Advanced Use of Eclipse 4's Dependency Injection Framework

Brian de Alwis
Manumitting Technologies, Inc
(includes work with Landmark Graphics Corp)
Agenda

Problem (Why DI?)

Why use DI in Eclipse? What’s different about Eclipse’s DI?

How can I use Eclipse’s DI?

Context functions

Custom annotations

Custom services
Cardinal Rules of SE

- Increase cohesion between related objects
  - Together, they form a meaningful unit
- Decrease coupling between unrelated components
  - Improves isolation: a component can be modified without breaking others
  - Improves flexibility: can reuse components in new situations

But it's a blurry distinction between what's "related" and "unrelated"
The Problem: Most objects are not created independently, but are created as part of something larger. Yet we build these objects as if they were independent, and must specify how to access their needed services. Result: structural dependencies (e.g., singletons), or creator responsible for extensive configuration (or service locators)

There are over 1600 references to PlatformUI.getWorkbench() in the Eclipse Platform alone! (Which likely makes the RAP developers cry.)

(Image from RJ Walker, GC Murphy (2000). Implicit Context: Easing Software Evolution and Reuse. FSE; they called these details “EEK”: External Extraneous Knowledge)
Let's consider a different situation: medical surgery. A surgeon obtains items and information from those in the environment. People in the environment notify surgeon when circumstances change. The surgeon is not responsible for knowing what is going on and where items are found. Thus transplant surgeons can travel long distances to unfamiliar hospitals and yet be immediately effective. They make assumptions of their environment, and their environment provides.

How do we take this approach to programming?
Annotations: the annotations spare us from writing a lot of boilerplate
What we want to be able to do is get rid of this EEK and instead say "find me the appropriate selection service / adapter manager / thingy."

But how do we get these “thingies”? We push this problem to whomever created us, transforming the problem into a runtime configuration issue.
Containers

Most objects are created in the context of some other object

Creator determines the lifecycle

That creator is responsible for providing the configuration required by its children

Container: some other component who determines this object’s lifecycle

Often used in the context of web apps (e.g., Tomcat, Jetty).

A container may have been itself created by some other container.

We now use DI extensively within the Eclipse UI, and due to how we need injection to work, we have our own injector.
Why use DI for Eclipse?

- Simplify access to workbench services
- Removes need for singletons
- Remove boilerplate (particularly listeners)
- Shrink service lifetimes
  - Avoid need to create on startup
  - Cleanup when no longer needed

DI provides advantages for code: separation of configuration from use
Simplify access: avoid IStatusLineManager problem
## Other Forms of DI

(AKA: Inversion of Control)

- Maven / Plexus
- OSGi Declarative Services (<reference>)
- Spring
- JSR330: Guice, Spring, and E4

In other DI systems, like Guice or Spring, there is a single-level of configuration. Ulis are a bit different, and we needed something more flexible.
E4 DI Differences

- Supports field, method, and constructor injection
- Supports custom annotations
  
  **Will re-inject whenever the injected values are changed or the injection situation changes**

The last item is key differentiator from other injectors.
public class PropertiesPart {
    @Inject IExtensionRegistry registry;
    @Inject @Optional
    public void setInput(
        @Named(IServiceConstants.ACTIVE_PART) MPart activePart,
        @Named(IServiceConstants.ACTIVE_SELECTION) Object selection) {
        // look up corresponding property page in extension registry
        // flip out the current page and replace
    }
}

Contrast @Named vs FQTN
Avoids boilerplate for installing/removing listeners.
Compare & contrast the code for tracking the active part with the IPartService.
public class CommandHandler {
    @CanExecute
    public boolean canExecute(ESelectionService selService) {
        return selService.getSelection() != null;
    }

    @Execute
    public void execute(MWindow window, EPartService partService,
                         EModelService modelService) {
        MPart part = modelService.createModelElement(MPart.class);
        part.setLabel("New Part");
        part.setContributionURI("platform:/plugin/bundle/classname");
        List<MPartStack> stacks = modelService.findElements(window, null,
                                        MPartStack.class, null);
        stacks.get(0).getChildren().add(part);
        partService.showPart(part, PartState.ACTIVATE);
    }
}
Simplify Code: Preferences

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

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

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

private void use8601Format() {
    preferences.put(dateFormat, ISO8601_FORMAT);
}
```
public class GameDisplayPart {
    @Inject MPart me;
    @Inject IEventBroker eventBroker;
    Game game;

    @Inject
    public void raisedToTop(
        @UIEventTopic(UIEvents.UILifeCycle.BRINGTOTOP) MPart part) {
        if (me == part) {
            refresh();
        }
    }

    public void newGame() {
        game = new Game();
        eventBroker.post("ca/mt/game/set", game);
    }
}

- UIEventTopic ensures the event is delivered on the UI thread using the Realm found on the part's context.
- IEventBroker is a wrapper around the OSGi EventAdmin service that presents a simplified listener; its lifetime is also bounded by the context in which it was created (rather than the bundle).
- Every model change in the E4 EMF-based UI model is turned into an event that is then reflected to the SWT/JavaFX/etc widgets
JSR 330: Annotations

JSR 330 introduces a set of standard annotations for DI in `javax.inject`

@Inject, @Named, @Singleton

`Provider<T>`: supports deferred injection

(@Scope & @Qualifier: used when defining other injection-related annotations)

JSR 250 `java.annotation`: @PostConstruct, @PreDestroy

But we have common underpinnings. It’s possible to write injectable code that should work across any JSR330-compliant injector, like the Eclipse injector.
And you can define your own field and parameter annotations to have them be custom injected
How Does E4 DI Work?

```java
public class PropertiesPart {

    @Inject IExtensionRegistry registry;

    @Inject
    PropertiesPart(I EclipseContext context) {
        /* … */
    }

    @Inject @Optional
    public void setInput(
        @Named(IServiceConstants.ACTIVE_PART) MPart activePart,
        @Named(IServiceConstants.ACTIVE_SELECTION) Object selection)
    {
        // look up corresponding property page in extension registry
        // flip out the current page and replace
    }

    @PostConstruct
    public void init(Composite parent) {
        // build up widget representation
    }

    @PreDestroy
    public void dispose() {
        /* … */
    }
}
```

Process @Inject items:

1. If created via IInjector#make(): Constructors marked with @Inject (and public or package protected) or 0-arg constructor
2. Fields with @Inject
3. Methods with @Inject
4. Methods with @PostConstruct
Where Do Injected Objects Come From?

IEclipseContext: tree-based String → Object map with inheritance in lookups

Key lookup starts at some child and proceeds up the chain until satisfied

Starting context is likely, but not necessarily, a leaf

You can create your own context hierarchy (ContextInjectionFactory) or…
Key workbench components have a corresponding IEclipseContext

Key: any string

An unsuccessful lookup at the editor level (e.g. for "IExtensionRegistry") traverses upwards

You can store whatever values you like, including custom app state

Top level looks up services in OSGi level, including DS
What If a Value is Unavailable from the Context?

Injection fails

…unless marked as @Optional: then injected with null
Injecting OSGi Services

Top-level of the Workbench IEclipseContext hierarchy is the OSGi service layer

Services obtained via BundleContext for org.eclipse.e4.ui.workbench
How Do I Change Values in the Context?

Get the context, and call “context.set(name, value)”

Will cause any places that requested injection to be re-injected with the new value

Can also declare a variable name at a particular context, then use “context.modify(name, value)” to set that value from any child context

Variables can be configured from App.e4xmi

Can always obtain the context used for injection via “@Inject IEclipseContext context”
Advanced DI
RunAndTracks (RATs)

A runnable that tracks changes relative to a particular context

Changes to those context variables causes the RAT to be re-run while the RAT returns “true”.
Context Functions: Computing Values on the Fly

Instead of setting a value, compute it using an IContextFunction

Provided the originating context

Values are (currently) aggressively cached by the originating context

Must subclass abstract ContextFunction class
Creating/Injecting

**Injector** has 4 principle methods:
- `#make(Class, PrimaryObjectSupplier)`
- `#inject(Object, PrimaryObjectSupplier)`
- `#uninject(Object)`
- `#invoke(Object, Annotation, PrimaryObjectSupplier)`

(But typically use ContextInjectionFactory)

An injector is used to create or inject objects; will be required for objects that aren’t managed by the Eclipse framework.

The injector attempts to resolve injection requests using a set of provided object suppliers; our primary supplier is based around the IEclipseContext hierarchy. Typically use the wrappers on ContextInjectionFactory.
Explicit Injection

```java
public abstract class InjectableViewPart extends ViewPart {
    private IEclipseContext _context;

    public void init(IViewSite site) throws PartInitException {
        _context = (IEclipseContext) site.getService(IEclipseContext.class);
        assert _context != null;
        super.init(site);
        ContextInjectionFactory.inject(this, _context);
    }

    public void dispose() {
        super.dispose();
        ContextInjectionFactory.uninject(this, _context);
    }
}
```

If you create/inject an object, you should remember to uninject
But IECs have a lifecycle: can be disposed, which cleans up any previously-injected objects. Child context lifetimes are bounded by the parent.
Custom Annotations

Create custom annotation
Register corresponding Extended Object Supplier as an OSGi service
Example: @ActiveEditor

```java
@Named("bug398728") required prior to 4.4

Need to be able to somehow identify a scope — what if we had multiple windows? A viewpart would likely want to be notified only of changes within its windows?
```
Custom annotation

```java
/**
 * This annotation causes arguments and fields in the requesting object to be resolved from the active editor's context
 * rather than the requesting object's source context. Values will be re-injected when a different editor is activated.
 * 
 * @Qualifier
 * @Documented
 * @Target({ ElementType.FIELD, ElementType.PARAMETER })
 * @Retention(RetentionPolicy.RUNTIME)
 * public @interface ActiveEditor {
 *     String value() default "";
 * }
 */
```
We expose the annotation and extended supplier via an OSGi Service. This is a specific detail to the default InjectorImpl.
Object Suppliers

Object suppliers (primary & extended) are provide object describing the injection site

**IObjectDescriptor**: provides access to the type and annotations of the field/parameter

**IRequestor**: the requesting location: the method, field, constructor; can re-trigger

The IRequestor also provides access to the object being injected.
public class ActiveEditorObjectSupplier extends ScopedObjectSupplier {
    @Override
    public Object get(IObjectDescriptor descriptor, IRequestor requestor, boolean track,
            boolean group) {
        if (descriptor == null) { return null; }
        String serviceName = getServiceName(descriptor);
        assert serviceName != null;
        Object scope = getLookupScope(requestor);
        assert scope != null;
        trackScopedRequestor(track, scope, requestor);
        IEditorPart editor = getSuitableEditor(scope);
        // IInjector.NOT_A_VALUE is analogous to undefined in JavaScript. It is distinct from a null value
        if (editor == null) { return IInjector.NOT_A_VALUE; }
        if (serviceName == null || serviceName.isEmpty() || serviceName.equals(IEditorPart.class.getName())) { return editor; }
        IEclipseContext context = (IEclipseContext) editor.getSite().getService(IEclipseContext.class);
        if (context == null) { return IInjector.NOT_A_VALUE; }
        if (track) {
            Map.Entry<IRequestor,String> key = new AbstractMap.SimpleEntry<IRequestor,String>(requestor, serviceName);
            ValueRequestor valueRequestor = _valueRequestors.get(key);
            if (valueRequestor == null) {
                _valueRequestors.put(key, valueRequestor = new ValueRequestor(key, context));
            }
            return valueRequestor.getValue(context);
        }
        Object value = context.get(serviceName);
        return value != null || context.containsKey(serviceName) ? value : IInjector.NOT_A_VALUE;
    }
}

We’ll explain scope in a sec
Note get(): “track” means whether changes to this value should be tracked and cause re-injection; “group” whether the requestor changes can be grouped together
getServiceName() looks at the IObjectDescriptor to get details of the @Named or the requested type name.
We’ll explain scope in a sec

ValueRequestor uses
public interface LookupScope {
    Object getLookupScope();
}

public abstract class ScopedObjectSupplier extends ExtendedObjectSupplier {
    ListMultimap<Object, IRequestor> _scopedRequestors …;

    protected Object getLookupScope(IRequestor requestor) {
        Object o = requestor.getRequestingObject();
        LookupScope lookup;
        Object scope = o;
        if (o instanceof LookupScope) {
            scope = ((LookupScope) o).getLookupScope();
        } else if ((lookup = _adapter.adapt(o, LookupScope.class)) != null) {
            scope = lookup.getLookupScope();
        }

        if (scope instanceof IEditorPart ||
            scope instanceof IWorkbenchWindow) {
            return scope;
        } else if (scope instanceof IWorkbenchPart) {
            return ((IWorkbenchPart) o).getSite().getWorkbenchWindow();
        } else if (scope instanceof WindowControlContribution) {
            return ((WindowControlContribution) scope).getWorkbenchWindow();
        }

        return getWorkbench();
    }

    protected void trackScopedRequestor(boolean track, Object scope, IRequestor requestor) {
        if (track && requestor.isValid()) {
            if (!_scopedRequestors.containsEntry(scope, requestor)) {
                _scopedRequestors.put(scope, requestor);
            }
        } else {
            _scopedRequestors.remove(scope, requestor);
        }
    }
}

// continued

We’ll explain scope in a sec

Note get(): “track” means whether changes to this value should be tracked and cause re-injection; “group” whether the requestor changes can be grouped together
public abstract class ScopedObjectSupplier  
   extends ExtendedObjectSupplier {  
      ListMultimap<Object, IRequestor> _scopedRequestors ...;

   @PostConstruct
   protected void init() {  
      /* install part and window listeners that reinject  
      all scoped IRequestors when the active  
      part/editor/window changes */
   }

   protected void editorActivated(IEditorPart editor,  
      IWorkbenchWindow window) {  
      processRequestors(_scopedRequestors.get(editor));  
      processRequestors(_scopedRequestors.get(window));  
      processRequestors(_scopedRequestors.get(window.getWorkbench()));
   }

   private void processRequestors(List<IRequestor> requestors) {  
      for(Iterator<IRequestor> iter = requestors.iterator(); iter.hasNext();) {  
         IRequestor r = iter.next();  
         if(r.isValid()) {  
            r.resolveArguments(false);
            r.execute();
         } else {  
            iter.remove();
         }
   }

   We’ll explain scope in a sec  
   Note get(): “track” means whether changes to this value should be tracked and cause re-injection; “group” whether  
   the requestor changes can be grouped together
Object Supplier Gotchas

Object Suppliers have to be multi-threaded

Calls may come in from anywhere

Object Suppliers need to track changes if track==true

And clean up any tracking if track==false
Installing services in the context hierarchy

How can we install a service into the contexts?

Create new IEclipseContext class, override #lookup() method, and instantiate in the right place

Provide E4 lifecycle manager and initialize application-level IEclipseContext

Add an MContext listener (UIEvents.Context.TOPIC_CONTEXT)

Provide IContextFunction-based service installer as OSGi-level Service
Workbench Services Installer

<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
   activate="configure">
   <service>
      <provide interface="org.eclipse.e4.core.contexts.IContextFunction"/>
   </service>
   <property name="service.context.key" type="String" value="com.lgc.dsaf.app.services.properties.DspAppPropertiesProvider"/>
   <implementation class="com.lgc.dsaf.ui.contexts.WorkbenchServiceInstaller"/>
   <property name="service.level.key" type="String" value="workbench"/>
   <property name="service.implementation.key" type="String" value="com.lgc.dsaf.app.impl.services.properties.DspPropertiesProviderImpl"/>
</scr:component>

Combines OSGi DS with E4 IContextFunctions. Context functions are provided the original look-up hierarchy. On access, WSI uses service.level.key to find appropriate IEclipseContext in the ancestry, and instantiates and registers the implementation.
ACTIVE_* vs @Active

```java
public class ActivePartLookupFunction extends ContextFunction {
    @Override
    public Object compute(IEclipseContext context, String contextKey) {
        MContext window = context.get(MWindow.class);
        if (window == null) {
            window = context.get(MApplication.class);
            if (window == null) {
                return null;
            }
        }
        IEclipseContext current = window.getContext();
        if (current == null) {
            return null;
        }
        return current.getActiveLeaf().get(MPart.class);
    }
}
```

If you create/inject an object, you should remember to uninject
But IECs have a lifecycle: can be disposed, which cleans up any previously-injected objects. Child context lifetimes are bounded by the parent.
Debugging/Diagnosis

Add an exception breakpoint on InjectionException: will provide the key being looked up

Poking around on the stack will reveal the requestor and suppliers
Case Study: Landmark

Singleton managers controlling access to “display list” and “cursor location”

527 references to singleton display list manager

41 references to singleton cursor location manager
So how might we use DI to remove the structural dependency for obtaining the DLIP or CursorLocation?

What this code says:
when processed by the injector, lookup two objects using the type names
whenever these values are changed, re-inject me
Note @Optional
if CLPGrMgr is removed, passed in null
if DLPFnMgr is removed, then results in an injection exception

Spares me from knowing how to obtain the service, and constraints on obtaining that service.
But still need to install listeners.
But we’re really not interested in getting the managers — we really want the actual DLIP and cursor location, and to know when they change.

Values looked up by @Name; if unspecified, use the FQTN of the field/parameter

I no longer need to manage listeners: cleaned up when this object’s container is disposed of. (Ignore “container” for the moment)
Improvements for 4.5

Improve service installing (make I EclipseContext#lookup() pluggable… somehow) — avoid WorkbenchServiceInstaller

Allow control over how method injection is performed (e.g., on SWT thread) [contribution from Markus Kuppe]

Make our DI faster via bytecode generation
Thank you!

bsd@mt.ca
Object Auto-Creation

Classes marked with @Creatable will be automagically created when a reference cannot be resolved.
Common Problems

Annotated methods aren't injected

Injector only processes methods with @Inject

But IInjector#invoke(o, @Annotation) doesn't require @Inject!
Injection on Handlers

Creation context is not the same as the execution context

Different context provided to IInjector's #invoke() vs #make()

Injection only via @CanExecute and @Execute
How to make use of DI now?

IServiceLocator#getService(XXX.class) does a context lookup

implemented by PartSite, IWorkbenchWindow, IWorkbench, and more

Easy to create Injectable variants of the standard Eclipse parts:

InjectableViewPart, InjectableEditorPart, InjectableWorkbenchContribution