White Paper: e4 Technical Overview

John Arthorne, IBM Canada Inc.
Revision: 0.9. Last modified July 29, 2009.

Executive Summary

The Eclipse platform was first targeted at building an extensible IDE component framework. It has since evolved into a general-purpose platform for building extensible software applications of all kinds. Eclipse applications are now found in such diverse deployment environments as web servers, web browsers, embedded clients, and traditional rich desktop applications. The e4 platform was designed to simplify development of software components and component-based applications to meet the demands of this ever changing computing landscape. This paper provides a technical overview of the e4 architecture and programming model.

What is e4?

To provide some context on the rest of this paper, it is useful to first clarify what precisely e4 is. The most useful definition is that e4 is a cluster of related technologies for building extensible component-based applications. Rather than a wholesale replacement of the Eclipse platform, e4 brings a new set of technologies into the existing Eclipse platform that make Eclipse components easier to write, more configurable by application developers and integrators, and easier to reuse in diverse runtime environments.

The first generation of the Eclipse platform (releases 1.0 to 2.1) was primarily an integration platform. Its main strength was pulling together diverse plug-ins written by different authors, and integrating them into a common application with a consistent and cohesive end user experience.

The second generation of the Eclipse platform (the 3.x releases), was powered by an OSGi runtime, making it a more powerful general purpose component-based application framework. This second generation platform was good at scaling up from very small embedded applications up to large rich client applications and web servers. However, each component (plug-in) was typically very difficult to reuse outside of the specific environment for which it was designed and tested. It was easy to add or remove components from the system, but often quite difficult to take a component designed for one application or runtime environment and reuse it in a completely different application or environment.

The e4 vision is to make it much easier to write components that are more reusable and customizable for a wide range of applications and environments. There are two ways to achieve this goal: reduce the external dependencies and assumptions made by components, and widen the set of languages and technologies that can be seamlessly integrated into the Eclipse runtime. Both of these approaches are taken in e4, through several avenues of exploration:

- A service-oriented programming model, based on OSGi, that provides better isolation of software components from their surrounding environment.
- The GUI is represented as a uniform model that can be generically queried, manipulated, tooled, and extended, allowing for rapid design and customization of the user interface with little or no coding effort.

- Use of web styling technology (CSS), allows the presentation of user interface elements to be infinitely tweaked and reconfigured without any modification of application code.
- Bringing Eclipse runtime technology into the JavaScript world, and enabling software written in JavaScript to be executed in the Eclipse runtime.
- A framework for defining the design and structure of Standard Widget Toolkit (SWT) applications declaratively. This eliminates writing of repetitive boilerplate SWT code, thus reducing development cost, improving UI consistency, and enabling customized application rendering.
- A new port of SWT, dubbed "browser edition", that allows existing SWT applications to be executed on web platforms such as ActionScript/Flash.
- In the development tools space, a more flexible resource model that provides better support for complex project layouts.

The remainder of this paper will outline these new technologies in more detail. A general knowledge of the current Eclipse platform is assumed, so readers unfamiliar with Eclipse should consult other resources such as the Eclipse Platform Technical Overview for background information.

Programming Model

The e4 programming model starts with the existing principles of programming in Eclipse:

- Applications are made up of modular, loosely coupled components called *plug-ins* or *bundles*. Bundles can be made up of code written in Java and other languages, and/or other resources such as documentation and source code.
- Bundles can declare points where they can be customized or extended using *extension points*, and customize or extend other bundles by defining *extensions*.
- A very large number of bundles can be installed at a time, but only those bundles that are actually in use will be loaded and consume system resources.

Where e4 differs from this traditional Eclipse programming model is in how bundles interact with each other outside the extension registry mechanism. Bundles often need to provide data and software services to other bundles in ways that aren't suited to the Eclipse extension registry. This was most commonly achieved by bundles *reaching out* to other bundles by directly referencing methods and constants defined in API Java classes. Bundles would typically define entry points for obtaining singleton instances of services (for example classes such as `Platform`, `IDE`, `ResourcesPlugin`, `JavaCore`, etc). This practice of reaching out led to tight coupling from bundles using services to a particular provider of that service, and the prevalence of singleton accessors made it difficult or impossible for alternate service implementations to be substituted, or for multiple implementations to be available at the same time. The resulting bundles were therefore difficult to reconfigure or reuse in different environments where different or multiple service implementations would be needed.

Service programming models generally define three distinct participants: service *providers*, service *consumers*, and a *broker* or *container* that manages binding of service providers to consumers. Basic implementations of this programming model, such as the OSGi service API or

the Eclipse extension registry, nicely decouple service providers from consumers, but often require service providers and consumers to have explicit knowledge of the particular container or service broker.

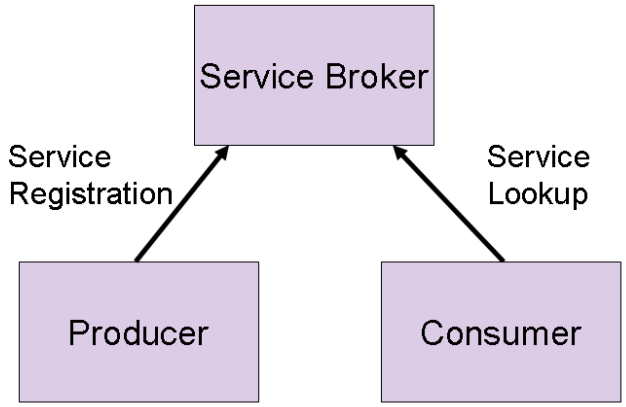

Figure 1 – Simple service architecture

The e4 programming model aims to further decouple these participants by reducing or eliminating these explicit links from the service producers and consumers to a specific service broker technology. This new flexibility is best explained by looking at how these three service participants are defined in e4.

Contexts: the service broker

e4 introduces the notion of *context* as a generic mechanism that stores or knows how to locate services and provide them to service consumers. At its basic level, an e4 context looks much like a Java `Map` storing values associated with some key. A client can put values into the map, or retrieve values from the map. The map can also store *context functions* that are pieces of code that know how to compute a context value lazily when values are requested by the client. When a client asks for a value not currently defined in the context, it will delegate to a parent context. This allows services or data to be stored in a central place and be consumed by many clients. Finally, contexts have a pluggable lookup strategy that allows external parties to "teach" the context how to retrieve certain kinds of values. The lookup strategy enables interoperability between e4 contexts and other service brokers such as the OSGi service registry. This flexibility means all manner of different service lookup and brokerage systems can be integrated into the e4 context mechanism. This ability to create context hierarchies and insert service lookup strategies allows contexts to scale up from very simple map-like service registration and lookup to highly complex and dynamic service arbitration mechanisms.

Injection: service consumers

The best practice in modern service programming models is that consumers receive dependencies via dependency injection. This theoretically allows application code to completely eliminate its dependency on a particular container technology, thus enabling greater reuse. e4 directly supports and encourages dependency injection as a means to supply services to clients. Constructor, method, and field injection are all supported, and injection points can be identified in client code using either naming conventions or Java annotations. For clients that find inversion of control

confusing and prefer code clarity over framework independence, the e4 context API can also be used directly (the service locator design pattern).

Service providers

There is great variety in the services and data that bundles make available for consumption by other bundles. Some services are very lightweight and may be consumed and discarded thousands of times, while others are heavyweight services designed to live for long periods of time. There are many different life-cycles to these services - some come and go based on the existence of a particular UI widget, others may be tied to the lifecycle of a bundle. Due to this great variety, there is no one single method of service publication that is appropriate for all service providers.

In general, services are published in e4 using the OSGi service mechanism. Services can either be registered and removed programmatically, or via OSGi declarative services. Declarative services allow the service instantiation to be delayed until required by some client. Of course, there are a wide variety of helper frameworks that can be used for publishing OSGi services, such as Spring DM, iPOJO, Peaberry, etc. By using OSGi services as the foundation for service publication, all of these frameworks can be used seamlessly in e4.

While most client code will use services, frameworks that create contexts can simply add services directly to a context where appropriate. For example a service associated with a widget may want to register with a context in the widget constructor and withdraw the service when the widget is disposed. Manual registration allows a service to be made available only to a specific context, rather than making it globally available via OSGi services. Finally, contexts support outjection (reverse injection) of services back into the context via a field or accessor method.

Putting this all together, we have a model where the typical service consumer knows nothing about who provided the service, or about what broker was used to bind and obtain the service. The service producer can also be largely decoupled from the service broker by separating the service configuration data from the service implementation itself (either using declarative mechanisms such as DS, outjection, or simply by separating the framework-aware code doing the registration from the service implementation itself. This model is illustrated in figure 2.

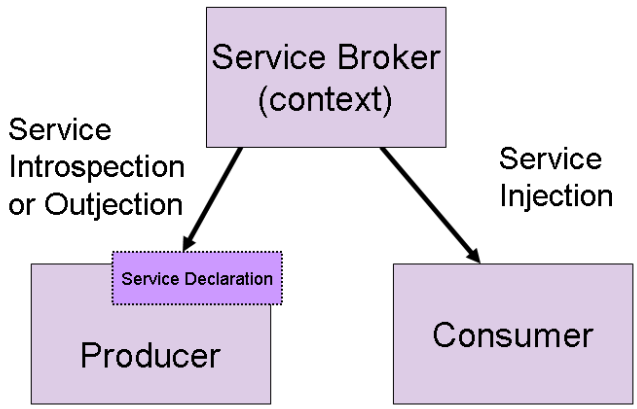

Figure 2 – e4 service architecture

Eclipse application services (the "twenty things")

To support this decoupled service programming model, Eclipse APIs need to be made available as services rather than via the singleton accessor model of the past (`Platform`, `IWorkbench`, etc). However, over the years the Eclipse platform has accumulated a vast API, of which many components use only a few. This API breadth alone make Eclipse a challenging platform to build applications on; the steep learning curve increases time to value and discourages casual developers from building on the platform. This is being addressed in e4 by defining a core set of services that capture a broad swath of useful platform functionality. The goal is that many developers should be able to build well integrated e4 bundles using only these core services. These core services are occasionally referred to in e4 discussions as the "twenty things", to capture the fact that it is a bounded set of important services. These core services also define a good starting point for integration of the Eclipse platform with other programming languages. This is discussed further in the later section on JavaScript integration.

Modeled User Interface

The previous generation of the Eclipse platform UI (called the *workbench*) was a complex and difficult to maintain piece of software. Although it has been made somewhat more flexible over the years, it still enforces a rigid, hard-coded model of the workbench structure and layout: a single workbench containing workbench windows, with each window containing one or more workbench pages, and each page made up of an editor area, a set of view stacks, and some hard-coded trim elements (perspective switcher, progress indicator, etc). Application designers have a strictly limited set of options when customizing the structure of Eclipse-based applications.

The e4 workbench greatly increases flexibility for application designers by providing a formal model of the elements that comprise an instance of the workbench. Applications can reconfigure or extend this model to arrive at very different presentations of their application with no additional coding required. Normalizing the workbench structure as a well defined model has the added benefit of making the code for the workbench itself much simpler and less error prone. Most importantly, this allows for very different workbench UI layouts, such as parts living outside of perspectives, views and editors in dialogs, and other designs not previously allowed by the older generation workbench with its rigid hand-crafted model. Having a model also allows for more advanced tool support for application designers, such as visual design tools.

The e4 workbench model can also be manipulated on the fly, and model changes are rendered immediately in the UI. This opens the door to scripted manipulation of the workbench structure and state, much like how JavaScript manipulates a document object model in a web browser. In figure 3, we see a model editor that is being used to customize the running application - in this case changing the name and tooltip of the traditional Eclipse problems view.

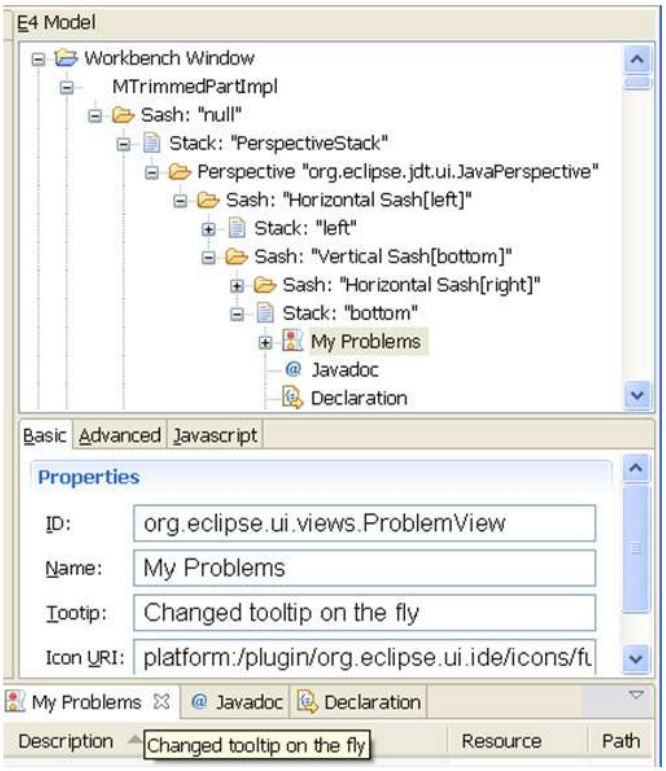

Figure 3 – e4 model editor

The e4 workbench model is translated into widgets via a *renderer*. The workbench comes with a default renderer that instantiates the model as SWT widgets, but alternate renders can be supplied to render model elements differently. This can be used to make subtle changes to the concrete widgets shown to the user, or even to allow rendering in a completely different widget library or runtime environment such as a web browser. Renderers are contributed at the level of individual model elements, rather than a single monolithic renderer for the entire application. A single renderer can supply rendering for one or more model elements, or conversely there can be multiple renderers available for a given model element. This fine granularity of extensibility allows clients to extend the workbench model with their own elements, and then insert custom renderers for rendering their own model elements in a particular way.

Declarative Styling

While the basic translation of the workbench model into widgets is performed by the renderer, a pluggable *styling engine* is used to customize the fonts, colors, and other aspects of widget presentation. As we have learned from the evolution of web presentation technologies, separating document structure (HTML) from style (CSS) is a powerful way to ensure a consistent look and feel across many documents, and to allow style changes to be made easily in a single place.

The e4 styling engine has no knowledge of the model-based UI, and in fact can run as a completely independent piece on earlier Eclipse versions. The engine takes concrete widgets and styling data as input, and performs the styling on the instantiated widgets to produce the styling

result. Figure 4 shows the flow from the model, into widgets via one or more renderers, and then to a styled output using the styling engine and the declarative styling data (CSS files).

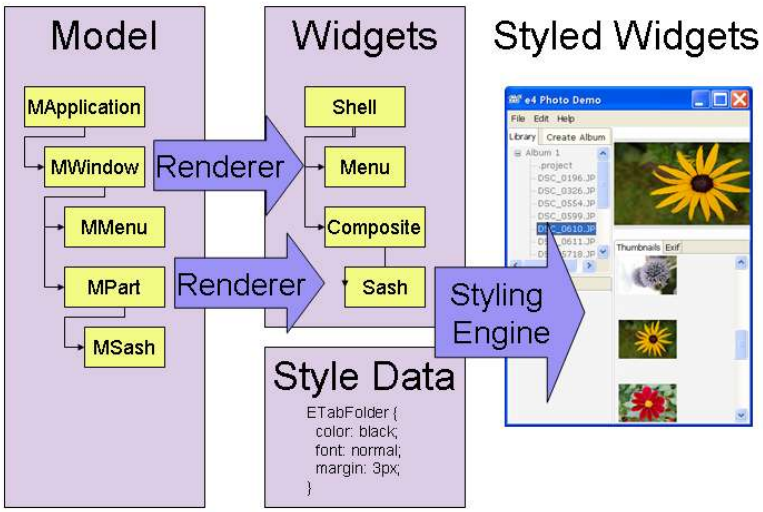

Figure 4 – Render and style dataflow

The declarative styling support in e4 is based on standard CSS syntax. A large subset of standard HTML CSS property names and type formats are supported, such as fonts, margins, and colors. In addition, e4 has custom properties and pseudo selectors specific to many Eclipse widgets.

CSS supports the standard set of selectors, mapped where appropriate to SWT. Where in CSS you would use type selectors, in e4 you can use the widget class name as a selector. CSS class selectors and IDs are also supported so that the same widget can be styled in different ways. For example, the views might be tagged ".views" and editors ".editors", with different styling rules for each. Similarly, CSS pseudo-classes are used to select style rules based on the widget state, allowing you to say style the selected tab with a different font color. For example, the following snippet specifies one color for general tab folders, but a different color for a selected editor:

```
CTabFolder {
  background-color: rgb(241, 240, 245);
  font: normal;
}
CTabFolder.editors:selected {
  background-color: rgb(255, 255, 255) rgb(255, 247, 229);
}
```

This use of CSS classes and pseudo-classes enables highly customized styling, both of particular workbench parts, and of particular part states. For example views can be tagged with *busy* or *updated* CSS class selectors when they are running background tasks, and the styling can describe what kind of presentation to associate with that state: different fonts, changed border, or even no styling at all if desired. The result is a more consistent application of GUI affordances and metaphors, making applications easier for users to understand and interact with.

Web to Desktop

The past few years has seen a blurring of distinctions between browser-based applications and traditional desktop applications. Browsers and the widget and language frameworks within them are beginning to approach the sophistication of desktop operating systems. At the same time,

desktop applications are becoming more web-enabled, and are adopting some of the user interaction and stylistic trademarks of web applications. While there will be demand for both traditional desktop and browser applications for some time, there is increasing demand for software *components* to be able to live in both of these worlds. In component-oriented software such as Eclipse, we no longer select a target platform and then build a monolithic software stack on top. We begin with a large collection of available components, and then stitch them together to build an application that suits our requirements. If we can reuse the same components in multiple runtime environments, it greatly reduces development and maintenance costs.

e4 is exploring component reuse across multiple target platforms and technologies in a number of ways. One such area is writing Eclipse components in JavaScript. As the de facto standard language for client-side browser programming, JavaScript is a good choice for anyone seeking to write components that will run in a broad set of runtime environments. To that end, e4 is investigating bringing both the benefits of Eclipse to the JavaScript world (modularity, extensibility, and tooling), and JavaScript components into the Eclipse desktop environment. While the current e4 focus is on JavaScript, the intent of this work more generally is to make it easier to write Eclipse components in a variety of different languages.

While JavaScript has long been used as a browser scripting language, it has rarely been used to build large scale applications like we see in the Java world. One weakness in this area is lack of a good modularity mechanism. There is no mechanism for defining and accessing namespaces, for expressing the dependencies of segments of JavaScript code, or for querying and consuming services defined in other components in an extensible way. In Java and Eclipse, we have the powerful and mature OSGi modularity system, among others, that satisfy these requirements.

To address these limitations, e4 includes a modularity framework based on OSGi for use in JavaScript applications. This allows creation of modular, large scale pure JavaScript applications, as well as integration of JavaScript bundles in a traditional Java-based OSGi runtime. Figure 5 illustrates the architecture of the e4 JavaScript framework and its relationship with pure JavaScript bundles as well as regular Java bundles.

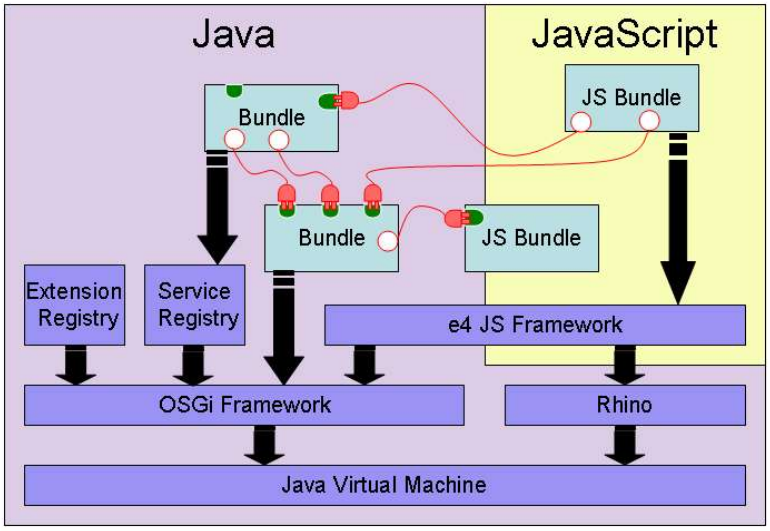

Figure 5 – e4 JavaScript framework

The e4 JavaScript framework is written in Java and runs as a pure OSGi bundle. It is responsible for parsing the manifests of JavaScript bundles and performing the resolution of dependencies between JavaScript bundles. The OSGi framework has no knowledge of JavaScript bundles or their dependencies - The JavaScript framework essentially runs as a complete nested framework within an OSGi instance. Dependencies between JavaScript bundles can be expressed at the bundle level (as with the `Require-Bundle` OSGi header), or at the level of individual script files (similar to the `Import-Package` OSGi header). These dependencies are expressed in the JavaScript manifest, written as a JSON file. Here is an example of a simple JavaScript bundle manifest:

```
{
  "Bundle-SymbolicName":"sample.jsbundle",
  "Bundle-Version":"1.0",
  "Bundle-ScriptPath":"script.js",
  "Import-Package":"a.resource;version=[1.0.0,2.0.0)",
  "Export-Package":"sample.resource;version=1.0.0",
  "Require-Bundle:"some.other.bundle",
}
```

A JavaScript bundle can be defined and installed into the framework programmatically, or the presence of a JavaScript bundle can be declared in a regular OSGi bundle using an additional manifest header:

```
JavaScript-Bundle: scripts/manifest.json
```

Although JavaScript and Java bundles can interact with each other by direct invocation, the recommended way to interact between the two worlds is via the OSGi service registry, or the Eclipse extension registry. For example, a JavaScript bundle can also declare itself as a regular OSGi bundle, and contribute a service to the OSGi service registry either programmatically or using declarative mechanisms such as OSGi DS. A regular OSGi bundle can then consume that service, without ever knowing the implementation of that service is written in JavaScript. Similarly, the e4 JavaScript framework provides an Eclipse extension factory for instantiating Eclipse extensions written purely in JavaScript. A JavaScript bundle can simply declare a `plugin.xml` file with their contributions to the extension registry, without writing a line of Java code. Bundles written in Java can then consume those extensions without knowing they are written in another language. Conversely, a JavaScript bundle can declare an extension point with some corresponding Java API that can be implemented by Java bundles, and then load and use extensions contributed to that extension point in JavaScript code. These types of interaction between JavaScript and Java bundles are illustrated in figure 5.

The e4 JavaScript framework itself defines API in both Java and JavaScript. Thus pure Java bundles can query or manipulate the JavaScript framework (for example install a new JavaScript bundle), and bundles written in JavaScript can interact with the framework through its JavaScript API. This is illustrated in figure 5 by the *e4 JS Framework* block extending across the Java/ JavaScript boundary.

The e4 programming model also facilitates integration of components written in other languages. With service-based interaction between components, and dependency injection, components never need to know what language services are implemented in, or what languages are being used to consume the services they expose. Similarly, the pared down Eclipse Application Services can be exposed as API stubs in other languages so that a broad segment of the Eclipse platform functionality can be quickly made available to other bundles written in other languages. A small JavaScript API for interacting with these core e4 services is currently available in the

bundle `org.eclipse.e4.ui.web`. As a proof of concept, e4 includes a re-implementation of the Plug-in Development Environment (PDE) update site editor written purely in JavaScript (illustrated in figure 6). This editor can be run in a web browser, or seamlessly integrated into the Eclipse platform user interface.

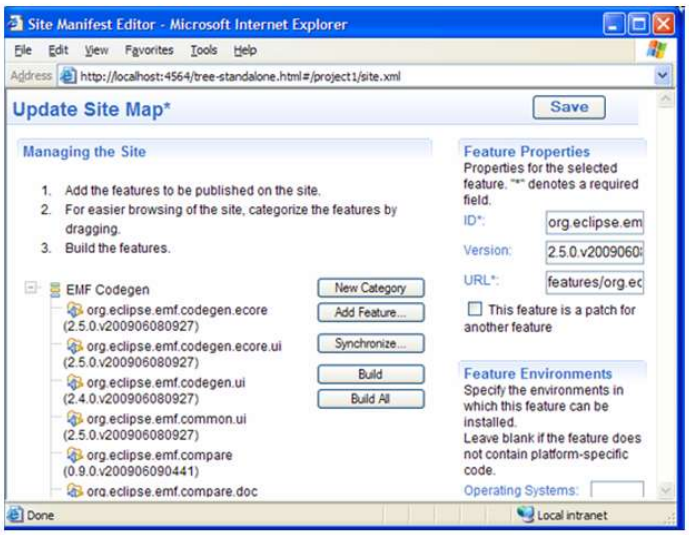

Figure 6 – e4 JavaScript site editor

Desktop to Web

We have shown how components built for the web can be integrated into the Eclipse platform. However, desktop/web interoperability can also be approached from the other direction. Applications written for the desktop using traditional enterprise languages such as Java can be ported to run on Web platforms. In e4 there are two areas of exploration for this kind of desktop-to-web integration: A new port of SWT, and the Eclipse Rich Ajax Platform (RAP).

The Eclipse Standard Widget Toolkit (SWT), provides a common API for graphical desktop applications across a wide range of operating systems and native widget toolkits. SWT allows developers to write an application once, and have it rendered with high performance and native platform look and feel on each target platform. Similarly, there are a wide range of programming languages and widget toolkits for web browser programming. This web technology landscape is changing rapidly, and application developers are reluctant to wholly embrace a single technology for fear of obsolescence or lock-in.

A new port of SWT in e4, called *SWT browser edition* (SWT/BE), aims to provide a common platform for web UI programming just as it has done for traditional desktop programming. This potentially allows existing SWT applications to run on the web, and allows developers to build web UI applications without being locked into a single web technology. SWT/BE currently supports ActionScript/Flex as a prototype example of the technology, but ports to other web platforms such as JavaScript/Dojo, Silverlight, Java FX, etc, are also possible. Figure 7 shows the SWT Control example, which contains all major SWT widgets, running on Adobe Flex.

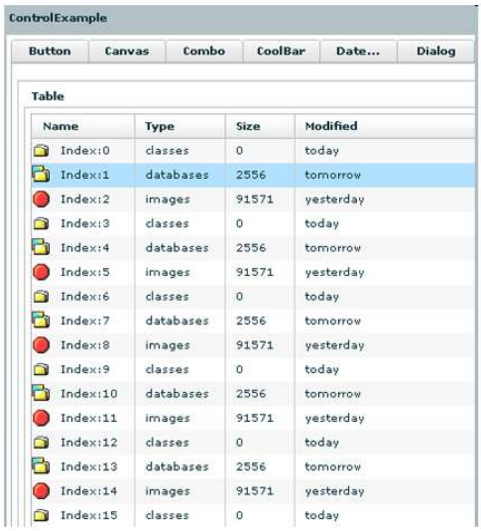

Figure 7 – SWT control example on Flex

The Eclipse Rich Ajax Platform (RAP) provides an implementation of Eclipse components such as SWT, JFace, and the Workbench UI that runs on the web. Similar to SWT/BE, RAP provides the opportunity for the same application to run on both the desktop and web with a common code base. However, the original RAP implementation required a fork of the Eclipse platform code to support the web target environment. In particular, web applications typically must support multiple concurrent user sessions in a single application instance, which is not generally supported by the Eclipse workbench due to the heavy use of singletons.

The e4 programming model is much more conducive to multiple concurrent sessions due to its service-oriented injection programming style. Early experiments with running e4 applications on the RAP runtime have been promising, with many fewer changes required to the workbench to support running on RAP. This RAP integration work provides validation that the e4 goal of supporting a wide range of runtime environments is attainable. Ongoing work with running e4 applications on target platforms such as RAP are helping to ensure the continued flexibility and openness of the e4 architecture and programming model.

Declarative Widgets

The modeled e4 workbench provides an abstract representation of the workbench itself: windows, pages, perspectives, view, etc. This model is then transformed into SWT using renderers and customized using the declarative styling engine. However, within individual workbench views the user interface is still constructed with plain SWT. This SWT code is often very repetitive, and hard-codes styling decisions such as font and margin sizes directly into the widget construction code. To avoid this repetitive and hard to customize SWT code, e4 is exploring ways of pushing the concept of model/renderer separation down into all places where SWT is used today. This exploration currently has two directions: XML-based widgets, and model-based widgets.

XWT: Declarative widgets in XML

XML UI for SWT (XWT), is a framework for writing SWT widgets declaratively in XML. In XWT, the complete structure of an application or widget hierarchy is expressed declaratively,

along with bindings of the widgets to some underlying application model or to Java-based call-backs implementing the widget behavior.

XWT takes declarative UI data as input, along with an application model that will be bound to the widgets at runtime. XWT includes a simple model for classes that conform to the JavaBean conventions of simple data accessor and setter methods. Additional models can also be defined and contributed to XWT via an extension point. The declarative UI data and model definition are combined by the XWT *UI generator* to produce the resulting SWT and JFace controls at runtime. This XWT architecture is illustrated in figure 8.

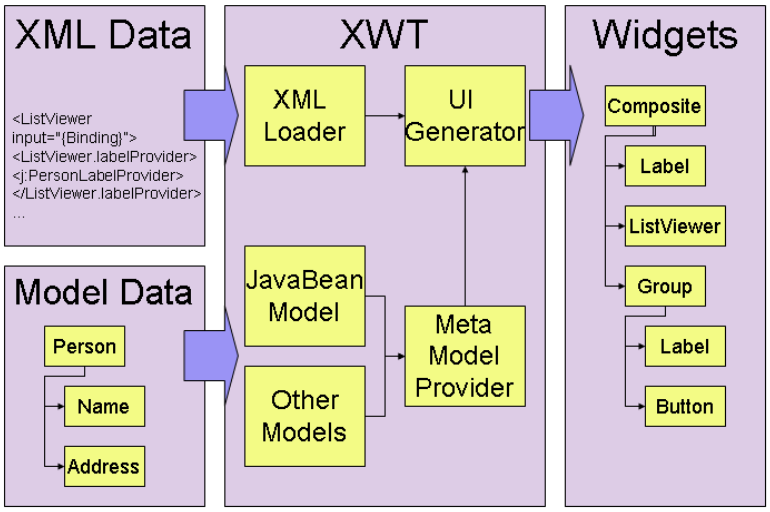

Figure 8 – XWT architecture

TM: Declarative widgets in EMF

The Toolkit Model (TM), is an abstract EMF model of user interface elements. This model is bound to a set of concrete widgets (such as SWT) at runtime. The TM builder maintains synchronization between this model and the concrete widgets as the application runs, propagating changes in both directions as required to keep the model and widgets synchronized. Applications typically interact at runtime with the toolkit model rather than the concrete widgets, resulting in a simpler abstraction for application developers to work with.

This higher level widget abstraction makes it possible to change the concrete widget implementation, either in subtle ways, or radical changes such as interacting with concrete widget instances running in a different process or a different physical machine. For example an application running on a web server can be manipulating a toolkit model living on the server, and the TM runtime can transparently implement that model with widgets running on a different machine, such as a browser-based web client.

Events in the toolkit model are handled by event handlers written in JavaScript, although Java can also be used. JavaScript interacts with the toolkit model in much the same way JavaScript manipulates an HTML document object model (DOM) in a web browser. Since graphical tools are available for building and manipulating EMF models, this combination of EMF and JavaScript allows for rapid prototyping of application design and behavior.

Figure 9 illustrates the Toolkit Model architecture. A *builder* is responsible for creating widgets and binding them to the model. The builder does this by finding a suitable *binder* for each kind of widget being constructed. The binder constructs the concrete widgets, and manages the synchronization between the widgets and model elements after construction. There is one binder for each kind of widget, rather than one binder per widget instance. JavaScript code supplies the model with behavior, both callbacks from widget events and other application behavior.

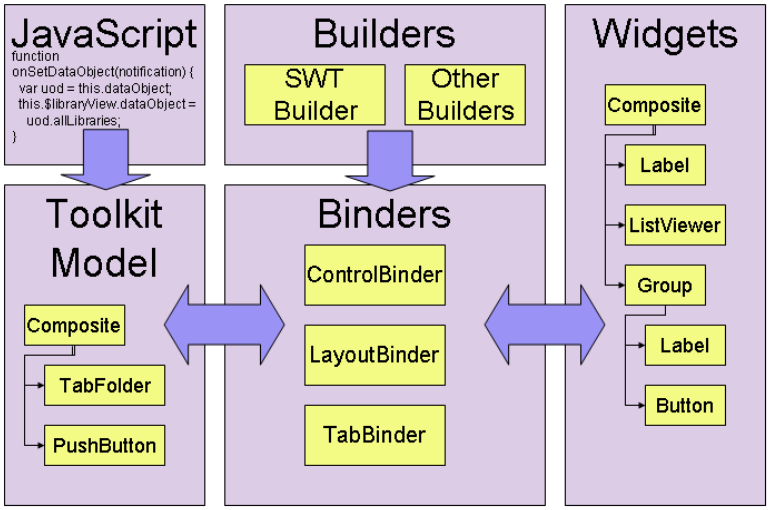

Figure 8 – Toolkit Model architecture

Flexible Resource Model

While much of e4 is directed at the Eclipse platform as a general-purpose application framework, development tools remain an important part of the Eclipse eco-system. One area many IDE developers found wanting in the previous generation of Eclipse was its support for more complex project layouts. Development tool users often have well established layouts of their source code and other development resources, and bringing these layouts into Eclipse-based IDEs was often challenging due to the rigidity of the Eclipse resource model. e4 includes a new enhanced resource model that provides better support for importing and managing more complex project layouts in Eclipse workspaces. Some enhancements in this model include:

- The ability to define path variables at the project level, and create linked resources relative to those project-specific path variables. This allows projects containing complex link structures to be moved around without breaking links.
- New virtual folders called *groups*. Groups don't exist in the file system, but they can be used to create arbitrarily complex virtual project trees containing links to concrete resources elsewhere on disk.
- Support for filters on projects and folders, which exclude specific resources or patterns of resources from the Eclipse workspace tree. Filters allow an Eclipse project to include folders that contains thousands of elements, but only include a fraction of them in the project, with no extra memory overhead for the excluded resources.

- Creation and manipulation of linked resources via drag and drop. When dragging a file tree from outside Eclipse, you can now drop the tree into the Eclipse workspace as a virtual tree made up of links and groups, rather than as a concrete file system tree.
- The ability to edit the location of existing linked resources, or to convert links between absolute or variable-relative paths.

Together these enhancements allow users to quickly set up complex project structures in the Eclipse workspace, and share those projects with other users while keeping the virtual project structure intact.

Conclusion

The e4 project introduces a broad set of new technologies that modernize the architecture and design principles of the Eclipse platform. These changes should position the Eclipse platform well for the future, opening up the platform to support more programming languages and target runtime environments. Components designed for e4 will be easier to customize, configure, and reuse in different applications without modification. The separation of style and presentation logic from application logic will allow Eclipse applications to be easily skinned in a consistent way. Developers building components and applications in accordance with e4 design principles will be more insulated from technology changes in underlying operating systems and web browsers, and will gain portability and flexibility above and beyond the abstraction provided by the Java virtual machine.

For more information, visit http://eclipse.org/e4, or the e4 wiki page at http://wiki.eclipse.org/e4.
