What's the Context?

John Arthorne
Paul Webster
Boris Bokowski
Oleg Besedin

IBM Rational

Overview

- Problems with the existing Eclipse platform application model
- Introduce notion of contexts
- How contexts solve these problems
- Applying contexts to the e4 workbench
- Current state

The Singleton Problem

- Most Eclipse code “reaches out” to various singleton methods to access the services they need:
  - PlatformUI.getWorkbench()
  - Platform.getExtensionRegistry()
  - ResourcesPlugin.getWorkspace()
  - JavaCore.createCompilationUnitFor(IFile)
  - IDE.getMarkerHelpRegistry()

The Singleton Problem

- This **seems** to work well:
  - Very simple clean client code
  - Isolated from implementation changes (accessor can return a different service instance without breaking clients)
  - Provides an entry-point into a pure interface-based API
  - Overall, a good solution if there will never be a different provider of that service, or multiple implementations

The Singleton Problem

- What if someone else wants to provide an implementation of the same service?
- What if there are multiple copies of the service available at any given time?
- What if someone reusing your code wants to select what implementation you use?
- What if you don't want to contaminate your code with references to service providers?
- Can your code be reused on a server? In a browser? In embedded devices?

The Singleton Problem

- Concrete example: embedding a view or editor in a dialog
- Most view and editor implementations “reach out” to workbench window or part site to obtain various things it needs: the selection, the parent shell, keybinding service, etc
- To re-host a view or editor elsewhere, we need to “fake” all of this surrounding context
- If a view or editor reaches out to a singleton, we are out of luck

The Singleton Problem

- Concrete example: only one IWorkspace
- In the Eclipse client IDE, there is only ever one IWorkspace
- Clients use ResourcesPlugin.getWorkspace()
- When we tried hosting the workspace on the Bespin server, we wanted one IWorkspace per user in the same runtime
- Removing this one singleton is **months** of work!

Requirements

- Prevent application code from “reaching out” to get the things they need
- Remove assumption of single service provider and single available implementation
- Enable overriding of service selection choices

Introducing Contexts

- A context sits between application code and the framework
- Brokers interaction with the framework: service lookups, service registration
- Similar role to BundleContext in OSGi world

```
public interface IEclipseContext {
    public boolean containsKey(String name);
    public Object get(String name);
    public Object get(String name, Object[] args);
    public void remove(String name);
    public void set(String name, Object value);
}
```

Context Hierarchy

- Contexts are hierarchical – requests that cannot be satisfied are delegated to a parent context
- Can create a child context to tweak or override aspects of the context's behaviour
- We can customize application code's view of the world by inserting another context
- We'll see later how this works very well in user interfaces

Computed Values

- Values in context can be plain old objects, or IComputedValue objects (functions)
- On lookup, IComputedValue evaluated to produce result
- Allows us to defer creation of expensive values until needed

```
public interface IComputedValue {
    public Object compute(IEclipseContext ctxt, Object[] args);
}
```

Computed Values

- Computed values are provided the **local** context in which the request was made
- A generic computed value defined higher in the context tree can make use of more specific context data when computing values

```
public interface IComputedValue {
    public Object compute(IEclipseContext ctxt, Object[] args);
}
```

Computed Value Example

```java
static enum Color {RED,BLUE,YELLOW,GREEN,ORANGE,PURPLE;}

static class ComplementaryColor implements IComputedValue {
    public Object compute(IEclipseContext context, Object[] args) {
        switch ((Color) context.get("color")) {
            case RED: return Color.GREEN;
            case GREEN: return Color.RED;
            case BLUE: return Color.ORANGE;
            case ORANGE: return Color.BLUE;
            case YELLOW: return Color.PURPLE;
            case PURPLE: return Color.YELLOW;
            default: return null;
        }
    }
}
```

Computed Value Example

```
IEclipseContext p = EclipseContextFactory.create();
p.set("complement", new ComplementaryColor());
IEclipseContext child =
    EclipseContextFactory.create(p,null);
child.set("color", Color.RED);
System.out.println(child.get("color")); --> “RED”
System.out.println(child.get("complement"));  --> “GREEN”
```

- Computed value only needs to be defined once
- All child contexts inherit function, but can override function inputs

Resource Selection Example

```
Object next = e.next();
if (next instanceof IResource) {
        if (resources == null)
                resources = new ArrayList(getStructuredSelection().size());
        resources.add(next);
        continue;
} else if (next instanceof IAdaptable) {
        Object resource = ((IAdaptable) next).getAdapter(IResource.class);
        if (resource != null) {
                if (resources == null)
                        resources = new ArrayList(getStructuredSelection().size());
                resources.add(resource);
                continue;
        }
} else {
        boolean resourcesFoundForThisSelection = false;
        IAdapterManager adapterManager = Platform.getAdapterManager();
        ResourceMapping mapping = (ResourceMapping) adapterManager.getAdapter(next, ResourceMapping.class);
        if (mapping != null) {
                ResourceTraversal[] traversals = null;
                try {
                        traversals = mapping.getTraversals(ResourceMappingContext.LOCAL_CONTEXT, new NullProgressMonitor());
                } catch (CoreException exception) {
                        IDEWorkbenchPlugin.log(exception.getLocalizedMessage(), exception.getStatus());
                }
                if (traversals != null) {
                        for (int i = 0; i < traversals.length; i++) {
                                IResource[] traversalResources = traversals[i].getResources();
                                if (traversalResources != null) {
                                        resourcesFoundForThisSelection = true;
                                        if (resources == null)
                                                resources = new ArrayList(getStructuredSelection().size());
                                        for (int j = 0; j < traversalResources.length; j++) {
                                                resources.add(traversalResources[j]);
}}}}}
```

Resource Selection Example

- Can pass arguments when looking up values
- Arguments passed to IComputedValue
- In this example we have a computed value that can convert a selection to resources
- Giant wad of code only has to be written once

```
IEclipseContext context = ...;
Object[] args = new Object[] {IResource.class};
IResource[] resources = context.get("Selection", args);
```

Events

- If you are interested in a value, you are often also interested in when that value changes
- A common idiom is that you have a chunk of update code to run when events occur
- You can register a runnable with a context, that will be re-run every time values accessed by that runnable change

```
public interface IEclipseContext {
  public void runAndTrack(final Runnable r);
  …
```

Run and Track Example

```
double total = 0;

public void price() {
    final IEclipseContext context = EclipseContextFactory.create();
    context.set("price", 19.99);
    context.set("tax", 0.05);
    context.runAndTrack(new Runnable(){
        public void run() {
            total = (Double)context.get("price") *
                (1.0 + (Double)context.get("tax"));
        }
    }, "calculator");
    print(total);                        -->  “$20.99”
    context.set("tax", 0.07);
    print(total);                        -->  “$21.39”
}
```

Reality Check

- Application code still “reaches out” to the context
- I have still contaminated my application code with Eclipse-specific APIs
- The “run and track” concept is hard to wrap your head around, and only works if you have a runnable that is a pure function of values in the context

Dependency Injection

- Injecting services into plain objects has become a popular solution to the singleton problem in the past five years:
  - PicoContainer
  - Spring
  - Google Guice
  - OSGi declarative services
- By combining DI with contexts, we get cleaner, simpler, more reusable application code

Injection Example

- Currently support Guice, JSR-250, and simple @In, @Out annotations
- Field/Method prefixes for < Java 5 targets

```java
class Crayon {
    @In
    Color color;
    @In
    Color complement;
    public void draw() {
        System.out.println("My ink is " + color);
        System.out.println("Complementary color: " + complement);
    }
}
```

Injection Example

```
IEclipseContext parent = EclipseContextFactory.create();
parent.set("complement", new ComplementaryColor());
IEclipseContext context =
    EclipseContextFactory.create(parent, null);
context.set("color", Color.YELLOW);
Crayon crayon = new Crayon();
ContextInjectionFactory.inject(crayon, context);
crayon.draw();
```

```
My ink is YELLOW
Complementary color: PURPLE
```

OSGi Services and Contexts

- OSGi services are a powerful mechanism for decoupling service providers from consumers
- Contexts support look-up of OSGi services
- Context manages service lifecycle for you
- Have services injected into your objects to simplify (remove) service-management code

OSGi Service Example

```
interface IPaletteService {
    public Color getColor();
}
class PaletteImpl implements IPaletteService{
    private final Color color;
    PaletteImpl(Color color) {
        this.color = color;
    }
    public Color getColor() {
        return color;
    }
}
```

OSGi Service Example

```
class Crayon {
 @In
 IPaletteService palette;
 public void draw() {
  if (palette == null)
   System.out.println("I'm out of ink!");
  else
   System.out.println("My ink is  " + palette.getColor());
 }
}
```

OSGi Service Example

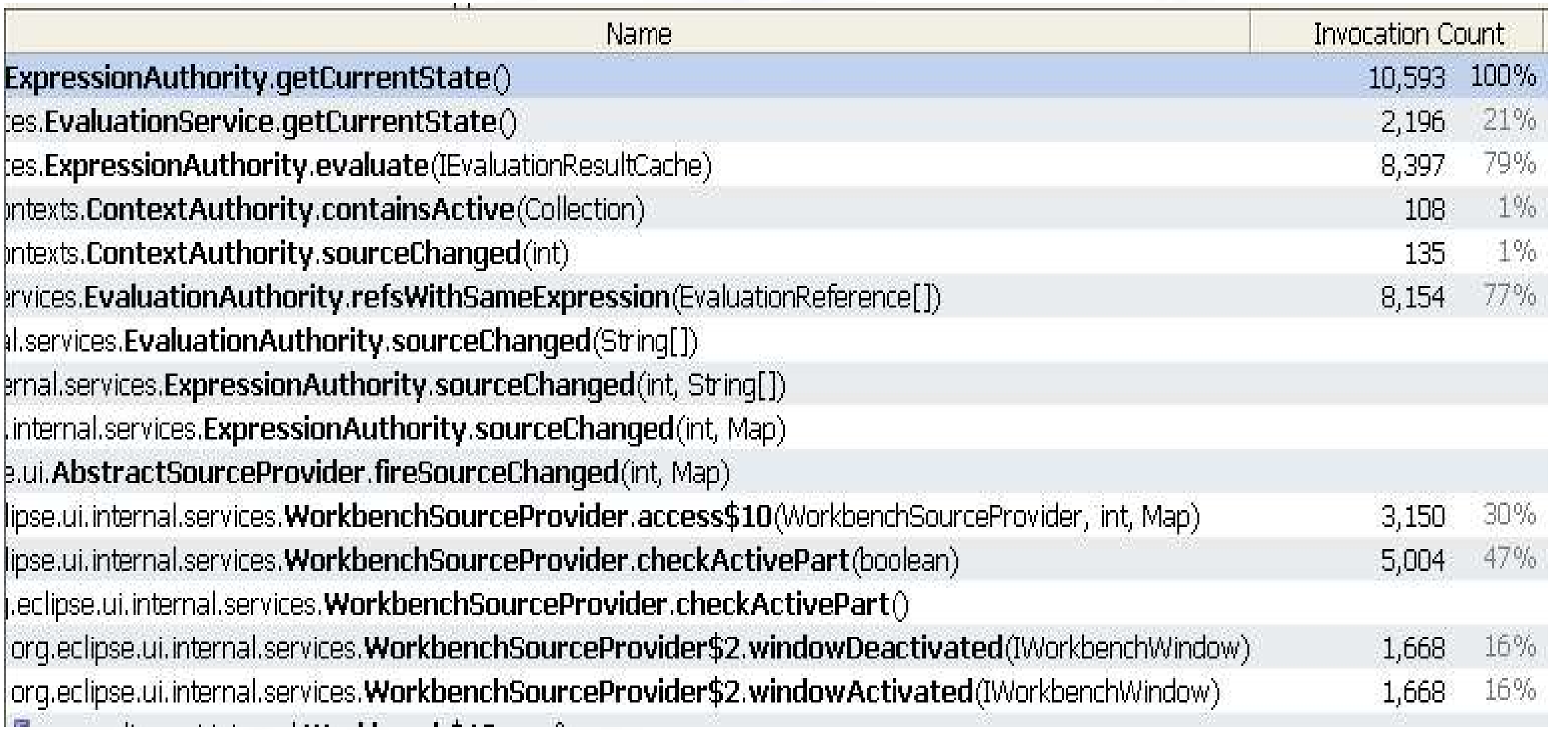
```
ServiceRegistration reg = Activator.bc.registerService(
    IPaletteService.class.getName(),
    new PaletteImpl(Color.BLUE), null);
IEclipseContext context =
    EclipseContextFactory.createServiceContext(Activator.bc);

Crayon crayon = new Crayon();
ContextInjectionFactory.inject(crayon, context);
crayon.draw();          -->  “My ink is  BLUE”
reg.unregister();
crayon.draw();          --> “I'm out of ink!”
```

The Event Storm Problem

- Wherever UI elements need to reflect an underlying model's state, they hook listeners to react to changes
- UI elements also need to reflect the state of other UI elements, so they hook listeners to react to changes in other parts of the UI
- A single trigger can lead to a massive sequence of events
- Often reacting to intermediate states rather than the final state when everything settles down

The Event Storm Problem

- Example: switch between workbench windows
- Thousands of events due to UI model changes



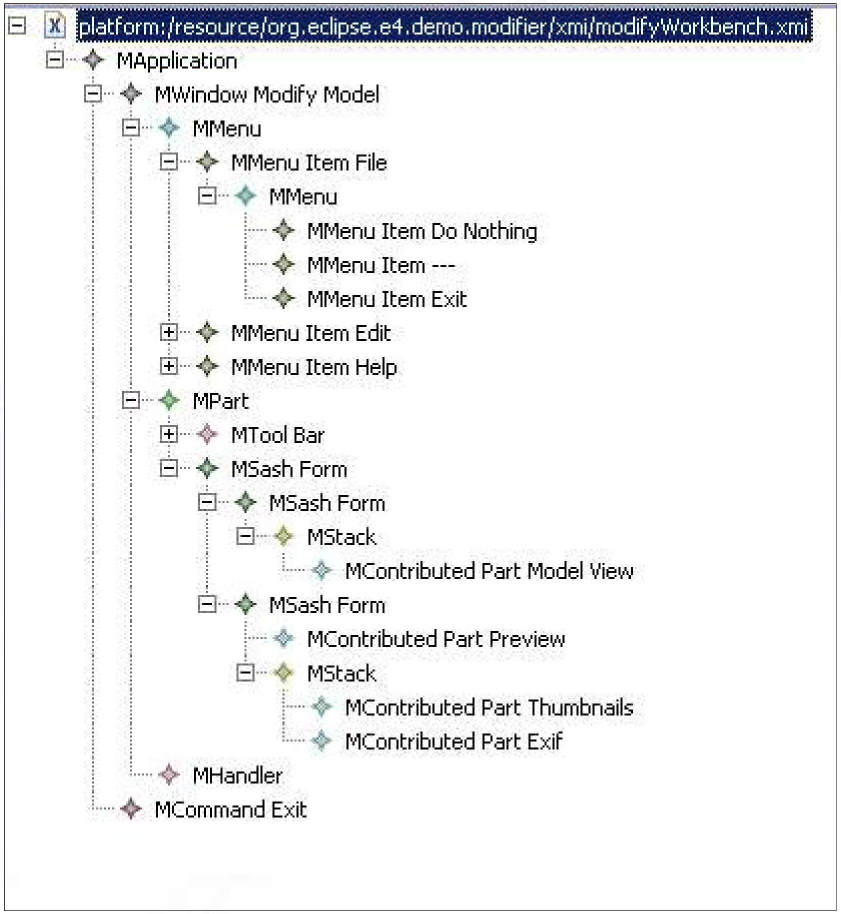
Name	Invocation Count	
ExpressionAuthority.**getCurrentState**()	10,593	100%
es.**EvaluationService.getCurrentState**()	2,196	21%
es.**ExpressionAuthority.evaluate**(IEvaluationResultCache)	8,397	79%
ntexts.**ContextAuthority.containsActive**(Collection)	108	1%
ntexts.**ContextAuthority.sourceChanged**(int)	135	1%
rvices.**EvaluationAuthority.refsWithSameExpression**(EvaluationReference[])	8,154	77%
l.services.**EvaluationAuthority.sourceChanged**(String[])		
rnal.services.**ExpressionAuthority.sourceChanged**(int, String[])		
.internal.services.**ExpressionAuthority.sourceChanged**(int, Map)		
.ui.**AbstractSourceProvider.fireSourceChanged**(int, Map)		
lipse.ui.internal.services.**WorkbenchSourceProvider.access$10**(WorkbenchSourceProvider, int, Map)	3,150	30%
lipse.ui.internal.services.**WorkbenchSourceProvider.checkActivePart**(boolean)	5,004	47%
.eclipse.ui.internal.services.**WorkbenchSourceProvider.checkActivePart**()		
org.eclipse.ui.internal.services.**WorkbenchSourceProvider$2.windowDeactivated**(IWorkbenchWindow)	1,668	16%
org.eclipse.ui.internal.services.**WorkbenchSourceProvider$2.windowActivated**(IWorkbenchWindow)	1,668	16%

Calming the Storm

- Contexts propagate changes in two phases:
  - Invalid context values affected by the change
  - Queue up runnables that will update state
  - Execute runnables after invalidation is complete
- Listeners no longer react and perform updates based on intermediate states
- All update code only runs once

How e4 Workbench uses Contexts

- Context hierarchy based on part hierarchy



```
platform:/resource/org.eclipse.e4.demo.modifier/xmi/modifyWorkbench.xmi
  MApplication
    MWindow Modify Model
      MMenu
        MMenu Item File
          MMenu
            MMenu Item Do Nothing
            MMenu Item ---
            MMenu Item Exit
        MMenu Item Edit
        MMenu Item Help
      MPart
        MTool Bar
        MSash Form
          MSash Form
            MStack
              MContributed Part Model View
          MSash Form
            MContributed Part Preview
            MStack
              MContributed Part Thumbnails
              MContributed Part Exif
      MHandler
    MCommand Exit
```

How e4 Workbench uses Contexts

- Views and Editors get injected on construction

```
public class ApplicationView {
    public ApplicationView(Composite parent,MApplication<MWindow<?>> app){
        Label label = new Label(parent, SWT.SHADOW_OUT);
        label.setText(app.eClass().getName() + "("+ app.getId() + ")");
    }
}
```

- Command handlers injected on execution

```
public class DeleteProjectHandler {
    // framework will ask handler if it can execute:
     ///public boolean canExecute(*);
     public void execute(IProject project, IProgressMonitor monitor,
           IExceptionHandler exceptionHandler) {
           // execute after being injected with information from context
```

Commands and Handlers

- In 3.x most of the command framework is tied to the global application context (maintained by the IEvaluationService)
- IEvaluationContext has global state that gets swapped according to context change (such as the focus control)
- There are too many parallel trees that mimic each other (widget tree, service locator tree, workbench part tree)

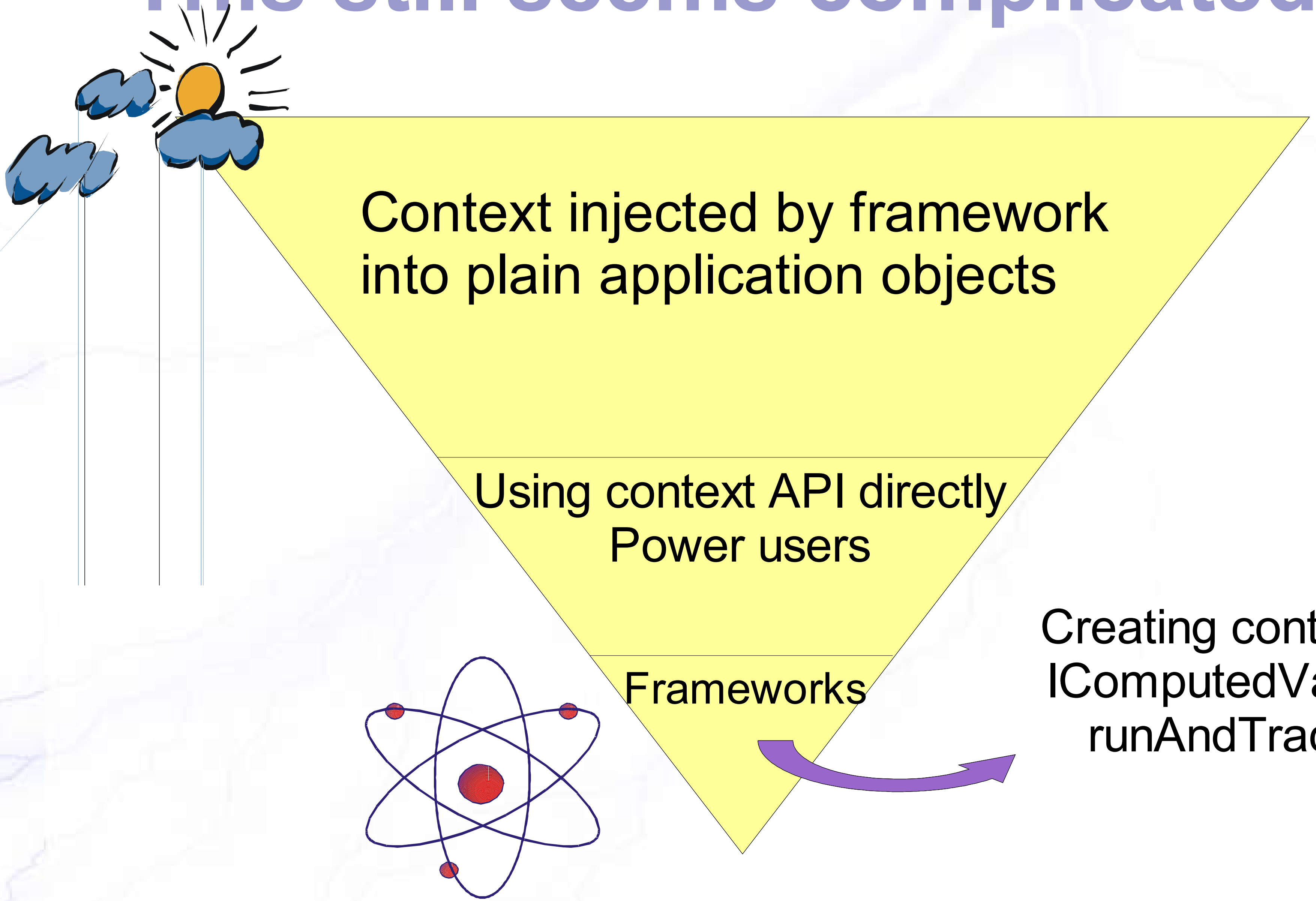
Commands and Handlers

- Investigated the notion of contexts for information and service lookup
- It is important that the contexts have:
  - The ability to replace and access local data
  - The notion that looking up a piece of data can depend on a strategy (IComputedValue in this implementation)
  - The ability to plug in different strategies at different levels of the workbench
- This allows a view's handler to react to its view's state without being affected by global changes.

Current State

- Working implementation of contexts, injection, and OSGi service support
- Current API is very rough, subject to change
- Please try it out and give feedback
- In e4 repository: org.eclipse.e4.core.services
- Beta release in July 2009

This still seems complicated...



Context injected by framework into plain application objects

Using context API directly
Power users

Frameworks

Creating contexts
IComputedValue
runAndTrack