested in different aspects of a car engine.

Web service A form of SOA intended for implementations of business services. Hallmarks include SOAP based communications and Application server hosts.

white box framework A framework strategy that requires developers to understand the library architecture in detail before the components can be properly used. Often characterized by a strong use of inheritance as the primary means of behavior extension.

Windows An operating system family published by Microsoft. Enterprise environments would typically host services on the NT (network technology) family of Windows, such as NT 4, 2000, and XP. Clients might use the NT family, but could also involve the 9x/ME/XP Home series if only a web browser interface or other light client were required.

WS See Web service.

Reference List

AIS. (2005, June 5). Design research in information systems. Retrieved July 13, 2005, from http://www.isworld.org/Researchdesign/drisISworld.htm

Alur, D., Crupi, J., & Malks, D. (2003). *Core J2EE patterns: Best practices and design strategies.* Palo Alto, CA: Sun Microsystems Press.

Anselmi, J., Ardagna, D., & Cremonesi, P. (2007). A QoS-based selection approach of autonomic grid services. *Paper presented at the 2007 Workshop on Service-Oriented Computing Performance: Aspects, Issues, and Approaches*, New York.

Apache. (2007, February 14). Hadoop map/reduce. Retrieved July 8, 2008, from http://wiki.apache.org/hadoop/ProjectDescription

Apple Computer. (2003). The Foundation Framework. Retrieved April 24, 2003, from http://developer.apple.com/techpubs/macosx/Cocoa/Reference/Foundation/Java/Intro/IntroFoundation.html#//apple_ref/doc/uid/20000688

Apple Computer. (2006). *Mac OS X technology overview.* Cupertino, CA.

Apple Computer. (2007). *Windows compatibility and Mac OS X* (Technical Brief). Cupertino, CA.

Apple Computer. (2008). Java. *Developer Connection.* Retrieved May 19, 2008, from http://developer.apple.com/java/

Armitage, J. (2004). Are agile methods good for design? *Interactions, 11*(1), 14-23.

Arnold, K., Gosling, J., & Holmes, D. (2005). *The Java programming language* (4th ed.). Upper Saddle River, NJ: Addison-Wesley.

Arsanjani, A., Hailpern, B., Martin, J., & Tarr, P. (2003). Web Services Promises and Compromises. *Queue, 1*(1), 48-58.

Arthur, J., & Azadegan, S. (2005). Spring framework for rapid open source J2EE Web application development: A case study. *Paper presented at the 6th International Conference on Software Engineering, Artificial Intelligence, Network and Parallel/Distributed Computing and 1st ACIS International Workshop on Self-Assembling Wireless Networks*, Washington DC.

Astels, D. R. (2003). *Test-driven development: A practical guide*. Upper Saddle River, NJ: Prentice Hall.

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., & Zhou, Y. (2007). Evaluating static analysis defect warnings on production software. *Paper presented at the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, CA.

Babaoğlu, Ö. (2005). *Self-star properties in complex information systems: Concepts and practical foundations*. New York: Springer.

Banes, J. (2004). Building the ultimate logging solution: …and solving common interoperability issues. *Java Developer's Journal, 9*(5), 30-32.

Bar, M., & Fogel, K. (2003). *Open source development with CVS* (3rd ed.). Scottsdale, AZ: Paraglyph Press.

Bare Bones. (2007). BBEdit. Retrieved June 12, 2008, from http://www.barebones.com/products/bbedit/

Barra, M., Cillo, T., Santis, A. D., Petrillo, U. F., Negro, A., & Scarano, V. (2002). Multimodel Monitoring of Web Servers. *Multimedia, 9*(3), 32-41.

Bauer, C., & King, G. (2006). *Java persistence with Hibernate*. Greenwich, CT: Manning.

Be. (1997). *Be developer's guide*. Cambridge: O'Reilly.

Berczuk, S. (2003). Pragmatic software configuration management. *IEEE Software, 20*(2), 15-17.

Berczuk, S., & Appleton, B. (2003). *Software configuration management patterms: Effective teamwork, practical integration*. Boston: Addison-Wesley.

Berner, S., Weber, R., & Keller, R. K. (2007). Enhancing software testing by judicious use of code coverage information. *Paper presented at the 29th International Conference on Software Engineering*, Washington DC.

Bhat, T., & Nagappan, N. (2006, September 21-22). Evaluating the efficacy of test-driven development: Industrial case studies. *Paper presented at the 5th ACM-IEEE International Symposium on Empical Software Engineering*, Rio de Janeiro, Brazil.

Biljon, J. V., Kotze, P., Renaud, K., McGee, M., & Seffah, A. (2004). The use of anti-patterns in human-computer interaction: Wise or ill-advised? *Paper presented at the The 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, South Africa.

Boekhoudt, C. (2003). The big bang theory of IDEs. *Queue, 1*(7), 74-82.

Bowyer, J., & Hughes, J. (2006, February 19-23). Assessing undergraduate experience of continuous integration and test-driven development. *Paper presented at the 34th SIGCSE Technical Symposium Computer Science Education*, Reno, NV.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., *et al.* (2000). Simple Object Access Protocol (SOAP) 1.1. Retrieved February 26, 2005, from http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2004, February 4). Extensible markup language (XML) 1.0. 3rd ed. Retrieved February 28, 2005, from http://www.w3.org/TR/REC-xml/

Brett, T. C. (2004). *Quality of service in e-commerce: A literature review of transaction and environment concerns.* Unpublished manuscript.

Brett, T. C. (2005). *Exploration of a query based engine to facilitate error recovery in a J2EE environment.* Unpublished manuscript.

Business Objects. (2008). Crystal Reports 2008. Retrieved July 7, 2008, from http://www.businessobjects.com/product/catalog/crystalreports/

Canfora, G., Cimitile, A., Garcia, F., Piattini, M., & Visaggio, C. A. (2006, September 21-22). Evaluating advantages of test driven development: A controlled experiment with professionals. *Paper presented at the 5th ACM-IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil.

Casey, J., Massol, V., Porter, B., Sanchez, C., & Zyl, J. v. (2006). *Better builds with Maven: The how-to guide for Maven 2.0* (1.0.2 ed.). Marina del Rey, CA: Mergere.

Cavaness, C. (2004). *Programming Jakarta Struts* (2nd ed.). Sebastopol, CA: O'Reilly.

Cavaness, C. (2006). *Quartz job scheduling framework: Building open source enterprise applications.* Upper Saddle River, NJ: Prentice Hall.

Chandra, K., Chandra, S. S., & Chandra, S. S. (2003). A comparison of VBScript, Javascript, and Jscript. *Journal of Computing Sciences in Colleges, 19*(1), 323-335.

Chavda, K. F. (2004). Anatomy of a Web Service. *Journal of Computing Sciences in Colleges, 19*(3), 124-134.

CodeHAUS. (2007, August 20). Maven 2 FindBugs Plugin - Introduction. Retrieved August 23, 2007, from http://mojo.codehaus.org/findbugs-maven-plugin/index.html

Cognos. (2008). IBM Cognos 8 business intelligence. Retrieved July 7, 2008, from http://www.cognos.com/products/cognos8businessintelligence/reporting.html

CollabNet. (2006). Subclipse. Retrieved July 24, 2007, from http://subclipse.tigris.org

CollabNet. (2007, January 18). Changes. *Subversion Releases* Retrieved May 17, 2007, from http://svn.collab.net/repos/svn/tags/1.4.3/CHANGES

Collins-Sussman, B., Fitzpatrick, B. W., & Pilato, C. M. (2004). *Version control with Subversion.* Sebastepol, CA: O'Reilly.

Compuware. (2005). DevPartner Studio Professional Edition. Retrieved April 3, 2005, from http://www.compuware.com/products/devpartner/studio.htm

Conboy, K., & Fitzgerald, B. (2004). Toward a conceptual framework of agile methods: A study of agility in different disciplines. *Paper presented at the 2004 ACM Workshop on Interdisciplinary Software Engineering Research*, Newport Beach, CA.

Cortes, M., Fontoura, M., & Lucena, C. (2003). Using refactoring and unification rules to assist framework evolution. *ACM SIGSOFT Software Engineering Notes, 28*(6), 1-5.

Cruz, S. M. S. d., Campos, L. M., Campos, M. L. M., & Pires, P. F. (2003). A data mart approach for monitoring Web services usage and evaluating quality of services. *Paper presented at the XVIII Brazilian Symposium of Data Bases.*

Cruz, S. M. S. d., Campos, M. L. M., Pires, P. F., & Campos, L. M. (2004, July 6-9). Monitoring e-business Web services usage through a log based architecture. *Paper presented at the IEEE International Conference on Web Services (ICWS'04)*, San Diego, USA.

D'Souza, D. F., & Wills, A. C. (1998). *Objects, components, and frameworks with UML: The Catalysis™ approach.* New York: Addison-Wesley.

Dashofy, E. M., Hoek, A. v. d., & Taylor, R. N. (2002). Towards architecture-based self-healing systems. *Paper presented at the First workshop on Self-Healing Systems*, Charleston, SC.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107-113.

Dutta, S. (2004). Building manageability. *Java Developer's Journal, 9*(11), 40-44.

Eclipse Foundation. (2008). About the Eclipse Foundation. Retrieved June 12, 2008, from http://www.eclipse.org/org/

Edwards, S. H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *Paper presented at the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, VA.

Enes, P. (2007). *Build and release management: Supporting development of accelerator control software at CERN.* Unpublished Master of Science Thesis, Norwegian University of Science and Technology.

Estublier, J., Leblang, D., Hoek, A. v. d., Conradi, R., Clemm, G., Tichy, W., *et al.* (2005). Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology, 14*(4), 340-383.

Farrell, T. (2004). Service-oriented architecture. *Java Developer's Journal, 9*(4), 12-14.

Fayad, M. E., & Schmidt, D. C. (1997). Object-oriented application frameworks. *Communications of the ACM, 40*(10), 32-38.

Fayerman, P. (2008, April 23). New BC health law could lead to privacy abuse. *The Vancouver Sun* Retrieved July 8, 2008, from http://www.canada.com/vancouversun/news/story.html?id=b16e1348-01a4-41c7-a199-a9d5819ffc72&k=23212

Fordin, S. (2004, October). Java architecture for XML binding. Retrieved September 25, 2005, from http://java.sun.com/webservices/jaxb/about.html

Forward, A., & Lethbridge, T. C. (2002). The relevance of software documentation, tools and technologies: A survey. *Paper presented at the ACM Symposium on Document Engineering*, McLean, VA.

Foster, J. S., Hicks, M. W., & Pugh, W. (2007). Improving software quality with static analysis. *Paper presented at the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, San Diego, CA.

Fowler, M. (2004, January 23). Inversion of control containers and the dependency injection pattern. Retrieved August 7, 2007, from http://www.martinfowler.com/articles/injection.html

Fowler, M. (2006, May 1). Continuous integration. Retrieved July 27, 2007, from http://martinfowler.com/articles/continuousIntegration.html

Freeman, S., Mackinnon, T., Pryce, N., & Walnes, J. (2004). Mock roles, objects. *Paper presented at the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada.

Gaffney, C., Trefftz, C., & Jorgensen, P. (2004). Tools for coverage testing: Necessary but not sufficient. *Journal of Computing Sciences in Colleges, 20*(1), 27-33.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Menlo Park, CA: Addison-Wesley.

Garlan, D., & Schmerl, B. (2002). Model-based adaptation for self-healing systems. *Paper presented at the First workshop on Self-Healing Systems*, Charleston, SC.

Garsombke, F. (2003). Taking continuous integration to the next level: Continuous what? *Java Developer's Journal, 8*(4), 30-34.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Lea, D., & Holmes, D. (2006). *Java concurrency in practice*. Boston: Addison-Wesley Professional.

Gombotz, R., & Dustdar, S. (2005). On Web service workflow mining. *Paper presented at the Third International Conference on Business Process Management*, Nancy, France.

Gospodnetic, O., & Hatcher, E. (2004). *Lucene in action*. Greenwich, CT: Manning Publications.

Graham, S., Davis, D., Simeonov, S., Daniels, G., Brittenham, P., Nakamura, Y., *et al.* (2005). *Building Web services with Java* (2nd ed.). Indianapolis, IN: Sams Publishing.

Greenfield, J., & Short, K. (2003). Software factories: Assembling applications with patterns, models, frameworks and tools. *Paper presented at the Conference on Object Oriented Programming Systems Languages and Applications*, Anaheim, CA.

Gulcu, C. (2002). *The complete Log4J manual*: QOS.ch.

Gulcu, C. (2005, May 5). Logging-log4j Wiki. Retrieved August 10, 2005, from http://wiki.apache.org/logging-log4j/

Gupta, S. (2003). *Logging in Java with the JDK 1.4 Logging API and Apache log4j*. New York: Springer-Verlag.

Gurp, J. v., & Bosch, J. (2001). Design, implementation and evolution of object oriented frameworks: Concepts and guidelines. *Software—Practice and Experience,* 31(3), 277-300.

Haftmann, F., Kossmann, D., & Lo, E. (2007). A framework for efficient regression tests on database applications. *The International Journal on Very Large Data Bases, 16*(1), 145-164.

Harold, E. R. (2005, May 3). Measure test coverage with Cobertura. *developerWorks*. Retrieved August 11, 2007, from http://www.ibm.com/developerworks/java/library/j-cobertura/

Hatcher, E., & Loughran, S. (2003). *Java development with Ant*. Greenwich, CT: Manning.

Hazzan, O., & Dubinsky, Y. (2007). Why software engineering programs should teach agile software development. *ACM SIGSOFT Software Engineering Notes, 32*(2), 1-3.

Helsinger, A., Lazarus, R., Wright, W., & Zinky, J. (2003, July 14-18). Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems. *Paper presented at the AAMAS'03*, Melbourne.

Herborth, C. (2006, July 11). How to use Subversion with Eclipse. *developerWorks* Retrieved July 20, 2007, from http://www-128.ibm.com/developerworks/opensource/library/os-ecl-subversion/

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly, 28*(1), 75-105.

Highsmith, J. (2002). *Agile software development ecosystems.* New York: Addison-Wesley Professional.

Holmes, A., & Kellogg, M. (2006, July 23-28). Automating functional tests using Selenium. *Paper presented at the AGILE 2006*, Minneapolis, MN.

Hovermeyer, D. H., & Pugh, W. W. (2007, May 31). FindBugs™ manual. Retrieved August 23, 2007, from http://findbugs.sourceforge.net/manual/index.html

HSQLDB Development Group. (2008, June 2). HSQLDB - 100% Java database. Retrieved June 12, 2008, from http://hsqldb.org

Hu, J., & Zhong, N. (2005). Clickstream log acquisition with web farming. *Paper presented at the International Conference on Web Intelligence*, France.

Huang, T.-Y., Chou, P.-C., Tsai, C.-H., & Chen, H.-A. (2007). Automated fault localization with statistically suspicious program states. *Paper presented at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*, San Diego, CA.

Hunt, A., & Thomas, D. (2006). *Pragmatic unit testing in Java with JUnit.* Raleigh, NC: The Pragmatic Programmers.

Iltchenko, A. (2006). Moving to SOA in J2EE 1.4: How to take an existing J2EE 1.3 bean and convert it into a Web service endpoint with almost no changes to its business interface. *Java Developer's Journal, 11*(2), 12-22.

JBoss. (2008). JBoss enterprise application platform. Retrieved June 9, 2008, from http://www.jboss.com/products/platforms/application

JPOX. (2008, July 9). JPOX: Java persistent objects. Retrieved July 9, 2008, from http://www.jpox.org/docs/

Kotula, J. (1998). Using patterns to create component documentation. *IEEE Software, 15*(2), 84-92.

Kramer, D. API documentation from source code comments: A case study of JavaDoc. *Paper presented at the 17th Annual International Conference on Computer Documentation*, New Orleans, LA.

Lafore, R. (2002). *Object-oriented programming in C++* (4th ed.). Indianapolis, IN: SAMS.

Lai, R., Steel, C., & Nagappan, R. (2005). *Core security patterns: Best practices and strategies for J2EE™, Web services, and identity management*. Upper Saddle River, NJ: Prentice Hall.

Landre, E., Wesenberg, H., & Olmheim, J. (2007). Agile enterprise software development using domain-driven design and test first*. Paper presented at the Conference on Object-Oriented Programming Systems Languages and Applications*, Montreal, Canada.

Larson, E., & Stephens, B. (2000). *Administrating Web servers, security, & maintenance*. Upper Saddle River, NJ: Prentice Hall.

Lee, J., Gunter, D., Stoufer, M., & Tierney, B. (2002). Monitoring data archives for grid environments*. Paper presented at the Conference on High Performance Networking and Computing*, Baltimore.

Louridas, P. (2005, July). JUnit: Unit testing and coding in tandem. *IEEE Software*, 22(4), 12-15.

Louridas, P., & Loucopoulos, P. (2000). A generic model for reflective design. *Transactions on Software Engineering Methodology, 9*(2), 199-237.

MacWorld. (2005a, March 18). BBEdit 8.0.3. Retrieved June 12, 2008, from http://www.macworld.com/article/43661/2005/03/bbedit803.html

MacWorld. (2005b, December 5). OmniGraffle Professional 4.0. Retrieved June 12, 2008, from http://www.macworld.com/article/48248/2005/12/omnigrafflepro4.html

MacWorld. (2006, August 7). WWDC: Microsoft kills Virtual PC for Mac. Retrieved June 12, 2008, from http://www.macworld.com/article/52243/2006/08/vpc.html

Mason, M. (2006). *Pragmatic version control using Subversion* (2nd ed.). Raleigh, NC: The Pragmatic Bookshelf.

Maximilien, E. M., & Williams, L. (2003). Assessing test-driven development at IBM*. Paper presented at the 25th International Conference on Software Engineering*, Portland, OR.

McGregor, C., & Schiefer, J. (2003). A framework for analyzing and measuring business performance with web services*. Paper presented at the Fifth International Conference on E-Commerce*, Pittsburgh, PA.

McGregor, L. (2003). Managing J2EE systems with JMX and JUnit. *Java Developer's Journal, 8*(11), 12-18.

McManus, E. (2002). JSR 3: Java Management Extensions (JMX) Specification. Retrieved February 16, 2005, from http://www.jcp.org/en/jsr/detail?id=3

McManus, E., & Vienot, S. (2003). JSR 160: Java Management Extensions (JMX) remote API 1.0. Retrieved February 16, 2005, from http://www.jcp.org/en/jsr/detail?id=160

Meadow, C. T., Boyce, B. R., & Kraft, D. H. (2000). *Text information retrieval systems* (2nd ed.). San Diego, CA: Academic Press.

Mergere. (2007). *Maven IDE plugins* (1.2 ed.). Marina del Rey, CA: Mergere.

Microsoft. (2003a). About the Microsoft Foundation Classes. Retrieved January 10, 2003, from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/_mfc_about_the_microsoft_foundation_classes.asp

Microsoft. (2003b). Microsoft Foundation Class Library Version 6.0. Retrieved January 20, 2003, from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcmfc98/html/_mfc_hierarchy_chart.asp

Microsoft. (2004, July 14). How to use compressed (zipped) folders in Windows XP. *Help and Support* Retrieved July 19, 2007, from http://support.microsoft.com/kb/306531

Milanovic, N., & Malek, M. (2004). Current solutions for Web service composition. *Internet Computing, 8*(6), 51-59.

Monarchi, D. E., & Puhr, G. I. (1992). A research typology for object-oriented analysis and design. *Communications of the ACM, 35*(9), 35-47.

Monson-Haefel, R. (2004). *Enterprise JavaBeans™* (4th ed.). Sebastopol, CA: O'Reilly.

Oak, H. (2004). *Pro Jakarta commons.* Berkeley, CA: Apress.

Olan, M. (2003). Unit testing: Test early, test often. *Journal of Computing Sciences in Colleges, 19*(2), 319-328.

OMG. (2001). *OMG unified modeling language specification.* Needham, MA: OMG.

OMG. (2006, January 5). Unified modeling language (UML) version 2.0. Retrieved February 11, 2006, from http://www.omg.org/technology/documents/formal/uml.htm

Pepperdine, K. (2003). Customizing Ant. *Java Developer's Journal, 8*(9), 36-40.

PKWare. (2007, July 19, 2007). ZIP file format specification. Retrieved September 10, 2008 from http://www.pkware.com/documents/casestudies/APPNOTE.TXT

Rainsberger, J. B., & Stirling, S. (2005). *JUnit recipies.* Greenwich, CT: Manning.

Reese, G. (2000). *Database programming with JDBC and Java* (2nd ed.). Sebastapol, CA: O'Reilly.

Richter, C. (1999). *Designing flexible object-oriented systems with UML.* Indianapolis, IN: Macmillan Technical Publishing.

Roberts, D., & Johnson, R. (n.d.). Evolving Frameworks. Retrieved November 15, 2002, from http://st-www.cs.uiuc.edu/users/droberts/evolve.html

Rosenstein, M. (2000, October 17-20). What is actually taking place on Web sites: e-Commerce lessons from Web server logs*. Paper presented at the EC'00*, Minneapolis.

Schach, S. R. (2002). *Object-oriented and classical software engineering* (5th ed.). New York: McGraw-Hill.

Schmidt, D. C., & Buschmann, F. (2003, May 3-10). Patterns, frameworks, and middleware: Their synergistic relationships*. Paper presented at the 25th International Conference on Software Engineering*.

Schmidt, D. C., Gokhale, A., & Natarajan, B. (2004). Leveraging application frameworks. *Queue, 2*(5), 66-75.

Schwalbe, K. (2006). *Information technology project management* (4th ed.). Boston: Thomson Course Technology.

Senthil, R., Kushwaha, D. S., & Misra, A. K. (2007). An improved component model for component based software engineering. *ACM SIGSOFT Software Engineering Notes, 32*(4), 1-9.

Shalloway, A., & Trott, J. R. (2001). *Design patterns explained: A new perspective on object-oriented design.* Upper Saddle River, NJ: Pearson Education.

Sherif, K., & Vinze, A. (1999). A qualitative model for barriers to software reuse adoption*. Paper presented at the 20th International Conference on Information Systems*, Charlotte, NC.

Simpson, B., & Toussi, F. (2005, May 29). HSQLDB user guide. Retrieved August 23, 2007, from http://hsqldb.sourceforge.net/web/hsqlDocsFrame.html

Singh, I., Brydon, S., Murray, G., Ramachandran, V., Violleau, T., & Stearns, B. (2004). *Designing Web Services with the J2EE™ 1.4 platform: JAX-RPC, SOAP, and XML technologies.* Santa Clara, CA: Addison-Wesley.

Singhal, A. (2001). Modern information retrieval: A brief overview. *Bulletin of the IEEE Computer Society Technical Committee on data Engineering*, 1-9.

Spiliopoulou, M. (2000). Web usage mining for web site evaluation. *Communications of the ACM, 43*(8), 127-134.

Staff, D., & Ernst, M. D. (2004a). An experimental evaluation of continuous testing during development. *Paper presented at the ISSTA '04*, Boston.

Staff, D., & Ernst, M. D. (2004b). Mock object creation for test factoring. *Paper presented at the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Washington, DC.

Stanek, W. R. (2002). *XML pocket consultant*. Redmond, WA: Microsoft Press.

Stevens, P., & Pooley, R. (2000). *Using UML: Software engineering with objects and components*. Harlow, England: Pearson Education Limited.

Sturm, T. (2002, November 8). UML Modeling Integrated with Sun ONE Studio. Retrieved May 2, 2003, from http://forte.sun.com/ffj/articles/poseidon.html

Sun Microsystems. (1999). *Java Management Extensions White Paper: Dynamic Management for the Service Age*. Palo Alto, CA: Sun Microsystems.

Sun Microsystems. (2001, Nov 26). Java logging overview. Retrieved February 28, 2005, from http://java.sun.com/j2se/1.5.0/docs/guide/logging/overview.html

Sun Microsystems. (2003). JAR file specification. Retrieved July 24, 2007, from http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html

Sun Microsystems. (2004). Package java.util.logging. *Java 2 Platform Standard Ed. 5.0* Retrieved February 28, 2005, from http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/package-summary.html

Sun Microsystems. (2005). adventurebuilder: Project home. Retrieved April 25, 2006, from https://adventurebuilder.dev.java.net

Sun Microsystems. (2006). BluePrints. Retrieved April 25, 2006, from http://java.sun.com/reference/blueprints/

Sun Microsystems. (2008a). GlassFish - Open source application server. Retrieved June 9, 2008, from https://glassfish.dev.java.net/

Sun Microsystems. (2008b). Java DB at a glance. Retrieved June 12, 2008, from http://developers.sun.com/javadb/

Sun Microsystems. (2008c). Java Platform, Enterprise Edition (EE): Compatibility. *Sun Developer Network (SDN)* Retrieved May 19, 2008, from http://java.sun.com/j2ee/compatibility_1.4.html

Tague, J., Salminen, A., & McClellen, C. (1991). Complete formal model for information retrieval systems. *Paper presented at the Proceedings of the 14th Annual Interna-*

tional ACM SIGIR Conference on Research and Development in Information Retrieval, Chicago.

Telles, M. (2001). *C# black book*. Sebastopol, CA: Paraglyph Press.

Telles, M., & Hsieh, Y. (2001). *The science of debugging*. Scottsdale, AZ: The Coriolis Group.

Thai, T., & Lam, H. Q. (2001). *.NET framework essentials*. Sebastopol, CA: O'Reilly.

TheServerSide.com. (2005, March 27). Application server matrix. Retrieved May 19, 2008, from http://www.theserverside.com/tt/reviews/matrix.tss

Tigris.org. (2007, January 18). Subversion 1.4.3 - Changes. Retrieved June 9, 2008, from http://svn.collab.net/repos/svn/tags/1.4.3/CHANGES

Tong, H., & Zhang, S. (2006). A fuzzy multi-attribute decision making algorithm for Web services selection based on QoS. *Paper presented at the 2006 IEEE Asia-Pacific Conference on Services Computing*, GuangZhou, China.

Toussi, F. (2008). Chapter 9. SQL syntax. Retrieved June 12, 2008, from http://hsqldb.org/doc/guide/ch09.html

Turner, S. (2004). Analog: The most popular logfile analyser in the world. Retrieved February 15, 2005, from http://www.analog.cx/

Tyagi, S., McCammon, K., Vorburger, M., & Bobzin, H. (2004). *Core Java data objects*. Palo Alto, CA: Prentice Hall PTR.

Valetto, G., & Kaiser, G. (2003). Using Process Technology to Control and Coordinate Software Adaptation. *Paper presented at the 25th International Conference on Software Engineering*, Portland, OR.

VMware. (2008). Virtualization basics. Retrieved June 12, 2008, from http://www.vmware.com/virtualization/

Walls, C., & Breidenbach, R. (2007). *Spring in action* (2nd ed.). Greenwich, CT: Manning Publications.

Wang, Q. (2005). Towards a rule model for self-adaptive software. *ACM SIGSOFT Software Engineering Notes, 30*(1), 8-12.

Whitten, J. L., Bentley, L. D., & Dittman, K. C. (2001). *Systems analysis and design methods* (5th ed.). New York: McGraw-Hill Higher Education.

Wick, M., Stevenson, D., & Wagner, P. (2005). Using testing and JUnit across the curriculum. *Paper presented at the 36th SIGCSE Technical Symposium on Computer Science Education*, St. Louis, MO.

Williams, K., & Daniel, B. (2004). An introduction to service data objects: Integrating relational data into Web applications. *Java Developer's Journal, 9*(10), 10-16.

Wysocki, R. K., Beck Jr., R., & Crane, D. B. (2000). *Effective project management* (2nd ed.). New York: John Wiley and Sons.

Yang, Q., Li, J. J., & Weiss, D. (2006). A survey of coverage based testing tools. *Paper presented at the 2006 International Workshop on Automation of Software Test*, New York.

yWorks. (2008). yDoc. Retrieved June 12, 2008, from http://www.yworks.com/en/products_ydoc.html

Zimmerman, O., Krogdahl, P., & Gee, C. (2004, June 2). Elements of service-oriented analysis and design. *developerWorks: SOA and Web services* Retrieved February 17, 2006, from http://www-128.ibm.com/developerworks/webservices/library/ws-soad1/

Zyl, J. V. (2006). Maven: A different way of looking at software development. *Java Developer's Journal, 11*(5), 42-46.