Eclipse4/RCP/Dependency Injection - Eclipsepedia

wiki.eclipse.org 1/Feb/2015

With 10 years of experience with the Eclipse platform, a number of difficulties were observed.

1. Code frequently uses global singleton accessors (e.g., Platform, PlatformUI) or required navigating a deep chain of dependencies (e.g., obtaining an *IStatusLineManager*). Singleton services is particularly problematic for app servers like RAP/Riena.
2. The use of singletons tightly coupled consumers of "things" to their producer/provider, and inhibits reuse.
3. Mechanisms are not very dynamic — plugins react to context changes drastically rather than incrementally (just close the affected controls)
4. IEvaluationContext has global state that gets swapped according to context change (such as the focus control)
5. Current solutions are not multi-threadable - they assume evaluation occurs in the UI thread
6. Scaling granularity
    - Current solutions don't scale down to services with brief lifetimes
    - Current solutions may not scale up to very large numbers of services
7. Currently don't track service consumers to notify when services come/go
8. Client code needs to know the internals of the Eclipse code base
9. No support for services lookup that are composed out of other services on the fly

~~These difficulties are seen in other frameworks and are not unique to the Eclipse application platform. The Eclipse 4 Application Platform has adopted the use of Dependency Injection (DI) to circumvent these problems: rather than require client code to know how to access a service, the client instead describes the service required, and the platform is responsible for configuring the object with an appropriate service. DI shields the client code from knowing the provenance of the injected objects.~~
The Eclipse 4 Application Platform provides a JSR 330-compatible, annotation-based dependency injection (DI) framework, similar to Spring or Guice. The injector is defined in several plugins: *org.eclipse.e4.core.di, org.eclipse.e4.core.di.extensions, and org.eclipse.e4.ui.di.*

Overview

DI separates the configuration of an object from its behaviour. Rather than litter an object with details on how to access its required dependencies, the dependencies are instead configured through the use of an injector. For example, a typical Eclipse 3.x view would access the Eclipse Help system via the PlatformUI singleton, and the status line manager through the part's site:

```
class MyView extends ViewPart {
  public void createPartControl(Composite parent) {
    Button button = ...;
    PlatformUI.getWorkbench().getHelpSystem().setHelp(button,
"com.example.button.help.id");

    getViewSite().getActionBars().getStatusLineManager().setMessage("Conf
iguring system...");
  }
}
```

With E4AP, **parts are POJOs** and have their application services directly injected into the part:

```
class MyView {
  @Inject
  public void create(Composite parent, IWorkbenchHelpSystem help) {
    Button button = ...;
    help.setHelp(button, "com.example.button.help.id");
```

```
    slm.setMessage("Configuring system...");
  }
}
```

DI provides a number of advantages:

- Clients are able to write POJOs and list the services they need.
- Useful for testing: the assumptions are placed in the DI container rather than in the client code

DI has some disadvantages too:

- Service discovery: cannot use code completion to find out what is available.
- Debugging why injection has not occurred can be frustrating!

Standard Annotations and Classes

E4AP's injector is based on the standard JSR 330 annotations:

@Inject (javax.inject)

@Inject marks a constructor, method, or field as being available for injection.

@Named (javax.inject)

Injected values are typically identified by a type. But there may be a number of available objects of a particular type (e.g., there are likely a number of available Strings). Multiple objects can be distinguished by providing a name, both on setting them as well as requesting them for injection. For example:

```
@Inject
@Named(E4Workbench.INSTANCE_LOCATION)
private Location instanceLocation;
```

@Singleton is class annotation Indicating that the class should only be instantiated once per injection scope. Typical E4AP applications have only a single injector scope for the application.
The Provider class defers the injection to demand-time. Any value that can be injected can also be obtained through a provider.
The @PostConstruct and @PreDestroy annotations provide lifecycle notification for created objects. All methods annotated with @PostConstruct are called *after an object has been fully injected*. All methods annotated with @PreDestroy are called **before** an object is to be *uninjected* and released.

E4AP-specific Annotations

E4AP's injection framework also supports other E4AP-specific annotations.

@Optional (org.eclipse.e4.core.di.annotations)

The @Optional annotation can be applied to methods, fields, and parameters to mark them as optional for the

dependency injection. Typically, if the injector is unable to find a value to inject, then injection will fail. However, if this annotation is specified, then:

- for parameters: a `null` value will be injected;
- for methods: the method calls will be skipped;
- for fields: the values will not be injected.

@Active (org.eclipse.e4.core.contexts)

@Active serves a similar purpose to @Named, indicating the the value should be resolved from the active context.
For example, a handler could obtain the active part with:

```
@Execute
public void execute(@Active MPart activePart) {
   ...
}
```

@Preference (org.eclipse.e4.core.di.extensions)

The @Preference provides simple interfacing with the Eclipse preferences framework. The following snippet illustrates its use:

```
 @Inject @Preference(nodePath="my.plugin.id", value="dateFormat")
 protected String dateFormat;
```

The "dateFormat" field is updated as the preference changes too; the "nodePath" is optional and defaults to the code's defining bundle.
A class can receive notification of a preference change by instead receiving the injection through a setter:

```
 @Inject
 private void setDateFormat(@Preference(nodePath="my.plugin.id",
value="dateFormat") String dateFormat) {
 this.dateFormat = dateFormat;
 // ... and do something ...
 }
```

Alternatively the preferences set for a node can be obtained to allow setting values too:

```
 @Inject @Preference(nodePath="my.plugin.id")
 IEclipsePreferences preferences;

 private void use8601Format() {
 preferences.put(dateFormat, ISO8601_FORMAT);
 }
```

@Creatable (org.eclipse.e4.core.di.annotations)

Classes annotated with @Creatable will be automatically created by the injector if an instance was not present in the injection context. The automatically-generated instance is not stored in the context.

This annotation was introduced to handle a rather somewhat problematic situation: what should the injector do when presented a injectable but non-@Optional field or method argument that cannot be satisfied? An earlier version of the E4 DI, that followed the approach taken by Guice, would automatically create an object of the field type, providing the type was a concrete class and had a zero-argument constructor. This behaviour was changed in E4 4.2 M6 as it lead to unexpected results by developers. Now unsatisfied injections are only autogened if the object type is annotated with @Creatable.
To understand why this change was made, consider the following example:

```
@Inject
public void nameChanged(Shell shell, @Preference(node="my.plugin.id",
value="username") String name) {
   MessageDialog.openInformation(shell, "Hi " + name);
}
```

This example would not behave as expected as normally there are no Shell objects available for injection. Under the old DI behaviour, since Shell had a zero-argument constructor, the injector would have autogenerated an instance of `Shell`. But because the shell wasn't the expected (or intended) shell, the dialog was mis-placed.
With the changed behaviour, the DI will raise an `InjectionException` since there is no `Shell` available for injection and `Shell` is not annotated with @Creatable.
(To obtain the active shell, use `@Named(IServiceConstants.ACTIVE_SHELL) Shell`.)

@CanExecute, @Execute (org.eclipse.e4.core.di.annotations)

The @CanExecute and @Execute annotations tag methods that should be executed for a command handler. Methods tagged with @CanExecute should return a boolean.

@Focus (org.eclipse.e4.ui.di)

Parts can specify this annotation on a method to be called when the part receives focus. Parts **must** implement this method in such a way that a child control of their part can receive focus. Note that label controls cannot take focus.

```
@Focus
void grantFocus() {
  // correct
  textField.setFocus();

  // incorrect
  // label.setFocus();
}
```

In the event that your part only has a textual label in it, you should ask your part's inner composite to take focus.

```
public class DetailsPart {

  private Composite composite;

  @PostConstruct
  void construct(Composite parent) {
```

```
    composite = new Composite(parent, SWT.NONE);
    composite.setLayout(new FillLayout());

    Label label = new Label(composite, SWT.LEAD);
    label.setText(/* ... */);
  }

  @Focus
  void grantFocus() {
    composite.setFocus();
  }
}
```

@AboutToShow, @AboutToHide (org.eclipse.e4.ui.di)

Used in dynamic menu contribution elements. The respective annotated methods are called on showing of the menu, and on hiding of the menu. In @AboutToShow an empty list is injected, whereas @AboutToHide is injected with the same list, containing the elements contributed in @AboutToShow. Do not put long-running code into @AboutToShow as this delays the opening process of the menu.
Usage examples:

```
@AboutToShow
public void aboutToShow(List<MMenuElement> items) {
 MDirectMenuItem dynamicItem = MMenuFactory.INSTANCE
     .createDirectMenuItem();
 items.add(dynamicItem);
}

@AboutToHide
public void aboutToHide(List<MMenuElement> items) {
  System.out.println("aboutToHide() items-size: " + items.size());
  addSecond = !addSecond;
}
```

@GroupUpdates (org.eclipse.e4.core.di.annotations)

The @GroupUpdates annotation indicates to the framework that updates should be batched.

```
   // injection will be batched
   @Inject @GroupUpdates
   void setInfo(@Named("string1") String s, @Named("string2") String s2)
{
      this.s1 = s;
      this.s2 = s2;
   }
```

The setInfo() method will be triggered by a call to IEclipseContext#processWaiting():

```
   IEclipseContext context = ...;
   context.set("string1", "a");
   context.set("string2", "b");
```

```
context.processWaiting();  // trigger @GroupUpdates
```

@EventTopic (org.eclipse.e4.core.di.extensions), @UIEventTopic (org.eclipse.e4.ui.di)

The @EventTopic and @UIEventTopic annotations tag methods and fields that should be notified on event changes. The @UIEventTopic ensures the event notification is performed in the UI thread. Both the event's DATA object and the actual OSGi Event object (`org.osgi.service.event.Event`) is available. See the Events section for more details about events.

```
@Inject
public void setSelection(@EventTopic(REFRESH_EVENT) Object data) {
	fullRefresh();
}
```

Advanced Topics

Injection is performed using the active context. Changes to values in the context will trigger re-injection.

- Adding, setting, or modifying variables
  - Why is modify different from set

Injection Order

The E4AP DI Framework supports three types of injection, and is performed in the following order:

1. Constructor injection: the public or protected constructor annotated with @Inject with the greatest number of resolvable arguments is selected
2. Field injection: values are injected into fields annotated with @Inject and that have a satisfying type
3. Method injection: values are injected into methods annotated with@Inject and that have satisfying arguments

In certain cases, the injector will trigger methods annotated with @PostConstruct and @PreDestroy when the associated object is created or prior to being disposed. These situations include:

1. When behavioral objects are created to correspond to a Modeled UI element.

The DI framework processes fields and methods found in the class hierarchy, including static fields and methods too. Shadowed fields are injectable, but overridden methods are not. The injector does not process methods declared on interfaces.
Note that as the backing context changes, changed values will be re-injected.

Extending the DI Framework

ExtendedObjectSupplier LookupContextStrategy

Configuring Bindings

You can have a factory classes by using IBinding:

```
InjectorFactory.getDefault().addBinding(MyPart.class).implementedBy(MyFact
ory.class)
```

or using ContextFunctions:

```
public class MyFactory extends ContextFunction {
 public Object compute(IEclipseContext context) {
  MPart result = ContextInjectionFactory.make(MyPart.class, context);
  doWhatever(result);
  return result;
}
```

Debugging

Tips for debugging injection issues:

- set an exception breakpoint on org.eclipse.e4.core.di.InjectionException

Considerations

- thread-safety: method injection may happen on any thread
- final fields can only be supported using constructor injection
- if several injectable fields required, then use a single @Inject method with multiple arguments and consider @GroupUpdate rather than using several single setter-style @Inject methods.

Be aware that injected values are dynamic: values that are changed in the context may be immediately propagated into injected fields/methods. Also, we have the "@Optional" annotation which allows values not currently in the context to be injected as "null" and re-injected later when the values are added to the context.

Current Caveats

Although E4AP's DI will inject OSGi services, it does not currently track changes to the service. This means that if the service is not available at time of initial injection, but subsequently becomes available, existing requestors for the the service are not notified. Nor are receivers notified should the service disappear. This work is being tracked across several bugs: bug 331235, bug 330865, bug 317706.

References

Dhanji Prasanna's *Dependency Injection* provides provides guidance on best practices on using DI in your application.

Disclaimer