Design NetUICompiler

NetUI Compiler

Introduction

The NetUI compiler layer consists of annotation processors for the following types of files:

- page flow controllers
- shared flow controllers
- form beans
- JavaServer Faces backing beans

Annotation processing runs under both apt and XDoclet, although apt (processing true Java 5 annotations) is currently the only supported tool.

The architecture involves several layers:

- **compiler-core**: The bulk of the annotation processing code; has no ties to either apt or XDoclet.
  - **typesystem**: Generic typesystem interfaces; are implemented as wrappers by the apt and XDoclet layers. The rest of the core is built on these interfaces.
  - **processor**: Core code that kicks off annotation processing for various file types.
  - **grammar**: Annotatation rules and custom attribute types to support the Checking phase of annotation processing.
  - **model**: The model that is built up for generated files. Once this is built up, it is used to write out the actual XML files that are generated.
  - **genmodel**: The classes here extend the classes in **model**, and are also aware of annotations. They are used to create model classes based on annotations in the source files.
- **compiler-apt**: Thin layer that runs annotation processing from within Sun's apt (or from within an IDE that implements Sun's Mirror interfaces, which are compile-time representations of types, annotations, etc.).
  - **apt**: actual apt `AnnotationProcessorFactory`(s) that use delegating `AnnotationProcessor`(s) to delegate to the processors in the core layer.
  - **typesystem.impl**: Mirror-based implementations of the **typesystem** interfaces from the core layer.
- **compiler-xdoclet**: Thin layer that runs annotation processing from within XDoclet.
  - *xdoclet*: Actual XDoclet task/subtask to delegate to the core annotation processors.
  - **typesystem.impl**: XDoclet-based implementations of the **typesystem** interfaces from the core layer.

Details

typesystem in compiler-core

The typesystem looks very much like Sun's Mirror interfaces. It contains the following sets of interfaces:

- **declaration**: Type declarations and everything underneath. These are the bulk of the interesting elements when processing classes. If you have a `ClassDeclaration`, for instance, you can get access to any annotations on the class, any method/field declarations within the class, etc.
- **type**: These are interfaces for *type instance* (sometimes called a "type usage"). They are different than type declarations because they represent the use of a type within a declaration. For example, this is a type instance in a field declaration, which would produce a `ClassType` for `MyObject`:

  ```
  public MyObject someField;
  ```

  Contrast this to the declaration for `MyObject`, which would produce a `ClassDeclaration`:

  ```
  public class MyObject
  {
      ...
  }
  ```

  In general, if you have a *type usage* (e.g., `ClassType`), you can get to its *type declaration* (e.g., `ClassDeclaration`).
- **env**: This contains base annotation processor environment, which is used to get things like the Messager (for emitting errors and warnings), the Filer (for writing out files), the list of type declarations that are being processed, etc.

processor in compiler-core

The main annotation processors here are `PageFlowAnnotationProcessor` and `FormBeanAnnotationProcessor`. Each one kicks off the appropriate **checker** for the class declaration that is being processed. `PageFlowAnnotationProcessor` chooses the checker (e.g., `PageFlowChecker`) based on the class-level annotation (e.g., @Jpf.Controller) and the base class for the one being processed (e.g., PageFlowController).

Each checker (all extend `BaseChecker`) is the starting point for checking the class, which mostly consists of delegating to the right annotation grammar(s) for the class, and for each method as appropriate.

grammar in compiler-core

The classes here extend `AnnotationGrammar`, to provide rules for annotations, and `AnnotationMemberType`, to provide rules for attributes (members) in annotations. For example, `ActionGrammar` provides rules for the @Jpf.Action annotation, and `JavaIdentifierType` provides checking for an annotation attribute that must be a valid Java identifier, like returnAction on @Jpf.Forward.

In general, annotation grammars provide five main things, which are used by the base `AnnotationGrammar` class:

- Declarations for member types (attributes) and nested annotation grammars, provided by `addMemberType`, `addMemberGrammar`, and `addMemberArrayGrammar`.
- **Mutually-exclusive attributes** (`getMutuallyExclusiveAttrs`): a set of arrays of attribute names that are mutually-exclusive, like outputFormBean and outputFormBeanType on @Jpf.Forward.
- **Required attributes** (`getRequiredAttrs`): a set of *arrays*, where each array specifies that **at least one of a group** of attributes is required. For instance at least one of path, navigateTo, returnAction, action, or tilesDefinition is required on @Jpf.Forward.
- **Attribute dependencies** (`getAttrDependenciese`): a set of arrays that describe attribute dependencies. The first element is the attribute in question, which is only allowed if at least one of the rest of the attributes is present. For example on @Jpf.Forward, the externalRedirect attribute is only allowed if the path is present.
- Custom checking, in `onBeginCheck`, `onCheckMember`, and `onEndCheck`.

Annotation member types simply provide custom checking by overriding `onCheck`.

All of the grammar/type checking is kicked off during the `check` phase; see `BaseAnnotationProcessor`.

genmodel in compiler-core

During the `generate` phase of annotation processing (see `BaseAnnotationProcessor`), instances of classes in **genmodel** are created based on declarations for annotated classes being processed. As an example, see `GenStrutsApp`, which accepts a `ClassDeclaration` (the annotated page flow or shared flow class) in its constructor. It builds up a model based on the annotations, using its base class `StrutsApp` setters. Once the model is built up, it is written to XML using base class methods in `StrutsApp`.

Future Directions

As Eddie has suggested, some of this code could move to a Jakarta Commons-type project. This would include the **typesystem** package in **compiler-core**, some basic annotation grammar/type classes, and some skeleton code for apt {{AnnotationProcessor}}s and XDoclet tasks from **compiler-apt** and **compiler-xdoclet**, respectively.