



**Openwings
Policy Services Specification
Ver 1.0 Final**

PREFACE

About the Open Specification Process

A brief overview of the community process for developing Openwings Specifications is discussed below. This process was developed by General Dynamics Decision Systems and is similar to the community process for developing Java specifications.

A Process Management Organization (PMO) oversees development and maintenance of all Openwings Specifications. Any company, organization or individual can join the Specification development team; team members are called Participants. The latest Openwings Specifications will be maintained on a Public Web Site.

Participants can submit requests to the PMO to modify existing or develop new Specifications. If the PMO accepts the request, the PMO will then begin the process of forming an Expert Team and will appoint a Specification Lead.

The Specification Lead will direct the Expert Team in developing the new Specification. Once the team agrees on a draft of the Specification, it will be posted on the Public Web Site for public review. A member of the Expert Team will also begin development of a Reference Implementation and Compatibility Test Suite for the new Specification.

Based on public comments, a final release of the Specification will be produced by the Expert Team. Concurrently, the Reference Implementation and Compatibility Test Suite will be completed. Once the Final Public Release of the Specification is completed and posted on the Public Web Site, the Expert Team will disband.

This process is intended to provide rich consensus based Specifications.

Contributors

We would like to recognize and thank the following people for their contributions to this document.

Author(s):

Michael Smith, Jeff Carpenter

Expert Team Members:

Stuart Lewin, BAE Systems North America

Ramesh Nagappan, Sun Microsystems

Ayal Spitz, Mitre Corp

Dr. Dave Usechak, Ocean Systems Engineering Corporation

1.	INTRODUCTION	1
1.1	PURPOSE	1
1.2	SCOPE	1
1.3	DEFINITIONS	2
1.4	OVERVIEW	2
1.4.1	<i>Openwings Overview</i>	2
1.4.2	<i>Document Overview</i>	3
2.	GOALS / REQUIREMENTS.....	4
3.	USE CASES.....	5
3.1	DEFINE POLICY	5
3.2	CREATE POLICY	5
3.3	GENERATE POLICY.....	5
3.4	DISCOVER POLICY.....	6
3.5	CONFIGURE POLICY	6
3.6	DEPLOY POLICY	6
4.	ARCHITECTURE.....	7
4.1	OVERVIEW	7
4.2	POLICY ARCHITECTURE	8
4.3	POLICY NAMING	8
4.4	POLICY ATTRIBUTES	9
4.5	POLICY LOADING	9
4.6	POLICY SERVICES AND CONTEXT SERVICES.....	10
5.	INTERFACES.....	11
5.1	NET.OPENWINGS.POLICY.POLICY	12
5.1.1	<i>loadPolicy</i>	13
5.1.2	<i>loadPolicy</i>	13
5.1.3	<i>savePolicy</i>	13
5.1.4	<i>savePolicy</i>	13
5.1.5	<i>savePolicy</i>	14
5.1.6	<i>setName</i>	14
5.1.7	<i>getName</i>	14
5.1.8	<i>setDescription</i>	14
5.1.9	<i>getDescription</i>	14
5.1.10	<i>setPolicyUpdated</i>	15
5.1.11	<i>isPolicyUpdated</i>	15
5.1.12	<i>getPolicyAttributes</i>	15
5.1.13	<i>getPolicyElements</i>	15
5.1.14	<i>getLastLoadURL</i>	15
5.2	NET.OPENWINGS.POLICY.POLICYEXCEPTION.....	16
5.2.1	<i>PolicyException</i>	16
5.2.2	<i>PolicyException</i>	16
5.2.3	<i>PolicyException</i>	17
5.2.4	<i>getEmbeddedException</i>	17
5.2.5	<i>getMessage</i>	17
5.2.6	<i>printStackTrace</i>	17
5.2.7	<i>printStackTrace</i>	17
5.2.8	<i>printStackTrace</i>	18
5.3	NET.OPENWINGS.POLICY.POLICYLOADER	18

5.3.1	<i>addPackageMapping</i>	19
5.3.2	<i>packageMapping</i>	19
5.3.3	<i>getPolicy</i>	20
5.3.4	<i>getPolicy</i>	20
5.3.5	<i>getPolicy</i>	21
5.3.6	<i>getPolicy</i>	21
5.3.7	<i>createNewPolicy</i>	22
5.4	NET.OPENWINGS.POLICY.POLICYLOADERFACTORY.....	22
5.4.1	<i>getPolicyLoader</i>	23
5.5	NET.OPENWINGS.POLICY.POLICYGENERATOR.....	23
5.5.1	<i>generatePolicy</i>	23
6.	EXAMPLES	24
6.1	SAMPLE POLICY INTERFACE.....	24
6.2	OPENWINGS POLICY INSTANCES.....	24
6.3	THE OPENWINGS TUTORIAL.....	25
7.	TOOLS	26
7.1	POLICY GENERATOR.....	26
7.2	POLICY EDITORS.....	26
8.	COMPLIANCE	27
9.	FUTURES	28
9.1	ATTRIBUTE CHANGE NOTIFICATION.....	28
9.2	POLICY MANAGEMENT.....	28
9.3	DYNAMIC POLICIES.....	28
9.4	POLICY RESOLUTION AND NEGOTIATION.....	28
10.	REFERENCES AND FURTHER READING	30

FIGURE 1:	DEFINITIONS.....	2
FIGURE 2:	GOALS / REQUIREMENTS TABLE.....	4
FIGURE 3:	POLICY USE CASES.....	5
FIGURE 4:	ELEMENTS OF A POLICY	8
FIGURE 5:	POLICY LOADER ARCHITECTURE.....	9
FIGURE 6:	POLICY INTERFACE ROLES.	11
FIGURE 7:	POLICY INTERFACES.....	11
FIGURE 8:	A SAMPLE POLICY INTERFACE.	24
FIGURE 9:	POLICY GENERATOR REFERENCE IMPLEMENTATION.....	26
FIGURE 10:	COMPLIANCE TABLE.	27

1. Introduction

A policy is defined in the dictionary as:

“A general principle or plan that guides the actions taken by a person or group.”

From a software perspective, policies are a combination of configuration data and configuration logic. This differs from the traditional concept of configuration files, where only state data is read in and used. In addition to providing rules through methods, policies can be dynamically resolved and negotiated at run time.

Within Openwings, policies provide a bridge through which components can interact with their run-time environment. This effectively decouples a component from its environment, enabling the component to be deployed in different environments without modification of its core logic. In short, policies simplify the reuse of components in different environments and enable the system to adapt to changes in the environment.

1.1 Purpose

This specification defines the concept of policies with respect to the Openwings architecture.

1.2 Scope

The potential scope of policies is quite large. Areas of interest include policy representation and the translation from representation to something that the system understands; policy negotiation and agreement, whereby conflicts between multiple policies are resolved; policy deployment and validation; policy discovery and policy management. This specification describes Openwings policy representation and translation and specifies interfaces for these interactions. Policy deployment is discussed while policy validation is deferred to a future revision of the specification. A naming convention is specified to support policy discovery. More advanced discovery can be implemented with the Openwings Component Services. There is some discussion of how policies should be managed, but the tools with which to manage them are not specified. Policy management is a subset of the Openwings management scheme, which is covered in a separate specification. The topics of policy resolution, policy negotiation and dynamic policies are discussed, but interfaces for these interactions are left for a future version of this specification.

This specification considers policies from the following perspectives:

1. Representation of policy data
2. Translation of policy data into Java objects
3. Negotiation
4. Deployment
5. Discovery and resolution
6. Management

1.3 Definitions

Term	Definition
Composite Policy	A single policy formed by the aggregation of multiple policies.
Configuration Data	A set of structured data that represents the attributes of the policy.
Configuration Logic	Methods of a policy that perform actions related to configuration. These can provide complex behaviors in the system.
Dynamic Policy	A policy whose attribute and behavior definitions can be modified at run time. Dynamic policy attribute values and class members can change at run time.
Fixed Policy	A policy whose attribute and behavior definitions are fixed at compile time. Fixed policy attribute values may be set at run time, but class members cannot change.
Managed Policy	A policy that supports the modification of attribute and operation data through an Openwings compliant management interface.
Policy Deployment	Configuring and packaging a policy for a run-time environment.
Policy Editor	Enables an entity (e.g., person, application) to define and update policies.
Policy Negotiation	The formation of a consistent policy from multiple, possibly conflicting, policies. Given a set of policies and a set of negotiation rules, policy negotiation returns a single policy created from the set of policies, using the negotiation rules.
Policy Negotiation Rules	A set of rules used in policy negotiation. For example, a policy rule might indicate that one policy has precedence over another.
Policy Repository	Persistent storage and retrieval of policies. A repository simply stores policy data, it does not in general process or act upon it.
Policy Resolution	The determination of which policies apply to a given component.
Policy Validation	Ensuring that policies are being properly applied. This may be done through system pre- and post-condition verification, for example.
Context	A context is a grouping of components. Contexts may be used to group services based on network topology or security, for example. Contexts are fully defined in the self-forming systems specification.
Unmanaged Policy	A policy that does not support the modification of attribute and operation data through an Openwings compliant management interface.

Figure 1: Definitions

1.4 Overview

1.4.1 Openwings Overview

Openwings™ is an open community, non-proprietary effort to define specifications for self-forming, self-healing systems. Motorola (now General Dynamics Decision Systems) and Sun Microsystems established the Openwings™ consortium in June of 1999. Since then, over 100 companies have registered to help mature its development, and more companies are registering daily.

The core Openwings™ framework is designed to incorporate existing commercial standards and is intended for use in both commercial and military environments. Openwings™ is being developed using a community development process modeled after the very successful Java Community Process. Anyone may join the Openwings™ community, participate on expert teams, or use the resulting specifications free of charge by merely signing up at the community web site (<http://www.openwings.org>).

The Openwings™ Architecture provides a framework for building plug-and-play, service-oriented, network-centric, self-forming, self-healing systems that are independent of middleware, databases, platforms, and deployment contexts. Openwings™ has a special focus on issues of availability, security, and interoperability. Openwings™ is the embodiment of a new movement in the software engineering community towards a paradigm known as Service-Oriented Programming (SOP). For an introduction to SOP, refer to <http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf>.

1.4.2 Document Overview

Policies represent a key step in achieving automated system management and configuration. Openwings policies allow configuration behavior to be specified and negotiated dynamically. Using policies, rules about how a component fits into its environment can be customized outside of the normal component business logic. This flexibility simplifies the reuse of components in different contexts and enables the system to adapt to changes in the environment. Policies may be defined for anything requiring configuration. Examples within the Openwings core architecture include policies for Component Services, Container Services, Management Services and others.

This specification follows the basic format of all of the Openwings specifications:

- Goals and Requirements – Describes requirements of this specification.
- Use Cases – Describes UML use cases for this specification.
- Architecture – Describes the various architectural aspects of this specification.
- Interfaces – Describes interfaces required for this specification.
- Tools – Describes various tools and utilities provided with the specification.
- Examples – Provides representative real world use cases for this specification.
- Compliance – This section describes what is required to comply with this specification.

2. Goals / Requirements

Here are the goals / requirement for this specification. Each requirement references one or more of the top-level requirements found in the Openwings Overview Specification.

#	Goals / Requirements	Ref
1.	Policies shall be used to isolate configuration data and configuration logic from normal business logic.	14
2.	Policies shall support external management at deployment or runtime. Through external management, system deployers and maintainers shall be able to configure policies.	7
3.	Policies shall be persistently stored in a portable form such as XML.	5
4.	Policies shall support automated configuration to reduce administration needs.	8
5.	Policies shall seamlessly plug into components and environment contexts.	7

Figure 2: Goals / Requirements Table

3. Use Cases

Figure 2 illustrates the use cases for policy creation and deployment. The use cases are described in the accompanying text.

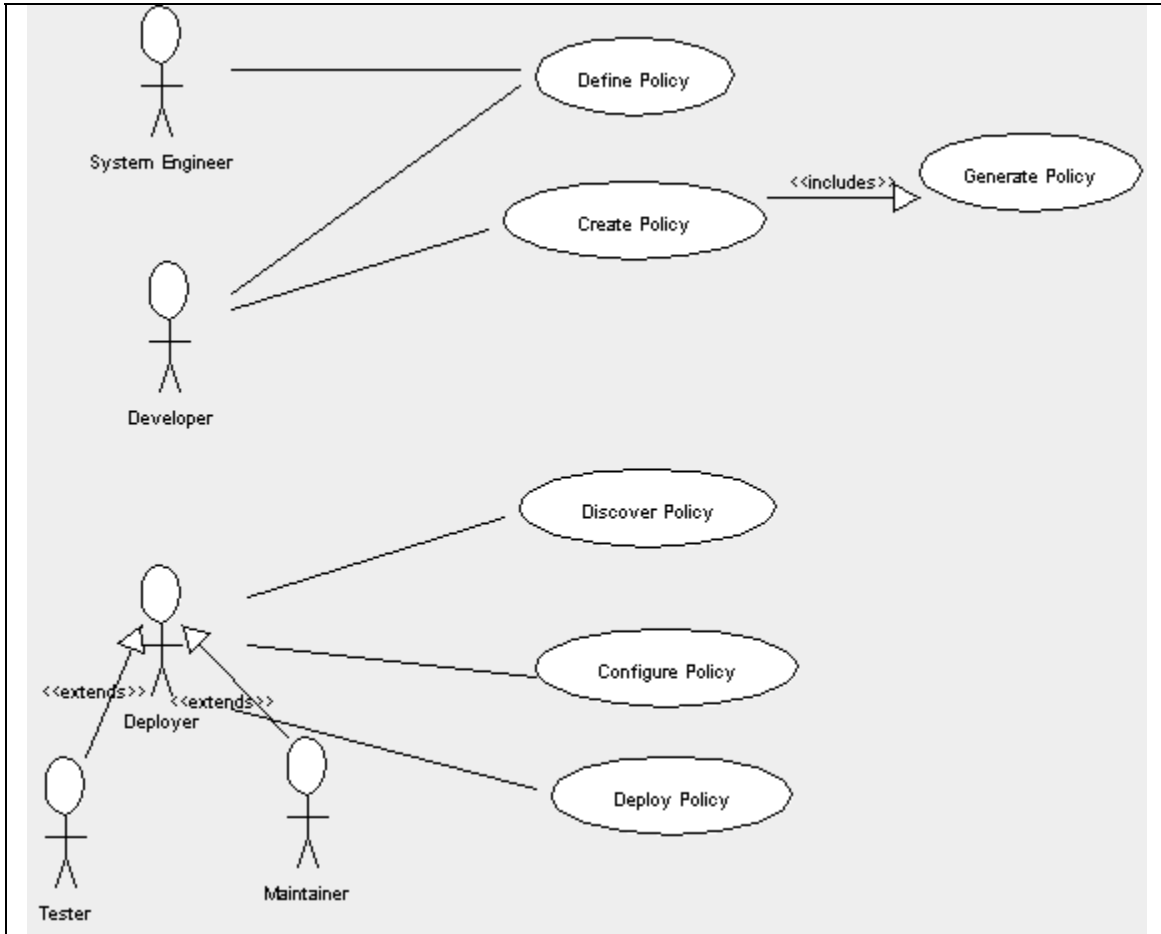


Figure 3: Policy Use Cases

3.1 Define Policy

In this use case, either a system engineer or a developer defines the configuration information for a particular application domain that should be stored in a policy. Example domains from Openwings include service lookup in Component Services and load balancing in Container Services.

3.2 Create Policy

In this use case, a developer creates a policy. This really means defining a policy interface.

3.3 Generate Policy

In this use case, a developer uses a policy generator tool to create a policy implementation based on the policy interface.

3.4 Discover Policy

In this use case, a particular policy is located by a tester, a deployer or a maintainer.

3.5 Configure Policy

In this use case, a particular policy is configured by a tester, a deployer or a maintainer.

3.6 Deploy Policy

A tester, a deployer or a maintainer deploys a policy by packaging it in a way that enables the system to access it.

4. Architecture

4.1 Overview

Traditional software systems have taken a static approach to using policies to convey environmental information. In these systems, policy data is typically stored in configuration files that are read when the system starts. The Openwings strategy is to define policies that are adaptive, thereby enabling the system to adapt to changes in its environment. In Openwings, policies are implemented as self-contained software components and therefore have the benefits of a component architecture. These benefits include support for attributes and methods, support for hierarchical organization, and support for dynamic modification.

As in the traditional approach, policy data is stored in a file (or some other persistent storage) and read when the policy is loaded by the system. In order to impose structure and validation on policy data, it is stored in a portable format such as XML. If a policy is manageable, its attribute values can be modified and reapplied at run time using a management application.

Applications interact with a policy through the policy's interface. This allows the application to be shielded from the implementation details and the underlying data representation. To request a policy, applications simply pass an interface to a policy loader. The loader will return an object that implements the interface, or an indication that the request could not be satisfied.

Policies need the ability to plug into a variety of environments, including environments that may not have been defined at the time the policy was created. To facilitate this, policies are designed to comply with the JavaBeans component model. JavaBeans are reusable software components that are implemented according to a fixed convention; this enables them to be manipulated through any JavaBean container (e.g., a BeanBox). The JavaBean model requires that the attributes, methods and events that define a bean follow design patterns from the JavaBean specification so that bean containers can use a predefined set of rules for interacting with the bean. For the simplest JavaBeans, the design patterns prescribe naming conventions that enable the default bean introspector to discover the bean's attributes, methods and events. More advanced JavaBeans implement `java.beans.BeanInfo` classes to explicitly describe their attributes, methods and events. `BeanInfo` classes provide support for more advanced introspection than the default introspector allows. The extra information in `BeanInfo` files can be used as an aid to adapting dynamic policies. Beans may also provide customizers or property editors that support interaction with the bean through a graphical user interface (GUI). This feature will provide a foundation for the management of policies through GUIs. By following the JavaBean convention, policy implementers ensure that their policies can be deployed in environments that were not envisioned at the time the policy was designed.

4.2 Policy Architecture

In its simplest form, a policy consists of three things: an interface, a class implementing the interface and a configuration file format definition. Application or framework developers perform the task of developing Policy interfaces. Components interact with the policy through its interface. The policy implementation class initializes the policy attributes based on the file format. A key element in the architecture is the policy generator, which takes a Java interface and creates a file format for the policy storage (such as an XML schema) and a policy interface implementation that can save and load policy data according to the file format. These elements are shown in the figure below.

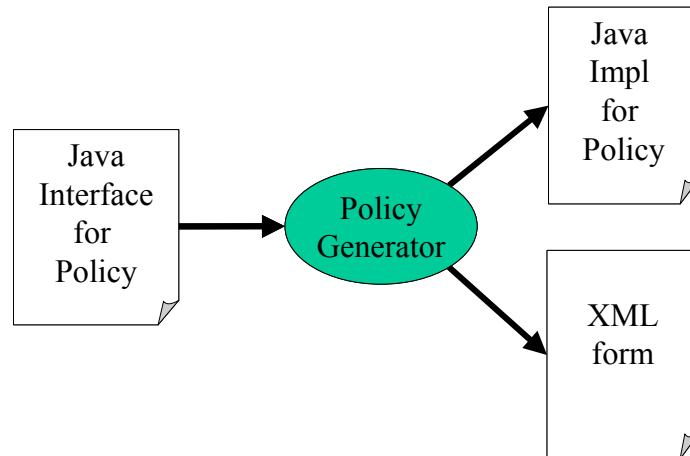


Figure 4: Elements of a Policy

4.3 Policy Naming

Policy naming conventions define the mapping from policy types to policy interfaces, from policy interfaces to fixed implementation classes, as well as the mapping from policy interfaces to policy data files.

The policy interface name for a policy type is formed by taking the type and appending “Policy”. Thus, for the policy type `My`, the policy interface name is `MyPolicy`. The policy class for a given policy interface is formed by appending “Impl” to the interface name. Thus, the class implementing the `MyPolicy` interface is `MyPolicyImpl`. However, there may be cases where a policy interface and policy implementation are defined in separate Java packages. This pattern occurs frequently in the Openwings reference implementation, where policy interfaces are declared in `net.openwings.*` packages and the implementations are generated in `com.gd.openwings.*` packages. In these cases, a Java property is used to describe the mapping from interface to implementation. The convention is as follows: the property name is the policy interface name, and the value is the name of the package containing the implementation. Example:

```

net.openwings.component.ComponentPolicy=
  com.gd.openwings.component.policy
  
```

The naming convention for policy data is only slightly more complex. There may be multiple data sets (e.g., files) for a particular policy interface so support is provided for

the application to request data by name. The policy data for an interface is found by appending the file format, such as “.xml” to the interface name *or* by appending a name specified by the requesting application to the interface, then appending the file format. For example, the data for `MyPolicy` can be referenced as `MyPolicy.xml` or as `MyPolicyDataSet1.xml`, `MyPolicyDataSet2.xml`, and so forth).

4.4 Policy Attributes

Policy attributes are defined in accordance with the JavaBean design patterns. Therefore, the definition of an attribute includes the attribute itself plus the accessor methods. The naming convention for accessor methods is “get” or “set” prepended to the name of the attribute, with the first letter of the attribute name capitalized. For example, accessor methods for the attribute `user` shall be named `getUser` and `setUser`. If an attribute is read-only, the “set” accessor should not be exposed. Likewise, if an attribute is write-only, its “get” accessor should not be exposed.

4.5 Policy Loading

The naming convention specifies how policy interface names are related to policy implementation names and policy file names. This convention is used to facilitate the association of a given policy object with an actual file location. In some cases, the actual file location of a policy and the implementation class are known to a user of Policy Services. However, the more usual case is that the application does not know the policy implementation class or the file location. To help address this issue, the architecture introduces the concept of a policy loader, shown in the figure below.

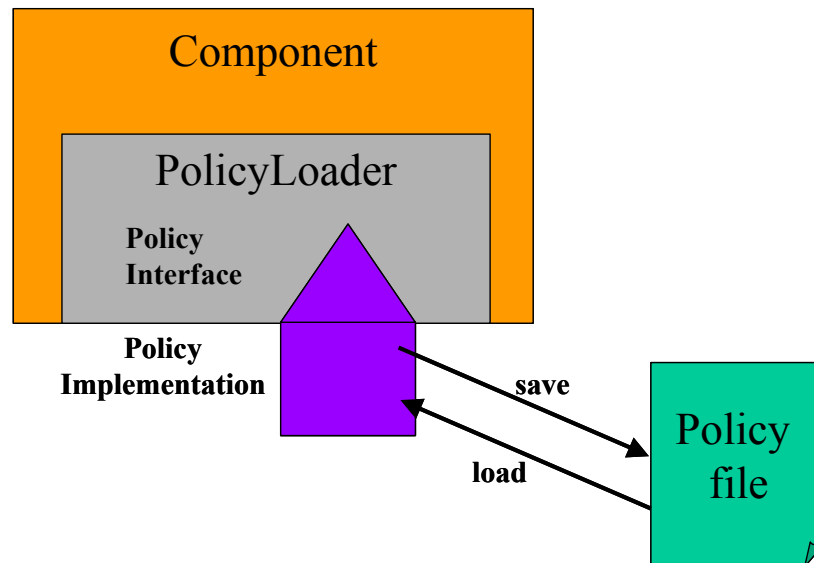


Figure 5: Policy Loader Architecture

A policy loader takes a policy interface and an optional policy name (as described above) and locates an appropriate policy implementation class and policy file. The policy loader reads properties that map policy interfaces to implementation classes. The policy loader also provides a programmatic interface to add mappings. The concept of a policy path is

introduced to help locate policy files. The path is a space-separated list of directories, defined by the property `net.openwings.policy.path`.

Most applications that use Policy Services will be Openwings components, packaged and deployed as described in the Openwings Install Service Specification. This specification provides for packaging components and describing inter-component dependencies. Each component has a list of properties, and can thus define their own policy paths. Components can also reference properties from other components; therefore the policy path of a component can incorporate the policy path of other components. See the Install Service Specification for more information on packaging components.

4.6 Policy Services and Context Services

Openwings Context Services provide a mechanism for logically grouping system components. Contexts provide the environmental context that determines how applications should run. Contexts may be nested, thereby providing a hierarchical grouping of system components. Contexts can contain policies and Java properties. Therefore Policies help to provide the bridge between applications and the context that they are deployed in. See the Openwings Context Specification for more information.

5. Interfaces

The interfaces defined for Openwings policies lay out a framework for defining policies, factories for obtaining policies and, generation tools for creating policy implementations. Figure 6 contains a table listing interfaces and the roles that use them. The role of the User refers to someone who uses an implementation of the policy specification; the role of developer refers to someone implementing the specification.

Interface	Role
Policy	User, Developer
PolicyException	User, Developer
PolicyLoader	User, Developer
PolicyLoaderFactory	User, Developer
PolicyGenerator	Developer

Figure 6: Policy Interface Roles.

Figure 7 presents a UML class diagram to sketch out how the Openwings policy interfaces interrelate. In the diagram, dashed lines with arrowheads indicate the implementation of an interface and solid lines with arrowheads indicate inheritance. As shown in the figure, policies are loaded via a factory.

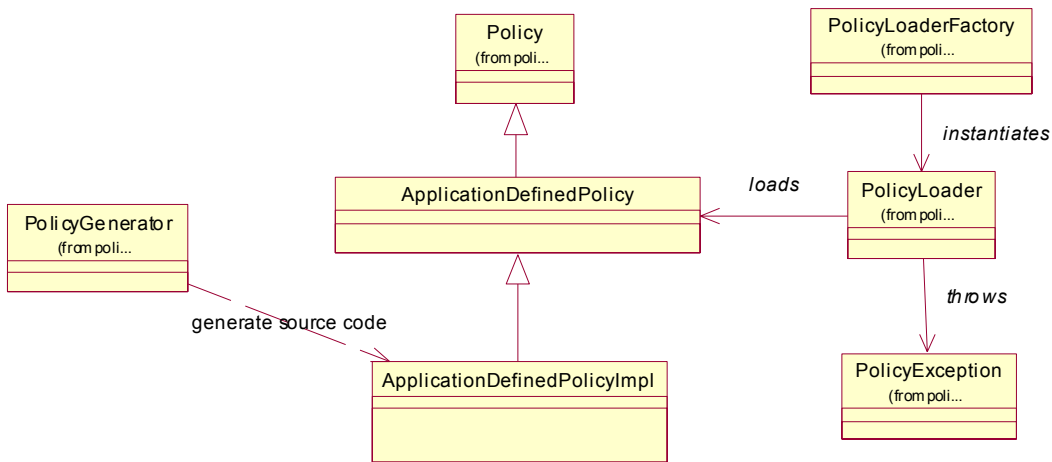


Figure 7: Policy Interfaces.

5.1 net.openwings.policy.Policy

public interface **Policy** extends Serializable

This interface provides the basis for defining policies in Openwings. Interfaces for custom policies extend this interface.

Method Summary	
String	getDescription() Get the human-readable description for this <code>Policy</code> .
URL	getLastLoadURL() Returns a <code>URL</code> corresponding to the last location from which the policy was loaded.
String	getName() Get the name of this <code>Policy</code> .
String[]	getPolicyAttributes() Returns an array of strings that contains the attributes for the policy in the order defined in the schema.
String[]	getPolicyElements() Returns an array of strings that contains the elements for the policy in the order defined in the schema.
boolean	isPolicyUpdated() Determines if this policy has been updated and is in need of being saved.
void	loadPolicy(File file) Load the policy with configuration data from the <code>File</code> .
void	loadPolicy(URL url) Load the policy with configuration data from the <code>URL</code> .
void	savePolicy() Save the policy configuration data to the file or <code>URL</code> where it was originally loaded from.
void	savePolicy(File file) Save the policy configuration data to the <code>File</code> .
void	savePolicy(URL url) Save the policy configuration data to the <code>URL</code> .
void	setDescription(String description) Set a human-readable description for this <code>Policy</code> .
void	setName(String name) Assign a name to this <code>Policy</code> .
void	setPolicyUpdated(boolean policyUpdated) Sets the update status of this policy.

Method Detail

5.1.1 loadPolicy

```
public void loadPolicy(URL url) throws PolicyException
```

Load the policy with configuration data from the `URL`. The values from the configuration data overwrite any previous values stored by the `Policy` object.

Parameters:

`url` - The location of the policy configuration data.

Throws:

throws - A `PolicyException` that wraps any exception generated when attempting to load the policy data.

5.1.2 loadPolicy

```
public void loadPolicy(File file) throws PolicyException
```

Load the policy with configuration data from the `File`. The values from the configuration data overwrite any previous values stored by the `Policy` object.

Parameters:

`file` - The file for the policy configuration data.

Throws:

throws - A `PolicyException` that wraps any exception generated when attempting to load the policy data.

5.1.3 savePolicy

```
public void savePolicy(URL url)
           throws PolicyException
```

Save the policy configuration data to the `URL`.

Parameters:

`url` - The location for the policy configuration data.

Throws:

throws - A `PolicyException` that wraps any exception generated when attempting to load the policy data.

5.1.4 savePolicy

```
public void savePolicy(File file)
           throws PolicyException
```

Save the policy configuration data to the `File`.

Parameters:

file - The file for the policy configuration data.

Throws:

throws - A `PolicyException` that wraps any exception generated when attempting to load the policy data.

5.1.5 savePolicy

```
public void savePolicy()  
    throws PolicyException
```

Save the policy configuration data to the file or URL where it was originally loaded from.

Throws:

throws - A `PolicyException` that wraps any exception generated when attempting to load the policy data.

5.1.6 setName

```
public void setName(String name)
```

Assign a name to this `Policy`. Naming is used to distinguish between policies of the same type.

Parameters:

name - The name to be used in identifying the `Policy`.

5.1.7 getName

```
public String getName()
```

Get the name of this `Policy`. Naming is used to distinguish between policies of the same type.

Returns:

The name of the `Policy`.

5.1.8 setDescription

```
public void setDescription(String description)
```

Set a human-readable description for this `Policy`. The description should be suitable for display in a graphical user interface.

Parameters:

description - The text to be used in describing the `Policy`.

5.1.9 getDescription

```
public String getDescription()
```

Get the human-readable description for this `Policy`. The description should be suitable for display in a graphical user interface.

Returns:

The text to be used in describing the `Policy`.

5.1.10 `setPolicyUpdated`

```
public void setPolicyUpdated(boolean policyUpdated)
```

Sets the update status of this policy.

Parameters:

`true` - if the policy has been updated; `false` otherwise.

5.1.11 `isPolicyUpdated`

```
public boolean isPolicyUpdated()
```

Determines if this policy has been updated and is in need of being saved.

Returns:

`true` if the policy has been updated; `false` otherwise.

5.1.12 `getPolicyAttributes`

```
public String[] getPolicyAttributes()
```

Returns an array of strings that contains the attributes for the policy in the order defined in the schema.

Returns:

array of strings containing attributes

5.1.13 `getPolicyElements`

```
public String[] getPolicyElements()
```

Returns an array of strings that contains the elements for the policy in the order defined in the schema.

Returns:

array of strings containing elements

5.1.14 `getLastLoadURL`

```
public URL getLastLoadURL()
```

Returns a `URL` corresponding to the last location from which the policy was loaded.

Returns:

The source of the xml file used to populate the policy

5.2 net.openwings.policy.PolicyException

```
public class PolicyException extends Exception
```

This class defines an exception specific to policies. This exception may be used to encapsulate another exception generated during policy operations.

Constructor Summary	
PolicyException()	Constructs a <code>PolicyException</code> with no specified detail message.
PolicyException (Exception e, String message)	Constructs a new <code>PolicyException</code> .
PolicyException (String message)	Constructs a <code>PolicyException</code> with the specified detail message.

Method Summary	
Exception	getEmbeddedException() Get the <code>Exception</code> that triggered this exception.
String	getMessage() Returns the error message string of this throwable object.
void	printStackTrace() Prints this <code>Throwable</code> and its backtrace to the standard error stream.
void	printStackTrace (PrintStream s) Prints this <code>Throwable</code> and its backtrace to the standard error stream.
void	printStackTrace (PrintWriter s) Prints this <code>Throwable</code> and its backtrace to the standard error stream.

5.2.1 PolicyException

```
public PolicyException()
```

Constructs a `PolicyException` with no specified detail message.

5.2.2 PolicyException

```
public PolicyException(String message)
```

Constructs a `PolicyException` with the specified detail message.

Parameters:

message - the detail message

5.2.3 PolicyException

```
public PolicyException(Exception e,  
                       String message)
```

Constructs a new `PolicyException`.

Parameters:

`e` - The `Exception` that triggered this exception. May be `null`.
`message` - The text message to be associated with this exception.

5.2.4 getEmbeddedException

```
public Exception getEmbeddedException()
```

Get the `Exception` that triggered this exception.

Returns:

The `Exception` that triggered this exception. May be `null`.

5.2.5 getMessage

```
public String getMessage()
```

Returns the error message string of this throwable object. Add the embedded exception message to the message.

Overrides:

`getMessage` in class `Throwable`

5.2.6 printStackTrace

```
public void printStackTrace()
```

Prints this `Throwable` and its backtrace to the standard error stream. This method prints a stack trace for this `Throwable` object on the error output stream that is the value of the field `System.err`. If an embedded exception is present, will print out the embedded exception stack trace, otherwise will print policy exception stack trace.

Overrides:

`printStackTrace` in class `Throwable`

5.2.7 printStackTrace

```
public void printStackTrace(PrintStream s)
```

Prints this `Throwable` and its backtrace to the standard error stream. This method prints a stack trace for this `Throwable` object on the error output stream that is the value of the field `System.err`. If an embedded exception is present, will print out the embedded exception stack trace, otherwise will print policy exception stack trace.

Overrides:

`printStackTrace` in class `Throwable`

Parameters:

`s` - `PrintStream` to use for output

5.2.8 printStackTrace

```
public void printStackTrace(PrintWriter s)
```

Prints this `Throwable` and its backtrace to the standard error stream. This method prints a stack trace for this `Throwable` object on the error output stream that is the value of the field `System.err`. If an embedded exception is present, will print out the embedded exception stack trace, otherwise will print policy exception stack trace.

Overrides:

`printStackTrace` in class `Throwable`

Parameters:

`s` - `PrintWriter` to use for output

5.3 net.openwings.policy.PolicyLoader

```
public interface PolicyLoader
```

This interface describes a utility that instantiates `Policy` objects and loads their saved state. Unless the location of the policy file is explicitly stated, policies are loaded from the policy path, defined by the property `net.openwings.policy.path`.

An implementation of this interface is obtained using the `PolicyLoaderFactory`. An Openwings Component can obtain a `PolicyLoader` object by calling the `net.openwings.component.ComponentComplex.getPolicyLoader()` method.

Method Summary

void	addPackageMapping (Class <code>policyType</code> , String <code>altPackage</code>) This method adds a mapping of a <code>Policy</code> interface to a package where a suitable implementation class may be found.
Policy	createNewPolicy (Class <code>policyType</code>) Create a new policy object that implements the interface specified by the <code>policyType</code> parameter.
Policy	getPolicy (Class <code>policyType</code> , File <code>configFile</code>) Get a policy object that implements the interface specified by the <code>policyType</code> parameter.
Policy	getPolicy (Class <code>policyType</code> , String <code>name</code>) Get a policy object that implements the interface specified by the <code>policyType</code> parameter.

Policy	getPolicy (Class policyType, String name, ClassLoader loader) Get a policy object that implements the interface specified by the policyType parameter.
Policy	getPolicy (Class policyType, URL location) Get a policy object that implements the interface specified by the policyType parameter.
Iterator	packageMapping (Class policyType) Get an Iterator over the mappings of a Policy class to an implementation package.

5.3.1 addPackageMapping

```
public void addPackageMapping(Class policyType,
                               String altPackage)
```

This method adds a mapping of a `Policy` interface to a package where a suitable implementation class may be found. No check is made to verify whether the package contains a suitable class at the time the mapping is made.

The mapping of interfaces to implementations is also achievable via Java properties. For example, to specify that the implementation of interface `mypackage.MyPolicy` is found in package `myimplpackage`, set a property with name equal to the interface name, and value equal to the package name. On the command line, this would look something like this:

```
-Dmypackage.MyPolicy=myimplpackage
```

To set multiple package mappings, use a space-separated list, enclosed in quotes:

```
-Dmypackage.MyPolicy="myimplpackage myotherimplpackage"
```

Parameters:

`policyType` - The interface of the `Policy`.

`altPackage` - the name of a package containing an implementation of the policy interface specified in `policyType`.

5.3.2 packageMapping

```
public Iterator packageMapping(Class policyType)
```

Get an `Iterator` over the mappings of a `Policy` class to an implementation package.

Parameters:

`policyType` - The interface of the `Policy`.

Returns:

An `Iterator` over the names of packages that may contain an implementation of the policy interface, or `null` if there is no mapping.

5.3.3 getPolicy

```
public Policy getPolicy(Class policyType,  
                        URL location)  
    throws PolicyException
```

Get a policy object that implements the interface specified by the `policyType` parameter. The name of the policy implementation class is created by appending "Impl" to the name of the interface given in the `policyType` argument. For example, for an interface of the class `MyPolicy`, the loader will attempt to load `MyPolicyImpl`.

The `Policy` object that is returned will be initialized with its configuration data. The configuration data file must be at the location specified by `URL location`.

Parameters:

`policyType` - The interface that the returned `Policy` object must support.

`location` - an URL to the policy object

Returns:

The `Policy` object, or null if none could be found.

Throws:

`PolicyException` - An exception indicating something went wrong in trying to load the `Policy` object or its configuration data. This exception may wrap a core JVM exception.

5.3.4 getPolicy

```
public Policy getPolicy(Class policyType,  
                        File configFile)  
    throws PolicyException
```

Get a policy object that implements the interface specified by the `policyType` parameter. The name of the policy implementation class is created by appending "Impl" to the name of the interface given in the `policyType` argument. For example, for an interface of the class `MyPolicy`, the loader will attempt to load `MyPolicyImpl`.

The `Policy` object that is returned will be initialized with its configuration data. The configuration data file must be at the location specified by `File configFile`.

Parameters:

`policyType` - The interface that the returned `Policy` object must support.

`configFile` - a file to the policy object

Returns:

The `Policy` object, or null if none could be found.

Throws:

`PolicyException` - An exception indicating something went wrong in trying to load the `Policy` object or its configuration data. This exception may wrap a core JVM exception.

5.3.5 getPolicy

```
public Policy getPolicy(Class policyType,  
                        String name)  
    throws PolicyException
```

Get a policy object that implements the interface specified by the `policyType` parameter. The name of the policy implementation class is created by appending "Impl" to the name of the interface given in the `policyType` argument. For example, for an interface of the class `MyPolicy`, the loader will attempt to load `MyPolicyImpl`.

The `Policy` object that is returned will be initialized with its configuration data. Configuration data is loaded from the same location as the implementation class. Configuration data will be loaded from a resource whose name is formed by concatenating the name of the interface class, the name parameter, and ".xml". If the interface class is `MyPolicy` and name is `Data1`, the configuration data will be loaded from `MyPolicyData1.xml`. The configuration data file must be available on the policy path.

If there is no mapping of the policy class to one or more packages, the loader will attempt to load the implementation class from the same package as the interface.

Parameters:

`policyType` - The interface that the returned `Policy` object must support.
`name` - A name to identify this instance of the policy. May be null.

Returns:

The `Policy` object, or null if none could be found.

Throws:

`PolicyException` - An exception indicating something went wrong in trying to load the `Policy` object or its configuration data. This exception may wrap a core JVM exception.

5.3.6 getPolicy

```
public Policy getPolicy(Class policyType,  
                        String name,  
                        ClassLoader loader)  
    throws PolicyException
```

Get a policy object that implements the interface specified by the `policyType` parameter. The name of the policy implementation class is created by appending "Impl" to the name of the interface given in the `policyType` argument. For example, for an interface of the class `MyPolicy`, the loader will attempt to load `MyPolicyImpl`. The class loader, if specified, will be used to load the implementation class and its configuration data.

The `Policy` object that is returned will be initialized with its configuration data. Configuration data is loaded from the same location as the implementation class. Configuration data will be loaded from a resource whose name is formed by concatenating the name of the interface class, the name parameter, and ".xml". If the

interface class is `MyPolicy` and name is `Data1`, the configuration data will be loaded from `MyPolicyData1.xml`. The configuration data file must be available on the policy path.

If there is no mapping of the policy class to one or more packages, the loader will attempt to load the implementation class from the same package as the interface.

Parameters:

`policyType` - The interface that the returned `Policy` object must support.

`name` - A name to identify this instance of the policy. May be null.

`loader` - The `ClassLoader` to use when loading the `Policy` and its configuration data. If this is null, the system `ClassLoader` will be used.

Returns:

The `Policy` object, or null if none could be found.

Throws:

`PolicyException` - An exception indicating something went wrong in trying to load the `Policy` object or its configuration data. This exception may wrap a core JVM exception.

5.3.7 createNewPolicy

```
public Policy createNewPolicy(Class policyType)
                        throws PolicyException
```

Create a new policy object that implements the interface specified by the `policyType` parameter. The name of the policy implementation class is created by appending "Impl" to the name of the interface given in the `policyType` argument. For example, for an interface of the class `MyPolicy`, the loader will attempt to load `MyPolicyImpl`.

The `Policy` object that is returned will be initialized with default values.

If there is no mapping of the policy class to one or more packages, the loader will attempt to load the implementation class from the same package as the interface.

Parameters:

`policyType` - The interface that the returned `Policy` object must support.

Returns:

The `Policy` object, or null if none could be found.

Throws:

`PolicyException` - An exception indicating something went wrong in trying to create the `Policy` object. This exception may wrap a core JVM exception.

5.4 net.openwings.policy.PolicyLoaderFactory

```
public final class PolicyLoaderFactory extends Object
```

This class provides a way to obtain an implementation of `PolicyLoader`. The choice of `PolicyLoader` implementation class is set using the `net.openwings.policy.implementation` property.

Method Summary	
<code>static PolicyLoader</code>	<code>getPolicyLoader()</code> This method is used to obtain a <code>PolicyLoader</code> .

5.4.1 getPolicyLoader

```
public static PolicyLoader getPolicyLoader()
```

This method is used to obtain a `PolicyLoader`.

Returns:

object implementing the `PolicyLoader` interface.

5.5 net.openwings.policy.PolicyGenerator

```
public interface PolicyGenerator
```

This interface describes a utility that can be used to generate policies.

Method Summary	
<code>void</code>	<code>generatePolicy(String className)</code> Generate policy source code and configuration data files using the information from the specified interface.

5.5.1 generatePolicy

```
public void generatePolicy(String className)
    throws ClassNotFoundException,
    PolicyException
```

Generate policy source code and configuration data files using the information from the specified interface.

Parameters:

`className` - The name of the interface.

Throws:

`ClassNotFoundException` - thrown if `className` is not a valid class

`PolicyException` - an error has occurred while generating a policy

6. Examples

6.1 Sample Policy Interface

A sample policy interface is shown in the figure below.

```
package com.gd.sample;

import URL;

/**
 * This interface is simply to illustrate the policy generation
 * process. Beyond that, it has no real utility.
 */
public interface SamplePolicy extends net.openwings.policy.Policy
{
    public int getValue();
    public void setValue(int value);

    public boolean getBool();
    public void setBool(boolean value);

    public URL getOutURL();
    public void setOutURL(URL url);
    public URL getErrURL();
}
```

Figure 8: A sample policy interface.

6.2 Openwings Policy Instances

There are several policy interfaces defined within the Openwings architecture itself:

- `net.openwings.install.InstallableComponentDescriptorPolicy`

The Install Service specification defines this policy interface, it is used to store a descriptor for an Openwings component.

- `net.openwings.management.ManagementPolicy`

The Management Services specification defines this policy interface, it is used to store information about default management beans and adapter plugins to the management framework.

- `net.openwings.component.UseServicePolicy`
- `net.openwings.component.ProvideServicePolicy`
- `net.openwings.component.EventServicePolicy`
- `net.openwings.component.PublishServicePolicy`

The Component Services specification defines these policy interfaces, they are used to store special configuration information about synchronous and

asynchronous service interactions, such as service attributes and user interfaces. Using these policies makes service interactions between components more configurable.

These policies and others are used throughout the Openwings architecture to provide configuration settings and pluggable behavior.

6.3 The Openwings Tutorial

The Openwings Tutorial contains an entire lesson on generating and using policies. See the lesson “Using Policies” under the tutorial trail entitled “Developing Components”. The tutorial is available at <http://www.openwings.org>.

7. Tools

7.1 Policy Generator

The Openwings reference implementation contains a Policy Generator. This tool is used to generate policies from interface definitions. Given a compiled interface, the generator uses introspection to create implementation files, the XML policy definition (in schema or DTD form), a sample XML data file that complies with the aforementioned definition, and parsing code to initialize the policy implementation object with XML data.

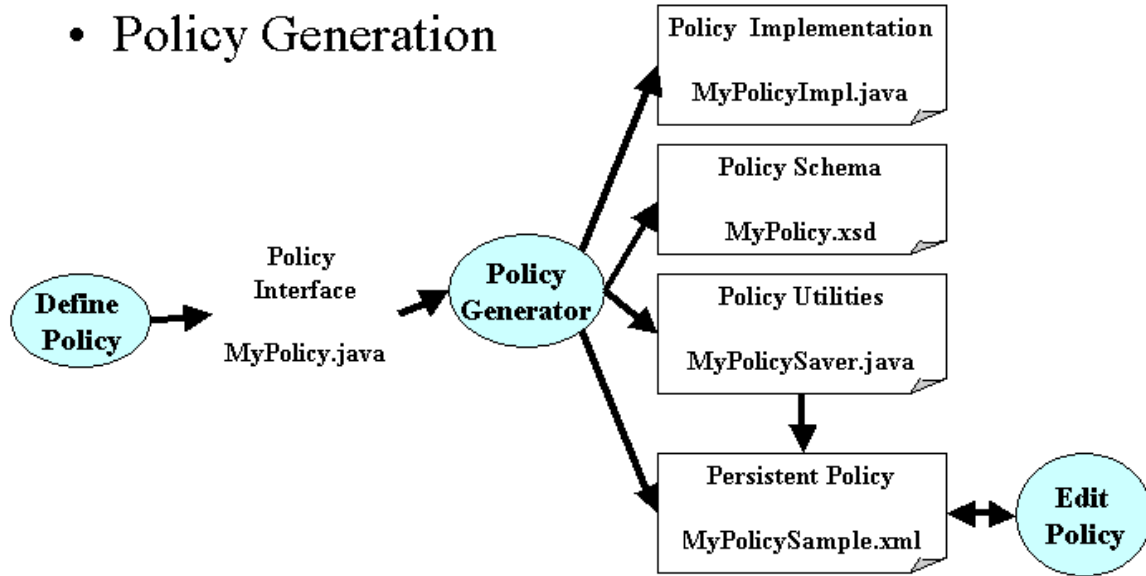


Figure 9: Policy Generator Reference Implementation

7.2 Policy Editors

The Openwings reference implementation contains a policy editor that edits `InstallableComponentDescriptorPolicies`. See the Install Service Specification for more information on the “ICD Editor”.

Future plans include development of a Generic Policy Editor that will allow any policy file to be edited and saved.

8. Compliance

The compliance checklist for this specification is as follows.

#	ITEM
1.	Shall comply with the API specified in this document.
2.	Shall pass the Openwings policy validation suite.

Figure 10: Compliance Table.

9. Futures

This portion of the document discusses concepts that may be considered for future versions of this specification.

9.1 Attribute Change Notification

Since attributes in the Openwings Policy architecture are modeled after JavaBeans, Policy attributes can be bound and constrained in the same way that JavaBeans properties can be bound and constrained. Attributes are bound when one or more `java.beans.PropertyChangeListener` objects are registered with the policy for information about changes in a particular attribute. When a policy attribute changes, all change listeners are sent an event informing them of the change. Attributes are constrained when one or more `java.beans.VetoableChangeListener` objects are registered with the policy for information about changes in a particular attribute. When an attribute is going to be changed, an event is sent to the `VetoableChangeListener` and they have the ability to veto (i.e., prevent) the change. This feature would involve adding methods to the `Policy` interface for adding change listeners.

9.2 Policy Management

Policies may be managed or unmanaged. Unmanaged policies do not support the modification of their attributes and operations at run time. However, there is often a part of policy configuration data that can only be defined at runtime. Policies should be managed in a way that is compliant with the Openwings Management Specification. By allowing policies to be operated upon through an Openwings Management Bean (`MBean`), runtime data can be passed into a policy and combined with other data to perform some configuration action.

9.3 Dynamic Policies

Dynamic policies are fully adaptive; their attributes and behavior can be modified at run time. New policies may be built dynamically from a set of existing policies. The adaptive nature of dynamic policies enables a variety of application strategies including the use of rules to combine a set of policies into a single, consistent policy for a given environment. If a dynamic policy is manageable, its attributes and its behaviors can be modified using a management application.

Dynamic policies can be composed of multiple objects and multiple sets of configuration data. Despite their greater complexity, dynamic policies can still be accessed through a single interface, just like fixed policies. This enables application development without regard to whether the application will use fixed or dynamic policies. The underlying implementation may use multiple classes to implement the behavior of a single interface.

9.4 Policy Resolution and Negotiation

When multiple policies exist in an Openwings context, complex interactions can result. Across nested contexts, there may be multiple policies that apply to a component and these policies may contain conflicting information. Policy resolution is used to

determine which policies in a set of contexts apply to a component. Policy negotiation is used to reconcile the differences in policies into a consistent policy. The rules for policy resolution and negotiation may themselves be contained in policies. A policy must be able to revert back to its previous state when it is pulled out of a context.

Policy resolution and negotiation should be invisible to the user of a policy. The user of a policy should be able to request a policy that complies with a specified interface and the policy provider should take the steps necessary to return either a policy that satisfies the request, or an indication that the request could not be satisfied.

10. References and Further Reading

1. Openwings Architecture Whitepaper, <http://www.openwings.org/download/specs/openwingswp.pdf>
2. Openwings Architecture Specification, http://www.openwings.org/download/specs/Openwings_Architecture_Description.pdf
3. Openwings Context Specification, http://www.openwings.org/download/specs/Openwings_Context.pdf
4. Openwings Interface Definition Specification, http://www.openwings.org/download/specs/openwings_interface.pdf
5. XML, <http://www.w3.org/XML/>
6. JavaBeans Specification, <http://java.sun.com/products/javabeans/>
7. IETF Policy Working Group drafts, <http://www.ietf.org/html.charters/policy-charter.html>