Design and Architecture of Eclipse and JDT

Martin Aeschlimann  
IBM Research, Switzerland  
martin_aeschlimann@ch.ibm.com

Tobias Widmer  
IBM Research, Switzerland  
tobias_widmer@ch.ibm.com
outline

- short introduction to eclipse
  - what are plug-ins?
  - what are the role of APIs?

- a design tour through JDT
  - Java™ model
  - search engine and type hierarchy
  - abstract syntax tree (AST)
  - J2SE 5.0
what is eclipse?

- an IDE and more…
  - it’s a Java development environment (JDT)
  - it’s a general tools and integration platform
  - it’s a general application platform (RCP)
  - it’s a scalable server platform (Equinox)

- an open source community

- an ecosystem to enable a integrated solution
  - including products by some major tool vendors

- a foundation to advance the eclipse platform
eclipse perception changes

3.3 “eclipse is a universal platform for rich client & server applications”

3.2 “eclipse is an even better application and tools platform and Java IDE

3.0/3.1 “eclipse is a general application platform”

2.0 “eclipse is a general tooling platform”

1.0 “eclipse is a Java IDE”

⇒ broader application improves the platform
platform vs. extensible application

- eclipse rich client platform
  - it has an open, extensible architecture
  - built out of layers of plug-ins

platform

```
plugins
runtime
```

extensible app

```
plugins
application
```
eclipse plugin architecture

- plugin = component
  - set of contributions
  - smallest unit of eclipse functionality
  - details spelled out in plugin manifest

- extension point – entity for collecting contributions
  - example: extension point to add additional spam filtering tools

- extension – a contribution
  - example: a specific spam filter tool

- rcp - platform – set of standard plugins
- runtime – controls and manages contributions
OSGi – Open System Gateway Initiative

- named, versioned bundles
- dependency management
- explicit imports/exports
- built-in security
- dynamic

- independent industry standard
- has become popular in cell phone system management and other areas

→ these are exactly the characteristics we want for advanced rich clients!
manifests – component model - OSGi

Bundle-Name: HP35-OSGi
Bundle-SymbolicName: HP35-OSGi; singleton:=true
Bundle-Version: 1.0.0
Bundle-Activator: HP35.HP35Plugin
Require-Bundle:
  org.eclipse.ui.views,
  org.eclipse.ui,
  org.eclipse.core.runtime
Eclipse-AutoStart: true
Export-Package:
  HP35,
  HP35.views

Plug-in identification
Required Plug-ins
exported packages
Declare contribution this plugin makes
manifests – extensions ➔ plugin.xml

```xml
<plugin>
  <extension>
    <view
      name="HP35"
      icon="icons/calculator.jpg"
      category="HP35"
      class="HP35.views.HP35View"
      id="HP35.views.HP35View">
    </view>
  </extension>
  <extension-point id="operators" name="Operators"
    schema="schema/operators.exsd"/>
</plugin>
```

declare contribution this plugin makes

define new extension point open for contributions
contribution cycle

- publish the plugin
  - create a feature
    - groups plug-ins into installable chunks
  - create an update site
    - contains zips for features and plugins

- enable extensions
  - define extension points

→ extenders can extend your extensions
plumbing things together: APIs

- APIs are interfaces with specified and supported behavior
- APIs matter
  - define consistent, concise interface
  - define API conventions (*.internal.* in eclipse)
  - don’t expose the implementation
  - develop implementation and client at the same time
- define APIs for stability
  - binary compatibility is highest priority
  - we would rather provide less API than desired (and augment) than provide the wrong (or unnecessary) API and need to support it indefinitely
API specs

- API specs play many key roles
  - tell client what they need to know to use it
  - tell an implementer how to implement it
  - tell tester about key behaviors to test
  - determines blame in event of failure
level of detail

- trade-off

  - if the specification is too specific or detailed, it is difficult to evolve later on.
  - if the spec is too vague, it is difficult for clients to know the correct usage.

- is the API designed to be implemented or extended by clients?

  - if yes, you need to spell out the contract for implementers/extenders
  - if possible, separate API for clients from API for extenders
API contract language

- RFC on specification language: http://www.ietf.org/rfc/rfc2119.txt

- **must, must not, required, shall**: it is a programmer error for callers not to honor these conditions. If you don’t follow them, you’ll get a runtime exception (or worse)

- **should, should not, recommended**: Implications of not following these conditions need to be specified, and clients need to understand the trade-offs from not following them

- **may, can**: A condition or behavior that is completely optional
API contract language

some eclipse project conventions:

- **not intended**: indicates that you won’t be prohibited from doing something, but you do so at your own risk and without promise of compatibility. Example: “This class is not intended to be subclassed”

- **fail, failure**: a condition where a method will throw a checked exception

- **long-running**: a method that can take a long time, and should never be called in the UI thread

- **internal use only**: an API that exists for a special caller. If you’re not that special caller, don’t touch it
specifications for subclasses

- subclasses may
  - "implement" - the abstract method declared on the subclass must be implemented by a concrete subclass
  - "extend" - the method declared on the subclass must invoke the method on the superclass (exactly once)
  - "re-implement" - the method declared on the subclass must not invoke the method on the superclass
  - "override" - the method declared on the subclass is free to invoke the method on the superclass as it sees fit

- tell subclasses about relationships between methods so that they know what to override
compatibility

- **contract** – are existing contracts still tenable?
- **binary** – do existing binaries still run?
- **source** – does existing source code still compile?
contract compatibility

before:

/** *
 * Returns the current display.
 * @return the display; never null
 */
public Display getDisplay();

• not contract compatible for callers of getDisplay

• contract compatible for getDisplay implementors

after:

/** *
 * Returns the current display, if any.
 * @return the display, or null if none
 */
public Display getDisplay();
contract compatibility

- weaken method preconditions – expect less of callers
  - compatible for callers; breaks implementors

- strengthen method postconditions – promise more to callers
  - compatible for callers; breaks implementors

- strengthen method preconditions – expect more of callers
  - breaks callers; compatible for implementors

- weaken method postconditions – promise less to callers
  - breaks callers; compatible for implementors
questions?

- after the break, we will have an introduction to implementing Eclipse/JDT plugins
Java Development Tooling

Adds Java development tools to Eclipse, by using extension points from the platform

**New views:**
- Package Explorer
- Java Editor
- Type Hierarchy

**New builders:**
- Java Compiler

**Other examples of extensions:**
- Java application launcher
- Java search
- Preference pages, wizards and actions
Java Development Tooling

Offers itself new extension points and APIs for clients

**APIs:**
- Java Model API

**Extension points:**
- Refactoring participants
- Compilation participants
- Search participants

- Quick fix processors
- Class path containers
Java Development Tooling

Separation between ‘Core’ and ‘User Interface (UI)’

**Core:**
- Java model API
- Search Engine API
- Compiler, Compiler AST API

No dependency on UI.

**UI:**
- Views and editors
- Preference pages, wizards and actions
Overview – The JDT Core APIs

**Java Model** – Lightweight model for views
- OK to keep references to it
- Contains unresolved information
- From project to declarations (types, methods..)

**Search Engine**
- Indexes of declarations, references and type hierarchy relationships

**AST** – Precise, fully resolved compiler parse tree
- No references to it must be kept: Clients have to make sure only a limited number of ASTs is loaded at the same time
- Fully resolved information
- From a Java file (‘Compilation Unit’) to identifier tokens
The 3 Pillars – First Pillar: Java Model

**Java Model** – Lightweight model for views
- Java model and its elements
- Java classpath
- Java project settings
- Creating a Java element
- Type hierarchy
- Code resolve

Search Engine

**AST** – Precise, fully resolved compiler parse tree
Java Elements API

IJavaElement form a hierarchy that renders the entire workspace from Java angle

Separated hierarchy from resources:
Using the Java Model

Setting up a Java project

- A Java project is a project with the Java nature set
- Java nature enables the Java builder
- Java builder needs a Java class path

```java
IWorkspaceRoot root = ResourcesPlugin.getWorkspace().getRoot();
IProject project = root.getProject(projectName);
project.create(null);
project.open(null);

IProjectDescription description = project.getDescription();
description.setNatureIds(new String[] { JavaCore.NATURE_ID });
project.setDescription(description, null);

IJavaProject javaProject = JavaCore.create(project);
javaProject.setRawClasspath(classPath, defaultOutputLocation, null);
```

Create a project

Set the Java nature

Set the Java build path
Java Classpath

The Java element hierarchy is defined by the Java classpath: Classpath entries define the roots of package fragments.
## Creating Java Elements

Set the build path

```java
IJavaProject javaProject = JavaCore.create(project);
IClasspathEntry[] buildPath = {
    JavaCore.newSourceEntry(project.getFullPath().append("src")),
    JavaRuntime.getDefaultJREContainerEntry()
};
javaProject.setRawClasspath(buildPath, project.getFullPath().append("bin"), null);
```

Create the source folder

```java
IFolder folder = project.getFolder("src");
folder.create(true, true, null);
```

Create the package fragment

```java
IPackageFragmentRoot srcFolder = javaProject.getPackageFragmentRoot(folder);
Assert.assertTrue(srcFolder.exists()); // resource exists and is on build path
```

Create the compilation unit, including a type

```java
String str =
    "package x.y;" + "\n" +
    "public class E {" + "\n" +
    "  String first;" + "\n" +
    "};";
ICompilationUnit cu = fragment.createCompilationUnit("E.java", str, false, null);
```

Create a field

```java
IType type = cu.getType("E");
type.createField("String name;", null, true, null);
```
Java Project Settings

Configure compiler settings on the project

- Compiler compliance, class file compatibility, source compatibility
- Compiler problems severities (Ignore/Warning/Error)

```java
javaProject.setOption(JavaCore.COMPILER_COMPLIANCE, JavaCore.VERSION_1_5);
```

If not set on the project, taken from the workspace settings

- Project settings persisted (project/.settings/org.eclipse.jdt.core.prefs).
  Shared in a team
- More project specific settings: Formatter, code templates...

See Platform preferences story

- Platform.getPreferencesService()}
Type Hierarchy

- Connect ITypes in a sub/super type relationship
- Used in Hierarchy view
Type Hierarchy - Design Motivation

Subtype hierarchies are expensive to create and maintain.

Why not having an API IType.getSubtypes()?
- Requires to index and resolve all compilation unit in a project and project dependants. Takes minutes for a normal-sized workspace

Why not keep a constantly updated hierarchy in memory?
- Does not scale for big workspaces. Eclipse is not just a Java IDE (can not hold on 10 MB of structure)
- Expensive updating. Every class path change would require types to recheck if they still resolve to the same type

Chosen solution:
- Instantiated hierarchy object
  - Defined life cycle
  - Well known creation costs (sub type relationship is stored in index files)
Code Resolve

- Resolve the element at the given offset and length in the source
  
  ```java
  javaElements= compilationUnit.codeResolve(50, 10);
  ```

- Used for Navigate > Open (F3) and tool tips
The Java Model Offering

Change notification – delta information on model changes

`JavaCore.addElementChangedListener(...)`

Navigation – resolve a name or source reference

`IType type= javaProject.findType("java.util.Vector");
elements= compilationUnit.codeSelect(offset, length);`

Code assist – evaluate completions for a given offset

`compilationUnit.codeComplete(offset, resultRequestor);`

Code formatting

`ToolFactory.createCodeFormatter(options)
   .format(kind, string, offset, length, indentationLevel, lineSeparator);`
The 3 Pillars – Second Pillar: Search Engine

Java Model – Lightweight model for views

Search Engine

- Design motivation
- Using the search engine
- Code example

AST – Precise, fully resolved compiler parse tree
Search Engine – Design Motivation

Need quick access to all references or declaration of a Java element

- Searching for all references of type “A”
- Call graphs
- All types in workspace
Search Engine – Using the APIs

- Creating a search pattern
  
  ```java
  SearchPattern.createPattern("foo*",
      IJavaSearchConstants.FIELD, IJavaSearchConstants.REFERENCES,
      SearchPattern.R_PATTERN_MATCH | SearchPattern.R_CASE_SENSITIVE);
  ```

- Creating a search scope
  
  ```java
  SearchEngine.createWorkspaceScope();
  SearchEngine.createJavaSearchScope(new IJavaElement[]{ project });
  SearchEngine.createHierarchyScope(type);
  ```

- Start the search engine
- Collecting results
  - Subclass SearchRequestor
  - Results are reported as SearchMatch
The 3 Pillars – Third Pillar: AST

Java Model – Lightweight model for views

Search Engine

**AST** – Precise, fully resolved compiler parse tree

- Overall design
- Creating an AST
- AST node details
- Bindings
- AST rewrite
- Refactoring toolkit
Abstract Syntax Tree - Design Motivation

Java Model and type hierarchy are on demand, fault-tolerant and optimized to present model elements in a viewer.

Refactorings and code manipulation features need fully resolved information down to statement level to perform exact code analysis.

Need way to manipulate source code on a higher abstraction than characters

Chosen solution:

- Instantiated abstract syntax tree with all resolved bindings
  - Defined life cycle
  - Well known creation costs

- Abstract syntax tree rewriter to manipulate code on language element level
Abstract Syntax Tree

Source Code

```java
return getPrefix() + count;
```

AST

```
ASTParser#createAST(...)
```

- `ReturnStatement` expression
- `InfixExpression`
  - `leftOperand`
  - `rightOperand`
  - `resolveBinding`
  - `IMethodBinding`
    - `resolveBinding`
  - `MethodInvocation`
  - `SimpleName`

© Copyright 2007 by IBM Corporation, made available under the EPL v1.0
Abstract Syntax Tree cond’t

A Java type for each syntactic construct
   Assignment, CastExpression, ConditionalExpression…

Bindings for type information
   Can resolve all references through bindings

Visitors and node properties for analysis

ASTRewriter to manipulate an AST
Bindings

Bindings are fully connected

- ITypeBinding has binding of super type, interfaces, all members
- IMethodBinding has binding of parameter types, exceptions, return type
- IVariableBinding has binding of variable type

Bindings are very expensive:

- Do not hold on bindings
- Do not hold on ASTNodes that contain bindings
AST View Demo

ReturnStatement [249, 27]
  EXPRESSION
    InfixExpression [256, 19]
      > (Expression) type binding: java.lang.String
      LEFT_OPERAND
        MethodInvocation [256, 11]
          > (Expression) type binding: java.lang.String
          > method binding: A.getPrefix()
          EXPRESSION
          NAME
          ARGUMENTS (0)
          OPERATOR: '+'
      RIGHT_OPERAND
        SimpleName [270, 5]
          > (Expression) type binding: int
          > variable binding: count
          IDENTIFIER: 'count'
          EXTENDED_OPERANDS (0)
AST Rewriting

- Instead of manipulating the source code change the AST and write changes back to source

- Descriptive approach
  - describe changes without actually modifying the AST
  - allow reuse of the AST over several operations
  - support generation of a preview

- Rewriter characteristics
  - preserve user formatting and markers
  - generate an edit script
Code Manipulation Toolkits

- Refactoring – org.eclipse.ltk.refactoring
  - refactorings - org.eclipse.ltk.core.refactoring.Refactoring
    - responsible for precondition checking
    - create code changes
  - code changes - org.eclipse.ltk.core.refactoring.Change
    - provide Undo/Redo support
    - support non-textual changes (e.g. renaming a file)
    - support textual changes based on text edit support
  - user interface is dialog based

- Quick fix & Quick Assist – org.eclipse.jdt.ui.text.java
  - AST based
  - processors - org.eclipse.jdt.ui.text.java.IQuickFixProcessor
    - check availability based on problem identifier
    - generate a list of fixes
  - user interface is provided by editor
Summary

- JDT delivers powerful program manipulation services
  - Java Model, Search engine and DOM AST
  - Add your own tool to the Eclipse Java IDE
  - but also in headless mode (can be used programmatically)
    - Visual Editor, EMF, metric tools, …
  - Full J2SE 5.0/6.0 support

- Community feedback is essential
  - bug reports: http://bugs.eclipse.org/bugs
  - mailing lists: http://www.eclipse.org/mail/index.html
  - newsgroups: news://news.eclipse.org/eclipse.tools.jdt
Legal Notice

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

- Other company, product, or service names may be trademarks or service marks of others.
Type Hierarchy

Create – on a type or on a region (= set of Java Elements)

```java
    typeHierarchy = type.newTypeHierarchy(progressMonitor);
    typeHierarchy = project.newTypeHierarchy(region, progressMonitor);
```

Supertype hierarchy – faster!

```java
    typeHierarchy = type.newSupertypeHierarchy(progressMonitor);
```

Get super and subtypes, interfaces and classes

```java
    typeHierarchy.getSubtypes(type)
```

Change listener – when changed, refresh is required

```java
    typeHierarchy.addTypeHierarchyChangedListener(..);
    typeHierarchy.refresh(progressMonitor);
```
Resolving the reference to “String” in a compilation unit

```java
String content = "public class X {
  String field;
}"

ICompilationUnit cu = fragment.createCompilationUnit("X.java", content, false, null);

int start = content.indexOf("String");
int length = "String".length();
IJavaElement[] declarations = cu.codeSelect(start, length);
```

Contains a single IType: `java.lang.String`
Search Engine – an Example

Searching for all declarations of methods “foo” that return an int

```java
SearchPattern pattern = SearchPattern.createPattern("foo(*) int",
IJavaSearchConstants.METHOD,
IJavaSearchConstants.DECLARATIONS,
SearchPattern.R_PATTERN_MATCH);

IJavaSearchScope scope = SearchEngine.createWorkspaceScope();

SearchRequestor requestor = new SearchRequestor() {
    public void acceptSearchMatch(SearchMatch match) {
        System.out.println(match.getElement());
    }
};

SearchEngine searchEngine = new SearchEngine();
searchEngine.search(
    pattern,
    new SearchParticipant[] { SearchEngine.getDefaultSearchParticipant() },
    scope,
    requestor,
    null /*progress monitor*/);
```
Creating an AST

```java
ASTParser parser = ASTParser.newParser(AST.JLS3);
parsersetSource(cu);
parsersetResolveBindings(true);
parsersetStatementsRecovery(true);
ASTNode node = parser.createAST(null);
```

Create AST on an element

```java
ASTParser parser = ASTParser.newParser(AST.JLS3);
parsersetSource("System.out.println();".toCharArray());
parsersetProject(javaProject);
parsersetKind(ASTParser.K_STATEMENTS);
parsersetStatementsRecovery(false);
ASTNode node = parser.createAST(null);
```

Create AST on source string
AST Visitor

```java
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(cu);
parser.setResolveBindings(true);

ASTNode root = parser.createAST(null);
root.accept(new ASTVisitor() {
  public boolean visit(CastExpression node) {
    fCastCount++;
    return true;
  }
  public boolean visit(SimpleName node) {
    IBinding binding = node.resolveBinding();
    if (binding instanceof IVariableBinding) {
      IVariableBinding varBinding = (IVariableBinding) binding;
      ITypeBinding declaringType = varBinding.getDeclaringClass();
      if (varBinding.isField() &&
          "java.lang.System".equals(declaringType.getQualifiedName())) {
        fAccessesToSystemFields++;
      }
    }
    return true;
  }
```

Count the number of casts

Count the number of references to a field of `java.lang.System` (`System.out`, `System.err`)
Example of the descriptive AST rewrite:

```java
public void modify(MethodDeclaration decl) {
    AST ast = decl.getAST();
    ASTRewrite astRewrite = ASTRewrite.create(ast);

    SimpleName newName = ast.newSimpleName("newName");
    astRewrite.set(decl, MethodDeclaration.NAME_PROPERTY, newName, null);

    ListRewrite paramRewrite =
        astRewrite.getListRewrite(decl, MethodDeclaration.PARAMETERS_PROPERTY);
    SingleVariableDeclaration newParam = ast.newSingleVariableDeclaration();
    newParam.setType(ast.newPrimitiveType(PrimitiveType.INT));
    newParam.setName(ast.newSimpleName("p1"));
    paramRewrite.insertFirst(newParam, null);

   .TextEdit edit = astRewrite.rewriteAST(document, null);
    edit.apply(document);
}
```