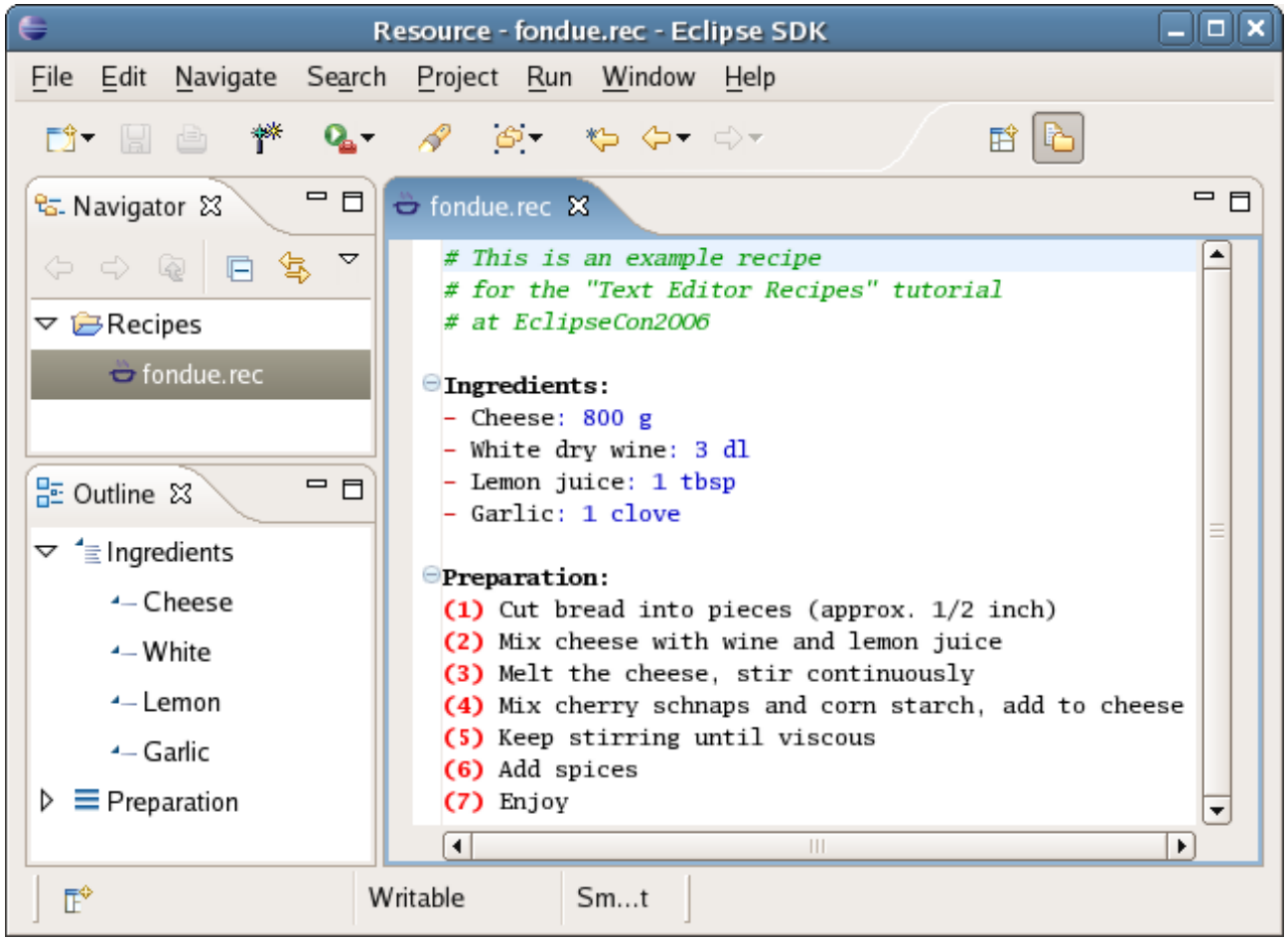
Text Editor Recipes

Season's recipes for your text editor

Tom Eicher, IBM Eclipse Team

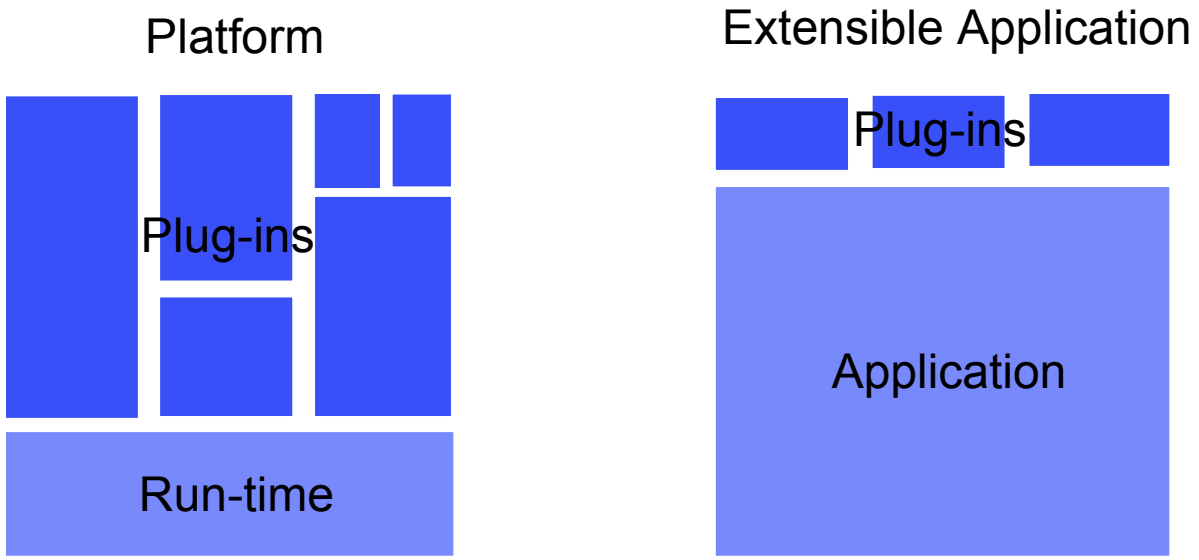
Goal for Today



Outline

- **Lesson 1: A Text Editor in 10 Minutes**
  - **Eclipse plug-ins revisited**
  - **No Java™ code**
  - **Check out all the stuff we got for free!**
- Lesson 2: Add Features
  - Syntax coloring
  - Content assist
- Lesson 3: Create and reconcile a text model
  - Spell checking
  - Outline
  - Folding

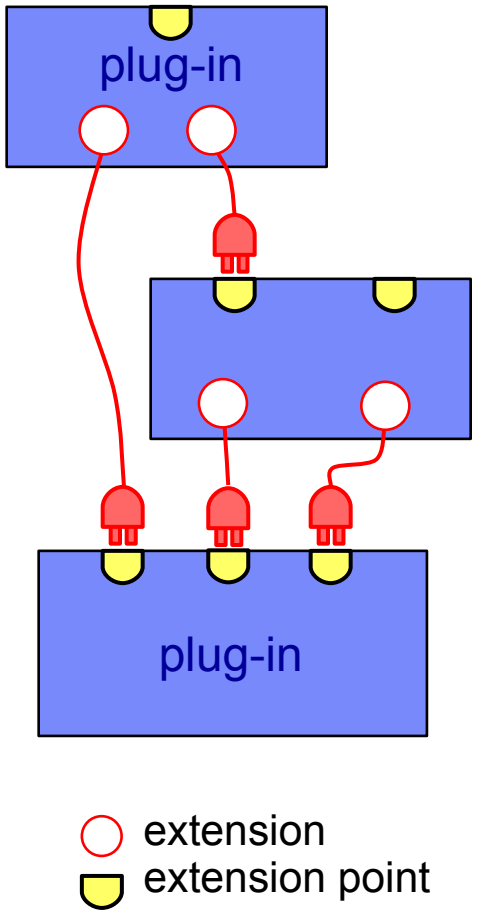
Platform vs. Extensible Application



- Eclipse is a platform with a small runtime kernel, which is an OSGi implementation

Eclipse Plug-in Architecture

- **Plug-in – set of contributions**
  - Smallest unit of Eclipse functionality
  - Big example: HTML editor
  - Small example: action that creates zip files
- **Extension point –** named entity for collecting contributions
  - Example: extension point for workbench preference UI
- **Extension –** a contribution
  - Example: specific HTML editor preferences



Contribute a Text Editor

- Contribute an editor by re-using existing implementations
  - Needs some searching around
  - Get the dependencies right
  - Existing editor class: **`TextEditor`**
  - Need to provide an icon
- Connect the menu and toolbar
  - Existing implementation: **`TextEditorActionContributor`**

Outline

- Lesson 1: A Text Editor in 10 Minutes
- **Lesson 2: Add Features**
  - **Choose an editor superclass**
  - **Create a SourceViewerConfiguration**
    - **Syntax coloring**
    - **Content assist**
    - **Templates**
- Lesson 3: Create and reconcile a text model

Configuration Areas

- Editor superclass
- Editor action bar contributor
- Source viewer configuration
- Text model: partitioning

Create Our Own Editor Class

- Text editor hierarchy
  - **`AbstractTextEditor`**
    - Find/Replace, URL hyperlink navigation, …
    - RCP
  - **`StatusTextEditor`**
    - Show status inside the editor area (unrelated to status line)
  - **`AbstractDecoratedTextEditor`**
    - Rulers, line numbers, quick diff, …
    - Support for General > Editors > Text Editors preferences
  - **`TextEditor`**
    - the default Eclipse text editor

What about the Java Editor?

- Most of the Java editing code is internal
- API
  - `JavaSourceViewerConfiguration`
  - some actions

 |

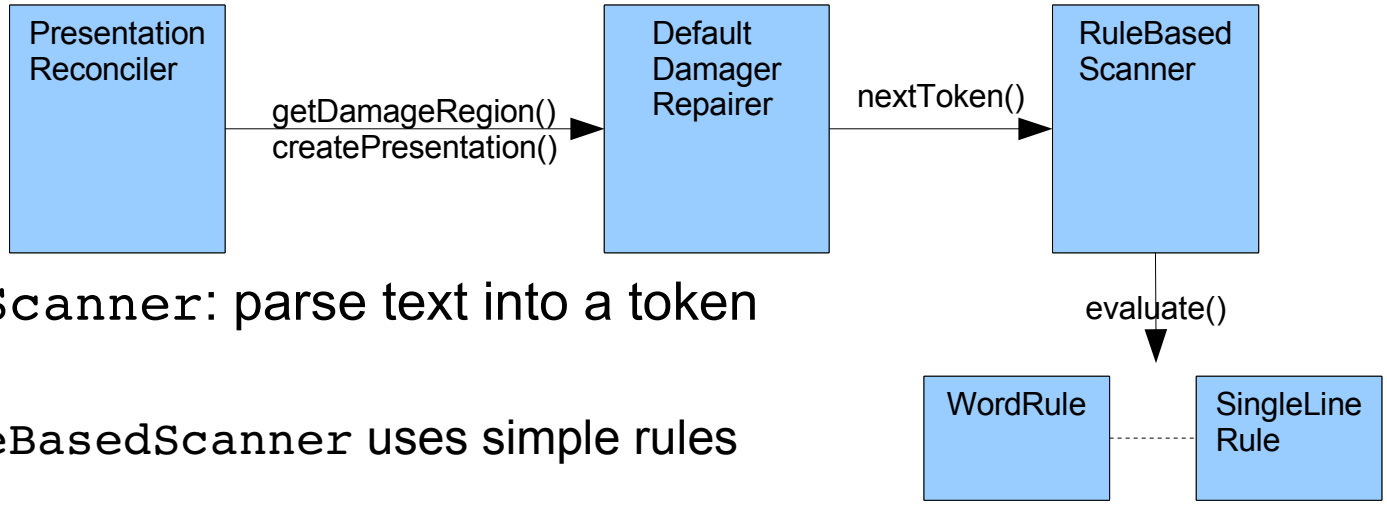
Source Viewer Configuration

- Bundles the configuration space of a source viewer
  - Presentation reconciler (syntax coloring)
  - Content assist
  - Hovers
  - Formatter
  - ...
- Many features can be provided separately for each ***partition type***

Syntax Highlighting: Damage & Repair

```
# Fondue!

Ingredients:
- Cheese: 1lb # Swiss cheese!

Preparation:
(1) Melt it
(2) Eat it
```

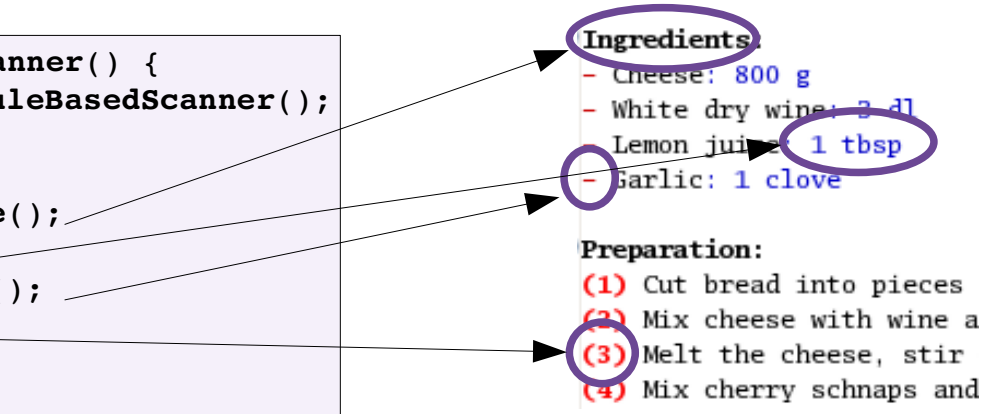
- `PresentationReconciler`
  - `IPresentationDamager`: define dirty region given a text change
  - `IPresentationRepairer`: recreate presentation for dirty region
  - `DefaultDamagerRepairer` does both, based on a token scanner



- `ITokenScanner`: parse text into a token stream
  - `RuleBasedScanner` uses simple rules

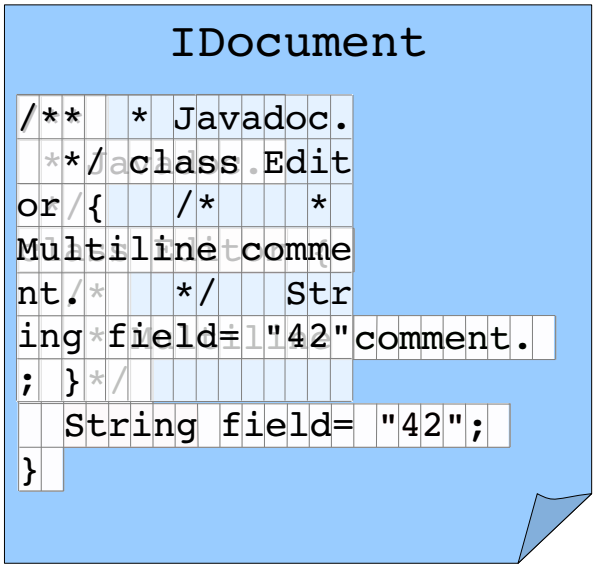
Set the Rules

```
private ITokenScanner getRecipeScanner() {
  RuleBasedScanner scanner= new RuleBasedScanner();

  IRule[] rules= new IRule[4];
  rules[0]= createSectionTitleRule();
  rules[1]= createQuantityRule();
  rules[2]= createLeadingDashRule();
  rules[3]= createStepRule();

  scanner.setRules(rules);
  return scanner;
}
```




```
private IRule createLeadingDashRule() {
  IToken dashToken= new Token(
    new TextAttribute(
      fColors.getColor(new RGB(200, 100, 100)),   // foreground
      null,                                       // background
      SWT.BOLD)                                   // style
  );
  WordRule wordRule= new WordRule(new SimpleWordDetector());
  wordRule.addWord("-", dashToken);
  wordRule.setColumnConstraint(0);
  return wordRule;
}
```

The `IDocument` Text Model



- Sequence of characters
  - Supports random access and replace
  - Event notifications via **`IDocumentListener`**
- Sequence of lines
  - Query by offset and line number
- Positions
  - Ranges that are adjusted to modifcations
  - **`IPositionUpdater`** strategies handles overlapping changes
- Partitions
  - Slice the document into segments of the same **content type**
  - Language dependent – a simple semantic model

Document Partitioning

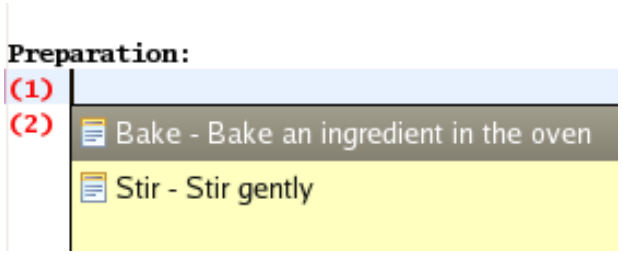
```
/**
 * Javadoc.
 */
class Editor {
  /*
   * Multiline comment.
   */
  String field= "42";
}
```

- Partitioning is always up-to-date
- Document provider ensures that the partitioning is installed
  - Documents support multiple partitionings
  - Document setup can also be managed by the file buffer manager (`o.e.core.filebuffers.documentSetup`)
  - ➔ File buffer document setup should only be used if the partitioning is considered of interest for non-UI clients and never contribute the default partitioning
- **`SourceViewerConfiguration`** needs to know the partitioning and supported partition types.

Partitioning a Document

```
class RecipeDocumentProvider extends FileDocumentProvider {

  protected void setupDocument(Object element, IDocument document) {
    if (document instanceof IDocumentExtension3) {
      IDocumentExtension3 ext= (IDocumentExtension3) document;
      IDocumentPartitioner partitioner= createRecipePartitioner();
      ext.setDocumentPartitioner(RECIPE_PARTITIONING, partitioner);
      partitioner.connect(document);
    }
  }

  private IDocumentPartitioner createRecipePartitioner() {
    IPredicateRule[] rules= {
        new SingleLineRule("#", null, new Token(RECIPE_COMMENT))
    };
    RuleBasedPartitionScanner scanner= new RuleBasedPartitionScanner();
    scanner.setPredicateRules(rules);
    return new FastPartitioner(scanner, CONTENT_TYPES);
  }
}
```

Content Assist



- Provides user help
- Important discovery tool
- Various degrees of smartness
- Different proposal types
- Let’s do it

Content Assist: Processor computes proposals

```
public IContentAssistant getContentAssistant(ISourceViewer sourceViewer) {
  ContentAssistant assistant= new ContentAssistant();
  assistant.setDocumentPartitioning(RECIPE_PARTITIONING);
  assistant.setContentAssistProcessor(
    new HippieProposalProcessor(), RECIPE_COMMENT);
  return assistant;
}
```

- The simplest proposal type: hippie proposals
  - complete a prefix to a word in any open text editor

Content Assist: Hook an Action

- Create an action
  - Register it with the editor
- Commands
  - Used to be called action definitions

```
protected void createActions() {
  super.createActions();

  IAction action= new ContentAssistAction(...);
  action.setActionDefinitionId(ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS);
  setAction("ContentAssist", action);
}
```

Editor Action Bar Contributor

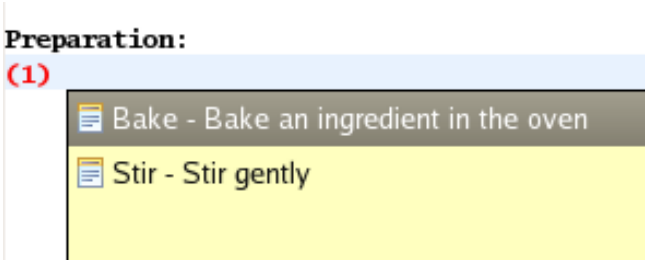
- Connects the editor to tool bar, menu bar and status line
- Shared by all editors of a kind
- **`BasicTextEditorActionContributor`** connects basic features
  - Status line contributions
  - Cut/copy/paste
  - Word completion
  - Find/replace, incremental find
- **`TextEditorActionContributor`** provides
  - Add/remove task/bookmark
  - Annotation navigation
  - Menu entry to change the encoding

Content Assist: Hook an Action

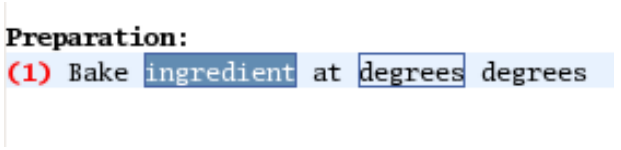
- Action Contributor
  - Actions for all editors of a kind
  - **`RetargetAction`** redirects menu actions to the active editor
  - Menu actions
  - Toolbar

```
private RetargetTextEditorAction fContentAssist;

public RecipeEditorActionContributor() {
  fContentAssist= new RetargetTextEditorAction();
  String cmd= ITextEditorActionDefinitionIds.CONTENT_ASSIST_PROPOSALS;
  fContentAssist.setActionDefinitionId(cmd);
}

@Override
public void contributeToMenu(IMenuManager menu) {
  IMenuManager editMenu= menu.findMenuUsingPath(M_EDIT);
  editMenu.appendToGroup(MB_ADDITIONS, fContentAssist);
}

@Override
public void setActiveEditor(IEditorPart part) {
  IAction editorAction= getAction(part, "ContentAssist");
  fContentAssist.setAction(editorAction);
}
```

Templates



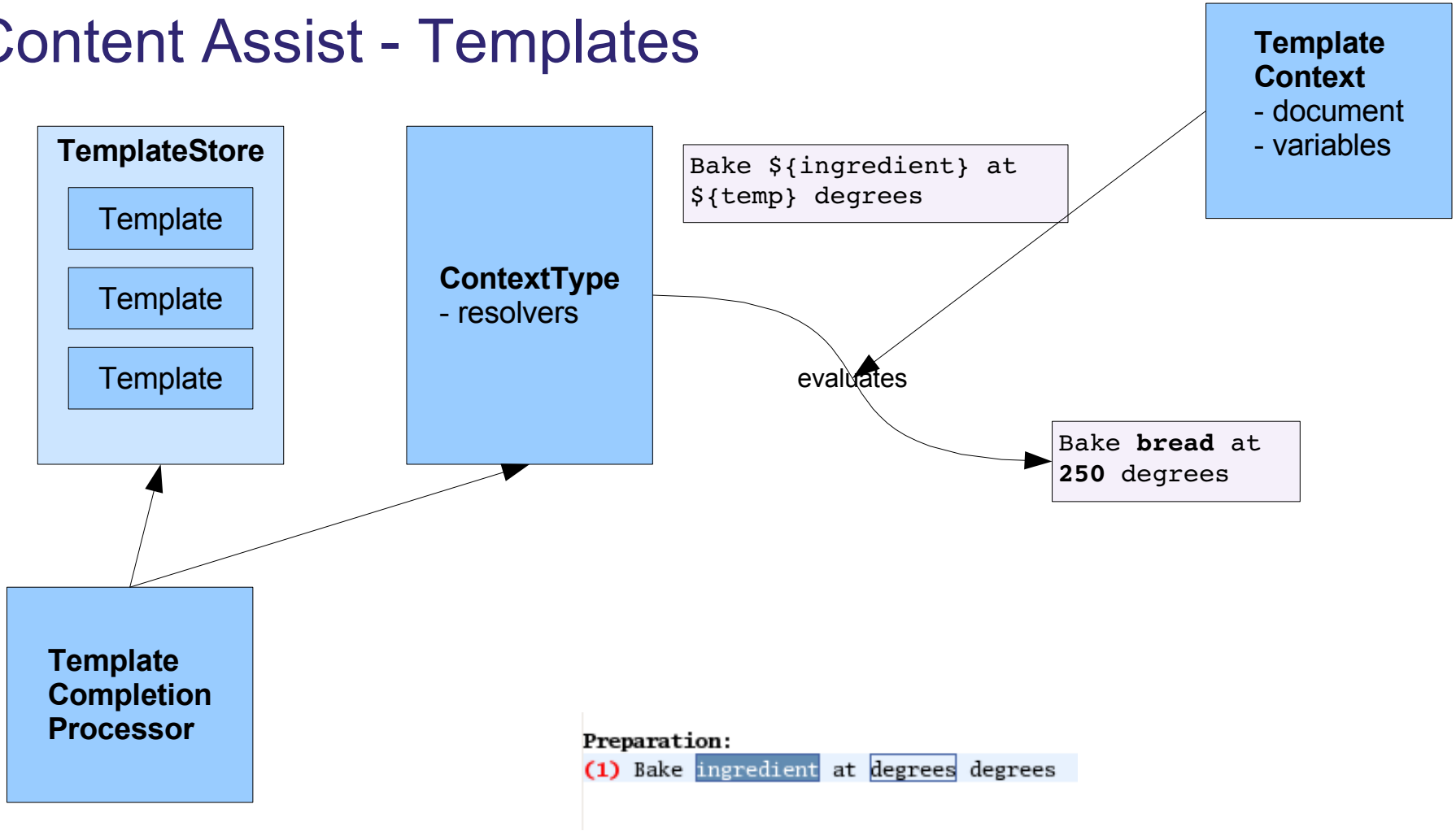
- Shown via Content Assist
  - Canned snippet for reoccurring patterns
  - Presents a lightweight input mask (stencil) when inserted
  - Additional templates can be contributed by other plug-ins
  - User may modify the templates
- Toolkit exists, but needs glue code
  - Singleton template store and context registry
  - Preference store
  - Image management



Templates – the Easy Way

```
class RecipeCompletionProcessor implements IContentAssistProcessor {
  private static final String CONTEXT_ID= "preparation"; //$NON-NLS-1$

  TemplateContextType fContextType= new TemplateContextType(
        CONTEXT_ID,
        "Preparation Templates");

  Template fTemplate= new Template(
      "Stir", // name
      "Stir gently", // description
      CONTEXT_ID,
      "Stir ${ingredient} gently", // pattern
      false); // image

  public ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset) {
    IDocument document= viewer.getDocument();
    Region region= new Region(offset, 0);
    TemplateContext context= new DocumentTemplateContext(fContextType, document, offset, 0);
    TemplateProposal proposal= new TemplateProposal(fTemplate, templateContext, region, null);

    ICompletionProposal[] result= { templateProposal };
    return result;
  }
}
```

Content Assist - Templates



Outline

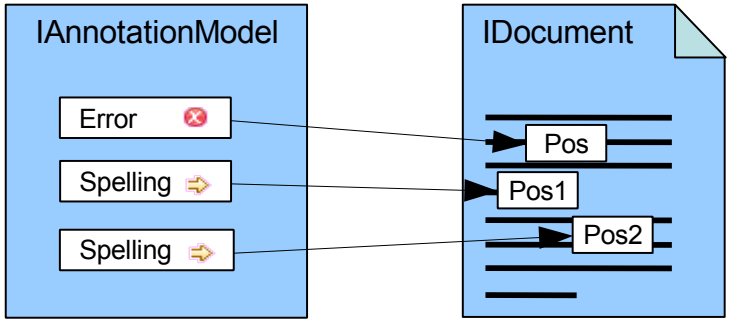
- Lesson 1: A Text Editor in 5 Minutes
- Lesson 2: Add Features
- **Lesson 3: Create a text model...**
  - **... and reconcile it**
  - **Spell checking**
  - **Outline**
  - **Folding**

Reconcile as You Type

- Mark errors and update the model while typing
- Too costly to be updated in the UI thread
- Reconciler runs analysis in separate thread (one per editor)
  - Kicked off after a typing break
  - Interrupted when the user types again
  - Synchronization!
  - Default implementation: **`MonoReconciler`**
  - Write our own: **`IReconcilingStrategy`**
- Reports back its results
  - Annotate the text model: **`IAnnotationModel`**

Annotation Model: Positional information

- Text Model consists of **`IDocument`** and **`IAnnotationModel`**
- Annotations are based on the position tracking in **`IDocument`**
- An Annotation is a piece of Information attached to a text location
- Annotations
  - have a type
  - may implement IAnnotationPresentation to provide an image



Spell Checking

- Needs a spelling engine
  - There is no dictionary in eclipse
  - CD contains a plug-in that contributes a spelling engine
  - Select the spelling engine in the preferences
- Annotation type for spelling problems is defined by the `org.eclipse.ui.editors` plug-in

```
public class RecipeReconcileStrategy implements IReconcilingStrategy {
  private List collectSpellingProblems() {
    SpellingService service= EditorsUI.getSpellingService();
    SpellingProblemCollector collector= new SpellingProblemCollector();
    service.check(fDocument, new SpellingContext(), collector, fMonitor);
    List problems= collector.getProblems();
    return problems;
  }
}
```

SpellingReconcilingStrategy

```
Annotation[] fPreviousAnnotations;
void reconcile() {
  List problems = collectSpellingProblems();
  Map annotations = createAnnotations(problems);

  IAnnotationModel model= fSourceViewer.getAnnotationModel();
  model.replaceAnnotations(fPreviousAnnotations, annotations);

  fPreviousAnnotations= annotations.keySet().toArray();
}

private static final String SPELLING_ID= "org.eclipse.ui.workbench.texteditor.spelling";

private Map createAnnotations(List problems) {
  Map annotations= new HashMap();
  for (Iterator it= problems.iterator(); it.hasNext();) {
    SpellingProblem problem= (SpellingProblem) it.next();
    Annotation annotation= new Annotation(SPELLING_ID, false, problem.getMessage());
    Position position= new Position(problem.getOffset(), problem.getLength() + 1);
    annotations.put(annotation, position);
  }
  return annotations;
}
```

Update Problem Annotations While Typing

- The harder problems:
  - Mapping temporary problems to markers
    - Markers: persistable information tags generated by the builder.
  - Mapping new problems to existing problem annotations
- Defining your own annotation types is a good idea

How to Add a Custom Annotation

- **`org.eclipse.ui.editors.annotationTypes`** extension-point
  - Defines a new annotation type
  - Allows to map an annotation type to a marker type

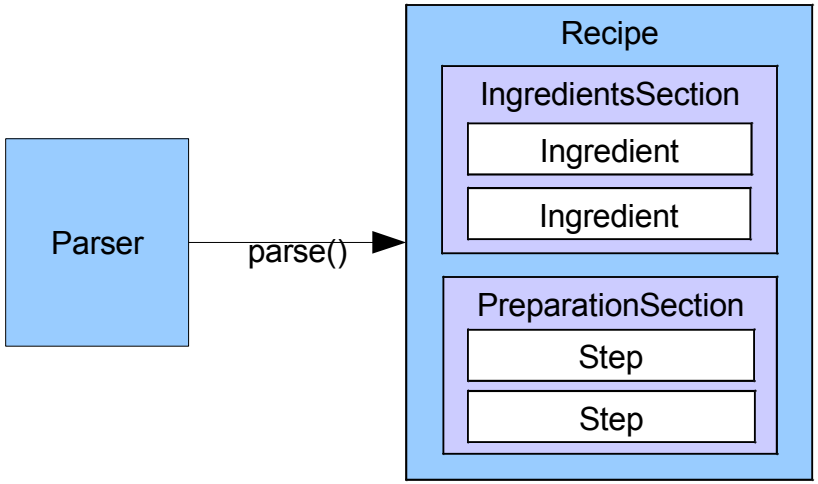
- **`org.eclipse.ui.editors.markerAnnotationSpecification`** extension-point
  - Defines where and how this annotation type is shown
  - Defines default values
  - Controls whether the annotation type appears on the **`General > Editors > Text Editors > Annotations`** preference page

Adding Domain Model Based Features

- Create a domain model
  - Takes too long for us – use the provided recipe parser
- Populate the **Outline View**
- **Folding** of document structures

The Recipe Model

- Creating a simple parser for recipes
  - Takes too long for this tutorial
  - Use the `org.recipeeditor.model` plug-in provided
  - Re-create the recipe in the reconciler



Reconciling the Model

```
public void reconcile(IRegion partition) {
  Recipe recipe= new RecipeParser().parse(fDocument);
  fEditor.setRecipe(recipe); // update model
  checkSpelling();
}
```

Adding an Outline

- Outline view is defined by Eclipse (the IDE)
- Outline content is a **`IContentOutlinePage`**
  - Editor provides its outline via **`IEditor.getAdapter()`**
- Editor knows its outline page and updates it
  - On input change
  - On selection change, if back linking is supported and enabled
- Outline is normally updated upon model changes
- Content outline page is built using JFace or SWT components

Creating an Outline Page

```
class RecipeOutlinePage extends ContentOutlinePage {

  public void createControl(Composite parent) {
    super.createControl(parent);
    TreeViewer viewer= getTreeViewer();

    ITreeContentProvider contentProvider= new RecipeOutlineContentProvider();
    viewer.setContentProvider(contentProvider);
  }

  public void updateRecipeModel(final Recipe recipe) {
    runInSWTThread(viewer, new Runnable() {
      public void run() {
        TreeViewer viewer= getTreeViewer();
        if (viewer != null)
          viewer.setInput(recipe);
      }
    });
  }
}
```

Projection Support (1/3)

- Editor creates a **`ProjectionViewer`** instead of a **`SourceViewer`**
  - Work around framework oversights...
- Editor installs the projection support:

```
public void createPartControl(Composite parent) {
  super.createPartControl(parent);

  ProjectionViewer projectionViewer= (ProjectionViewer) getSourceViewer();
  fProjectionSupport= new ProjectionSupport(projectionViewer, ...);
  fProjectionSupport.install();
  projectionViewer.doOperation(ProjectionViewer.TOGGLE);
}
```

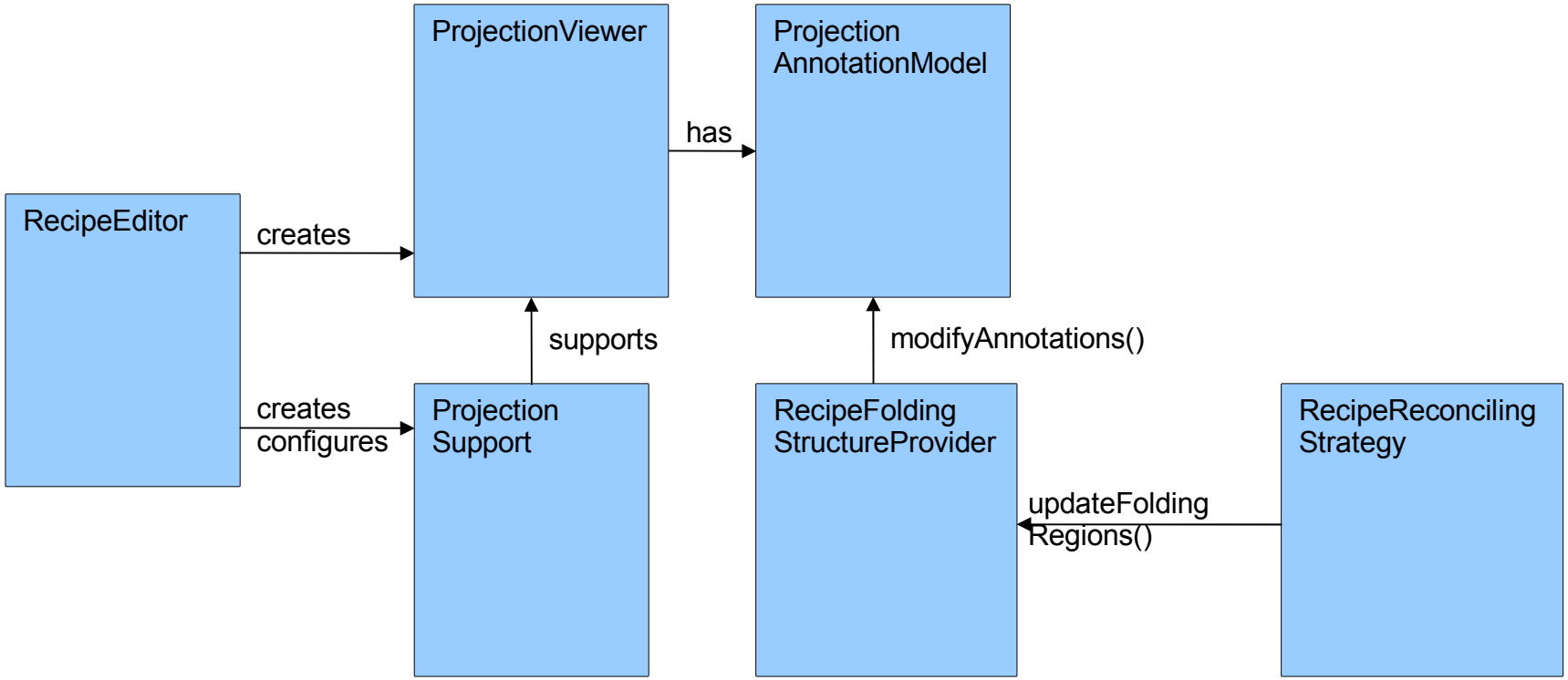
- Editor provides access to the projection annotation model
  - Forward **`ITextEditor.getAdapter()`** to the projection support

Projection Support (2/3)

- A folding structure updater updates the projection model
- Updater is installed on a text editor or projection viewer
- Run the structure updater in the reconciler
- The folding structure updater
  - Computes the folding structure difference
  - Updates the projection annotations in the projection annotation model

 |

Projection Support (3/3)



Thank You – Questions

Legal Notices

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.