Embedded Device Solution Life Cycle Support with Eclipse

ChangWoo Jung
IBM Ubiquitous Computing Laboratory
The MMAA Building, 467-12 Dogok-dong,
Gangnam-gu, Seoul, Korea
jungcw@kr.ibm.com

Han Chen
IBM T.J. Watson Research Center
19 Skyline Dr, Hawthorne, NY 10532
chenhan@us.ibm.com

ABSTRACT

The effectiveness of life cycle management of an embedded device solution is crucial to its value proposition. This paper shows that Eclipse technology can be used for both tooling and runtime support of device solutions using a visual flow language. We introduce a prototype implementation based on Eclipse 3.2 and Equinox and demonstrate that it helps a user in each stage of a solution life cycle.

1. INTRODUCTION

Computerized sensors and actuators are used in diverse applications such as industrial automation, asset management, smart office/living space, healthcare, etc. For example, a smart office [5] may be laden with sensors which measure ambient light, temperature, humidity, human presence, etc., and computer-addressable actuators such as light switches, dimmers, Heating, Ventilation and Air Conditioning (HVAC) blowers, dampers, and so on. In retail industry passive RFID technology is used to improve the efficiency of supply chain. RFID-enabled receiving portal is one example. It is a dock door integrated with motion detectors for sensing the presence of incoming pallets, RFID readers for acquiring IDs from the tagged cases, and a light stack for indication of operational status.

In order to carry out functions desired by users, these sensor and actuator devices must be coordinated. This is usually accomplished by connecting the devices to a controller node via wireless links or wired connections, as seen in Figure 1. The controller node executes certain application logic so that the office space is customized according to the occupant's preference and the receiving portal alerts the operator automatically when unexpected shipment occurs. We generally refer to the application logic as an *Embedded Device Solution*.

How easily an embedded device solution can be created, deployed, and managed has a direct impact on its value to the user. We have proposed Rapid Integrated Solution Enablement (RISE) as a framework for managing the life cycle of embedded solutions [4]. We argued that a high-level visual programming model is better suited for these solutions than textual languages such as Java. As a result we created the Graphical Composition Language (GCL), a model-driven component-based programming language, to describe the behavior of an integrated device (Figure 1). Based on Java, GCL defines the component interface in high level terms such as ports and parameters. Developers can create primitive (or atomic) components using the Java base classes or construct composite components from other components hierarchically.

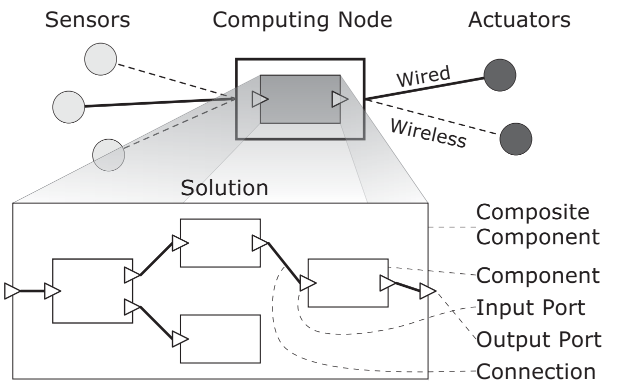


Figure 1: An integrated embedded device solution

While [4] discusses the design methodology and architecture of RISE, this paper focuses on how Eclipse technology is used to support the entire life cycle of a solution. The rest of the paper is organized as follows. Section 2 discusses the life cycle of an embedded device solution in general and how Eclipse technology fits in the picture. The updated implementation of RISE is described in Section 3 and Section 4 shows how the tool is used. Section 5 concludes the paper and points out some future work.

2. SUPPORTING EMBEDDED DEVICE SOLUTION LIFE CYCLE WITH ECLIPSE

Like any piece of software code an embedded device solution passes through several stages from the time of being conceived to that of decommission. During its life cycle there are different roles involved in each stage. A well designed tool should support the entire life cycle of a solution so that all user roles can perform their tasks easily.

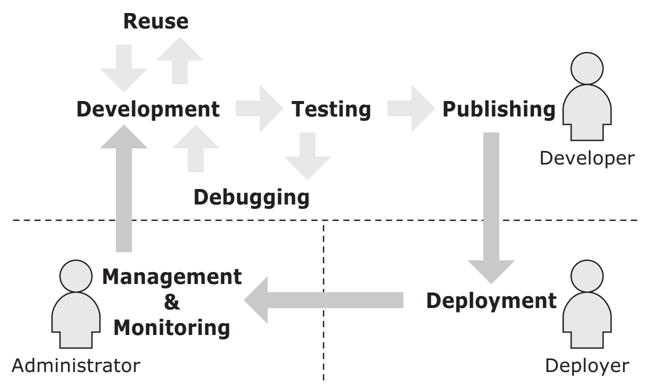


Figure 2: Life cycle of an embedded device solution

2.1 Life Cycle of a Solution

Following a model-based development methodology to encourage software reuse we partition the life cycle of an embedded device solution into three broad stages: development, deployment, and management, as shown in Figure 2. Some stages can be further decomposed into smaller steps, as outlined in the following list.

Design and Development During this stage a developer creates reusable components or an entire solution.

Reuse During the development stage a developer may choose to import existing components or solution templates instead of re-implementing them from scratch. These components may come from various external sources, for example, local file systems, company-wide repositories, or Internet update sites.

Testing During development the developer tests a solution to verify that it is functioning correctly. Testing can be achieved with either a model-driven formal verification tool or via a simulator which offers quick visual feedbacks but lacks rigorous proof. Development and testing typically alternate; this loop will go on for multiple iterations before the solution is finalized.

Debugging If problems are uncovered during testing a developer may need to debug a solution interactively in order to locate any design flaws and correct them.

Publishing After a developer finishes a solution or a component that is deemed general enough to be reusable, he publishes the solution or component to some external destinations. This allows it to be reused by other solutions later.

Deployment During deployment a deployer, who can be different from the developer, installs a solution on a target runtime that controls the devices.

Management After a solution is installed, an administrator supervises its execution by performing management tasks such as pausing, stopping, restarting, reconfiguring parameters, etc.

Monitoring While a solution is running an administrator monitors its health and performance status, for example, message throughput, latency, etc. When the administrator encounters problems or the user requirements cannot be met by the current solution, he requests another iteration for solution upgrade, which leads to the design and development stage again.

2.2 Eclipse's Role

Eclipse is an open-source software that is gaining wide acceptance in the software engineering community. At its core Eclipse is an extensible plug-in management engine. A plug-in is a software module that provides a set of features according to a well-defined specification of an extension point. A plug-in can define extension points thus allowing its functionality to be extended by other plug-ins. This modular architecture gives developers opportunities to create an endless variety of tools.

Most of tools rely on the workbench plug-in for user interface; it defines the extension points of editor, view, and perspective. A tool maker can create an editor plug-in for a specific file type in a workspace. He can use view plug-ins to offer detailed representations of key aspects of a development process. A perspective plug-in allows the tool to organize the myriad of editors and views in logical groups to target different user roles.

Its perspective-based UI, extensible plug-in architecture, and other supporting tool projects such as Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF) make Eclipse the perfect tooling platform for embedded device solution life cycle support.

Since version 3.0 Eclipse has incorporated the Open Services Gateway Initiative (OSGi) Service Platform [2] in its runtime architecture. The specifications of OSGi define how service modules, or bundles, are managed inside a Java virtual machine. With OSGi bundles can be installed, started, stopped, updated, or uninstalled dynamically. This provides a flexible and powerful mechanism to any application that requires modularity and collaboration among its components. Therefore we choose Equinox, the Eclipse OSGi implementation, as the runtime for an embedded device solution, which should result in easier component management and better interoperability.

3. PROTOTYPE IMPLEMENTATION

The RISE software architecture consists of three main platforms: tooling, runtime, and component repository, as shown in Figure 3. The tooling platform provides an integrated development environment (IDE) for developers to build, deploy, and manage embedded device solutions. The runtime platform executes the solutions. The repository allows components and solutions to be shared and reused at build-time and dynamically loaded at run-time.

The original proof-of-concept prototype in [4] used Eclipse 2.1 for tooling support only. We have since updated the implementation to leverage Eclipse 3.2 as both the tooling and runtime technology provider.

3.1 Development Tool

A unit of development is called component. A component can be either atomic or composite. An embedded device solution typically takes the form of a composite component. A composite component is created as an event flow diagram written in GCL with an optional state machine that controls the flow. This model is captured and persisted in a

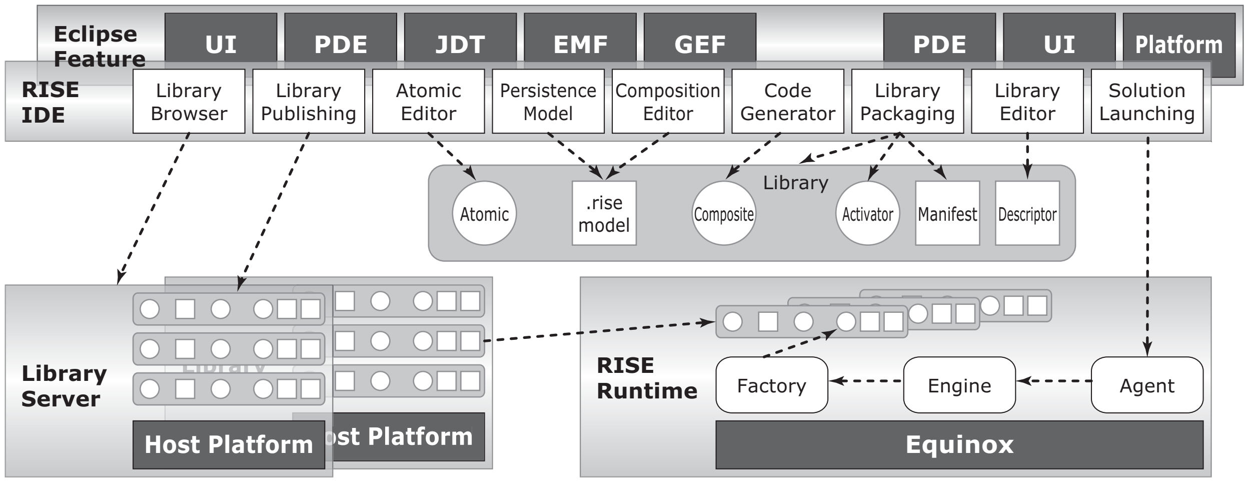


Figure 3: Software architecture of RISE

description language; a code generator is used to transform it into executable Java code. Logically related components are packaged into a library. Library is the unit of publishing and dynamic loading.

The IDE consists of several major functional blocks, many of which are implemented with support of Eclipse platform and other related tool projects. Details are shown in Figure 3.

Persistence model An EMF ecore model is created for the GCL. The resulting EMF classes provide an persistent format in XMI and serializers and deserializers for the model.

Composition editor The composition editor allows a user to edit a composite component visually, for example, creating ports and parameters, making connections between ports, importing existing components from a library server, etc. The editor is based on GEF, which provides visual layout and rendering of graphical objects.

Atomic editor There is no special plug-in for this part, but we provide a set of base classes for a developer to create atomic component using Java Development Tools (JDT).

Code generation Whenever a composite component model is modified and saved, the incremental builder invokes the code generator which transforms the model into Java classes.

Library browser This is a view that enables a developer to browse existing components published on library servers.

Library packaging We map the library concept into Eclipse plug-in (or OSGi bundle) and leverage Plug-in Development Environment (PDE) to create bundle structure and manifest file. The tool then inserts markers in the manifest to indicate that it is a RISE library. It also creates special bundle activator that assists the dynamic loading of components. A descriptor file is also packaged in the root folder; it lists the components available in the library.

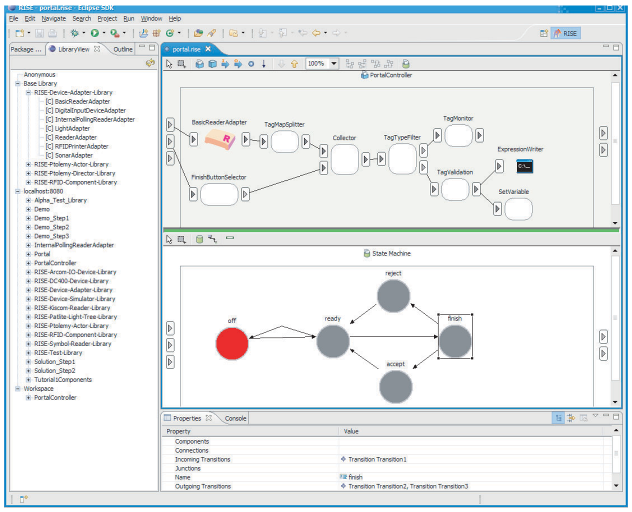


Figure 4: User interface of RISE IDE

Library editor This tool is used by a developer to manually manipulate the library descriptor file.

Library publishing This tool creates a JAR for a library and submits it to target library servers.

Solution launching This enables a developer to launch a solution (component) on a target runtime.

The user interface of the IDE is shown in Figure 4. It shows the RISE perspective, which contains the composition editor, the library browser, and the property sheet. Not shown in the screenshot are various wizards and dialogs that assist a developer throughout the design process.

3.2 Runtime Platform

The runtime platform is based on Equinox, the Eclipse implementation of OSGi specification. The three major runtime components, Agent, Engine, and Factory are all packaged as bundles (see Figure 3). The agent receives commands from outside and controls the runtime. The engine executes the generated solution code. The factory handles the dynamic loading of components.

A component is uniquely identified by a Uniform Resource Identifier (URI) which includes a protocol, a library name, and a component name. Because libraries are packaged as bundles, the factory first uses a protocol handler to load the library binary into the runtime with OSGi support. It then delegates the instantiation of component to the library's activator.

This provides a convenient, dynamic, and distributed mechanism of composition during run-time. In particular, the child components of a solution need not be loaded from the same library, nor do all the libraries have to reside on the same library server.

3.3 Library Server

A library server serves as a repository of reusable components and solutions. During the deployment phase it delivers libraries to a runtime in response to requests coming from the Factory (see Figure 3).

A desirable feature of the library server is automatic prerequisite resolution, that is, generating a list of bundles (or libraries) a requested library depends on. This simplifies the runtime management of libraries. It can be computed by inspecting the manifest of a library for any import packages and services.

In this implementation we continue to use the bundle server from Service Management Framework (SMF) [1] as the library server. We extend the Equinox bundle installation mechanism to support automatic pre-requisite resolution by creating a custom protocol handler. It takes advantage of the server-side dependency calculation feature of the SMF bundle server. Alternatively we could build a protocol handler that computes the dependency at client side; this would be useful for less sophisticated library server platform such as a web server.

4. USING RISE

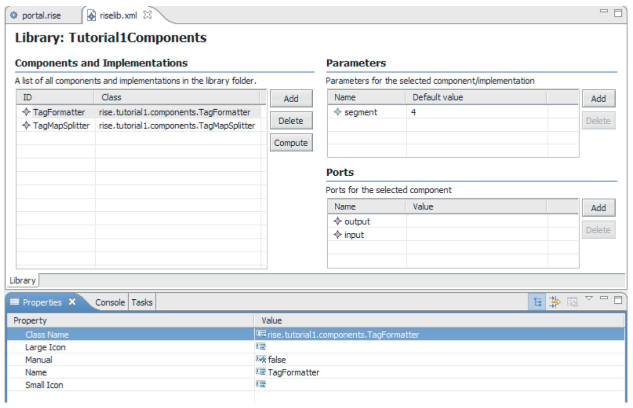
In this section we demonstrate how the RISE tool is used throughout the life cycle of an embedded device solution.

4.1 Creating a Library

After switching to the RISE perspective, the user creates a new RISE project with the creation wizard. A RISE project is bound with RISE nature, which uses an incremental project builder to generate code from models.

The user can convert a RISE project or a source folder within it into a library folder by using a wizard. The wizard generates a manifest file (`META-INF/MANIFEST.MF`), a library descriptor file (`riselib.xml`), and a bundle activator class. The manifest contains special markup attributes so that the bundle can be recognized as a library.

The library descriptor editor, as shown in Figure 5, is opened when the user double-clicks on the library descriptor files. It shows a list of components contained in this library and the ports and parameters of each component. A search feature is included in the editor to discover manually created atomic components.



Library: Tutorial1Components

Components and Implementations

A list of all components and implementations in the library folder.

ID	Class
TagFormatter	rise.tutorial1.components.TagFormatter
TagMapSplitter	rise.tutorial1.components.TagMapSplitter

Parameters

Parameters for the selected component/implementation

Name	Default value
segment	4

Ports

Ports for the selected component

Name	Value
output	
input	

Property	Value
Class Name	rise.tutorial1.components.TagFormatter
Large Icon	
Manual	false
Name	TagFormatter
Small Icon	

Figure 5: Library editor

4.2 Creating an Atomic Component

An atomic component is implemented in Java using the RISE component interface. The user creates an atomic component just like he would a regular Java class. In the Java class creation wizard the user specifies `AtomicComponent` as the super class. JDT creates a skeleton code and the user has to fill in unimplemented methods. Ports and parameters are declared as member variables of the class.

4.3 Creating a Composite Component

The user uses the new RISE component wizard to create a composite component. After specifying the destination folder and file name, a new `.rise` file is created in the designated package. The associated composition editor is also opened. The user can then create and modify the event flow diagram by adding children components, creating ports and parameters, and making connections between ports. When the editor is saved, the code generator will create a Java file from the model in the same package that the `.rise` resides.

The GCL allows a composite component to be either stateless or stateful. When a composite component is created it is stateless by default. To give a component stateful behavior, the user clicks the "State Machine Editor" button on the toolbar of the composition editor. A state machine is then created and associated with the current composite component. The "State Machine Editor" appears as a pane at the bottom half of the editor (see Figure 4). The user can create states and transitions among them on the drawing canvas by using the proper toolbar buttons. The user can create a different event flow diagram for each state using the editor.

4.4 Browsing Library Servers

The library browser view, shown in Figure 4, shows all the components that are available for reuse in a composite component. It is a tree viewer with several top-level branches. The first one is "Anonymous", which contains all the anonymous inner components nested in the current composite component being edited; this branch is only visible while editing a compositing component. The second one is "Base Library", which is a list of standard components that come with RISE. The third one is "Workspace", which contains a list of all RISE library folders in the current workspace. Within each library folder there is a list of components. Each one of the rest of the branches represents a library server. Under each library server node there is a list of libraries that are discovered from it.

The user can add new library server locations to the list or remove existing ones from it.

4.5 Reusing a Component

With the library browser view open a user can simply drag a component from the browser view and drop it onto the drawing canvas for an composite component. Because the library descriptor file contains the description of a component including its ports and parameters, when it is dropped onto the editor canvas correct ports and parameters are shown.

4.6 Publishing a Library

After the user has finished developing a component or a solution he may want to make it available for reuse. He can submit the library using the library submission wizard. The target library server locations are selected in the wizard. The tool then generates a JAR for the library and sends it to the library servers.

After refreshing the library browser view the user will notice that the newly added library is shown under the target library server folder. It is now ready to be used by others.

4.7 Starting the Runtime

Before a solution can be tested or deployed on a device the RISE runtime needs to be prepared. The user launches the Equinox OSGi framework on the target device and installs the runtime bundles that constitute the Agent, the Engine, and the Factory, as described in Section 3.2. After these bundles are installed and started the runtime is ready to be managed by the IDE.

To date the runtime software with Equinox has been tested on Windows and embedded Linux on Arcom Viper.

4.8 Running a Solution

We use the standard Eclipse run wizard to manage the running of solutions. The user creates a new run configuration for each solution he intends to deploy. He specifies the URI of the solution, the address of the target runtime, and several other options in the dialog page. If the solution has parameters, the user can set their values in the same dialog.

After making sure that the target runtime is properly prepared and all necessary libraries are submitted to a library server, the user can activate a run configuration. The wizard sends the URI and parameter values to the agent on the target runtime, which then loads and starts the solution.

4.9 Configuring a Solution

While a solution is running its parameters can be dynamically changed to modify its behavior. This is done through a web application hosted by the runtime.

5. CONCLUSION AND FUTURE WORK

Nowadays many applications call for the integration of sensors and actuators to create intelligent composite devices. The effectiveness of managing the life cycle of an embedded device solution can have a direct impact on the total cost of ownership of such devices. In this paper we have showed that Eclipse technology can play a significant role in the lifecycle of such solutions. We have created a prototype system consisting of an IDE, a runtime, and a repository based on various Eclipse related technologies and have demonstrated that it can simplify the creation, deployment, and management of solutions.

As we continue to refine the current design and explore new opportunities in this space, we have identified some additional enhancements. We consider them as future work.

Event monitoring Monitoring a solution as it runs can provide valuable insight into its performance and help identify latent design flaws. We envision that the composition editor can be enhanced to support a monitor mode, in which an administrator can dynamically attach watch windows to ports or components and monitor events as they occur.

Debugging Although there is a debugging feature available in JDT, it has very little value when a solution is created using the flow language. A much more powerful visual debugger would allow a developer to set breakpoints on components or ports with filtering conditions.

New library server The component libraries are ideally comprised of a local library and a remotely hosted, pooled component library. The remote component library would be beneficial for the developer to find and leverage pre-built components. This vision is analogous to the OSGi Bundle Repository (OBR) concept [3]. Supporting OBR as a library platform would make the system more open and interoperable.

6. ACKNOWLEDGMENT

This work is partially supported by the IT839 project from the Institute of Information Technology Assessment and Ministry of Information and Communication in Republic of Korea. We would also like to thank all the team members who have contributed to the design and implementation of the prototype system: Paul Chou, JeongAh Jang, Andy Kim, SooYeon Kim, KangYoon Lee, Sam Lee, Johnathan Reason, JiHye Rhim, YongHoon Um, and Danny Wong.

7. REFERENCES

[1] IBM. Service Management Framework. `http://www-306.ibm.com/software/wireless/smf/`.

[2] OSGi Alliance. OSGi Service Platform, Release 4 Specification, 2005. `http://www.osgi.org`.

[3] OSGi Alliance. RFC-0112 Bundle Repository, 2006. `http://bundles.osgi.org/rfc-0112_BundleRepository.pdf`.

[4] J. Reason, H. Chen, C.-W. Jung, S.-W. Lee, D. Wong, A. Kim, S.-Y. Kim, J.-H. Rhim, P. Chou, and K.-Y. Lee. A Framework for Managing the Solution Life Cycle of Event-Driven Pervasive Applications. In *Proceedings of IFIP International Conference on Embedded and Ubiquitous Computing*, 2006.

[5] S. Yoshihama, P. Chou, and D. Wong. Managing Behavior of Intelligent Environments. In *Proceedings of IEEE International Conference on Pervasive Computing and Communications*, 2003.