Best Practices for Programming Eclipse and OSGi

BJ Hargrave
Jeff McAffer

IBM Lotus
IBM Rational Software
Introduction

- During the Eclipse 3.0 and OSGi R4 development cycles, OSGi and Eclipse began to influence each others work
  - Eclipse 3.0 adopted the OSGi framework as the “footing” of the Eclipse platform
  - OSGi R4 incorporated features to support important Eclipse use cases
- This led to the OSGi R4 Framework specification of which Eclipse Equinox is an implementation
  - The Equinox code is currently being used by OSGi as the framework reference implementation for R4
Overview

- As a result of the incorporation of the OSGi framework into Eclipse, there are several additional capabilities now available to Eclipse plug-in writers that should be considered.

- Today we will look at two areas:
  - Modularity – Techniques for sharing classes and resources between bundles (aka. plug-in)
  - Collaboration – Techniques for inter bundle collaboration (which build upon class sharing)
Modularity

- “(Desirable) property of a system, such that individual components can be examined, modified and maintained independently of the remainder of the system. Objective is that changes in one part of a system should not lead to unexpected behavior in other parts.”
  www.maths.bath.ac.uk/~jap/MATH0015/glossary.html

- We need to be able to share classes and resource between bundles (modules) while supporting a proper level of modularity

- Eclipse and OSGi offers two main ways of sharing
  - Require-Bundle
  - Import-Package
Require-Bundle

- Mechanism for a bundle to gain access to all the packages exported by another bundle – “bulk import”

- Advantages
  - Can be used for non-code dependencies: e.g. Help
  - Convenient shorthand for multiple imports
  - Joins split packages

- Disadvantages
  - Tight coupling – can be brittle since it requires the presence of a specific bundle
  - Split packages – Completeness, ordering, performance
  - Package shadowing
  - Unexpected signature changes
Import-Package

- Mechanism for a bundle to import specific packages

- Advantages
  - Loose coupling – implementation independence
  - Arbitrary attributes allow sophisticated export matching
  - No issues with package splitting or shadowing – whole package

- Disadvantages
  - More metadata to be created and maintained – each imported package must be declared
  - Only useful for code (and resource) dependencies
  - Can’t be used for split packages
Best Practices: Modularity

- In general, Import-Package is recommended
  - PDE (or other tools) can help with metadata management for packages used
  - Loosest coupling
    - More opportunities for resolver to successfully resolve
  - Provides more information to management system

- Require-Bundle used for complex scenarios
  - Refactoring bundles which results in splitting a package across more than one bundle
  - Have dependencies on a specific bundle and version
    - This could still be done with Import-Package
    - Also is a simple place to start when first modularizing legacy code

- To some degree, the choice is a trade off that you must make
  - Simplicity vs. flexibility
Collaboration

- Modularization is powerful
  - Decouples elements
  - More flexible configurations
  - Dynamic behavior

- Decoupled components need a way of interacting and cooperating
  - Our resolved bundles need to collaborate

- Eclipse includes three mechanisms for inter bundle collaboration
  - Extension registry
  - Service registry
  - Declarative Services – builds upon the service registry
Extension Registry

- *Extension Registry*: declarative relationships between plug-ins
- *Extension Point*: plug-ins open themselves for configuration/extension
- *Extension*: plug-in extends another by contributing an extension

“Plug-ins can contribute *actionSets* extensions that define actions with an id, a label, an icon, and a class that implements the interface *IActionDelegate*. The UI will present that label and icon to the user, and when the user clicks on the item, the UI will instantiate the given action class, cast it to *IActionDelegate*, and call its *run()* method.”
Extension Registry

- The extension registry provides a per extension point list of contributed extensions
  - Aggregates all the extensions for the extension point
  - Provides a “private context” for the extension point and its extensions
    - Only the extension point will call the extension
- Tightly coupled model
  - Each extension is bound to a specific extension point
  - Extension point are no longer bound to specific bundles
- Declarative – plugin.xml
- Lazy loading of extension class
  - Metadata enables registration and attribute interrogation
- Life cycle scoped to resolved state of bundle
- Lifecycle is highly dynamic
  - Extension may be published or unpublished at any time (after bundle resolved)
  - Lifecycle event notifications
- No security to control
  - Which bundle can declare an extension point
  - Which bundle can contribute an extension
Service Registry

- The service registry is a publish/find/bind model
- Public context
  - Single service registry (within a framework instance)
- Loosely coupled model
  - Any bundle can bind to a service
- API based – non declarative
  - Service can be published with key/value pair metadata
- Eager loading of service class
  - Service object is published
- Life cycle scoped to started state of bundle
- Lifecycle is highly dynamic
  - Service may be published or unpublished at any time (after bundle started)
  - Lifecycle event notifications
- Permissions to control whether a bundle can publish, find or bind to a service
Declarative Services

- Declarative service model built upon service registry
  - Adds a declarative mechanism specifying
    - Provided service
    - References services
  - Simplified programming model
    - POJO with dependency injection and contextualized lookup
  - Can conceal dynamism of services from programmer
  - Lazy loading of service class
  - Lifecycle managed by central runtime
  - Interoperates with service registry

- Vaguely similar to Spring but supports the dynamic component model of OSGi and Eclipse
Extensions

extension point

uses

contributes

extension

implements

Services

uses

implements

binds

consumer

service

uses = contract

consumes = consumer

provides = provider
Example: Event Listeners

- Event listeners are connected to the event source.
- An event is fired and the event source must notify each event listener.
- Event listeners can supply metadata describing interest in event, for example:
  - Event subtypes of interest
  - Frequency of notification
  - etc.
Extension Approach

- Event source
  - Declaratively defines and exposes extension point (e.g. `eventSource`)
  - Defines the listener interface (e.g. `IEventListener`)

- Listener
  - Implements `IEventListener`
  - Declaratively contributes extension for the `eventSource` extension point that defines
    - `IEventListener` class to run
    - Listener metadata

- Event source extension point discovers each registered `IEventListener` extension

- When an event is fired, the event source
  - Evaluates the event against each listener extension’s metadata
  - Can then load and run the listener’s code to deliver the event
Extension Approach

Event Source

- `eventSource`
- `IEventListener`

Contributes:

- `Extension`

Implements:

- `ListenerImpl`

Instantiates calls to notify
Services Approach: Whiteboard

- Event source
  - Defines the listener interface (e.g. IEventListener)

- Listeners
  - Implements IEventListener
  - Registers an IEventListener instance as a service with properties containing listener metadata

- When an event is fired, event source
  - Finds all registered IEventListener services
  - Evaluates the event against each listener service’s metadata
  - Can then bind to and run the listener’s code to deliver the event
Services Approach: Whiteboard

- Service Registry
  - finds
  - instantiates publishes

- Event Source
  - IEventListener
  - binds
  - implements
  - calls to notify

- ListenerImpl
  - Listener
Services Approach: Registration

- Event source
  - Defines the event source interface (e.g. IEventSource)
  - Defines the listener interface (e.g. IEventListener)
  - Implements IEventSource
  - Registers an IEventSource instance as a service

- Listeners
  - Implements IEventListener
  - Finds and binds to the IEventSource service
  - Registers an IEventListener instance with the IEventSource service along with listener metadata

- When an event is fired, event source
  - Evaluates the event against each listener’s metadata
  - Can then run the listener’s code to deliver the event
Services Approach: Registration

- Event Source
- ListenerImpl
- Service Registry

- IEventSource
- IEventListener
- Listener

- Service Registry finds
- instantiates publishes
- Service Registry registers
- Service Registry implements
- calls to notify
Declarative Services Approach: IoC Whiteboard

- **Event source**
  - Defines the listener interface (e.g. IEventListener)
  - Declaratively defines component with a dynamic, 0..n cardinality, dependency injection reference to IEventListener services

- **Listeners**
  - Implements IEventListener
  - Declaratively defines component providing IEventListener service with listener metadata

- **SCR will create and inject instance of IEventListener service into event source component**

- **When an event is fired, event source**
  - Queries listener for metadata and evaluates the event against metadata
  - Can then run the listener’s code to deliver the event
Declarative Services Approach: IoC Whiteboard

- **Service Component Runtime**
- **Event Source**
  - **IEventListener**
  - **ListenerImpl**
  - **Listener**

- **setter injection**
- **instantiates**
- **implements**
- **calls to notify**
Discussion

- In Extension and Service Registration approaches, the listener is able to select the specific event source with which it’s listener will be registered
  - Names extension point of extension/binds to event source service
- In the Service Whiteboard and Declarative Services, control is inverted and the event source select the listener
- Extension approach allows lazy loading of listener class
- The Services approaches require eager loading of listener class
- The Declarative Service with DI also requires eager loading but a variation can be made which allows lazy loading at the “expense” of using container API
  - Contextualized lookup
  - Injection of ServiceReference
Compare and Contrast

- **Extensions**
  - Private contract with specific consumer (extension point)
  - Can deliver data-only payload
  - Lazily loaded and run
  - Lifecycle scoped to resolved state of bundles

- **Services**
  - Public contract
  - No data-only payload
  - Eager loading
  - Lifecycle scoped to started state of bundles

- **Declarative Services**
  - Public contract
  - No data-only payload
  - Lazily loaded and run
  - Lifecycle scoped to started state of bundles
Best Practices: Collaboration

- Extension Registry
  - Use when a tightly coupled relationship exists such as contributing UI elements

- Declarative Services
  - Use when providing a service usable by any consumer (loosely coupled relationship) such as a data validation service
  - Use when substitutability of service providers and consumers is desired

- Service Registry
  - Same as Declarative Services
  - But Declarative Services is preferred unless you have a complex need outside scope of Declarative Service’s capabilities
  - Useful for highly dynamic service such as publication upon external event