EFI ToolShed Project Functional Specification

Jack Hodges
January 17, 2005

ToolShed Project Team: Jack Hodges, Gaoxiang Xu, Rajesh Poddar

Contents

Abstract **3**
Discussion **4**
Objective 4
Overview of Problem 4
Example 4
Constraints 5
Possible Solution Strategies 5
Proposed Solution Strategy 6
Project Deliverables 7
ToolShed Project Footprint 7
Outline 8
ToolShed Platform Architecture **9**
Tools 9
Mechanisms 10
Approach and Background 10
ToolShed Application Architecture 15
Package and Component Interactions 16
Workflow Mechanisms 18
Package and Class Identification 19
XML Schema for Model Validation 20
XML Packages 27
Models 30
Views 39
UI Package 43
Beans Package 44
Actions 45
Application-Specific Packages (appls) 46
Data Validation 47
Generate-Time Validation 47
Runtime Validation 47
Constraint Representation and Validation (should be level 4 heading) 72
Constraint Representation with RuleML 73
RuleML Parsing to Java Rules 76
Rule-Based Processing 76
Constraint Validation 77
Server Communications 77
What is SOAP, and Why Use It? 79
SOAP in ToolShed 81
SOAP Requirements in ToolShed 85
Localization and Internationalization 86
General Localization Approach 86
String Acquisition 86
String Translation 87
Design Requirements **89**
Use Cases 89
Actors 89
User-Driven Use Cases 89
USER SelectTSSetup into TS 95
User Interface Elements 95
JTabbedPanel 96

JLabel Component 97
JButton Component 98
JCheckbox Component 99
Jlist Component 100
JTextField 101
JComboBox Component 102
JRadioButton Component 103
JTable Component 104
Implementation and Resource Requirements 106
Phased Implementation Plan 106
Resource Allocation Requirements 107
References 108
Tools Used in ToolShed Design and Implementation 108
Links and Reference Documents 108
Links to ToolShed Documents 109

Abstract

This document serves to define the basic market and user requirements for the tools that will allow EFI customers to use Fiery Setup and Installation while enabling EFI project managers to configure the setup/installation user interface without direct engineering intervention. This project will consist of a redesign/redevelopment effort targeted at replacing the current WebTools client and server with new versions that perform the same tasks as before but in a more effective/efficient manner. The new model introduces a new [default] client look and feel but incorporates a consistent functionality with the existing client.

The requirements/constraints for the project will be specified by the combination of the functional use cases for the client, the user interface (UI) elements, the storyboards of pages that will implement the use cases pertinent to the tool, the object model, and the functional mechanisms in the tool. This document represents the engineering response to the Fiery System 5.5 Server Product Specification, dated 10/21/2002, under remote setup, pages 39-61.

This is a long and complex document and it is not expected that everyone who picks it up will need to read all of it. Below is a table giving an idea as to which sections will be of interest to which groups. It is only a guideline.

Who Should Read	Document Section	Pages
Marketing, Planners	Abstract Discussion	3 4 - 8
Designers	Abstract Discussion Design Requirements (UI elements)	3 4 – 8 95 - 101
Engineering Managers	Abstract Discussion Architecture (tools, mechanisms, approach) Data Validation Constraint Validation Communications Localization and Internationalization Implementation References	3 4 – 8 9 – 13 47, 48 72 77 86, 85 106, 105 108
QA	Design Requirements (actors, use cases)	89-93
Development Engineers	Entire document	

Discussion

Objective

The ToolShed project is intended to provide content usability support by providing the user with a mechanism for installing and setting up a fiery server, and to provide interface configuration support by providing staff with a mechanism for configuring the user interface.

Overview of Problem

There are seven motivations for redesigning/developing webtools related applications, as follows:

- Multiple software versions exist for different products, making maintenance difficult and time consuming.
- Product managers want to have interface control over the software look and feel for different clients, but often making these gratuitous changes is problematic for the overall codebase.
- Integration issues with the Fiery server continue to be problematic, in that changes to the server require updates and changes to the client software that should not be required.
- Software engineers are required to make adhoc changes to the client software that reduce its coherence, integrity, and performance.
- General solutions require larger footprints than the client is allowed to have.
- Changes to server or PPD definitions require changes in how the client is accessed, making it difficult to maintain the software for multiple clients.
- It is desired to have a single application for all instances of WebTools and WebSetup, eliminating the Fiery Setup Bar.
- The WebTools 1 codebase needs to be retired, because it was developed without any coherent architectural approach, has been modified over the years so that it is no longer coherent to whatever design approach was used, makes use of outdated components and programming mechanisms, makes use of brittle custom components that cannot be easily updated or extended, makes use of embedded logic, implements some of the data statically or using hardcoded values instead of property managers, has no consistent style or documentation. All of these factors render it very difficult to maintain the software.

Example

An illustrative example of this problem is one in which a new product is developed for a customer, and the customer wants to add/remove buttons from the main menu, or wants to customize the user interface for its particular needs. Using the current methodology, the former functionality can be achieved, as WebTools has a setup mechanism that can be configured at the server. The user interface, however, is quite restrictive in what can be done in the way of customization. For example, were the product developers interested in moving the location of buttons or tabs, or wanted to change the color scheme, or wanted to add branding to the entire application, it would simply be (practically speaking) impossible, because the same software is supporting all applications and cannot be modified to support just one. A unique build would have to be created and maintained, and this would not be cost effective in terms of engineering resources.

The proposed approach provides product developers the flexibility to both add and remove functionality, while at the same time enabling UI customization on an unprecedented level.

Constraints

Using a redesigned WebTools presentation tier, the user should be able to use any existing component of WebTools setup or install as it functions today. The project manager should be able to modify the default client interface in such as way as to customize it for particular clients without compromising the underlying functionality. These mechanisms must satisfy the following twelve constraints:

- The implementation must support all fiery server products with a common codebase.
- The implementation must support rapid prototyping, development, release, and maintenance.
- The implementation minimizes footprint by using off-the-shelf components available in the JDK wherever possible.
- The implementation extends its functionality by using open standard communications protocols wherever possible.
- The server installation total footprint remains less than 1.5 MB.
- The client interface response time must be better than the current WebTools implementation.
- Eliminates the problem with client strings which is pervasive across product lines.
- Support ‘undo’ when a user selects ‘cancel’, requiring local/remote persistence of the user interface values.
- Easily customizable by project groups, to add/remove functionality and interface components.
- Support data type validation.
- Support behavioral (i.e., business rules) validation.
- Deploys as a web application (on any OS) and as a standalone application on Windows architectures.

Possible Solution Strategies

There are three viable approaches that could be leveraged to resolve the issues that meet some or all of the above project constraints: (1) a web-based client-server model, (2) an

XML-based model with a soap-based server interaction, or (3) a hybrid XML-based model with the current (direct) server interactions.

The most general approach would be a client server model that would support applets, applications, and multimodal web-based interactions. Unfortunately, this approach would be best suited to the use of N-Tiered architectures, the combination of which could never be developed within the hardware/software limitations of the product. The associated application server alone would require more resources than the entire embedded footprint allows.

The next best approach would be to use a general approach for representing the user interface requirements, a small-footprint method to create the display, and to revamp the communications mechanism to be a single, uniform approach, such as SOAP that could leverage the existing web server. This approach is feasible, but works best if the number of ways that server-based data is represented is minimized and the backward compatibility requirement is enforced at the server and not at the client. This would mean that the client would communicate to the server through the server, and the server would, at worst, delegate to other data models if necessary, but would preferably create a unified data model. This approach would also require a SOAP server running on the server side, which would add some runtime overhead to the server, though the server is already running an web server and the SOAP server would piggyback on the web server's resources.

The third approach would be a hybrid of the second, XML-based, approach and the currently-supported communications protocols integrated into a rewrite of the client's server code. Although this approach satisfies many of the stipulated constraints, I does nothing to migrate the communications layer to one in which the client is ignorant of how the server provides data, which has been an ongoing problem between the client and server development groups.

Proposed Solution Strategy

The solution strategy that provides the front-end flexibility required for the client and meets the other project constraints data models is the second, or SOAP, approach mentioned above. In this approach, the client is redeveloped to work as an XML-based approach that functions as follows:

- The server delivers the baseline configuration, page, object, and constraint models, written in XML, and adhering to a schema model of supported components and parameters, to the client host. These files completely describe the application being constructed.
- The client applet/application parses the XML files both into java-based swing components and their contents and constructs the page using an application toolkit. As part of this process, the application-specific representations are mapped into application-generic components by the toolkit.
- The page is rendered on the client host's interface browser by the toolkit.
- The user performs normal client-type actions (e.g., button presses, selections, text input, tab selections, etc.). These are handled by application-specific components that interact with the application-specific data models. Any rendering is handled by the toolkit.

- The interface enforces UI-specific data constraints, and catches and handles any violations between UI-specific data constraints.
- The validated page contents are submitted to the server using the SOAP protocols.
- The SOAP server translates the SOAP request into C++ objects and Harmony ATTR API calls.
- The Harmony results are aggregated into the C++ objects, packaged up into a SOAP response envelope, and returned to the client.
- The client evaluates and renders the response.

Project Deliverables

Engineering will produce a ToolShed implementation which meets the spirit of the requirements as outlined in the table in section 11.1 of the Fiery Product Specification, dated 4/16/2003. The look and feel have been finalized at the time of this writing, but the usage of UI components has been restricted to Swing components and the ToolShed implementation will support the currently-provided designs for the EFI WebTools2 UI Spec, dated 9/24/2003.

ToolShed Project Footprint

ToolShed must fit inside a 1.2 MB footprint. At present, the following space requirements are necessary for the functional aspects of the project:

Client Component	**Library Size**
XML Parser (crimson.jar)	201 KB
JDOM (jdom.jar)	125 KB
Communications (soap.jar)	237 KB
ToolShed	281 KB[1]
Rule Engine (Mandarax)	400 KB
Server Component	**Library Size**
gSOAP server	150 KB
Server side C++ classes/files	100 KB
Total without rule engine	994 KB
Total with Mandarax	1494 KB

Table 1: Approximate ToolShed required library footprint.

The Java runtime environment and the web server have been left out of this table because they are part of the current WebTools or server footprints and, by themselves, exceed the entire project space allocation. It is anticipated that some of the space requirement identified in the table can be reduced by removing some of the unnecessary SOAP components. For example, the Velocity Exchange project was able to reduce its SOAP component size from several MB to several KB by removing classes that aren't required. It is also anticipated that the use of another (or home-grown) rule engine will further reduce the space requirements. As an illustration, the Mandarax project supports database access, and those libraries, though fully integrated into the model, account for

[1] This is the size of required classes in the current prototype and will increase as full functionality is implemented.

approximately 120 KB alone. Thus approximately 150 KB must be saved assuming that the Java runtime is a cost not attributed to the ToolShed footprint, but it is assumed that this can be made up through judicious conservation in the library files.

In addition to the requirements specified above, additional components are required for generating communications calls on the server side. It is currently unknown what the size of these classes will be, but they cannot be shared with the Java client because they are implemented in C++.

There is currently no way to identify CPU/processing requirements as they impact the server when it is co-hosted with the client. The processing requirements are not expected to exceed those of the current client.

Outline

The remainder of this document presents the architectural requirements of the proposed ToolShed project engineering solution. These include the tools that must be supported, the mechanisms needed to support the implementation of those tools, the design of those mechanisms (class diagrams, sequence diagrams, examples, algorithms) used to implement the project, and a review of the use cases that must be supported by the product.

ToolShed Platform Architecture

ToolShed is the project name for the toolkit used to implement WebTools2. As such, it is supports a redesign/reimplementation of an existing EFI WebTools application, but can also be used to implement applications other than WebTools. While the client is being redesigned, the baseline look and feel of the client is also being redesigned, along with the server communications mechanism. The client no longer includes WebSpooler, but now includes new and repackaged features, as described below.

The ToolShed implementation will be phased. In phase 1, which has been completed at the time of this writing, a demonstration prototype will be presented that illustrates the approach applied to the WebScan application. In phase 2 the Installer, WebSetup, and remaining functional features will be prototyped. Phase 2 is also complete. In phase 3 the same functionality will be extended and deployed. In phase 4 and beyond the remaining of features will be added.

Tools

The major items that are accessible from WebTools2 implemented using ToolShed are: installer, print, scan, jobs, job log, setup, manage, vb manage, paper catalog, and estimate. The installer is a cgi script. VB manage, paper catalog, and estimate each point to separate applications. In the first implementation, only setup will be supported directly, leaving install, print, jobs, job log, and manage for future design and implementation/migration to ToolShed, if desired. Since the first deployments are expected to implement all of the functional features that would be used by these remaining applications, it is not anticipated that their implementations will require significant design/development time.

As a toolkit, ToolShed is comprised of the following three tools/components:

1) **Product Creation Management Tool**: This tool/control allows product developers to create, edit, configure, and manage the GUI for their product while remaining consistent with the ToolShed (and application) functional capabilities and constraints. It produces three files: a configuration file, a UI specification file, and a rules file which, together, define an application's interface.

2) **Localization Server/Builder Tool/Component**: This tool/control allows the developer to find out whether required strings exist in the localization repository, and to acquire localized strings for an application based on the strings found in the UI Specification XML file at build time. The strings are aggregated into a resource and made available at run time based on the selected locale.

3) **Toolkit Essentials Tool**: This component supports the parsing, rendering, event handling, and constraint validation required to implement an application. It is the core of the toolkit.

Using these tools and components, any application can be constructed, in particular WebTools. Moreover, applications can be aggregated easily, or made into standalone component functionalities, with the same toolkit.

Mechanisms

Nine interacting mechanisms are required to develop ToolShed (and hence to develop applications using ToolShed):

1) **UI Representation Mechanism:** This mechanism enables a generic description of the user interface that will support the design requirements of the project. The proposed implementation uses a set of XML description files (and associated schemas) to describe the interface. The schemas enforce compliance with the objects and object attributes that are supported by the toolkit.

4) **Configuration Mechanism**: This mechanism enables product developers to select functionality, to show/hide elements, to set branding, to set look and feel capabilities, and the like, for the entire application or at the component level. This mechanism is an artifact of the representation format selected for the UI/data files, but is listed separately because it relates directly to project use cases and is enforced externally with a visual XML editor. This mechanism is mostly associated with the Design Tool.

5) **Rendering Mechanism**: This mechanism determines how XML is parsed into and rendered into generic Swing-based user interface components. Discussion of this mechanism will be paired with the UI representation mechanism.

6) **Event Handling Mechanism**: This mechanism represents and processes events based on actions from users of the UI. This mechanism will be discussed in parallel with the UI representation and parsing mechanism.

7) **Exception Handling Mechanism**: This mechanism handles exceptions in a consistent and coherent manner. The exception handling mechanism is a combination of the built-in exception handling mechanism in Java, as well as how the exceptions are relayed to the user. This mechanism will be discussed in parallel with the UI representation and parsing mechanism.

8) **Validation Mechanism**: This mechanism validates the UI model and data values against their data types/ranges.

9) **Constraint Representation and Validation Mechanism**: This mechanism represents the data constraints across multiple objects, and requires action when the user makes selections.

10) **Communications Mechanism**: This mechanism dictates how information is conveyed to and from the WebTools client and the Fiery server.

11) **Localization/Internationalization Mechanisms**: These mechanisms ensure that localized strings and internationalized structure are supported in the new interface.

Approach and Background

The general ToolShed/WebTools2 architecture will integrate with other EFI components as shown in the Figure 1:

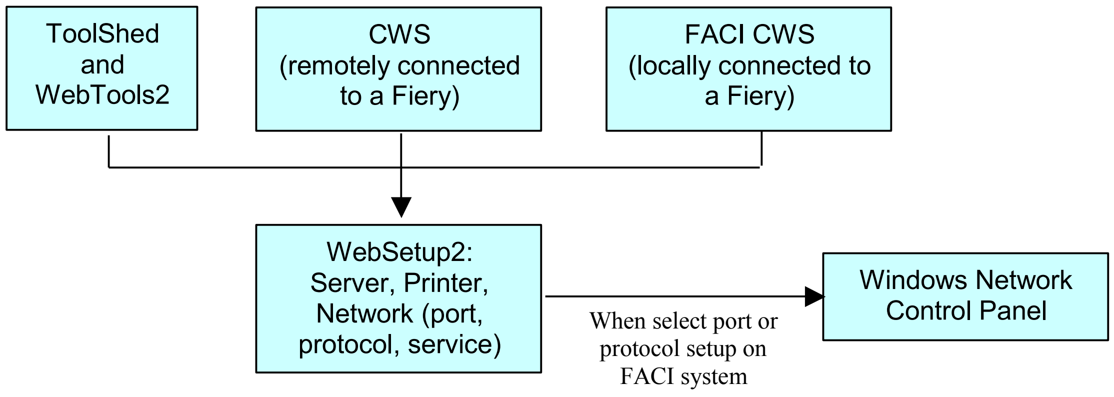

ToolShed and WebTools2
CWS (remotely connected to a Fiery)
FACI CWS (locally connected to a Fiery)
WebSetup2: Server, Printer, Network (port, protocol, service)
Windows Network Control Panel
When select port or protocol setup on FACI system

Figure 1: ToolShed integration with CWS/Fiery.

Simply put, what was once split into two components, where the FACI CWS interacted directly with the Windows Network Control Panel, now has a single point of interaction using the WebTools2 application software. All versions of CWS and ToolShed interact directly with this component. From the WebTools2 client perspective, the interaction appears as shown in Figure 2:

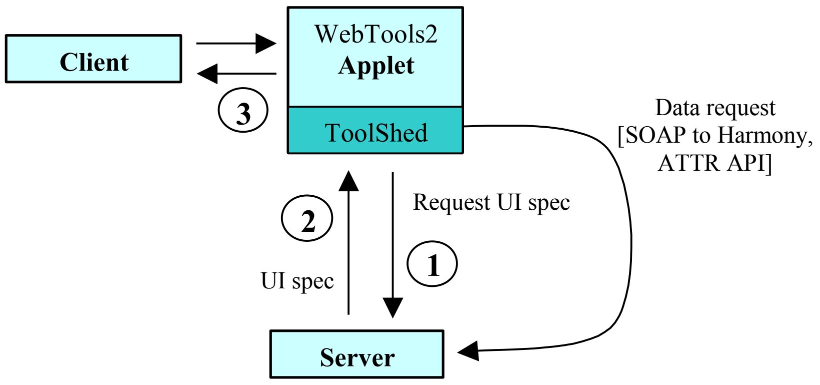


Figure 2: WebTools/ToolShed high-level component interaction.

The WebTools2 applet portion (at launch time) requests a UI specification from the Fiery server (at **1**), which is returned to WebTools2 by the server (**2**). The ToolShed toolkit then parses the application-specific XML into renderable components and forwards the markup to the client browser (**3**). User events are converted by ToolShed to appropriate application-specific data requests and submitted to the server. All data requests are mediated through SOAP to the server, which translates the requests to Harmony ATTR calls. The ToolShed structure is itself shown in

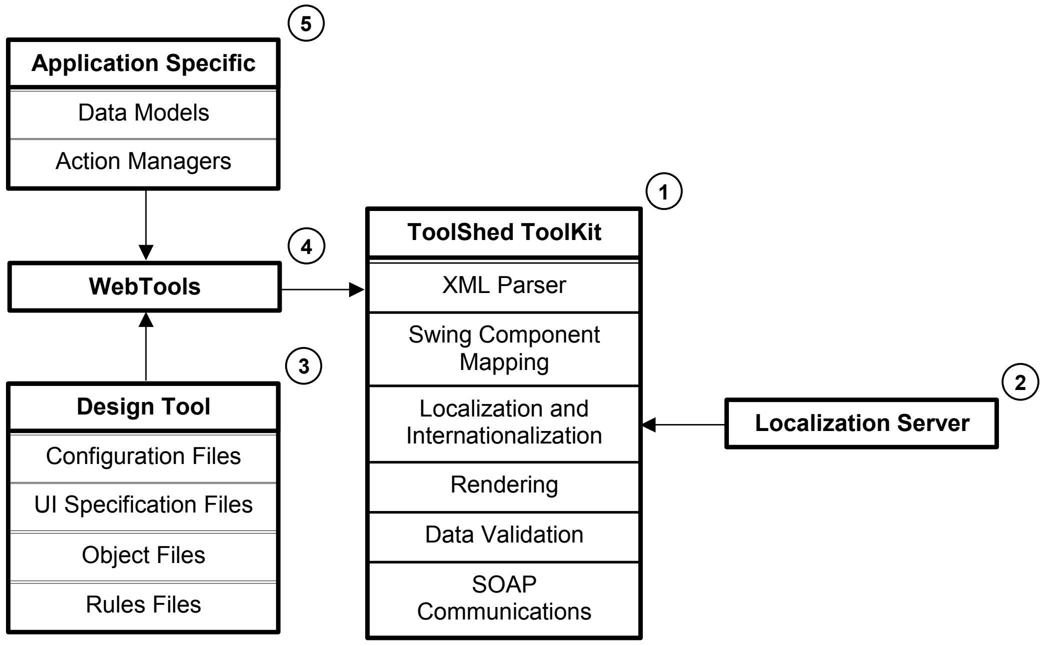


Figure 3: ToolShed component architecture.

The ToolShed toolkit (at **1**) takes as primary input XML files produced by a design tool (at **3**), and some application specific classes that define the behavior of objects and the behavior of ui components (at **5**). The files produced by the design tool describe the runtime configuration of the application, the UIs, the objects used in the UI and the constraints (as rules) between objects in the UI. These are clearly decoupled. The data models and action managers map the items defined in these files to the toolkit internals. During page launch, the XML files are parsed by the toolkit into Java Swing components which are then rendered. When actions are performed, the action manager interacts with the data model and through the communications layer (where required). Since the toolkit internal structure is defined using standardized components, and the design tool produces XML, and the SOAP communications layer is standard and open, the only application-specific files are those that define the data model and action handling.

The proposed architecture makes use of an XML layer on the presentation interface between the applet and the client that provides a uniform and implementation-independent representation of the interface and the data it will display. The approach supports the following six capabilities:

- Widget implementations are kept distinct/separate from widget requirements descriptions in the user interface. That is, they are decoupled.
- Functionality at varying levels of granularity can be added/removed at launch time.
- The user interface can be customized by the product development group as long as it adheres to the XML Schema provided by the ToolShed development group.
- Coarse validation can be performed on either/both the server and client (if the server groups implement an XML data hierarchy and adopt the ToolShed constraint validation mechanism).

- Granular validation can be represented in the interface and enforced by the client at runtime.
- Event handling is represented at the client level and either handled locally or passed to the server through normal communications channels.

Demonstration prototypes have been constructed using the XMLTalk methodology, which is presented herein for background. XMLTalk is a presentation layer that removes much of the chore/overhead of developing and maintaining user interfaces by development engineers, but also supports customization by product development teams. The XMLTalk architecture is presented in Figure 4:

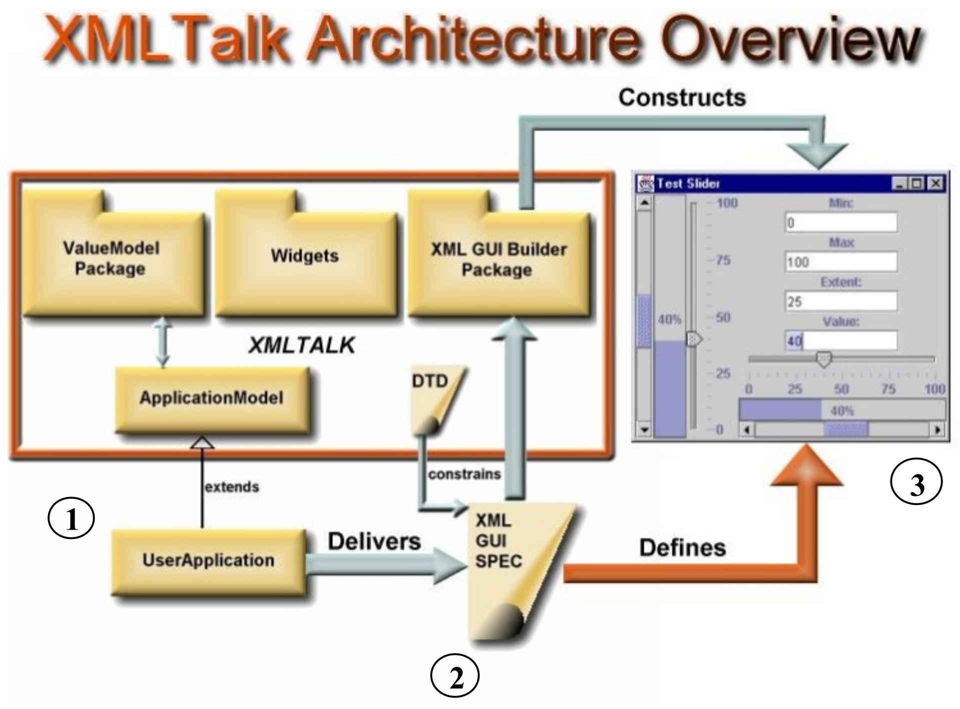


Figure 4: XMLTalk architecture.

The UserApplication (at **1** in Figure 4:) represents the WebTools2/ToolShed applet/application, which implements the XMLTalk functionality as well as the current WebSTools functionality. A set of components that implements the same functionality as XMLTalk is constructed within the application, along with a parser that accepts an XML page description and translates to this model. The page template is constructed in XML (**2**) using the formalisms in this model, which is then parsed at runtime inside the applet/application and rendered into the user interface (**3**).

The cornerstone of this approach is the ValueModel java bean, which is needed to represent an attribute value in XML. This (abstract) bean defines accessors for an object instance (i.e. setValue/getValue) and fires a PropertyChangeEvent when the value changes, which can lead to rerendering of the associated component. Inherent in this approach is a relationship between the XML value model and an associated Java domain model, as shown in Figure 5:

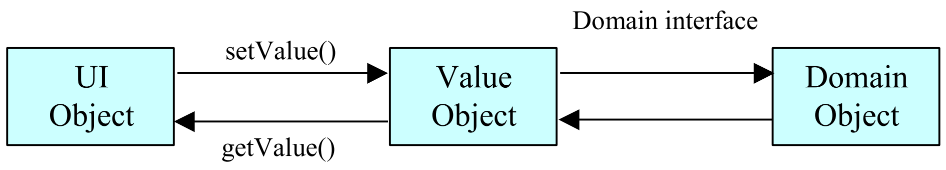


Figure 5: Basic ValueModel integration approach.

In the standard manner of presentation layers, the UI object has embedded calls for dynamic data that invoke set/get methods on, in this case, the value object. In turn, the value object has a one-to-one correspondence with the domain model. In the form shown above, the ValueModel is of little practical value, since there is no value association to an attribute in the domain model. The object hierarchy used to implement real/viable interfaces is shown in Figure 6:

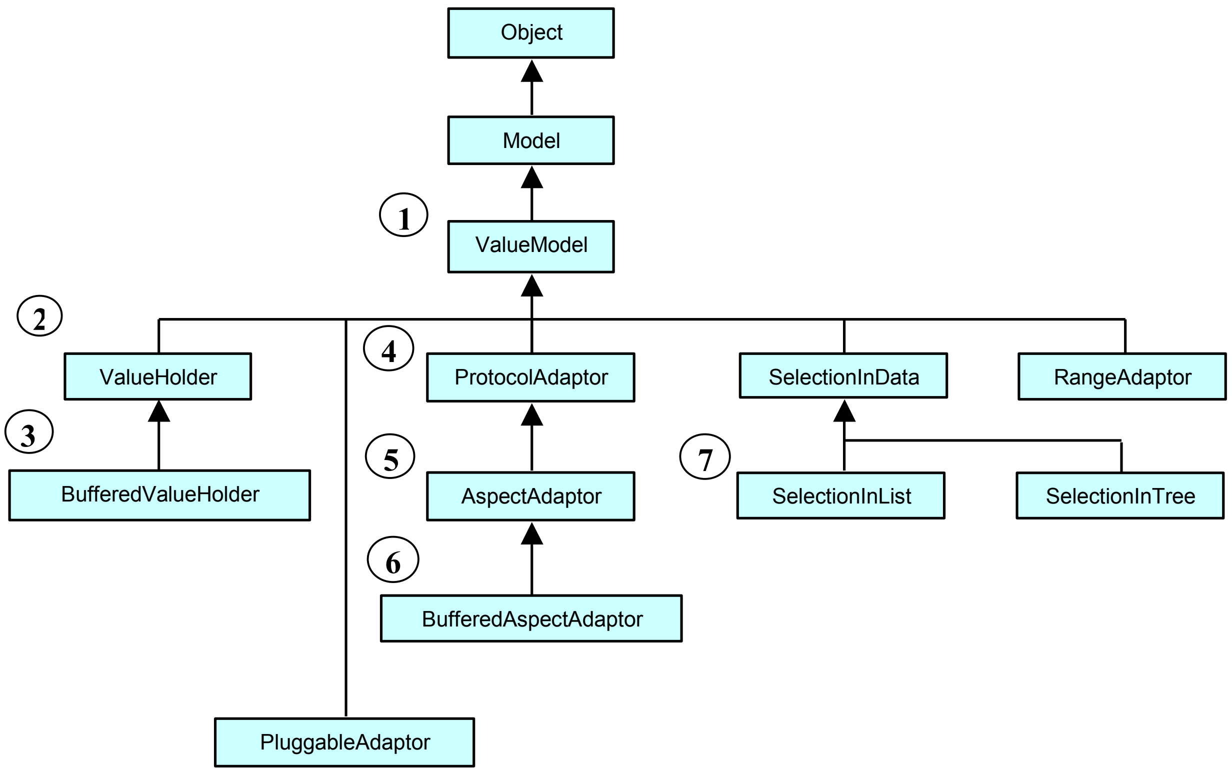


Figure 6: ValueModel class hierarchy.

ValueModel class (at **1**) is an abstract class that provides the foundation for simple single-attribute bean support. It also inherits from property change listener. The ValueHolder class (at **2**) is a class that extends ValueModel. Implementation of a read/write object requires the BufferedValueHolder (**3**), which represents another ValueHolder that maintains the new attribute name/value until a swap is required, say during a property change event.

A standard bean interface can be implemented using the above approach if the getValue/setValue pair can be adapted to a particular naming convention associated with a particular attribute. This is achieved with the ProtocolAdaptor class (at **4**), which adapts the ValueModel to different interface protocols, and the AspectAdaptor (**5**)/BufferedAspectAdaptor (**6**) classes, which are used for creating bean correlations. This group is illustrated in Figure 7:

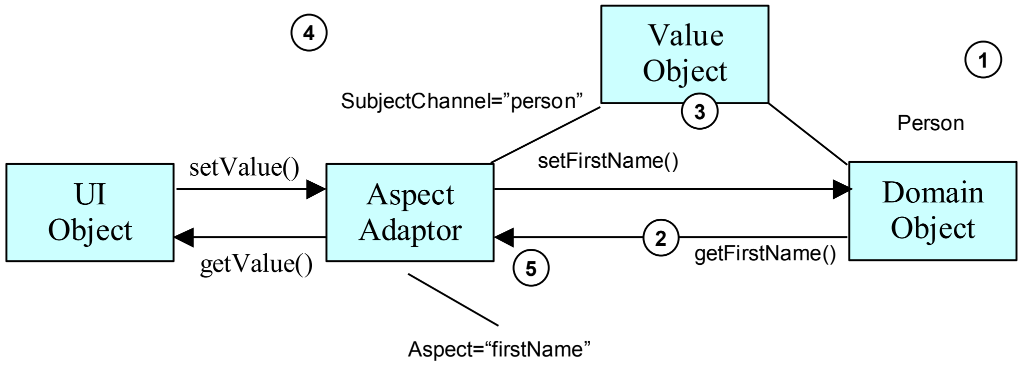


Figure 7: AspectAdaptor model and usage with shared channel. In this example a second Aspect might be "lastName", and the shared channel might be "Person".

The intent of the AspectAdaptor is to support the representation of an arbitrary java bean using XML. For example, if a Person Class is being implemented (at **1**), it might have attributes for firstName and lastName. The Person Class would also have accessors getFirstName/getLastName (at **2**), and mutators setFirstName/setLastName (at **3**). The name of the class, Person, represents what is called the shared subject channel (at **4**), which provides the glue that associates the aspects firstName (at **5**) and lastName together with Person.

The combination enables a one-to-one mapping between XML components and a domain object. The aspect adapter enables the conversion from the generic accessor/mutator to the bean-specific (attribute-based) versions. Hence, getValue, with a SubjectChannel of "person" and an AspectAdaptor of "firstName" produces an accessor/mutator pair of getFirstName/setFirstName.

ToolShed Application Architecture

ToolShed is the presentation layer of WebTools2. This means that it is responsible for the representation and rendering of pages and page content, for verifying and maintaining data relationships concurrent with server requirements, and for supporting the flexibility the project requires.

The reference implementation for ToolShed is based on a combination of technologies and techniques, and demonstrated on the WebTools WebScan application. The architecture is the same for WebSetup and other WebTools2 presentation components. The interaction with the WebTools2 client, per se, isn't depicted in this diagram at present. Communications between client and server are mediated by the WebTools2 client.

The following sections will present the overall package model, followed by a simplified workflow of how the project functions. These will be followed by discussions of the processing tiers.

Package and Component Interactions

The project is divided into three major functional areas, as depicted by the pseudo package/component diagram shown in Figure 8:

- Parsing (includes customization, localization, page construction, exceptions, and event handling)
- Communications
- Logic

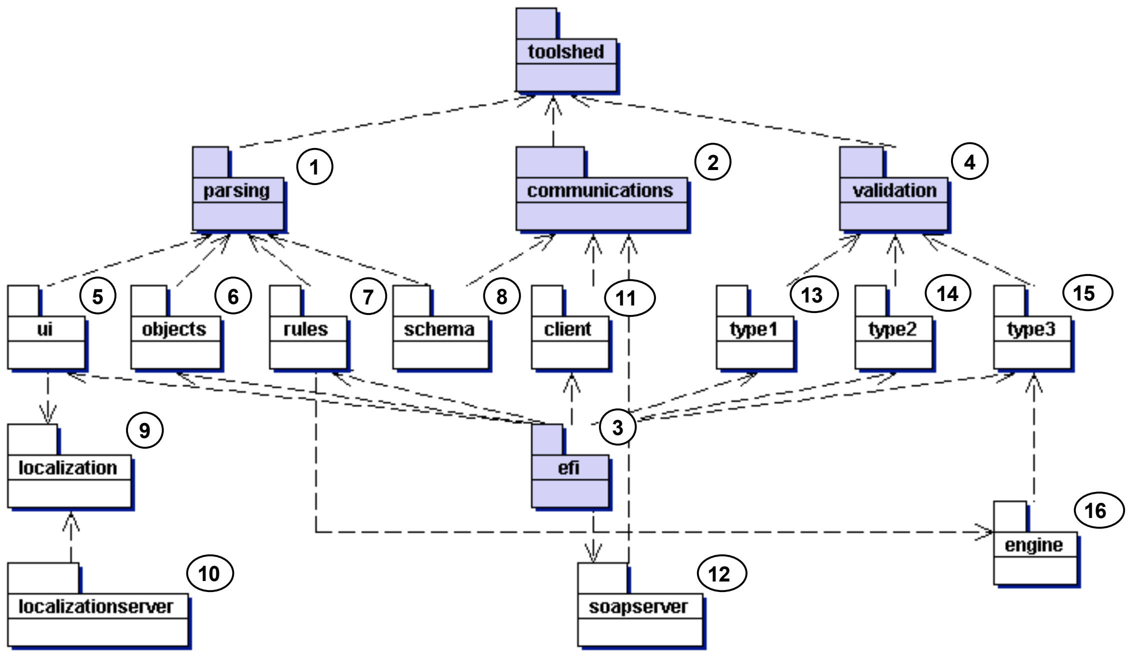


Figure 8: ToolShed component diagram.

The primary mechanisms in this package diagram are highlighted in blue. Parsing (at **1**) is the primary ToolShed component, as the model is based on XML-based input setup and configuration files. Communications (at **2**) takes two forms: acquisition of the configuration files, and data transactions. Finally, the underlying logic required to manage the tool but cannot efficiently be captured in XML is represented in the Logic package (efi, at **3**). Much of this logic is directly associated with component rendering, some of which can be handled through data validation (at **4**).

Returning to the parsing functionality, ToolShed parses XML data files of three types: setup/configuration, user interfaces (at **5**), and object data (at **6**). Object data embeds both type requirements and inter-object rules (at **7**). All parsing, and communications, must maintain coherence with and synchrony with a single data model. The data model is represented with XML Schemas for each of the ui and object types used by ToolShed (at **8**). The parsing process identifies the content type in the XML files, but requires a construction/generation phase to map the files to Java components.

The construction of the user interface, and the objects the interface will present, requires mapping from XML elements to Java components. This requires non-interface classes that are part of the logic package (at **3**). At construction time, labels and other ui component-related strings are localized (at **9**) using string resources acquired at build time from a Localization Server (at **10**). Part of the construction phase is a mapping from efi ui components to JFC (aka Swing) components, which assign event listeners and event handlers to the components. Once the construction phase is complete the interface can be rendered and respond to events.

As mentioned, a primary role of the communications package is to interact with the Fiery Server at run time, to acquire data or modify setup values. The ToolShed communications package is divided into client (at **11**) and server (at **12**) components, both of which depend on the same XML Schema (at **8**) to map data types across the network interface. In this manner, the client and server logic can be implemented on different architectures and in different languages. The client package is part of the logic component (at **3**), while the server package is its own component outside of ToolShed proper.

Data validation (at **4**) is a significant aspect of the ToolShed architecture, because it is defined in the XML files and follows an XML Schema. This enables the ui-specific logic associated with objects (types and interactions) to be represented outside of the internal program logic, and thus to be shared across systems. ToolShed employs three validation types: type 1 (at **13**), which is used to validate object existence; type 2 (at **14**), which is used to validate object value type, range, and defaults; and type 3 (at **15**), which enforces inter-object constraints. These validation types are part of the ToolShed exception handling mechanism.

In many page models there are data constraints that exist across multiple fields or views. These cannot be handled by a simple data validation mechanism. The ToolShed approach to resolving these constraints is to use a rule-based engine (at **16**). When a user makes a modification to the value of a component, the engine cycles through the rules associated with the related functionality, identifies violations based on values elsewhere in the system, and interacts with the user to ameliorate the conflicts. The rules (at **8**) are parsed from RuleML into a Java-based knowledge base at read time.

Workflow Mechanisms

Figure 8: presents the packages that will be used to implement the ToolShed project, but not their flow. Specific mechanisms will be shown later, in greater detail, but the overall workflow is depicted in Figure 9:

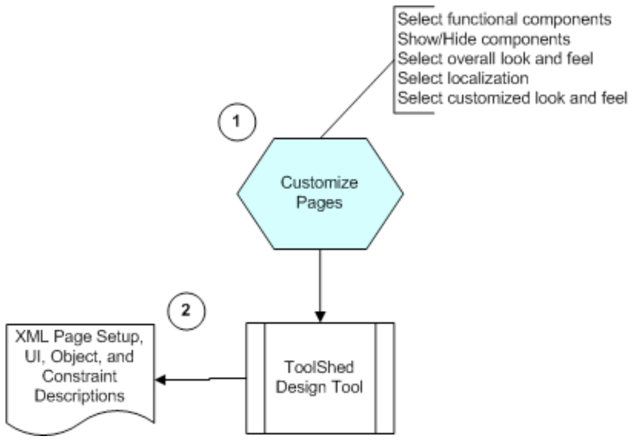

Select functional components
Show/Hide components
Select overall look and feel
Select localization
Select customized look and feel
1
Customize Pages
2
XML Page Setup, UI, Object, and Constraint Descriptions
ToolShed Design Tool

Figure 9: ToolShed page construction workflow diagram.

The configuration phase (depicted by the blue trapezoid at **1**) involves the page designer and product manager, who determine which of the functional components will be in their product, whether they will be displayed, what the look and feel will appear like, and what the language will be. This is accomplished both by editing the setup document (at **2**) and by editing the page descriptions themselves with a visual XML IDE such as the ToolShed Design Tool (at **3**). This workflow component is depicted separately from the application because it is being implemented as an independent application and used off-line by product developers rather than by the Client Applications engineering team. This is the component referred to by the first tool item at the beginning of this document.

The primary ToolShed workflow is depicted in Figure 10:

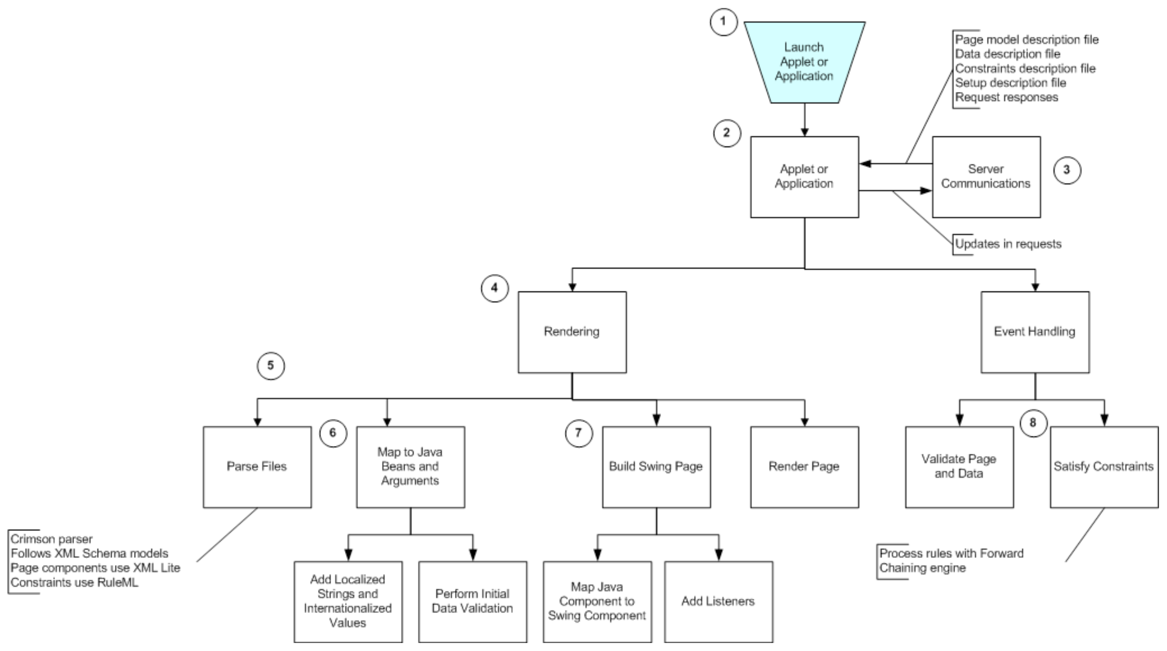

1
Launch Applet or Application
Page model description file
Data description file
Constraints description file
Setup description file
Request responses
2
Applet or Application
Server Communications
3
Updates in requests
4
Rendering
Event Handling
5
Parse Files
6
Map to Java Beans and Arguments
7
Build Swing Page
Render Page
Validate Page and Data
8
Satisfy Constraints
Crimson parser
Follows XML Schema models
Page components use XML Lite
Constraints use RuleML
Add Localized Strings and Internationalized Values
Perform Initial Data Validation
Map Java Component to Swing Component
Add Listeners
Process rules with Forward Chaining engine

Figure 10: ToolShed primary operations workflow diagram.

The application phase (depicted by the blue trapezoid at **1**) is divided into launch and interaction modes, both of which utilize the communications and rendering packages. In launch mode, the applet or application (at **2**) brokers communications with the server (at **3**) to obtain the setup, page, data, and constraint data files. During the interaction mode, requests are sent to the server, and responses received from the server, with the notion of updating the existing page models.

In both modes, rendering (shown at **4**) is accomplished as a four-step process: (1) parsing, (2) mapping XML to the Java bean model, (3) constructing the Swing page model, and (4) rendering the page. The parsing process (at **5**) is noteworthy because it mandates conformation with an XML Schema that represents the possible page, data, and constraint objects, their attributes, and their attribute types. Conformation with the schema guarantees the page designer that their pages will render, and it guarantees the applet/application developer that only conforming models will be dispatched to the applet/application. Also embedded in the model are the data-object constraints. These are represented in RuleML and parsed into Mandarax-type Java rules. Initial data/constraint validation is performed during object construction (at **6**), and is then performed whenever a data value is modified during event handling (at **8**). Once object construction is complete, the Java objects can be mapped to their Swing counterparts (at **7**), listeners added to the Swing components, and the page can be rendered.

Package and Class Identification

The individual packages will now be presented in greater detail, by type. The following class packages implement these object types and will be presented in the following order:

- XML Schema validation of UI XML files

4

2

3

- The xml packages (xml, model)
- The beans package associated with XML Lite
- The ui package
- Project specific (appls)

XML Schema for Model Validation

XML Schema is to XML that Java classes are to Java instance objects; it defines the organizational structure that instance files are validated against. By providing XML Schema for the UI and Object models, the notions of product developer flexibility are supported, because if they product validated XML files according to the schema provided by Client Applications, they files will render/function properly. A sample XML Schema that matches the kinds of applications that ToolShed supports is provided in Figures 8-15 below. The overall structure is illustrated in Figure 11:

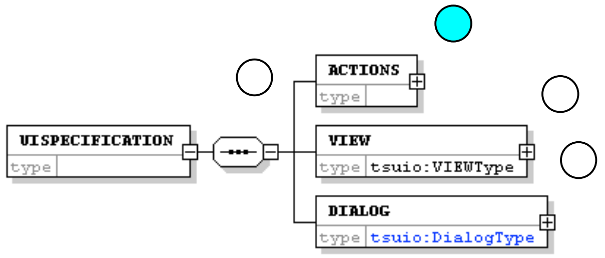


Figure 11: XML Schema for overall user interface application modeling.

The '-/+' notation in the figure (e.g., at **1**) has the standard meaning, that the item can be further expanded. The '…' (in oblong, at **4**) notation means that the items that follow it represent a required sequence of elements, and that the order must be maintained.

Notice that a general UI Specification is comprised of three components: (**1**) actions, which represents applicable behaviors on the interface components; (**2**) views, which represents the UI components and their layout, respectively; and (**3**) dialogs, which represent the exception-handling UI components and their layout, respectively.

The Actions component is further fleshed out in Figure 12:

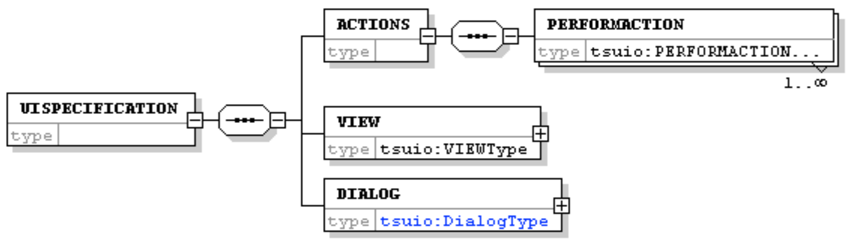


Figure 12: XML Schema expansion for the Actions component.

The Actions component simply identifies the action to be performed when referenced, and defines attribute values for action name, the action to perform, and the associated value.

The DialogType component is fleshed out further in Figure 13:

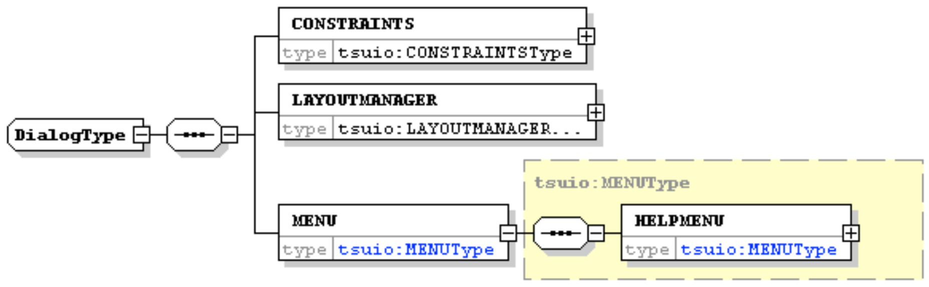


Figure 13: The DialogType XML Schema component.

The ViewType component is shown in Figure 14:

2

3

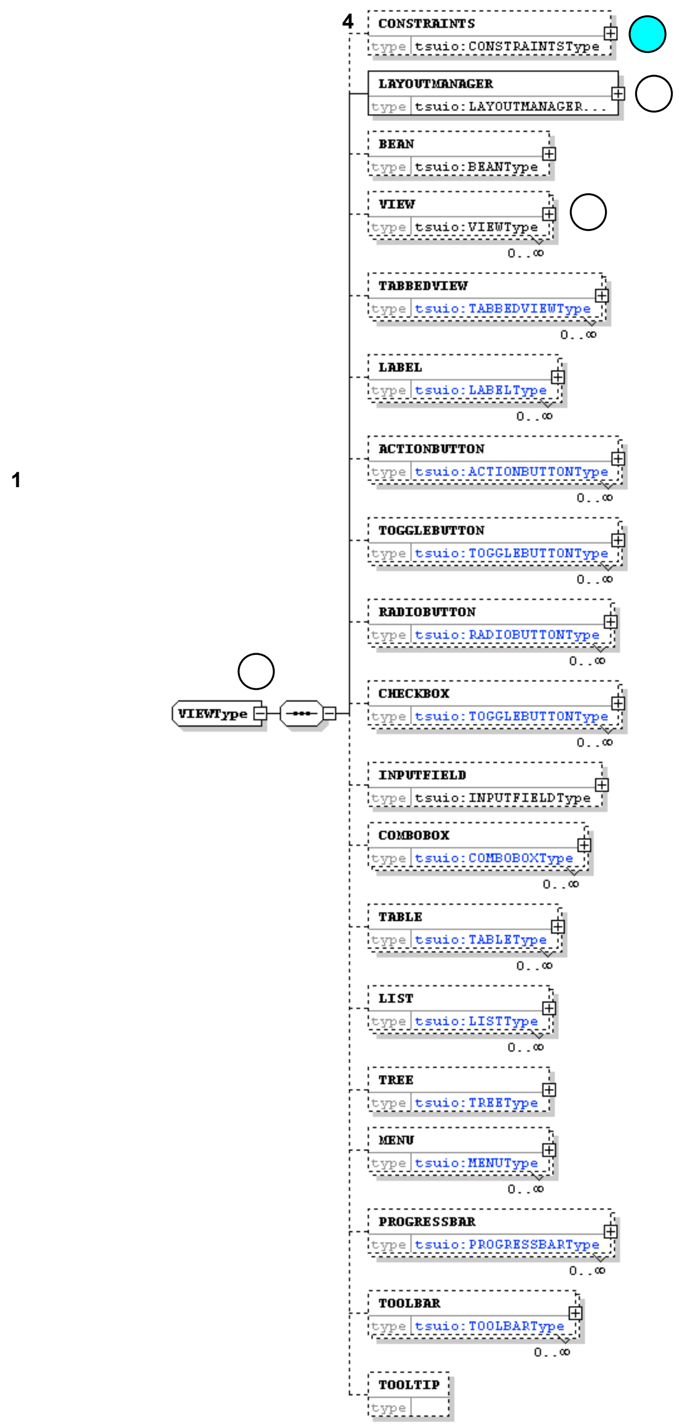


Figure 14: XML Schema expansion for the ViewType component.

The ViewType component (at **1**) is comprised of the elements that can be used to construct the user interface. Common to most components is the Constraints component, which represents the component location, size, and border attributes (at **2**). Common to all container components is the layout manager used to arrange components within the container (at **3**). The ViewType, as a container, has instances of both the Constraints and

LayoutManager components. It can also recursively contain other instances of the View component (at **4**). The remaining component types are the items that can be placed into a View component. The dotted lines leading to all of the components within ViewType (except LayoutManager) mean that these components are optional. The “0..oo” notation under many of the components means that any number of this type of component can be included in the view. Notice that the view can have only one Constraints and LayoutManager instance, as well as a single Bean component instance.

The nineteen components defined in the ToolShed XML Schema represent the visible components that will be supported by ToolShed when it is fully implemented. Many of these components have no utilization in current product designs but are anticipated for future designs and, as such, are included as part of the schema.

The ConstraintsType object is shown in Figure 15:

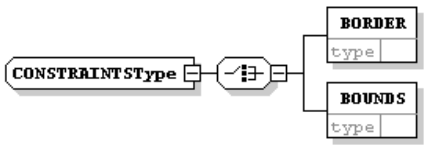


Figure 15: The object ConstraintsType component.

Most visual components have border and bounds constraints, so these are represented at the type level. The Border component has three attributes: side, width, and height, which describe where the text will be located and the size of the border itself. The Bounds component describes placement and size of the component, and has x, y, width, and height attributes.

The LayoutManagerType component is fleshed out in Figure 16:

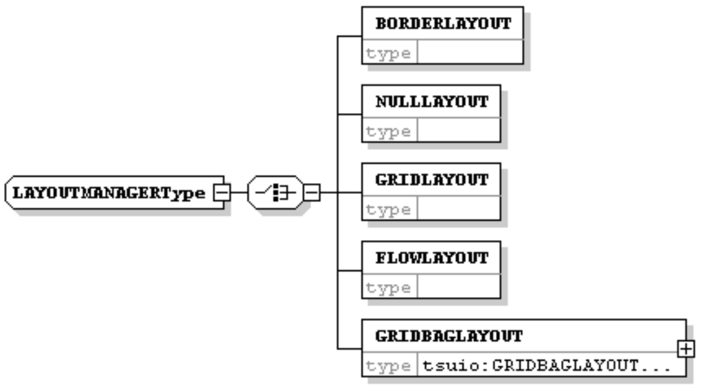


Figure 16: LayoutManager XML Schema component.

The LayoutManager component represents a selection between five container layout types: (1) border layout, (2) null layout, (3) grid layout, (4) flow layout, and (5) gridbag layout. These have exact counterparts to Java awt layout components by the same names. All but the Null Layout component have two attributes: hgap and vgap, which define the horizontal and vertical gaps between components within the layout.

In a border layout there are five locations in which components can be placed, each associated with a compass rosette: north, south, east, west, and center.

In a flow layout components are laid out in the order that they are added to the container, but there is an additional attribute: alignment, which defines the alignment order of components (left to right, right to left).

The grid layout describes components as an array, so it has two attributes: rows, cols that define the number of items that it contains. Each grid cell takes on the size of the largest item contained.

The gridbag layout is the most complex layout mechanism, allowing for grid layout where the cells can have differing sizes. As a result there must be a constraints object that tells how the cells will relate to one another, with ten attributes: gridx, gridy, gridwidth, gridheight, weightx, weighty, anchor, fill, padx, and pady. See Java documentation for explanations on how to use gridbag layout attributes to create complex layouts.

The Bean component is shown in Figure 17:

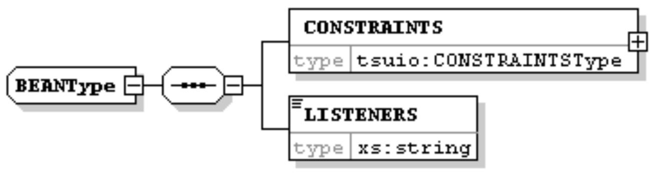


Figure 17: Bean XML Schema component.

Beans are objects that either extend the functionality of known Swing components or are data/processing specific within the com.efi subhierarchy. As such, they listen to events and thus the Listeners component in addition to the Constraints component.

A table showing the attributes of the remaining components is shown as Table 2:

View Component	
Attribute Name	**Attribute Description**
type	Not sure what this is (string)
background	Background color (hex color string)
width	Width of view (int)
height	Height of view (int)
TabbedView Component	
Attribute Name	**Attribute Description**
title	String to associate with the overall component
tabplacement	Placement on the component (int) for left, right, top, bottom
layoutpolicy	An integer representing what?
model	The model used to populate the tabs (string)
selectedtab	The default tab for the component (int)
Label Component	
Attribute Name	**Attribute Description**
horizontalalignment	Alignment fo the label (int) for left or right
icon	Optional icon to associate with the label (string path)
textposition	? (string)
icontextgap	Gap between the icon and the label text (int)
ActionButton Component	
Attribute Name	**Attribute Description**
text	The text associated with the action
action	The action to take when selected (string)
icon	The optional icon to associate with the button (string path)
ToggleButton Component	
Attribute Name	**Attribute Description**
text	The text to associate with the toggle button (string)
action	The action to take when the button is selected (string)
icon	The optional icon for the component (string path)
selected	The selected item (int)
RadioButton Component	
Attribute Name	**Attribute Description**
text	The text to show with the radio button (string)
model	The model used to fill the radio button
horizontalalignment	The horizontal alignment of the items (string)
onvalue	? (token)
icon	An optional icon for the radio button (string path to image)
selected	Which item is selected by default (int)
Checkbox Component	
Attribute Name	**Attribute Description**
action	The action to take when an item is selected (string)
icon	An optional icon to associate with the checkbox (string path)
text	The text to associate with the checkbox (string)
selected	The default selected item (int)
InputField Component	
Attribute Name	**Attribute Description**
model	The model associated with the field
columns	The number of default columns visible in the field (int)
type	Not sure what this means today (string)

action	The action to take on selection (string)
ComboBox Component	
Attribute Name	**Attribute Description**
model	The model used to fill the box
text	A string label for the box
action	The action to perform when an item is selected (string)
numVisibleRows	The number of rows visible in the box (int)
selectedIndex	The number of the selected item (int)
Table Component	
Attribute Name	**Attribute Description**
model	The model used to fill the table
rowColSectionEnabled	A Boolean representing whether rows and columns can be selected
colSelectionEnabled	A Boolean representing whether columns can be selected
gridColor	The color of the grid (hex string representing color)
rowHeight	The height of table rows (int)
rowMargin	The margin between rows (int)
selectionFGColor	The foreground color of selected items (hex color string)
selectionBGColor	The background color of selected items (hex color string)
showGrid	A Boolean representing whether to show the grid boundaries
showHorizontalLines	A Boolean representing whether to show row boundaries
showVerticalLines	A boolean representing whether to show column boundaries
List Component	
Attribute Name	**Attribute Description**
model	The model used to fill the list with items
cellHeight	The height of each cell (int)
cellWidth	The width of each cell (int)
orientation	The orientation of the list (int)
selectedIndex	Which item in the list is selected (int)
selectedBGColor	The background color of the selected item (hex string for color)
selectedFGColor	The foreground color of the selected item (hex string for color)
visibleRows	The number of rows that are visible (int)
Tree Component	
Attribute Name	**Attribute Description**
model	The model used to fill the tree elements
editable	A Boolean describing whether the tree is editable
rowHeight	An integer describing the height of rows
scrollable	A Boolean describing whether the tree is scrollable
showRootHandles	A Boolean describing whether to show the handles
visibleRowCount	An integer describing how many rows to of the tree to display
Menu Component	
Attribute Name	**Attribute Description**
selectedComponent	An integer describing which menu item is selected by default
ProgressBar Component	
Attribute Name	**Attribute Description**
model	The data model used to make it work
orientation	An integer describing the progress bar orientation
min	The minimum value for the bar (int)
max	The maximum value for the bar (int)
progressString	An associated display string

value	The default value (int)
ToolBar Component	
Attribute Name	**Attribute Description**
orientation	An integer describing the orientation (horizontal/vertical) of the bar
rollover	A Boolean representing whether rollover is supported
ToolTip Component	
Attribute Name	**Attribute Description**
parent	Maybe not necessary, indicates associated object (by name)
text	The text in the tooltip

Table 2: Visual component attributes.

Most of the remaining components have only a Constraints component in addition to their element-level attributes. The ToolBar component is a container and so it also has a LayoutManager subcomponent, as shown in Figure 18:

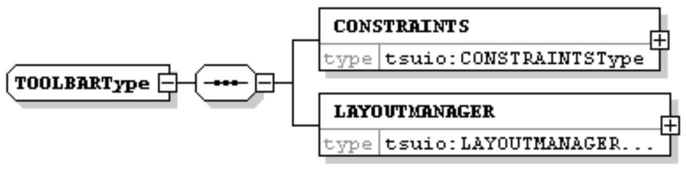


Figure 18: ToolBar XML Schema component.

XML Packages

XML is the mechanism whereby ToolShed functions. All data (layout, data, and constraints) is provided and modified through XML-formatted files. The UI-related XML files are subdivided into three general types: (1) models, which describe objects; (2) views, which describe interfaces and, possible, values of objects; and (3) actions, which describe operations on models and ui components. The overall package diagram for the XML package is shown in Figure 19:

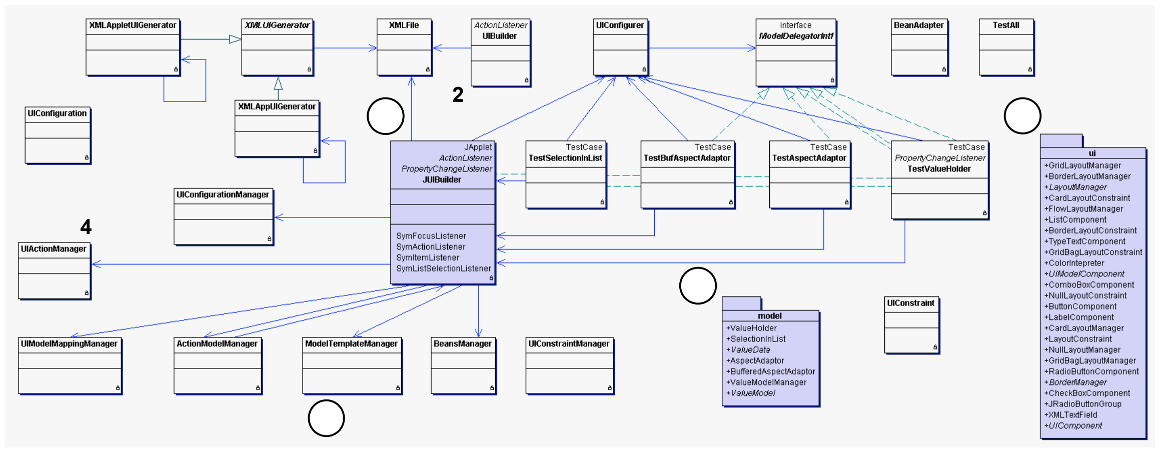


Figure 19: XML package diagram.

Much of the operational functionality for the XML package resides in the JUIBuilder class shown at **1** in the figure). The JUIBuilder class will be described inline with the other two packages, model (at **2**), and ui (at **3**) within the XML package. Models, views, and actions are managed with manager objects (UIConfigurationManager, UIActionManager, UIModelMappingManager, ActionModelManager, ModelTemplateManager, BeansManager, and UIConstraintManager, at **4**), which are collection objects. The overall initialization method, initializeUI, is shown below in Figure 20:

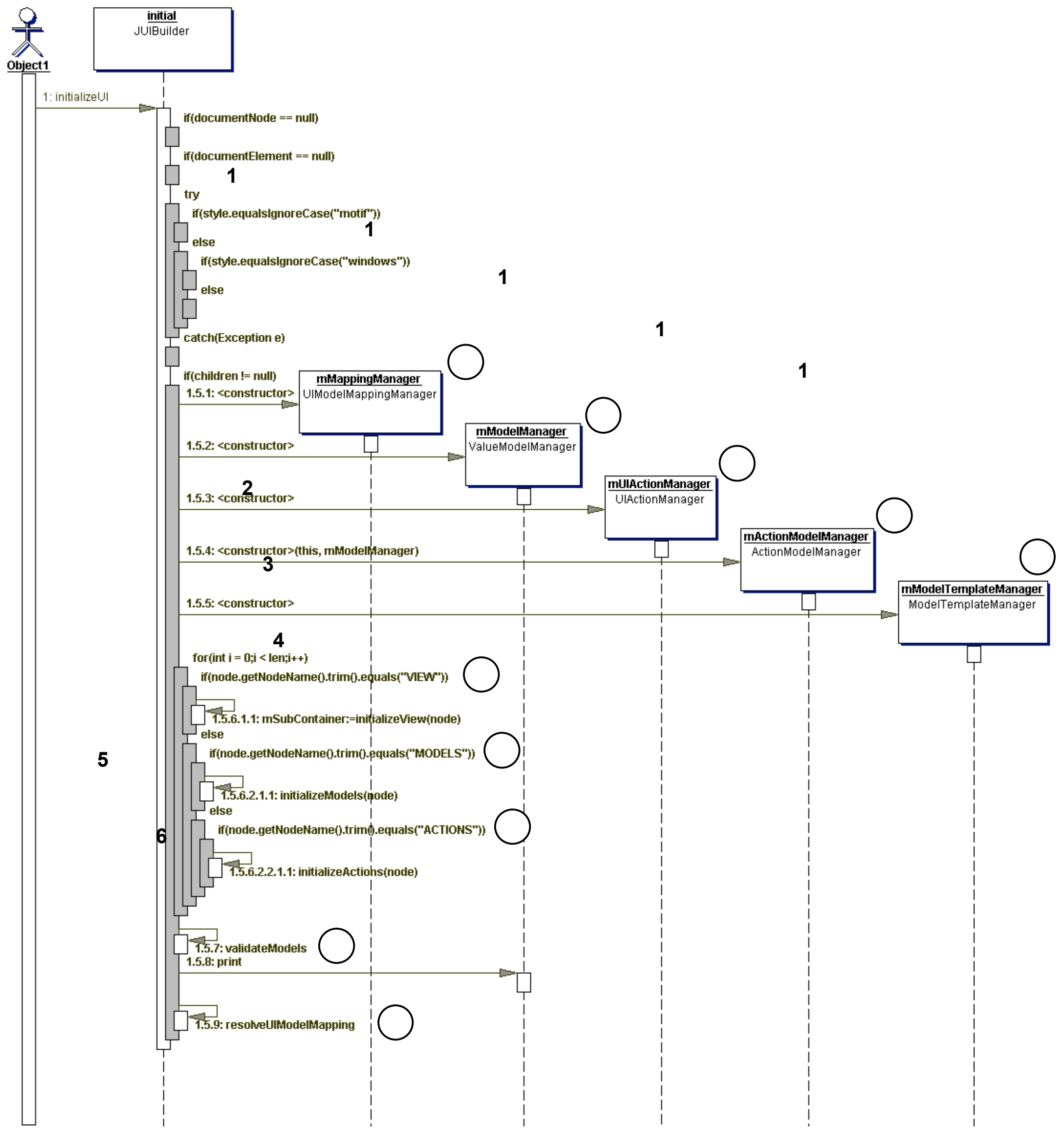

Object1
initial
JUIBuilder
1: initializeUI
if(documentNode == null)
if(documentElement == null)
1
try
if(style.equalsIgnoreCase("motif"))
1
else
if(style.equalsIgnoreCase("windows"))
1
else
1
catch(Exception e)
1
if(children != null)
mMappingManager
UIModelMappingManager
1.5.1: <constructor>
mModelManager
ValueModelManager
1.5.2: <constructor>
mUIActionManager
UIActionManager
1.5.3: <constructor>
2
mActionModelManager
ActionModelManager
1.5.4: <constructor>(this, mModelManager)
3
mModelTemplateManager
ModelTemplateManager
1.5.5: <constructor>
4
for(int i = 0;i < len;i++)
if(node.getNodeName().trim().equals("VIEW"))
1.5.6.1.1: mSubContainer:=initializeView(node)
else
5
if(node.getNodeName().trim().equals("MODELS"))
1.5.6.2.1.1: initializeModels(node)
else
6
if(node.getNodeName().trim().equals("ACTIONS"))
1.5.6.2.2.1.1: initializeActions(node)
1.5.7: validateModels
1.5.8: print
1.5.9: resolveUIModelMapping

Figure 20: JUIBuilder class initializeUI method.

The initializeUI method creates the collections (shown at **1**) used to manage the various object types. The current node is then checked to see if it is a view (at **2**), a model (at **3**), or an action (at **4**), in which case the element is mapped by way of an initialization method to the appropriate Java class instance and added into the appropriate manager collection. When the entire file has been parsed, the resulting models are validated (at **5**), and the UI objects are mapped to their Swing counterparts (at **6**) and rendered.

Models

Models represent the data objects that are used in the interface, and make use of the XML Lite architecture, which is implemented with the XML Model package. Both static and dynamic models are implemented with this package.

XML Model Package

3 1 2 6

Model creation is performed in ToolShed using a subset of the XML Talk architecture, (XML Lite), as Java classes. Specifically, the clases implemented are:

- ValueModel
- ValueHolder
- ValueData
- AspectAdaptor
- BufferedAspectAdaptor
- SelectionInList
- ValueModelManager

4 5 7

The class diagram for this hierarchy is shown in Figure 21:

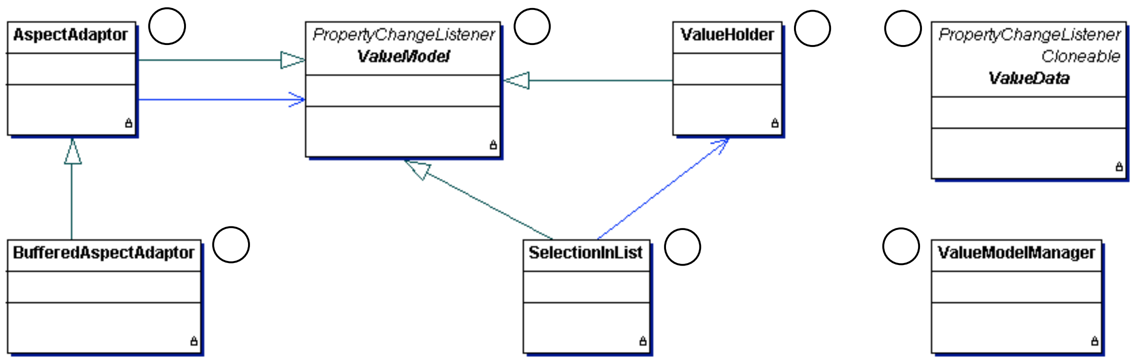


Figure 21: Demonstration model ValueModel class hierarchy.

As mentioned previously, the ValueModel (at **1**) is an abstract class that defines a simple attribute, and a ValueHolder (at **2**) extends ValueModel. AspectAdaptor (at **3**) and BufferedAspectAdaptor (**4**) are used to represent bean attributes. SelectionInList (at **5**) is used to represent a select collection. ValueData (at **6**) is an abstract class used to represent data objects and collections, and ValueModelManager (**7**) is used to manage the ValueModel instances as a collection (currently a HashTable).

There are two types of models that are created in ToolShed: (1) static models, and (2) dynamic models. Static models aren't modified at load time by dynamic data, while dynamic models are.

Static Model Creation

An example of ValueHolder, AspectAdaptor, and SelectionInList is shown in Figure 22:

```
<STATIC>
  <VALUEHOLDER   name="person"       value="getPerson"/>
  <ASPECTADAPTOR aspect="firstNameP" name="firstName" subjectChannel="person"/>
  <ASPECTADAPTOR aspect="lastNameP"  name="lastName"  subjectChannel="person"/>
</STATIC>
```

Figure 22: XML examples of ValueHolder and AspectAdaptor components in a static model.

The manner in which these items are constructed in Java (currently in the JUIBuilder class) is depicted in the following figures. The overall method is createStaticModels, and is shown as a sequence diagram in Figure 23:

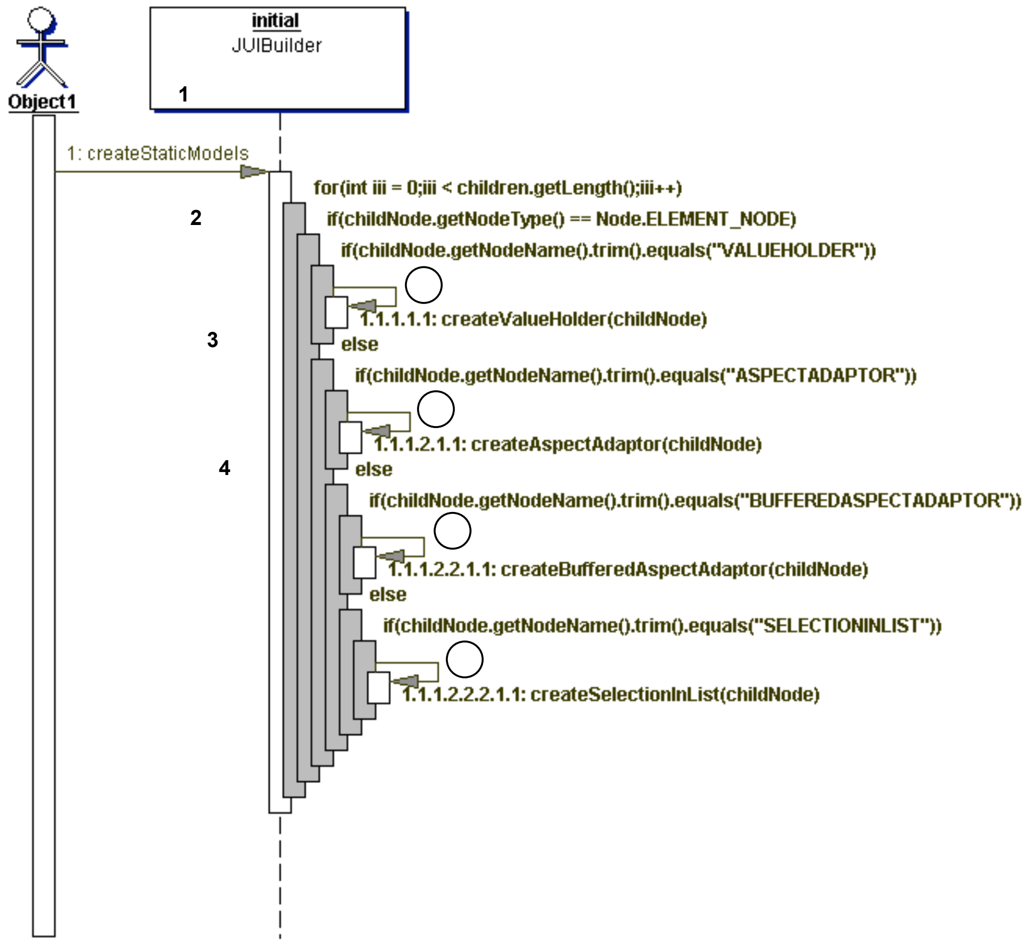

initial
JUIBuilder
Object1
1
1: createStaticModels
2
3
4
for(int iii = 0;iii < children.getLength();iii++)
if(childNode.getNodeType() == Node.ELEMENT_NODE)
if(childNode.getNodeName().trim().equals("VALUEHOLDER"))
1.1.1.1.1: createValueHolder(childNode)
else
if(childNode.getNodeName().trim().equals("ASPECTADAPTOR"))
1.1.1.2.1.1: createAspectAdaptor(childNode)
else
if(childNode.getNodeName().trim().equals("BUFFEREDASPECTADAPTOR"))
1.1.1.2.2.1.1: createBufferedAspectAdaptor(childNode)
else
if(childNode.getNodeName().trim().equals("SELECTIONINLIST"))
1.1.1.2.2.2.1.1: createSelectionInList(childNode)

Figure 23: [JUIBuilder] createStaticModels sequence diagram.

Notice that this method is a switching method, and dispatches control to one of createValueHolder (at **1**), createAspectAdaptor (at **2**), createBufferedAspectAdaptor (at **3**), or createSelectionInList (at **4**), which will be shown next. The createValueHolder sequence diagram is illustrated in Figure 24:

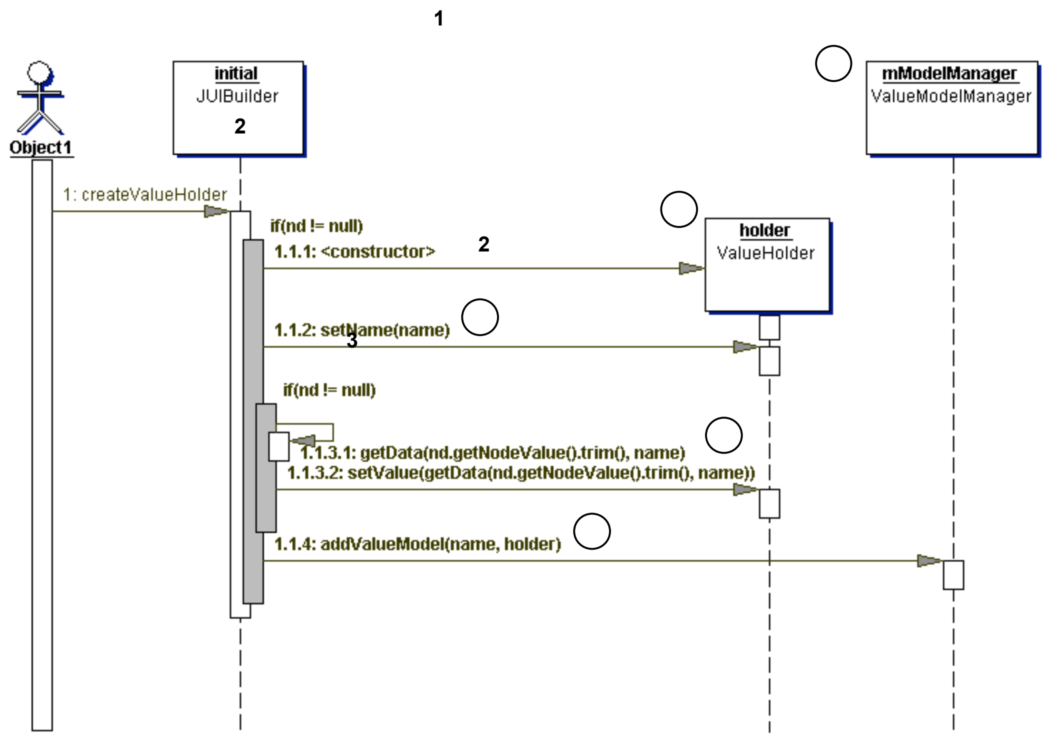


Figure 24: Sequence diagram for [JUIBuilder] createValueHolder.

In createValueHolder, the ValueHolder instance (at **1**) is populated (at **2**) and then added into the ValueModelManager (at **3, 4**). The createAspectAdaptor sequence diagram is illustrated in Figure 25:

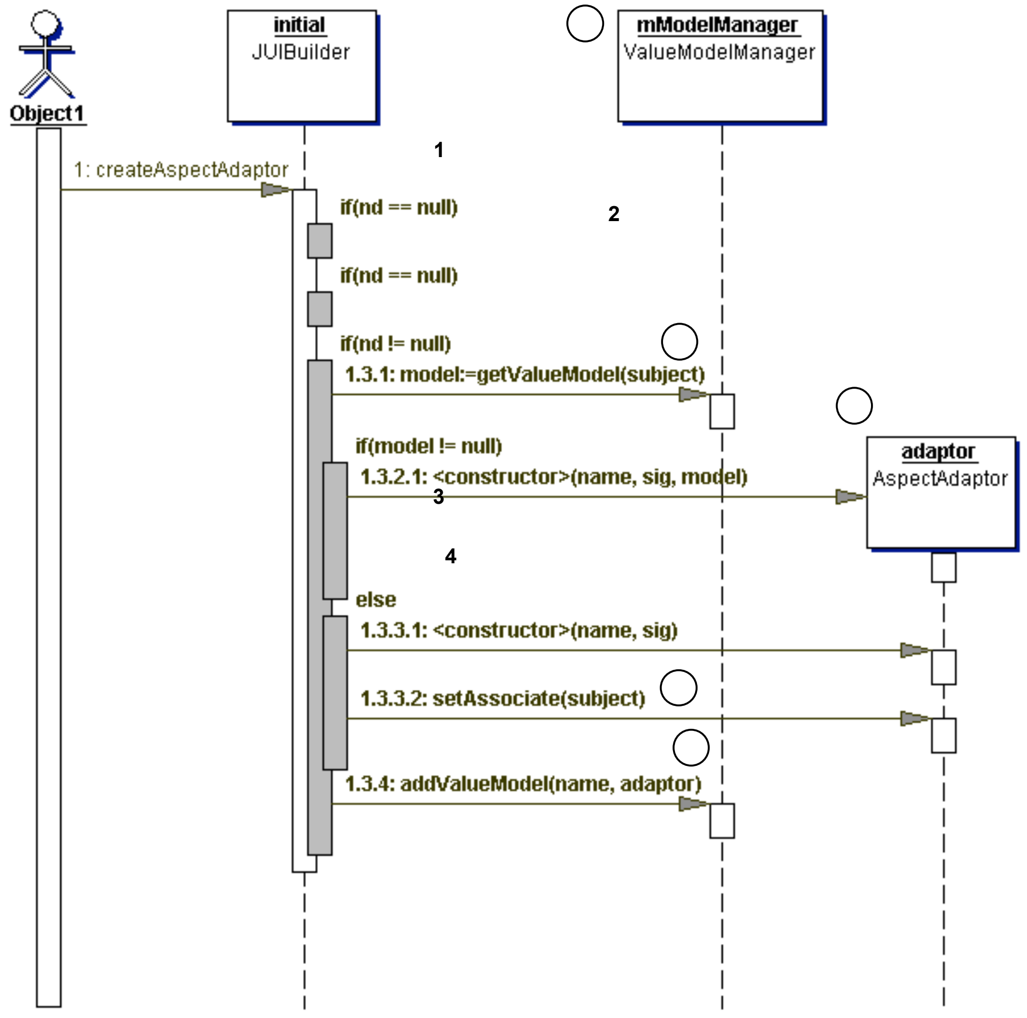

Object1
initial
JUIBuilder
mModelManager
ValueModelManager
adaptor
AspectAdaptor
1: createAspectAdaptor
if(nd == null)
if(nd == null)
if(nd != null)
1.3.1: model:=getValueModel(subject)
if(model != null)
1.3.2.1: <constructor>(name, sig, model)
else
1.3.3.1: <constructor>(name, sig)
1.3.3.2: setAssociate(subject)
1.3.4: addValueModel(name, adaptor)
1
2
3
4

Figure 25: [JUIBuilder] createAspectAdaptor sequence diagram.

In this figure, the AspectAdaptor object (at **1**) is instantiated, populated, and then added into the ValueModelManager (at **3**, **4**).

Dynamic Model Creation

Dynamic models can also be read using the same mechanisms and classes as static models, but also accommodate templates and mappings. The XML for an example dynamic model is illustrated in Figure 26:

```
<DYNAMIC template="select-model" id="select-model1">
  <template selection="a">
    <VALUEHOLDER name="person" value="getPerson"/>
    <BUFFEREDASPECTADAPTOR aspect="firstName" name="firstName" subjectChannel="person"/>
    <BUFFEREDASPECTADAPTOR aspect="lastName" name="lastName" subjectChannel="person"/>
    <SELECTIONINLIST name="persons" selectionHolder="person" list="getPersons"/>
    <mappings>
      <mapping key="tempModel" value="person" />
      <mapping key="listModel" value="persons" />
      <mapping key="model1" value="firstName" />
      <mapping key="model2" value="lastName" />
    </mappings>
  </template>
  <template selection="b">
    <VALUEHOLDER name="employee" value="getEmployee"/>
    <ASPECTADAPTOR aspect="firstName" name="firstName" subjectChannel="employee"/>
    <ASPECTADAPTOR aspect="employeeID"  name="employeeID" subjectChannel="employee"/>
    <SELECTIONINLIST name="employees" selectionHolder="employee" list="getEmployees"/>
    <mappings>
      <mapping key="tempModel" value="employee" />
      <mapping key="listModel" value="employees" />
      <mapping key="model1" value="firstName" />
      <mapping key="model2" value="employeeID" />
    </mappings>
  </template>
</DYNAMIC>
```

Figure 26: XML for dynamic model.

As can be seen in the example XML, the dynamic model, like the static model, is constructed using ValueHoder, AspectAdaptor, BufferedAspectAdaptor, and SelectionInList components. The difference is that the dynamic model can be associated with more than one model, and the model can be switched at run time. As such, it defines templates that are used to represent the possible models, and mappings between the items and the appropriate model. For example, template "a" (at **1**) uses a "person" model, where as template "b" (at **3**) uses an "employee" model. Each template in the dynamic node must have a mapping to the model attributes, using the "mapping" element. The mapping for template "a" is shown at (**2**), and for template "b" (at **4**). The sequence diagram for parsing dynamic models, using the createDynamicModels method, is depicted in Figure 27:

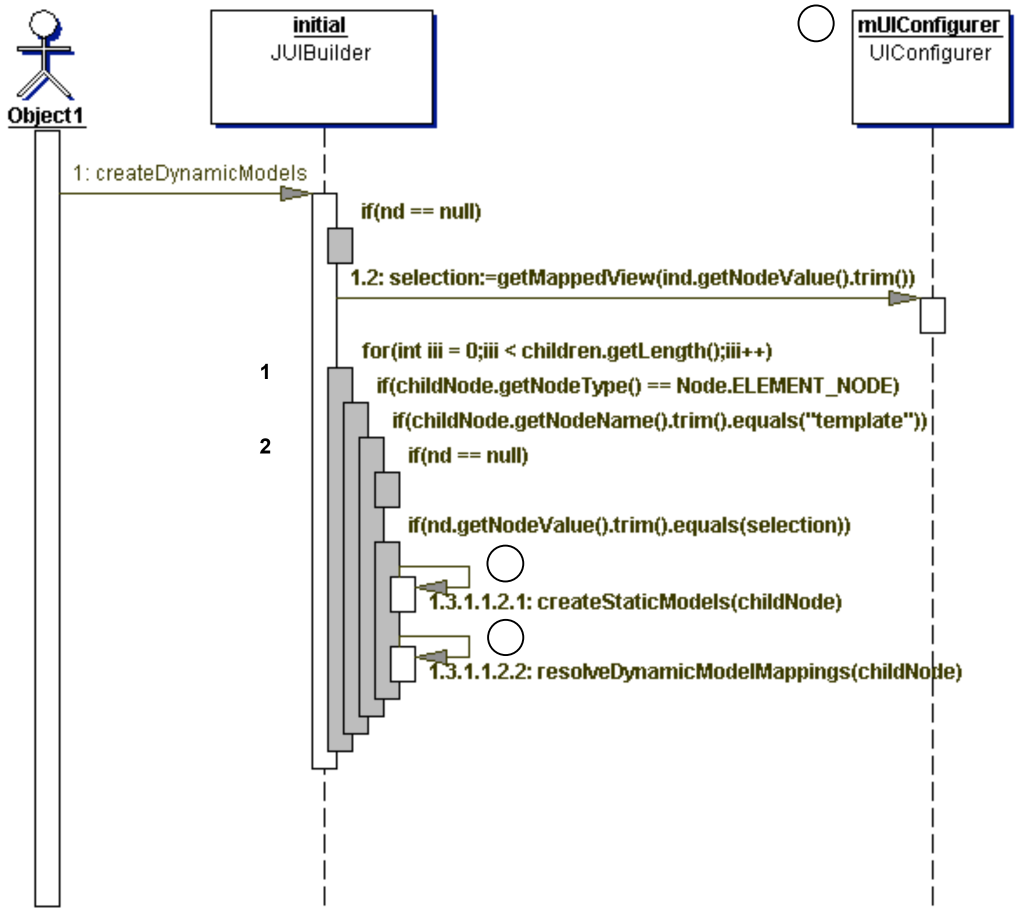

Object1
initial
JUIBuilder
mUIConfigurer
UIConfigurer
1: createDynamicModels
if(nd == null)
1.2: selection:=getMappedView(ind.getNodeValue().trim())
for(int iii = 0;iii < children.getLength();iii++)
1
if(childNode.getNodeType() == Node.ELEMENT_NODE)
if(childNode.getNodeName().trim().equals("template"))
2
if(nd == null)
if(nd.getNodeValue().trim().equals(selection))
1.3.1.1.2.1: createStaticModels(childNode)
1.3.1.1.2.2: resolveDynamicModelMappings(childNode)

Figure 27: [JUIBuilder] createDynamicModels sequence diagram.

In this figure, any static models that must be created for the node are done so at (**1**), and the resolution of template and template mappings is done at (**2**) with a call to resolveDynamicModelMappings. The latter parses the nodes children for any items marked “mapping” and calls a method that iterates through the node to get the attribute names and keys, and to add them to the template model manager. The createBufferedAspectAdaptor sequence is shown, since it appeared in Figure 16, in Figure 28:

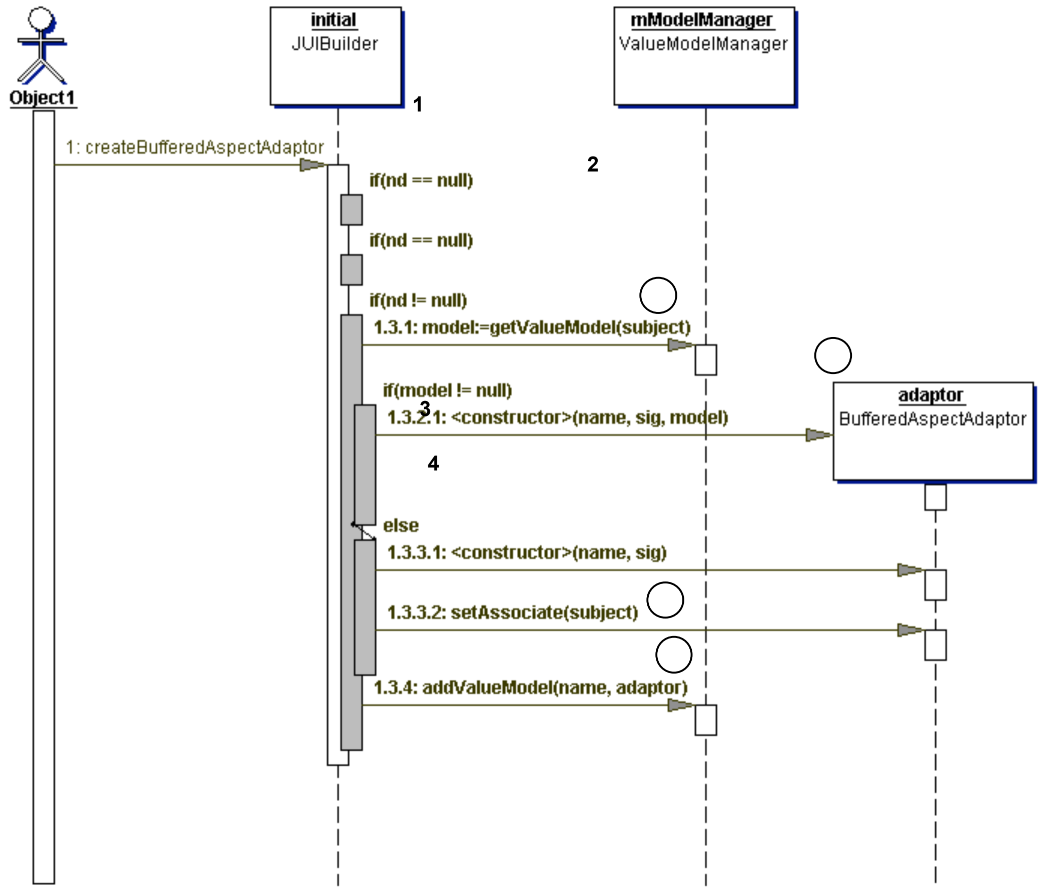

Object1
initial
JUIBuilder
mModelManager
ValueModelManager
1
2
1: createBufferedAspectAdaptor
if(nd == null)
if(nd == null)
if(nd != null)
1.3.1: model:=getValueModel(subject)
if(model != null)
3
1.3.2.1: <constructor>(name, sig, model)
adaptor
BufferedAspectAdaptor
4
else
1.3.3.1: <constructor>(name, sig)
1.3.3.2: setAssociate(subject)
1.3.4: addValueModel(name, adaptor)

Figure 28: [JUIBuilder] createBufferedAspectAdaptor sequence diagram.

Notice that, like the AspectAdaptor in Figure 15, the BufferedAspectAdaptor retrieves the ValueModel (at **1**) sets the subject (at **3**), and adds the ValueModel to the model manager (at **4**). The createSelectionInList flow (for the SelectionInList item shown in Figure 16) is shown in Figure 29:

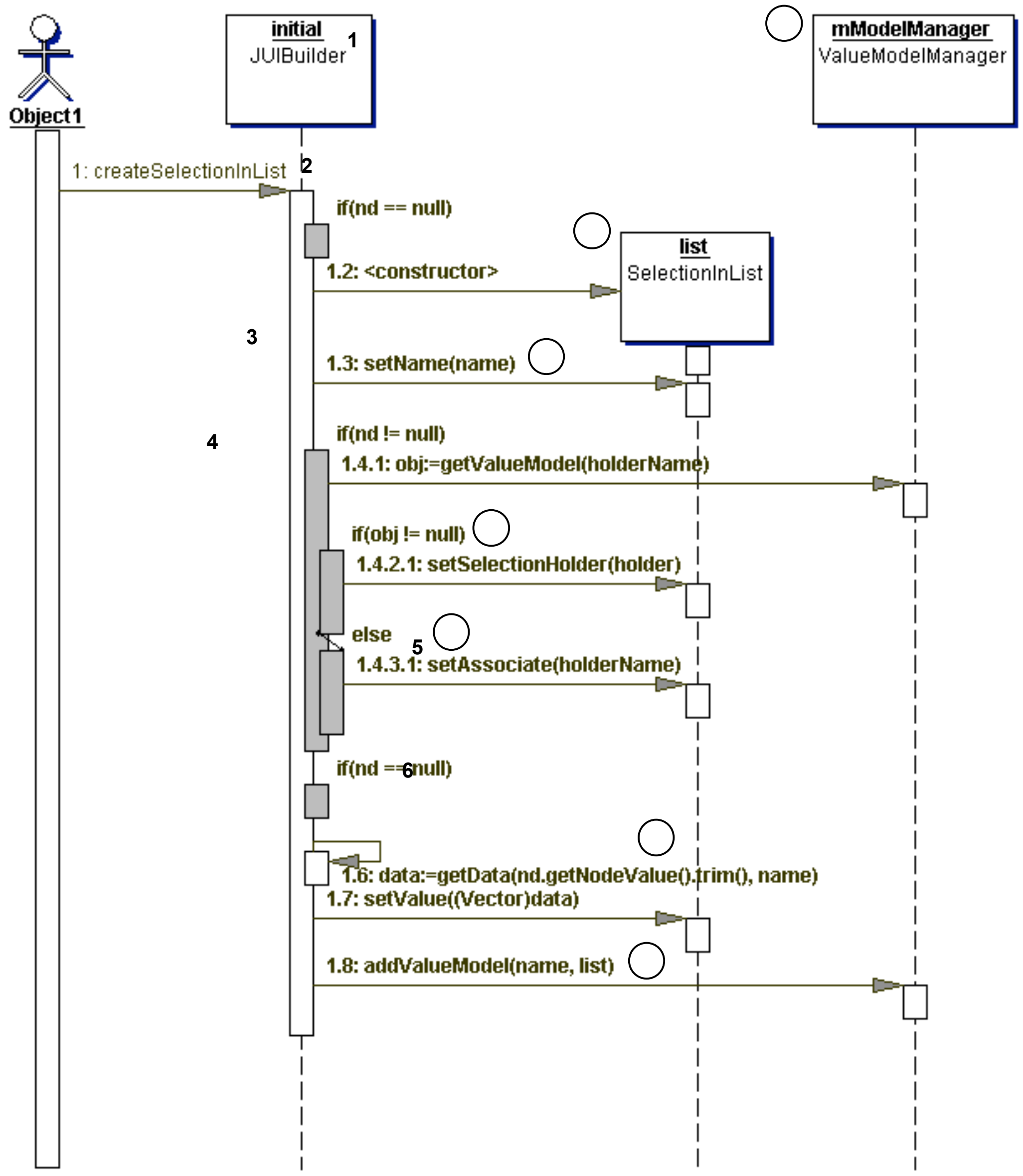

initial
JUIBuilder
mModelManager
ValueModelManager
Object1
1: createSelectionInList
if(nd == null)
list
SelectionInList
1.2: <constructor>
1.3: setName(name)
if(nd != null)
1.4.1: obj:=getValueModel(holderName)
if(obj != null)
1.4.2.1: setSelectionHolder(holder)
else
1.4.3.1: setAssociate(holderName)
if(nd == null)
1.6: data:=getData(nd.getNodeValue().trim(), name)
1.7: setValue((Vector)data)
1.8: addValueModel(name, list)
1
2
3
4
5
6

Figure 29: [JUIBuilder] createSelectionInList sequence diagram.

In this figure, a SelectionInList object is first created (at 1), the name is assigned from the element data (at **2**), and the ValueModel is retrieved and set to the selection holder (at **3**). This is the same as the AspectAdaptor. The BuffereAspectAdaptor differs in that its value can be changed dynamically (for example to update the value in a UI object), so it has a buffering adaptor called the associate. This is set (at **4**). The data is then retrieved from the element (at **5**), and added into the model manager (at **6**, **7**).

Notice that all four methods (createValueModel, createAspectAdaptor, createBufferedAspectAdaptor, and createSelectionInList) follow the same strategy of instantiating and populating the object from the XML data, and then adding the object instance into the ValueModelManager collection.

Views

Views represent the UI components themselves. They include layout managers, panels, and the specific components that reside in same. An example XML depicting a fragment of a view is shown in Figure 30:

```
<VIEW name="top">
  <CONSTRAINTS>
    <BORDER side="North" width="250" height="40"/>
  </CONSTRAINTS>
  <LAYOUTMANAGER>
    <BORDERLAYOUT hgap="0" vgap="0"/>
  </LAYOUTMANAGER>
  <VIEW name="right">
    <CONSTRAINTS>
      <BORDER side="East" width="250" height="40"/>
    </CONSTRAINTS>
    <LAYOUTMANAGER>
      <NULLLAYOUT />
    </LAYOUTMANAGER>
    <LABEL text="MailBox" name="MailBox" horizontalalignment="LEFT">
      <CONSTRAINTS>
        <BOUNDS x="10" y="10" width="60" height="25"/>
      </CONSTRAINTS>
    </LABEL>
    <INPUTFIELD model="Mailbox_Field"
                columns="20"
                name="Mailbox_Field"
                type="Integer"
                action="refresh">
      <CONSTRAINTS>
        <BOUNDS x="70" y="10" width="40" height="25"/>
      </CONSTRAINTS>
    </INPUTFIELD>
    <ACTIONBUTTON name="Refresh" text="Refresh" action="refresh">
      <CONSTRAINTS>
        <BOUNDS x="120" y="10" width="80" height="25"/>
      </CONSTRAINTS>
    </ACTIONBUTTON>
  </VIEW>
  <VIEW name="left">
    <CONSTRAINTS>
      <BORDER side="West" width="120" height="40"/>
    </CONSTRAINTS>
    <LAYOUTMANAGER>
      <NULLLAYOUT />
    </LAYOUTMANAGER>
    <ACTIONBUTTON name="About" text="About" action="about">
      <CONSTRAINTS>
        <BOUNDS x="10" y="10" width="80" height="25"/>
      </CONSTRAINTS>
    </ACTIONBUTTON>
  </VIEW>
</VIEW>
```

Figure 30: XML fragment for a View.

The view is decomposed into standard components in the XML: view (at**1**), which can include subviews; constraints (at **2**), which refer to properties of the object they reference. For example, the labeled constraint refers to the view border; layout manager (at **3**, which defines the placement of components within the view; and components (label, at **4**), input field (at **5**), and action button (at **6**).

UI-related components are parsed using the JUIBuilder initializeUI method, which calls intializeView, which calls initializeComponent. The sequence diagram for this last method is shown in Figure 31:

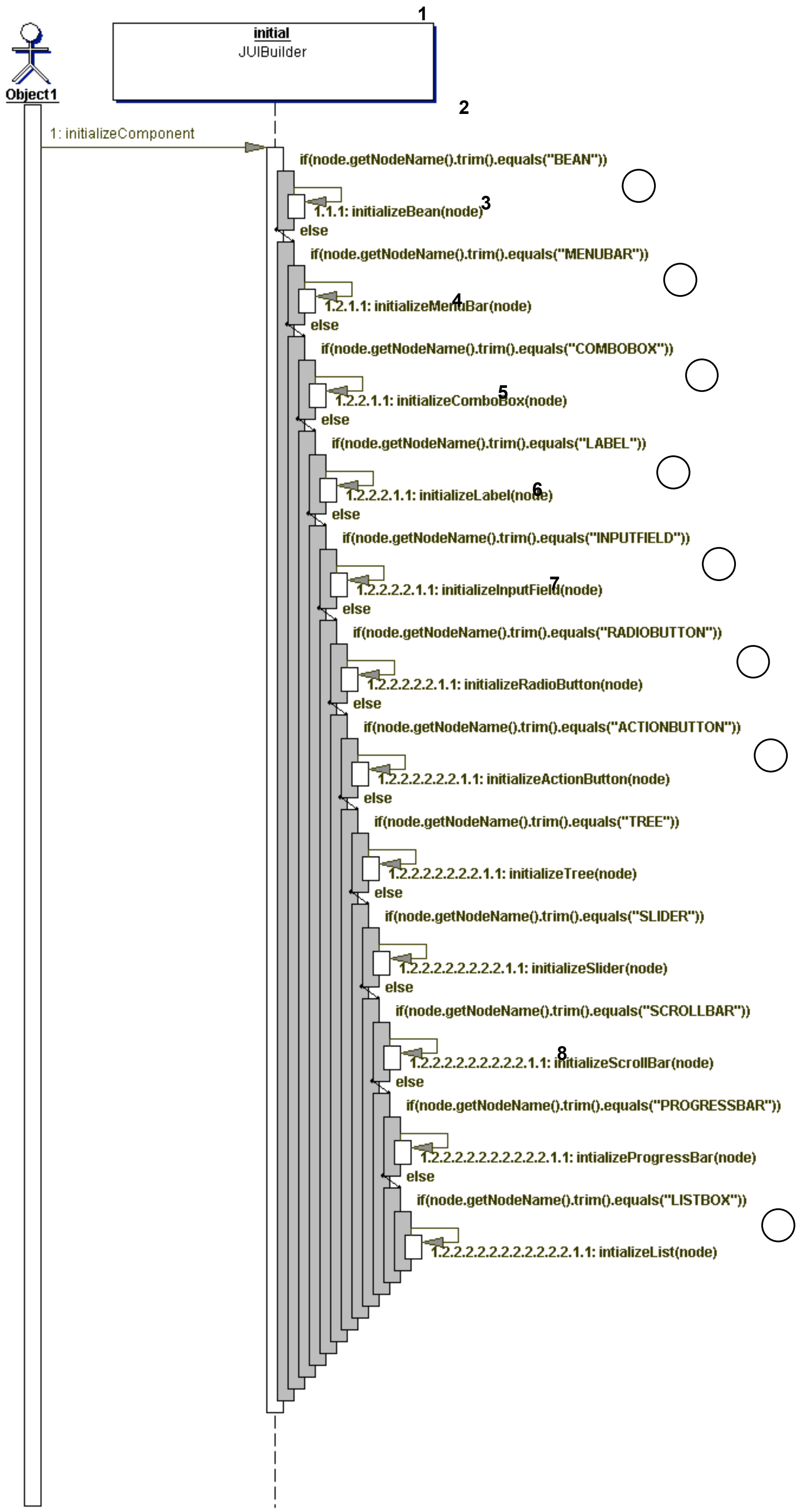

1
2
initial
JUIBuilder
Object1
1: initializeComponent
if(node.getNodeName().trim().equals("BEAN"))
1.1.1: initializeBean(node)
3
else
if(node.getNodeName().trim().equals("MENUBAR"))
1.2.1.1: initializeMenuBar(node)
4
else
if(node.getNodeName().trim().equals("COMBOBOX"))
1.2.2.1.1: initializeComboBox(node)
5
else
if(node.getNodeName().trim().equals("LABEL"))
1.2.2.2.1.1: initializeLabel(node)
6
else
if(node.getNodeName().trim().equals("INPUTFIELD"))
1.2.2.2.2.1.1: initializeInputField(node)
7
else
if(node.getNodeName().trim().equals("RADIOBUTTON"))
1.2.2.2.2.2.1.1: initializeRadioButton(node)
else
if(node.getNodeName().trim().equals("ACTIONBUTTON"))
1.2.2.2.2.2.2.1.1: initializeActionButton(node)
else
if(node.getNodeName().trim().equals("TREE"))
1.2.2.2.2.2.2.2.1.1: initializeTree(node)
else
if(node.getNodeName().trim().equals("SLIDER"))
1.2.2.2.2.2.2.2.2.1.1: initializeSlider(node)
else
if(node.getNodeName().trim().equals("SCROLLBAR"))
1.2.2.2.2.2.2.2.2.2.1.1: initializeScrollBar(node)
8
else
if(node.getNodeName().trim().equals("PROGRESSBAR"))
1.2.2.2.2.2.2.2.2.2.2.1.1: intializeProgressBar(node)
else
if(node.getNodeName().trim().equals("LISTBOX"))
1.2.2.2.2.2.2.2.2.2.2.2.1.1: intializeList(node)

Figure 31: JUIBuilder initializeComponent method, which is used to map XML nodes to Java UI components.
1

As can be 2 seen, the initializeComponent method is a switching method, based on the node name in the XML element. Items at labels (**1**-**8**) refer to the different UI components currently in use in the demonstration prototype, namely: Bean, MenuBar, ComboBox, Label, InputField, RadioButton, ActionButton, and ListBox, though other elements can be parsed in the method. When one of these items exists in the XML, the proper component based on the 3 name of the node is initialized. The various initializations wrap instantiations to Swing classes in ToolShed-specific components, copy the properties parsed from the XML into the components, and add the resulting components to a component manager. This process is illustrated, for combo box, in Figure 32:
4

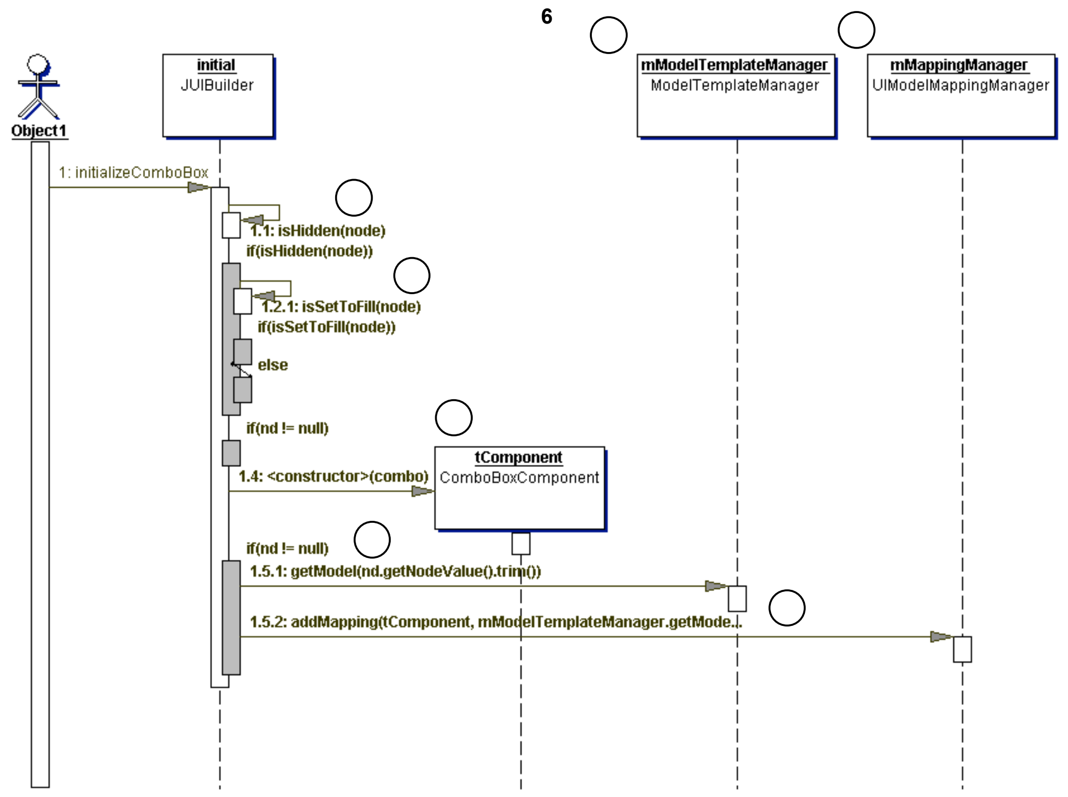


Figure 32: JUIBuilder initializeComboBox method.

The purpose of initializeComboBox, like that of the other UI component initialization methods, is to retrieve the component-specific data from the XML, to instantiate the proper Swing component, and to add the component to the collection. Before this process takes place, it must first be determined whether or not this UI component is hidden (at **1**). If not, then the second step is to find out if it is set to fill (at **2**). Then the object can be instantiated (at **3**). This is followed by retrieving the component data from the model template manager (at **4**, **5**), and adding the object component to the mapping manager (**6**, **7**).

1

UI Package

Each of the initialization methods in Figure 31: is used to map an XML element to a Swing Java class. This is mediated by a package of classes called the UI package, presented in Figure 33:

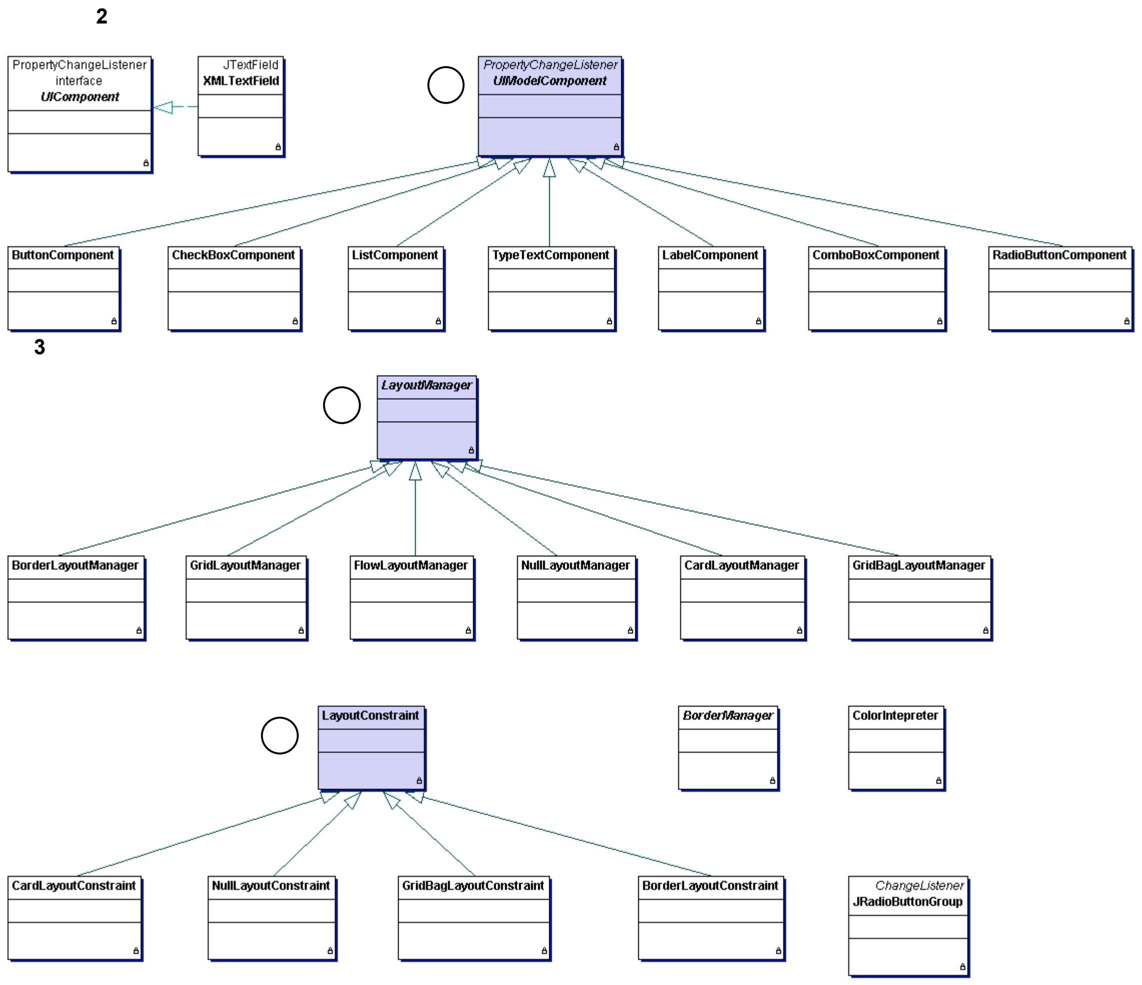


Figure 33: UI package components.

The components that extend the abstract class UIModelComponent (shown at **1**) each implement a particular Swing class. The UIModelComponent class extends PropertyChangeListener, so these components can accept event data and perform updates. Thus the classes extending UIModelComponent map as follows:

- ButtonComponent ⇨ JButton
- CheckBoxComponent ⇨ JCheckBox
- ListComponent ⇨ JList
- TypeTextComponent ⇨ JTextField
- LabelComponent ⇨ JLabel

- ComboBoxComponent ⇨ JComboBox
- RadioButtonComponent ⇨ JRadioButtonGroup
- TreeComponent ⇨ Jtree
- ProgressBarComponent ⇨ JProgressBar
- ViewComponent ⇨ JPanel
- DialogComponent ⇨ JDialog
- TableComponent ⇨ JTable
- ToolBarComponent ⇨ JToolBar
- TabbedPanelComponent ⇨ JTabbedPanel

1

The classes that inherit from the LayoutManager abstract class (at **2**) implement the standard layout types (null, flow, border, grid, gridbag, and card).

Finally, the classes that inherit from LayoutConstraint (at **3**) define the types of constraints that can be placed on these objects.

Beans Package

The last class package that is used by ToolShed to produce the user interface from XML is the beans package, which is a collection of custom objects developed at EFI. The beans package diagram is shown in Figure 34:

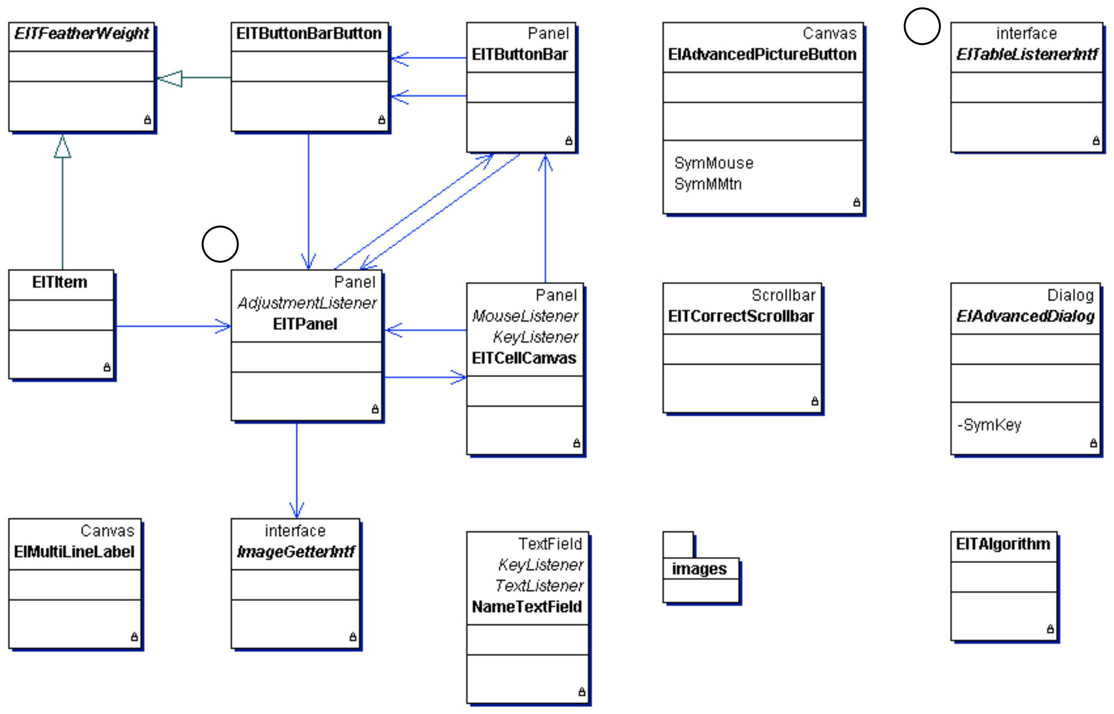


Figure 34: Beans package of EFI custom classes.

Only two of the beans package classes: EITPanel (at **1**) and EITableListenerIntf (at **2**) are used in the current prototype. **Note**: It has been decided that any custom components developed by EFI for ToolShed will inherit from Swing classes, so these classes will be modified to inherit from Swing classes or will be eliminated in the final ToolShed design.

Actions

Actions define what behavior can take place in the user interface other than updating widget values, which is performed by the property change listeners. The related XML elements are the ACTIONS and PERFORMACTION elements, as exemplified in Figure 35:

```
<ACTIONS>
  <PERFORMACTION name="delete" text="Delete" method="delete"/>
  <PERFORMACTION name="save" text="Save" method="save"/>
  <PERFORMACTION name="format" text="Format" method="format"/>
  <PERFORMACTION name="refresh" text="Refresh" method="refresh"/>
  <PERFORMACTION name="about" text="About" method="about"/>
</ACTIONS>
```

Figure 35: XML example of an actions segment. This one defines the actions in menus.

This figure illustrates the mechanism of registering actions for the application. In particular, note the “delete” method. When parsed, the delete method and its parameters are added to the ActionModelManager collection and then becomes available for use in UI components such as buttons. This is done in the initializeActions method, shown in Figure 36:

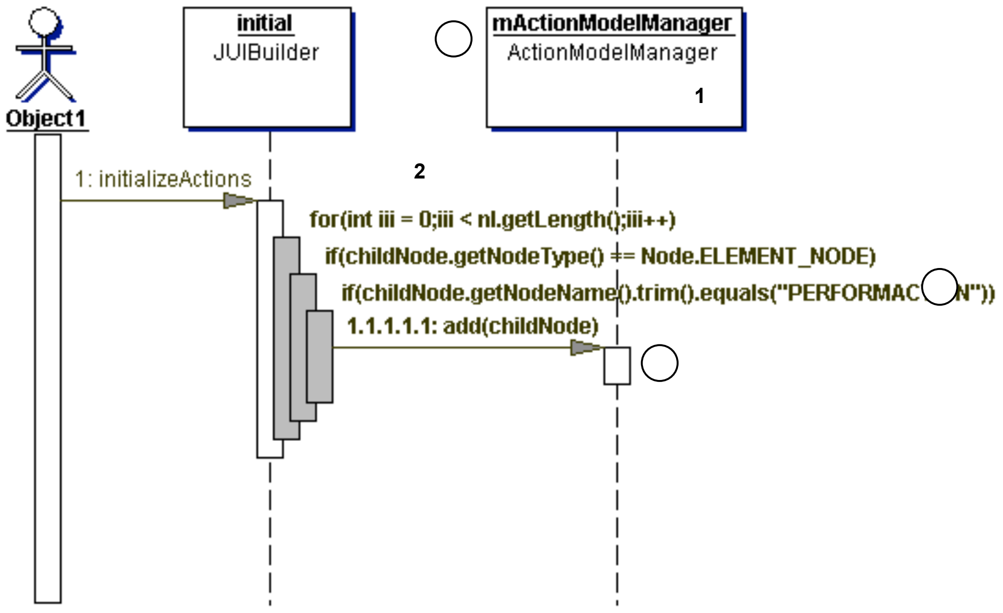

initial
JUIBuilder
mActionModelManager
ActionModelManager
1
Object1
1: initializeActions
2
for(int iii = 0;iii < nl.getLength();iii++)
if(childNode.getNodeType() == Node.ELEMENT_NODE)
if(childNode.getNodeName().trim().equals("PERFORMACTION"))
1.1.1.1.1: add(childNode)

Figure 36: JUIBuilder sequence diagram for initializeActions.

The entry parameter to initializeActions is an "actions" element. The children of this element are iterated upon and any that are "performaction" elements (at **1**) are then added to the action model manager (at **2**, **3**). The XML mapping between an action defined and an action applied to a widget is illustrated in Figure 37:

```
<ACTIONBUTTON name="Delete" text="Delete" action="delete">
  <CONSTRAINTS>
    <BOUNDS x="10" y="10" width="80" height="25"/>
  </CONSTRAINTS>
</ACTIONBUTTON>
```

Figure 37: XML example showing how the "delete" action from the Figure 24 is used in an ActionButton.

This XML fragment relates an ActionButton (which is mapped to a ButtonComponent and becomes a JButton Swing component) to the delete action that was registered in the ACTIONS element. That is, the actions possible to perform are registered and then referenced in the actual elements. Part of the verification process, presumably, makes certain that any action referenced in a UI component has been registered in the ACTIONS element.

Application-Specific Packages (appls)

Event handling in ToolShed is partly a UI-specific task, and partly a behavioral task. The UI components must listen to events and respond to them, but business logic support is needed to respond to events in a coherent and coordinated fashion. Although it is theoretically possible to implement all of the business logic using the same rule-based mechanisms as will be used for UI constraint validation (below), it is unlikely that this mode of evaluation can be as effective as direct object development. As such, the behaviors of existing objects will be migrated from WebTools 1 and refactored to accommodate the ToolShed architecture. To be completed.

Data Validation

Validation is performed at three stages in the ToolShed model: (1) during generation when the XML files are validated against the XSD, (2) at run time when the UI models are added into the manager collections, and (3) at runtime when new events are handled.

Generate-Time Validation

When the files are initially created, they must satisfy the constraints imposed by the XML Schema (XSD) that defines the models and their data elements. This level of validation essentially certifies that every element in the model follows the model definition and that the data elements included therein follow the type and range definitions of the model. This level of certification is static, in that it is an initialization phase of validation only.

Runtime Validation

There are three types of runtime validation: (1) models have values, (2) models have valid types and values, and (3) objects have correct values with respect to the entire interface/server. They will sometimes be referred to as Type 1, Type 2, and Type 3 validation in this document. The current prototype has a validation mechanism that meets the first requirement. That is, a model is valid if it has a value/object associated with it. This will be called object, or Type 1, validation. Validation, in this context, means only that since there is an object associated with the component, there is reasonable cause to assign a property change listener to it and, otherwise, to ignore it. This mechanism will be described in this section, and the second and third validation mechanisms will be discussed in the following sections.

Object Validation

During initialization, data values may be provided as initial/default values by the product designers or by server requirements. These may be validated by the design application, but they also may not, so a software validation is needed to simplify the client-side processing. This mechanism is invoked during initialization. In the current demonstration prototype, every page that is displayed executes an [JUIBuilder] initializeUI method, which is re-presented from Figure 14 below, as Figure 38:

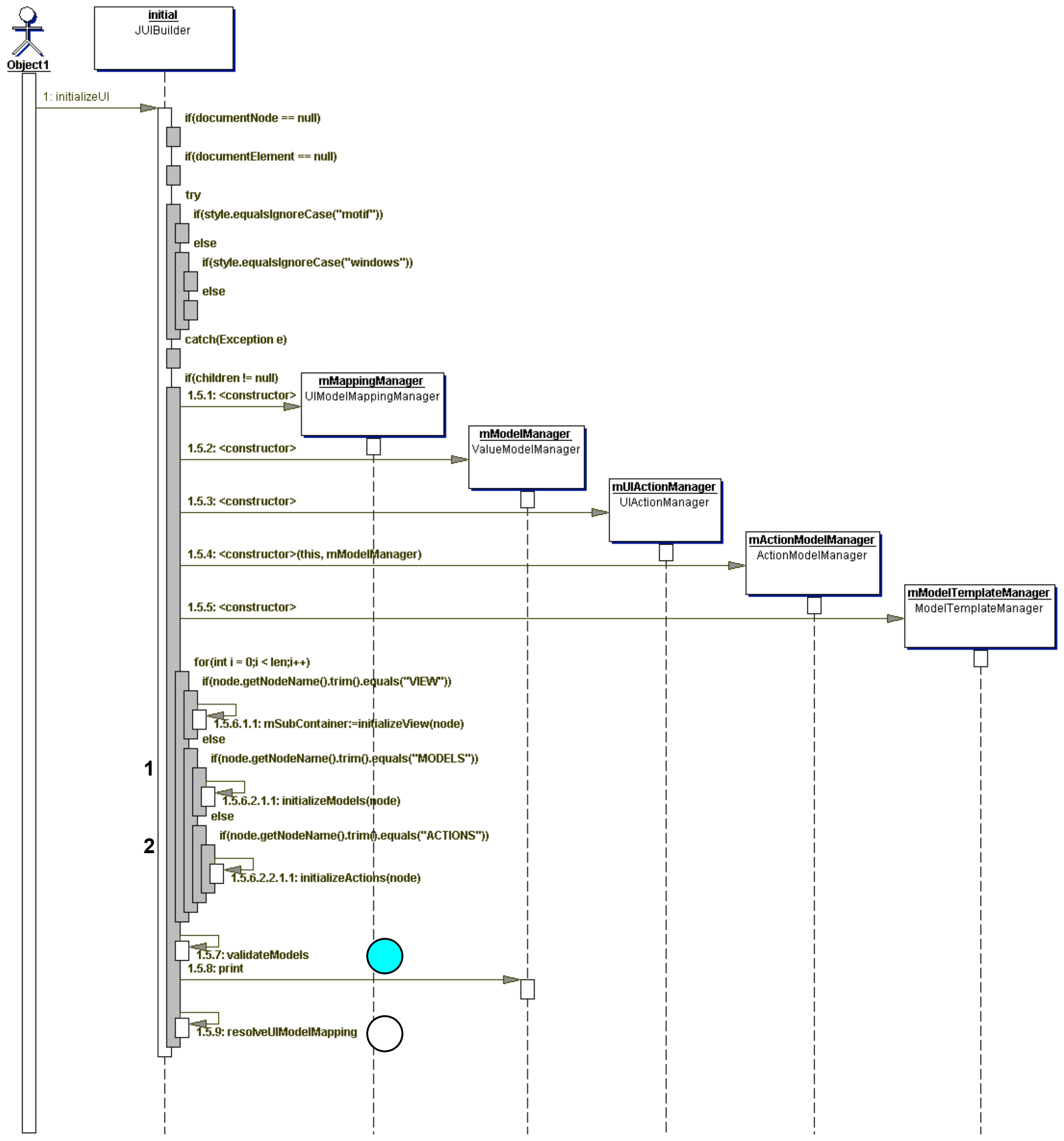

Object1
initial
JUIBuilder
1: initializeUI
if(documentNode == null)
if(documentElement == null)
try
if(style.equalsIgnoreCase("motif"))
else
if(style.equalsIgnoreCase("windows"))
else
catch(Exception e)
if(children != null)
1.5.1: <constructor>
mMappingManager
UIModelMappingManager
1.5.2: <constructor>
mModelManager
ValueModelManager
1.5.3: <constructor>
mUIActionManager
UIActionManager
1.5.4: <constructor>(this, mModelManager)
mActionModelManager
ActionModelManager
1.5.5: <constructor>
mModelTemplateManager
ModelTemplateManager
for(int i = 0;i < len;i++)
if(node.getNodeName().trim().equals("VIEW"))
1.5.6.1.1: mSubContainer:=initializeView(node)
else
if(node.getNodeName().trim().equals("MODELS"))
1.5.6.2.1.1: initializeModels(node)
else
if(node.getNodeName().trim().equals("ACTIONS"))
1.5.6.2.2.1.1: initializeActions(node)
1.5.7: validateModels
1.5.8: print
1.5.9: resolveUIModelMapping
1
2

Figure 38: [JUIBuilder] initializeUI method reproduced from Figure 10. Focus is on validateModels call.

After all models have been parsed, instantiated into their respective UI classes, and added into the model collections, they must be validated. The calls to validateModels (shown at **1**), and resolveUIModelMapping (at **2**), are the final initialization steps. The validateModels method is further delineated below, as Figure 39:

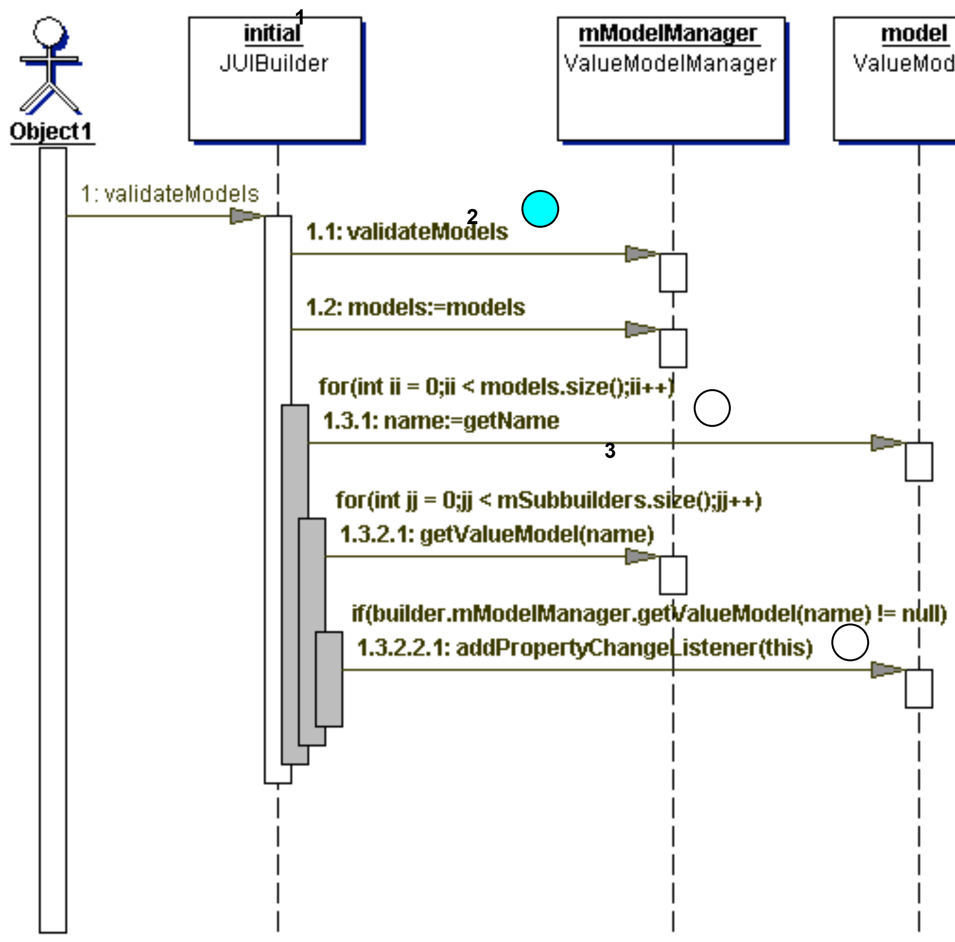

1
initial
JUIBuilder
mModelManager
ValueModelManager
model
ValueModel
Object1
1: validateModels
2
1.1: validateModels
1.2: models:=models
for(int ii = 0;ii < models.size();ii++)
1.3.1: name:=getName
3
for(int jj = 0;jj < mSubbuilders.size();jj++)
1.3.2.1: getValueModel(name)
if(builder.mModelManager.getValueModel(name) != null)
1.3.2.2.1: addPropertyChangeListener(this)

Figure 39: [JUIBuilder] validateModels method.

The validateModels method serves two purposed. First, it calls the validateModels method in the ValueModelManager (shown at **1**). Second, it iterates through all the models that were validated in the model manager (at **2**), gets each model and its name, and adds a property change listener to each subbuilder that matches a name (at **3**). So what is a subbuilder? Recall that the presentation may be comprised of arbitrarily deeply nested UI components. These nested components are the subbuilders. The ValueModelManager validateModels method, itself, is further broken down in Figure 40:

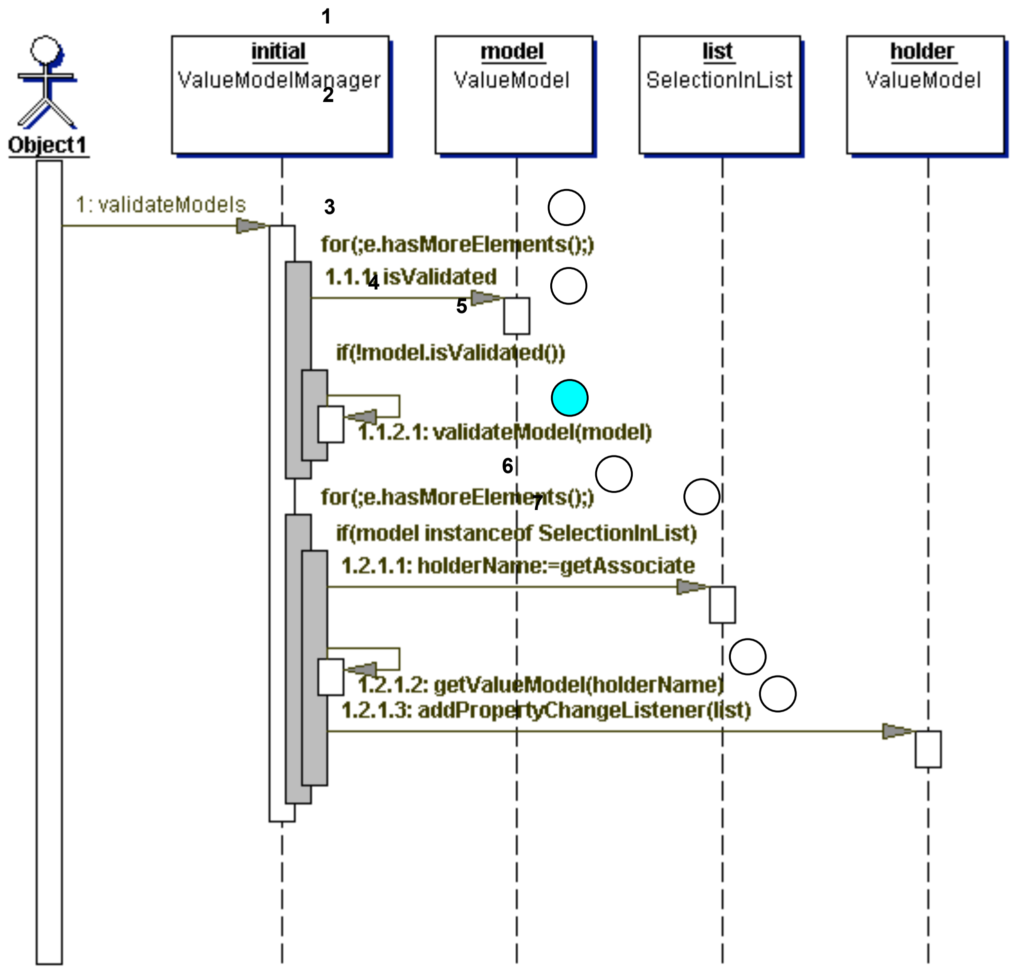

1
initial
ValueModelManager
2
model
ValueModel
list
SelectionInList
holder
ValueModel
Object1
1: validateModels
3
for(;e.hasMoreElements();)
1.1.1: isValidated
4
5
if(!model.isValidated())
1.1.2.1: validateModel(model)
6
for(;e.hasMoreElements();)
7
if(model instanceof SelectionInList)
1.2.1.1: holderName:=getAssociate
1.2.1.2: getValueModel(holderName)
1.2.1.3: addPropertyChangeListener(list)

Figure 40: Sequence diagram for [ValueModelManager] validateModels.

The ValueModelManager validateModels method is similar to the JUIBuilder version, in that it first checks to see if a model is validated and also adds property change listeners to the object. It achieves this by first retrieving the model names from the manager collection as an enumeration (shown at **1**) and then iterates through this enumeration. In the iteration, it gets a model and checks to see if it is already validated (at **2**). If not, an attempt to validate it is made by calling validateModel (at **3**). The method then recreates the enumeration from the manager and iterates on the collection (at **4**). For any item that is a SelectionInList element (at **5**), a property change listener is added to its value holder instance (at **6**, **7**).

The validateModel method is further decomposed in Figure 41:

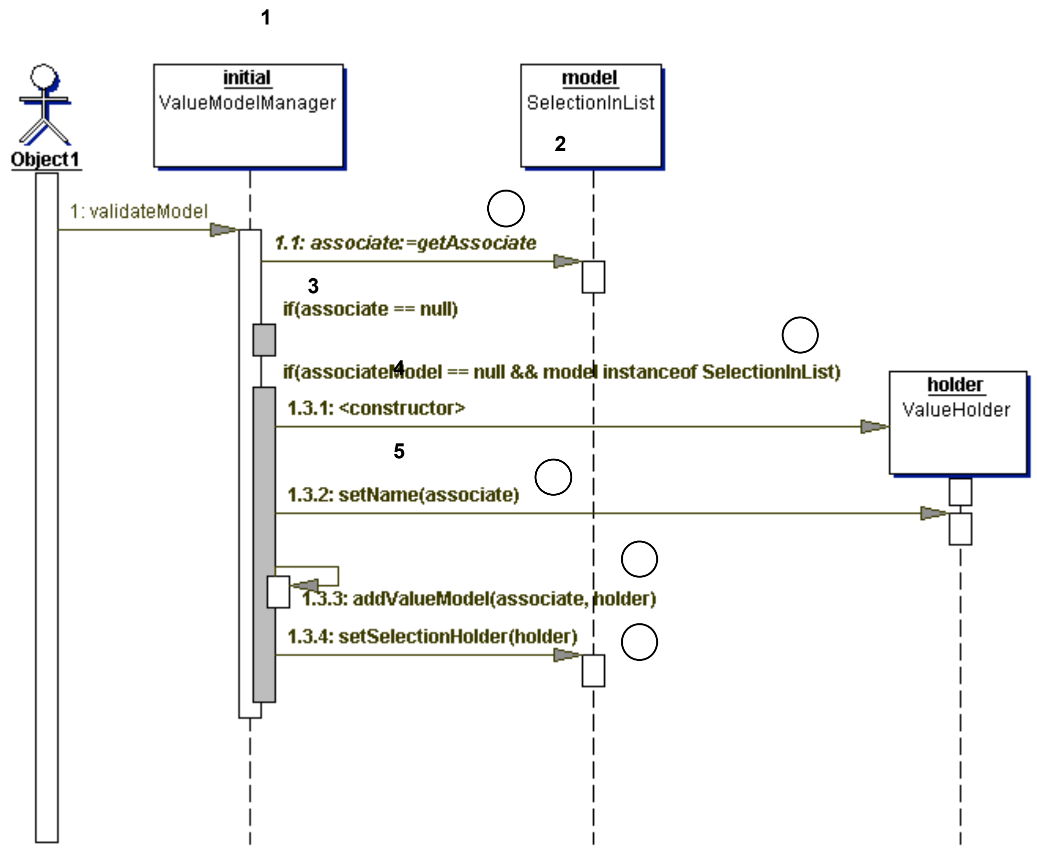

1
initial
ValueModelManager
model
SelectionInList
2
Object1
1: validateModel
1.1: associate:=getAssociate
3
if(associate == null)
if(associateModel == null && model instanceof SelectionInList)
4
holder
ValueHolder
1.3.1: <constructor>
5
1.3.2: setName(associate)
1.3.3: addValueModel(associate, holder)
1.3.4: setSelectionHolder(holder)

Figure 41: [ValueModelManager] validateModel method sequence diagram.

The validateModel method checks to see if there is an associate (at **1**), meaning that there is a ValueHolder for the model. If not (at **2**), then if the model is also a SelectionInList the method creates a new ValueHolder (at **3**), sets its name to the name of the associate (at **4**), and binds the ValueHolder to the model (at **5**).

Data Type and Value Validation

During runtime, data values can be provided by users or by the system in response to user actions, such as typing data into text fields. In the case where data is provided by users, the data must be revalidated according to the model definition provided at read time. The data file must be formatted in such a way that it can be read both by the data type and value validator and by the constraint validation mechanism. Depending on the communications source (SOAP or SNMP/Harmony/ Dictionary), the data may take different forms, so the same data must be validated based on the source, the type it is supposed to be implemented as, and its value. This will be referred to as Type 2 validation.

Representative data types and default value tests for Server Setup are shown below, as taken from the Fiery WebTools2 Product Specification, dated 4/16/2003. The contents of section 11.1 of that document have been translated into three tables, one for server setup, network setup, and printer setup, as shown in Tables 2-4 below. It should be noted that the representation mechanisms described in this section apply equally well to any application that can be presented by ToolShed, and that setup has been illustrated due to the fact that it is the single most complicated application presented by ToolShed and thus covers the variety of functional issues that must be addressed by the model.

UI Tab	Option	Data Type	Data Value/Range
Server Setup	Server Name	String	15 chars max
Server Setup	Date	String (3)	dd, mm, yy
Server Setup	Time	String (2)	hh, mm, 24-hr clock, 00-23
Server Setup	Enable Printed Queue	boolean	true (def) false
Server Setup	Jobs Saved in Printed Queue	int	10 (def), 1-99
Server Setup	Preview While Processing	boolean	true false (def)
Server Setup	Use Character Set	String (select)	Macintosh Windows (def) DOS
Server Setup	Print Start Page	boolean	true false (def)
Server Setup	Enable Printing Groups	boolean	true false (def)
Server Setup	Clear Each Scan Job	String (select)	Delete All Scan Jobs 1 day after scan (def) 1 week after scan Manual
Server Setup	Clear Each Scan Job Now	boolean	true false (def)
Server Setup	Administrator	String	19 chars max
Server Setup	Operator	String	19 chars max
Server Setup	Auto Print Job Every 55 Jobs	boolean	true false (def)
Server Setup	Auto Clear Job Every 55 Jobs	boolean	true false (def)
Server Setup	Job Log Page Size	String (select)	Tabloid/A3 Letter/A4
Server Setup	Fiery Contact Name	String	18 chars max
Server Setup	Fiery Contact Phone	String	18 chars max
Server Setup	Fiery Contact E-mail	String	
Server Setup	Device Contact Name	String	18 chars max
Server Setup	Device Contact Phone	String	18 chars max
Server Setup	Device Contact E-mail	String	

Table 3: Server Setup data types and (default/range) values.

Records appearing in cyan refer to requirements from the Fiery Product Specification 5.5e, dated 2/28/2003 that failed to appear in Fiery Product Specification dated 4/16/2003. They are retained but highlighted in this document until their status is ascertained. Although non-string values are identified in this table, it should be clear to the reader that all the values can be represented as strings and have in past incarnations of WebTools. Similar values are tabulated, for Network Setup, in Table 4:

UI Tab	Option	Data Type	Data Value/Range
Network Setup	Enable Ethernet	boolean	true (def) false
Network Setup	Ethernet Speed	String (select)	Auto Detect (def) 1 Gbps 100 Mbps Full Duplex 100 Mbps Half Duplex 10 Mbps Full Duplex 10 Mbps Half Duplex
Network Setup	Enable Apple Talk	boolean	true (def) false
Network Setup	Apple Talk Zone	String (select)	network defined, only enabled when the Enable Apple Talk is 'true'
Network Setup	Enable TCP/IP for Ethernet	boolean	true (def) false
Network Setup	Enable AutoIP for Ethernet	boolean	true (def) false, only enabled when Enable Ethernet is 'true'
Network Setup	Select Protocol	String (select)	DHCP (def) BOOTP Only enabled when Enable Ethernet is 'true'
Network Setup	IP Address	String (4)	127.0.0.1 (def, network defined Fiery address), user editable, only enabled when the Enable Ethernet is 'true' and when the AutoIP is 'false'. Each value is 0-255.
Network Setup	Subnet Mask	String (4)	255.255.255.0 (def), enabled only when Enable Ethernet is 'true' and when AutoIP is 'false'. Each value is 0-255
Network Setup	Enable Gateway Automatically	boolean	true (def) false
Network Setup	Gateway Address	String	127.0.0.1 (def), user editable
Network Setup	Enable TCP/IP for Token Ring	boolean	true (def) false
Network Setup	Enable DNS	boolean	true (def) false
Network Setup	Get DNS Address Automatically	boolean	true (def) false Only enabled if AutoIP configuration is 'true'
Network Setup	Primary DNS Server IP Address	String	127.0.0.1 (def), only enabled if AutoIP Configuration is 'false'
Network Setup	Secondary DNS Server IP Address	String	127.0.0.1 (def), on enabled if AutoIP Configuration is 'false'
Network Setup	Domain Name	String	Only enabled if AutoIP Configuration is 'false'
Network Setup	Use DNS on	String (select)	Any (def)

			Ethernet Token Ring Only enabled if AutoIP Configuration is 'true', token ring is installed on the Fiery
Network Setup	Host Name	String	Only enabled if AutoIP Configuration is 'false'
Network Setup	Configure IP Ports	boolean	true false (def)
Network Setup	Enabled IP Ports	String	80 (HTTP) 137-139 (NETBIOS) 161-162, (SNMP) 515 (LPD) 631 (IPP) 9100-9103 (9100) EFI Ports All (by default) are selected, any deselected also disable the associated service
Network Setup	Enable IPX Auto Frame Type	boolean	true false (def)
Network Setup	Select Frame Types	String (select)	Ethernet 802.2 Ethernet 802.3 Ethernet II Ethernet SNAP Token Ring Token Ring SNAP
Network Setup	Clear Frame Types	boolean	true false (def)
Network Setup	Enable LPD	boolean	true (def) false
Network Setup	Enable PServer	boolean	true false (def)
Network Setup	NetWare Server PServer Poll Interval in Seconds	int	15 (def), 1-3600 seconds
Network Setup	Enable NDS	boolean	true false (def)
Network Setup	Select NDS Tree	String (select)	Network defined
Network Setup	Delete Bindery setup and continue	boolean	true false (def)
Network Setup	Is user login needed to browse NDS tree?	boolean	true false (def)
Network Setup	NDS Tree Name	String	Network defined
Network Setup	Current path	String	Network defined
Network Setup	Enter Password	String	
Network Setup	Current path	String	Network defined
Network Setup	Enter Your Print Server Password	String	
Network Setup	Server should look for print queues in	String	Entire NDS tree (def), network defined
Network Setup	NDS Tree Name	String	Network defined

Network Setup	Current path	String	Network defined
Network Setup	Choose File Server	String	Network defined
Network Setup	Supported Servers	String	Network defined
Network Setup	Remove support for	String	Network defined
Network Setup	Select File Server	String	Network defined
Network Setup	Enter First Letters of Server Name	ch	'A', user defined
Network Setup	Add Server	String	'AAAAA', network defined
Network Setup	Add Server	String	'Server1', network defined
Network Setup	File Server Login	String	
Network Setup	Enter Your Login Name	String	'guest', user defined
Network Setup	Enter Your File Server Password	String	user defined
Network Setup	NetWare Print Server	String	network defined
Network Setup	Enter Your Print Server Password	String	user defined
Network Setup	Enable Windows Printing	boolean	true (def) false
Network Setup	Use WINS Name Server	boolean	true false (def) enabled when Enable Windows Printing is 'true', and when Auto IP is 'false'
Network Setup	WINS IP Address	String	127.0.0.1 (def), enabled only if Use WINS Name Server is 'true'
Network Setup	Server Name	String	Enabled when Enable Windows Printing is 'true', 15 chars max
Network Setup	Server Comments	String	Enabled when Enable Windows is 'true', 15 chars max
Network Setup	Set Domain Name	String	Select from List (def) Enter Manually Enabled when Enable Windows Printing is 'true', 15 chars max
Network Setup	Workgroup or Domain	String	network defined
Network Setup	Choose Domain	String	network defined
Network Setup	Set Driver Type	String	PS (def) PCL Only available if Windows Printing is supported, is 'true'
Network Setup	Enable Web Services	boolean	true false (def)
Network Setup	Enable IPP	boolean	true (def) false
Network Setup	Enable SNMP	boolean	true (def) false
Network Setup	Specify Read Community Name	String	Public (def) User defined
Network Setup	Specify Write Community Name	String	Public (def) User defined
Network Setup	Enable Port 9100	boolean	true (def) false

Network Setup	Enable Parallel Port	boolean	true (def) false
Network Setup	Ignore EOF Character	boolean	true false (def)
Network Setup	Parallel Port Timeout (seconds)	int	5 (def), 5-60
Network Setup	Enable on Ethernet	boolean	true (def) false
Network Setup	Auto Select/Manual Select	boolean	true false
Network Setup	Select Frames	boolean	true false, enabled when Manual Select is 'true'
Network Setup	Bindery Setup	boolean	true false (def) Enabled when Enable PServer is 'true'
Network Setup	Change Trees	boolean	true false (def) Enabled when Enable NDS is 'true'
Network Setup	Enable LPD Printing Service	boolean	true (def) false
Network Setup	Enable FTP Services	boolean	true (def) false

Table 4: Network Setup data types and (default/range) values.

Records colored in orange lack sufficient information in the Fiery Product Specification, dated 04/16/2003, to determine their data type or range of values. Records colored in cyan were missing from the Fiery Product Specification, dated 04/16/2003. Cells colored in green represent business logic that must be implemented using Type III rules. Similar values, for Printer Setup, are shown in Table 5:

UI Tab	Option	Data Type	Data Value/Range
Printer Setup	Enable Print Queue	boolean	true (def) false
Printer Setup	Enable Direct Connection	boolean	true (def) false
Printer Setup	Enable Hold Queue	boolean	true (def) false
Printer Setup	Default Paper Sizes	String (select)	US (def) Metric
Printer Setup	Convert Paper Sizes	String (select)	No Letter/Tabloid>A4/A3 A4/A3>Leter/Tabloid (def)
Printer Setup	Page Order	String (select)	Forward (def) Reverse
Printer Setup	Default Color Mode	String (select)	CMYK (def) Grayscale
Printer Setup	Print Cover Page	boolean	true false (def)
Printer Setup	Allow Courier Substitution	boolean	true (def)

			false
Printer Setup	Print to PS Error	boolean	true false (def)
Printer Setup	Print Master	boolean	true false (def)
Printer Setup	Select Printer	String	
Printer Setup	Printer Type	String	
Printer Setup	Parallel Connection	String (select)	Print Queue (def) Hold Queue Direct Connection Only published queues may be selected
PCL Setup	Paper Size	String	PPD defined
PCL Setup	Orientation	String (select)	Portrait (def) Landscape
PCL Setup	Form Length	int	60 (def), 5-128
PCL Setup	Font Source	String (select)	Internal (def) Softfont (Internal)
PCL Setup	Font Number	int	0 (def), 0-999
PCL Setup	Font Pitch	real	10.00 (def), 0.44-999.99
PCL Setup	Points (Font Size)	real	12.00 (def), 4.0-999.75
PCL Setup	Symbol Set	String (select)	ASCII (def), ROMAN_8, ECMA_94L1, PC_8, DN, PC_850, ISO_SWED_NAMES, ISO_NORWEGIAN, LEGAL, VENTURA_INTNTL, VENTURA_USA, DESKTOP, WINDOWS_L1, PS_TEXT, ISO_ITALIAN, ISO_FRENCH, MATH_8, PS_MATH, PI_FONT, PC_852, WINDOWS_L2, VENTURA_MATH, WINDOWS31_L1, ISO_LATIN2, ISO_LATIN5, MICROSOFT_PUB, PC_TURK, WIN_LATIN5, ISO_UK, ISO_GERMAN (ASCII)

Table 5: Printer Setup data types and (default/range) values.

The objects identified in Tables 2-4 represent three data object groups: (1) Server, (2) Network, and (3) Printer. Table records colored cyan have been removed from the design specification without explanation. Cells colored in green represent business logic that must be implemented using Type III rules. Table records colored orange have unspecified data types or values in the specification.

An XML Schema has been created to represent these object values for parsing. The entire current schema will be presented over the coarse of Figures 40-54. The overall (Setup) schema is shown in Figure 42:

1 1 2 2 3

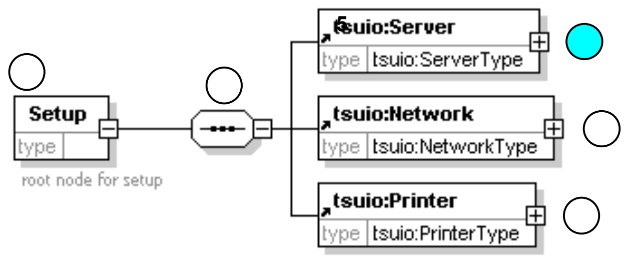


Figure 42: XML Schema setup-specific data.

3

As can be seen, the Setup application (in schema, at **1**) is a sequence (symbol shown at **2**) of the three aforementioned Server (at **3**), Network (at **4**), and Printer (at **5**) components, each of which is required. In XML Schema, the items in a sequence must appear in the order of the schema, so the Server component would always appear first, etc. The ServerType component is further fleshed out to its terminal components in Figure 43:

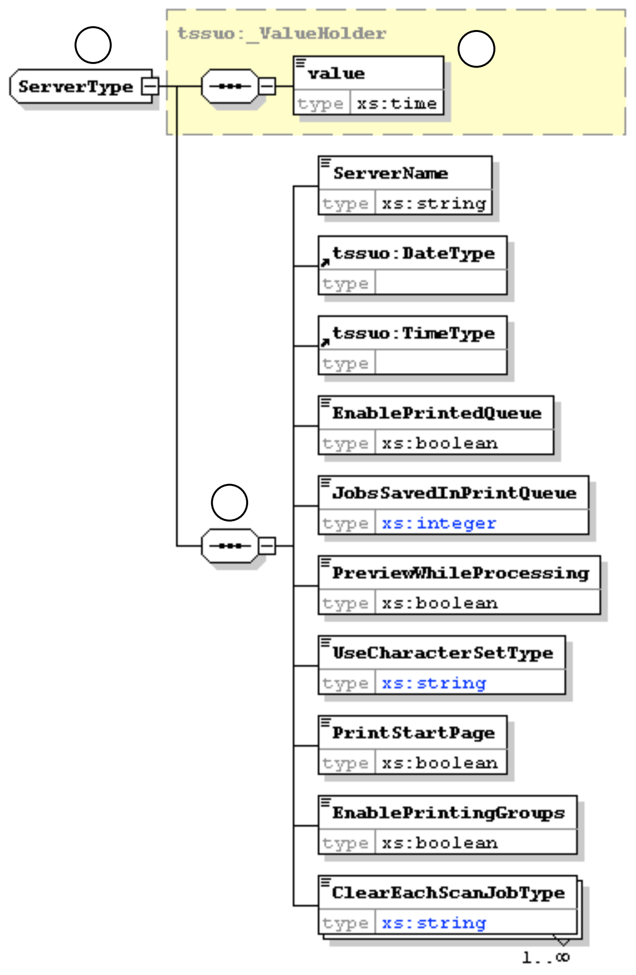


Figure 43: XML Schema setup ServerType component.

1

2

The ServerType component (at **1**) is an XMLTalk 1 ValueHolder component (at **2**) comprising of 10 objects (at **3**) that make up the content of the Server Setup interface. The associated data types, default values, and ranges are embedded in the schema.

2

The ValueHolder, BufferedAspectAdapter, and SelectionInList components are shown in Figure 44:

3 3

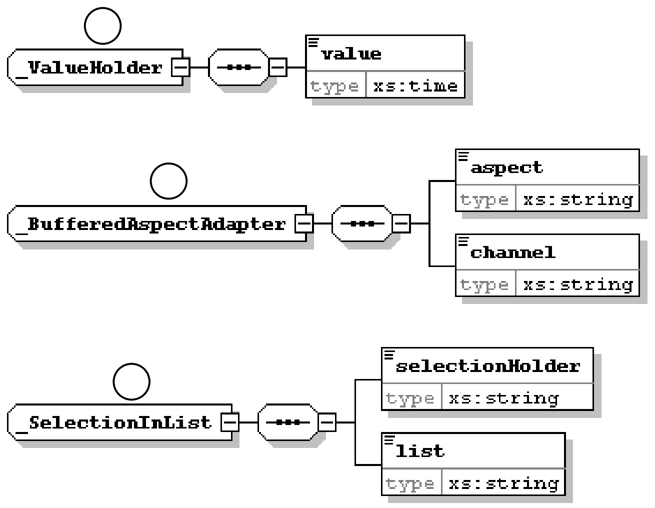


Figure 44: XML Schema for ValueHolder, BufferedAspectAdapter, and SelectionInList components.

These components are used to instantiate XML-based object models. The ValueHolder (at **1**) is a simple object that just a value. The BufferedAspectAdapter is an object that maps to a key/value pair and can be modified (hence the buffer to hold the intermediate value, at **2**). The SelectionInList component represents the selected item and the list of values (at **3**).

Returning to the setup component, the same type of diagram is shown, for the NetworkType component, in Figure 45:

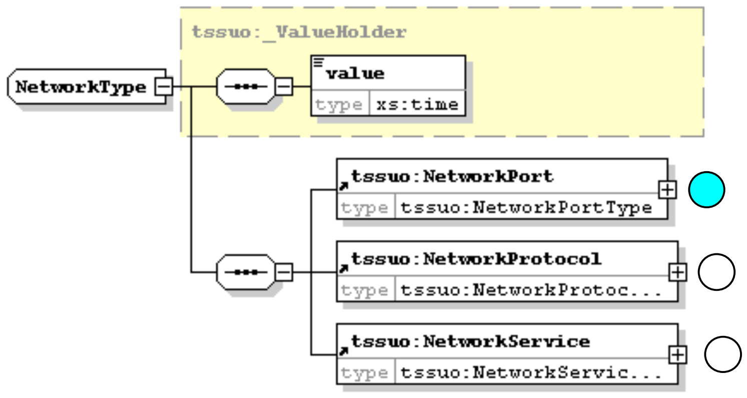


Figure 45: XML Schema for setup NetworkType component.

1

2

1

Notice that the Network component is also a ValueHolder component comprised of three new components: (1) NetworkPort, (2) NetworkProtocol, and (3) NetworkService. The NetworkPortType component is shown in Figure 46:

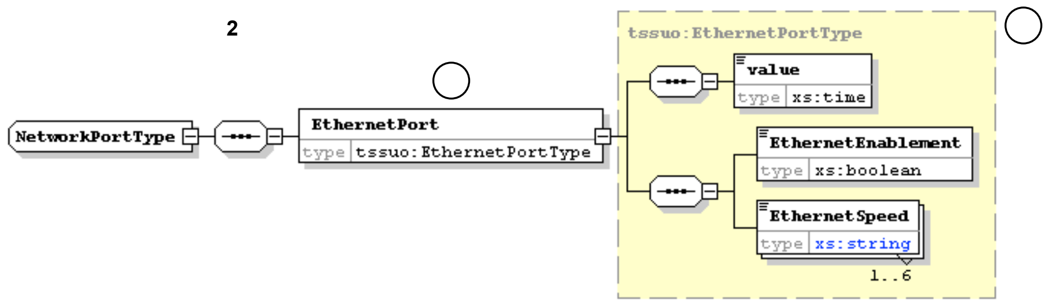


Figure 46: XML Schema for the setup NetworkPortType component.

The NetworkPort has a single component, EthernetPort (at **2**), which is comprised of a value, an EthernetEnablement, and an EthernetSpeed components. The Protocol component is shown in Figure 47:

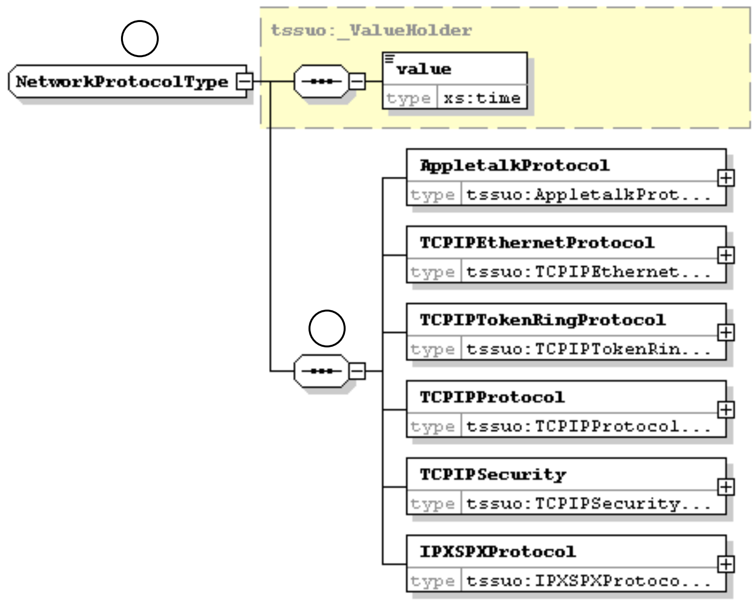

tssuo:_ValueHolder
NetworkProtocolType
value
type xs:time
AppletalkProtocol
type tssuo:AppletalkProt...
TCPIPEthernetProtocol
type tssuo:TCPIPEthernet...
TCPIPTokenRingProtocol
type tssuo:TCPIPTokenRin...
TCPIPProtocol
type tssuo:TCPIPProtocol...
TCPIPSecurity
type tssuo:TCPIPSecurity...
IPXSPXProtocol
type tssuo:IPXSPXProtoco...

Figure 47: XML Schema for the setup NetworkProtocolType component.

The NetworkProtocolType component (at **1**) is a ValueHolder component comprised of 6 components (at **2**) representing the setup information for their respective protocols. The terminal protocol components are illustrated in Figures 46-49:

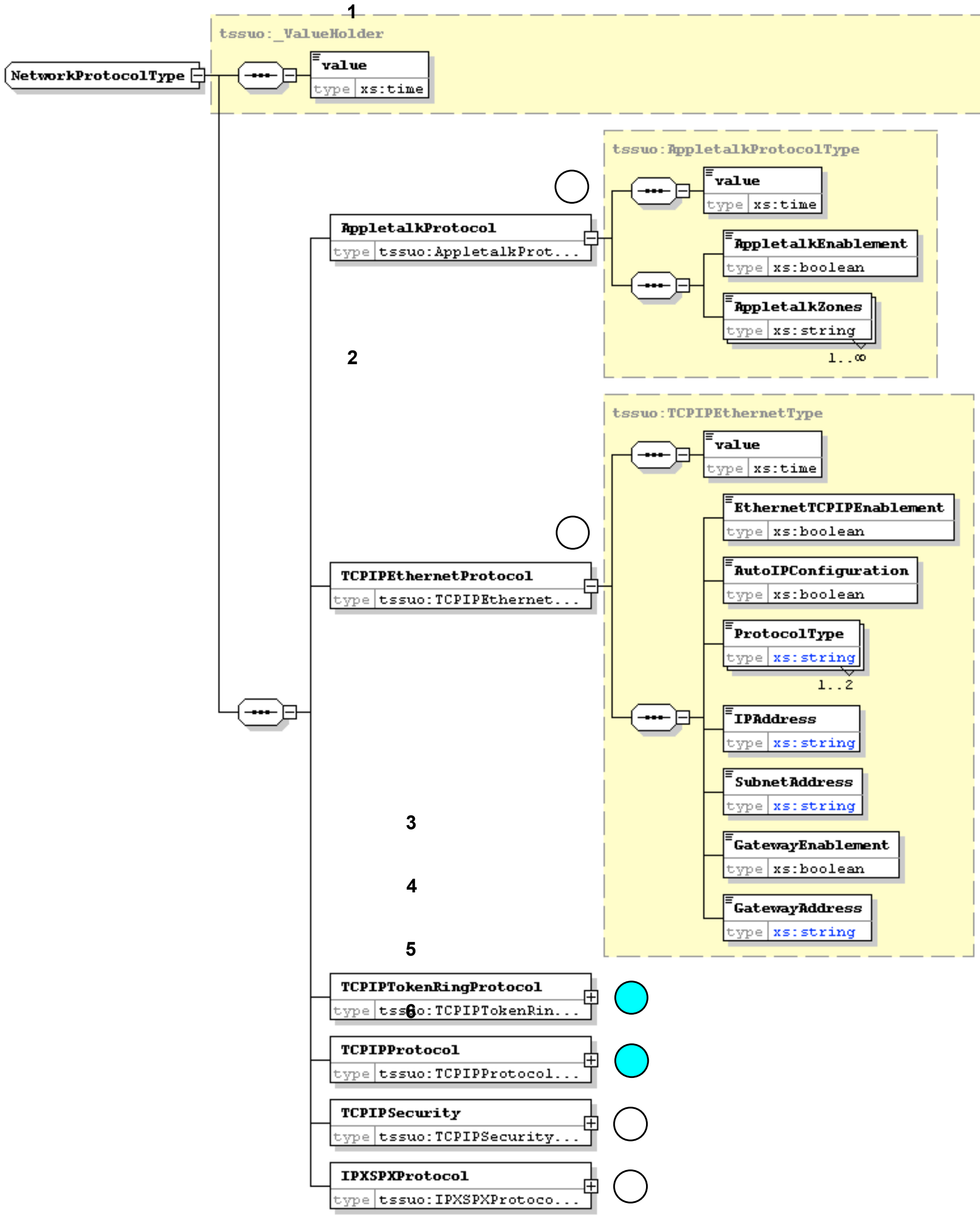


Figure 48: XML Schema setup network protocol components for appletalk and tcpip.

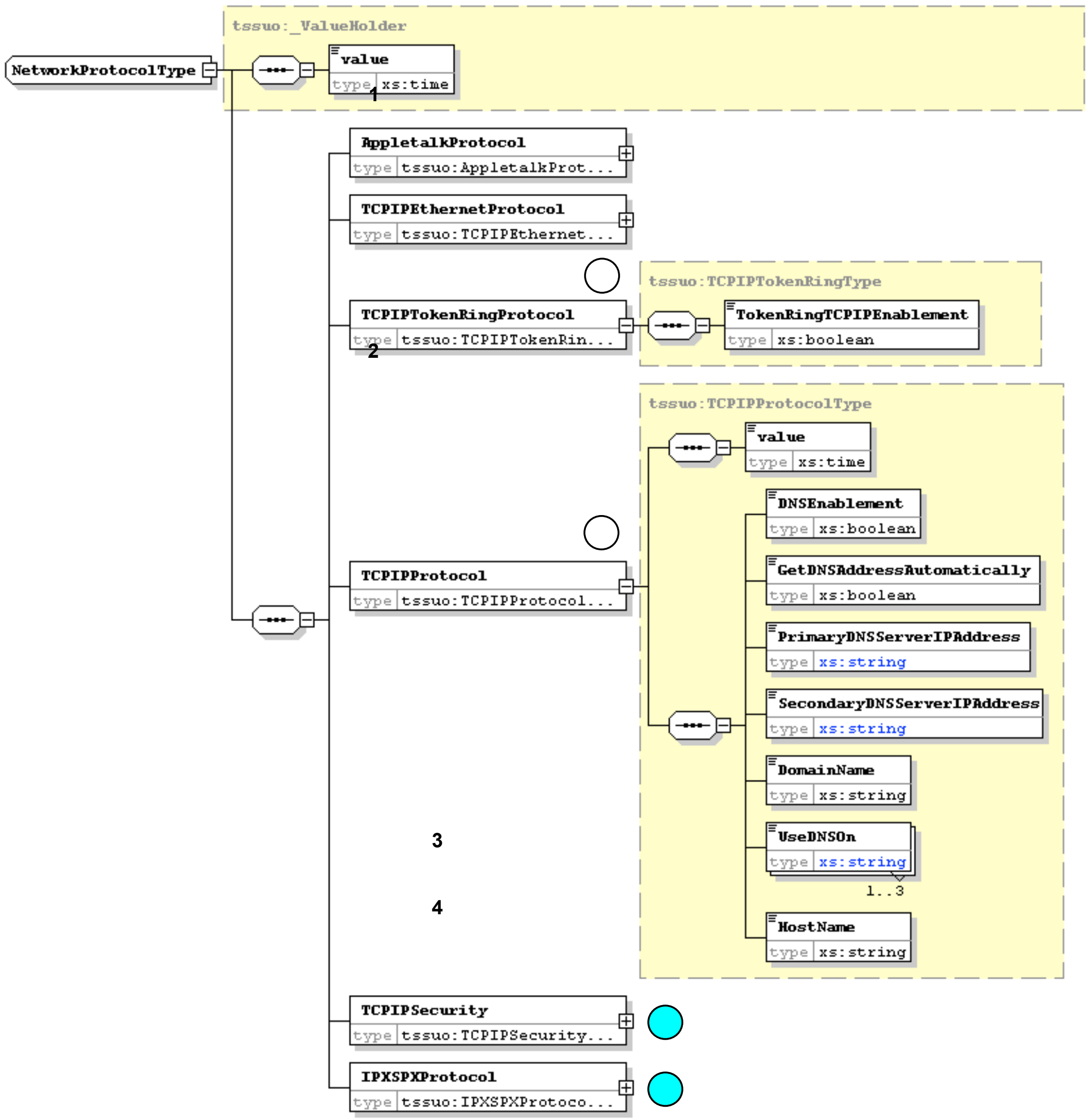


Figure 49: XML Schema setup network for token ring and TCPIP protocol components.

The TCPIP protocol component consists of 7 terminal fields.

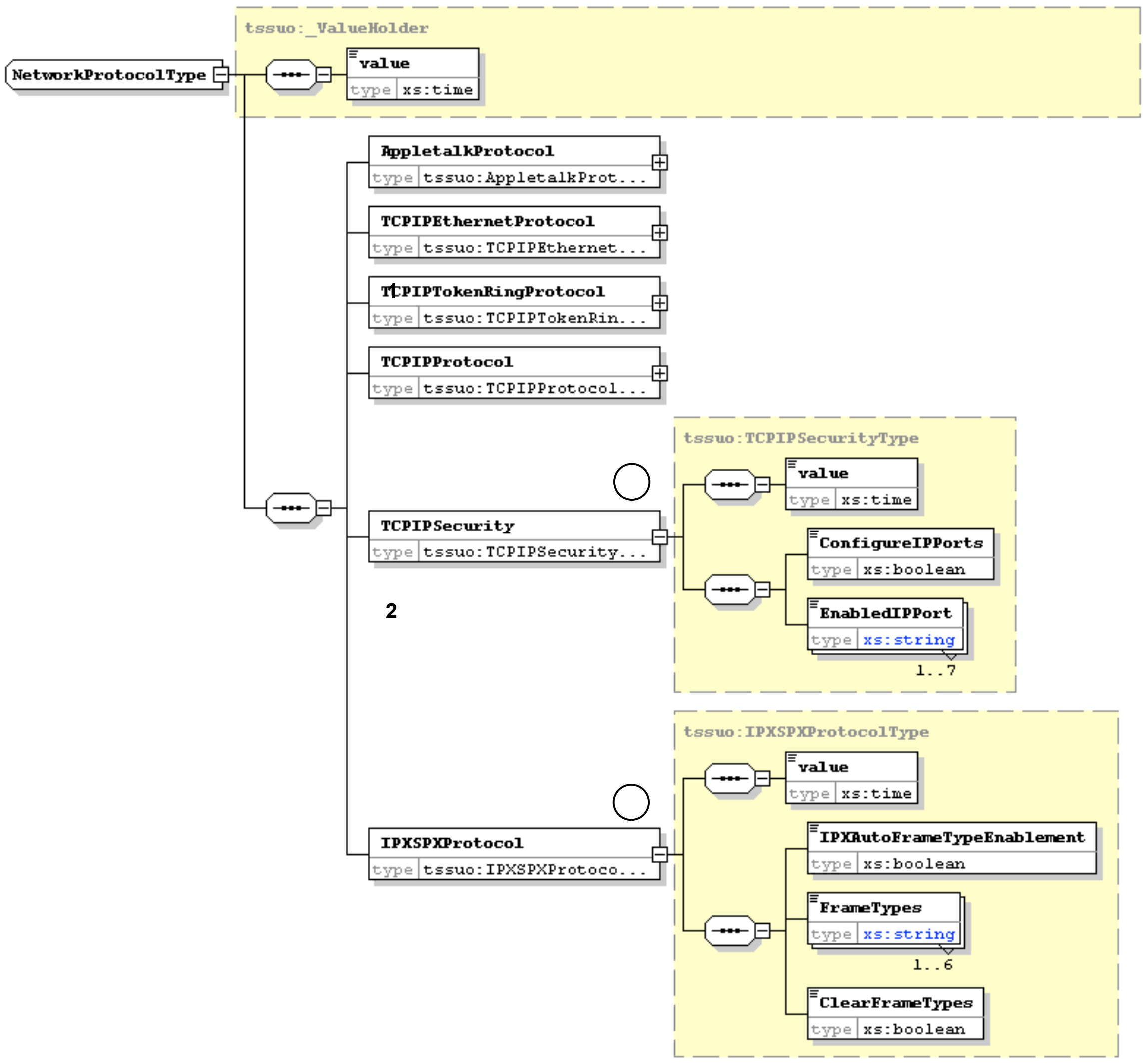


Figure 50: XML Schema setup network TCPIP protocol component.

The TCPIP protocol component is comprised of 7 terminal components.

Returning to the NetworkType component, the NetworkServiceType subcomponent is fleshed out in Figure 51:

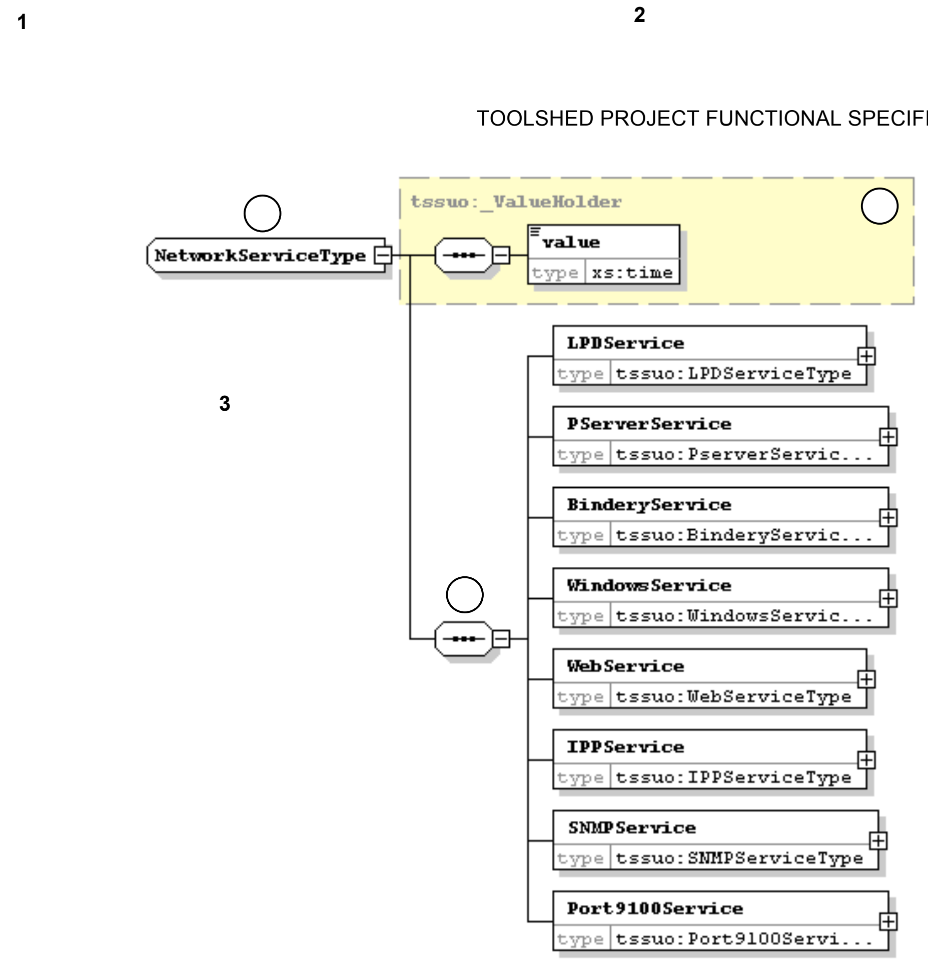

1
2
3
NetworkServiceType
tssuo:_ValueHolder
value
type xs:time
LPDService
type tssuo:LPDServiceType
PServerService
type tssuo:PserverServic...
BinderyService
type tssuo:BinderyServic...
WindowsService
type tssuo:WindowsServic...
WebService
type tssuo:WebServiceType
IPPService
type tssuo:IPPServiceType
SNMPService
type tssuo:SNMPServiceType
Port9100Service
type tssuo:Port9100Servi...

Figure 51: XML Schema setup network Service component.

The NetworkServiceType component (at **1**) is a ValueHolder (at **2**) comprised of 8 components (at **3**) representing the setup information for their respective services. The terminal components for these services are illustrated in Figures 49-52:

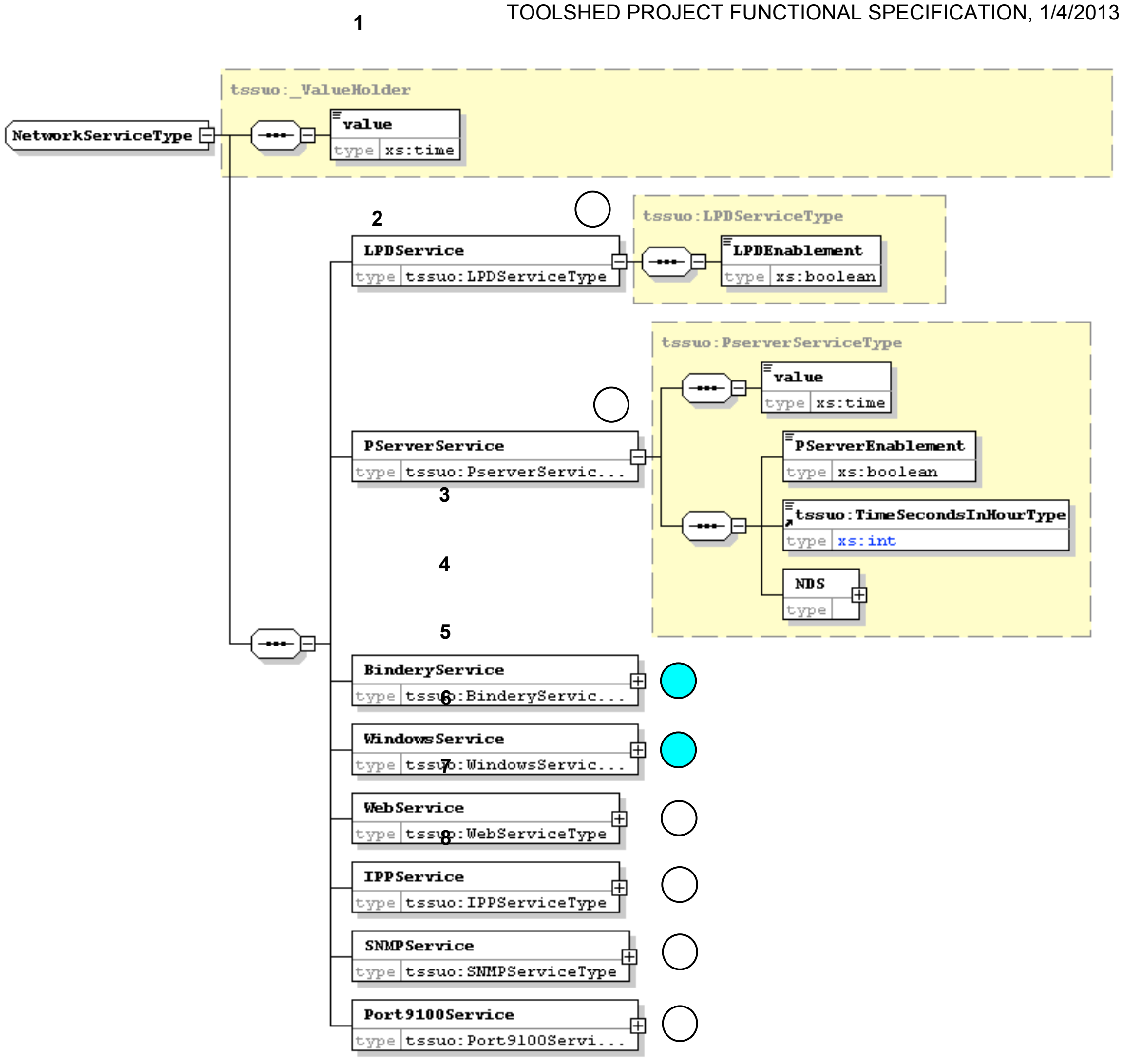


Figure 52: XML Schema setup network service components for lpd and pserver.

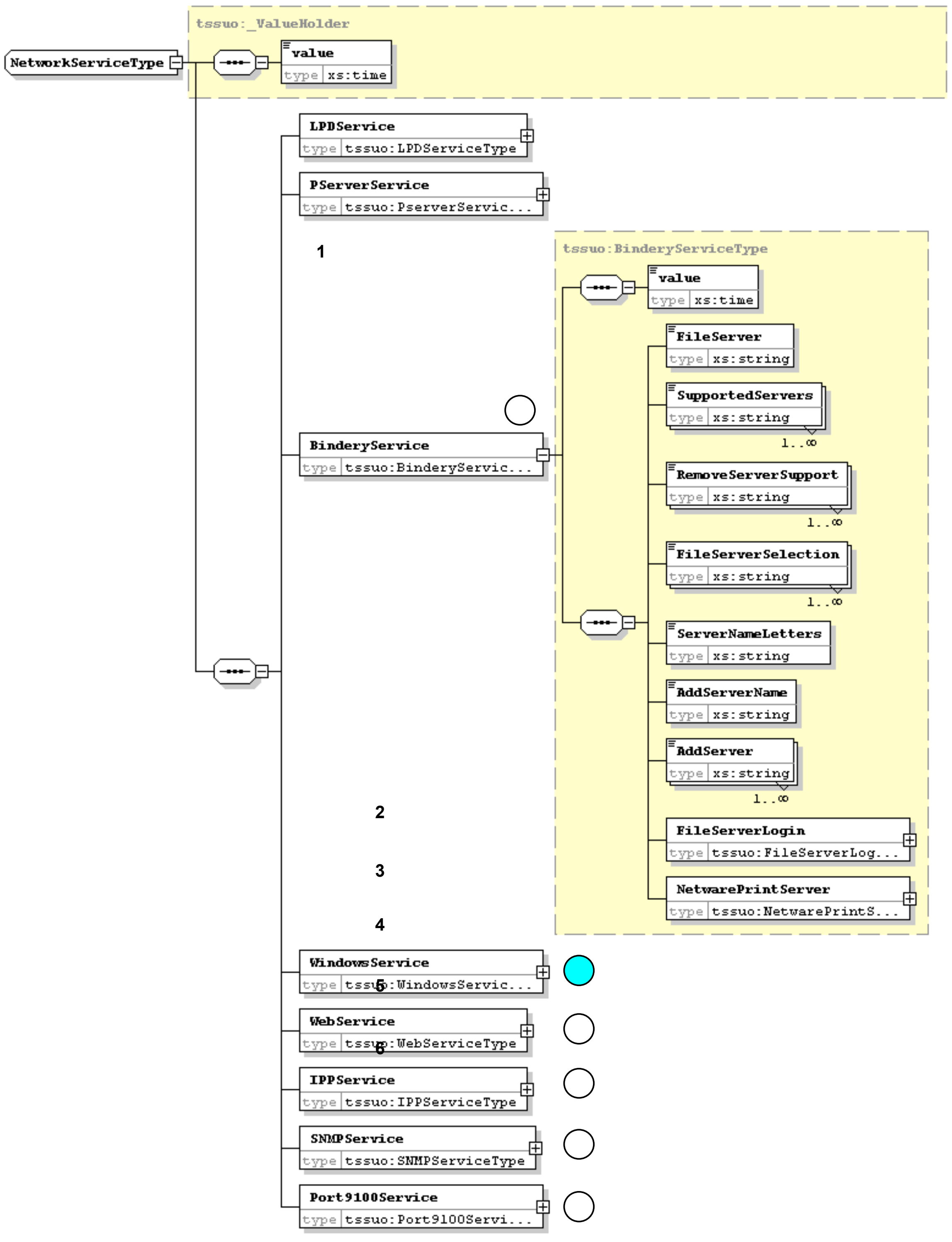


Figure 53: XML Schema setup network bindery service component.

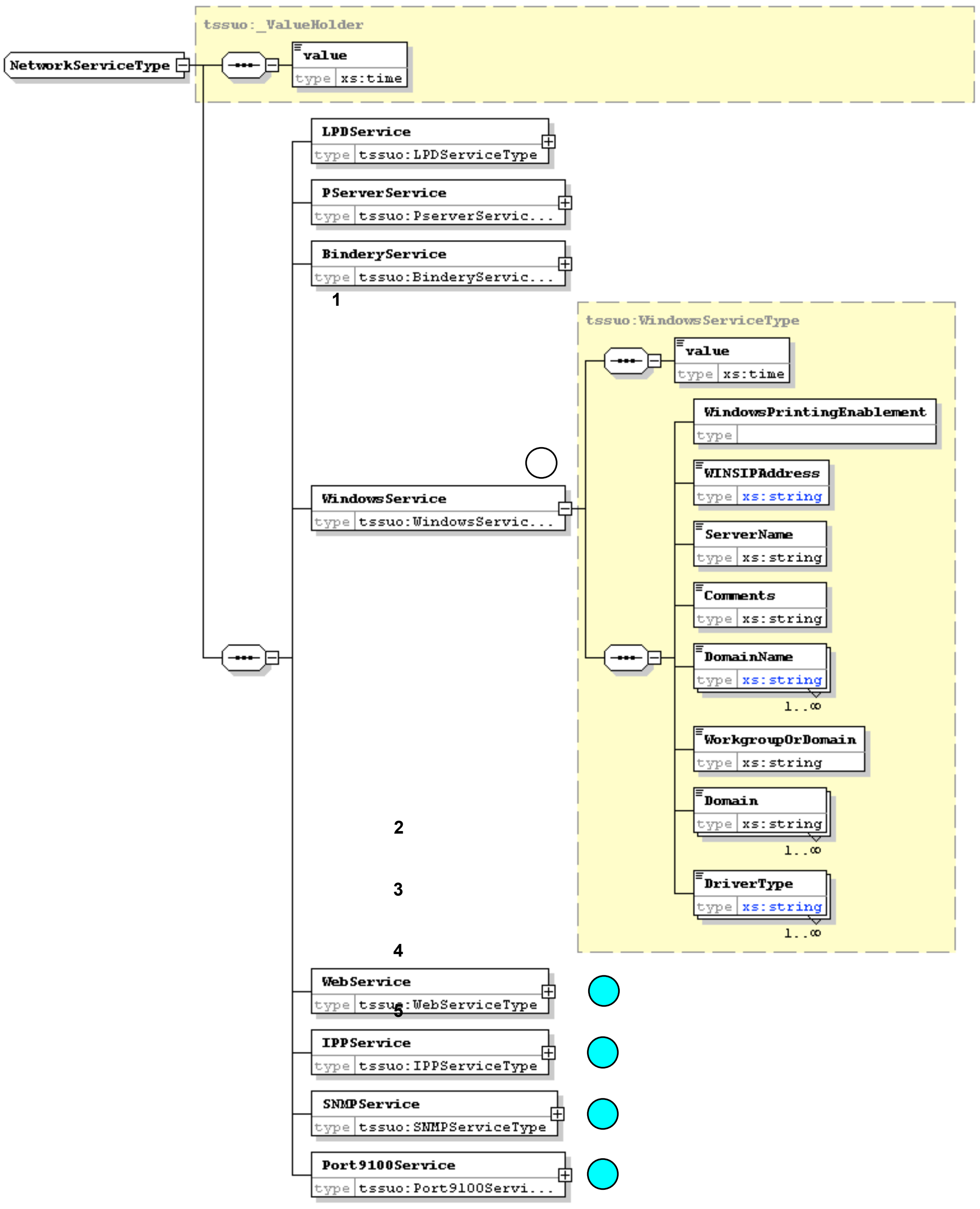


Figure 54: XML Schema setup network windows service component terminals.

There are eight windows service component terminals.

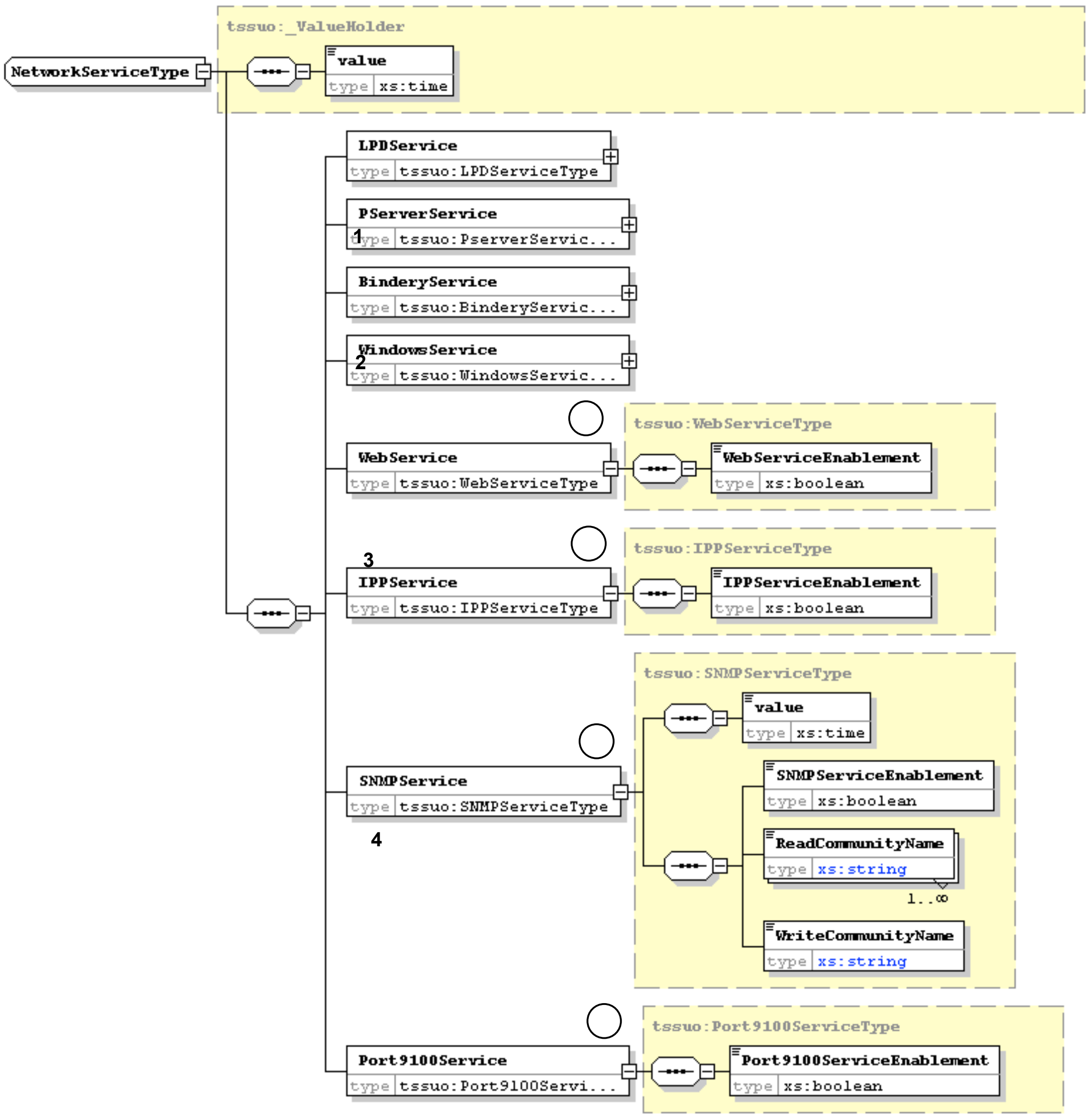


Figure 55: XML Schema setup network service web, IPP, SNMP, and Port9100 components.

The final setup-related component is the PrinterType component, as shown in Figure 56:

1 2

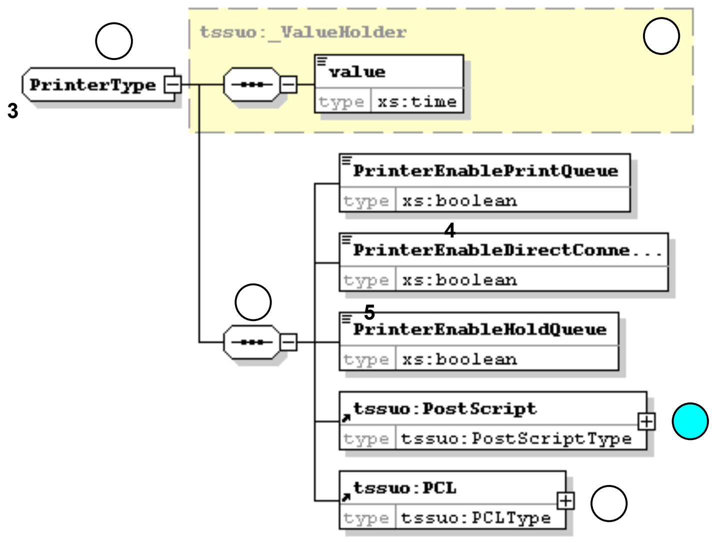


Figure 56: XML Schema for setup PrinterType component

The PrinterType component (at **1**) is a ValueHolder component (at **2**) that consists of five subcomponents (at **3**), three of which enable the type of queue (print, direct, or hold). The PostScript component is fleshed out in Figure 57:

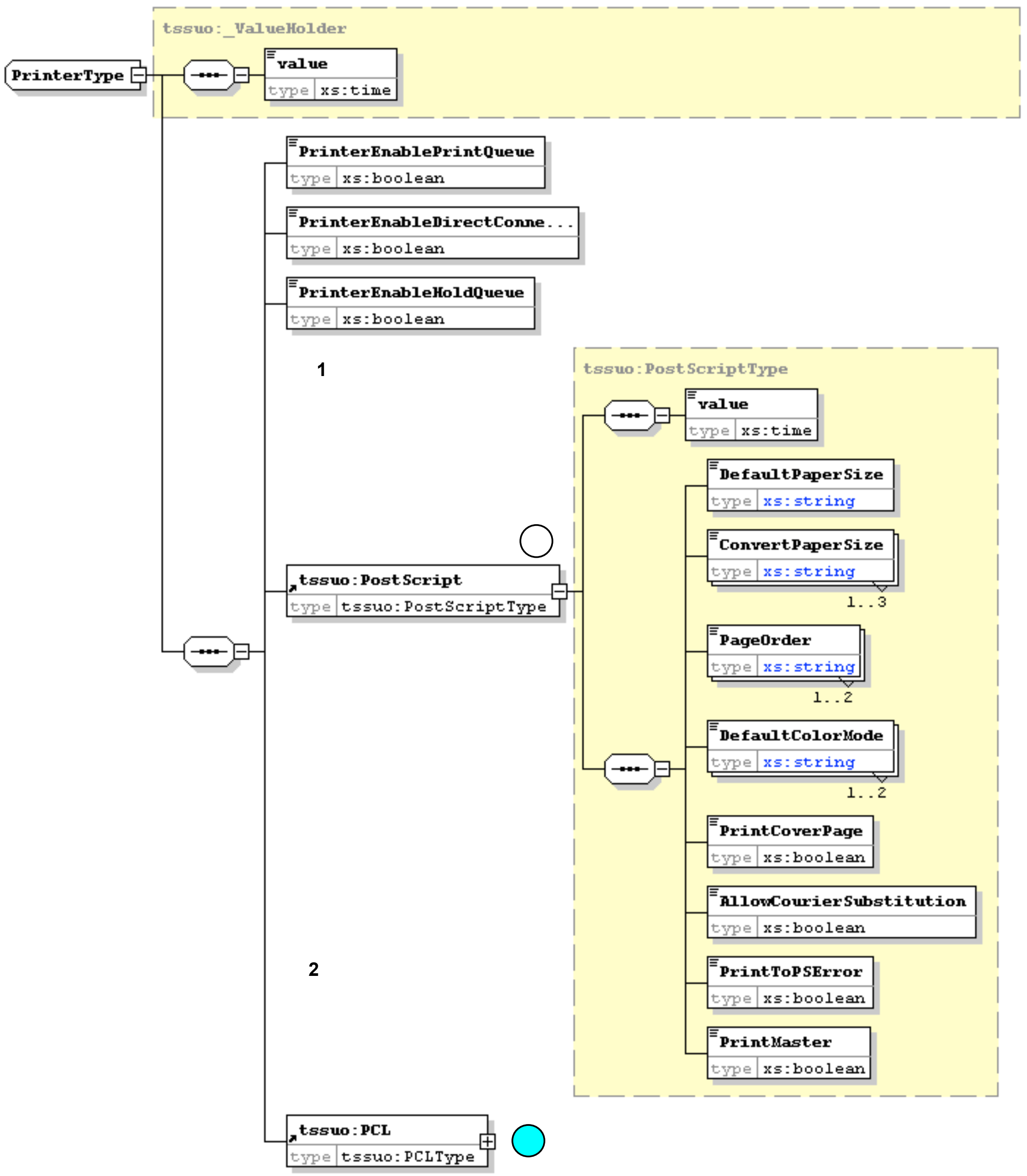


Figure 57: XML Schema for PostScript Printer setup component terminals.

There are 8 terminals specific to the PostScript printer setup. The PCL component terminals are shown in Figure 58:

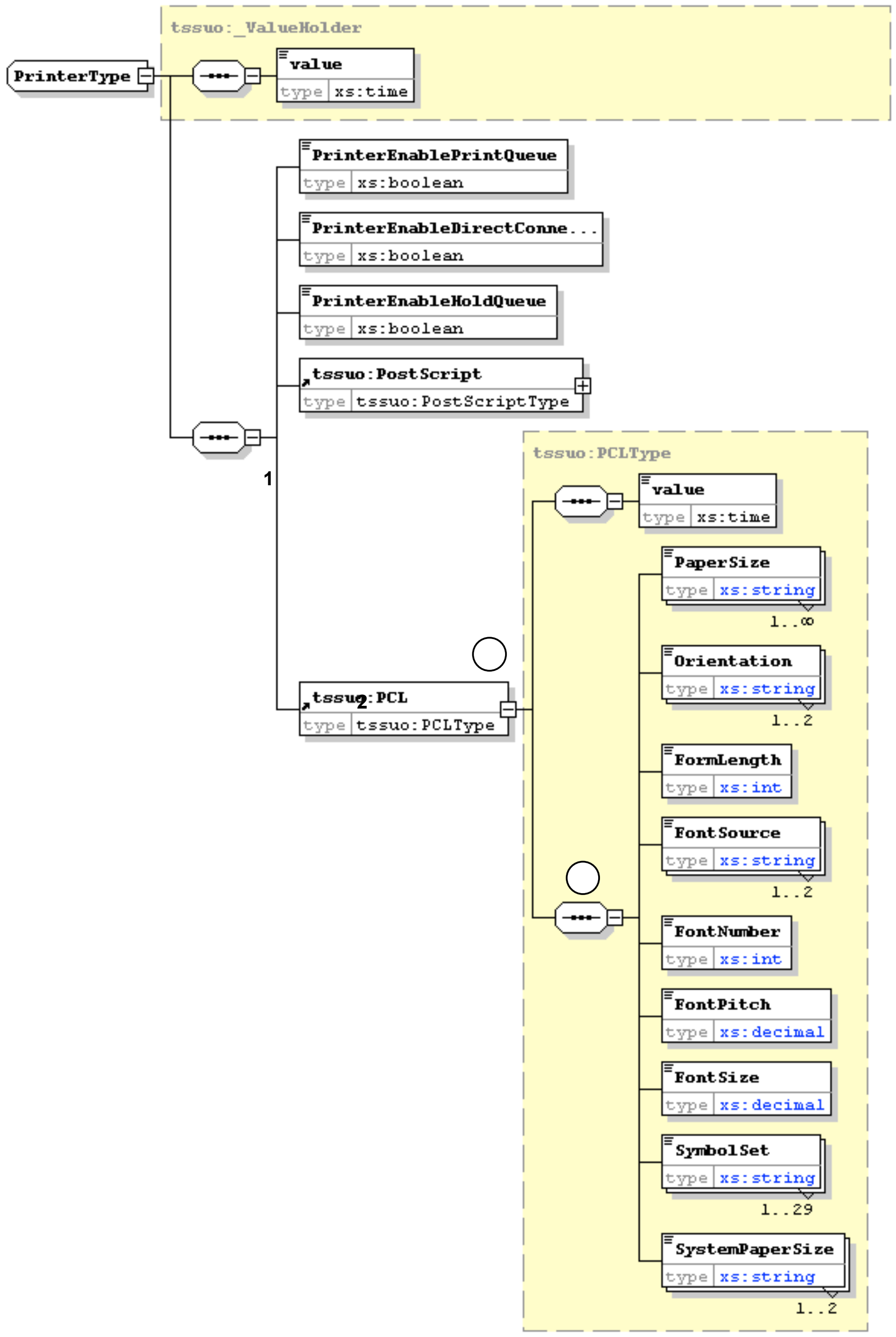


Figure 58: XML Schema for PCL Printer setup component terminals.

There are 9 terminals specific to the PCL printer (at **2**) setup, shown at **3**.

Validation Process

Data type, value, and range is processed at two times during the existence of the ToolShed interface: (1) during initialization, and (2) when server requests are made. The processing mechanism for the former mode is identical to the one for object existence validation:

- Store the data type/value key in the XML file with the real data
- Iterate through the object models
- Retrieve the ValueHolder for the model
- Test the data type against that expected
- Test the value against that expected
- Set data validation to true/false

When actions are performed in the interface that require data requests or updates, a new data/value validation must be performed. The processing mechanism for this mode is similar to the one for initial data validation:

- Iterate through the data fields on the submitted form
- Retrieve the associated Java objects
- Test the data type against that expected
- Test the value against that expected
- Set data validation to true/false
- Associate a dialog string for invalid data
- After user provides a data value, run check as above

The processing mechanism is similar, as mentioned, to the object validation mechanism, but the tests are different. Also, because errors can be made that shouldn't be sent back to the server, dialog functionality is also required.

Constraint Representation and Validation (should be level 4 heading)

In reality, data, like user interfaces, does not exist in a vacuum but, rather, along with other components. That is, there are bound to be dependencies on the data presented in one UI component or page and others. When a user types an IP address into a text field, not only must that IP address be a valid string, and lie within a particular range (xxx.yyy.zzz.www), it may also have to be on a particular subnet. Just as importantly, if a fixed IP address is selected in one portion of the page, other values may have to change on that page or others. For example, if we choose to use Auto IP detection, then we won't need to ask the user to set the DNS server address, since it would be inappropriate. In a user interface where the user can type in the DNS server address, and did so, a conflict would arise.

Inter-object dependencies must be validated, or satisfied, in the same way as object validation or type/value validation. The difference is that there is now a context, and the previous object values that could be evaluated on their own merit can no longer be

2 evaluated in the same way. What *is* required is a constraint validation mechanism and a set of rules for validating constraints when new and conflicting data is identified.

In some cases, 5 constraints can be satisfied by disabling UI components, but components which are text fields would be problematic to constrain in such a way, so an alternate constraint validation mechanism is indicated. This may be referred to as Type 3 validation.

Constraint Representation with RuleML

3 One mechanism being developed to address this issue is the Rule Markup Language (or RuleML). This is an XML-based language for representing semantic constraints of arbitrary complexity. Using a language such as RuleML, inter-object constraints can be defined in a constraints file, parsed into rules during application initialization, and then the rules can be cycled through as a event-handling mechanism to validate the constraints prior to rendering a new page or, more importantly, submitting a server request. Cycling through the rules requires a rule engine such as Jess (or Mandarax) and a means to translate the XML-based RuleML rules into the format required by the rule engine. The RuleML structure will be presented first, followed by the translation into a Java-based rule syntax and then the processing mechanism whereby rules are applied at runtime.

RuleML Structure and Capabilities

The general structure of RuleML (version 0.8) is shown in Figure 59:

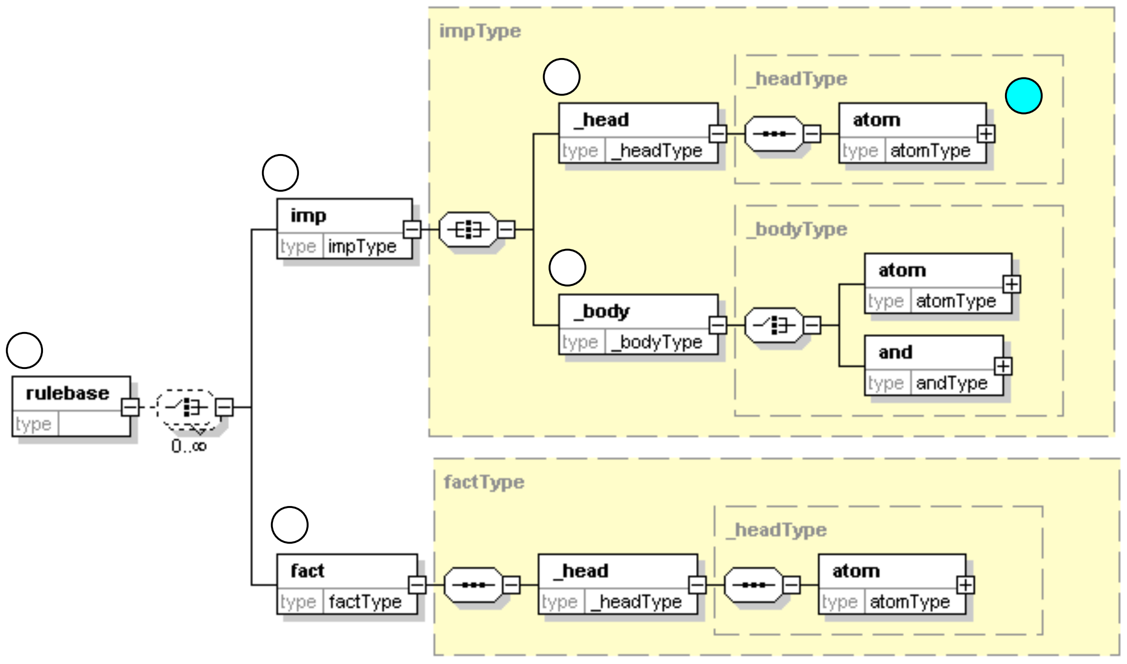


Figure 59: RuleML rulebase XML Schema structure.

A rulebase is a collection of rules which are cycled through and examined. The rulebase element (at **1**) is defined as an (unbounded) collection of implications (imp, at **2**), and facts

3

2

(at **3**). An implication is defined as a head (at **4**) and a body (at **5**), where the head defines a test condition and a body defines a conclusion to be made (or an action to be performed) if the condition is true. As can be seen in the figure, both head and body are decomposed into conjunctive terms and atoms, which are themselves shown in Figure 60:

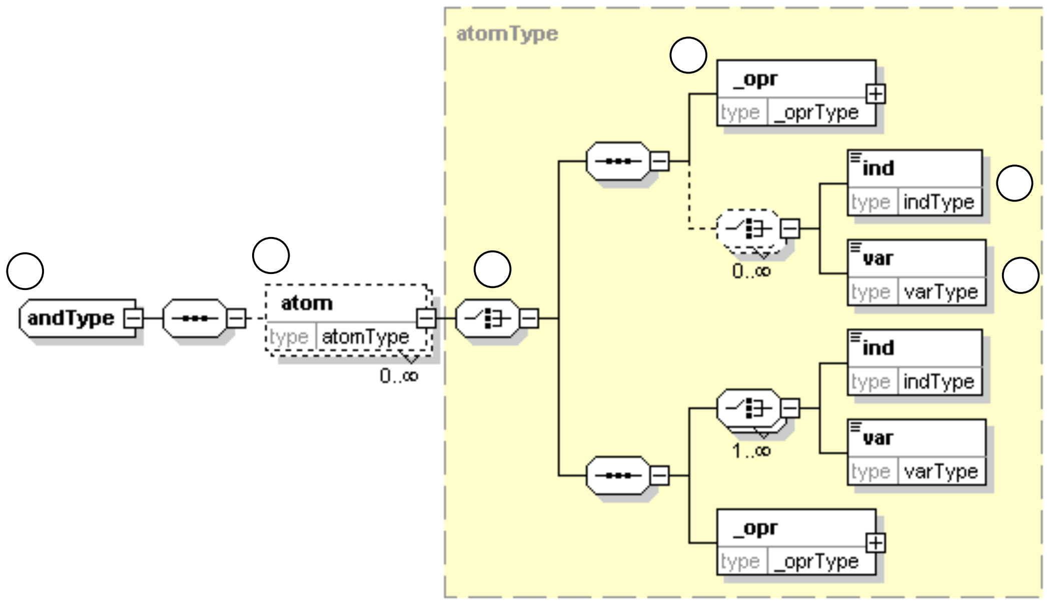


Figure 60: RuleML and and atom schema definitions.

This definition states that a conjunction (at **1**) is a sequence of (possibly) many atoms (at **2**), and that an atom can be an operator (at **3**), (possibly) followed by an identifier (at **4**) or variable (at **5**), or an identifier or variable followed by an operator. The 0..oo notation means an unbounded number of items, and the switch notation (at **6**) denotes a choice between the items the follow it. Dotted lines are placed (e.g., at **2**) are placed around any element that can have 0 elements. The definition supports the representation of infix or postfix notations.

RuleML Representation

An example showing how RuleML is used to represent constraints, for a single rule, is illustrated in Figure 61:

```
<Rulebase ID="sweetjessRuleBase">
  <direction>bidirectional</direction>
  <imp>
    <Imp about="#imp1"/>
  </imp>
  <fact>
    <Fact about="#fact1"/>
  </fact>
</Rulebase>

<Imp ID="imp1">
  <_head>
    <_Head about="#head1"/>
  </_head>
  <_body>
    <_Body about="#body1"/>
  </_body>
</Imp>

<_Head ID="head1">
  <atom>
    <Atom about="#atom1"/>
  </atom>
</_Head>

<Atom ID="atom1">
  <_opr>
    <_Opr about="#opr1"/>
  </_opr>
  <ind>
    <Ind about="#ind1"/>
  </ind>
  <var>
    <Var about="#var1"/>
  </var>
</Atom>

<_Opr ID="opr1">
  <rel>
    <Rel about="#rel1"/>
  </rel>
</_Opr>

<Rel ID="rel1">
  <value>giveDiscount</value>
</Rel>

<Ind ID="ind1">
  <value>percent10</value>
  <position>2</position>
</Ind>
```

```
<Var ID="var1">
  <value>customer</value>
  <position>1</position>
</Var>

<_Body ID="body1">
  <atom>
    <Atom about="#atom2"/>
  </atom>
</_Body>

<Atom ID="atom2">
  <_opr>
    <_Opr about="#opr2"/>
  </_opr>
  <var>
    <Var about="#var2"/>
  </var>
</Atom>

<_Opr ID="opr2">
  <rel>
    <Rel about="#rel2"/>
  </rel>
</_Opr>

<Rel ID="rel2">
  <value>premium</value>
</Rel>

<Var ID="var2">
  <value>customer</value>
  <position>1</position>
</Var>

<Fact ID="fact1">
  <_head>
    <_Head
about="#head2"/>
  </_head>
</Fact>

<_Head ID="head2">
  <atom>
    <Atom about="#atom3"/>
  </atom>
</_Head>
```

```
<Atom ID="atom3">
  <_opr>
    <_Opr about="#opr3"/>
  </_opr>
  <ind>
    <Ind about="#ind2"/>
  </ind>
</Atom>

<Ind ID="ind2">
  <value>Allan</value>
  <position>1</position>
</Ind>

<_Opr ID="opr3">
  <rel>
    <Rel about="#rel3"/>
  </rel>
</_Opr>

<Rel ID="rel3">
  <value>premium</value>
</Rel>
```

Figure 61: A rule/constraint represented with RuleML.

The namespace references have been removed from the preceding figure to collapse the size a bit. This is a pretty obscure way of looking at the rule. The DOM tree this rule represents is illustrated below in Figure 62:

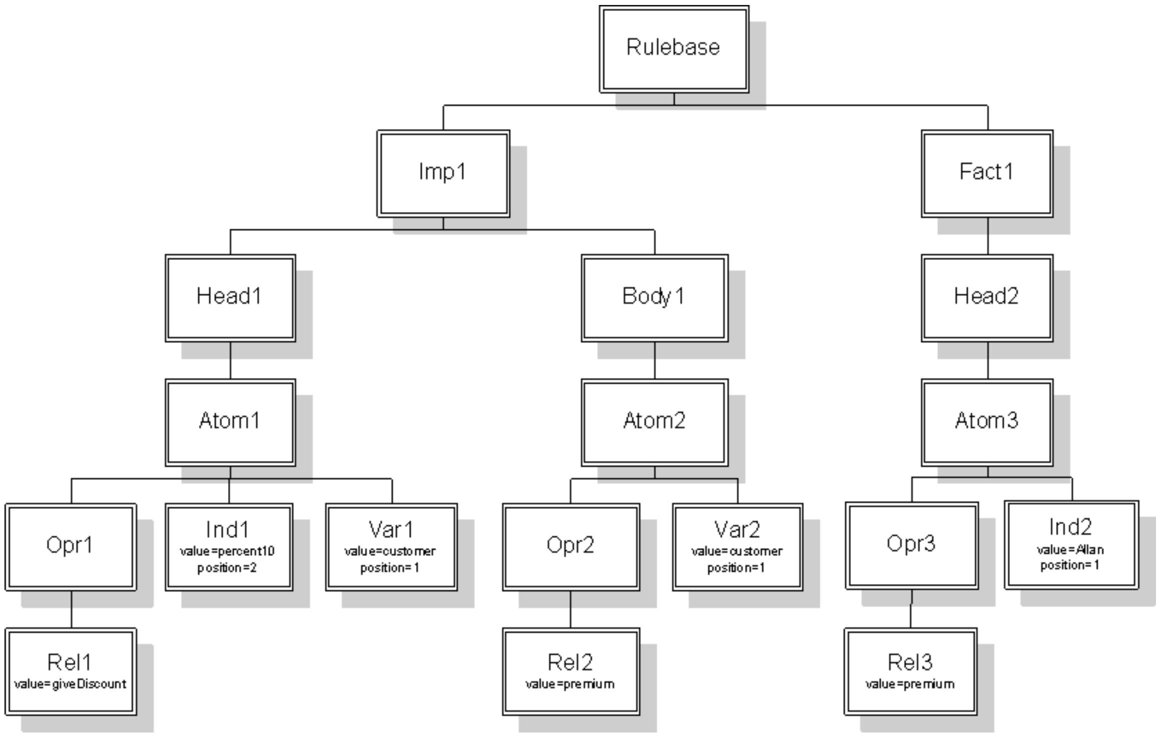


Figure 62: Tree representation of sample rule.

RuleML Parsing to Java Rules

RuleML forms cannot be processed directly, so they must first be parsed into Java so that the Java-based rule engine can interpret them. Since the rules do not change with time and can be loaded at launch time, and since ToolShed doesn't have a large number of business rules to begin with, this extra processing shouldn't force a noticeable performance loss. Mandarax parses directly from RuleML into its knowledge base of rules. The parsing model for RuleML in Mandarax is shown in

Figure 63: Mandarax RuleML parsing model.

Rule-Based Processing

Once rules have been parsed from RuleML or some suitable format into the rule engine's knowledge base, the rules can be processed. The Mandarax project utilizes a backward-chaining (BC) algorithm to perform its analysis because it is a deductive model. That is, it is trying to prove that a conclusion is correct, and BC is a good approach for that. A forward-chaining (FC) algorithm, which is also deductive, can also be used to identify rule conflicts. As this is the primary intent of rule-based processing in ToolShed (i.e., constraint validation as opposed to predictive modeling), a forward-chaining algorithm is more appropriate in ToolShed. In both approaches, rules are represented with the same condition/action pairs but the way conflict sets are constructed and analyzed differs.

In the FC approach, the algorithm starts from a set of known facts, and matches those to the condition parts of available rules. Those rules that satisfy the current knowledge/fact base are collected, sorted, and executed in order, leading to predictions.

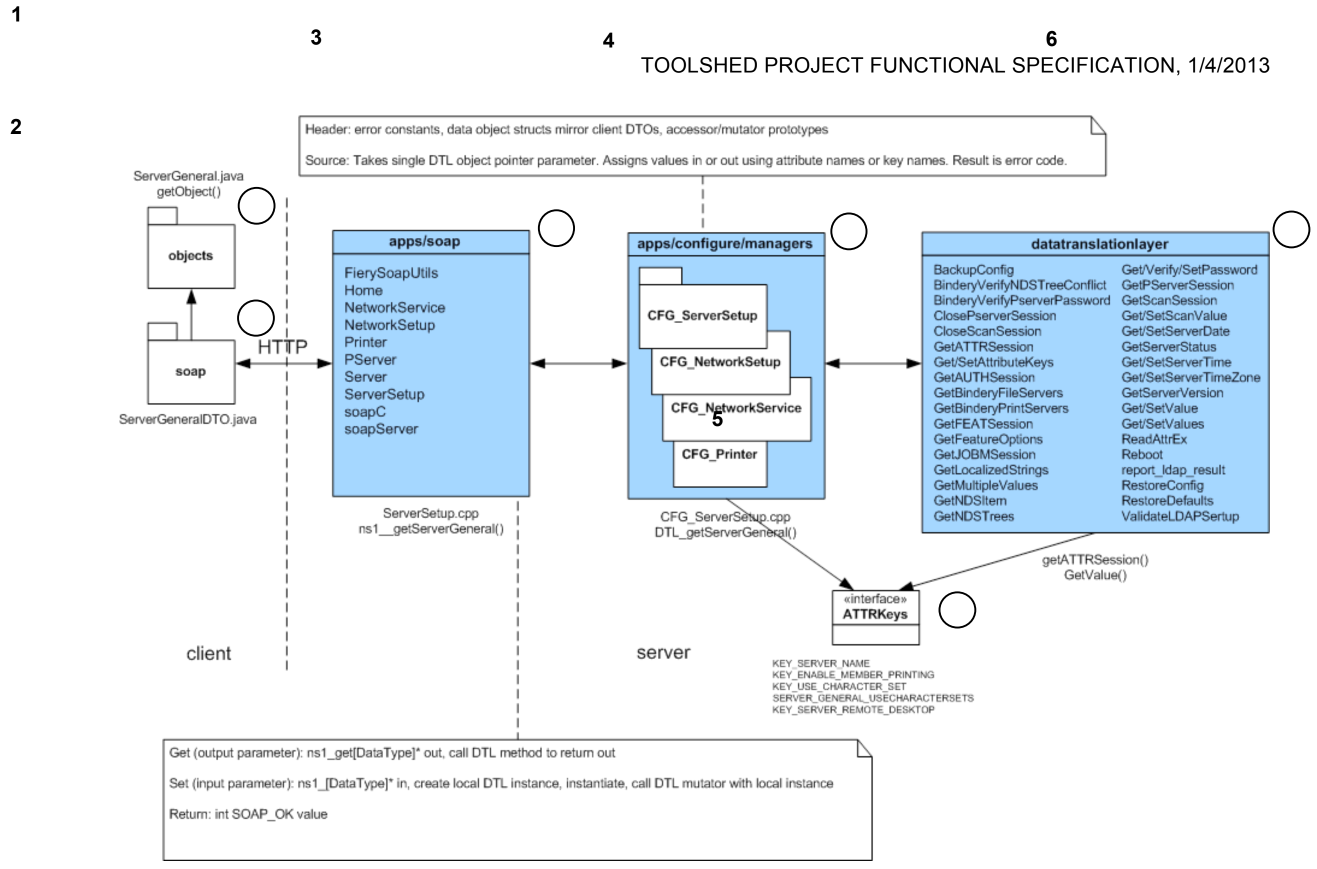
Constraint Validation

In ToolShed, what is sought is a situation where the set of rules that describe a condition are violated by the current/available user-supplied information. The available information is comprised of the values in the various Java models associated with the page contents. That is, the combined set of conditions *should* evaluate to true but doesn't. In this respect, the complexity of rule-based processing is really not needed in ToolShed, because a predictive model isn't currently part of the design requirements. A predictive model *could* be employed, e.g. to fill in values on one page once values on another page are selected, but that is a separate matter for discussion.

Once violated rules are identified (i.e., a special form of exception), associated with them should be a (localized) display string that can be rendered in a dialog.

Server Communications

ToolShed Fiery-based applications communicate to the server using SOAP. There is a client side layer and two layers on the serverside: a SOAP layer and a data translation layer. The SOAP server provides maximum transparency to applications by providing a consistent data and method API. The data translation layer interacts with a number of Harmony APIs (e.g., ATTR, SYSDICT, SCAN). This approach provides the most stable, supported, and compatible communications mechanism but also provides transparency for other applications that might want to communicate with the server. The overall architecture is shown in Figure 64:



Header: error constants, data object structs mirror client DTOs, accessor/mutator prototypes

Source: Takes single DTL object pointer parameter. Assigns values in or out using attribute names or key names. Result is error code.

Get (output parameter): ns1_get[DataType]* out, call DTL method to return out

Set (input parameter): ns1_[DataType]* in, create local DTL instance, instantiate, call DTL mutator with local instance

Return: int SOAP_OK value

Figure 64: ToolShed communications architecture.

The figure above illustrates the communications flow between the Java SOAP client side (items **1** and **2**) and the C++ SOAP server and data translation layer (items **3**-**6**). The retrieval and assignment of values in UI pages is handled by SOAP client object classes (in this case implemented in Java, at **1**). Each class represents the semantic content of a page and implements both a getObject() and setObject() method. In the figure above this is illustrated with the ServerGeneral object (at **1**). The getObject method invokes the call() SOAP method using a data transfer object or DTO. In the figure above this is represented for ServerGeneral by ServerGeneralDTO (at **2**).

The DTO class is a data-only object class. The DTO has a paired class on the SOAP server. These two classes must match in number, type, and order or the serialization across HTTP cannot work properly[2]. The SOAP server object class for ServerGeneral is ServerSetup.cpp. The method paired to getObject is ns1__getServerGeneral. There are commensurate SOAP server classes paired for each SOAP client class (at **3**). It is the SOAP server class that constructs the intermediate object that the data translation layer uses (at **4**) and returns field mask values and return objects based on the success of the call.

The data translation layer performs two duties. First it interacts directly with Harmony APIs which are key/value based, thus providing a direct way to test the Harmony API. Second, it aggregates the key/value pairs into the semantic objects used by the SOAP server to interact with the SOAP client objects. An example is illustrated by DTL_getServerGeneral() in CFG_ServerSetup.cpp (at **4**). This method is called by the

[2] This is an artifact of gSOAP. In Axis this isn't necessary because a mapping between object types and methods for client/server components would be made explicit in the deployment descriptor.

SOAP server ns1__getServerGeneral() method and returns an error code and an object if successful. Inside it makes individual calls to the data translations layer for specific keys and aggregates key-based errors into a fields mask that represents all of the errors for this call.

An individual call to the data translation layer will require opening or accessing a session object for ore or more Harmony APIs and then making one or more calls, such as GetValue or SetValue along with the associated keys (at **6**). The key names are mapped between data translation layer keys and Harmony keys (at **5**) so that the data translation layer can remain transparent to the Fiery server naming conventions while remaining synchronized to the names themselves.

ASIDE: What is SOAP, and Why Use It?

SOAP is a combination of XML representation and a network communications protocol. That is, RPC capability with XML encoding. The most popular communications protocol used for SOAP is HTTP, but it can use any networking protocol. With respect to presentation tiers, SOAP replaces the need for a server to support a client interface, because it enables a different server (or client) to make secure method calls on its data objects. As a result, communications is limited to data transfer, which limits the size of the pipe that has to be opened up between the client and server. The client can also then generate whatever interface it chooses.

The transport protocols used for communications can be Java Remote Method Invocation (RMI), Microsoft Distributed Component Object Model (DCOM), and SOAP/XML. RMI and DCOM are more tightly coupled than SOAP/XML and generally provide faster information translation because they transfer binary data as opposed to formatted text. If very tight coupling between applications isn't required, SOAP/XML has many advantages over either RMI or DCOM:

- **Simplicity**: SOAP/XML utilizes simple text to describe the transfer of data, so it may be easily understood.
- **Platform Agnostic:** SOAP/XML uses simple text, so any platform that supports the underlying protocol to SOAP/XML may send or receive XML messages. This includes 8-bit and 16-bit embedded systems.
- **Protocol Support:** SOAP/XML is most commonly run over HyperText Transfer Protocol (HTTP). This allows it to more effectively pass through firewalls. It can, however, be run over TCP, FTP and a variety of other protocols.
- **Language Independent:** SOAP/XML is text-based, and therefore generally any language supporting text manipulation may be used to support SOAP/XML.
- **Integration with PC and Enterprise Systems:** Most PC and Enterprise systems today are capable of communication using SOAP/XML. For this reason, device integration with these systems is quite feasible using SOAP/XML.

SOAP, being implemented using XML and XML Schema, is architecture, operating system, and language independent. The SOAP server mediates communications between servers by translating the requesting server XML method call into whatever procedural call is defined to satisfy that interface on the requested server side, as shown in Figure 65:

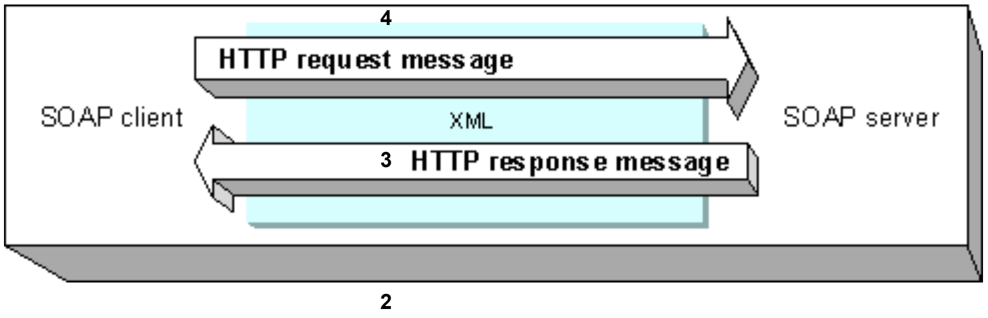


Figure 65: SOAP architecture for HTTP networking/rpc protocol.

The request and response are both mediated, in the ToolShed case, by HTTP. All SOAP toolkits provide a proxy component that parses and interprets the SOAP message to invoke application code. The proxy understands encoding styles, translation of native data types to/from XML, etc. The proxy performs the following three tasks:

- Deserialize the message from XML into some native format suitable for passing off to the code.
- Invoke the code.
- Serialize the response back into XML and hand back to the transport listener for delivery back to the requestor or forwarding to the next client.

The translation interface, i.e., the component that tells the proxy which code to invoke, is called (for Apache SOAP) a Deployment Descriptor and can be implemented with the Web Services Description Language (WSDL) to enable dynamic discovery of the Web Services capabilities and to ease/organize maintenance of the capabilities. The networking layers associated with SOAP are depicted in Figure 66:

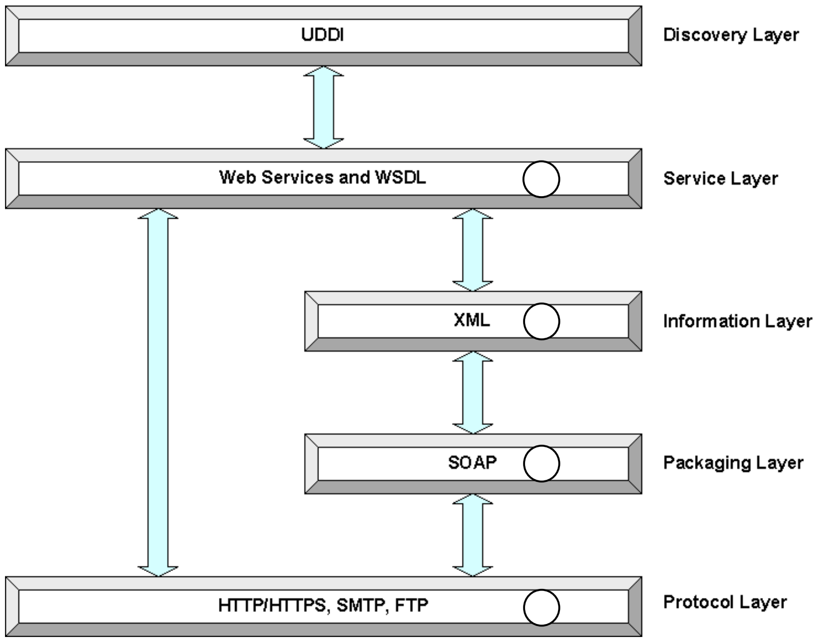


Figure 66: Networking components associated with SOAP.

Assuming that universal discovery isn't important (i.e., the schemas are used only by EFI), the first 4 layers of Figure 57 are what will be used in ToolShed. HTTP/HTTPS (at **1**) provides the transport protocol, SOAP provides for message parsing and creation (at **2**), XML provides for the information representation and content (at **3**), and WSDL provides the language definition and translation template (at **4**).

SOAP in ToolShed

An illustrative example of SOAP usage in ToolShed is the sequence associated with WebSetup => Network => Protocol => TCP/IP Ethernet data. There are 3 components that must be implemented in the ToolShed communications path between the toolkit and the server data: (1) server side code representing the C++ classes constructed from the server data, (2) the deployment descriptor which maps from the server side code to the client-side model, and (3) the ToolShed client code associated with component events.

gSOAP Server-Side Implementation

ToolShed will have its own SOAP server, which will be architected as shown in Figure 67:

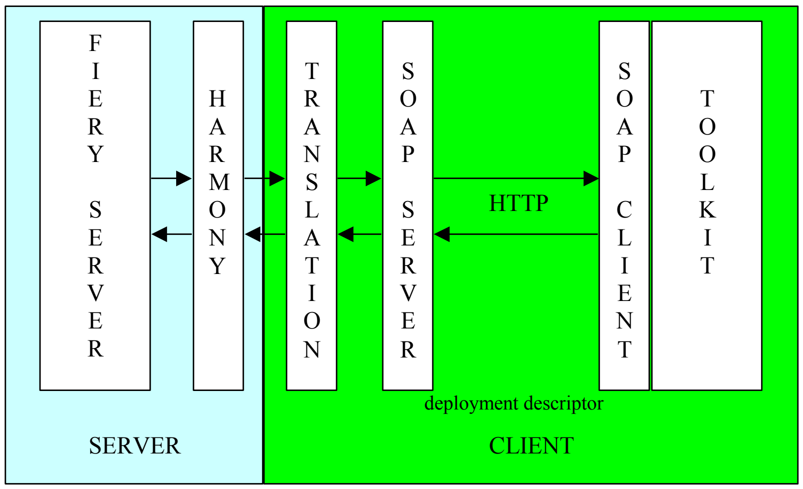


Figure 67: ToolShed SOAP server architecture.

The server side Java for this example is shown in Figure 68:

1

```
int setup__getAvailableSpeeds(struct soap *soap,  void *in,
                              ArrayOfSpeeds *out)
{
    out->__ptr    = (char**) soap_malloc(soap, 2);
    out->__size   = 2;
    out->__ptr[0] = (char*) soap_malloc(soap, 1024);

    strcpy(out->__ptr[0], "this");
    out->__ptr[1] = (char*) soap_malloc(soap, 1024);
    strcpy(out->__ptr[1], "that");
    return(SOAP_OK);
}
```

Figure 68: Server side C++ for hello world SOAP example.

All that we provide is the normal interface in terms of the method class and name (at **1**).

SOAP Deployment Descriptor

The SOAP server will encode the parameters for parsing, and the Deployment Descriptor provides the mapping, as shown in Figure 69:

1

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="setup"
 xmlns="http://schemas.xmlsoap.org/wsdl/"                    2
 xmlns:SOAP="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:WSDL="http://schemas.xmlsoap.org/wsdl/"
 targetNamespace="http://localhost/soap/setup.wsdl"          3
 xmlns:tns="http://localhost/soap/setup.wsdl"
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"                4
 xmlns:setup="urn:setup">

  <types>
    <schema targetNamespace="urn:setup"
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:setup="urn:setup"                                5
      xmlns="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="unqualified"
      attributeFormDefault="unqualified">

      <complexType name="empty">
        <sequence>
        </sequence>
      </complexType>
      <complexType name="ArrayOfstring">
        <complexContent>
          <restriction base="SOAP-ENC:Array">
            <sequence>                                       6
              <element name="item" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
            </sequence>
            <attribute ref="SOAP-ENC:arrayType" WSDL:arrayType="xsd:string[]"/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
  </types>
  <service name="setup">
    <documentation>Setup Service</documentation>
    <port name="setup" binding="tns:setupBinding">
      <SOAP:address location="http://localhost/soap"/>
    </port>
  </service>
  <* TYPE DEFINITIONS HERE /*>
</definitions>
```

Figure 69: Server side Deployment Descriptor (WSDL) for TCP/IP Ethernet speeds component, header components.

As can be seen, the DeploymentDescriptor identifies a name (“setup”) for the definitions and the service (at **1** and **6**, respectively). It also defines the urn (“urn:setup”) for the schema namespace and the namespace reference (at **3** and **4**). A complex type is also defined for returning an array of strings (at **5**). The missing component is the type and operation mappings, which is shown in

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="setup"                                             3
  <message name="getAvailableSpeedsRequest"></message>
  <message name="getAvailableSpeedsResponse">
    <part name="out" type="setup:ArrayOfstring"/>
  </message>
  <portType name="setupPortType">
    <operation name="getAvailableSpeeds">                             4
      <documentation>Service definition of function setup__getAvailableSpeeds
      </documentation>
      <input message="tns:getAvailableSpeedsRequest"/>
      <output message="tns:getAvailableSpeedsResponse"/>
    </operation>
  </portType>
  <binding name="setupBinding" type="tns:setupPortType">
    <SOAP:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getAvailableSpeeds">
      <SOAP:operation soapAction=""/>
        <input>
          <SOAP:body use="encoded" namespace="urn:setup"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </input>
        <output>
          <SOAP:body use="encoded" namespace="urn:setup"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
    </operation>
  </binding>
</definitions>
```

Figure 70: Remaining WSDL definition for Ethernet available speeds components.

This figure concludes the previous figure in providing the server-side data type mapping and operation mapping within the WSDL file. The “definitions” tag has been provided for continuity to the previous figure only. The message definitions (at **1**) define allowable message. The portType operation (“getAvailableSpeeds” refers to the input and output operations name from the tns namespace (at **2**, **3**). The binding operation “getAvailableSpeeds” (at **4**) defines the input and output urns and encodings.

SOAP Client

Finally, on the client side we have code that communicates with the server to retrieve the requested data, shown in Figure 71:

1

2

```
public class TCPIPEnetSpeedsClient
{
    public static void main (String[] args) throws Exception
    {
        URL    url    = new URL("http://localhost:1100");
        Call   call   = new Call();                          3

        call.setTargetObjectURI("urn:Setup");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC;);

        System.out.println("The SOAP server says: ");

        try
        {
            call.setMethodName("getAvailableSpeeds");
            Response resp = call.invoke(url, "" );
                                                             4
            if (resp.generatedFault())
            {
                Fault fault = resp.getFault();
                System.out.println("Fault code   = "+ fault.getFaultCode() );
                System.out.println("Fault string = "+ fault.getFaultString() );
            }
            else
            {
                Parameter ret   = resp.getReturnValue();
                Object    value = ret.getValue();

                System.out.println(value);
                String[] speeds = (String[]) value;

                for (int i = 0; i < speeds.length; i++)
                    System.out.println(speeds[i]);
            }
        }
        catch (SOAPException e)
        {
            e.printStackTrace();
        }
    }
}
```

Figure 71: SOAP client for server TCP/IP Ethernet speeds access.

As can be seen, this particular example is implemented as a Java class. The Url is canned to use the localhost server and port 1100 (at **1**). In a live ToolShed, these would be provided at the command line, through the initiating url parameter list, or as applet parameters. A Call object is also created here. The target urn and encoding style are then defined and associated with the call (at **2**). The method to call on the server side is then associated with the call prior to invoking the call on the server (at **3**). Finally, the result, assuming there is one, is retrieved and associated with a local value (at **4**). This example corresponds closely to the model that would be used to handle server data access in ToolShed.

In the ToolShed context, SOAP can thus act as a single point of communication for the various server-side data sources. A request is made for information, and the client need not know whether that information is being served from SNMP, Harmony, or Dictionary (or some other source). All that is required is a method name and class name.

SOAP Requirements in ToolShed

SOAP with HTTP has four requirements: (1) server side SOAP service, (2) SOAP client support, (3) a deployment descriptor, and (4) language support to perform the requested operations on client and server side. The soap.jar file is required of any client that would make use of the SOAP service on the server, to provide access to SOAP-specific classes and methods. As previously mentioned, the SOAP server must have defined locally, or remotely using WSDL, a deployment descriptor file. This file provides all of the management and interface information needed to support the translation from XML to native language requests and back again. If WSDL is used, it can be hosted from another server and not post any additional storage requirements on the server. Finally, there must be language support for the method calls themselves. The SOAP client will be using Java, so a JVM must be resident on the server if the client is hosted from the server. The SOAP server is using C/C++, so the associated classes must exist on the server side and have an entry point. Since the Harmony APIs are being used to acquire the server data, the SOAP server C++ classes must interact with the Harmony API.

Client Applications/ToolShed team will create the WSDL file for the server side, but will work closely with the server engineering team to define the WSDL. This work has been initiated. The ToolShed team is trying to resolve footprint problems relating the embedded SOAP server requirement using the gSOAP server. In order to use gSOAP with ToolShed, the client and server XML must be the same. Thus a common WSDL must be used and that WSDL must be used to create the .jar file for the ToolShed client. This work is ongoing.

Localization and Internationalization

Currently WebTools localization strings are partly implemented on the server and partly on the client. This is due to the fact that C cannot (currently) represent Unicode double-byte character formats. Thus strings that must work on the hardware, or C-only platforms, are maintained on the server, and the remaining strings are maintained on the client. Due to this particular problem, it is *not* possible, as long as the server-side implementations are required, or as long as C doesn't have full Unicode support, to have all of the strings hosted by the server or the client.

The bulk of internationalization rests with the structure of certain strings, such as dates and times, and symbols (such as units symbols), which aren't themselves the string content. Both of these problems must be addressed by the ToolShed approach.

General Localization Approach

The approach being used for ToolShed is to design and implement a new Localization Server (LS, see Localization Server), implemented using the ToolShed architecture, that can be polled for the localizations at build time. The resulting strings are saved to resource files and incorporated into the interface at runtime.

String Acquisition

At build time, strings are acquired from the Localization Server using the SOAP service. The process is illustrated in Figure 72:

```
 Initialize resource files for each EFIGSBPJDC language
 Parse UI XML file
   Iterate through elements
     If element is a textual component
       Iterate through EFIGSBPJDC
         Make a SOAP call to LS with value as key

         If a value is returned for language
           Write a key/value line into resource file
Terminate resource files for each EFIGSBPJDC language
```

Figure 72: Pseudocode for parsing localization files.

There are two components of this process that warrant discussion: (1) the parsing component, and (2) the SOAP call. The actual file construction will be straightforward, in that a preamble will be written, the key/value content, and then a postamble. The parsing, which is performed at build time, makes use of a LocalizationBuilder class, similar to JUIBuilder. The difference is that instead of creating Swing components it writes the strings to files. The method used is called initializeLocalizations() and is called from initialize().

String Translation

There are two classes associated directly with localization translation in ToolShed: (1) ResourceManager, and (2) ToolShedResource. The former is an abstract class and the latter extends it. The primary purpose of the resource manager is to select a resource based on the selected locale, and then to translate from the string key to the localized string value. The resources themselves are in files named: ToolShedString_lang_spec, where “lang” is a locale such as “en” and “spec” is a specialization such as “us”. These files are produced at build time.

In the JUIBuilder class, there is an initializeUI method that takes a Node and handles it. In the case of components, which account for all localization strings, the call sequence is shown in Figure 73:

method invocation order

method content

```
initializeUI(Node)
  initializeView(Node)
    initializeComponent(Node)
      initializeLabel(Node)


JUIBuilder:

  ToolShedResource mToolShedResource;

initializeLabel:

  JLabel           label = new JLabel();
  NamedNodeMap     nnp   = node.getAttributes();
  Node             nd    = nnp.getNamedItem("text");

  mToolShedResource = (ToolShedResource) ToolShedResource.getInstance();

  if (nd != null)
    label.setText(mToolShedResource.getString(nd.getNodeValue().trim()));

  nd = nnp.getNamedItem("name");

  if (nd != null)
    label.setName(mToolShedResource.getString(nd.getNodeValue().trim()));
```

Figure 73: String localization construction in ToolShed.

The initializeView() method (at **1**) is responsible for parsing the UI content and calls the initializeComponent() method (at **2**) which parses the specific components. The example provided above illustrates the parses the content for a label and constructs the Java Jlabel component. In this case, the base JUIBuilder class instantiates the ToolShedResource object (a singleton). The individual components access this object and set their string components to the localized versions based on the search key (the nominal English value). When the component is rendered, the localized value is displayed.

Design Requirements

This section is intended to give describe the ToolShed project design model. There are four components to the design:

- Use cases
- User interface elements
- Page flows (storyboards)
- Object model

This section hasn't really been fleshed out yet, but will include elements from the requirements document.

Use Cases

These talk about the use cases for each tool that is specified and the primary and major secondary scenarios fore each use case. First the actors for the project are defined, and then the use cases themselves.

Actors

There are eleven possible actors identified by the use cases for the ToolShed model:

- **USER:** Configuring/viewing the fiery server
- **CRIMSON:** The Crimson XML parser
- **TSAPPLET:** The UI Applet
- **TSAPP:** The desktop application
- **CWS:** Command Work Station
- **SOAP:** Soap (in whatever form, such as gSoap, Apache, Axis)
- **HARM:** Harmony
- **SNMP:** Simple Network Management Protocol
- **DICT:** Dictionary
- **STRINGS:** The Strings server
- **SERVER:** The fiery server

User-Driven Use Cases

Use cases for Setup fall into three general functional categories with respect to Fiery Server support, as defined in Fiery System 5.5 Server Product Specification, Remote Setup, pages 39-61:

- Server Setup

- Network Setup
- Printer Setup

In addition, ToolShed introduces new use cases to support configuration of the interface. Fiery Server support 'use cases', presented as client requirements, for the Server Setup, are presented in Table 6:

UI Tab	Option	Widget Type	Default Value
Server Setup	Server Name	JTextField	
Server Setup	Date	JTextField (3)	
Server Setup	Time	JTextField (2)	
Server Setup	Enable Printed Queue	JCheckbox	Selected
Server Setup	Jobs Saved in Printed Queue	JTextField	10
Server Setup	Preview While Processing	JCheckbox	Not selected
Server Setup	Use Character Set	JComboBox	Macintosh Windows (selected) DOS
Server Setup	Print Start Page	JCheckbox	Not selected
Server Setup	Enable Printing Groups	JCheckbox	Not selected
Server Setup	Clear Each Scan Job	JComboBox	Delete All Scan Jobs 1 day after scan (selected) 1 week after scan Manual
Server Setup	Clear Each Scan Job Now	JCheckbox	Not selected
Server Setup	Administrator	JTextField	
Server Setup	Operator	JTextField	
Server Setup	Auto Print Job Every 55 Jobs	JCheckbox	Not selected
Server Setup	Auto Clear Job Every 55 Jobs	JCheckbox	Not selected
Server Setup	Job Log Page Size	JComboBox	Tabloid/A3 Letter/A4
Server Setup	Fiery Contact Name	JTextField	
Server Setup	Fiery Contact Phone	JTextField	
Server Setup	Fiery Contact E-mail	JTextField	
Server Setup	Device Contact Name	JTextField	
Server Setup	Device Contact Phone	JTextField	
Server Setup	Device Contact E-mail	JTextField	

Table 6: Server support, Server Setup tab use cases.

Records colored in cyan were missing from the Fiery Product Specification, dated 04/16/2003. Fiery Server Support use cases for Network Setup are presented in Table 7:

UI Tab	Option	Data Type	Data Value/Range
Network Setup	Enable Ethernet	JCheckbox	Selected
Network Setup	Ethernet Speed	JComboBox	Auto Detect (selected) 1 Gbps 100 Mbps Full Duplex 100 Mbps Half Duplex 10 Mbps Full Duplex 10 Mbps Half Duplex
Network Setup	Enable Apple Talk	JCheckbox	Selected
Network Setup	Apple Talk Zone	JComboBox	Only enabled when the

			Enable Apple Talk is selected
Network Setup	Enable TCP/IP for Ethernet	JCheckbox	Selected
Network Setup	Enable AutoIP for Ethernet	JCheckbox	Selected, only enabled when Enable Ethernet is selected
Network Setup	Select Protocol	JComboBox	DHCP (selected) BOOTP Only enabled when Enable Ethernet is selected
Network Setup	IP Address	JTextField (4)	127.0.0.1 (def), network defined Fiery address, user editable, only enabled when the Enable Ethernet is selected and when the AutoIP is not selected. Each value is 0-255.
Network Setup	Subnet Mask	JTextField (4)	255.255.255.0 (def), enabled only when Enable Ethernet is selected and when AutoIP is not selected. Each value is 0-255
Network Setup	Enable Gateway Automatically	JCheckbox	Selected
Network Setup	Gateway Address	JTextField	127.0.0.1 (def)
Network Setup	Enable TCP/IP for Token Ring	JCheckbox	Selected
Network Setup	Enable DNS	JCheckbox	Selected
Network Setup	Get DNS Address Automatically	JCheckbox	Selected, only enabled if AutoIP configuration is selected
Network Setup	Primary DNS Server IP Address	JTextField	127.0.0.1 (def), only enabled if AutoIP Configuration is selected
Network Setup	Secondary DNS Server IP Address	JTextField	127.0.0.1 (def), on enabled if AutoIP Configuration is selected
Network Setup	Domain Name	JTextField	Only enabled if AutoIP Configuration is selected
Network Setup	Use DNS on	JComboBox	Any (selected) Ethernet Token Ring Only enabled if AutoIP Configuration is 'true', token ring is installed on the Fiery
Network Setup	Host Name	JTextField	Only enabled if AutoIP Configuration is not selected
Network Setup	Configure IP Ports	JCheckbox	Not selected
Network Setup	Enabled IP Ports	JComboBox	80 (HTTP) 137-139 (NETBIOS) 161-162, (SNMP) 515 (LPD) 631 (IPP) 9100-9103 (9100) EFI Ports

			All (by default) are selected, any deselected also disable the associated service
Network Setup	Enable IPX Auto Frame Type	JCheckbox	Not selected
Network Setup	Select Frame Types	JComboBox	Ethernet 802.2 Ethernet 802.3 Ethernet II Ethernet SNAP Token Ring Token Ring SNAP
Network Setup	Clear Frame Types	JCheckbox	Not selected
Network Setup	Enable LPD	JCheckbox	Selected
Network Setup	Enable PServer	JCheckbox	Not selected
Network Setup	NetWare Server PServer Poll Interval in Seconds	JTextField	15
Network Setup	Enable NDS	JCheckbox	Not selected
Network Setup	Select NDS Tree	String (select)	Network defined
Network Setup	Delete Bindery setup and continue	JCheckbox	Not selected
Network Setup	Is user login needed to browse NDS tree?	JCheckbox	Not selected
Network Setup	NDS Tree Name	JTextField	Network defined
Network Setup	Current path	JTextField	Network defined
Network Setup	Enter Password	JTextField	
Network Setup	Current path	JTextField	Network defined
Network Setup	Enter Your Print Server Password	JTextField	
Network Setup	Server should look for print queues in	JTextField	Entire NDS tree (def), network defined
Network Setup	NDS Tree Name	JTextField	Network defined
Network Setup	Current path	JTextField	Network defined
Network Setup	Choose File Server	JTextField	
Network Setup	Supported Servers	JTextField	
Network Setup	Remove support for	JTextField	
Network Setup	Select File Server	JTextField	
Network Setup	Enter First Letters of Server Name	JTextField	A
Network Setup	Add Server	JTextField	AAAAA
Network Setup	Add Server	JTextField	Server1
Network Setup	File Server Login	JTextField	
Network Setup	Enter Your Login Name	JTextField	guest
Network Setup	Enter Your File Server Password	JTextField	
Network Setup	NetWare Print Server	JTextField	
Network Setup	Enter Your Print Server Password	JTextField	
Network Setup	Enable Windows Printing	JCheckbox	Selected
Network Setup	Use WINS Name Server	JCheckbox	Not selected, enabled when Enable Windows Printing is selected, and when Auto IP is selected

Network Setup	WINS IP Address	JtextField (4)	127.0.0.1 (def), enabled only if Use WINS Name Server is selected
Network Setup	Server Name	JTextField	Enabled when Enable Windows Printing is selected, 15 chars max
Network Setup	Server Comments	JTextField	Enabled when Enable Windows is selected, 15 chars max
Network Setup	Set Domain Name	JTextField	Select from List (def) Enter Manually Enabled when Enable Windows Printing is 'true', 15 chars max
Network Setup	Workgroup or Domain	JTextField	
Network Setup	Choose Domain	JTextField	
Network Setup	Set Driver Type	JComboBox	PS (selected) PCL Only available if Windows Printing is supported, is selected
Network Setup	Enable Web Services	JCheckbox	Not selected
Network Setup	Enable IPP	JCheckbox	Selected
Network Setup	Enable SNMP	JCheckbox	Selected
Network Setup	Specify Read Community Name	JComboBox	Public (selected)
Network Setup	Specify Write Community Name	JComboBox	Public (selected) User defined
Network Setup	Enable Port 9100	JCheckbox	Selected
Network Setup	Enable Parallel Port	JCheckbox	Selected
Network Setup	Ignore EOF Character	JCheckbox	Not selected
Network Setup	Parallel Port Timeout (seconds)	JTextField	5
Network Setup	Enable on Ethernet	JCheckbox	Selected
Network Setup	Auto Select/Manual Select	JCheckbox	Not selected
Network Setup	Select Frames	JCheckbox	Not selected, enabled when Manual Select is selected
Network Setup	Bindery Setup	JCheckbox	Not selected, enabled when Enable PServer is selected
Network Setup	Change Trees	JCheckbox	Not selected, enabled when Enable NDS is selected
Network Setup	Enable LPD Printing Service	JCheckbox	Selected
Network Setup	Enable FTP Services	JCheckbox	Selected

Table 7: Server support, Network Setup tab use cases.

Records colored in orange lack sufficient information in the Fiery Product Specification, dated 04/16/2003, to determine their widget type or range of values. Records colored in cyan were missing from the Fiery Product Specification, dated 04/16/2003. Cells colored in green embed business logic that must be implemented with Type III validation.

Fiery Server Support use cases for Printer Setup are presented in Table 8:

UI Tab	Option	Data Type	Data Value/Range
Printer Setup	Enable Print Queue	JCheckbox	Selected
Printer Setup	Enable Direct Connection	JCheckbox	Selected
Printer Setup	Enable Hold Queue	JCheckbox	Selected
Printer Setup	Default Paper Sizes	JComboBox	US (selected) Metric
Printer Setup	Convert Paper Sizes	JComboBox	No Letter/Tabloid>A4/A3 A4/A3>Leter/Tabloid (selected)
Printer Setup	Page Order	JComboBox	Forward (selected) Reverse
Printer Setup	Default Color Mode	JComboBox	CMYK (selected) Grayscale
Printer Setup	Print Cover Page	JCheckbox	Not selected
Printer Setup	Allow Courier Substitution	JCheckbox	Selected
Printer Setup	Print to PS Error	JCheckbox	Not selected
Printer Setup	Print Master	JCheckbox	Not selected
Printer Setup	Select Printer	JTextField	
Printer Setup	Printer Type	JTextField	
Printer Setup	Parallel Connection	JComboBox	Print Queue (selected) Hold Queue Direct Connection Only published queues may be selected
PCL Setup	Paper Size	String	PPD defined
PCL Setup	Orientation	JComboBox	Portrait (selected) Landscape
PCL Setup	Form Length	int	60
PCL Setup	Font Source	JComboBox	Internal (selected) Softfont (Internal)
PCL Setup	Font Number	int	0
PCL Setup	Font Pitch	real	10.00
PCL Setup	Points (Font Size)	real	12.00
PCL Setup	Symbol Set	JComboBox	ASCII (selected), ROMAN_8, ECMA_94L1, PC_8, DN, PC_850, ISO_SWED_NAMES, ISO_NORWEGIAN, LEGAL, VENTURA_INTNTL, VENTURA_USA, DESKTOP, WINDOWS_L1, PS_TEXT, ISO_ITALIAN, ISO_FRENCH, MATH_8, PS_MATH, PI_FONT, PC_852, WINDOWS_L2, VENTURA_MATH, WINDOWS31_L1, ISO_LATIN2, ISO_LATIN5, MICROSOFT_PUB, PC_TURK, WIN_LATIN5,

			ISO_UK, ISO_GERMAN (ASCII)

Table 8: Server setup Printer Setup tab use cases.

Records colored in orange lack sufficient information in the Fiery Product Specification, dated 04/16/2003, to determine their widget type or range of values. Records colored in cyan were missing from the Fiery Product Specification, dated 04/16/2003. Cells colored in green embed business logic that must be implemented with Type III validation.

ToolShed configuration support use cases are presented in Table 9:

Type	Configuration Option	
Configuration	Select Functionality	
Configuration	Show/Hide Functionality	
Customization	Customize Layout	
Customization	Customize Look and Feel	

Table 9: ToolShed configuration support use cases.

USER SelectTSSetup into TS

Precondition: Opening screen is viewable on the USERs browser.

Flow of Events:

Primary Scenario

1) USER selects setup button from application menu of main page

12) TS interacts with SERVER to obtain UI, Data, and Constraint information

13) TS parses, constructs, and renders the main setup page

Postcondition: The setup interface is displayed

User Interface Elements

Functional UI Elements provide details on key screen and user interface components. The application interface is constructed entirely from Java Swing (Java Foundation Class) instances. Representative examples of a variety of Swing components (using the Metal look and feel) are provided in this section. The look and feel as well as the method of implementation (dialog box, full screen, etc.) will be determined as part of a separate Look and Feel and usage model design effort. The figures presented below are provided as illustrations of the capabilities of the Swing component package, and have been taken directly from the Sun SwingSet demo:

http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html

As previously mentioned, the following components are required in the known (from the most recent UI mockups, dated 2/5/2003) ToolShed-supported applications:

- TabbedPanel ⇨ JTabbedPanel
- Label ⇨ JLabel
- Button ⇨ Jbutton
- CheckBox ⇨ JCheckBox
- List ⇨ JList
- TypeText ⇨ JTextField
- ComboBox ⇨ JComboBox
- RadioButton ⇨ JradioButtonGroup
- Table ⇨ Jtable
- ScrollText ⇨ ?
- In addition, a few Swing components or necessary and/or may come in handy:
- JPanel
- JFrame
- JScrollBar
- JSlider
- JTree
- JProgressBar

Each of these will be illustrated with a figure from the SwingSet demonstration.

JTabbedPanel

This component is heavily used in WebTools/ToolShed and is somewhat configurable as can be seen in Figure 74:

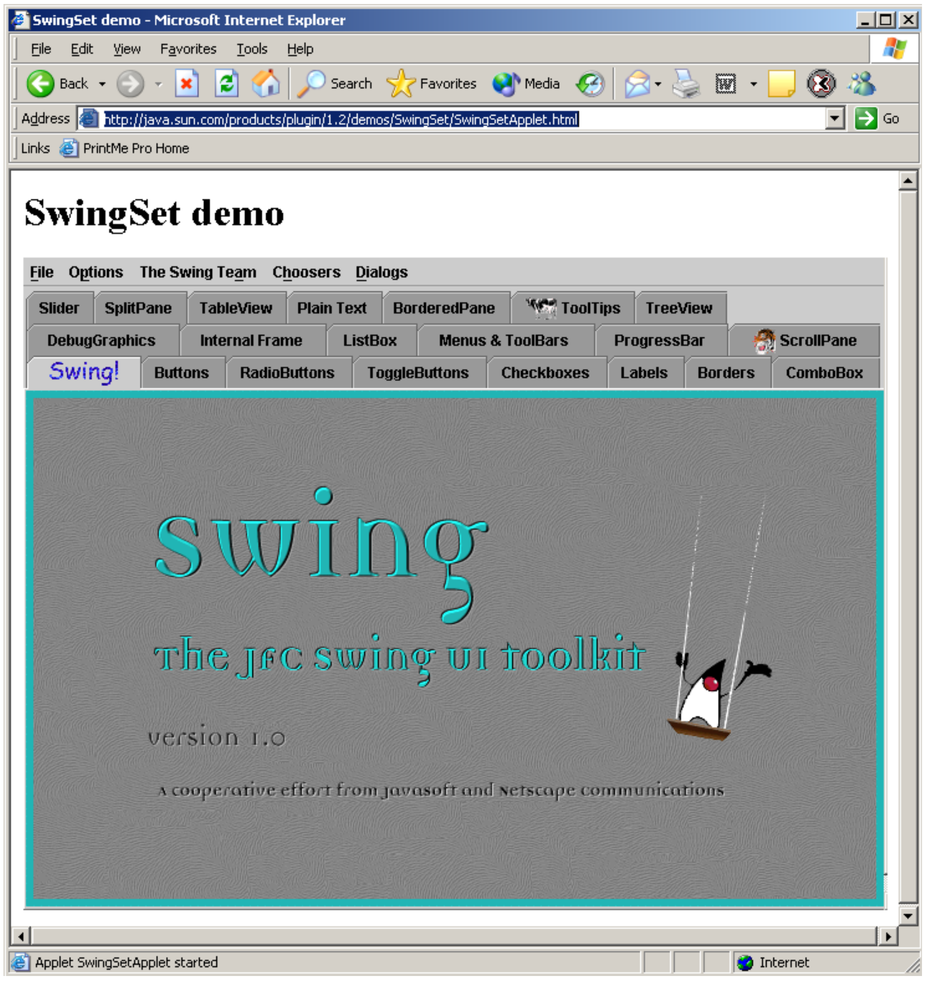

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
swing
the JFC swing UI toolkit
version 1.0
A cooperative effort from javasoft and netscape communications
Applet SwingSetApplet started
Internet

Figure 74: Swing tabbed panel component, JTabbedPanel.

JLabel Component

Provides standard label functionality, and is shown in Figure 75:

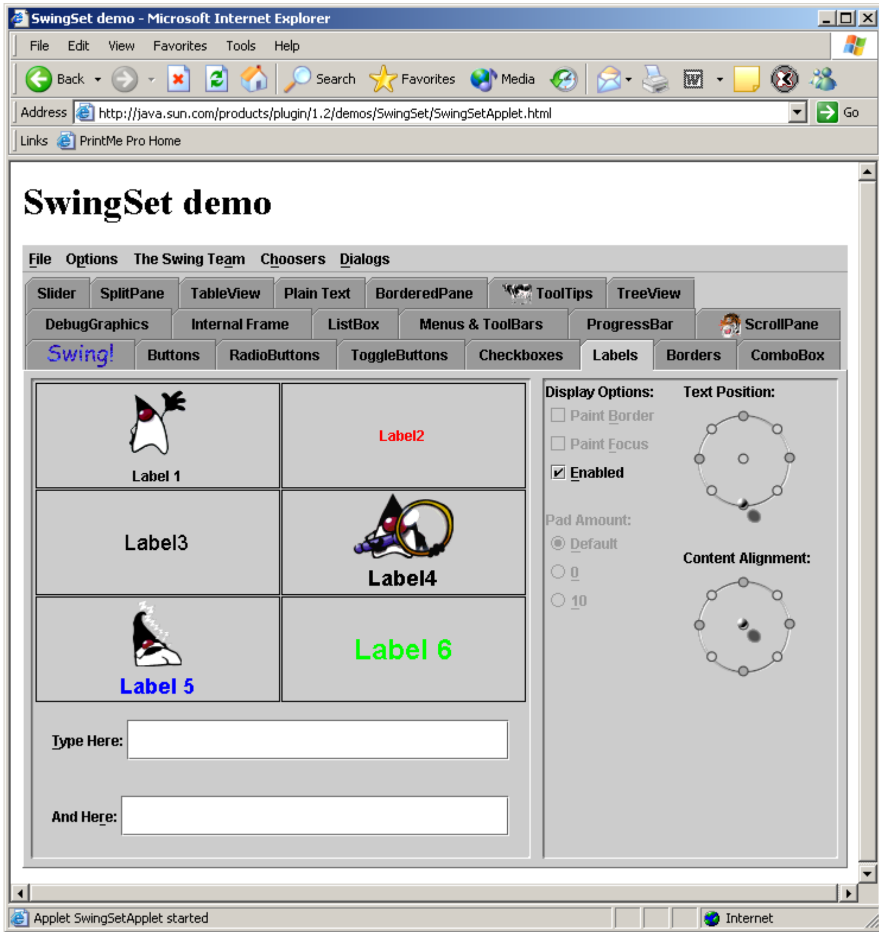

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Go
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Label 1
Label2
Label3
Label4
Label 5
Label 6
Type Here:
And Here:
Display Options:
Paint Border
Paint Focus
Enabled
Text Position:
Pad Amount:
Default
0
10
Content Alignment:
Applet SwingSetApplet started
Internet

Figure 75: Swing label component, JLabel.

JButton Component

Provides standard button functionality, and is shown in Figure 76:

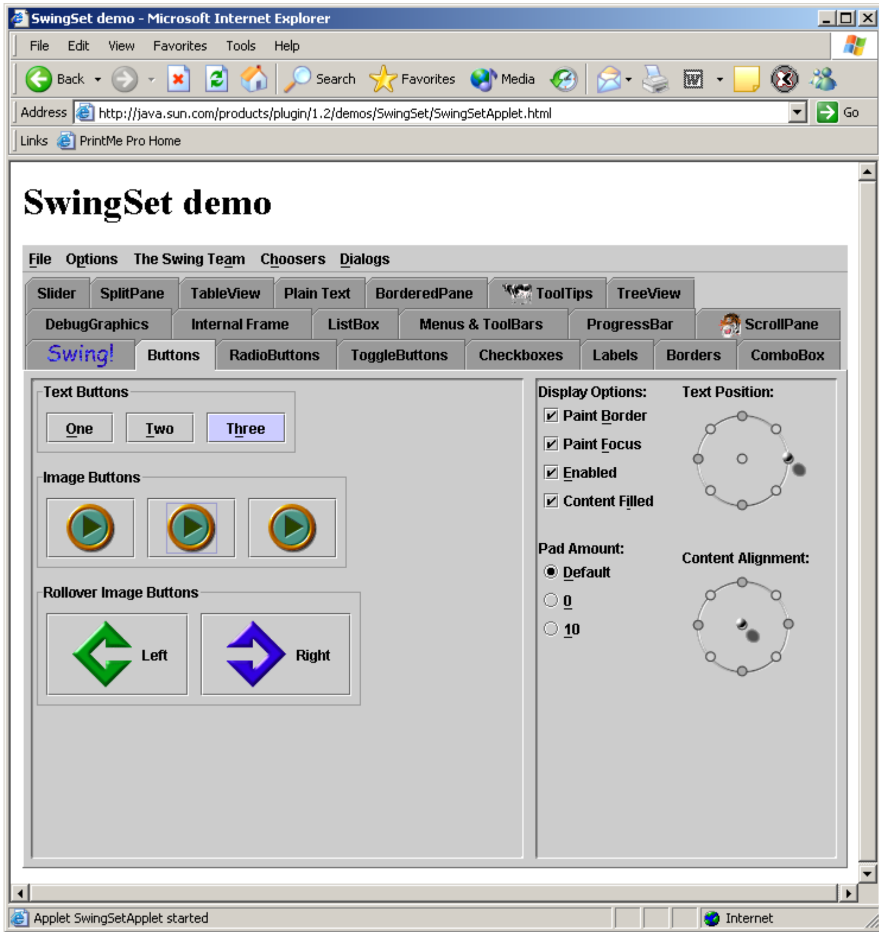

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Text Buttons
One Two Three
Image Buttons
Rollover Image Buttons
Left Right
Display Options:
Paint Border
Paint Focus
Enabled
Content Filled
Text Position:
Pad Amount:
Default
0
10
Content Alignment:
Applet SwingSetApplet started
Internet

Figure 76: Swing button component, JButton.

JCheckbox Component

Provides standard checkbox functionality, and is shown in Figure 77:

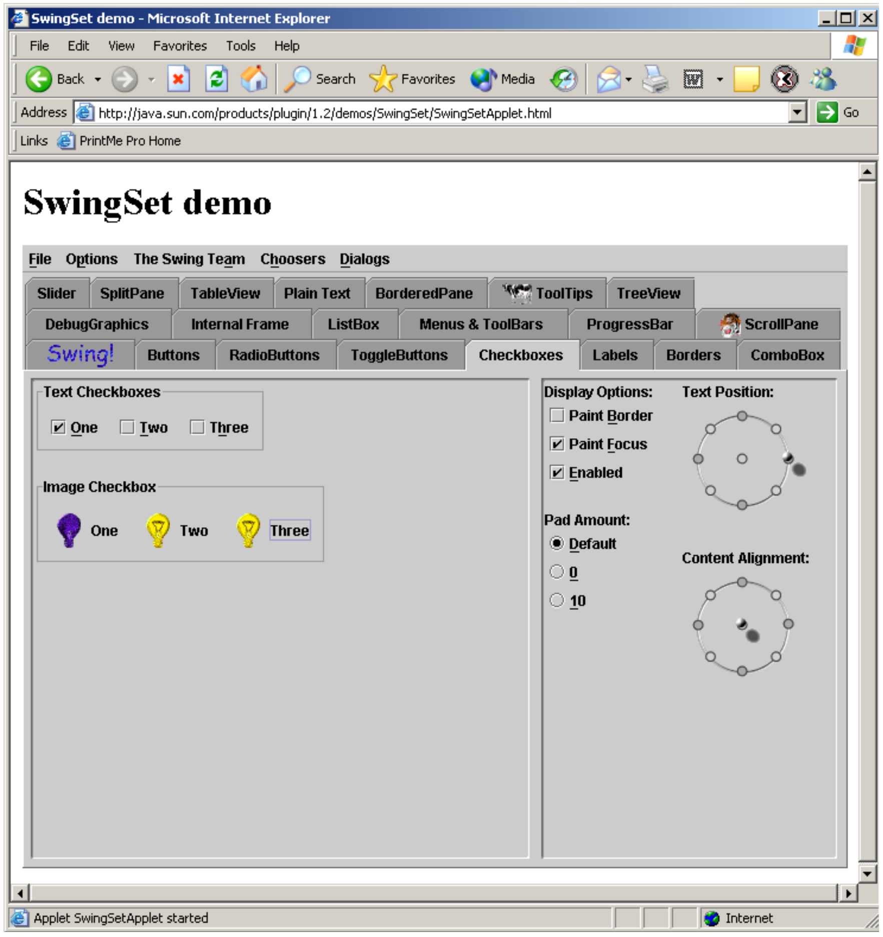

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Text Checkboxes
One Two Three
Image Checkbox
One Two Three
Display Options:
Paint Border
Paint Focus
Enabled
Text Position:
Pad Amount:
Default
0
10
Content Alignment:
Applet SwingSetApplet started
Internet

Figure 77: Swing checkbox component, JCheckbox.

Jlist Component

Provides standard list functionality, and is shown in Figure 78:

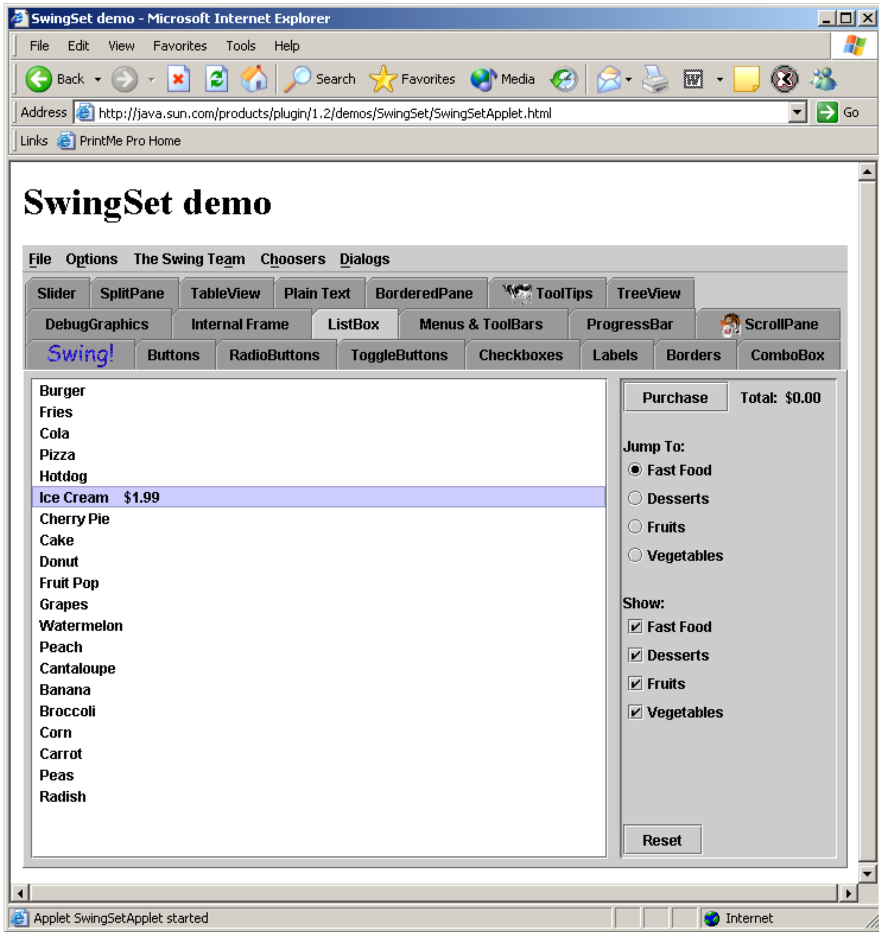

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Go
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Burger
Fries
Cola
Pizza
Hotdog
Ice Cream $1.99
Cherry Pie
Cake
Donut
Fruit Pop
Grapes
Watermelon
Peach
Cantaloupe
Banana
Broccoli
Corn
Carrot
Peas
Radish
Purchase
Total: $0.00
Jump To:
Fast Food
Desserts
Fruits
Vegetables
Show:
Fast Food
Desserts
Fruits
Vegetables
Reset
Applet SwingSetApplet started
Internet

Figure 78: Swing list component, JList.

JTextField

Provides standard text field functionality, and is shown in Figure 79:

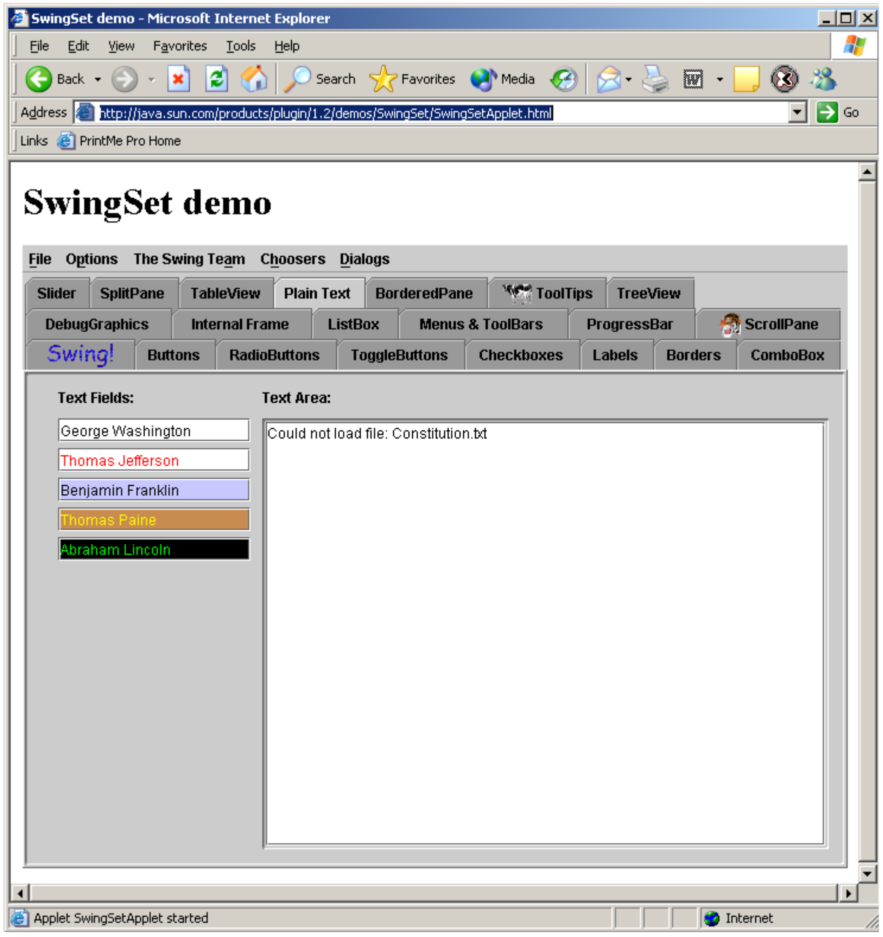

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Text Fields:
Text Area:
George Washington
Thomas Jefferson
Benjamin Franklin
Thomas Paine
Abraham Lincoln
Could not load file: Constitution.txt
Applet SwingSetApplet started
Internet

Figure 79: Swing textfield component, JTextField.

JComboBox Component

Provides standard combo box functionality, and is shown in Figure 80:

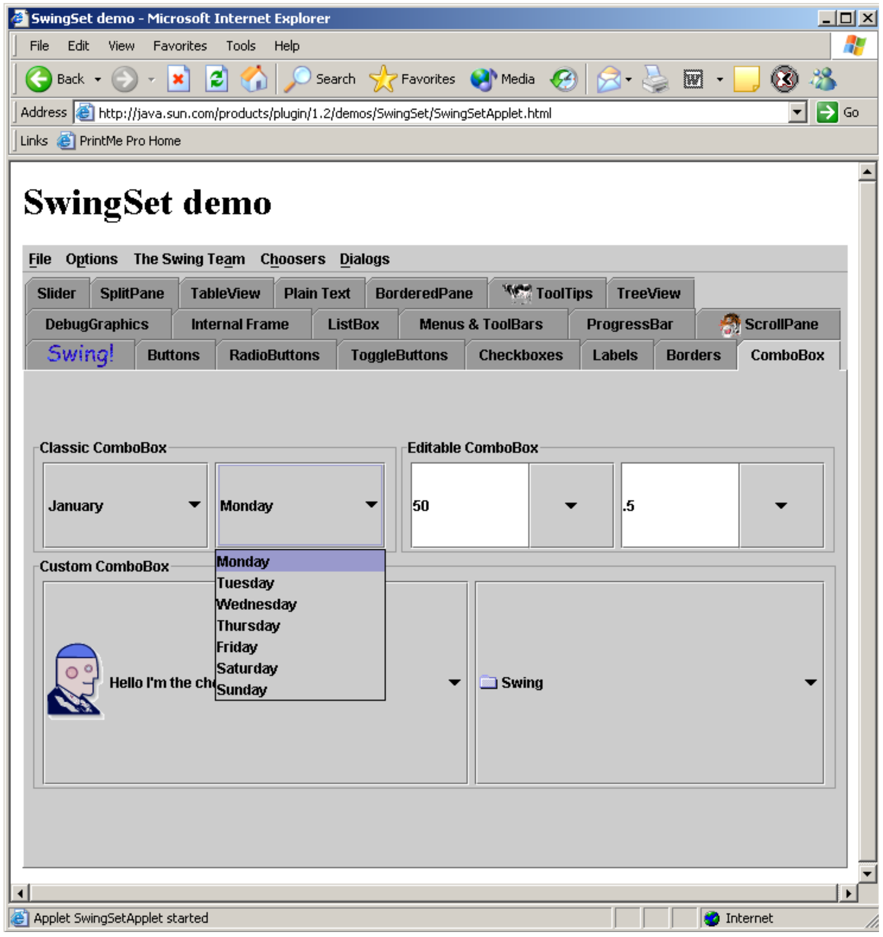
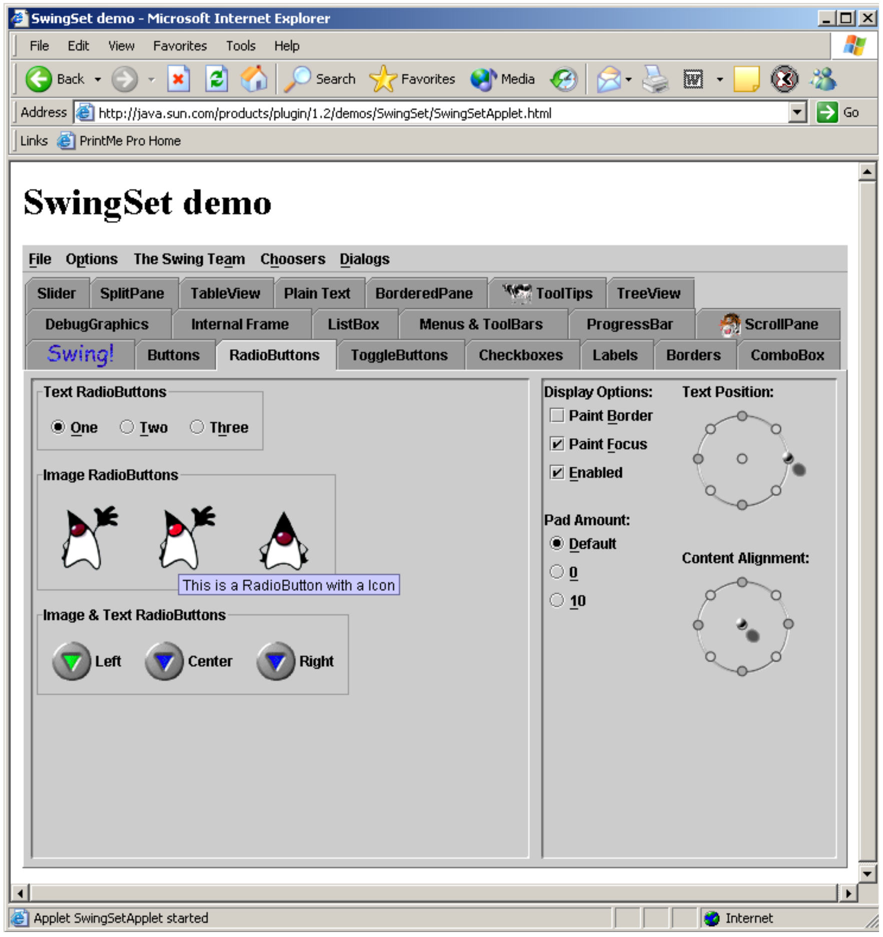

SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Go
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Classic ComboBox
January
Monday
Editable ComboBox
50
.5
Custom ComboBox
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Hello I'm the ch
Swing
Applet SwingSetApplet started
Internet

Figure 80: Swing combo box component, JComboBox.

JRadioButton Component

Provides standard radio button functionality, and is shown in Figure 81:


SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html
Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Text RadioButtons
Image RadioButtons
Image & Text RadioButtons
Left Center Right
Display Options:
Text Position:
Pad Amount:
Content Alignment:
Applet SwingSetApplet started
Internet

Text RadioButtons: [✓ One] Two Three

This is a RadioButton with a Icon

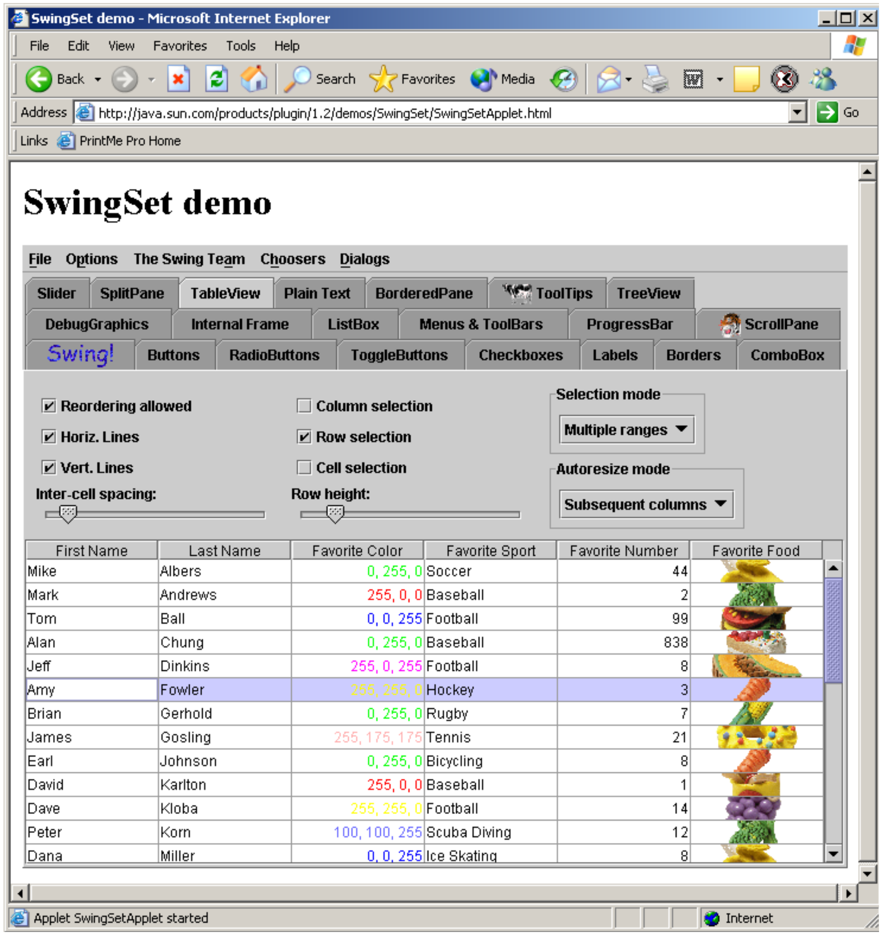
Display Options: Paint Border [✓ Paint Focus] [✓ Enabled]

Pad Amount: [✓ Default] 0 10

Figure 81: Swing radio button component, JRadioButton.

JTable Component

Provides standard table functionality, and is shown in Figure 82:


SwingSet demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://java.sun.com/products/plugin/1.2/demos/SwingSet/SwingSetApplet.html Go
Links PrintMe Pro Home
SwingSet demo
File Options The Swing Team Choosers Dialogs
Slider SplitPane TableView Plain Text BorderedPane ToolTips TreeView
DebugGraphics Internal Frame ListBox Menus & ToolBars ProgressBar ScrollPane
Swing! Buttons RadioButtons ToggleButtons Checkboxes Labels Borders ComboBox
Applet SwingSetApplet started
Internet

- [✓] Reordering allowed
- [✓] Horiz. Lines
- [✓] Vert. Lines
- [ ] Column selection
- [✓] Row selection
- [ ] Cell selection

Inter-cell spacing:

Row height:

Selection mode: Multiple ranges

Autoresize mode: Subsequent columns

First Name	Last Name	Favorite Color	Favorite Sport	Favorite Number	Favorite Food
Mike	Albers	0, 255, 0	Soccer	44	
Mark	Andrews	255, 0, 0	Baseball	2	
Tom	Ball	0, 0, 255	Football	99	
Alan	Chung	0, 255, 0	Baseball	838	
Jeff	Dinkins	255, 0, 255	Football	8	
Amy	Fowler	255, 255, 0	Hockey	3	
Brian	Gerhold	0, 255, 0	Rugby	7	
James	Gosling	255, 175, 175	Tennis	21	
Earl	Johnson	0, 255, 0	Bicycling	8	
David	Karlton	255, 0, 0	Baseball	1	
Dave	Kloba	255, 255, 0	Football	14	
Peter	Korn	100, 100, 255	Scuba Diving	12	
Dana	Miller	0, 0, 255	Ice Skating	8	

Figure 82: Swing table component, JTable.

▪

Implementation and Resource Requirements

The following sections address the suggested technical development strategy for the ToolShed project, along with an initial look at the resource requirements for the project. It should be emphasized that at this stage of the project firm estimates will be difficult to produce, let alone to live up to, especially considering that some of the items being developed are bleeding edge technology items.

Phased Implementation Plan

The implementation of this project is comprised of three separate stages:

1. Construct initial prototypes. This phase included the following prototypes:

   - initial XMLTalk UI (read only, write, read write) demonstrations, along with demonstrations of different UI components
   - Demonstration model of WebScan which used a Harmony call and a Java bean to instantiate the mailbox information.
   - Use of UI constraints.
   - User of Type 1 validation.
   - Use of setup files.

2. Construct remaining functional prototypes. This phase will include the following prototypes:

   - ToolShed executive prototype.
   - WebSetup prototype with full UI functionality.
   - Development of XML Schema models for UI and Object models.
   - Separation of UI-specific and Object-specific XML files.
   - Type 2 validation prototype.
   - Type 3 validation prototype.
   - SOAP/Communications prototype.
   - End-to-End integration prototype.
   - Localization prototype.
   - Unit tests created.

3. Alpha version of ToolShed. This phase will include the following:

   - All UI components functional.
   - All Object types can be communicated to/from the server with selected communications mechanism(s).
   - All validation types supported.
   - Localization support functional.
   - Product application prototype.
   - Release/User manuals drafted.
   - Test cases developed.

Resource Allocation Requirements

At the time of printing, noone is working on the ToolShed project full time. No progress can continue without at least one fulltime software engineer working on the project. The Phase 1 demonstrations have been completed, and a draft of the functional specification is complete and is updated as new prototypes are constructed. A true functional specification, one in which the architectures of all functional prototypes are merged into a single coherent architecture will be developed when prototyping is complete.

Phase 2 will require the addition of two Client Applications software engineers at the senior level, due to the independence required in the prototyping phase. These engineers will be responsible for following the project architectural design and propagating that design into their implementation designs and the associated implementations so that massive rework/reengineering isn't later required. At full time, the prototypes in this phase can be designed and implemented in approximately 8 weeks.

Phase 3 will require the same team of engineers, the participation of the project manager at 30% time, and the participation of the first product manager at 10% time. It will, in the later stages of the phase, involve QA personnel to assist in developing test cases to validate the model. It is anticipated that this phase will take approximately 8-12 weeks.

References

Tools Used in ToolShed Design and Implementation

Spec: MS Word

Diagrams: Visio

UML: TogetherSoft Control Center 6.1

Planning: MS Project (none yet)

Development: XML Spy, Jbuilder 7, Jakarta Ant, CVS

Links and Reference Documents

XMLTalk

- http://www.trcinc.com/knowledge/articles/XMLTalk.pdf
- http://www.trcinc.com/knowledge/software/xmltalk/xmltalk.asp
- http://groups.yahoo.com/group/xmltalk-dev/

RuleML Related

- http://www.dfki.uni-kl.de/ruleml/
- Extensible Rule Markup Language (XRML): http://xrml.kaist.ac.kr
- Object Constraint Language (OCL): http://www.csci.csusb.edu/dick/samples/ocl.html
- Bowers, S. and Delcambre, L., "Representing and Transforming Model-Based Information", Oregon Graduate Institute.
- Boley, H., Tabet, S, and Wagner, G., "Design Rationale of RuleML: A Markup Lanaguage for Semantic Web Rules": http://www.di.ufpe.br/~compint/aulas-IAS/artigos/BolyTabetWagnerRuleML.html
- SweetJess: http://userpages.umbc.edu/~mgandh1/
- JESS: http://herzberg.ca.sandia.gov/jess/
- Mandarax: http://www.mandarax.org

SOAP

- GSOAP: http://sourceforge.net/projects/gsoap2

Validation

- Validation with Java and XML Schema: http://www.javaworld.com/javaworld/jw-09-2000/jw-0908-validation_p.html

Localization Server

- Draft specification: Functional Specification 11/24/2003

Links to ToolShed Documents

Client Applications Forum on Teamsite: Index

Client Applications Java Style Guidelines: JavaStyleConventions-071003

Client Applications ToolShed Development Plan: Implementation Plan