Modularity with Eclipse RCP

Martin Bayer & Tomáš Vejpustek

May 13, 2015

CGI

Experience the commitment®

Outline

Basic understanding of RCP:

1. What is Eclipse RCP?
2. Dependency injection, inversion of control
3. Application architecture
4. Communication
5. Modularity with plugins

Eclipse Rich Client Platform

- software framework
- thick clients with GUI
- not plugins for Eclipse IDE
- layout looks like Eclipse IDE ☞

Dependency Injection

= means of accessing software libraries

1. I need service
2. I ask framework for interface
3. framework finds (and initializes) implementation

- implementations are interchangeable

Dependency Injection in RCP

- based on OSGi
- annotations ☞1
  - fields, constructors – `@Inject`
  - methods – behavioral (`@PostConstruct`, . . .)

Context

= storage for injectable objects

= mechanism of searching injectable objects

“Let’s put this object into context”

Pros and Cons of Dependency Injection

- + independence on concrete implementation
- + easy testing
- + independent development (only need API)
- + reduces boilerplate dependency obtaining
- − difficult to trace and debug (document well!)

Inversion of Control

“Don’t call us, we’ll call you.”

$\approx$ software framework

- DI is one implementation
- user writes snippets and framework calls them
- user writes GUI elements and framework places them
- when/where $\Leftarrow$ configuration (XML files)

Application Model

- basis of RCP framework
- Java classes assigned to elements
- behaviour ruled by annotations
- elements ☞ 2
  - graphical: window, perspective, part
  - commands, handlers, menus, tool items
  - addons (provide services)
- can be changed dynamically (`EModelService`) ☞ 4

Commands and Handlers

Command abstract action (save, . . .)

Handler actual implementation

- IoC in practice ☞3
  - ❶ button calls command
  - ❷ framework finds closest handler
- different "save" for different editors
- handlers can be called from code ☞3

Break: Questions?

CGI

Experience the commitment®

Communication

How to pass information/objects between modules/plugins?

Communication in RCP:

- DI-based

– weird, hard-to-trace errors (missing producer)

+ great decoupling

Eclipse Context

- basis of DI in RCP
- storage of objects

Passing objects ☞ 5

- send using `IEclipseContext` (from DI)
- receive via DI (can listen)

used as	$\texttt{Class}\langle\texttt{T}\rangle \rightarrow \texttt{T}$
implemented as	$\texttt{String} \rightarrow \texttt{Object}$ (`@Named`)

Context Hierarchy

- own context for app model elements – `MContext`

  application ∋ window ∋ perspective ∋ part; popup menu

- DI look-up

  ❶ search current context
  ❷ not found ⇒ search parent context; repeat
  ❸ not found in application context ⇒ `null`/error

- Use common context for communication! ☞ 6

Event Service*

- global messages (context-independent)
- stringly typed (message topics)

Passing objects ☞ 7

- send using IEventBroker (from DI)

  send (sync), post (async)
- receive via DI

  @EventTopic, @UIEventTopic (UI thread)

Context vs Event Service*

- objects vs actions
- persistence vs differentiation
- persistence important for inactive parts

I need to distinguish actions ⇒ use Event Service
I need to retrieve it later ⇒ use Context
I need both ⇒ use both ☞ 7

- use context for parts and handlers
- handlers do not need notification [!]

Selection Service*

- context service
- active GUI selection
- window- and part- specific

Passing objects ☞ 8

- send using `ESelectionService.setSelection`
- receive via DI: `@Named(IServiceConstants.ACTIVE_SELECTION)`

Break: Questions?

CGI

Experience the commitment®

Plugin

= application module

1. describes what it provides/extends
2. framework plugs it in

- defined in `plugin.xml`
- dependencies and provided packages in `MANIFEST.MF`
- plugin $\subseteq$ feature $\subseteq$ product
- source not needed to extend RCP app

Extensions

- built on OSGi extensions (simplified)

❶ define extension point – contract for extensions

- attributes – primitive types (XML Schema)
- executable extension – implements/extends

❷ register contributing extensions (`plugin.xml`)

❸ process contributions (`IExtensionRegistry` via DI)

☞ 9

Executable Extensions

- framework creates classes from contributing plugin
- actual implementation inaccessible (dependency)
- created implicitly (default constructor)
- cannot access context – use `ContextInjectionFactory`

  must be called in processor

Fragments

- adds to application model
- GUI elements, commands, handlers, . . . ☞10
- extension (`org.eclipse.e4.workbench.model`)
- GUI decomposition – plugins for perspectives, parts, . . .

Addons

- model fragments ⇒ use DI
- behavioral annotations
- typically ☞ 11
  1. provide context objects (services)
  2. listen to messages
  3. process extensions

Extensions vs Addons*

1. prefer addons (simpler, DI)
2. extensions for existing extension points
3. custom extension points
   - model for future extensions
   - multiple similar extensions
   - (usually) used in one place

- extensions must be explicitly processed
- plugin defines extension point and processes extensions via addon (puts results into context)

Example Extension: Preference Pages

Each plugin can provide its preference page.

Extension point

- page id
- parent page id (optional)
- implementation extends `IPreferencePage`

Command w/ parameter: opened page id

Handler

1. reads preference pages
2. creates tree structure
3. opens `PreferenceDialog` on given page

☞12

Break: Questions?

Experience the commitment®

Miscellaneous

- Eclipse 4 RCP (e3 was different)
- RCP uses SWT (heavyweight components)
- RAP = RCP on web (in development)

Summary*

1. dependency injection: I want something, framework finds it
2. inversion of control: I write snippets, framework uses them
3. application model – IoC implementation, hierarchical
   - windows, perspectives, parts, . . .
   - commands (actions), handlers (implementation)
4. context – DI implementation, object storage
   - hierarchical: application, windows, perspectives, parts
   - persistent, changes not notified
5. event service – global event sending
   - for persistence, add context (uninitialized parts)
6. extensions – metadata, executable extensions
7. fragments – add to application model
8. addons – provide services, listen to events, process extensions

Sources

- official page

  https://wiki.eclipse.org/index.php/Rich_Client_Platform
- tutorials from Lars Vogel
  - Eclipse 4 RCP: The complete guide to Eclipse application development
  - http://www.vogella.com/tutorials/EclipseRCP/article.html
- slides and examples

  https://github.com/martinbayer/com.cgi.example.e4.rcp

THANK YOU

Any questions?

Experience the commitment®