INTEGRATION OF BUSINESS EVENTS AND RULES MANAGEMENT WITH THE WEB SERVICES MODEL

By

KARTHIK NAGARAJAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003

Copyright 2002

by

Karthik Nagarajan

I dedicate this to my family

ACKNOWLEDGMENTS

I would like to take this opportunity to offer my highest gratitude to Dr. Herman Lam, chairman of my supervisory committee, for giving me the guidance and motivation to complete this thesis. I also sincerely appreciate Dr. Stanley Y. W. Su, my supervisory committee member, for his valuable comments and suggestions during my thesis work. I would also like to thank Dr. Joachim Hammer, my supervisory committee member, for his precious time. I feel proud to have people of such stature involved in my work.

I also sincerely thank Ms. Sharon Grant for making the Database Center a pleasant place to work. I would like to thank Mr. John Bowers and Ms. Nisi Caudle for continuously guiding me through my graduate studies.

I am also very thankful to my mother, Mrs. Padmaja Nagarajan, for giving me the opportunity to pursue graduate studies at such an esteemed institution. I thank my sisters, Mrs. Sowmya Kannan and Ms. Nithya Nagarajan, for their continuous encouragement.

My sincere thanks also go to Raman Chikkamagalur and Althea Liang for sharing their valuable knowledge with me.

TABLE OF CONTENTS

page

LIST OF FIGURES

Figure page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

INTEGRATION OF BUSINESS EVENTS AND RULES MANAGEMENT WITH THE WEB SERVICES MODEL

By

Karthik Nagarajan

May 2003

Chair: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

Web services technology is designed to support interactions among applications in such a way as to better support the sharing of application functionality and business processes among organizations that conduct collaborative e-business over the Internet. Web services are network accessible applications and business processes published by business organizations that can be used by other organizations to develop distributed applications. While the Web services technology provides a promising foundation to describe, publish, find, bind and invoke Web services, much research is needed to make the Web services paradigm viable to fully support the needs of real e-business and other collaborative integration endeavors. In particular, interactions among business organizations need to follow the policies, regulations, security rules and other types of business rules of the organizations.

This thesis focuses on incorporating the business event and rule-management concepts and techniques into the Web services model at the service provider side. Based on a code-generation approach, we have developed techniques and tools to generate Web service "wrappers" and other objects required to integrate ETR technology with Web services technology. Using a GUI, a service provider is allowed to choose the operations he wishes to expose along with the events (such as before operation invocation and/or after operation invocation) he wishes to install. Based on the inputs, a Web service wrapper is generated that implements the posting of the installed events. Also, the interface and implementation WSDL documents of the selected Web service are generated and placed in the appropriate directory. The Web service wrapper is then deployed as the actual Web service. At run time, when a client invokes an operation in the Web service, the appropriate method in the wrapper is invoked to post the appropriate events to trigger any required business rules, in addition to invoking the service implementation.

CHAPTER 1
INTRODUCTION

In today's shrinking world, effective communication and interaction among businesses are essential for the success of any business enterprise. The Internet has revolutionized the way business enterprises communicate and interact. To remain competitive in this e-world, a business enterprise needs to apply all available Web and information technologies to its own business operations and processes. E-business is rapidly emerging as a major application of Internet based distributed computing, allowing multiple organizations to collaborate with each other and thus leading to the automation of key processes.

Existing distributed objects technologies have been found to be inadequate in supporting the needs of e-business applications that are distributed, heterogeneous *and dynamic.* Tightly coupled and statically bound integration of applications using distributed objects technologies demands too much dependency among collaborating organizations. This results in complexity of applications integration, brittleness in their implementation and difficulty in making changes. The problems get aggravated as the scale of e-business increases.

The current Web services technology is emerging as a promising infrastructure to support loosely coupled, Internet-based applications that are distributed, heterogeneous, and dynamic. It provides a standards-based, process-centric framework for achieving sharing of distributed heterogeneous applications. The Web services model provides dynamic binding to services and allows larger granules of application system

functionalities and business processes to be shared over the Internet. Due to the encapsulation of Web services and use of standards for their description, discovery and integration, Web services can be accessed independently of the communication mechanisms, programming languages and frameworks used to implement them. This loosely coupled and dynamic nature of Web services simplifies the development of new e-business applications and business processes and promotes interoperability by minimizing the dependency among collaborating organizations.

While the emerging Web services technology provides a promising foundation for developing distributed applications for e-business, additional features are required to make this paradigm truly useful in the real world. In particular, interactions among business organizations need to follow the policies, regulations, security and other business rules of the organizations. An effective way to control, restrict and enforce business rules in the use of Web services is to integrate business event and rule management concepts and techniques into the Web services model. Things of importance that can happen within or outside of collaborating business organizations can be defined as events. For example, operations that change the data states of a system, actions taken by users through a browser, before and/or after invocation of a Web service, can be events of interest to business organizations. In a business environment, the occurrence of an event may require the invocation of some business rules to enforce security and integrity constraints, policies and regulations, or enact a business process.

An ongoing research project at the Database Systems Research and Development Center at the University of Florida explores the integration of business events and rules management with the Web services model through the use of an Event-Trigger-Rule

(ETR) technology. The purpose is to enhance the Web services model by using concepts and techniques of events, event filters, event notifications and business rule processing to control, monitor and restrict the publication, discovery and access of Web services.

1.1 Focus of this Thesis

This thesis focuses on incorporating the business event and rule management concepts and techniques into the Web services model at the service provider side. Based on a code-generation approach, we have developed techniques and tools to generate Web service "wrappers" and other objects required to integrate the ETR technology with the Web services technology on the service provider side. The build-time tool used in the system consists of four main components: GUI for Web Service Creation, Wrapper Generator, WSDL Document Generator and the Events Installer.

Using the GUI, a service provider is allowed to choose the operations he wishes to expose along with the events (such as before operation invocation and/or after operation invocation) he wishes to install on an ETR server. Based on the inputs, a Web service wrapper is generated that implements the posting of the installed events. Also, the interface and implementation WSDL documents of the selected Web service are generated and placed in the appropriate directory. The Web service wrapper is then deployed as the actual Web service. The wrapper provides an interface identical to that of the underlying service implementation. At run time, when a client invokes an operation in the Web service, the appropriate method in the wrapper is invoked to post the appropriate events to trigger any required business rules, in addition to invoking the service implementation.

The rest of this thesis is organized as follows. Chapter 2 provides a survey of the enabling standards and technologies. Chapter 3 describes the general architecture of the

ETR-enabled Web services model. The remainder of the chapters details the system components required to support ETR-enabled Web services at the provider side. Chapter 4 describes the *Graphical User Interface used for Web service creation* and Chapter 5 describes the *Wrapper Generator, WSDL Document Generator and Events Installer* components. Finally, Chapter 6 provides a summary and conclusion for the thesis.

CHAPTER 2
SURVEY OF ENABLING TECHNOLOGIES

This chapter provides a survey of the standards, tools and software that have been used in developing the ETR-enabled Web services technology. Section 2.1 gives an overview of the existing Web services technology and its advantages. Section 2.2 provides an outline of the Web Services Description Language (WSDL) used to describe Web services. Section 2.3 describes the Simple Object Access Protocol (SOAP) and its use as an XML based protocol for exchange of information in a distributed environment. Section 2.4 briefly describes the Apache Axis Toolkit that provides a framework for constructing SOAP processors and implementing Web services. A review of the Event-Trigger-Rule technology used in this thesis will be given in Chapter 3.

2.1 Web Services Technology

The Web services technology could be seen as a platform to support distributed computing over the Web. At the core of the Web services model is the notion of a "Service," which is defined as a collection of operations that carry out some type of a task. A Web service is defined as an interface that describes a collection of operations that are network-accessible through standardized XML messaging. As described by Gottschalk (2000), a Web service is described using a standard, formal XML notation called its service description. It specifies all the details necessary to interact with the service, including message formats, transport protocols and location. The interface hides the implementation details of the service, allowing it to be used independently of the hardware and software platform on which it is implemented and also independently of the

programming language in which it is written. This allows Web services based applications to be loosely coupled, component oriented and cross-technology implementations. Thus, Web services can be used alone or with other Web services to carry out a complex aggregation of tasks in a business transaction.

Shown in Figure 2-1 is a general Web services model. It defines three roles: Service Provider, Service Requestor and Service Registry. Web services are implemented and published by Service Providers. From a business perspective, this is the owner of the service and from an architectural perspective, this is the platform that is used to host access to the service. A Service Requestor is the entity that discovers and invokes a Web service. From a business perspective, a Service Requestor is the business that requires certain functions to be performed and from an application perspective, this is the application that is looking for and initiating an interaction with a service. The Service Registry is a searchable registry of service descriptions where service providers publish their service descriptions, which is used by service requestors to find the services and obtain their binding information.

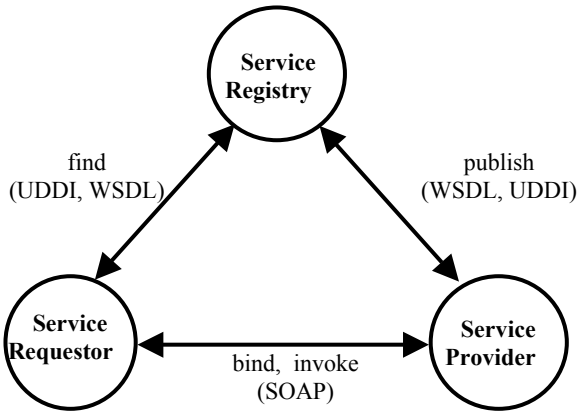


Figure 2-1 Web services model

The three fundamental operations shown in Figure 2-1 can be described as follows

- **Publish** – performed by the Service Provider to advertise the existence and capabilities of a service.
- **Find** – performed by the Service Requestor to locate a service that meets the needs at hand.
- **Bind** – performed by the Service Requestor to invoke the service being provided by the Service Provider.

The Web services architecture is implemented through four types of technologies organized into layers that build upon one another as shown in Figure 2-2. The XML messaging, description and discovery layers in the Web services stack are the layers essential for providing just-in-time integration capabilities and the much needed platform neutral programming model.

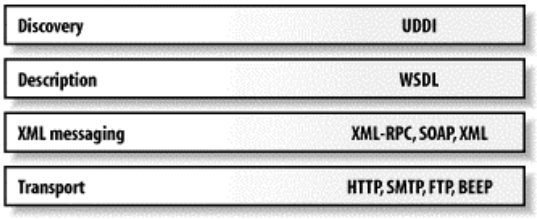


Figure 2-2 Web services stack

The Discovery layer provides the mechanism for clients to fetch the description of providers. One of the most widely recognized discovery mechanisms used is the Universal Description, Discovery and Integration (UDDI) project. Bellwood et al. (2002) provide a detailed specification for the UDDI project. This is used almost ubiquitously. However, recently IBM and Microsoft have jointly proposed an alternative to UDDI, called the Web Services Inspection language (WSIL). WSIL is described by Ballinger et al. (2001) and Nagy and Ballinger (2001).

When a Web service is implemented, it must make decisions at each level about the network, transport and packaging protocols it will support. This Description is used by the service requestor to contact and invoke the service. The Web Services Description Language (WSDL) is a standard for providing these descriptions.

For application data to be moved around in a network (the transport layer), it must be packaged in a format understood by all parties. This process is otherwise called marshalling or serialization. XML provides a foundation for message communication among most present day Web services. It provides a means to represent the meaning of the data being transferred. Furthermore, XML parsers are now ubiquitous. Simple Object Access Protocol (SOAP) is a common packaging format built on XML and is widely used for communication in the Web services model. The transport layer maps to the TCP and IP protocols of the HTTP protocol.

2.2 Web Services Description Language

The Web Services Description Language (WSDL) is a specification language defining how to describe Web services in a common XML grammar. As described in the WSDL specification provided by Christensen et al. (2001), submitted to W3C, WSDL describes four critical pieces of data:

- Interface information describing all publicly available methods.
- Data type information for all message requests and message responses.
- Binding information about the transport protocol to be used.
- End point address information for locating the specified service.

WSDL essentially represents a contract between the service requestor and the service provider that provides all the information a service requestor needs to invoke a Web service. It provides a common language for describing Web services and a platform

for automatically integrating those services. The main advantage of using WSDL is its platform and language independence.

There are several tools available that generate the WSDL documents for a service given the service application. Java2WSDL is one such tool that generates interface and implementation WSDL documents for a Web service given the Java class corresponding to the application. The Java2WSDL tool also allows us to specify the methods that we wish to expose as Web service operations and the mode of invocation of these operations namely – One way, Request Response, Solicit Response and Notification. In this thesis we make use of the Java2WSDL tool provided in the Apache Axis toolkit.

2.3 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) is a simple, lightweight XML based protocol that uses HTTP for exchange of information in a distributed environment. It defines formats for messages exchanged between distributed applications. SOAP is language and platform independent because it is based on XML. It provides a means for applications running on different operating systems, with different technologies and programming languages to communicate with each other. In SOAP, everything that goes across the wire is expressed in terms of HTTP or SMTP headers, MIME encoding and special XML grammar as defined by the SOAP specification. Box et al. (2000) describe the SOAP specification in detail.

A SOAP message consists of the following parts:

- A SOAP envelope that defines the content of the message
- An optional SOAP header that contains header information
- A SOAP body that contains call and response information

The SOAP envelope declaration is the outermost XML tag that delineates the boundaries of the SOAP document. The SOAP header and body elements are syntactically similar. The SOAP header contains the <Header> element, which is used for things targeted at the underlying infrastructure, such as transaction ID, where transaction ID is not part of the method signature. It is intended for the SOAP processor that receives the message, which may be a J2EE server with a transaction manager. The SOAP body is intended for the actual data, or message payload, to be consumed and processed by the ultimate receiver. The <Body> element is reserved purely for the method call and its parameters.

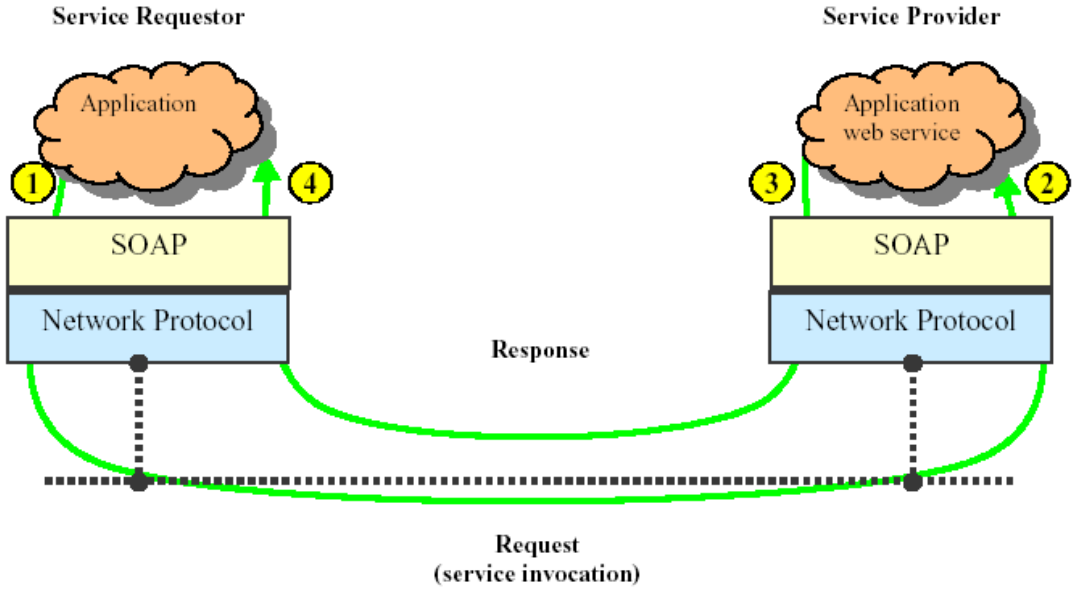


Figure 2-3 XML messaging using SOAP

Figure 2-3 shows how XML messaging and network protocols form the basis of the Web services architecture. XML messaging using SOAP is a four-step process.

1. Client-side application at the service requestor side makes the service invocation request. The SOAP engine generates the SOAP message for this request and the Network protocol takes care of converting this SOAP message to the appropriate format and routes it to the Web service provider.

2. At the Web service provider side the SOAP engine identifies the service request and invokes the appropriate method in the server-side application.

3. The server-side application returns the results of the invocation to the SOAP engine and the results are routed back to the client in the form of a Web service response.

4. The results of service invocation are returned to the client-side application.

2.4 Axis Toolkit

Axis is a Web services toolkit developed by Apache. The Axis User's Guide describes the usage of this toolkit. It is a SOAP engine that provides a framework for constructing SOAP processors such as clients, servers and gateways. The existing version of Axis is written using Java. Axis provides many features that allow it to act as a server that plugs into a servlet engine such as Tomcat. Among other things, it provides the Java2WSDL tool that is used to generate WSDL documents from a Java class. We will use this tool to generate interface and implementation WSDL documents for the Web service wrappers. It also provides a tool for monitoring TCP/IP packets that allow us to view the structure of SOAP messages exchanged between applications.

As described by Snell (2002), Axis has features that allow deployment of Web services in a flexible way by making use of a Web Services Deployment Descriptor (WSDD). The Axis API, org.apache.axis.client.AdminClient is used to deploy and undeploy a Web service. A deployment descriptor contains the details that the Axis engine needs to know to deploy a Java class as a Web service. It contains details such as the Java class name for a given service, mapping of QName (qualified name) information to Java classes for the purpose of serialization and deserialization. Axis also provides the API, org.apache.axis.client.Call that is used to invoke a Web service operation. The API

encapsulates generation and decoding of SOAP messages and provides a high-level interface to invoke the Web service operations.

The Axis user's guide gives a good description for using the Axis toolkit. The Axis Architecture Guide explains the overall architecture and the individual components of the toolkit.

CHAPTER 3
ARCHITECTURE OF THE ETR-ENABLED WEB SERVICES MODEL

The Web services technology provides a framework to realize better sharing of distributed heterogeneous applications. Their flexibility lies in the fact that their access does not depend on the communication mechanisms, programming languages and frameworks used to implement them. Thus, they provide a loosely coupled infrastructure to develop new e-business applications that ease interoperability among collaborating organizations. However, to effectively use Web services to conduct business, interactions between business organizations need to follow the policies, regulations, security and other business rules of the organizations. An effective way to control, restrict and monitor the access of Web services is to integrate business event and rule management concepts and techniques into the Web services model. Things of importance that can happen within or outside of collaborating business organizations can be defined as events. For example, operations that change the data states of a system, actions taken by users through a browser, before and/or after invocation of a Web service etc., can be events of interest to business organizations. In a business environment, the occurrence of an event may require the invocation of some business rules to enforce security and integrity constraints, policies and regulations, or enact a business process. This chapter describes the general architecture of a system that integrates event and rule management concepts with the Web services model.

Section 3.1 provides a description of the Event-Trigger-Rule (ETR) technology. Section 3.2 describes the architecture of the ETR-enabled Web services model that

integrates the ETR technology into the Web services model. Section 3.3 outlines a wrapper generation approach to support the ETR-enabled Web services model.

3.1 Event-Trigger-Rule Technology

The Event-Trigger-Rule (ETR) paradigm described by Lam and Su (1998) is a generalization of the Event-Condition-Action (ECA) paradigm. Unlike the ECA paradigm, the ETR paradigm separates event and rule specifications and uses trigger specifications to relate events with rule structures, as shown in Figure 3-1. Events can be "triggering event" or events that participate in a composite event expression (or event history) in our trigger specifications. This separation is important because in some real world situations, only some events appearing in a composite event (or event history) specification should trigger the evaluation of composite event expression and the firing of rules. Triggers are specifications that relate events with rule structures, making it possible to fire structured rules upon the occurrences of events. When a triggering event occurs, the corresponding triggers are activated for processing. During the processing of a trigger, the event history is evaluated. If it evaluates to true, then the corresponding rules are fired. Each rule specifies some condition (C) that needs to be verified to determine the execution of a structure of operations (A) or an alternative structure of operations (A). A structure of CAA rules explicitly specifies a large granule of control and logic. A single CAA rule can participate in multiple rule structures, thus making each rule reusable in building a larger granule of business control and logic. Events, triggers, and rules are now described in more detail.

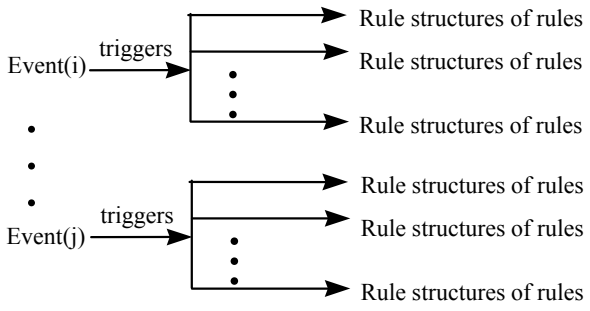


Figure 3-1 ETR paradigm

3.1.1 Events

An event is an occurrence of anything of interest to people or software systems. It can be the reading or updating of a data field or record, or a failure signal issued by the software that controls a disk, or a sudden drop of inventory of a certain product, or the signal of a timer that keeps track of the available time, etc. Events can be broadly categorized into three types: method-associated events, explicitly posted events, and timer events.

A method-associated event is raised when a specific method executes. The raising of an event can be done either before the method, after the method, or at the commit time of a transaction, which contains the method execution. These different times of posting events with respect to the method execution time are called coupling modes. The three coupling modes just described are called before, after, and on-commit. These three coupling modes are synchronous. When a synchronous event is posted, the execution of the event posting method/program is suspended until a response is returned from the rule processor. If an event is posted asynchronously, the event posting method/program does not wait for the response but continues its execution. Two other coupling modes that are useful are instead-of and decoupled. The instead-of mode allows the rule execution to replace the method body under a certain condition. That is, the rule execution will be

carried out instead of the method execution when a certain condition is satisfied. The decoupled mode allows an event to be posted asynchronously.

An explicitly posted event can be raised independent of any method execution. That is, the event is not tied to any specific method and can be raised in the body of any desired method via a 'PostSynchEvent' or 'PostAsynchEvent' call using an event instance as the parameter of the call. An explicitly posted event can be posted synchronously or asynchronously.

A timer event is an event that is related to some predefined time of interest. It is raised when the predefined time of interest has come. The timer event is asynchronous by nature.

In this work, we are mainly interested in method-associated events. We will apply this concept within the Web services model to define such events for the operations of a Web service.

3.1.2 Rules

A rule is a high-level declarative specification of a granule of executable code that can respond to an event or events. A rule is composed of a condition, action, and alternative action clauses. A single or multiple rules form a rule structure. In the ETR paradigm, when an event is posted, the rule structures that are associated with the event can be triggered for processing. When a rule is processed, the condition clause of the rule is first evaluated. If the condition is True, the statements in the action clause are executed. Otherwise, the statements in the alternate action clause are executed. A rule has an interface that specifies what parameters are used in the rule body (i.e., condition, action, alternative action). The actual values of these parameters are provided by the event at run time. Here is an example syntax of a rule:

```
RULE            rule_name (parameter list)
[RETURNS        return_type]
[DESCRIPTION    description_text]
[TYPE           DYNAMIC/STATIC]
[STATE          ACTIVE/SUSPENDED]
[RULEVAR        rule variable declarations]
[CONDITION      guarded expression]
[ACTION         operation block]
[ALTACTION      operation block]
[EXCEPTION      exception & exception handler block]
```

In the rule syntax, clauses that are surrounded by brackets are optional. A rule can return a value whose type is indicated by the RETURNS clause. A general description of the rule can be recorded in the DESCRIPTION clause. The TYPE clause indicates whether or not the rule is going to be modified after the initial definition. A DYNAMIC rule indicates that the rule may be changed at run-time; whereas, a STATIC rule means that the rule is less likely to be changed. This information is used for optimizing the performance of the rule by generating the proper form of rule code. The STATE clause indicates whether the rule will be initially active or suspended after definition. If the rule is initially suspended after definition, it will not get triggered until it is activated. The RULEVAR clause has the declaration of the variables that are used in the rule body. The CONDITION clause is composed of a guarded expression. A guarded expression has two parts: a guard part and a condition expression part. The guard part is composed of a sequence of expressions. If any expression in the guard evaluates to false, the rule is

skipped. Otherwise, the condition expression part is evaluated. The guard part is provided to screen out cases where the rule must be skipped, such as error situations or invalid data values, etc. Depending on the result of the condition expression evaluation, the ACTION clause or the ALTACTION clause is executed. ACTION and ALTACTION clauses are composed of statements to be executed, such as method calls or assignment statements, statements that post events, etc. During the execution of a rule, an exception may occur. This is handled by the EXCEPTION clause where the exception type is paired with an exception handler.

3.1.3 Triggers

A trigger specifies which event(s) triggers the processing of a rule structure. It also can support composite events and does the parameter mapping between the event parameters and rule parameters. An important functionality of the trigger is the capability to specify parallel rule executions. The constructs of a trigger specification are given as follows:

TRIGGER	trigger name (trigger parameters)
TRIGGEREVENT	events connected by OR
[EVENTHISTORY	event expression]
RULESTRUC	structure of rules
[RETURNS	return_type : rule_in_rulestruct]

The clauses that are surrounded by brackets are optional. The TRIGGER clause specifies the name of a trigger and the trigger parameters. The trigger parameters are used to bridge between the event parameters and the rule parameters. The TRIGGEREVENT clause specifies the events that can trigger the structure of rules specified in the RULESTRUC clause. Several events can be OR-ed (i.e., connected with a disjunctive

operator), which means that the occurrence of any one of the events can trigger the rules. The EVENTHISTORY allows for checking past event occurrences that form a composite event. The RULESTRUC clause specifies the set of rules to be executed and also in what order (or structure) that the rules should be executed. The RETURNS clause is optional, and is used when the trigger needs to return a value in the case of a synchronous event that triggers the rules. The return type and the specific rule that should provide the return value are specified in this clause.

A rule structure is formed by a composition of four basic constructs: sequential, parallel, AND-synchronized, and OR-synchronized as shown graphically below:

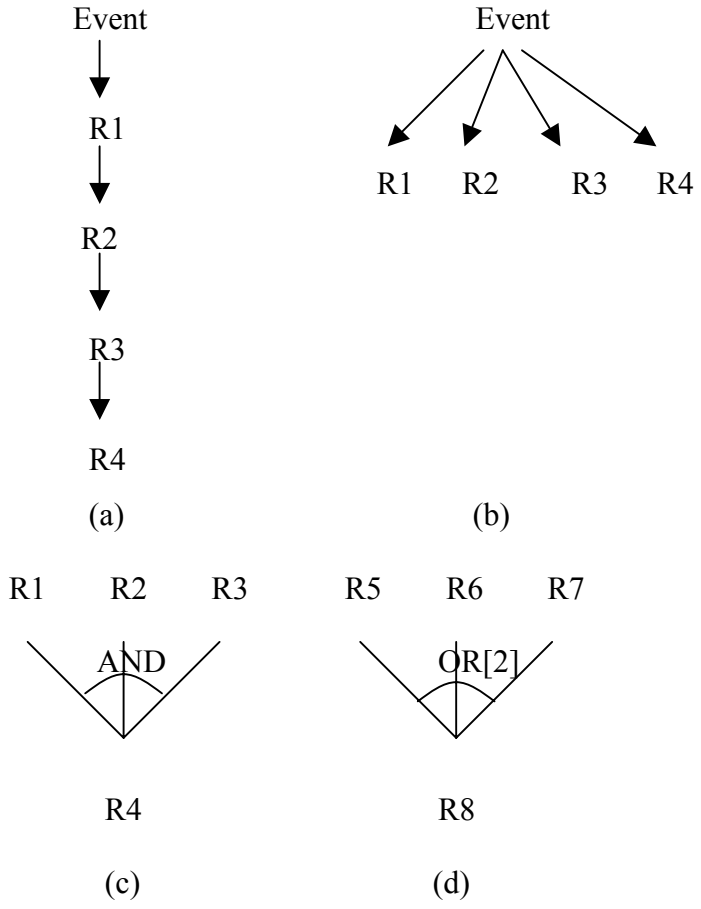


Figure 3-2 Various rule execution structures

Figures 3-2 (a), (b), (c) and (d) show the sequential, parallel, AND-synchronized and OR-synchronized rule structures. The sequential rule execution of R1-R4 is expressed in a rule specification language by (R1 > R2 > R3 > R4), the parallel rule execution by (R1, R2, R3, R4), the AND-synchronized execution by AND (R1, R2, R3) > R4, and the OR-synchronized execution by OR[2] (R5, R6, R7) > R8. Here, OR[2] means that the completion of any two of the three rules R5-R7 can activate R8. By nesting these four constructs (e.g., AND (R4, R5, R6, OR[2] (R1,R2,R3)) > R7), a more complex structure of rules can be specified.

3.2 ETR-Enabled Web Services Model

The general Web services model was explained in Chapter 2 and illustrated in Figure 2-1. For the reader's convenience, Figure 2-1 is reproduced as Figure 3-4.

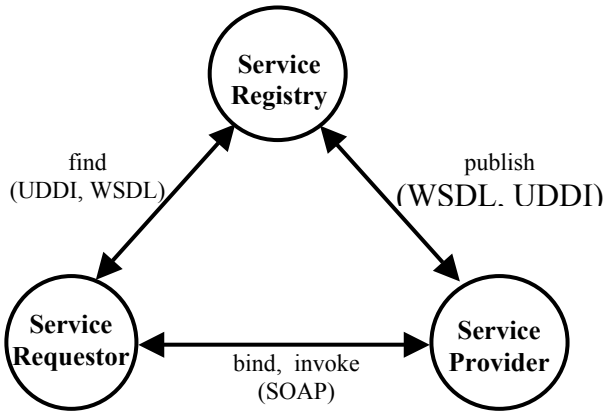


Figure 3-3 Web services model

As shown in Figure 3-3, the Service Requestor queries the Service Registry to locate a Service Provider. Once it has found an appropriate Service Provider it binds to it and invokes the necessary operations. Figure 3-4 shows the interactions between the Service Requestor and Service Provider in greater detail. The interactions are shown by the paths labeled 1 through 6.

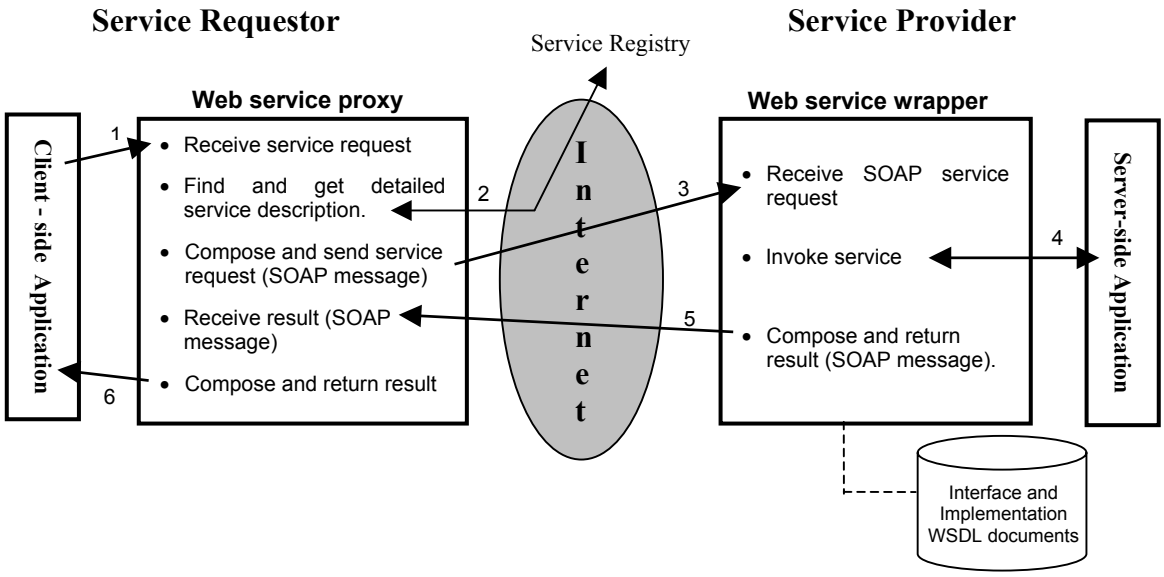


Figure 3-4 Interactions between a service requestor and a service provider

1. **Client makes Web service request.** The client can make a Web service request by making a call to a Web service proxy. Note that the proxy may be generated automatically based on a service request specification using a Web services toolkit.

2. **Service discovery.** The Web service proxy contacts a broker (that implements a registry) to find an appropriate service provider to execute the service request. The proxy then gets the address of the Web service provider from the broker.

3. **Compose and send SOAP message.** Once the appropriate service provider has been located, the Web service proxy accesses the interface and implementation WSDL documents and composes and sends a SOAP message that contains the service request. Note that the SOAP message request is sent to a Web service wrapper that encapsulates the actual Web service.

4. **Wrapper invokes Service.** The Web service wrapper on the service provider side performs the necessary tasks to interpret the SOAP message request and invokes the actual service that contains the business logic for the operation.

5. **Return result (server side).** The Web service wrapper then composes a SOAP message containing the results of the method invocation and returns it to the client side.

6. **Return result (client side).** The Web service proxy returns the result to the actual client application.

As discussed earlier, interactions among business organizations need to follow their organizations' policies, regulations, security and privacy rules. We believe an effective way to enforce these rules in the context of the Web services model is to integrate business event, trigger and rule (ETR) management technologies with the Web services model. Figure 3-5 illustrates how the ETR technology is used to realize an ETR-Enabled Web services model.

In Figure 3-5, a client application wishes to invoke a remote Web service. It issues a Web service request by making a call to a Web service proxy (label 1 in Figure 3-5). The general course of actions (find, bind, invoke and return result) is shown by the path labeled 2, 3, 4, 5 and 6. The ETR technology is incorporated at the points labeled 1a, 3a, 4a and 5a. Synchronous and/or asynchronous events can be posted at these points to trigger business rules stored in the ETR server. For example, on the Service Requestor side, upon receiving the request the service proxy may post Before Request (BR) event(s) (labeled 1a in the figure) before the service proxy contacts a broker to find the appropriate Web service provider (label 2). The BR event can trigger some global (mutually agreed upon) or local business or security rules.

At the Service Provider side the Web service wrapper receives the request from the client. The code for the Web service wrapper is generated using the Wrapper Generation tool to be described in Chapter 5. Events can be injected at points 3a and 4a during the code generation process so that they can be posted at runtime to trigger the execution of business rules. For example on the service provider side after receiving the service request, a Before Invocation (BI) event can be posted to trigger any global or local rules (label 3a) before the actual service is invoked (label 4).

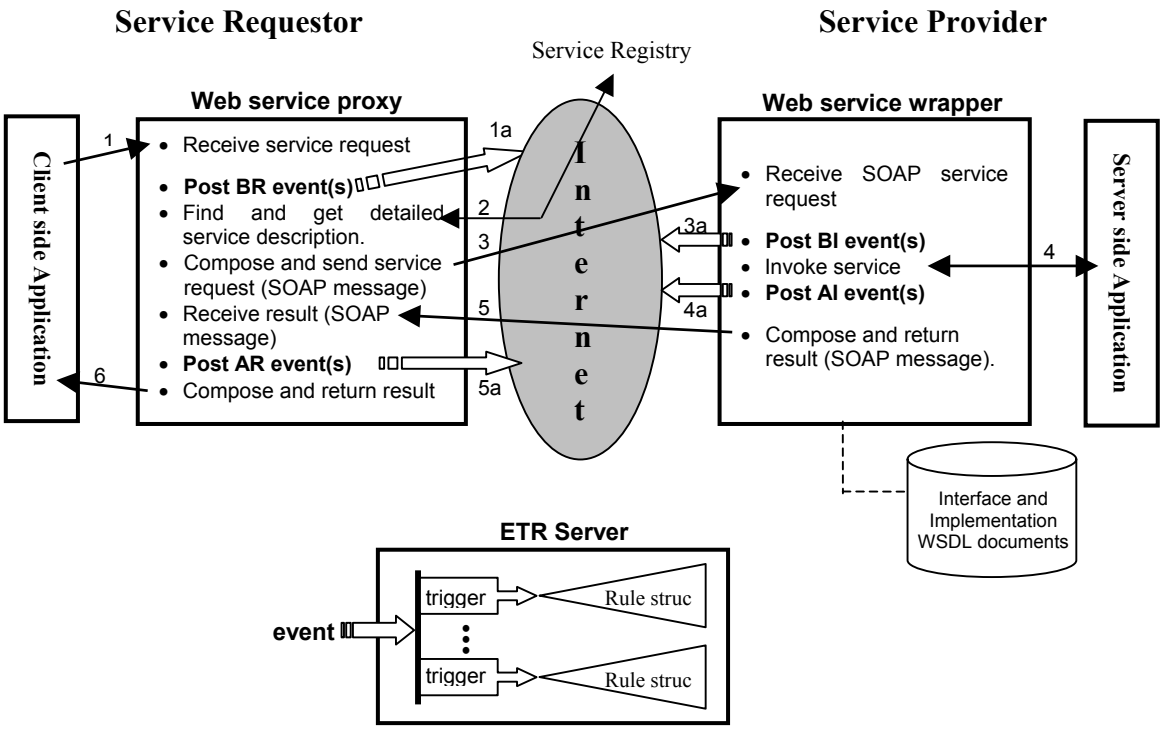


Figure 3-5 Integration of business events/rules with Web service invocation

The service request may be modified to enforce some business or security rules. After the execution of the service, an After Invocation (AI) event can be posted to trigger any necessary rules (label 4a). For example, privacy rules can be enforced by performing post-processing or data filtering on the result to be returned by the provider side application (label 5). Back on the requestor side, the Web service proxy can post an After Request (AR) event to trigger some other rules (label 5a) before returning the result to the client application (label 6).

These interactions between the Service Requestor and Provider shown in Figure 3-5 illustrate several key concepts:

- E-businesses are collaborative. Collaborative e-businesses have mutually agreed upon businesses rules and regulations. These global rules need to be enforced during the request and use of Web services. The ETR technology provides a flexible mechanism to insert and enforce these rules at different points within the Web services model. Control and logic specified by rules will not be hard-coded

in application programs or agents. Rules can be more easily changed and understood.

- E-businesses are loosely coupled. Although collaborative, the organizations involved in e-business are loosely coupled. Their autonomy needs to be maintained. Thus, local business rules have to be enforced. The ETR technology provides a flexible (and locally controlled) mechanism to insert and enforce these rules within the Web services model.

- E-businesses (and business rules) are dynamic. The ETR technology provides a very dynamic and flexible mechanism to specify and enforce business rules. Rules can be added, deleted, or modified at runtime without any effect on the Web service interface or the Web service implementation.

3.3 Wrapper Generation Approach to Support ETR-Enabled Web Services Model

The objective of this thesis is to develop the techniques and tools to generate the Web service wrapper and other objects required to integrate the ETR technology with the Web services technology on the service provider side of the Web services model. The build-time tools used in the system are shown in Figure 3-6. It consists of four main components: GUI for Web Service Creation, Wrapper Generator, WSDL Document Generator and the Events Installer.

The following briefly describes each component involved in the system.

1. **GUI for Web service creation.** The GUI for Web Service creation is essentially what the user sees. This wizard-like GUI accepts information regarding the server side application that is to be made into a Web service. From this application, which is in the form of a Java class, the user chooses the methods to be exposed as Web service operations. For each operation the user can define before and/or after operation events. The GUI then transforms this information into more useful forms for use by the Wrapper Generator, WSDL Document Generator and the Events Installer. The GUI also monitors the progress of these activities and reports it to the user in the form of a progress bar.

2. **Wrapper generator.** This component is responsible for generating the wrapper code for the server-side application. The Wrapper Generator takes as input the Web service interface information and the events to be posted for each of the operations. For each exposed operation of the Web service, code is generated for the posting of before and/or after operation events (if any) to the ETR server and the invoking of the appropriate method in the underlying server side application. This wrapper is then deployed as the actual Web service.

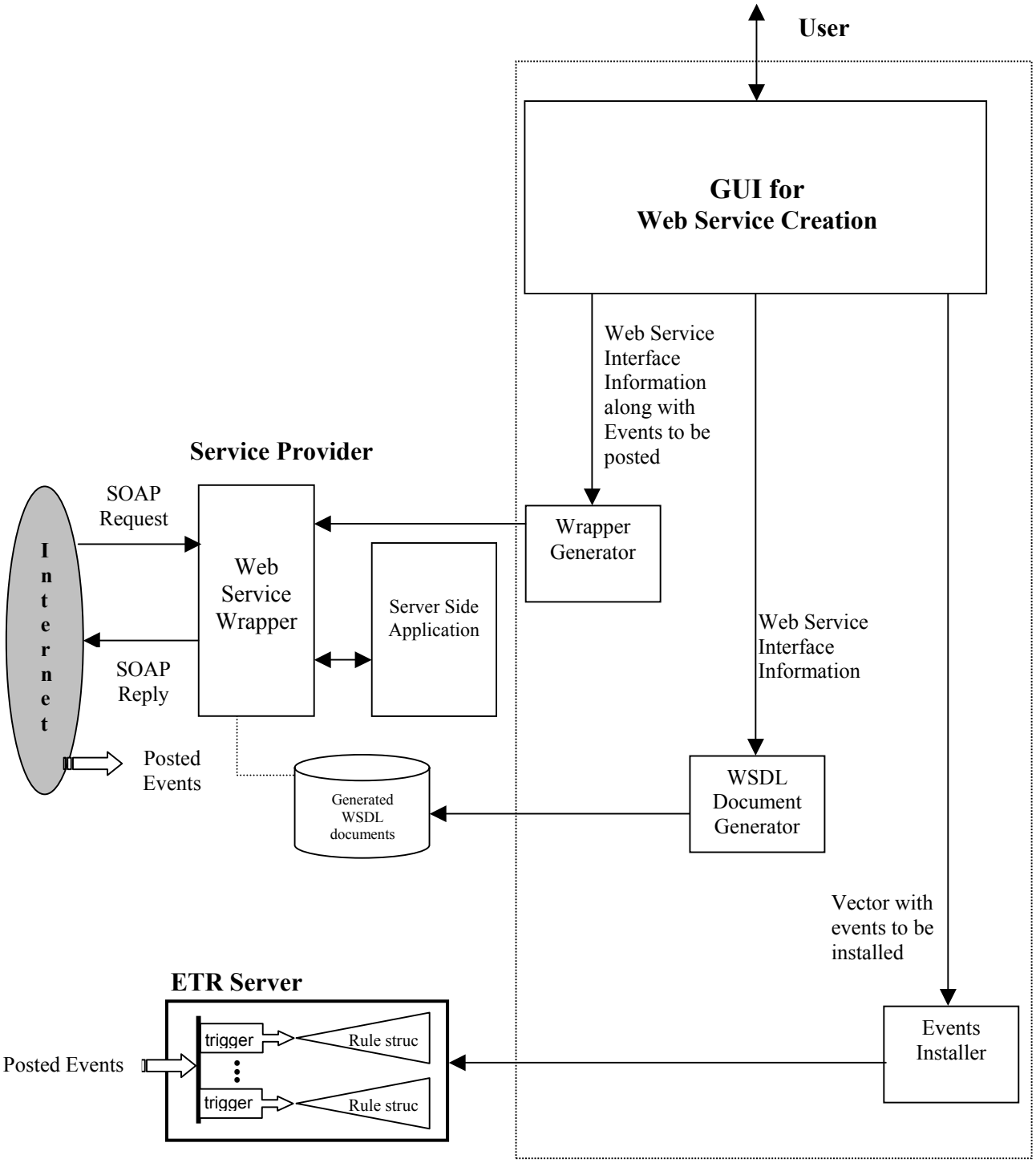


Figure 3-6 Build-time tools used to support the ETR-enabled Web service model

3. **WSDL document generator.** The essential input to this component is the Web service interface information, i.e. the methods that are to be exposed as Web service operations. Other inputs include the names and paths of the output WSDL documents, target namespaces to be used etc. This component makes use of the Java2WSDL tool provided by the Axis toolkit to create WSDL documents from a

Java class. The tool introspects the Java class and then generates the interface and implementation WSDL documents. These documents are then stored persistently to be used at runtime by Web service clients.

4. **Events installer.** For each event defined by the user, the Events Installer component contacts an Events Installation service to create and install the event types on the ETR server. The event types are thus stored persistently in the ETR server and are used by the ETR server at run time to process incoming events. The parameters of the event types created by the Events Installer are same as the parameters of the associated methods of the server-side application. This allows the inputs of the methods to be used in decisions made during the enforcement of business rules.

The details of these four components will be described in the remainder of this thesis. In Chapter 4, we discuss the design and implementation of the GUI for Web service creation. Chapter 5 deals with the design and implementation of the Wrapper Generator, WSDL Document Generator and the Events Installer components.

CHAPTER 4
GRAPHICAL USER INTERFACE FOR WEB SERVICE CREATION

This chapter describes the design and implementation of the Graphical User Interface (GUI) for Web service creation. As shown in the architecture diagram in Figure 3-7, the user interacts with the Graphical User Interface to specify information required to produce ETR-Enabled Web services. For each Java class, the user selects the methods to be exposed, the events to be tracked, etc. This wizard-like GUI accepts all the information and invokes the other components to generate the WSDL documents, create and install the event types in the ETR server and generate the Web service wrapper. As shown in Figure 4-1, the use of the GUI involves 3 distinct phases, namely the Service definition phase, Operations selection and events definition phase, and the Generation phase. Each of these phases is explained in a separate section.

To show the use of the GUI, an example Web service called the Gator directory service is used. The Gator directory service is a typical address book service that maintains a mapping of names of people to their address information. Its interface includes typical operations like getAddressFromName, addEntry, removeEntry, getAllListings etc. Let us assume that this has been implemented in Java. This would serve as our server-side application that we wish to provide as a Web service. We will use the GUI to convert this server-side application Java class into a full-fledged ETR-Enabled Web service capable of tracking specific events for each of its exposed operations.

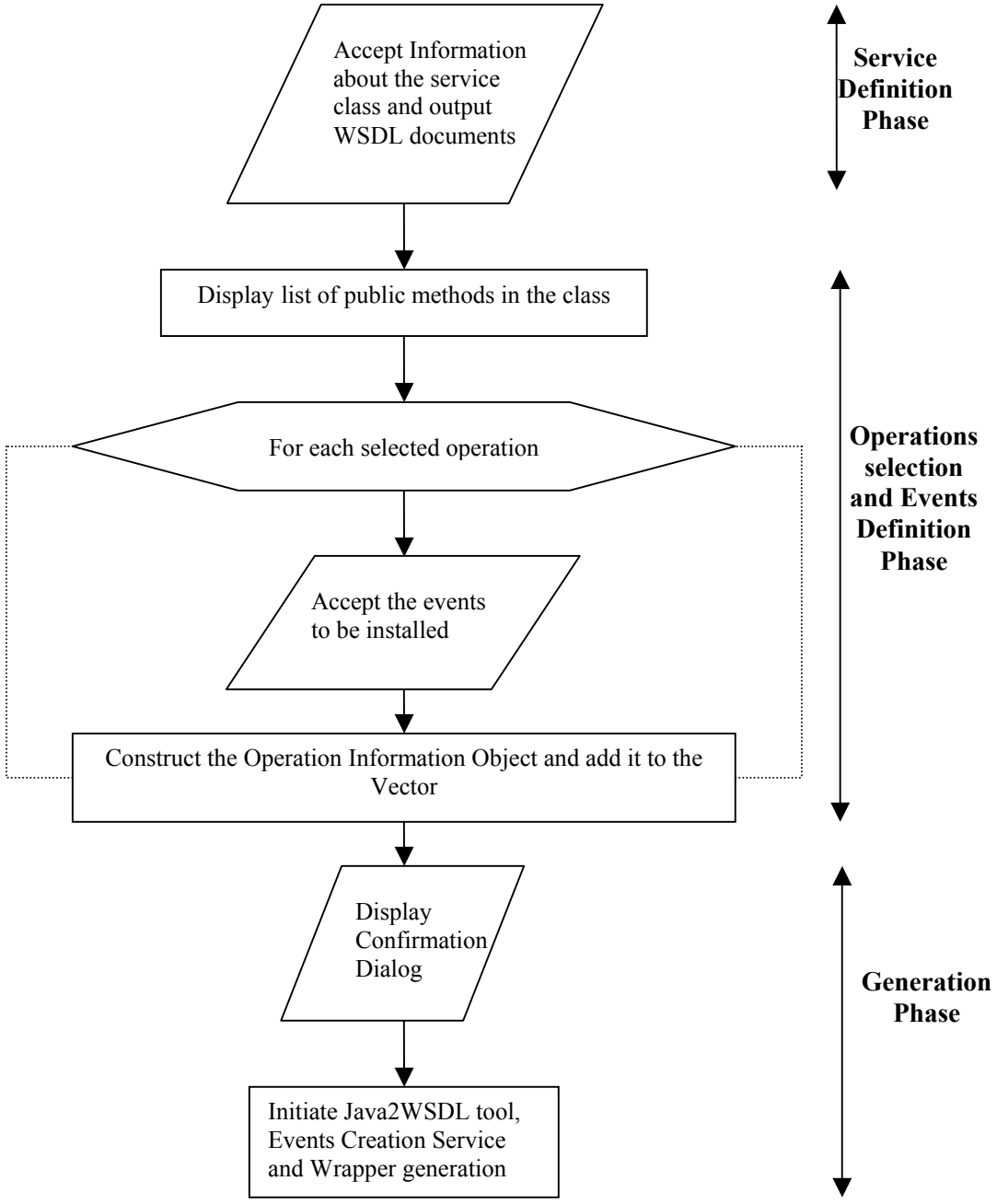


Figure 4-1 Design of the Graphical User Interface

4.1 Service Definition Phase

In the Service definition phase, the user specifies the information about the service class and the WSDL documents to be outputted. A screen shot of the GUI used

for this phase is shown in figure 4-2. The GUI is implemented using Java Swing graphical toolkit.

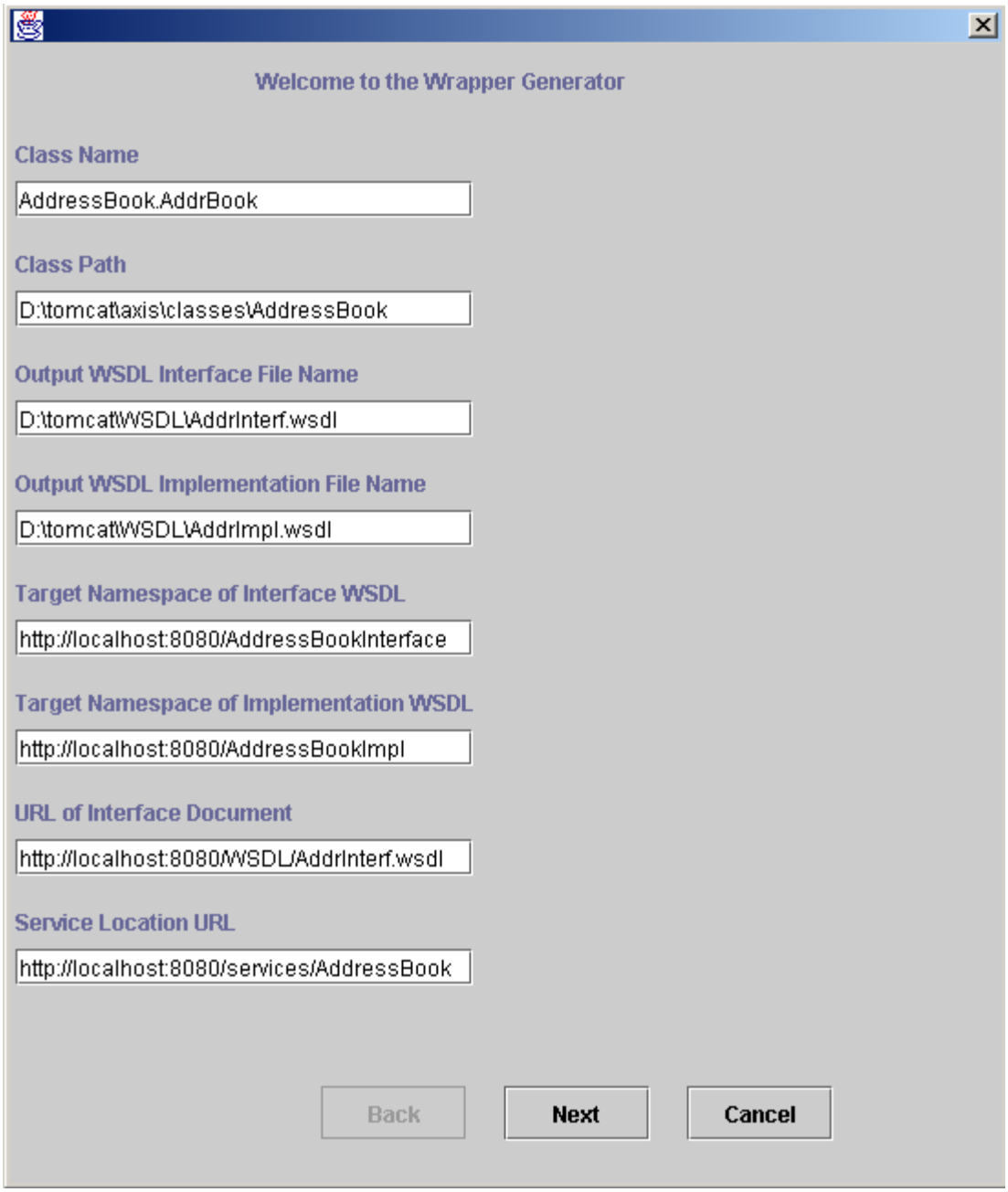


Figure 4-2 Service definition phase

In the Service definition phase, the service provider enters the following information:

- **Class name, Class path.** The name of the Java class to be made into a Web service and its physical location.

- **Output WSDL interface file name.** In order to make a Java class into a Web service, a WSDL interface document needs to be created. This document specifies the interface information such as the Web service operations, their input parameters, output values, etc. This input specifies the location where the output WSDL interface document needs to be stored.

- **Output WSDL implementation file name.** The implementation WSDL document is used to specify information about a particular Web service that provides the same interface as specified by the interface WSDL document. This input specifies the location where the output WSDL implementation document needs to be stored.

- **Target namespace for interface WSDL document.** An XML namespace is a collection of names, which are used in XML documents as element types and attribute names. The namespace is identified by a URI reference. It is then used to qualify the element types and attribute names. Since the interface WSDL document specifies the Web service operations as element types we need a namespace for it. This input specifies the target namespace for the interface WSDL document.

- **Target namespace of the implementation WSDL document.** The implementation WSDL document specifies the URL of a particular Web service. This input specifies the target namespace for the implementation WSDL document.

- **URL of interface WSDL document.** This is used to import the interface WSDL document in the implementation document.

- **Service location URL.** The actual location where the Web service would be deployed.

The inputs for this phase for the example Gator directory service are shown in Figure 4-2. These inputs are collected and passed to the Operations selection and events definition phase.

4.2 Operations Selection and Events Definition Phase

This phase makes use of the information entered in the Service definition phase to load the Java class corresponding to the server-side application. A custom class loader has been designed and implemented for this purpose. This class loader takes the physical

location of the class as input and loads the class into memory in the form of a Class object.

Once the class has been loaded into memory, Java's Reflection function is used to obtain a list of all the public methods of the class. This list is displayed to the user in the form of a table containing check boxes to allow the selection of operations and definition of events as shown in Figure 4-3. The implementation of this part of the GUI uses the Model-View-Controller design pattern. The model consists of a collection of table data objects. The table data object consists of a boolean field to indicate that a method has been chosen, the method signature and the events to be tracked for the method. Cell renderers constitute the view component. The Editors for the expose method column and the events to track column constitute the controller component. They accept the user input and instruct the model and view port to update themselves based on that input. This approach may seem unnecessarily complex, but it allows a single component to draw all the table's cells instead of requiring the table to allocate a component for each cell. Allocating a component for each cell would result in the wastage of a lot of precious memory and would also look clumsy.

Thus the user need not remember any of the method names that need to be exposed as Web service operations. He/she can simply use the GUI to select the methods and the events he wishes to track for each of them. The GUI also displays method signatures to distinguish between overloaded methods. The check boxes in the GUI make the process of selection of operations and events very easy. This approach would allow a person with little technical knowledge about Web services and ETR technology to create a full-fledged ETR-Enabled Web service given the corresponding Java class.

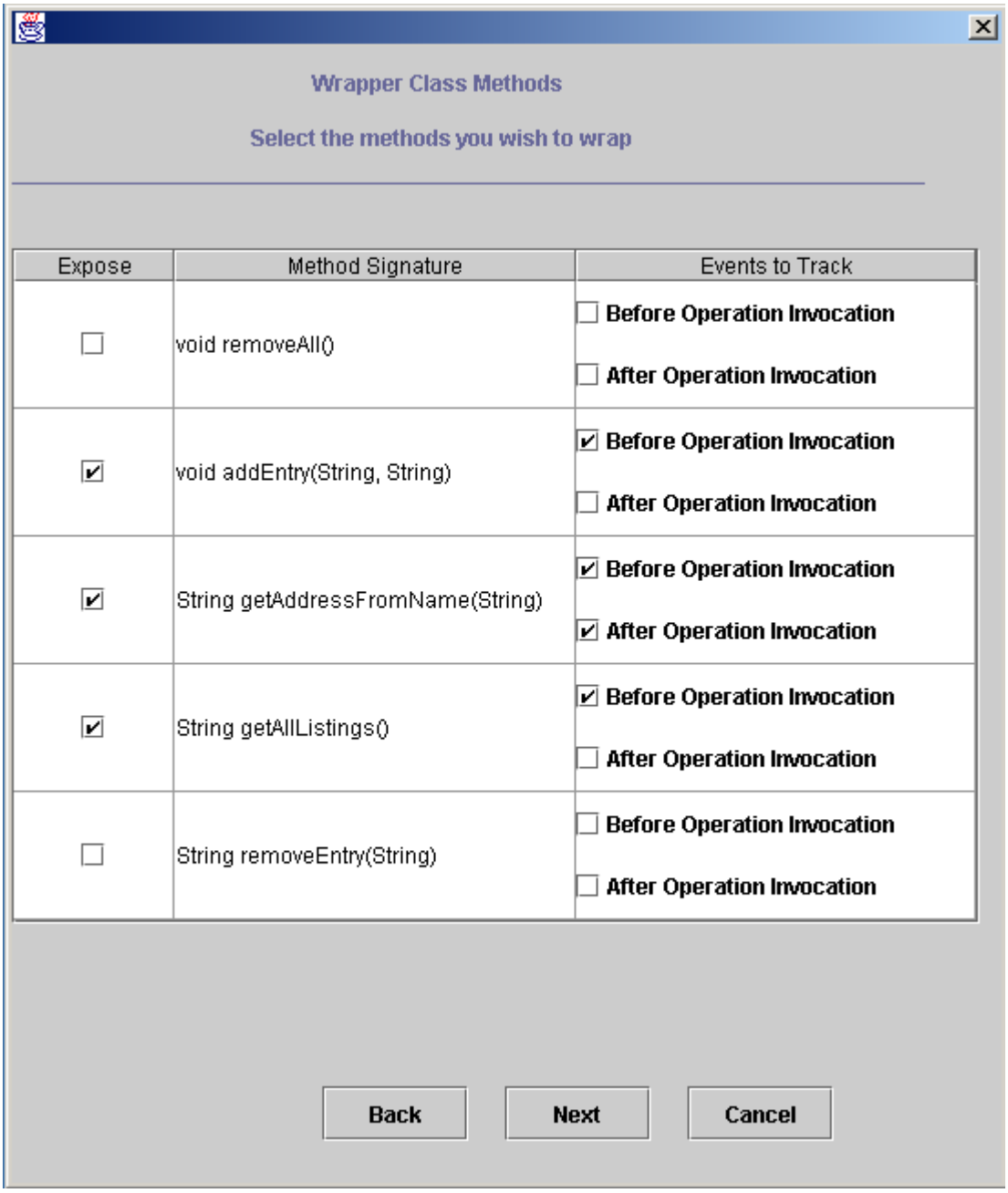


Figure 4-3 Operations selection and events definition phase

Once the user has finalized the selection of the methods and the definition of events to be tracked, a method information object is created for each selected method and added to a vector. Thus the vector then contains method information objects for all the selected methods. The structure of this method information object is shown in Figure 4-4.

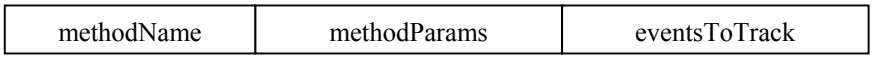


Figure 4-4 Method information object structure

The methodName member is a string that stores the name of the method. The methodParams member is another string variable that stores the formal parameters accepted by the method. The eventsToTrack is an integer variable that keeps track of the events selected by the user for the method. The values held by this variable are –

- **eventsToTrack = 0 :** Neither Before Operation, nor After Operation Invocation events have been defined.
- **eventsToTrack = 1 :** Only After Operation Invocation event has been defined.
- **eventsToTrack = 2 :** Only Before Operation Invocation event has been defined.
- **eventsToTrack = 3 :** Both Before Operation and After Operation Invocation events have been defined.

Thus at the end of this phase, the vector has all the information regarding the methods that were selected and the events chosen for each method. This information is passed on to the next phase, namely the Generation phase.

In the Gator directory service example the service provider may choose to expose only the addEntry, getAddressFromName and getAllListings methods. A screen shot for this phase is shown in Figure 4-3. As shown in the figure the service provider chooses the operations he wishes to expose and the events he wishes to track for each operation.

4.3 Generation Phase

In the Generation phase, the information entered by the user in the prior stages is displayed. The Java class chosen to act as the server-side application is displayed along with the names of the output interface and implementation WSDL documents. If the user wishes to change any of these inputs he/she can just press the “Back” button to go back to

the Service definition phase. The operations and events chosen by the user in the Operations selection and events definition phase are displayed in the form of a tree, with each operation represented as an inner node along with its associated event selections. Information about an event is displayed when the user selects an event in the tree.

In Figure 4-5 the Before Operation Invocation event of the getAllListings method is shown highlighted. The information related to this event is shown in the right hand panel. The user can also undo the selection of an event by selecting the event and hitting the Delete key. After the user has verified the information, he can choose to finish, by pressing the “Finish” button. A screen shot of this stage is shown in Figure 4-5.

Once the user clicks on the Finish button, the GUI initiates the execution of the Wrapper Generator, the WSDL Document Generator and the Events Installer components. Each of these components makes use of information passed to this phase from the Operations selection and events definition phase. A progress bar is displayed that indicates the percentage completion of these activities. The user can also stop the execution at any point of time.

Upon completion of all activities the wrapper class is compiled and placed in the appropriate directory location. The wrapper implements posting of the selected methods and also invokes the methods of the underlying Java class. The events chosen by the user are also installed on the ETR server, so that they can be posted at run-time by the wrapper. The interface and implementation WSDL documents are also generated and placed in the appropriate directory location. Thus the completion of all activities produces a full-fledged ETR-Enabled Web service that can accept Web service requests and post events to an ETR server at run-time.

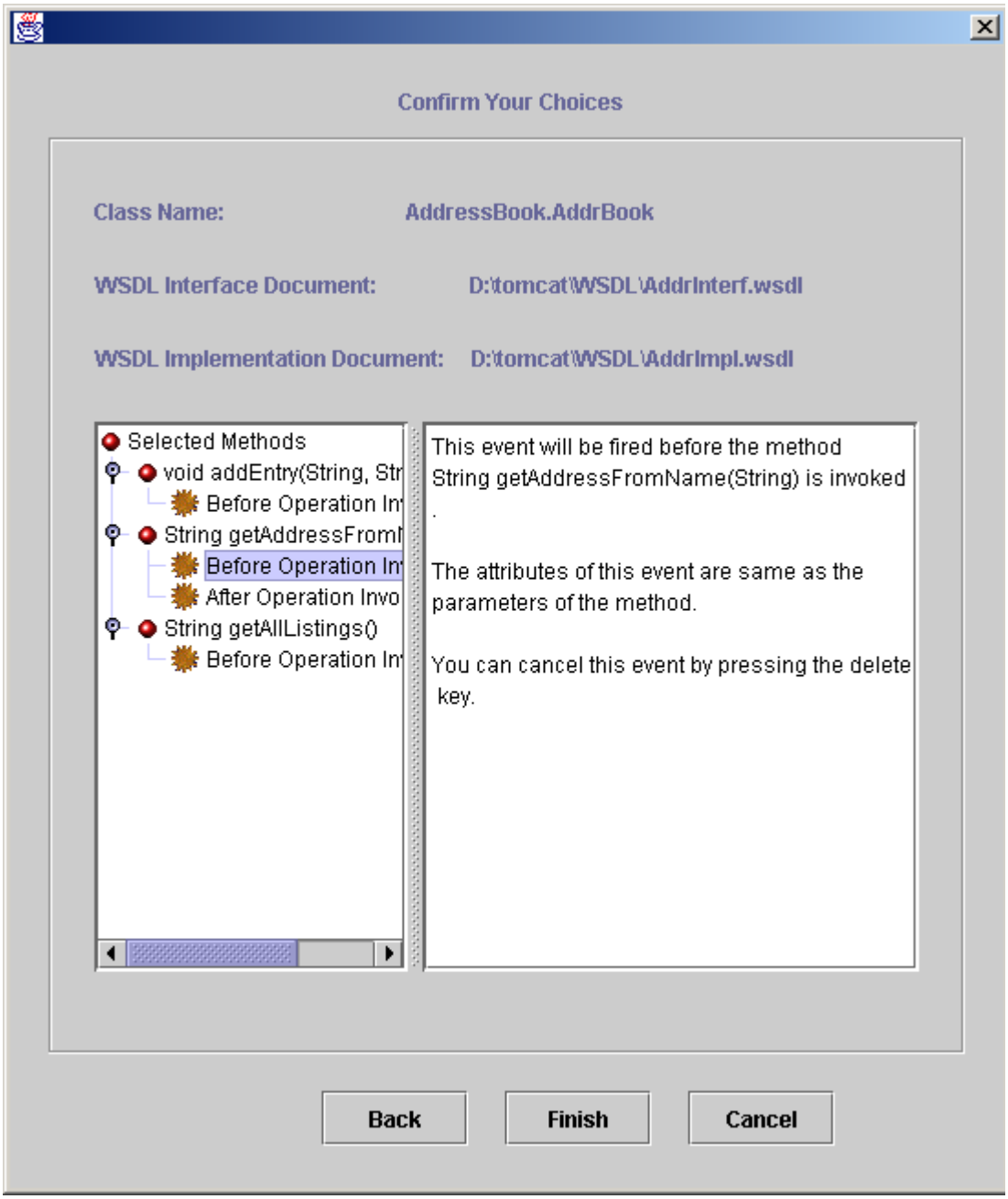


Figure 4-5 Generation phase

The next chapter describes the Wrapper Generator, WSDL Document Generator and Events Installer components used to create an ETR-Enabled Web service.

CHAPTER 5
WRAPPER GENERATOR, WSDL DOCUMENT GENERATOR AND EVENTS INSTALLER

The Wrapper Generator, WSDL Document Generator and the Events Installer are the key components involved in the creation of an ETR-Enabled Web service. As described in chapter 4, the user interacts with the GUI to provide the required inputs. The GUI passes on input information to the generator and installer components and keeps track of the overall progress made. This chapter deals with the design and implementation of these components. The organization of the remainder of this chapter is as follows. Section 5.1 describes the design and implementation of the Wrapper Generator. Section 5.2 describes the working of the WSDL Document Generator and Section 5.3 describes the design and implementation of the Events Installer. In Section 5.4 we present a brief summary of how all the components work together in synergy and describe the run-time scenario of the example Gator directory service.

5.1 Design and Implementation of the Wrapper Generator

The design of the Wrapper Generator is shown in Figure 5-1 in the form of a component diagram. The service provider, through the GUI, indicates the directory path to the application Java class to be made into a Web service and specifies the methods to be exposed along with the events to be tracked for each method. These inputs are passed to the Wrapper Generator as shown in Figure 5-1. The output of this Wrapper Generator is a wrapper class that encapsulates the application Java class. Additionally, the wrapper class contains the code required to post appropriate Before Operation and/or After

Operation events. The following sections will describe the details of the design and implementation of each component of the Wrapper Generator.

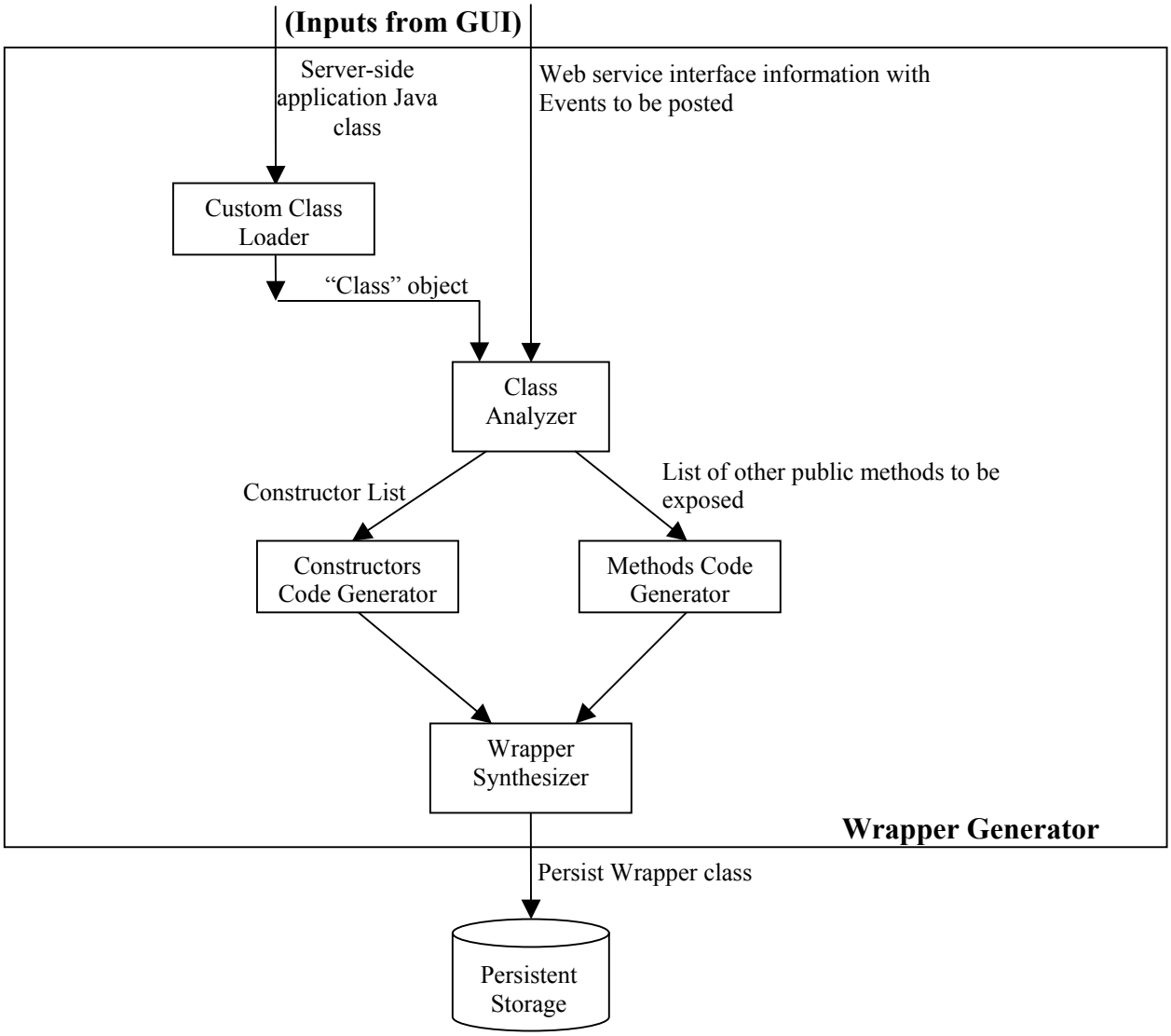


Figure 5-1 Design of the wrapper generator

5.1.1 Custom Class Loader

Java uses a system class loader for loading an application Java class and all other associated classes. The CLASSPATH environment variable is used to determine the location of these classes. However, in our case we need to be able to load the application Java class located in any directory path. Hence we have designed and implemented a

Custom Class Loader. It takes as input the directory path of the server-side application Java class. It loads this class along with all other associated classes. This class loader follows a delegation model, thus overriding only the findClass method of its parent class – Java's ClassLoader class. In the delegation model, class loaders have a hierarchical relationship, each class loader having a parent class. When a class loader is asked to load a class, it consults its parent class loader before attempting to load the class itself. The parent in turn consults its parent and so on. So it is only after all of the ancestor class loaders cannot find the item, that the current class loader gets involved. In our case, the Custom Class Loader delegates the responsibility of class loading to the system class loader, which is its parent. In case the system class loader cannot load the class, the findClass method of the Custom Class Loader is called. Thus it is only responsible for loading the classes not available to the system class loader. It loads the requested class(es) and returns a Class object for the application Java class. This Class object is passed as an input to the Class Analyzer.

5.1.2 Class Analyzer

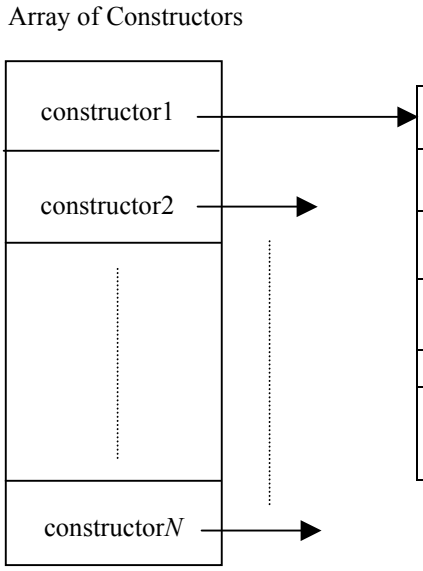
The inputs to the Class Analyzer are the Web service interface and event information (from the GUI) and the Class object (from the Custom Class Loader). The outputs of the Class Analyzer are:

- An array consisting of constructor objects to the Constructors Code Generator.
- A list of method objects to the Methods Code Generator.

The Class Analyzer uses Java's reflection capabilities to introspect the Class object supplied by the Custom Class Loader to determine its constructors and public

Array of Constructors

in Figure 5-2, the array of constructor objects has an element for each overloaded constructor of the application Java class.



Field Name	Field Type	Field Value
Declaring Class	Class	Class object representing the class that declares this constructor
Exception Types	Class[ ]	Array of Exception Types declared to be thrown
Modifiers	Int	Modifiers of this constructor such as PUBLIC, PRIVATE etc.
Name	String	Name of the Constructor
Parameter Types	Class[ ]	Array of Class objects representing formal parameter types in declaration order

Figure 5-2 Array of constructor objects

The following describes each field of the constructor object –

- **Declaring class.** This field specifies the class that has declared the constructor represented by that constructor object.
- **Exception Types.** The exception types field specifies the types of exceptions declared to be thrown by the constructor represented by this Constructor object. We can use the throw statement to declare the exception types that could be thrown.
- **Modifiers.** The modifiers field is basically an integer that specifies the modifiers of this constructor object. Examples of modifiers include PUBLIC, PRIVATE, PROTECTED etc. These are just symbols representing certain constant integer values corresponding to the public, private and protected modifiers.
- **Name.** This field stores the name of the constructor as a string.
- **Parameter Types.** The constructor may have several formal parameters declared. This field is essentially an array of class objects that represents the formal parameters in the order they are declared in the constructor.

Figure 5-3 shows an example constructor objects array that corresponds to our example Gator directory service Java class. Since the Java class corresponding to this

application has only one constructor, the array of constructors shown below has only one element in it.

Array of Constructors

constructor1 →

Field Name	Field Type	Field Value
Declaring Class	Class	Class object for the class AddressBook.AddrBook
Exception Types	Class[ ]	Null
Modifiers	Int	PUBLIC
Name	String	AddrBook
Parameter Types	Class[ ]	Null

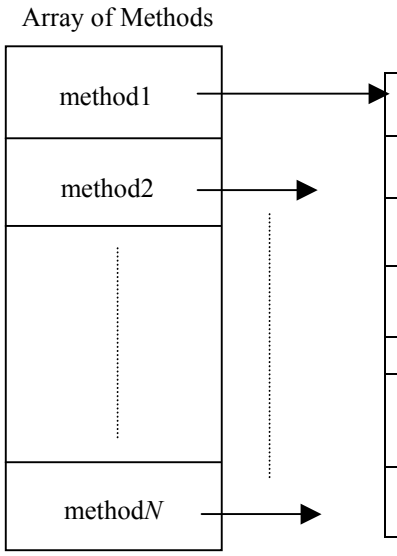
Figure 5-3 Array of constructor objects for the Gator directory service

As shown in Figure 5-3, the Java class corresponding to the Gator directory service is called AddrBook. It is inside a package called AddressBook. The constructor is not declared to be throwing any types of exceptions. This is indicated by the Null value in the Field Value column of the Exception Types field. Also the constructor of this class is declared public as indicated by the Modifiers field. Since the name of the constructor is always same as that of the class, the Name field has the value AddrBook. Finally the constructor takes no arguments, hence the Parameter Types field has a Null value in this field.

The Class Analyzer also supplies the Methods Code Generator component with an array of method objects. These method objects correspond to the public methods of the Java class chosen to be exposed as Web service operations, by the service provider. Figure 5-4 shows a schematic diagram of the method objects array.

The array of method objects is similar to the array of the constructor objects. This is because a constructor is also a specific type of a method in a class. The only difference is that the method object has an additional field for the return type of the method. This is

basically a Class object representing the return type of the method. Since a constructor cannot have a return type, we don't have this field in a constructor object.

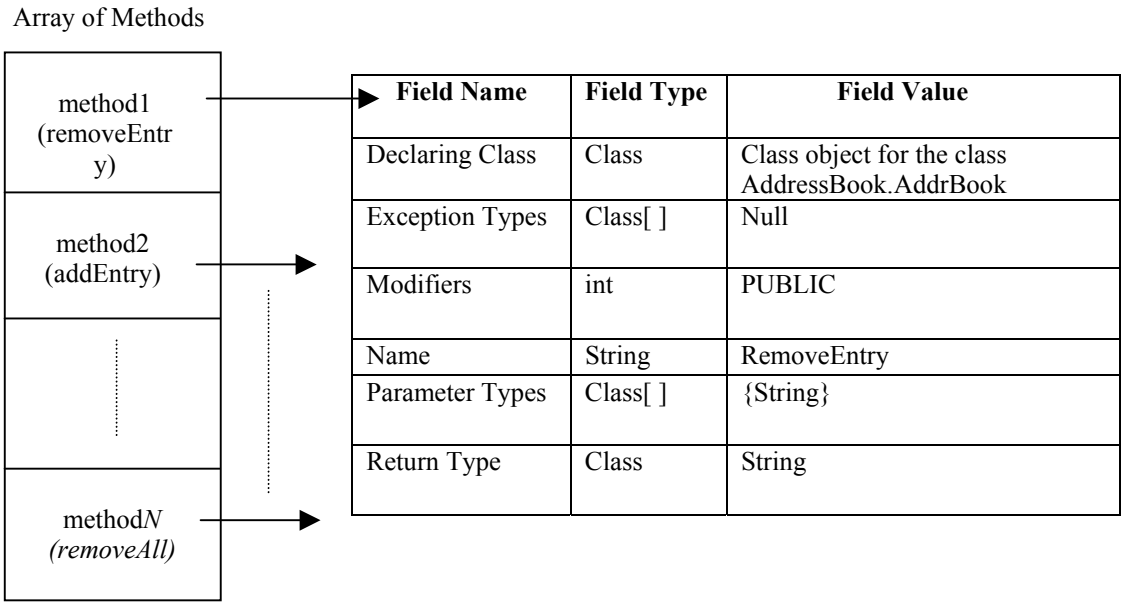


Field Name	Field Type	Field Value
Declaring Class	Class	Class object representing the class that declares this method
Exception Types	Class[ ]	Array of Exception Types declared to be thrown
Modifiers	Int	Modifiers of this method such as PUBLIC, PRIVATE etc.
Name	String	Name of the method
Parameter Types	Class[ ]	Array of Class objects representing formal parameter types in declaration order
Return Type	Class	Class object representing the return type of this method

Figure 5-4 Array of method objects

Figure 5-5 shows the array of method objects for the Gator directory service Java class. The Java class has methods named removeEntry, getAddressFromName, addEntry, getAllListings and removeAll. Figure 5-5 shows the details of the method object corresponding to the removeEntry method. The details of the other method objects are similar.

As shown in Figure 5-5, the Java class corresponding to the Gator directory service is called AddrBook. It is inside the package called AddressBook. The method is not declared to throw any types of exceptions; hence the exception types field is Null. This method is declared to be public as shown by the modifiers field. The name field indicates that the name of the method is removeEntry. Also this method takes a string input as shown by the parameter types field. Finally, this method returns a string as shown by the return type field.



Field Name	Field Type	Field Value
Declaring Class	Class	Class object for the class AddressBook.AddrBook
Exception Types	Class[ ]	Null
Modifiers	int	PUBLIC
Name	String	RemoveEntry
Parameter Types	Class[ ]	{String}
Return Type	Class	String

Figure 5-5 Example of an array of method objects

This array of method objects is used by the Methods Code Generator to generate code for the wrapper class methods.

5.1.3 Constructors Code Generator

The Constructors Code Generator component accepts the array of constructor objects and generates code for each overloaded constructor. The signatures of the generated constructors are identical to those of the corresponding constructors in the underlying application Java class. Code is generated to instantiate the objects of the underlying Java class in these constructors. Figure 5-6 describes the sequence of steps involved in the generation of each constructor for the wrapper class. This process is repeated to generate the code for each overloaded constructor.

The Constructors Code Generator generates code for every constructor object in the constructor objects array. It does this by building a string for the constructor code. As shown in Figure 5-6, the code generation starts off by appending the Web service name to

the constructor code string. This is followed by adding any formal parameters that the constructor object may have.

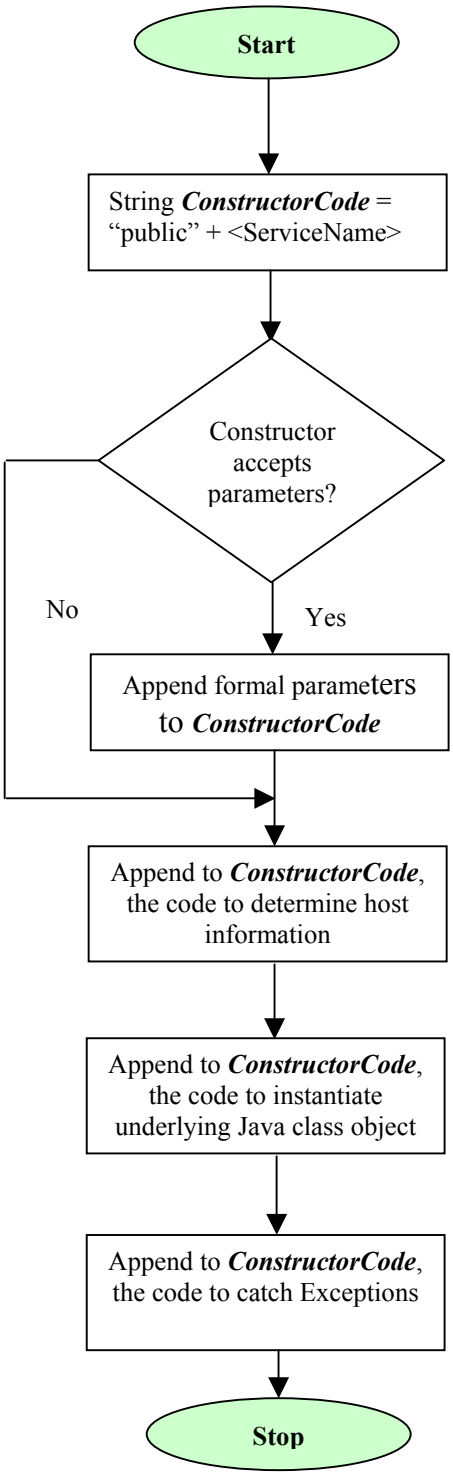


Figure 5-6 Sequence of steps involved in constructor code generation

Once the method signature has been constructed we add code to determine the host information on which the Web service runs. This information would be used at run-time by the methods of this class to post before and/or after operation events to the ETR server. Next, we generate code to instantiate an object of the underlying Java class. Finally, code is generated to catch any exceptions that may occur at run time during the execution of the constructor.

```
1 public AddressBookService()
2   {
3         try
4         {
5                 InetAddress localhost = InetAddress.getLocalHost();
6                 localhostName = localhost.getHostName();
7                 localhostIPAddr = localhost.getHostAddress();
8         //Instantiate Underlying class Object
9                 AddressBookobj = new AddressBook();
10        }
11        catch (Exception e)
12        {
13                System.out.println(" Exception in Wrapper Constructor"+e);
14        }
15 }
```

Figure 5-7 Constructor generated for the wrapper of the Gator directory service

Figure 5-7 shows the constructor generated for the Gator directory service. As shown in the lines 5 – 7, an instance of the InetAddress class is used to obtain the host information of the host running the service. The constructor creates a new instance of the AddressBook object (line 9), which is the underlying Java class that contains the actual business logic to maintain the directory service. Hence whenever an object of the wrapper class is created, an object of the underlying application Java class is created. If an exception occurs at run-time, it is printed on the screen.

5.1.4 Methods Code Generator

Once the code for the constructors has been generated, we need to generate the code for the methods that have been chosen to be exposed as Web service operations.

Figure 5-8 shows the sequence of steps involved in the generation of each method by the Methods Code Generator component.

The Methods Code Generator accepts as input an array of method objects and the events that need to be posted for each method. These are the public methods that the service provider wishes to expose as Web service operations and the events he wishes to track for each of them. As shown in Figure 5-8, method code generation starts off by appending the method signature to the MethodCode string. Next, we check to see if the method accepts any parameters. If it does, then the appropriate parameters are appended to MethodCode. We then check to see if there is a Before Operation Invocation (BOI) event defined for this method. If such an event has been defined, code is generated to post the BOI event. The parameters of the method are passed as attributes to the BOI event, which in turn pass them to the rules that are triggered by the event. These attributes can be used by the rules to make decisions and enforce business rules. This step is skipped if there is no BOI event defined. Following this, we append code to call the underlying class method that performs the actual business operation requested by the user. We then check to see if there has been an After Operation Invocation (AOI) event defined for this method. Again, if such an event is defined, code is generated to post the AOI event. The step is skipped if no AOI event is defined. The parameters of the method are also passed as attributes for the AOI event. We then append code to catch any exceptions that might occur at run time. Finally code is generated to return any values returned by the underlying class method.

Thus the methods generated by the Methods Code Generator implement posting of events to the ETR server in addition to calling the appropriate method of the underlying server-side application Java class.

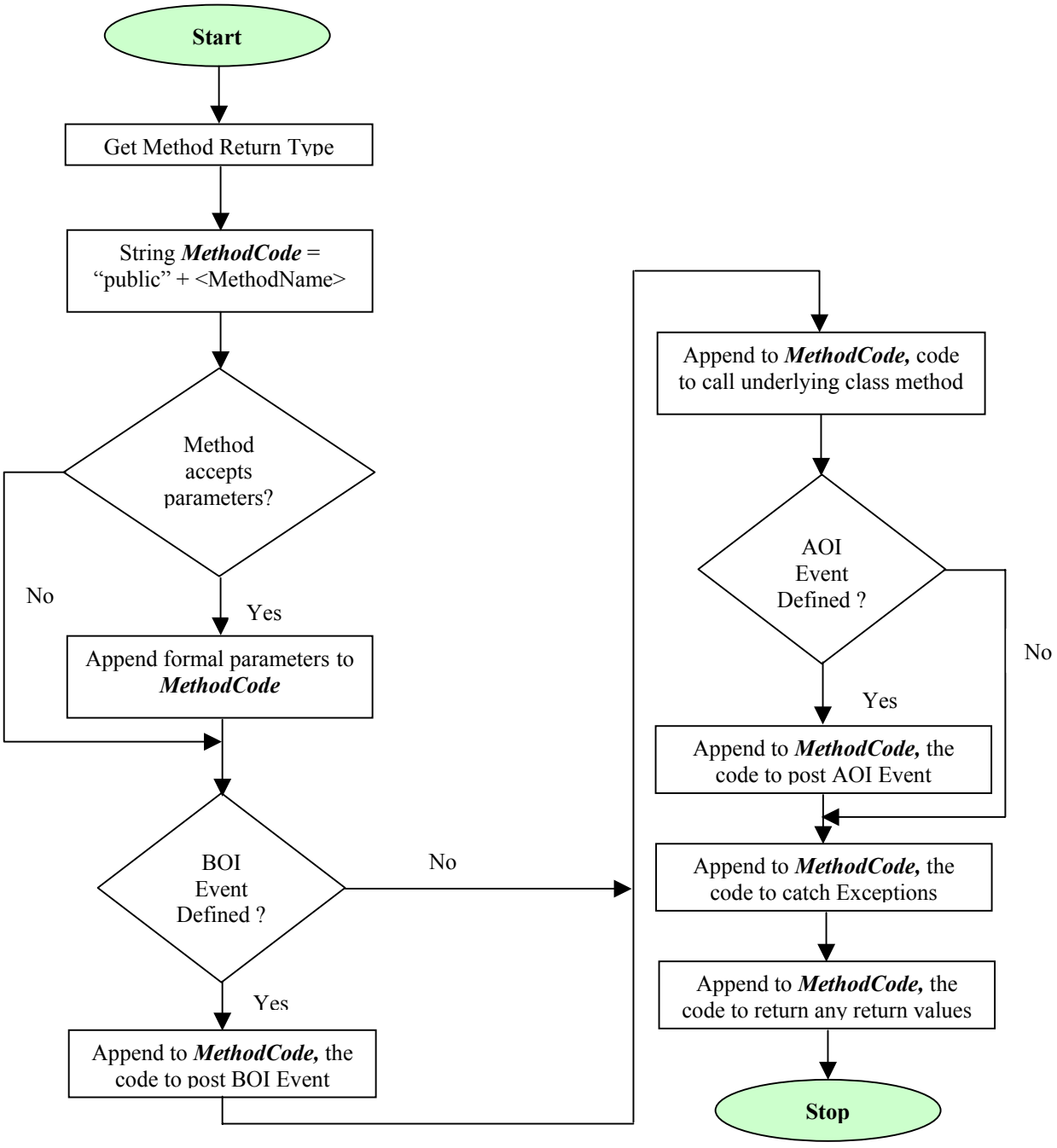


Figure 5-8 Sequence of steps involved in method code generation

Figure 5-9 shows the code generated for the removeEntry method of the Gator directory service. As shown in the figure, the service provider has chosen to track only the BOI event of this method. Hence we see code generated to post the BOI event before the actual underlying Java class method is called. The BOI event here is called AddressBook_String_removeEntry_String__BeforeOperationInvocation. Since the ETR server as such has been made into a Web service, posting of events now happens as Web service method invocations. We begin the construction of the SOAP request by setting the endpoint address of the service and the operation to be invoked (lines 12 – 17). We then specify the appropriate serializers and deserializers for the SOAP engine to be able to perform the marshalling and unmarshalling operations (line 21). The *post* method of the ETR server is used to post the event (lines 17 and 27). The Apache Axis toolkit takes care of building the SOAP message. This is done when we call the invoke method on an instance of the Call class (line 27). The post method takes as input the host name and IP address (of the node trying to post the event) along with the actual event object. This is then followed by calling the removeEntry method of the underlying Java class (line 31). The value “b13” returned by the underlying method is then returned by the wrapper class method (line 37).

Thus we see that the removeEntry method is a characteristic operation of an ETR-Enabled Web service. In addition to performing the normal task of removing an entry from an address book, it also implements posting of specific events to an ETR server. The service provider may have defined business rules associated to these events to enforce certain policies or regulations. Hence at run-time when the operation is invoked the events are posted and the appropriate business rules are invoked. The business rules

may also be changed dynamically without affecting the Web service in any way. Also since the parameters of the method are passed as attributes to the event, more informed decisions can be made in the business rules. This provides a very flexible mechanism to integrate ETR technology with Web services technology.

```
//removeEntry method
1 public java.lang.String removeEntry(java.lang.String a0)
2 {
3       java.lang.String  b13 = null;
4       try
5       {
6               QName EvQName=null;
7
8               Integer result;
9
10              //Generate code to post Before Operation Invocation Event
11
12              String endPoint=EventDistributor.DistributorConstants.DistributorAddr;
13              EvQName =new Qname ("urn:Ns119",
                "AddressBook_String_removeEntry_String__BeforeOperationInvocation");
14              Service beforeOperationService=new Service();
15              Call beforeOperationCall=(Call)beforeOperationService.createCall();
16              beforeOperationCall.setTargetEndpointAddress( new java.net.URL(endPoint) );
17              beforeOperationCall.setOperationName(new QName("", "post"));
18
19              //Specify Serializers and Deserializers
20
21              beforeOperationCall.registerTypeMapping(EventDistributor.AddressBook_String_
                removeEntry_String__BeforeOperationInvocation.class,EvQName,new
                BeanSerializerFactory(EventDistributor.AddressBook_String_removeEntry_String__Bef
                oreOperationInvocation.class,EvQName),new
                BeanDeserializerFactory(EventDistributor.AddressBook_String_removeEntry_String__B
                eforeOperationInvocation.class,EvQName));
22
23              EventDistributor.AddressBook_String_removeEntry_String__BeforeOperationInvocatio
                n  beforeOpnEvent = new
                EventDistributor.AddressBook_String_removeEntry_String__BeforeOperationInvocatio
                n(a0);
24
25              //Post the Before Operation Invocation Event
26
27              result = (Integer)beforeOperationCall.invoke(new Object[] {
                localhostName,localhostIPAddr,beforeOpnEvent});
28
29              // Call the underlying class method
30
31              b13 = AddressBookobj.removeEntry( a0);
```

Figure 5-9 Code generated for the removeEntry method of the Gator directory service

```
32          }
33          catch (Exception e)
34          {
35              e.printStackTrace();
36          }
37          return b13;
38  }//End of removeEntry method
```

Figure 5-9 Continued

5.1.5 Wrapper Synthesizer

Once the code for all the constructors and methods has been generated, the Wrapper Synthesizer is invoked. It integrates the code generated by the Constructors and Methods Code Generator components to generate the final wrapper class. The name of this class is same as that of the underlying Java class with the word "Service" appended to it. This generated Java class is persisted and then compiled. The compiled wrapper class is placed in the appropriate directory. Figure 5-10 shows a part of the code generated for the wrapper for the Gator Directory Service application.

```
//Generated Class
1 package AddressBook;
2 import org.apache.axis.client.Call;
3 import org.apache.axis.client.Service;
4 import javax.xml.namespace.QName;
5 import org.apache.axis.encoding.ser.BeanDeserializerFactory;
6 import org.apache.axis.encoding.ser.BeanSerializerFactory;
7 import java.net.InetAddress;
8 class AddressBookService
9 {
10      private String localhostName;
11      private String localhostIPAddr;
12      private AddressBook AddressBookobj;
13
14      //Constructor
15
16      public AddressBookService()
17      {
18              try
19              {
20                      InetAddress localhost = InetAddress.getLocalHost();
21                      localhostName = localhost.getHostName();
```

Figure 5-10 Wrapper class for the Gator directory service

```
22                      localhostIPAddr = localhost.getHostAddress();
23
24                      //Instantiate Underlying class Object
25
26                      AddressBookobj = new AddressBook();
27              }
28              catch (Exception e)
29              {
30                      System.out.println(" Exception in Wrapper Constructor"+e);
31              }
32      }
33
34
35
36      //removeEntry method
37
38      public java.lang.String removeEntry(java.lang.String a0)
39      {
40              java.lang.String  b13 = null;
41              try
42              {
43                      QName EvQName=null;
44                      Integer result;
45
46                      //Generate code to post Before Operation Invocation Event
47
48                      String endPoint=EventDistributor.DistributorConstants.DistributorAddr;
49                      EvQName =new Qname
                        ("urn:Ns119","AddressBook_String_removeEntry_String__BeforeOpera
                        tionInvocation");
50                      Service beforeOperationService=new Service();
51                      Call beforeOperationCall=(Call)beforeOperationService.createCall();
52                      beforeOperationCall.setTargetEndpointAddress( new java.net.URL(endPoint) );
53                      beforeOperationCall.setOperationName(new QName("", "post"));
54
55                      //Specify Serializers and Deserializers
56
                        beforeOperationCall.registerTypeMapping(
                        EventDistributor.AddressBook_String_removeEntry_ String__
                        BeforeOperationInvocation.class,EvQName,new
                        BeanSerializerFactory(EventDistributor.AddressBook_String_removeEntry_Stri
                        ng__BeforeOperationInvocation.class,EvQName),new
                        BeanDeserializerFactory(EventDistributor.AddressBook_String_removeEntry_
                        String__BeforeOperationInvocation.class,EvQName));
57                      EventDistributor.AddressBook_String_removeEntry_String__
                        BeforeOperationInvocation beforeOpnEvent = new
                        EventDistributor.AddressBook_String_removeEntry_String__BeforeOperationI
                        nvocation( a0);
58
59                      //Post the Before Operation Invocation Event
60
61                      result = (Integer)beforeOperationCall.invoke(new Object[] {
                        localhostName,localhostIPAddr,beforeOpnEvent});
```

Figure 5-10 Continued

```
62
63                      // Call the underlying class method
64
65                      b13 = AddressBookobj.removeEntry( a0);
66
67              }
68              catch (Exception e)
69              {
70                      e.printStackTrace();
71              }
72              return b13;
73  }//End of removeEntry method
74
75 } //End of wrapper
```

Figure 5-10 Continued

As shown in Figure 5-10, the name of the generated wrapper class is called AddressBookService. It is placed in the AddressBook package (line 1), which is same as that of the underlying Java class. The wrapper includes an object of the underlying Java class as a member variable. This role is played by the AddressBookobj variable (line 12). The packages imported by the wrapper in lines 2 – 6 are those needed to make Web service operation invocations of the post method of the ETR server. The rest of the wrapper code is the concatenation of the previously generated constructors and methods code.

5.2 WSDL Document Generator

The WSDL Document Generator component takes as input the Web service interface information along with information about the WSDL documents to be generated as described below:

- Name and directory path of the WSDL interface document to be generated.
- Name and directory path of the WSDL implementation document to be generated.
- Target namespace of the interface WSDL document.
- Target namespace of the implementation WSDL document.
- URL of the interface WSDL document.
- URL of the target Web service.

The generated WSDL documents are placed in the specified directory path. Since the WSDL documents describe types and operation names, we use a namespace to restrict their scope. Using namespaces allows us to use the same names in different contexts, by qualifying the names with their respective namespace.

The interface WSDL document describes the complete interface of the Web service, such as the operation names, their parameters and return values etc. The implementation WSDL document imports the interface WSDL document using its URL. The implementation WSDL document also specifies the URL at which the actual Web service is located. Thus we can have several service providers providing a single type of a Web service share a single interface document but having separate implementation WSDL documents.

We make use of the Java2WSDL tool provided by the Apache Axis toolkit in the WSDL Document Generator. This tool creates the interface and implementation WSDL documents for an existing Java class to be deployed as a Web service. This command line tool creates WSDL documents that will contain appropriate WSDL types, messages, port type information, binding information and service descriptions. The syntax for the usage of this tool is as follows:

java org.apache.axis.wsdl.Java2WSDL [options] service-class-name

Some of the options used by this tool include

- -o Output interface WSDL file name.
- -O Output implementation WSDL file name.
- -m List of methods to be exposed.
- -l Service location URL.
- -n Target namespace of the interface WSDL document.
- -L URL of interface WSDL document.

Thus, this tool creates the interface and implementation WSDL documents and stores them persistently. Figure 5-11 shows the Java2WSDL command used to generate the WSDL documents for the Gator Directory Service. As shown in the Figure, the WSDL interface and implementation WSDL documents here are AddrInterf.wsdl and AddrImpl.wsdl respectively.

```
java org.apache.axis.wsdl.Java2WSDL -N "http://localhost:8080/services/AddrBookImpl" -n
"http://localhost:8080/services/AddrBookInterf" -o J:\jakarta-tomcat-4.0.1\WSDL\AddrInterf.wsdl -O
J:\jakarta-tomcat-4.0.1\WSDL\AddrImpl.wsdl   -L "http://localhost:8080/AddrInterf.wsdl" -m
"getAllListings removeEntry " -l "http://localhost:8080/services/AddressBook"
AddressBook.AddrBookService
```

Figure 5-11 Java2WSDL command for the Gator directory service

The target namespace of the interface WSDL document is specified to be http://localhost:8080/services/AddrBookInterf and that of the implementation WSDL document is http://localhost:8080/services/AddrBookImpl. The URL of the interface WSDL document is http://localhost:8080/AddrInterf.wsdl and the URL of the Web service is http://localhost:8080/services/AddressBook.

5.3 Design and Implementation of the Events Installer

The Events Installer accepts as input a vector consisting of the events that need to be installed on the ETR server. Figure 5-12 describes the Events Installation process. Based on the information in the input vector, the Events Installer builds the event name for each event to be installed. The event name is built by concatenating with the class name of the server-side application Java class, the method signature of the appropriate method followed by “BeforeOperationInvocation” or “AfterOperationInvocation” depending on whether it’s a Before Operation Invocation or an After Operation

Invocation event. Such a scheme for building the event name assures unique identification of an event given its name.

The Events Installer then creates an Event Definition object for an event using the information about the event, such as its name, its attributes etc. The attributes of an event are same as the parameters of the method it is associated to. For example the event associated with the method with signature addEntry (String name, String address) would also have two string attributes. This allows the triggered rules to make more informed decisions based on the values of the event's attributes at run time. Upon creation of the Event Definition object, the Events Installer invokes the insertEvent and writeEventClassFile methods of the Events Installation Service of the ETR Server. The Events Installer also returns status information regarding the number of events installed to the GUI for Web service creation.

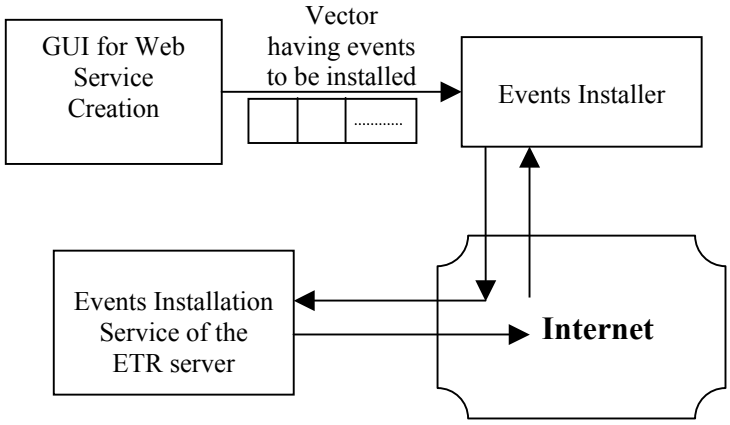


Figure 5-12 Events installation process

The Events Installation Service (EIS) of the ETR server is a full-fledged Web service that interacts with the ETR server to install events on it. It has three operations namely,

- **makeConnection.** The makeConnection method is invoked when the EIS is first deployed and is up and running. This method authenticates the EIS with the ETR server.

- **InsertEvent.** The insertEvent method adds the new event definition in the ETR server. The event definition object encapsulates event details like its name, its attributes, its owner etc.

- **writeEventClassFile.** The writeEventClassFile method generates a Java class for the event. This Java class is then persisted.

Thus the insertEvent and writeEventClassFile methods are responsible for creating a persistent copy of the event in the ETR server. Figures 5.13 and 5.14 show the SOAP messages exchanged by the Events Installer and the EIS during the insertEvent method invocation. Figure 5-13 shows the SOAP message request sent by the Events Installer component to the EIS, requesting it to install the Before Operation Invocation event for the removeEntry method. Figure 5-14 shows the SOAP message response from the EIS.

```
POST /axis/services/EventCreationService HTTP/1.0
Host: localhost

Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-Length: 1713

<?xml version="1.0" encoding="UTF-8"?>
    <eventName xsi:type="xsd:string">AddrBook_String_removeEntry_String__BOI</eventName>
   <triggerClass xsi:type="xsd:string"></triggerClass>
   <triggerMethod xsi:type="xsd:string"></triggerMethod>
   <codeGenState xsi:type="xsd:boolean">false</codeGenState>
  </multiRef>
  <multiRef id="id1" SOAP-ENC:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns2:Vector"
xmlns:ns2="http://xml.apache.org/xml-soap">
   <item href="#id2"/>
  </multiRef>
  <multiRef id="id2" SOAP-ENC:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns3:ParameterPair"
xmlns:ns3="urn:PpTypes">
   <parameterName xsi:type="xsd:string">a0</parameterName>
   <parameterType xsi:type="xsd:string">java.lang.String</parameterType>
   <value xsi:type="xsd:string"></value>
  </multiRef>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 5-13 SOAP request sent to EIS by the events installer

```
HTTP/1.1 200 OK

Content-Type: text/xml; charset=utf-8

Date: Sun, 27 Oct 2002 00:02:28 GMT

Server: Apache Tomcat/4.0.1 (HTTP/1.1 Connector)

Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
 <soapenv:Body>
  <insertEventResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <insertEventReturn href="#id0"/>
  </insertEventResponse>
  <multiRef id="id0" soapenc:root="0"
soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xsi:type="ns1:Vector"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="http://xml.apache.org/xml-
soap">
   <item xsi:type="xsd:string">SUCCESS_I</item>
  </multiRef>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 5-14 SOAP response sent by the EIS to the events installer

5.4 Tying It All Together

Figure 5-15 shows the run-time scenario of a service requestor invoking the removeEntry operation of the Gator directory Web service. Note that the server-side application includes the original Java implementation of the removeEntry method. The Gator directory Web service wrapper is the code generated by the Wrapper Generator component described in Section 5.1. The wrapper code is deployed as a Web service. The interface and implementation WSDL documents shown in Figure 5-15 (generated by the WSDL Document Generator described in Section 5.2), describe the removeEntry operation. Finally, the Before Invocation event for the removeEntry operation has been installed in the ETR server by the Events Installer component described in Section 5.3.

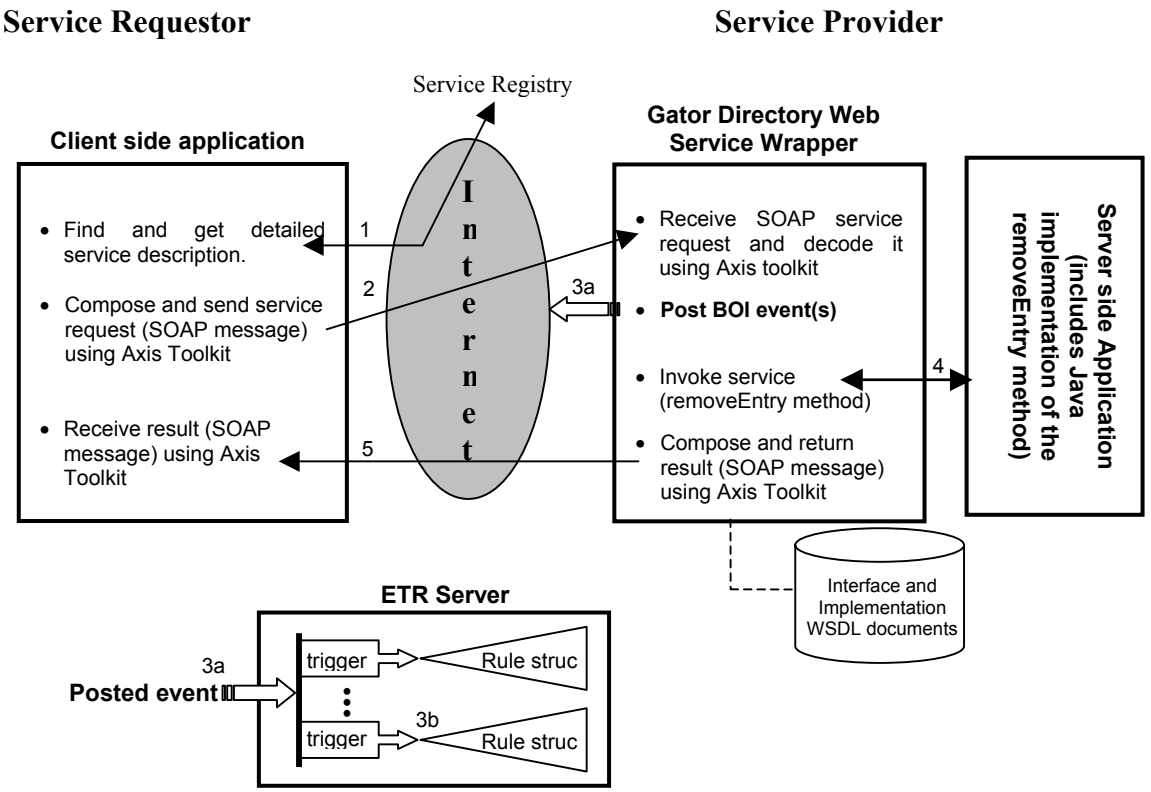


Figure 5-15 Summary of the runtime scenario of the Gator directory service

At run-time, as shown in Figure 5-15, the Service Requestor contacts the Service Registry (broker) to find an appropriate service provider (step 1). Once it has found a service provider, it uses its WSDL documents to obtain information (such as the service URL) about the service to compose and send a SOAP message to invoke the removeEntry method (step 2).

On the Service Provider side the Web service wrapper receives and decodes the SOAP message using the Apache Axis toolkit. It then posts a BOI event for the removeEntry operation (step 3a). The event triggers the rules that are associated to that event to enforce any business and security rules (step 3b). The wrapper then invokes the removeEntry method of the underlying server-side application Java class (step 4). After

execution of the removeEntry method, the wrapper composes and sends the result SOAP message back to the client again using the Apache Axis toolkit (step 5).

CHAPTER 6
SUMMARY AND CONCLUSION

In this thesis, we have described the techniques and tools that were developed to integrate business events and rules management with the Web services model. Based on a code-generation approach, we have designed and implemented a build-time tool to generate Web service wrappers and other objects required to integrate ETR technology with Web services technology on the service provider side. The build-time tool used in the system consists of four main components: GUI for Web Service Creation, Wrapper Generator, WSDL Document Generator and the Events Installer.

The GUI for Web Service Creation is a wizard-like interface that allows a service provider to specify information regarding the server-side application, such as its Java class file name, names of the output WSDL documents, service location URL etc. Based on this information, a list of public methods available in the server-side application is displayed. The user can select the methods to be exposed as Web service operations and define events (before and/or after operation invocation) he wishes to track for each operation. The information provided by the user in the GUI is then given to the other components.

The Wrapper Generator follows a code generation approach to generate a Web service wrapper for the server-side application. It takes as input the Web service interface information (i.e. the methods to be exposed as Web service operations) and the events that need to be tracked for each of these operations. It analyzes the server-side application Java class using Java's reflection capabilities and generates code for the Web service

wrapper. McCluskey (1998) illustrates the use of Java's reflection capabilities. This wrapper is then compiled, placed in the appropriate directory and deployed as the actual Web service.

The WSDL Document Generator makes use of the Java2WSDL tool provided by the Axis toolkit to generate WSDL documents. The tool introspects the service Java class to generate interface and implementation WSDL documents. These documents are then stored persistently to be used by a service requestor.

The Events Installer component contacts the Events Installation Service of the ETR Server to install the chosen events on the ETR server.

At run-time when a service requestor invokes an operation, the Web service wrapper not only invokes the appropriate underlying operation, but also posts the appropriate events to the ETR server. These events may trigger the rules associated with them, to enforce any business and security rules. The posting of events and execution of rules is transparent to the service requestor who has made the Web service request.

Thus the Web services model has been enhanced to support business events and rules-management technology at the service provider side. By specifying appropriate rules, service providers could use this model to enforce security and integrity constraints, policies and regulations, or carry out business strategies and tactics. The ETR technology provides a flexible mechanism to specify and enforce these rules. The control and logic specified by rules are not hard-coded in application programs. Rules can be more easily understood and changed. The ETR technology also provides a dynamic mechanism to specify and enforce these rules. A rule can be added, deleted, or modified at run-time without any effect on the Web service interface or the Web service implementation.

Further enhancements to the system include providing a help facility assisting the user at each phase of the GUI for Web service creation. More importantly, we can continue to integrate the event and rule management technology to the rest of the Web services model. For example, at the service requestor side, a client side Web service proxy could be generated based on a service request specification, which includes information such as the desired service and transmission primitive, the events to be posted on the requestor side, etc. This generated code for the service proxy could then post events to trigger business rules in addition to sending messages to find, bind and invoke a service.

LIST OF REFERENCES

Axis Architecture Guide. Available from: URL: http://docs.pushtotest.com/axisdocs/architecture-guide.html. Accessed: 07/2002.

Axis User Guide. Available from: URL: http://docs.pushtotest.com/axisdocs/user-guide.html. Accessed: 07/2002.

Ballinger K, Brittenham P, Malhotra A, Nagy W, Pharies S. Web Services Inspection Language (WS-Inspection) 1.0. November 2001. Available from: URL: http://www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html. Accessed: 07/2002.

Bellwood T, Clement L, Ehnebuske D, Hatley A, Hondo M, Husband Y, Januszewski K, Lee S, McKee B, Munter J, Riegen C. UDDI Version 3.0. July 2002. Available from: URL: http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm. Accessed: 09/2002.

Box D, Ehnebuske D, Kakivaya G, Layman A, Mendelsohn N, Nielsen H, Thatte S, Winer D. Simple Object Access Protocol (SOAP) 1.1. May 2000. Available from: URL: http://www.w3.org/TR/SOAP/. Accessed: 02/2002.

Christensen E, Curbera F, Meredith G, Weerawarana S. Web Services Description Language (WSDL) 1.1. March 2001. Available from: URL: http://www.w3.org/TR/wsdl. Accessed: 11/2001.

Gottschalk K. Web Services Architecture Overview – The next stage of evolution for e-business. September 2000. Available from: URL: http://www-106.ibm.com/developerworks/web/library/w-ovr. Accessed: 08/2001.

Lam H, Su SYW. Component Interoperability in a Virtual Enterprise Using Events/Triggers/Rules. Proceedings of OOPSLA 1998 Workshop on Objects, Components, and Virtual Enterprise, Vancouver, BC, Canada, Oct. 18-22, 1998, pp. 47-53.

McCluskey G. Using Java Reflection. January 1998. Available from: URL: http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/. Accessed: 04/2002.

Nagy W, Ballinger K. The WS-Inspection and UDDI Relationship. November 2001. Available from: URL: http://www-106.ibm.com/developerworks/webservices/library/ws-wsiluddi.html. Accessed 07/2002.

Snell J. Web Services Interoperability. January 2002. Available from: URL: http://www.xml.com/pub/a/2002/01/30/soap.html. Accessed 04/2002.

Su SYW, Lam H, Lee M, Bai S, Shen Zuo-Jun. An Information Infrastructure and E-services for Supporting Internet-based Scalable E-business Enterprises. Proceedings of the 5th International Enterprise Distributed Object Conference (EDOC 2001), Seattle, WA, Sept. 4-7, 2001, pp. 2-13.

BIOGRAPHICAL SKETCH

Karthik Nagarajan is a native of Odaiyalur, a small village in Tamil Nadu, India. He earned his high school diploma at Vidya Mandir Senior Secondary School located in Chennai, India. He completed his bachelor's degree in computer science and engineering from the University of Madras in May 2000 under the support of a merit scholarship awarded by the government of India. He then came to the University of Florida in fall 2000 to pursue a master's degree in computer engineering. Karthik is an avid player of tennis and has been playing the game for the past 8 years. He also plays soccer and cricket.