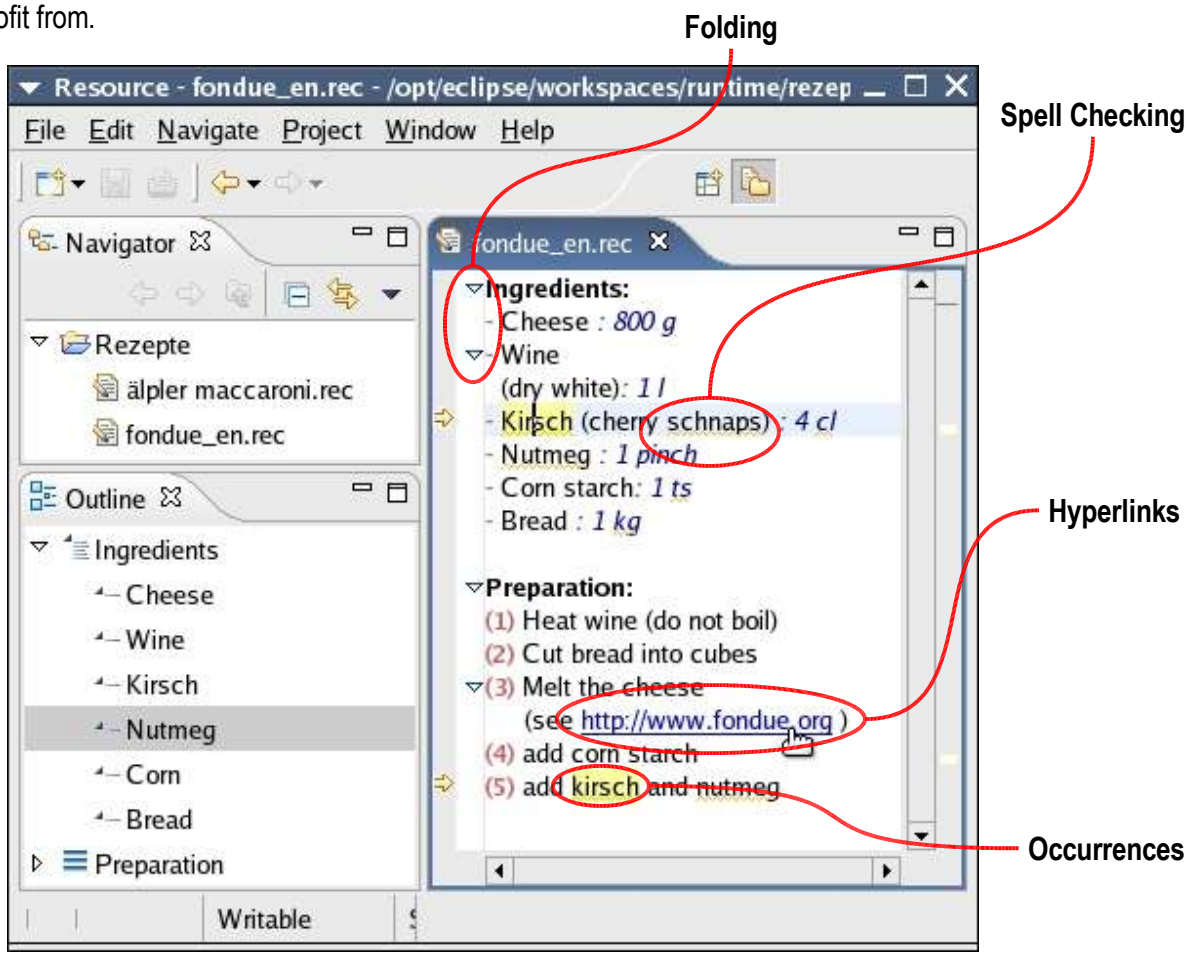
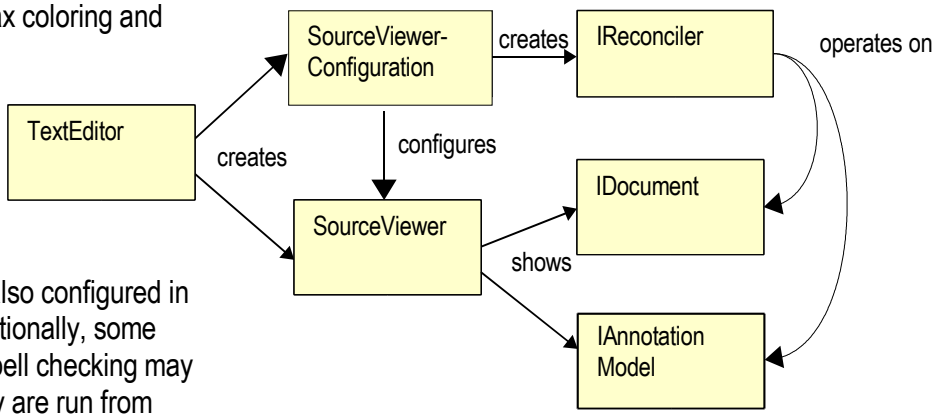
Season's Text Editor Recipes

New Editor Features in Eclipse 3.1
Tom Eicher, IBM Resarch, JDT / Platform Text

Eclipse 3.1 adds a number of additional features that any text editor can profit from.



A source code editor uses a `SourceViewer` (view) to display an `IDocument` (text model) and associated `Annotations`. A `SourceViewerConfiguration` configures most aspects of the viewer, like syntax coloring and content assist.



The new features in Eclipse 3.1 are also configured in the source viewer configuration. Additionally, some operations like model creation and spell checking may take a long time to run, therefore they are run from inside a separate thread, the `Reconciler`.

Hyperlinks

Web Hyperlinks

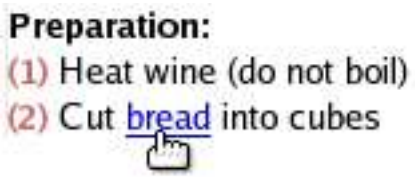
Eclipse 3.1 text editors support hyperlink navigation. To make web URLs clickable, our `SourceViewerConfiguration` needs to create an `IHyperLinkPresenter` configured with a default color.

RecipeSourceViewer-Configuration

```
@Override
public IHyperlinkPresenter
getHyperlinkPresenter(ISourceViewer sourceViewer) {
   RGB linkColor= new RGB(0, 0, 200);
   return new DefaultHyperlinkPresenter(linkColor);
}
```

Custom Hyperlinks

To support custom links, for example to let the user navigate from a reference to a declaration, an `IHyperlinkDetector` is registered in `SourceViewerConfiguration`. The detector creates instances of a custom `IHyperlink` implementation that will be executed when the user selects the link.



RecipeSourceViewer-Configuration

```
@Override
public IHyperlinkDetector[]
getHyperlinkDetectors(ISourceViewer sourceViewer) {
   return new IHyperlinkDetector[] {
         new URLHyperlinkDetector(sourceViewer),
         new IngredientLinkDetector(fEditor)};
}
```

creates

IngredientLinkDetector

```
@Override
public IHyperlink[]
detectHyperlinks(ITextViewer viewer, IRegion region, ...) {
   IDocument document= viewer.getDocument();
   IRegion linkRegion= computeWordRegion(document, region);
   String possibleLink= extractWord(document, linkRegion);
   IRegion target= resolveIngredientDeclaration(possibleLink);
   return new IHyperlink[] {
         new IngredientHyperLink(viewer, linkRegion, target) };
}
```

creates

IngredientHyperLink

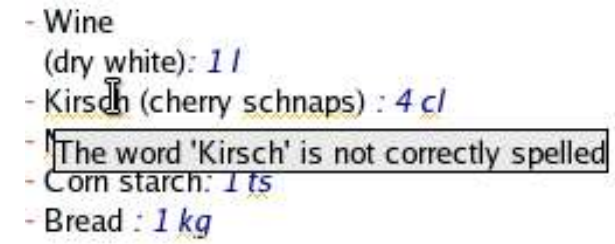
```
@Override
public void open() {
   fTextViewer.setSelectedRange(fTargetRegion.getOffset(),
                                fTargetRegion.getLength());
}
```

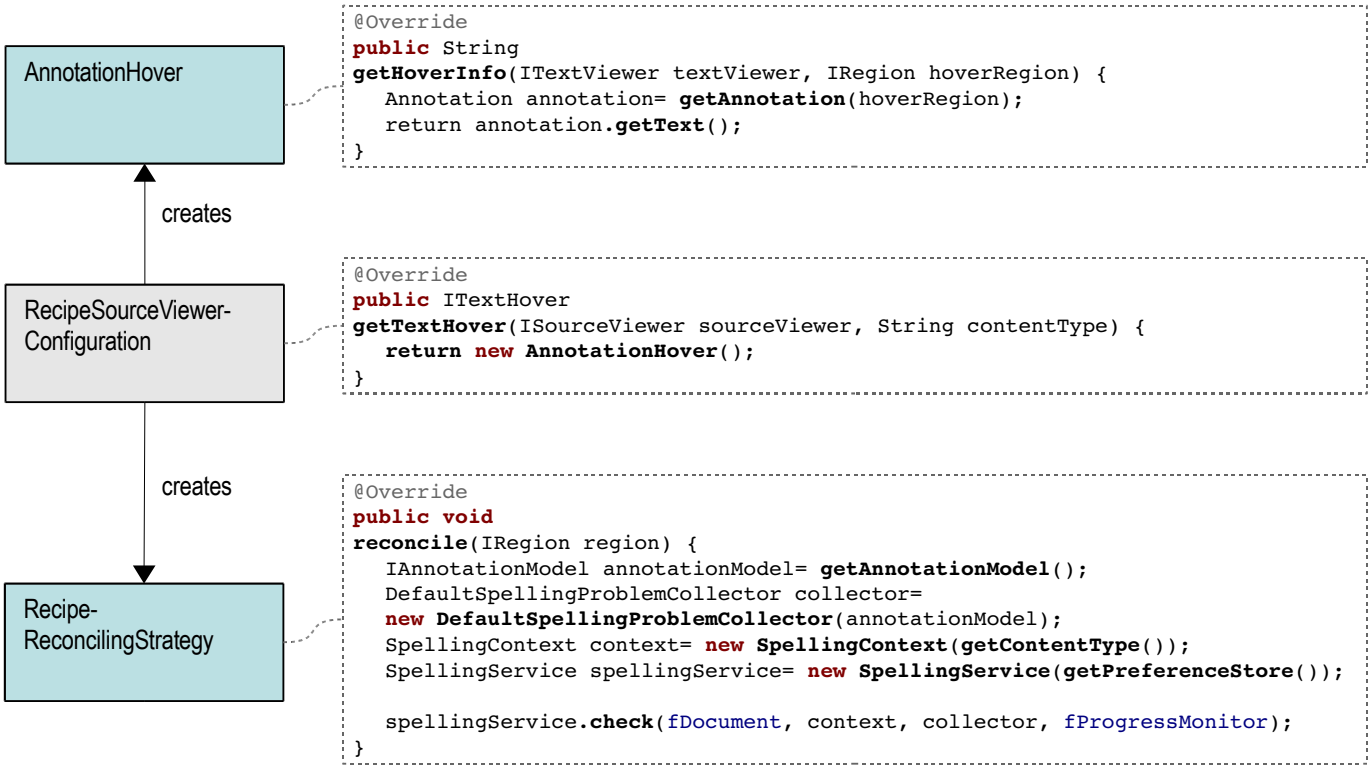
Spell Checking

Eclipse 3.1 contains a framework for spell checking that text editor implementors can leverage. You can either create your own `SpellingService` or obtain the default from `EditorsUI.getSpellingService`.

Since spell checking is a potentially long-running operation, it should be performed in a background thread, preferably in the reconciler.

In order to display hovers for spelling problems, `SourceViewerDecoration` installs an `AnnotationHover`.





Note: eclipse does not contain a spelling dictionary. Spell checking needs a dictionary to work.

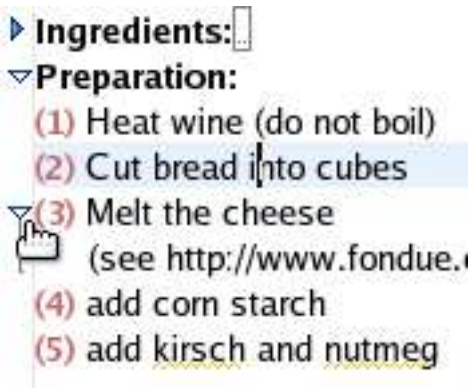
Folding

For complex source files, it may be desireable to fold certain regions of text, so that the user can focus on the interesting parts.

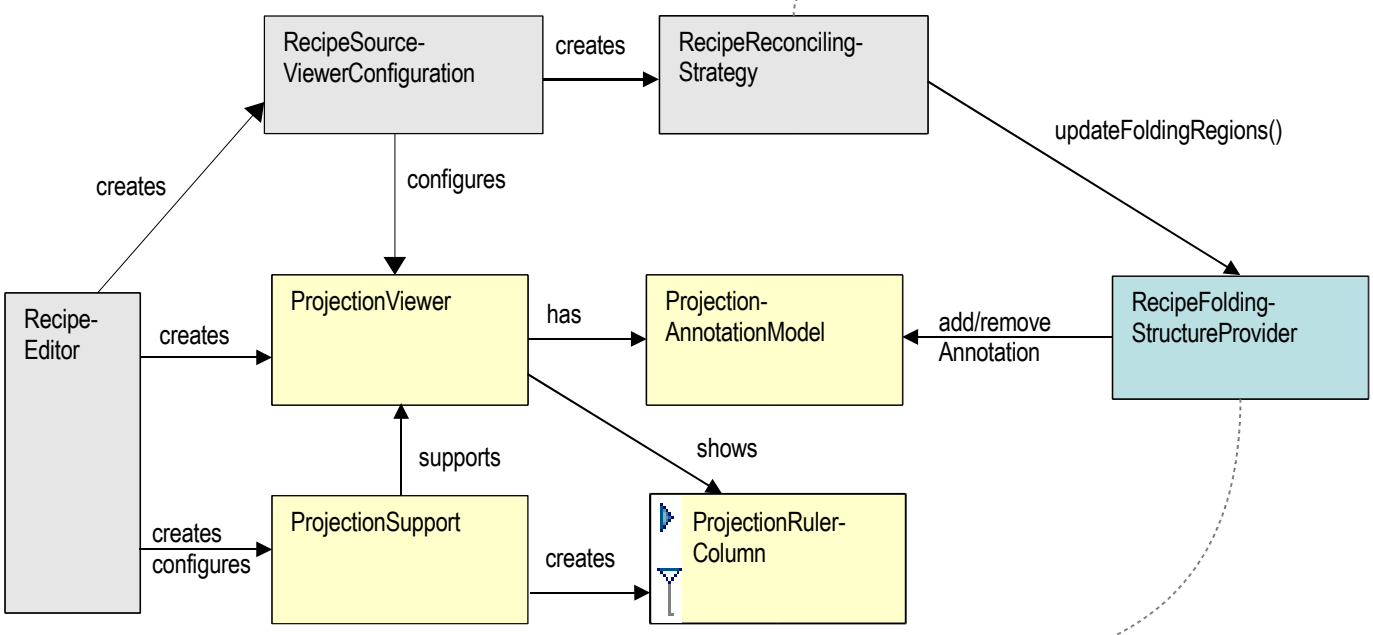
In Eclipse, folding regions are defined using a special kind of `Annotation`. Since computing the folding structure may be a complex process, it is best performed in a background thread, ideally in the reconciler.

The steps performed in a folding structure provider are as follows:

- compute the new folding structure
- compare the structure with the currently displayed structure and compute the delta
- update the `AnnotationModel` with the delta




```
@Override
public void
reconcile(IRegion region) {
  Recipe recipe= parseRecipe();
  fFoldingStructureProvider.updateFoldingRegions(recipe);
}
```



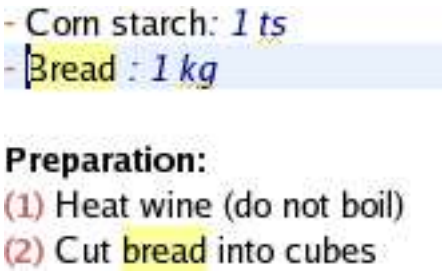

```
void updateFoldingRegions(Recipe recipe, ProjectionAnnotationModel model) {
  Set<Position> structure= createFoldingStructure(recipe);
  Annotation[] deletions= computeDeletions(model, structure);
  Map<Annotation,Position> additions= computeAdditions(model, structure);
  model.modifyAnnotations(deletions, additions, EMPTY);
}
```

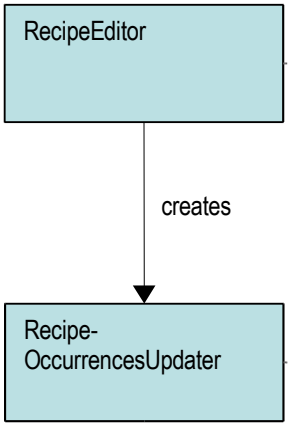
Mark Occurrences

Similar to the Java editor, we would like to highlight occurrences of certain syntax elements.

The `RecipeOccurrencesUpdater` registers as post selection listener. Whenever the selection changes, the updater checks if the current selection is an ingredient; if yes, any other occurrences in the document are colored.

The colored regions are `Annotation` objects on the document's annotation model.






```
@Override
public void createPartControl(Composite parent) {
  super.createPartControl(parent);
  fOccurrencesUpdater= new RecipeOccurrencesUpdater(this);
}
```

```
public
RecipeOccurrencesUpdater(RecipeEditor editor) {
  ((IPostSelectionProvider) editor.getSelectionProvider()).
      addPostSelectionChangedListener(this);
  fEditor= editor;
}

@Override
public void selectionChanged(SelectionChangedEvent event) {
  ISourceViewer viewer= (ISourceViewer) event.getSource();
  IDocument document= viewer.getDocument();
  IAnnotationModel model= viewer.getAnnotationModel();

  removeOldAnnotations(model);

  String word= getWordAtSelection(document);
  if (isIngredient(word))
    createNewAnnotations(word, document, model);
}
```