PLDI 2005, Chicago, IL USA

Static Analysis for Java in Eclipse

The Dirty Little Secrets

Dr. Robert M. Fuhrer

Research Staff Member

Program Analysis & Transformation Group

IBM Watson Research Center

rfuhrer@watson.ibm.com

© Robert M. Fuhrer 2005

Tutorial Motivation

- Eclipse (http://www.eclipse.org): one of the most popular Java (IDE's)
  - open-source distribution
  - portability
  - powerful plug-in extension mechanism
    - extend environment with tools, views, and analyses to suit specific needs
  - advanced feature set
  - mature Java development tool chain
    - stable API's for representing and manipulating Java programs
    - support for the latest language features in Java 5.0
- An ideal platform for hosting commercial & experimental Java analyses and tools!
- But: size & complexity of API's make for daunting learning curve!
  - So far, few researchers implement their analyses in Eclipse
- Tutorial Goal: help bridge the gap between potential and reality
  - give participants insight into important aspects of developing static analyses within the Eclipse IDE and exposing their results to users.

Tutorial Prerequisites

- **Who:**
  - researchers and practitioners interested in implementing static analyses and tools for Java in the setting of a realistic IDE
- **Prerequisites:**
  - working knowledge of Eclipse as development environment for ordinary Java applications
  - knowledge of Java language syntax and semantics
  - basic knowledge of fundamental static analysis techniques

Tutorial Overview

- Part I: Eclipse 3.1 Overview (1 hour)

  <break>

- Part II: Intraprocedural Flow Analysis (1.25 hours)

  <break>

- Part III: Interprocedural Type Analysis (1.25 hours)

Part I: Eclipse 3.1 Overview (1 hour)

- Purpose
  - flow of plug-in development
  - hook into the user interface to trigger analyses and present analysis results
  - describe Eclipse Java Development Toolkit (JDT) API's

Eclipse 3.1 Overview: Topics

- Plug-in architecture
  - plug-ins, extension points and extensions
  - creating plug-in projects
  - managing plug-in dependencies
- Resources
  - builders & markers
- Contributing user interface actions
  - e.g. view-specific context menu items
- Java API's (“JDT/Core” & “JDT/UI”)
  - high-level Java model
  - abstract syntax trees (AST's)
    - parsing, traversing, rewriting
  - type representations & type hierarchies
  - searching

Plug-in Architecture

- All functionality provided by some plug-in
  - Core resource API's
    - representing artifacts (folders, source files, class files, projects)
  - UI componentry (SWT/JFace/Workbench)
    - tables, tree widgets, text, menu items, dialogs
  - Views
    - Java Editor, Package Explorer, Problems View
  - Java Compiler, Java Debugger
  - N.B.: Even non-Eclipse functionality must be encapsulated in a plug-in!
    - ANT, xalan
- Plug-ins are lazily instantiated to reduce memory footprint and speed launching

Plug-in Architecture

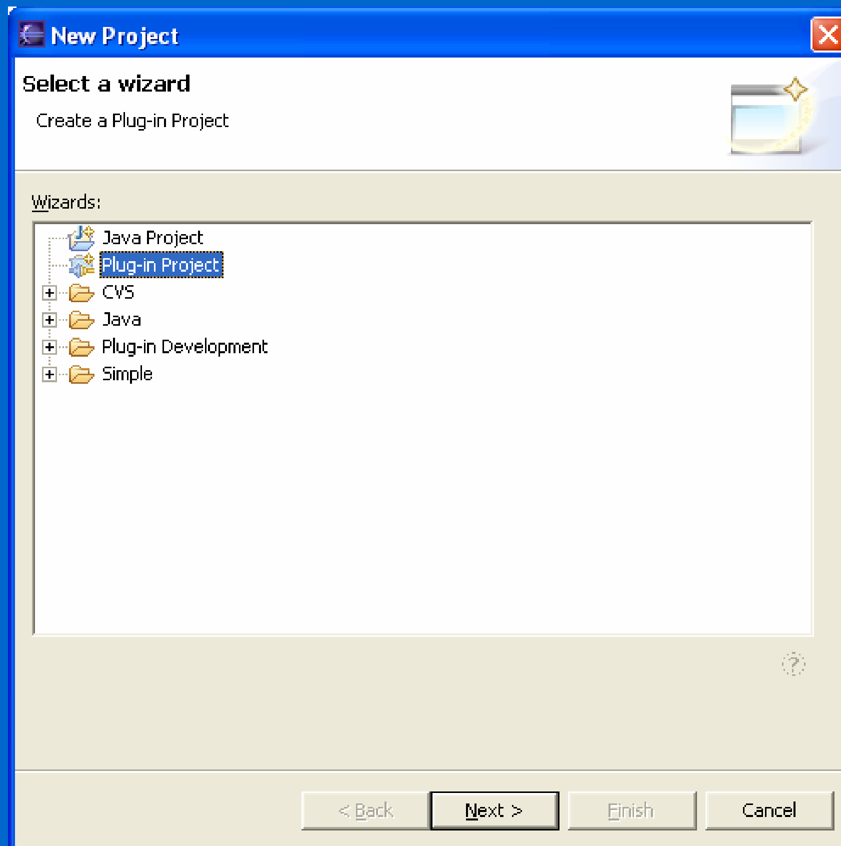
- Plug-in consists of:
  - ID (e.g. `org.eclipse.jdt.ui`)
  - Name (human-readable, e.g. “JDT/Core”)
  - Version
  - Plug-in class (plug-in initialization/teardown)
  - 0 or more dependencies on other plug-ins
  - 0 or more extensions
    - e.g.: menu items, views, builders
  - 0 or more extension points
    - defines sites for other plug-ins to add functionality

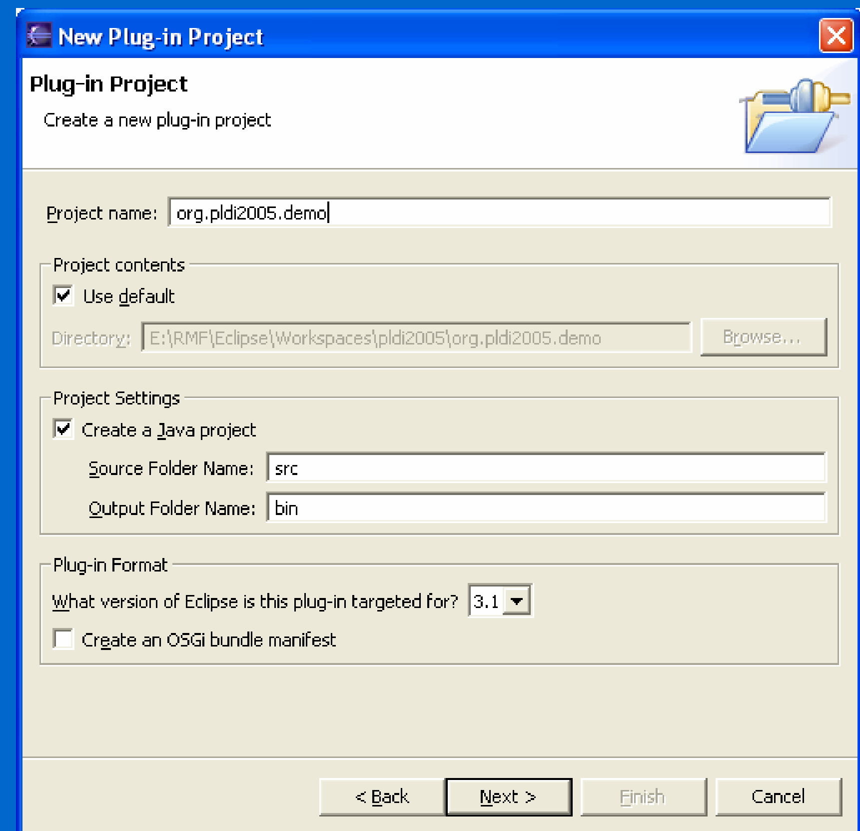
Plug-in Architecture: plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="com.ibm.watson.pldi2005"
   name="PLDI 2005 Demo"
   version="1.0.0"
   provider-name="rfuhrer@watson.ibm.com"
   class="com.ibm.watson.pldi2005.PLDI2005Plugin">

   <runtime>
      <library name="pldi2005.jar"/>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.jdt.core"/>
   </requires>

</plugin>
```

specify dependencies

Creating a Plug-in Project, 1/3





Creating a Plug-in Project, 2/3

New Plug-in Project

Plug-in Content

Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID: org.pldi2005.demo

Plug-in Version: 1.0.0

Plug-in Name: Demo Plug-in

Plug-in Provider: rfuhrer@watson.ibm.com

Runtime Library: demo.jar

Plug-in Class

☑ Generate the Java class that controls the plug-in's life cycle (recommended)

Class Name: org.pldi2005.demo.DemoPlugin

☑ This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? ○ Yes ● No

< Back | Next > | Finish | Cancel

New Plug-in Project

Templates

Select one of the available templates to generate a fully-functioning plug-in.

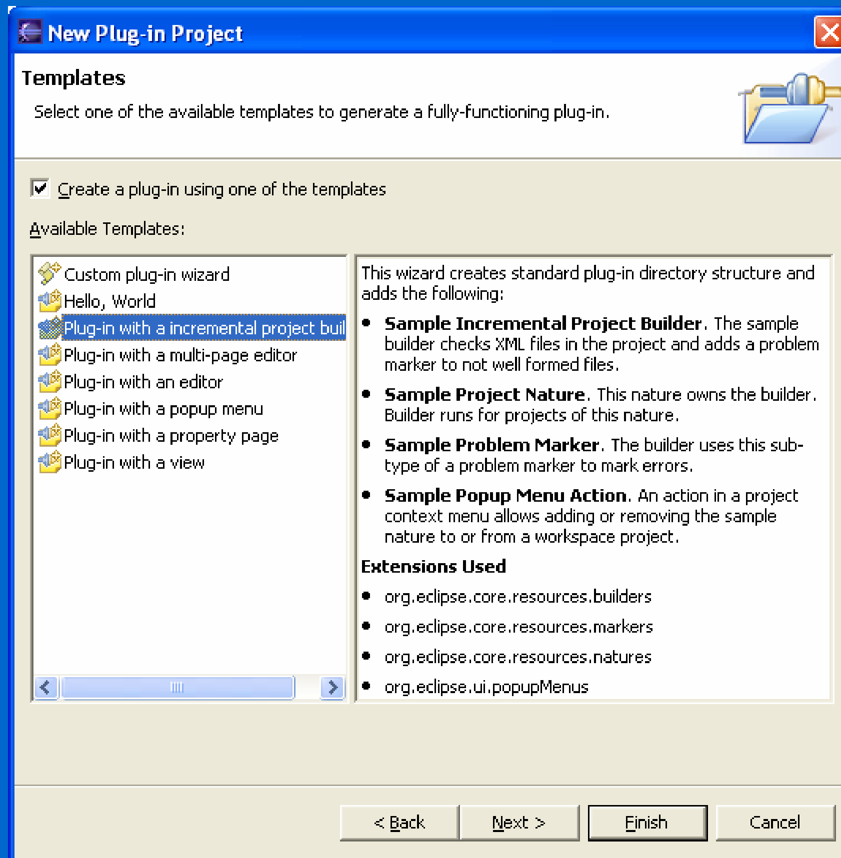
☑ Create a plug-in using one of the templates

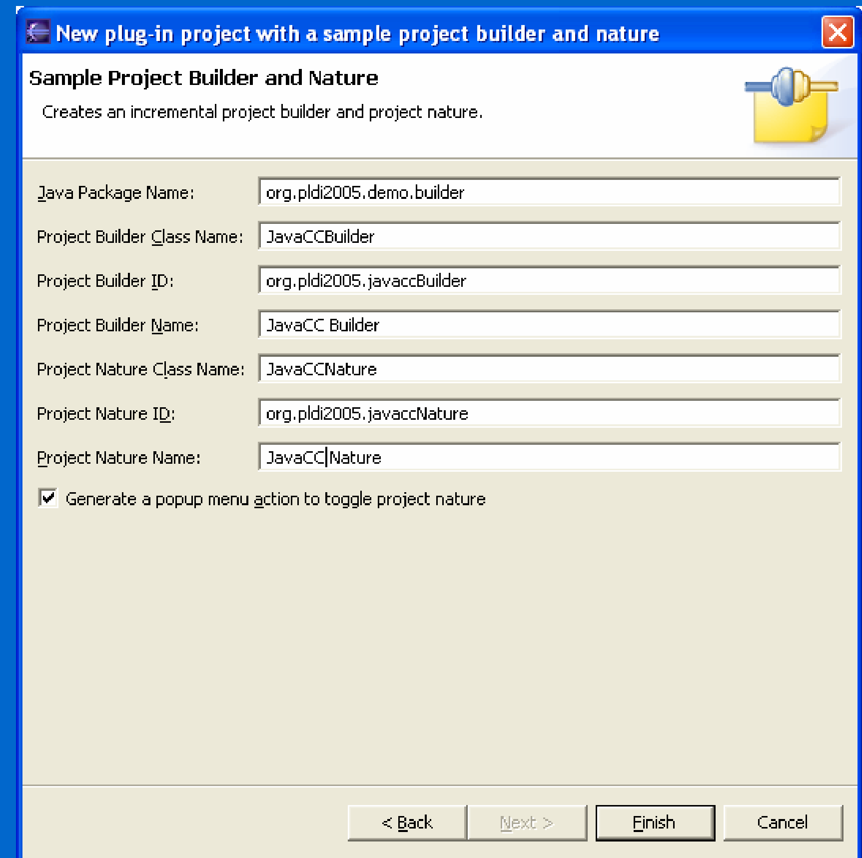
Available Templates:

- Custom plug-in wizard
- Hello, World
- Plug-in with a incremental project builder
- Plug-in with a multi-page editor
- Plug-in with an editor
- Plug-in with a popup menu
- Plug-in with a property page
- Plug-in with a view

< Back | Next > | Finish | Cancel

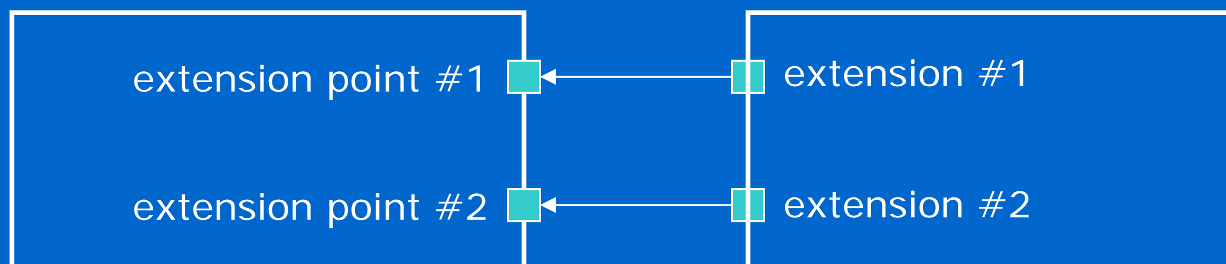
Creating a Plug-in Project, 3/3





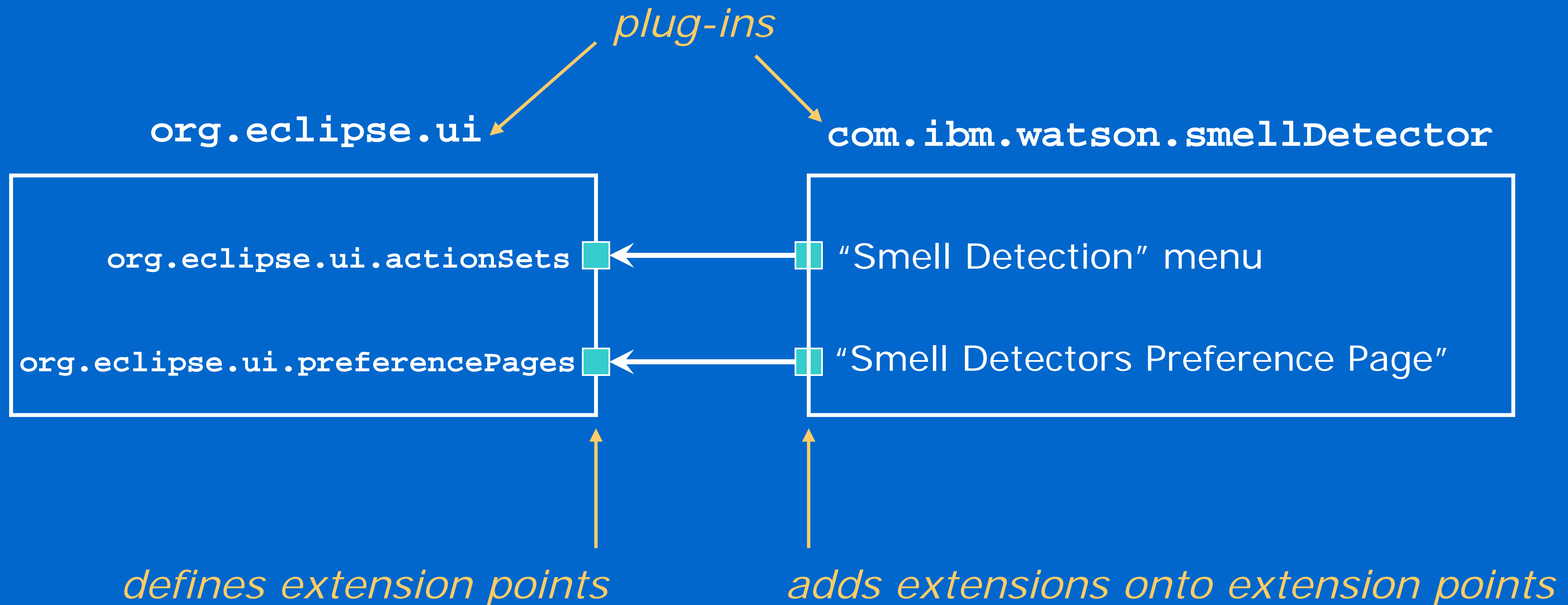
Plug-in Architecture: Extensions

- extension point – "a socket to plug extensions into"
  - ID (e.g. **`org.eclipse.ui.actionSets`**)
  - *<extension-specific structure>*
  - e.g. **`<view name="…" icon="…" category="…" class="…"/>`**
- extension – "added functionality to plug into an extension pt"
  - ID of extension point being extended
  - *<structure consistent with extension point defn>*
- Most extension points are hooks for Java-implemented functionality
  - *no code need be involved (pure metadata): key bindings, help*



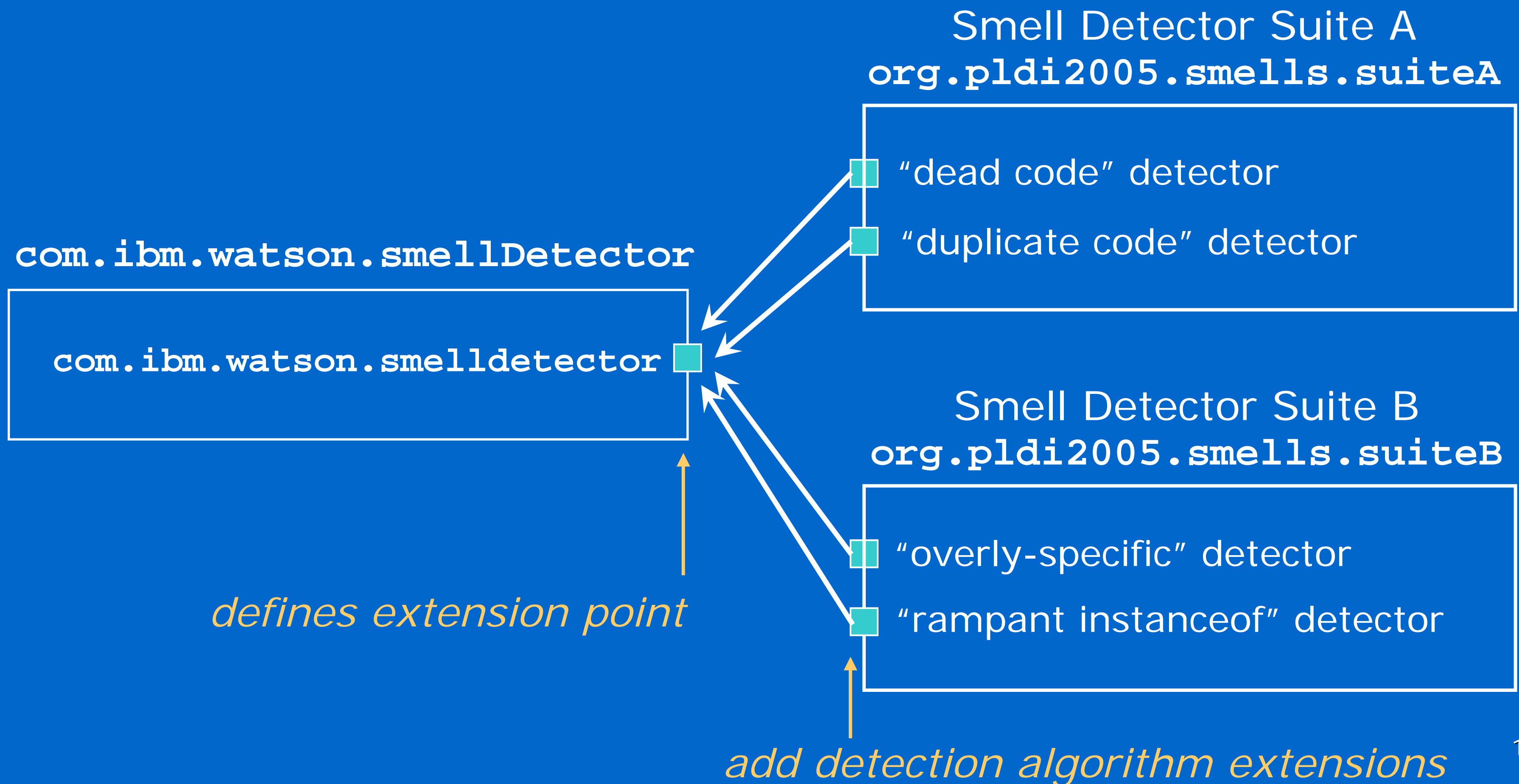
Plug-in Architecture: Extensions

- Example 1: UI Extensions



Plug-in Architecture: Extensions

- Example 2: non-UI extensions



Plug-in Architecture: Extension Point Schemas

- XML Schema defines structure of an extension point
- Use the Extension Point Schema Editor!
- Constraints on type of implementation class for an extension are either:
  - EXPLICITLY defined in extension point schema (but NOT CHECKED statically!)
  - IMPLICITLY defined by casts in extension point implementation (NOT CHECKED statically!)

Sample Extension Point

General Information

This section describes general information about this schema.

Plug-in ID: com.ibm.watson.pldi2005

Point ID: sample

Point Name: Sample Extension Point

Extension Point Elements

The following XML elements and attributes are allowed in this extension point:

- extension
  - point
  - id
  - name
- tool
  - id
  - class

New Element

New Attribute

Element Grammar

View or modify the content model of the selected element

DTD approximation:

Description

Add short description of elements and attributes for documentation purposes. Use HTML tags where appropriate.

[Enter description of this extension point.]

Apply

Reset

Definition | Documentation | sample.exsd

Plug-in Architecture: Extension Instantiation

When your plug-in needs to instantiate the extensions that hook into one of its extension points:

```
class MyPlugin extends AbstractUIPlugin {
   // The following two strings must match what’s in plugin.xml!!!
   static final String pluginID = “org.pldi2005.demo”;
   static final String pointID  = “org.pldi2005.demo.tool”;

   private void createExtensions() {
     IExtensionRegistry er = Platform.getExtensionRegistry();
     IExtensionPoint    ep = er.getExtensionPoint(pluginID, pointID);
     IExtension[]       exts = ep.getExtensions();

     for(int i=0; i < exts.length; i++) {
        IConfigurationElement[] ces =
                                   exts[i].getConfigurationElements();
        for(int j=0; j < ces.length; j++) {
           MyExtensionIntf ext = (MyExtensionIntf)
                    ces[i].createExecutableExtension(“elementID”);
           ext.doStuff();
        }
     }
   }
}
```

metadata ops

instantiate extension implementation class

Plug-in Architecture: Plug-in State

- private state used by your plug-in
- persists across workbench invocations
- distinct from preferences store
- stored in user's workspace at
  `.metadata/.plugins/<your-plug-in-dir>`

```
private File getStateFile() {
  MyPlugin plugin = MyPlugin.getInstance();
  IPath path = plugin.getStateLocation();

  path = path.append("state.dat"); // or whatever you want
  return path.toFile();
}
```

UI Contributions: Menu Actions

- Main menu contributions

augmenting an existing menu: repeat the menu definition

```
<extension point="org.eclipse.ui.actionSets">
  <actionSet label="Watson Refactorings"
             description="Watson Refactorings"
             visible="true"
             id="com.ibm.watson.refactoring.actionSet">
    <menu label="Refactor"
          path="edit"
          id="org.eclipse.jdt.ui.refactoring.menu">
          <separator name="watsonGroup"/>
    </menu>
    <action
      label="Infer Type Arguments"
      class="com.ibm.watson.refactoring.actions.InferTypeArgsAction"
      menubarPath="org.eclipse.jdt.ui.refactoring.menu/watsonGroup"
      id="com.ibm.watson.refactoring.inferTypeArguments">
    </action>
  </actionSet>
</extension>
```

path = id + group

implements org.eclipse.ui.IWorkbenchWindowActionDelegate

UI Contributions: Pop-up Menu Actions

- Viewer context-menu contributions
  - E.g., Task List context menu contribution:

```
<extension point="org.eclipse.ui.popupMenus">
   <viewerContribution id="com.xyz.C2"
                 targetID="org.eclipse.ui.views.TaskList">
      <action
          id="com.xyz.showXYZ"
          label="&amp;Show XYZ"
          style="toggle"
          state="true"
          menubarPath="additions"
          icon="icons/showXYZ.gif"
          class="com.xyz.actions.XYZShowActionDelegate">
      </action>
   </viewerContribution>
</extension>
```

to this view…

… add this action

UI Contributions: Pop-up Menu Actions

- "Object contributions" – common actions appearing in pop-up menus for selected entities of a given type
- Appear in, e.g., Navigator and Outline views

```
<extension point="org.eclipse.ui.popupMenus">
  <objectContribution objectClass="org.eclipse.jdt.core.IMember"
                      id="watson">
     <menu label="Refactor"
           path="additions"
           id="org.eclipse.jdt.ui.refactoring.menu">
        <separator name="undoRedoGroup"/>
        <separator name="reorgGroup"/>
        <separator name="typeGroup"/>
        <separator name="codingGroup"/>
     </menu>
     <action
        label="Infer Type Arguments"
        class="com.ibm.watson.refactoring.actions.InferTypeArgsAction"
        menubarPath="org.eclipse.jdt.ui.refactoring.menu/typeGroup"
        id="com.ibm.watson.refactoring.inferTypeArguments">
     </action>
  </objectContribution>
</extension>
```

for objects of this type...

... provide this action

UI Contributions: plugin.xml Editor

"Plugin Development Environment" (PDE) plugin editor permits easier editing of plug-in descriptor than raw XML

Overview

General Information

This section describes general information about this plug-in:

ID:	com.ibm.watson.pldi2005
Version:	1.0.0
Name:	PLDI 2005 Demo
Provider:	rfuhrer@watson.ibm.com
Class:	com.ibm.watson.pldi2005.PLDI2005Plugin [Browse...]

Plug-in Content

The content of the plug-in is made up of four sections:

- Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
- ① Runtime: lists the libraries that make up this plug-in's runtime.
- Extensions: declares contributions this plug-in makes to the platform.
- Extension Points: declares new function points this plug-in adds to the platform.

For this plug-in to take advantage of additional runtime capabilities, you need to create an OSGi bundle manifest.

or

Testing

You can test the plug-in by launching a second (runtime) instance of an Eclipse application:

- Launch an Eclipse application
- Launch an Eclipse application in Debug mode

Exporting

To package and export the plug-in:

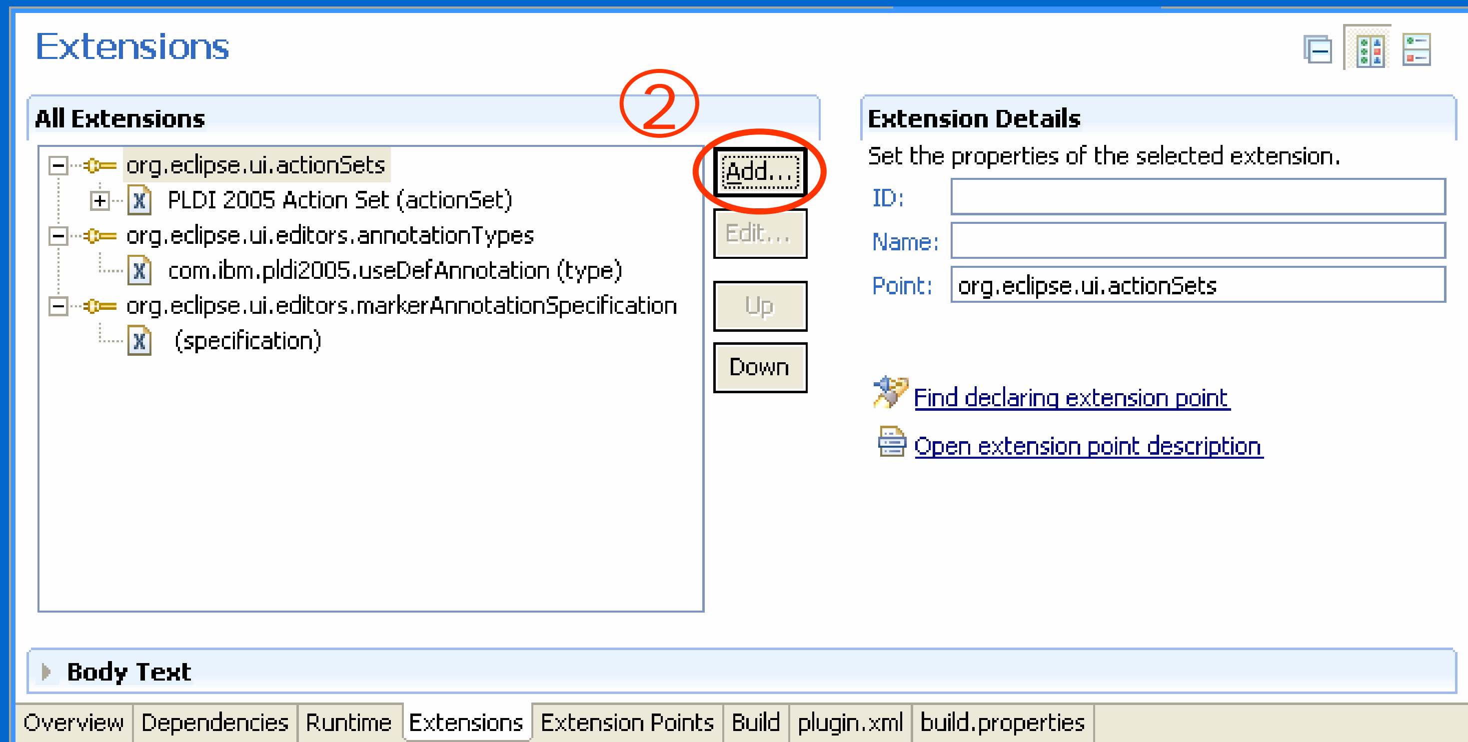
1. Specify what needs to be packaged in the deployable plug-in on the Build Configuration page
2. Export the plug-in in a format suitable for deployment using the Export Wizard

①

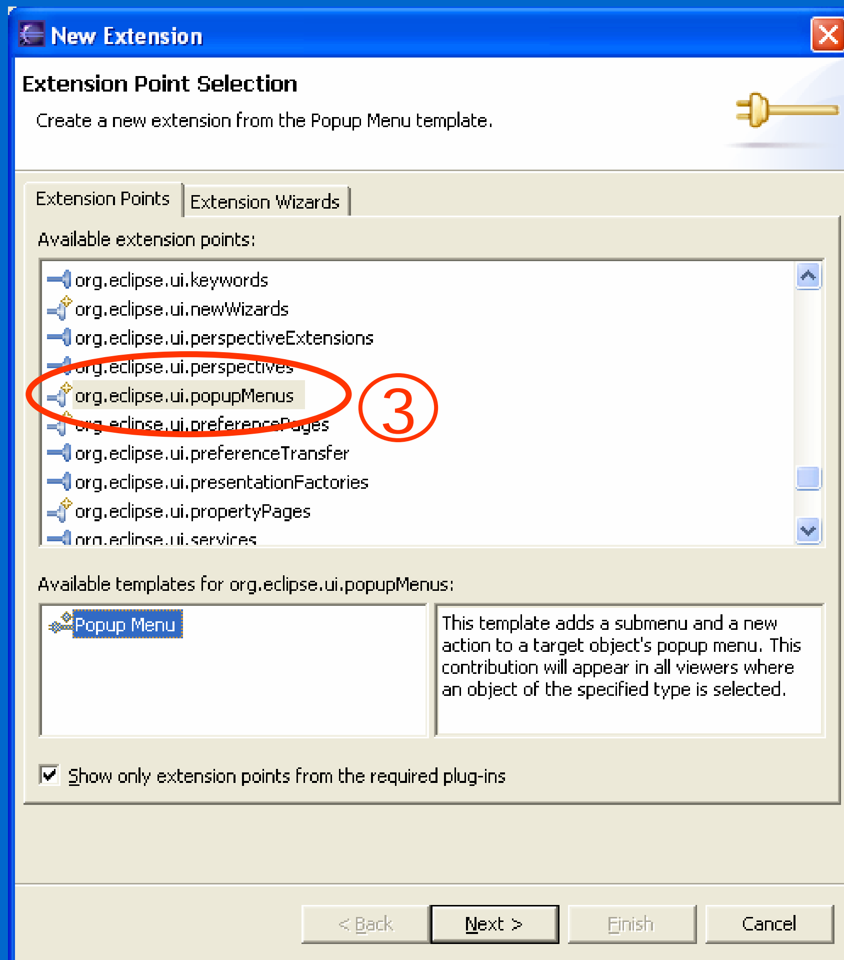
Overview | Dependencies | Runtime | Extensions | Extension Points | Build | plugin.xml | build.properties

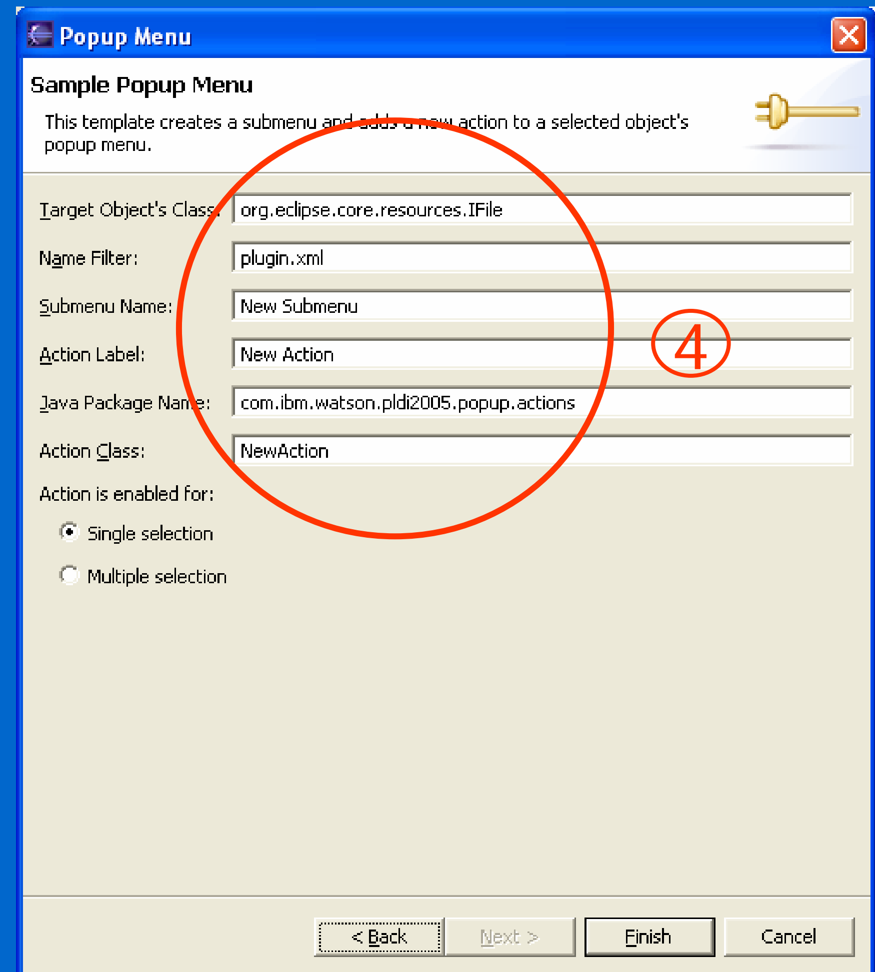
UI Contributions: Extension Wizards

"Plugin Development Environment" (PDE) provides wizards to browse/edit/add extensions from within plugin.xml editor:



UI Contributions: Extension Wizards





Extension Points: More Info

- Documentation on Eclipse-provided extension points available from within a running workbench:
  - Help -> Help Contents -> Platform Plug-in Developer Guide -> Reference ->
    - Extension Points Reference
    - API Reference

Eclipse API's: Resources

`org.eclipse.core.resources`

- aka "The Workspace"
- Completely language- (Java-) agnostic
- Workspace = tree of resources
  - Folders (`bin/`, `src/`, `src/org/eclipse/…`)
  - Files (`Foo.java`, `Foo.class`, `foo.properties`, `rt.jar`)
- Likewise: resource = item in Workspace tree
  - i.e., to first order: if it's not in the workspace, the resource API doesn't know about it

- N.B.: Package Explorer mostly shows Java resources; Navigator shows everything

Resources API

- `IPath` – encapsulation of file system location
  - used to identify locations of:
    - resources in the workspace
    - entities in a class path
  - relative and absolute
- `IResource` – encapsulation of a possibly non-existent file/folder
  - `getName()`, `getParent()`, `getModificationStamp()`
  - various flavors of `copy()`, `delete()`, `move()`
  - `IPath getLocation()`, `IPath getFullPath()`
  - `refreshLocal()` – pick up file system changes (optionally, recursively)
  - `createMarker()`, `deleteMarkers()`, `findMarkers()`
  - `exists()` ← N.B. resource need not exist in filesystem!
- Typical non-Java resource creation sequence:

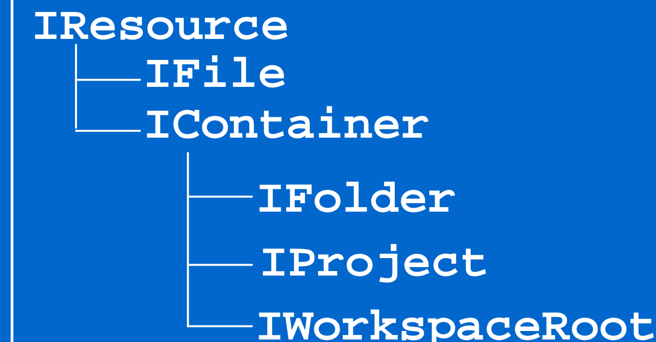
```
myProject.getFolder(folderName).create(…)
```

- Note: *Can* use Java IO to manipulate file system and let refreshLocal() pick up resource changes, but **NOT** recommended! (slow and error-prone) Use IResource API's!

Resources API: Hierarchy

- **`IFile`** – stream of bytes
  - `create(InputStream, …)`
    - exception if already exists
  - `String getFileExtension()`
  - `InputStream getContents()`
  - `setContents(InputStream)`
  - `IFileState[] getHistory()`
  - `setContent(IFileState)`

interface hierarchy:



- **`IContainer`** – something that can have children
  - `members(), findMember()`
  - `IFile getFile(IPath to)` careful: need not exist!
  - `IFolder getFolder(IPath to)` careful: need not exist!
- **`IFolder`**
  - `create(…)` – exception if already exists
  - `IFile getFile(String nm)` careful: need not exist!
  - `IFolder getFolder(String nm)` careful: need not exist!

Resources API: Hierarchy

containment hierarchy:

```
IWorkspaceRoot
  └──IProject "MyProject"
        ├──IFile ".project"
        └──IFolder "src"
              └──IFile "foo.java"
```

- `IWorkspaceRoot` – top-level folder of workspace
  - `IProject[] getProjects()`
  - `IProject getProject(String name)` careful: need not exist!
- `IProject` – root of a project's contents
  - `create()` – exception if already exists
  - `IProjectDescription getDescription()` – access to natures, build commands, project dependencies, etc.
  - `build(int kind)` – initiate rebuild (shouldn't need to do this yourself)

Resources API: Hierarchy Traversal

```
interface IResourceVisitor {
   public boolean visit(IResource resource);
}
```

- Usage:

```
class MyVisitor implements IResourceVisitor {
   public boolean visit(IResource res) {
      // your code here, e.g.:
      if (res instanceof IFile)
         System.out.println("File: " + res.getFullPath());
      return true; // visit child resources
   }
}

IWorkspace     ws = ResourcesPlugin.getWorkspace();
IWorkspaceRoot root = ws.getRoot();
IProject       project = root.getProjects()[0];

project.accept(new MyVisitor());
```

Builders

- E.g. Eclipse Java compiler, JavaCC encapsulation
- Incremental (most common) or full build
  - Invoked in response to resource changes
  - Receive “resource deltas” describing what changed
- Generally:
  - create resources (e.g. class-files), possibly triggering more building
    - infrastructure also useful for analyzers that don’t generate any resources (e.g. smell detectors)
  - associate “problem markers” with resources to indicate build errors/warnings
    - appear in various views, e.g., Problems View
- May be “chained”
  - e.g. JavaCC generates Java code that needs compilation
- Key API to implement:

```
abstract class IncrementalProjectBuilder {
   abstract IProject[]  build(int kind, Map args,
                              IProgressMonitor pm);
   final IResourceDelta getDelta(IProject p) { … }
   // …
}
```

Builders: Builder Implementation

```
class JavaCCBuilder extends IncrementalProjectBuilder {
    IProject[] build(int buildKind, Map args, IProgressMonitor monitor) throws CoreException {
       IProject project = getProject();

       monitor.beginTask("Scanning for and compiling JavaCC source files...", 0);

       if (buildKind == FULL_BUILD) {
          project.accept(new IResourceVisitor() {
             boolean visit(IResource res) {
                String ext = res.getFileExtension();

                if (res.exists() && res instanceof IFile && ext != null && ext.equals("jj")) {
                   IFile file = (IFile) res;
                   clearMarkersOn(file);   // clear markers left by previous invocation
                   invokeJavaCC(file);     // call out to external tool
                   forceFileUpdates(file); // call refreshLocal() on generated files
                   return false; // skip sub-resources
                } else
                   return true; // need to descend into sub-resources
             });
       } else { // incremental or auto build
          getDelta(project).accept(new IResourceDeltaVisitor() {
             public boolean visit(IResourceDelta delta) throws CoreException {
                if (delta.getKind() == IResourceDelta.ADDED/CHANGED &&
                    delta.getResource() is a .jj file)
                   <run JavaCC> // (see above)
                return false;
             }
          });
       }
       monitor.done();
       return new IProject[] { project };
    }
}
```

Builders: Natures

- `IProjectNature` – encapsulates the augmentation of projects with behavior
  - Primarily used to associate builders with projects
    - E.g. "Java Nature" enables Eclipse Java builder (compiler) to run on a project
  - Natures often associated with projects when project created
    - Alternatively: by specific user-triggered action (e.g. "Enable Foo" context menu pick on project)

Builders: Nature Configuration

must match builder spec in plugin.xml

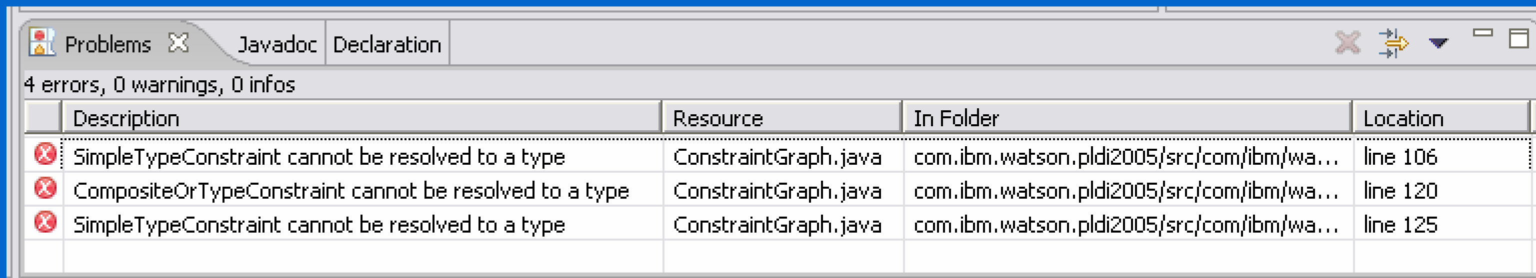
```
class MyNature implements IProjectNature {
    static final String myBuilderID = "org.pldi2005.builder";
    static final String myNatureID = "org.pldi2005.myNature";

    IProject fProject;

    void setProject(IProject p) { fProject = p; }

    void configure() {
        IProjectDescription pd = fProject.getDescription();
        ICommand[] cmds = pd.getBuildSpec();

        if (no entry in cmds has ID myBuilderID) {
            ICommand myCmd = pd.newCommand();

            myCmd.setBuilderName(myBuilderID);

            <insert myCmd in cmds>

            // N.B.: element order determines run order
            // (use care if myBuilder generates Java source)
            pd.setBuildSpec(cmds);
            fProject.setDescription(pd, null);
        }
    }
}
```

Builders: Adding a Project Nature

```
class MyAction implements IWorkbenchWindowActionDelegate {
   void run(IAction action) {
      IJavaProject project = /* get currently selected project */;

      addMyNature(project);
   }

   void addMyNature(IJavaProject javaProject) {
     IProject project = javaProject.getProject();
     IProjectDescription description = project.getDescription();
     String[] natures = description.getNatureIds();
     String[] newNatures = new String[natures.length + 1];

     // Insert new nature ID into the nature array…
     System.arraycopy(natures, 0, newNatures, 0, natures.length);
     newNatures[natures.length] = MyNature.myNatureID;

     description.setNatureIds(newNatures);
     project.setDescription(description, null);
   }
}
```

Resources API: Markers

- **`IMarker`** – associate metadata w/ resources, e.g.
  - "problem markers" – compile errors
  - "task markers" – user-specified "tasks" like TODO's
  - "book marks" – user-specified sites in text
- Marker consists of:
  - **`IResource`**
  - Time stamp
  - ID (unique relative to a given **`IResource`**)
  - "Type" (a string tag, e.g. **`"org.eclipse.core.resources.marker"`**)
  - Additional attributes (key/value pairs)



Resources API: Markers

- Marker types are defined as extensions in plugin.xml:
  - Unique ID (e.g. **`"com.ibm.watson.smellmarker"`**)
  - List of marker "super-type" ID's
  - List of standard attribute keys
  - Persistent flag
    - Unfortunately: even persistent markers not stored in CVS ☹
- Create instances using **`IResource.createMarker()`**:

```
IFile file= …; // or any kind of IResource
IMarker m= file.createMarker(markerTypeID);

m.setAttribute(IMarker.SEVERITY, IMarker.SEVERITY_ERROR);
m.setAttribute(IMarker.LINE_NUMBER, 25);
m.setAttribute(IMarker.CHAR_START, 65); // char offsets relative…
m.setAttribute(IMarker.CHAR_END, 72);   // …to beginning of file
```

Resources API: Marker Resolution

- Aka "quick fixes"
  - associate code to run to resolve issue indicated by marker
- `IMarkerResolution` – how to resolve
  - `String getLabel()` – UI presentation
  - `run(IMarker)` – do it
- `IMarkerResolutionGenerator`
  - `IMarkerResolution[] getResolutions(IMarker)`
- If marker needs more info than line/col # to identify what to operate on and how => add your own attributes
- *<example in Part III>*

Resources API: Resource Changes

- `IWorkspace` – maintains set of state listeners
  - `addResourceChangeListener(IResourceChangeListener)`
- `IResourceChangeListener` –implement to be notified of changes
  - `resourceChanged(IResourceChangeEvent)`
- `IResourceChangeEvent`
  - `IResourceDelta getDelta()` – what changed
  - `IResource getResource()` – root resource bounding change
- `IResourceDelta` – hierarchical description of changes to workspace
  - `getKind() => ADDED, REMOVED, CHANGED`
  - `IResourceDelta[] getAffectedChildren()`
  - `accept(IResourceDeltaVisitor)`
- *See Java Developer's Guide to Eclipse, p. 587+ for details on*
  - *timing of resource events relative to building, editing, etc.*
  - *traversal of resource deltas*

Progress Monitors

- Provide user with progress feedback during long-running operations (e.g. building, CVS operations, refactoring)
- `IProgressMonitor`
  - Usually passed to you from above, but can instantiate:
    - `NullProgressMonitor` (no-op implementation)
    - `SubProgressMonitor` (for nested tasks)
  - or access the progress monitor from the `StatusLine` (see Eclipse 3.0 FAQ)

Eclipse API's: Java Structures

- 3 domains of Java entities:
  - **`IJavaElement`** hierarchy
    - "Summary" information
    - Includes method signatures, but not bodies
    - Instances not canonicalized; use **`equals()`**
  - AST's (**`ASTNode`** hierarchy)
    - Fully detailed, with complete method bodies
    - **`AST`** node factory and **`ASTRewrite`** for manipulation
  - **`IBinding`** hierarchy
    - Resolved references to types, members, variables
    - Instances not canonicalized; use **`Bindings.equals()`**

Java Structures: `IJavaElement`

- **AKA "Java Model"**
- `org.eclipse.jdt.core.IJavaElement` and sub-types
- "Handle-based" representations of:
  - Projects, packages, compilation units
  - Types & members
- Lazily populated and cached
  - Even trivial queries like `IType.isInterface()` may require re-parsing source!
  - Many queries throw `JavaModelException` (e.g., when invoking operations on non-existent entities)
- Enough information for UI tasks; also returned by `SearchEngine`
  - BUT: No access to method bodies, field initializers, ... (need AST's for those)
- Careful: can accidentally create non-existent elements; problem manifests as `JavaModelException` on subsequent operations
  - `IType.getField(String name)`
  - `IType.getMethod(String name, String[] types)`

Java Structures: `IJavaElements`

- `IJavaElement` – base type for all Java entities
  - May appear:
    - in "structured selections" in Java views (e.g. Pkg Explorer, Outline)
    - in results from SearchEngine queries
    - in results from ITypeHierarchy queries
  - `String getElementName()`
  - `int getElementKind()` – `PROJECT, FRAGMENT, UNIT, TYPE,…`
  - `IJavaElement getParent()`
  - `IJavaProject getJavaProject()`
  - `IResource getCorrespondingResource()` - possibly `null`, e.g., for `IMembers`
  - `IPath getPath()`
  - `boolean isStructureKnown()` – "Can you get an AST for this?"
  - `boolean exists()`
- `IParent` – extended by almost all other `IJavaElement` interfaces
  - `boolean hasChildren()`
  - `IJavaElement[] getChildren()`

Java Structures: `IJavaElements`

- `IJavaModel` – "Java root" of workspace
  - `IJavaProject getJavaProject(String name)`
  - `IJavaProject[] getJavaProjects()`
- `IJavaProject`
  - `IPackageFragmentRoot[] getPackageFragmentRoots()`
  - `IPackageFragment[] getPackageFragments()`
  - `IClasspathEntry[] getRawClasspath()` – unexpanded variables
  - `IClasspathEntry[] getResolvedClasspath()` – fully expanded
  - `Map getOptions()`
  - `IType findType(String qualifiedName)`
- `IPackageFragmentRoot` – project source folders, jars, zip files
  - `getKind()` => `K_BINARY` or `K_SOURCE`
  - `IPackageFragment getPackageFragment(String name)`

Java Structures: `IJavaElements`

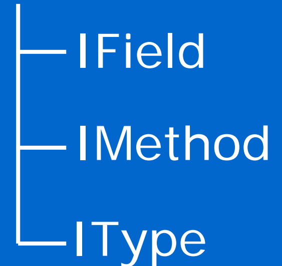
- `IPackageFragment` – packages and sub-packages within a given `IPackageFragmentRoot`
  - `ICompilationUnit[] getCompilationUnits()`
  - `IClassFile getClassFiles()`
  - `boolean hasSubpackages()`
  - `ICompilationUnit createCompilationUnit(String nm, String source)`
- `ICompilationUnit` – a single Java source file's contents
  - Has single top-level visible type whose name matches that of the source file (the "primary type" of the CU)
  - 0 or more top-level non-public types
  - `IType findPrimaryType()`
  - `IType[] getAllTypes()` – all top-level and nested types
  - `IImportDeclaration getImports()`
  - `IPackageDeclaration[] getPackageDeclarations()`
  - `IType createType(…)`
  - `IType getType()` – careful: may not exist!

Java Structures: Classpath Scanning

```
void processClassPath(IJavaProject project) throws JavaModelException, IOException {
    IClasspathEntry[] classPathEntries = project.getResolvedClasspath(true);
    IWorkspaceRoot    wsRoot = ResourcesPlugin.getWorkspace().getRoot();

    for(IClasspathEntry cpe: classPathEntries) {
       IPath entryPath = cpe.getPath();
       switch(cpe.getEntryKind()) {
          case IClasspathEntry.CPE_LIBRARY: {
             File file = entryPath.makeAbsolute().toFile();

             if (!file.isFile())
                file = wsRoot.getLocation().append(entryPath).toFile();
             if (file.isFile())
                processFile(realFile);
             break;
          }
          case IClasspathEntry.CPE_PROJECT: {
             File outputDir = cpe.getOutputLocation().toFile();

             if (outputDir.isDirectory())
                processDirectory(outputDir);
             break;
          }
          case IClasspathEntry.CPE_SOURCE: {
             IPath outputPath = cpe.getOutputLocation();
             File  outputDir;

             if (outputPath != null) outputDir = outputPath.toFile();
             else outputDir = wsRoot.getLocation().append(entryPath).toFile();

             if (outputDir.isDirectory())
                processDirectory(outputDir);
             break;
          }
       }
    }
}
```

Java Structures: `IJavaElement`

- **`IMember`** – base for IField, IMethod, IType
  - `getFlags()` – modifiers
  - `getCompilationUnit()` – only if a source member
  - `getClassFile()` – only if a binary member
  - `getDeclaringType()` – warning: sometimes null



- **`IType`** – N.B.: returns **unresolved** type references!
  - `String getFullyQualifiedName()`
  - `String getKey()` – unique "binding key"
  - `boolean isClass(), isInterface(), isAnonymous(), isLocal(), isEnum(), isAnnotation()`
  - `IMethod[] getMethods()` – always exist
  - `IField[] getFields()` – always exist
  - `IType[] getTypes()` – always exist
  - `String getSuperclassName()` – **unresolved name** for source types
  - `String[] getSuperInterfaceNames()` – **unresolved names** for source types
  - `ITypeParameter[] getTypeParameters()` – if generic
  - `getField(String),getMethod(String,…),getType(String)` – may not exist!
  - `createField(…), createMethod(…), createType(…)` – specify source text

Java Structures: `IJavaElement`

- `IMethod`
  - `String[] getParameterTypes()` – unresolved names if source
  - `String getReturnType()` – unresolved name if source
  - `ITypeParameter[] getTypeParameters()` – if generic
  - `String[] getExceptionTypes()` – unresolved names if source
  - `boolean isConstructor()`
  - `String getKey()` – unique “binding key”
- `IField`
  - `String getKey()` – unique “binding key”

JavaCore Utility Methods

`org.eclipse.jdt.core.JavaCore`

- static utility methods to "translate" from `IResource` domain to `IJavaElement` domain
  - `IJavaProject JavaCore.create(IProject)`
  - `IJavaElement JavaCore.create(IFile)`
  - `IJavaElement JavaCore.create(IFolder)`
- additional utilities for manipulating project options, classpaths, …
  - `Hashtable getDefaultOptions()`
  - `IClasspathEntry newLibraryEntry(…)`
  - `IClasspathEntry newProjectEntry(…)`
  - `run(IWorkspaceRunnable)`

Java AST's: Overview

`org.eclipse.jdt.core.dom`

- **`ASTNode`** – base type of a rich hierarchy of AST node types (good, lots of type safety)
- Produced from source by **`ASTParser`**
- Eclipse 3.1 has full support for Java 5.0 language
- Create directly using **`AST`** – **`ASTNode`** factory

- **`IBinding's`** – resolved entity references (type, field, method, variable)
  - available via **`xxx.resolveBinding()`**

- **`ASTView`** – visualization of AST of Java source file
  - http://eclipse-plugins.info/eclipse/plugins.jsp?category=Code+mngt

Java AST's: Parsing 1 CU

- Parsing for 1 CU:

```
CompilationUnit parseString(String src, String fileName, IJavaProject p) {
   Document doc = new Document(src);
   ASTParser parser = ASTParser.newParser(AST.JLS3); // JLS3 == Java 1.5

   parser.setSource(doc.get().toCharArray());
   parser.setProject(p);
   parser.setUnitName(fileName);
   parser.setResolveBindings(true); // else foo.resolveBinding() == null!

   return (CompilationUnit) parser.createAST(new NullProgressMonitor());
}

CompilationUnit parseFile(ICompilationUnit icu) {
   ASTParser parser = ASTParser.newParser(AST.JLS3); // JLS3 == Java 1.5

   parser.setSource(icu);
   parser.setResolveBindings(true); // else foo.resolveBinding() == null!

   return (CompilationUnit) parser.createAST(new NullProgressMonitor());
}
```

- AST nodes keep back-pointers to parent
  - Keep reference to 1 node => you're keeping them all
- AST's are memory-intensive (>= 1MB/compilation unit)
  - Only hold onto small constant # of AST's at any time

Java AST's: Parsing Multiple CU's

```
// Global analysis: use "parsing pipeline" (shares IBindings), process 1 at a time
class BatchASTCreator {
    private IProgressMonitor fMonitor;
    private ASTVisitor fVisitor;

    public BatchASTCreator(ASTVisitor visitor, IProgressMonitor pm) {
        fMonitor = pm;
        fVisitor = visitor;
    }
    private ASTParser getParser(WorkingCopyOwner wco, IJavaProject javaProject) {
        ASTParser parser = ASTParser.newParser(AST.JLS3);
        parser.setProject(javaProject);  // Set parser options
        parser.setResolveBindings(true);
        parser.setWorkingCopyOwner(wco);
        parser.setCompilerOptions(ASTParser.getCompilerOptions(javaProject));
        return parser;
    }
    public void collect(final ICompilationUnit[] cus, WorkingCopyOwner wco) {
        IJavaProject project = cus[0].getJavaProject();
        ASTParser    p = getParser(wco, project);
        ASTRequestor requestor = new ASTRequestor() {
            public void acceptAST(ICompilationUnit source, CompilationUnit ast){
                if (BatchASTCreator.this.fMonitor.isCanceled())
                    throw new OperationCanceledException("Cancelled.");
                ast.accept(fVisitor); ← process AST but DON'T KEEP IT!
            }
        };
        p.createASTs(cus, new String[0], requestor, fMonitor);
    }
}
```

ASTVisitor for processing

accepts 1 AST @ a time

Java AST's: Correlating to Other Types

- `ASTNode` → `IBinding`
  - various nodes provide a `resolveBinding()` method e.g.
    - `MethodInvocation.resolveBinding()`
    - `MethodInvocation.resolveMethodBinding()` – call target
    - `Name.resolveBinding()`
    - `FieldAccess.resolveBinding()`
- `IBinding` → `ASTNode`
  - `CompilationUnit.findDeclaringNode(IBinding)`
- `IBinding` → `IType/IField/`…
  - `IBinding.getJavaElement()`
  - `Bindings.findType(ITypeBinding, IJavaProject)`
- `IType/IField/`… → `ASTNode`
  - `ASTNodeFinder.findField(IField)`, …

N.B.: IBindings are not canonicalized across compilation unit boundaries when AST's are created by separate calls to `ASTParser.createAST()`

- Use `Bindings.equals()` to compare in that case

Java AST's: Visitor Interface

```
public abstract class ASTVisitor {
    // The following methods get called before any children
    // are visited. If a given visit() method implementation
    // returns false, its children are NOT visited.
    boolean visit(MethodDeclaration decl);
    boolean visit(MethodInvocation inv);
    boolean visit(Assignment a);
    boolean visit(ArrayAccess aa);
    boolean visit(Initializer init);
    //…

    // The following methods get called after children have
    // been visited, regardless of whether children get visited.
    boolean endVisit(MethodDeclaration decl);
    boolean endVisit(MethodInvocation inv);
    boolean endVisit(Assignment a);
    //…
}

class ASTNode {
    //…
    void accept(ASTVisitor v);
}
```

Java AST's: Visitor Example

```
boolean isMisplacedMethod(MethodDeclaration method) {
    final List<IVariableBinding> params = new ArrayList(); // parameter IBindings

    for(SingleVariableDeclaration svd: method.parameters())
       params.add(svd.resolveBinding());

    final boolean[] gotAnAnswer = new boolean[] { false; };
    IVariableBinding fTargetParam = null;

    method.accept(new ASTVisitor() {
       public boolean visit(MethodInvocation inv) { // look at call sites
          Expression rcvr = inv.getExpression(); // null if implicit 'this' call

          if (rcvr == null) { // definitely not a candidate
             fTargetParam = null;
             gotAnAnswer[0] = true;
             return false; // don't bother looking at children (actual arguments)
          } else if (!(rcvr instanceof SimpleName))
             return true; // examine children (actual arguments)

          SimpleName rcvrNm = (SimpleName) rcvr;
          IBinding rcvrBinding = rcvrNm.resolveBinding(); // what does this refer to?

          if (!params.contains(rcvrBinding)) return false; // not a param reference

          if (fTargetParam == null && !gotAnAnswer[0]) {
             fTargetParam = (IVariableBinding) rcvrBinding;
             gotAnAnswer[0] = true;
          } else if (!Bindings.equals((IVariableBinding) rcvrBinding, fTargetParam))
             fTargetParam = null;
          return true;
       }
    });
    return (fTargetParam != null);
 }
```

Java AST's: Rewriting

- Rewriting operations encapsulated as a **`Change`** object
- Core AST modification API: **`ASTRewrite`**
  - **`void remove(ASTNode, …)`**
  - **`void replace(ASTNode, ASTNode, …)`**
  - **`void set(ASTNode, StructuralPropertyDescriptor)`**
  - **`ListRewrite getListRewrite(ASTNode, ChildListPropertyDescriptor)`**
  - **`TextEdit rewriteAST(IDocument, …)`**
- N.B.: Refactoring infrastructure wraps much of this, so that refactorings only have to produce a **`Change`** object

Java AST's: Rewriting Top-Level Flow

```
class ProtectConstructor {
   IMethodBinding fCtorBinding; // got this from somewhere…

   CompilationUnit getAST(ICompilationUnit icu) {/* use ASTParser shown earlier */}

   public Change createChange(ICompilationUnit icu) throws CoreException {
      ITextFileBufferManager bufMgr     = FileBuffers.getTextFileBufferManager();
      CompilationUnit        unitAST    = getAST(icu);
      CompilationUnitChange  unitChange = new CompilationUnitChange("protect",icu);
      ASTRewrite    cuRewriter = ASTRewrite.create(unitAST.getAST());
      MultiTextEdit root       = new MultiTextEdit();

      try {
         ITextFileBuffer buf = bufMgr.getTextFileBuffer(icu.getFullPath());
         TextEditGroup   eg  = new TextEditGroup("protect ctor"); // UI label

         unitChange.setEdit(root);

         protectConstructor(unitAST, cuRewriter, eg); // rewriting happens here

         unitChange.addTextEditGroup(eg);
         root.addChild(cuRewriter.rewriteAST(buf.getDocument(),
                       icu.getJavaProject().getOptions(true)));
      } finally {
         bufMgr.disconnect(icu.getFullPath());
      }
      return unitChange;
   }
   // … continued on next slide …
}
```

Java AST's: Rewriting Details

```
class ProtectConstructor {
    // …continued…

    // Does the actual rewriting
    void protectConstructor(CompilationUnit unitAST, ASTRewrite cuRewriter,
                            TextEditGroup eg) {
      AST ast = unitAST.getAST(); // get the node factory

      // First, find the node to rewrite by IBinding
      MethodDeclaration ctor  =
          (MethodDeclaration) unitAST.findDeclaringNode(fCtorBinding);

      // Next, get a helper for rewriting a list of AST nodes
      ListRewrite modRewriter =
          cuRewriter.getListRewrite(ctor,
                                    MethodDeclaration.MODIFIERS2_PROPERTY);

      // Create the new Modifier node using the AST node factory
      Modifier newMod = ast.newModifier(Modifier.ModifierKeyword.PROTECTED_KEYWORD);

      // Add new Modifier to beginning of modifier list
      modRewriter.insertFirst(newMod, eg);
    }
}
```

Java AST's: Applying a Change

```
Change change = createChange(); // create a change…

try {
    change.initializeValidationState(pm);
    //…
    if (!change.isEnabled())
        return;

    RefactoringStatus valid =
                change.isValid(new NullProgressMonitor());
    if (valid.hasFatalError())
        return;

    Change undo = change.perform(new NullProgressMonitor());

    if (undo != null) {
        undo.initializeValidationState(new NullProgressMonitor());
        // do something with the undo object
    }
} finally {
    change.dispose();
}
```

JDT Structures: Type Representations

- **IType's** (**IJavaElement's**)
  - super-types are unresolved **Strings**
  - returned by **SearchEngine** queries, produced by certain Java-oriented views (e.g. Package Explorer, Outline)
  - hard/impossible to find **IType** for certain cases of anonymous/nested types
  - no representation for array types, and can't create yourself
- **ITypeBinding's** (**IBinding's**)
  - associated with AST nodes
  - representations exist for every type explicitly manifested in the program
  - can't create yourself (constructors private)
  - fully resolved, cover everything, but expensive
- **Type's** (ASTNode wrapping a type name)
  - unresolved; need to call **resolveBinding()**
  - expensive; holding onto these holds onto entire AST's
- **TType's** (JDT/UI refactoring) –representation of choice for global analysis!
  - lightweight, creatable from **IBinding's**, handle generics and wildcards
  - constant-time isSupertype() query

Java Structures: ITypeHierarchy

- Create using, e.g.
  - `IJavaProject.newTypeHierarchy()`
  - `IType.newSupertypeHierarchy()`
- API:
  - `IType[] getAllClasses()`
  - `IType[] getAllInterfaces()`
  - `IType[] getAllSubtypes(IType ofType)`
  - `IType[] getSubtypes(IType ofType)`
  - …
- Caveats:
  - Very slow to build complete hierarchy
  - Omissions (certain interfaces may not appear)
  - `java.lang.Object` is not a supertype of any interface

Java Structures: Search Engine

Searching for references to a given **IJavaElement**:

```
IJavaSearchScope createSearchScope(IMethod ctor) throws JavaModelException {
   return SearchEngine.createJavaSearchScope(new IJavaElement[] { method });
}

SearchPattern createSearchPattern() {
   return SearchPattern.createPattern(method,
                                      IJavaSearchConstants.REFERENCES,
                                      SearchUtils.GENERICS_AGNOSTIC_MATCH_RULE);
}

SearchMatch[] searchForCalls(IProgressMonitor pm) throws CoreException {
   IMethod method = …; // get this from somewhere, e.g., Outline View
   IJavaProject javaProject = method.getJavaProject();
   SearchEngine engine = new SearchEngine();
   final List/*<SearchMatch>*/ results = new ArrayList();

   engine.search(createSearchPattern(),
      new SearchParticipant[]{ SearchEngine.getDefaultSearchParticipant() },
      createSearchScope(method, javaProject),
      new SearchRequestor() {
         public void acceptSearchMatch(SearchMatch m)
               throws CoreException {
            results.add(m);
         }
      },
      pm);
   return (SearchMatch[]) results.toArray(new SearchMatch[results.size()]);
}
```

Additional Reference Material

- "Java Developer's Guide to Eclipse," **2nd Edition (for Eclipse 3.0)**,
  D'Anjou, Fairbrother, Kehn, Kellerman, McCarthy,
  Addison-Wesley, 2005
- "Contributing to Eclipse"
  Gamma, Beck, Addison-Wesley, 2004
- "Official Eclipse 3.0 FAQ"
  Arthorne, Laffra, Addison-Wesley, 2004
  - http://eclipsefaq.org (partial online version)
- Eclipse Bugzilla DB: *http://bugs.eclipse.org*
- Use the source, Luke!

Break #1: 15 minutes

- Topics:
  - how to not be seen
  - lemmings I have known
  - a funny thing happened on the way to the browser...
  - an XML schema for haiku

Part II: Intraprocedural Use/Def Analysis (1.25 hrs)

- Purpose
  - Provide encapsulation that triggers analysis as UI-invokable gestures for exploring intraprocedural static data-flow relationships in a Java program
- Specifically: display and navigate use-def/def-use (UD-/DU-) chains within the Java source editor
  - modal button that toggles highlighting of UD/DU information (like Java Editor's "mark occurrences")
  - user selects a local variable reference
    - reaching definitions are highlighted
  - user selects a local value definition
    - references that might "see" that definition are highlighted

Use/Def Analysis: Topics

- Anatomy of intraprocedural analysis algorithm for computing local use-def/def-use relationships
- Using Eclipse Java API's
- Creating document and selection listeners
- Creating “annotations” to mark source code entities

Use/Def Analysis: Example

```
class Foo {
    public void foo() {
        int x = 5;
        int y = 12;

        y = 17;

        for(int i=0; i < 5; i++) {
            x = x + y;
        }
        System.out.println(x);
    }
}
```

N.B.: Only concerned with ***local*** *variables*

Use/Def Analysis: Approach

- Cast in terms of "reaching definitions" analysis
  - For each AST node **N**:
    - RD(**N**) = { (**v**,**N'**) | def of **v** at **N'** reaches **N** }
- Follow reaching definitions analysis by simple filter:
    - UD(ref **v**) = { (**v**,**N**) | (**v**,**N**) ∈ RD(**v**) }
    - DU(**v**,**N**) = { ref **v** | (**v**,**N**) ∈ RD(**N**) }

Reaching Definitions Analysis: Constraint Variable Notation

RD_{entry}[**n**]	the set of definitions reaching the entry point of AST node **n**
RD_{exit}[**n**]	the set of definitions leaving the exit point of AST node **n**
(**v**,**n**)	a definition of variable **v** at AST node **n**
(**v**,*)	a definition of variable **v** at *any* AST node

Reaching Definitions Analysis: Constraint Notation

$RD[\mathbf{n}] \subseteq RD[\mathbf{n}']$	The set of reaching definitions of AST node **n** is a subset of that of **n**′
$(\mathbf{v},\mathbf{n}) \in RD_{exit}[\mathbf{n}]$	The definition of variable **v** at AST node **n** reaches AST node **n**′
$S \setminus S'$	set difference $\{ d \mid d \in S \wedge d \notin S' \}$

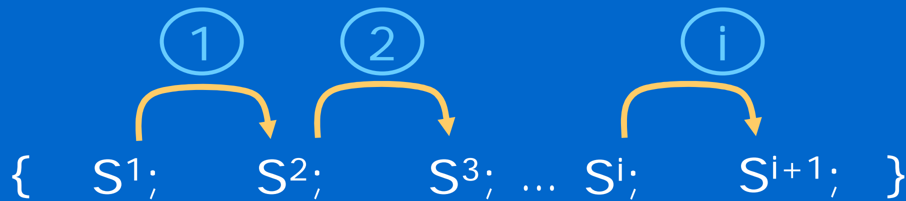
Reaching Definitions Constraints: Data-flow

construct	constraints	description
v = E	$(\mathbf{v},\mathbf{v{=}E}) \in RD_{exit}[\mathbf{v{=}E}]$	definition of value for **v** reaches exit
" "	$RD_{entry}[\mathbf{v{=}E}] \setminus \{(\mathbf{v},*)\} \subseteq RD_{exit}[\mathbf{v{=}E}]$	anything not killed by definition reaches exit
v++	*<similar to assignment>*	

Reaching Definitions Constraints: Control-flow

- In general: if statement S flows to S′ generate constraint:
  $RD_{exit}[S] \subseteq RD_{entry}[S']$

Reaching Definitions Control-flow Constraints: Blocks

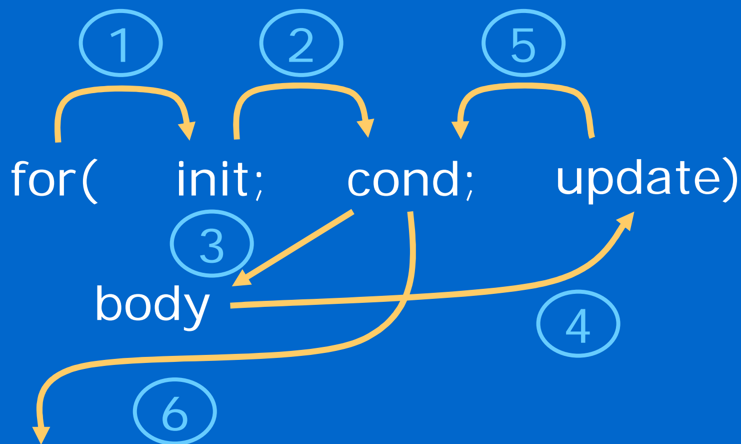


1. $S^1_{exit} \subseteq S^2_{entry}$
2. $S^2_{exit} \subseteq S^3_{entry}$

…

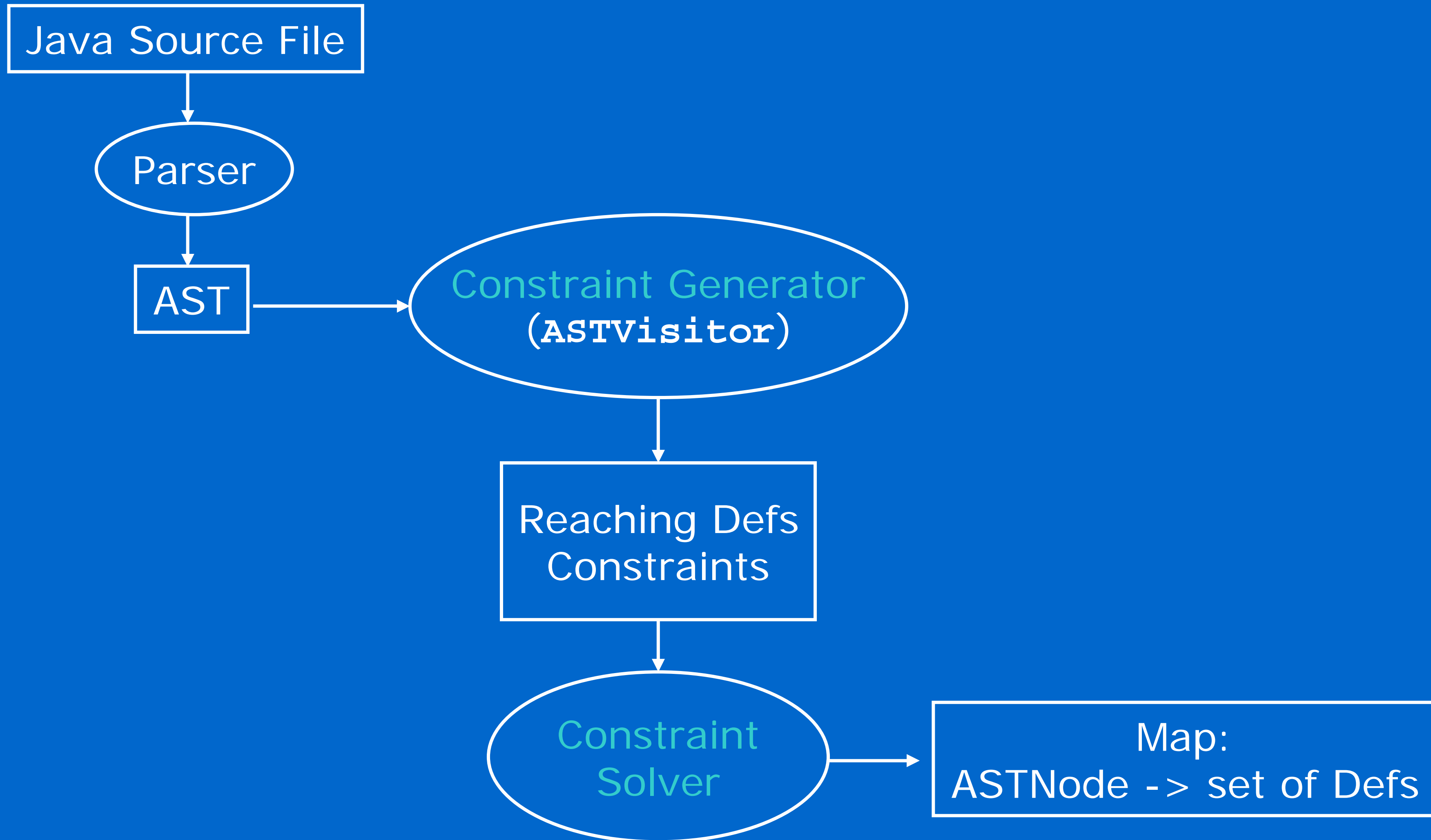
i. $S^i_{exit} \subseteq S^{i+1}_{entry}$

Reaching Definitions Control-flow Constraints: For Loops



1. $for_{entry} \subseteq init_{entry}$
2. $init_{exit} \subseteq cond_{entry}$
3. $cond_{exit} \subseteq body_{entry}$
4. $body_{exit} \subseteq update_{entry}$
5. $update_{exit} \subseteq cond_{entry}$
6. $cond_{exit} \subseteq for_{exit}$

Anatomy of Reaching Defs Analysis: Solution Architecture



Anatomy of Reaching Defs Analysis: Generic Constraint Generation API's

Based (somewhat loosely) on API's in

`org.eclipse.jdt.internal.corext.refactoring.typeconstraints2`

```
abstract class ConstraintTerm { // a node in constraint graph
    public interface ITermProcessor {
        void processTerm(ConstraintTerm term);
    }

    public void recomputeEstimate(IEstimateEnvironment env) { }
    abstract void processTerms(ITermProcessor processor);
}

abstract class ConstraintOperator {}//sub-class for specific analyses

class Constraint { // an edge in the constraint graph
    ConstraintTerm fLHS, fRHS; ConstraintOperator fOperator;

    Constraint(ConstraintVariable l, ConstraintOperator o,
               ConstraintVariable r) {
        fLHS = l; fRHS = r; fOperator = o;
    }
    ConstraintTerm getLHS() { return fLHS; }
    ConstraintTerm getRHS() { return fRHS; }
    ConstraintOperator getOperator() { return fOperator; }
}
```

Anatomy of Reaching Defs Analysis: Generic Constraint Generation API's

```
class ConstraintVisitor extends ASTVisitor {// traverse AST &
                                            // generate constraints
   ConstraintCreator fCreator;
   List<Constraint> fCons = new HashSet(); // collects results

   ConstraintVisitor(ConstraintCreator cc) { fCreator = cc; }

   boolean visit(ArrayAccess access) {
      fCons.addAll(fCreator.create(access));
   }
   boolean visit(Assignment assign) {
      fCons.addAll(fCreator.create(assign));
   }
   //…
}

abstract class ConstraintCreator {
   // generate constraints for each language construct
   abstract List<Constraint> create(ArrayAccess);
   abstract List<Constraint> create(Assignment);
   abstract List<Constraint> create(ConditionalExpression);
   abstract List<Constraint> create(MethodDeclaration);
   abstract List<Constraint> create(MethodInvocation);
   //…
}
```

Anatomy of Reaching Defs Analysis: Constraint Generation

```
class RDConstraintTermFactory {
   // … implementation on following slide …
   ConstraintTerm createEntryLabel(ASTNode node);    // RDentry[n]
   ConstraintTerm createExitLabel(ASTNode node);     // RDexit[n]
   ConstraintTerm createDefinitionLiteral(IVariableBinding v,ASTNode n);//(v,n)
   ConstraintTerm createDefinitionWildcard(IVariableBinding v); // (v,*)
}

// Intraprocedural single CU analysis: ok to hold onto ASTNodes and IBindings
class NodeLabel extends ConstraintTerm {
   ASTNode fNode;
   NodeLabel(ASTNode node) { fNode= node; }
}
class EntryLabel extends NodeLabel { // RDentry[n]
   EntryLabel(ASTNode node) { super(node); }
   public String toString() { return "RD@entry[" + node + "]"; }
}
class ExitLabel extends NodeLabel { // RDexit[n]
   ExitLabel(ASTNode node) { super(node); }
   public String toString() { return "RD@exit[" + node + "]"; }
}

class DefinitionLiteral extends ConstraintTerm { // (v,n)
   IVariableBinding fVarBinding; ASTNode fLabel;
   DefinitionLiteral(IVariableBinding v) { this(v, null); } // (v,*)
   DefinitionLiteral(IVariableBinding v, ASTNode n){ fVarBinding = v;fLabel = n;}
   public String toString() { return "(" + fVarBinding + "," + fLabel + ")"; }
}
```

Anatomy of Reaching Defs Analysis: Constraint Generation

```
class RDConstraintTermFactory {
   // Responsible for "canonicalizing" constraint terms
   Map<ASTNode, ConstraintTerm> fTermMap;

   ConstraintTerm createEntryLabel(ASTNode n) {
      ConstraintTerm t = fTermMap.get(n);
      if (t == null)
         fTermMap.put(n, t = new EntryLabel(n));
      return t;
   }

   Map<IVariableBinding,Map<ASTNode,DefinitionLiteral>> fVarMap =
         new LinkedHashMap(); // LinkedXXX() for determinism

   ConstraintTerm createDefinitionLiteral(IVariableBinding b, ASTNode n) {
      Map<ASTNode,DefinitionLiteral> label2DefLit = (Map) fVarMap.get(b);

      if (label2DefLit == null)
         fVarMap.put(var, label2DefLit = new LinkedHashMap());

      DefinitionLiteral d = (DefinitionLiteral) label2DefLit.get(label);

      if (d == null) {
         d = new DefinitionLiteral(var, label);
         label2DefLit.put(label, d);
      }
      return d;
   }
   //… similar methods for creating other ConstraintTerm types…
}
```

Anatomy of Reaching Defs Analysis: Constraint Generation

```
class SubsetOperator extends ConstraintOperator { }

class RDConstraintCreator extends ConstraintCreator {
   RDConstraintTermFactory fFactory;

   // convenience method
   Constraint newSubsetConstraint(ConstraintTerm l,ConstraintTerm r) {
      return new Constraint(l, r, SubsetOperator.getInstance());
   }

   //
   // 1 method per language construct to generate constraints
   //
   List<Constraint> create(Assignment a) {
      //… see next slide …
   }

   List<Constraint> create(ForStatement f) {
      //… see subsequent slide …
   }

   // … other language constructs …
}
```

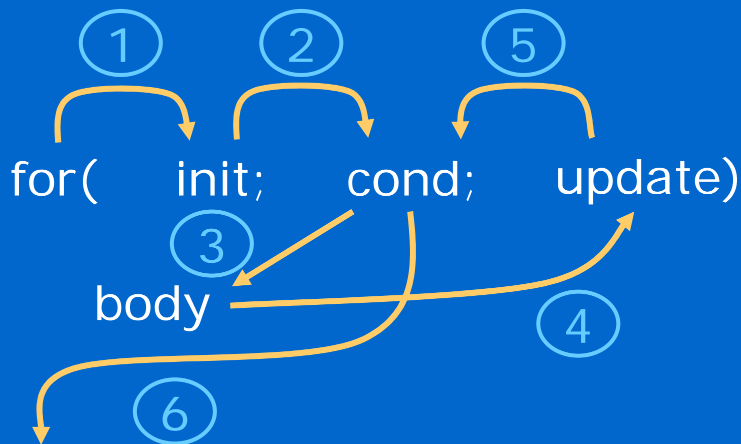
Data-flow constraints: Assignment

construct	constraints	description
v = E	(**v**,**v=E**) $\in$ RD_{exit}[**v=E**]	definition of value for **v** reaches exit
" "	RD_{entry}[**v=E**] \ {(**v**,*)} $\subseteq$ RD_{exit}[**v=E**]	anything not killed by definition reaches exit

Data-flow constraints: Assignment

```
public List<Constraint> create(Assignment assign) {
    // Restriction: only handle local variables (intraprocedural)
    Expression lhs = assign.getLeftHandSide();
    Expression rhs = assign.getRightHandSide();

    // if LHS isn't a simple name, it can't be a local variable
    if (lhs.getNodeType() != ASTNode.SIMPLE_NAME) return EMPTY_LIST;

    SimpleName name = (SimpleName) lhs;
    IBinding   nameBinding = name.resolveBinding();

    // if name isn't a variable reference, ignore it
    if (nameBinding.getKind() != IBinding.VARIABLE) return EMPTY_LIST;

    IVariableBinding varBinding= (IVariableBinding) nameBinding;

    // if variable reference refers to a field, ignore it
    if (varBinding.isField()) return EMPTY_LIST;

    ConstraintTerm assignEntry = fVariableFactory.createEntryLabel(assign);
    ConstraintTerm def       = fVarFactory.createDefinition(varBinding, assign);
    ConstraintTerm defWild   = fVarFactory.createDefinition(varBinding); // (v,*)
    ConstraintTerm rdExit    = fVarFactory.createExitLabel(assign);
    ConstraintTerm diff      = new ReachingDefsDifference(assignEntry, defWild);
    List<Constraint> result = new List<Constraint>();

    result.add(newSubsetConstraint(def, rdExit)); // (v,v=E) ∈ RD_exit[v=E]
    result.add(newSubsetConstraint(diff,rdExit)); // RD_entry[v=E] \ {(v,*)} ⊆ RD_exit[v=E]
    return result;
}
```

Control-flow Constraints: For Loops



1. $for_{entry} \subseteq init_{entry}$
2. $init_{exit} \subseteq cond_{entry}$
3. $cond_{exit} \subseteq body_{entry}$
4. $body_{exit} \subseteq update_{entry}$
5. $update_{exit} \subseteq cond_{entry}$
6. $cond_{exit} \subseteq for_{exit}$

Control-flow Constraints: For Loops

```
public List<Constraint> create(ForStatement forStmt) {
    // Simplification: exactly one init expr, a condition, exactly one update expr
    Statement body = forStmt.getBody();
    Expression        cond    = forStmt.getExpression();
    List<Expression> inits   = forStmt.initializers();
    List<Expression> updates = forStmt.updaters();
    Expression        init    = (Expression) inits.get(0);   // assume one init
    Expression        update  = (Expression) updates.get(0); // assume one update
    List<Constraint> result  = new ArrayList();

    ConstraintTerm forEntry    = fVariableFactory.createEntryLabel(forStmt);
    ConstraintTerm forExit     = fVariableFactory.createExitLabel(forStmt);
    ConstraintTerm initEntry   = fVariableFactory.createEntryLabel(init);
    ConstraintTerm initExit    = fVariableFactory.createExitLabel(init);
    ConstraintTerm condEntry   = fVariableFactory.createEntryLabel(cond);
    ConstraintTerm condExit    = fVariableFactory.createExitLabel(cond);
    ConstraintTerm updateEntry = fVariableFactory.createEntryLabel(update);
    ConstraintTerm updateExit  = fVariableFactory.createExitLabel(update);
    ConstraintTerm bodyEntry   = fVariableFactory.createEntryLabel(body);
    ConstraintTerm bodyExit    = fVariableFactory.createExitLabel(body);

    result.add(newSubsetConstraint(forEntry,   initEntry));  // 1. for_entry ⊆ init_entry
    result.add(newSubsetConstraint(initExit,   condEntry));  // 2. init_exit ⊆ cond_entry
    result.add(newSubsetConstraint(condExit,   bodyEntry));  // 3. cond_exit ⊆ body_entry
    result.add(newSubsetConstraint(bodyExit, updateEntry));  // 4. body_exit ⊆ update_entry
    result.add(newSubsetConstraint(updateExit, condEntry));  // 5. update_exit ⊆ cond_entry
    result.add(newSubsetConstraint(condExit,   forExit));    // 6. cond_exit ⊆ for_exit

    return result;
}
```

Anatomy of Reaching Defs Analysis: Constraint Solution

Build Constraint Graph

Initialize Estimates

Process Work-List

```
class ConstraintGraph {
   List<Constraint>    fConstraints;
   Set<ConstraintTerm> fAllTerms;
   Map<ConstraintTerm,List<Constraint>> fEdgeMap;

   class TermDecorator implements ITermProcessor {
      Constraint fConstraint;
      void setConstraint(Constraint c) {fConstraint=c;}
      public void processTerm(ConstraintTerm term) {
         addToEdgeList(term, fConstraint);
         fAllTerms.add(term);
      }
   }
   void initialize() { // turn Constraints into graph
      TermDecorator decorator = new TermDecorator();
      for(Constraint c: getConstraints()) {
         ConstraintTerm lhs = c.getLeft();
         ConstraintTerm rhs = c.getRight();

         decorator.setConstraint(c);
         lhs.processTerms(decorator);
         rhs.processTerms(decorator);
      }
   }
}
```

Anatomy of Reaching Defs Analysis: Constraint Solution

Build Constraint Graph

1 Initialize Estimates

2 Process Work-List

sets monotonically increase in size!

1

```
void initializeEstimates() {
    for(ConstraintTerm t: graph.getVariables()) {
        if (t instanceof DefinitionLiteral)
            setEstimate(t, new DefinitionSet(t));
        else
            setEstimate(t, new DefinitionSet());
    }
}
```

2

```
void solveConstraints() {
    while (!workList.empty()) {
        ConstraintTerm t = workList.pop();
        for(c: getConstraintsInvolving(t)) {
            satisfyConstraint(c);
        }
    }
}
void satisfyConstraint(IConstraint c) {
    ConstraintTerm lhs = c.getLHS();
    ConstraintTerm rhs = c.getRHS();
    DefinitionSet lhsEst = getEstimate(lhs);
    DefinitionSet rhsEst = getEstimate(rhs);
    if (!rhsEst.containsAll(lhsEst))
        setEstimate(rhs, rhsEst.unionWith(lhsSet));
}
```

Computing Def/Use Relationships from Reaching Definitions

refsTo(**def**) = { nodes **n** | isRef(**n**) ^ **def** ∈ reachingdefs(**n**) }

```
Set<ASTNode> findRefsToDef(ASTNode def,
                           final IEstimateEnvironment reachingDefs) {
   final Set<ASTNode> result= new HashSet();

   ASTNode    method = getOwningMethod(def);
   SimpleName name   = (SimpleName) ((Assignment) def).getLeftHandSide();

   final IVariableBinding  defBinding =
                                        (IVariableBinding) name.resolveBinding();
   final DefinitionLiteral defLit     = new DefinitionLiteral(defBinding, def);

   // Search AST for variable references that refer to def
   method.accept(new ASTVisitor() {
      public boolean visit(SimpleName node) {
         if (!Bindings.equals(node.resolveBinding(), defBinding))
            return false;

         DefinitionSet rds =
            reachingDefs.getEstimate(fVariableFactory.createEntryLabel(node));

         if (rds.contains(defLit))
            result.add(node);
         return false;
      }
   });
   return result;
}
```

Computing Use/Def Relationships from Reaching Definitions

$defsOf(ref) = \{ d \in reachingdefs(\mathbf{ref}) \mid var(d) = binding(\mathbf{ref}) \}$

```
Set<ASTNode> findDefsForRef(ASTNode ref,
                            IVariableBinding varBinding,
                            IEstimateEnvironment rds) {
   DefinitionSet defs =
          rds.getEstimate(fVariableFactory.createEntryLabel(ref);

   Set<ASTNode> result = new HashSet();

   for(DefinitionLiteral d: defs) {
      if (Bindings.equals(varBinding, def.getVarBinding()))
         result.add(def.getLabel());
   }
   return result;
}
```

Use/Defs UI Integration: Overview

- Basic components:
  - Create toolbar Action to toggle "highlight uses/defs" mode
  - Re-analyze when Java editor source document changes
    - create a "Document Listener" to trap document changes
  - Update highlighting when selection changes
    - create a "Selection Listener" to trap editor selections
    - create "Annotations" to indicate desired source highlighting

Use/Defs UI Integration: Action

```
class MarkUseDefsAction implements IWorkbenchWindowActionDelegate {
   boolean                   fInstalled = false;
   AbstractTextEditor        fEditor;
   IDocumentListener         fDocumentListener = new MDUDocumentListener();
   ISelectionChangedListener fSelectListener  = new MDUSelectionListener(document);

   public void run(IAction action) {
      fEditor = (AbstractTextEditor) PlatformUI.getWorkbench().
                     getActiveWorkbenchWindow().getActivePage().getActiveEditor();
      IDocument doc = getDocumentProvider().getDocument(getEditorInput());

      if (!fInstalled) {
         registerListeners(doc);
         fInstalled = true;
      } else {
         unregisterListeners(doc);
         fInstalled = false;
      }
   }

   void registerListeners(IDocument document) {
      getSelProvider().addSelectionChangedListener(fSelectListener);
      document.addDocumentListener(fDocumentListener);
   }
   void unregisterListeners(IDocument document) {
      getSelProvider().removeSelectionChangedListener(fSelectListener);
      document.removeDocumentListener(fDocumentListener);
   }
   ISelectionProvider getSelProvider() { return fEditor.getSelectionProvider(); }
   IDocumentProvider getDocProvider()  { return fEditor.getDocumentProvider(); }
}
```

register listeners

Use/Defs UI Integration: Listeners

```
class MarkDefsUseAction {
   // …
   CompilationUnit fCompilationUnit = null; // AST cache

   // … nested class, since needs access field fCompilationUnit …
   class MDUDocumentListener implements IDocumentListener {

      public void documentAboutToBeChanged(DocumentEvent event) {
         // … do nothing …
      }

      public void documentChanged(DocumentEvent event) {
         fCompilationUnit = null;
      }
   }
}
```

Use/Defs UI Integration: Listeners

```
class MarkDefsUseAction {
   // …

   // … nested class, since needs access field fCompilationUnit …
   class MDUSelectionListener implements ISelectionChangedListener {
      private final IDocument fDocument;

      private SelectionListener(IDocument document) {
         fDocument = document;
      }

      public void selectionChanged(SelectionChangedEvent event) {
         ISelection selection = event.getSelection();

         if (selection instanceof ITextSelection) {
            ITextSelection textSel = (ITextSelection) selection;

            int offset = textSel.getOffset();
            int length = textSel.getLength();

            recomputeAnnotationsForSelection(offset, length, fDocument);
         }
      }
   }
}
```

Use/Defs UI Integration: Annotations

```
class MarkDefsUseAction {
   // …
   void recomputeAnnotationsForSelection(int offset, int length,
                                          IDocument document) {
      IAnnotationModel annotationModel =
           fDocumentProvider.getAnnotationModel(getEditorInput());

      // Get AST for the editor document & find the selected ASTNode
      CompilationUnit cu   = getCompilationUnit(); // use ASTParser
      ASTNode selectedNode = NodeFinder.perform(cu, offset, length);

      // Call the analyzer described earlier
      UseDefAnalyzer uda      = new UseDefAnalyzer(cu);
      Set<ASTNode>   usesDefs = uda.findUsesDefsOf(selectedNode);

      // Convert ASTNodes to document positions (offset/length)
      Position[] positions = convertNodesToPositions(usesDefs);

      placeAnnotations(
         convertPositionsToAnnotationMap(positions, document),
         annotationModel);
   }
}
```

call analyzer

add annotations

Use/Defs UI Integration: Annotations

```
class MarkDefsUseAction {
   // …
   Map<Annotation,Position>
   convertPositionsToAnnotationMap(Position[] positions,
                                   IDocument document) {
      Map<Annotation,Position> posMap = new HashMap(positions.length);

      // map each position into an Annotation object
      for(int i = 0; i < positions.length; i++) {
         Position pos = positions[i];

         try { // create Annotation consisting of source text itself
            String message = document.get(pos.offset, pos.length);

            posMap.put(
               new Annotation("com.ibm.pldi2005.useDefAnnotation",
                              false, message),
               pos);
         } catch (BadLocationException ex) {
            continue; // shouldn't happen (we got positions from AST)
         }
      }
      return posMap;
   }
}
```

Use/Defs UI Integration: Annotations

```
class MarkDefsUseAction {
   // …

   void placeAnnotations(Map<Annotation,Position> annotationMap,
                         IAnnotationModel annModel) {
      Object lockObject = getLockObject(annModel);

      synchronized (lockObject) {
         if (annModel instanceof IAnnotationModelExtension) {
            // THE EASY WAY: the more functional API is available
            IAnnotationModelExtension iame =
                                    (IAnnotationModelExtension) annModel;

            iame.replaceAnnotations(fOldAnnotations, annotationMap);
         } else {
            // THE HARD WAY: remove the existing annotations one by one,
            // and add the new annotations one by one…
            removeExistingOccurrenceAnnotations();

            for(Map.Entry<Annotation,Position> e: annotationMap.entrySet()) {
               annModel.addAnnotation(e.getKey(), e.getValue());
            }
         }
      }
   }
}
```

Break #2: 15 minutes

- Topics:
  - Grant proposals for the Ministry of Silly Walks
  - Coding for offensive architectures
  - Lazy/implicit deallocation
  - Curmudgeon, pidgeon and other "woody" words

Part III: Type Analysis (1.25 hours)

- Purpose:
  - implement a global type analysis engine to detect "overly-specific variables"
  - encapsulate as a "smell detector" extension in a simple framework
  - implement a remediating refactoring/ quick-fix

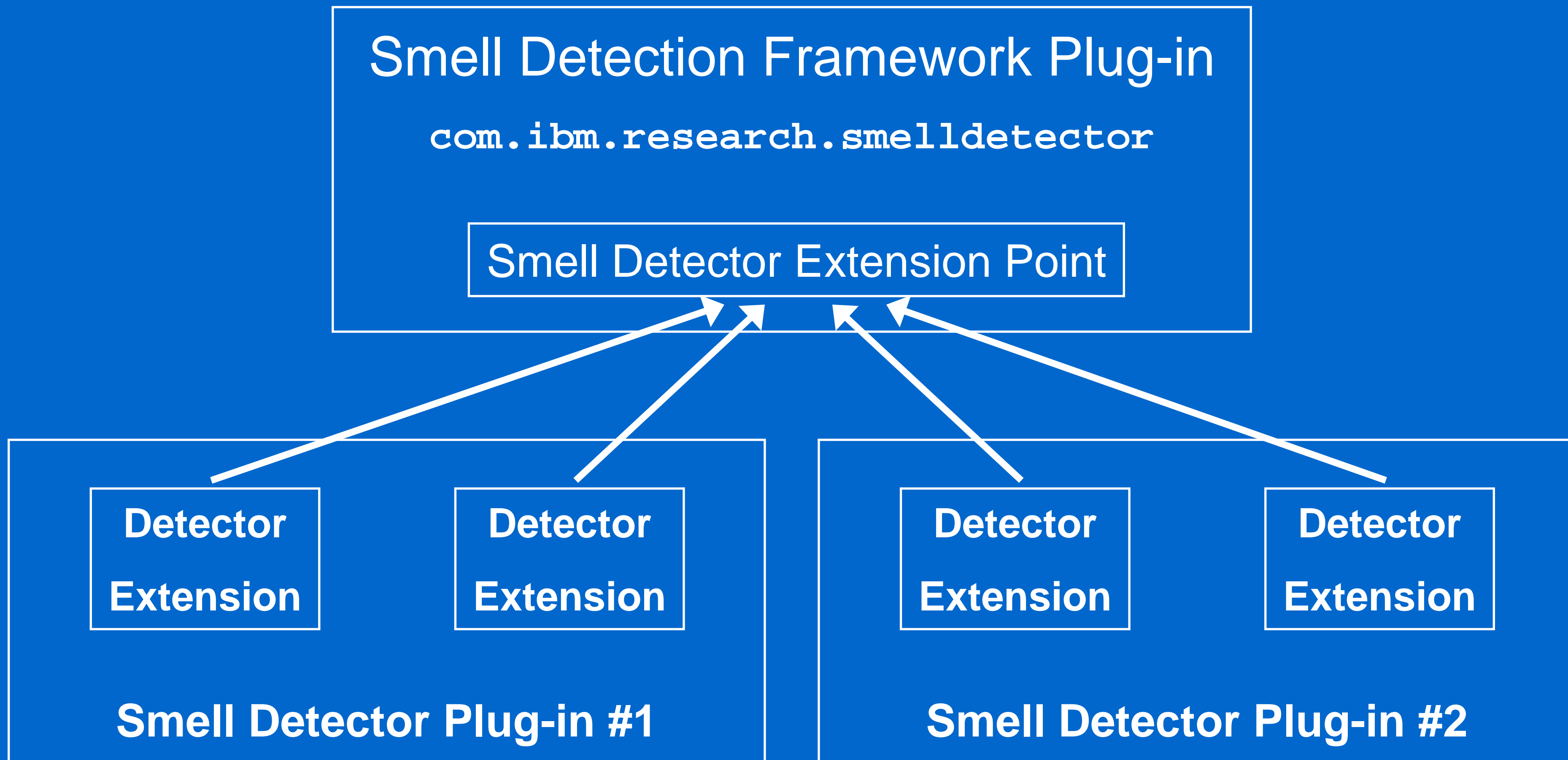
Type Analysis: Topics

- Pluggable "smell detection" framework
  - defined using the Eclipse extension-point mechanism
  - defining a smell detector extension
- Anatomy of a type analysis engine for Java
  - built on the JDT "type constraint" infrastructure
- Type analysis to detect overly-specific variables
- Creating "problem markers" from analysis results
- Creating a quick-fix to rewrite the declaration of an overly-specific variable to the most general possible type as determined by the analysis

Pluggable Smell Detection

- *"If it stinks, change it"* – Grandma Beck
- Code smell: any of a variety of structural defects or undesirable characteristics:
  - duplicated code
  - overly complex methods
  - "shotgun surgery"
  - lack of appropriate reuse
  - inability to reuse component
  - structure does not reflect behavior
  - monolithic class should be a set of components
    - *[Fowler 2000]*

Pluggable Smell Detection: Implementing a Simple Detector



Smell Detector Extension Point

```
<extension-point id="detectors"
  name="Smell Detectors"
  schema="schema/com.ibm.research.smelldetector.detectors.exsd"/>
```

plugin.xml

```
<element name="extension">
  <complexType>
    <sequence>
      <element ref="detector"/>
    </sequence>
    …
  </complexType>
</element>
<element name="detector">
  <complexType>
    <attribute name="name" type="string"/>
    <attribute name="class" type="string">
      <annotation>
        <appInfo>
          <meta.attribute kind="java"/>
        </appInfo>
      </annotation>
    </attribute>
  </complexType>
</element>
```

detectors.exsd
(XML Extension Point Schema)

Smell Detector Extension Point

Smell Detectors

General Information

Extension Point Elements

The following XML elements and attributes are allowed in this extension point:

- extension
  - point
  - id
  - name
- detector
  - name
  - class
- propertyPage
  - class
  - name

New Element

New Attribute

Element Grammar

View or modify the content model of the selected element

- Sequence
  - detector

DTD approximation:
(detector)

Description

Add short description of elements and attributes for documentation purposes. Use HTML tags where appropriate.

Apply

Reset

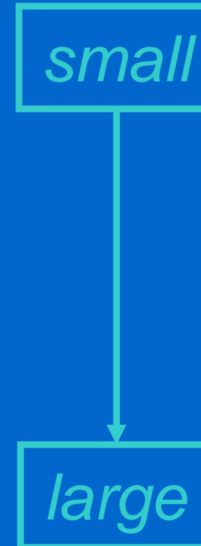
Definition | Documentation | com.ibm.research.smelldetector.detectors.exsd

Smell Detector Extension

```
<extension
  point="com.ibm.research.smelldetector.detectors">
  <detector
      name="Overly Specific Variable"
      class="org.smellsrus.OverlySpecificVariable">
  </detector>
</extension>
```

Smell Detector Extension: Executable Extension

- Implements one or more of these interfaces, depending on granularity of smell:
  - `IFieldSmellDetector`
  - `IMethodSmellDetector`
  - `ITypeSmellDetector`
  - `IUnitSmellDetector`
  - `IPackageSmellDetector`
  - `IProjectSmellDetector`



Smell Detector Interfaces

```
interface ISmellDetector {
  // The marker type indicating a Java code smell
  static final String k_smellMarkerType =
                    "com.ibm.research.smelldetector.smellmarker";

  // Indicates an attribute on a marker used to identify the particular
  // type of smell, for use in remediation.
  static final String k_smellMarkerKind =
                    "com.ibm.research.smelldetector.smellmarkerkind";

  String getName();
}

interface IFieldSmellDetector {
   void runOn(FieldDeclaration field, ICompilationUnit icu, IFile file);
}

interface IMethodSmellDetector extends ISmellDetector {
   void runOn(MethodDeclaration method, ICompilationUnit icu, IFile file);
}

interface ITypeSmellDetector {
   void begin(TypeDeclaration type, ICompilationUnit icu, IFile file);
   void end(TypeDeclaration type, ICompilationUnit icu, IFile file);
}
```

Smell Detection: Overly Specific Variables

Example:

could be just List

```
class Foo {
  public ArrayList toList(String[] args) {
     ArrayList list = new ArrayList();
     for(int i=0; i < args.length; i++)
        list.add(args[i]);
     return list;
  }
  public void foo() {
     List l2 = toList(new String[] { "a", "b" });
     for(Iterator it = l2.iterator(); iter.hasNext();)
        System.out.println(it.next());
  }
}
```

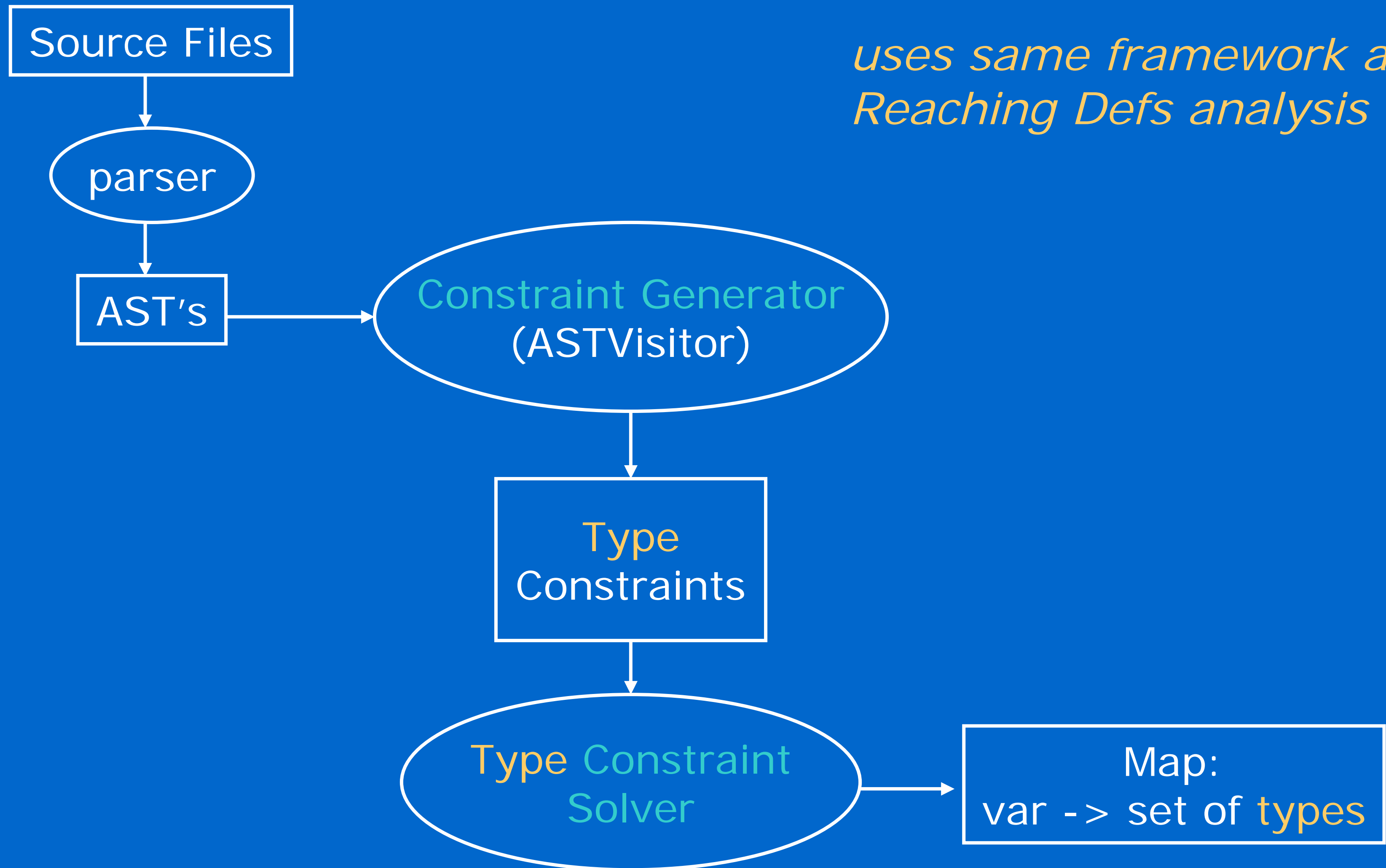
Implementing a Smell Detector: Overly Specific Variables

- Step 1: create new plug-in project
- Step 2: add plug-in dependency for **`smelldetector`** framework plug-in
- Step 3: create extension of **`smelldetector`** extension point
- Step 4: create class implementing **`IUnitSmellDetector`**
- Step 5: create remediator as class implementing **`IMarkerResolutionGenerator`**

Implementing a Smell Detector: Overly Specific Variables

```
class OverlySpecificDetector extends SmellDetectorBase
                             implements IUnitSmellDetector {

   void unitBegin(CompilationUnit unitAST, ICompilationUnit unit, IFile file) {

      OverlySpecificAnalyzer analyzer = new OverlySpecificAnalyzer(unit);

      Map<ICompilationUnit,Map<ConstraintTerm,TypeSet>> unitMap =
                                   analyzer.computeOverlySpecificVariables();

      // Create a marker for each overly-specific variable
      for(ICompilationUnit icu: unitMap.keySet()) {
         Map<ConstraintTerm,TypeSet> termMap = unitMap.get(icu);

         for(ConstraintTerm t: termMap.keySet()) {
            TypeSet ts = termMap.get(t);
            IMarker m  = createMarker(file,
                                      t.toString() + " could be " + ts.enumerate(),
                                      …);
            // crude: pick any member in the estimate TypeSet's upper bound
            m.setAttribute(NEW_TYPE,
                           ts.getUpperBound().anyMember().getQualifiedName());

            // distinguish this smell from other types of smells
            m.setAttribute(SMELL_KIND, "org.pldi2005.overlySpecificVar");
         }
      }
   }
}
```

Anatomy of a Type Analysis Engine



uses same framework as Reaching Defs analysis

Smell Detection: Overly Specific Variables

```
class OverlySpecificAnalyzer {
    Map<ICompilationUnit, Map<ConstraintTerm, TypeSet>>
    computeOverlySpecificVariables() {

        collectConstraints();
        solveConstraints();

        Map<ICompilationUnit, Map<ConstraintTerm, TypeSet>> unitMap =
            new HashMap<ICompilationUnit, Map<ConstraintTerm, TypeSet>>();

        // Examine estimates to determine what's more specific than necessary
        for(n: graph.getNodes()) {
            est = getEstimate(n);

            // if type more specific than necessary, add to result map
            if (estimateMoreGeneralThanDecl(est, n)) {
                ICompilationUnit icu = v.getCompilationUnit();

                Map<ConstraintTerm,TypeSet> termMap =
                                            getOrMakeEntry(unitMap, icu);

                termMap.put(n, est);
            }
        }
        return unitMap;
    }
}
```

Anatomy of a Type Analysis Engine: Overview

- formalism of Palsberg & Schwartzbach, developed in 1990s
  - captures relationships among program constructs
  - original purpose: type inference
    - prove that certain kinds of errors cannot occur at run-time
      - e.g., no "message not understood" errors
- we adapted/extended the formalism to capture the type semantics of Java
- references:
  - "Refactoring for Generalization", Tip, Kiezun, Baeumer, OOPSLA '03
  - "Efficiently Refactoring Java Applications to use Generic Libraries", Fuhrer, Tip, Kiezun, Keller, ECOOP '05

Anatomy of a Type Analysis Engine: Constraint Variable Notation

[E]	the type of expression E
[M]	the return type of method M
[F]	the type of field F
Decl(M)	the type that contains member M
Param(M,i)	the i-th parameter of method M
$<$, $\leq$	subtype relation

Anatomy of a Type Analysis Engine: Type Constraint Notation

$[E] = [E']$	the type of expression E must be the same as the type of expression E′
$[E] < [E']$	the type of expression E is a proper subtype of the type of expression E′
$[E] \leq [E']$	either $[E] = [E']$ or $[E] < [E']$
$[E] \equiv T$	the type of expression E is defined to be T
$[E] \leq [E1]$ or … or $[E] \leq [Ek]$	disjunction: at least one of $[E] \leq [E1]$, …, $[E] \leq [Ek]$ must hold

Anatomy of a Type Analysis Engine: Type Constraint Generation

declaration T v	[v] ≡ T
assignment E1 = E2	[E2] ≤ [E1]
access E.f to field F	[E.f] ≡ [F] [E] ≤ Decl(F)
return E in method M	[E] ≤ [M]
method M in type T	Decl(M) ≡ T
this in method M	[this] ≡ Decl(M)
direct call E.m(E1,...,En) to method M	[E.m(E1,...,En)] ≡ [M] [Ei] ≤ [Param(M,i)] [E] ≤ Decl(M)

Anatomy of a Type Analysis Engine: Generic Constraint Generation API's

```
abstract class ConstraintTerm { // a node in constraint graph
   public interface ITermProcessor {
      void processTerm(ConstraintTerm term);
   }

   public void recomputeEstimate(IEstimateEnvironment env) { }
   abstract void processTerms(ITermProcessor processor);
}

abstract class ConstraintOperator { }

class Constraint { // an edge in the constraint graph
   ConstraintTerm fLHS, fRHS; ConstraintOperator fOperator;

   Constraint(ConstraintVariable l, ConstraintOperator o,
              ConstraintVariable r) {
      fLHS = l; fRHS = r; fOperator = o;
   }
   ConstraintTerm getLHS() { return fLHS; }
   ConstraintTerm getRHS() { return fRHS; }
   ConstraintOperator getOperator() { return fOperator; }
}
```

<as presented in Part I>

Anatomy of a Type Analysis Engine: Generic Constraint Generation API's

```
class ConstraintVisitor extends ASTVisitor {// traverse AST &
                                            // generate constraints
   ConstraintCreator fCreator;
   List<Constraint> fCons = new HashSet(); // collects results

   ConstraintVisitor(ConstraintCreator cc) { fCreator = cc; }

   boolean visit(ArrayAccess access) {
      fCons.addAll(fCreator.create(access));
   }
   boolean visit(Assignment assign) {
      fCons.addAll(fCreator.create(assign));
   }
   //…
}

abstract class ConstraintCreator {
   // generate constraints for each language construct
   abstract List<Constraint> create(ArrayAccess);
   abstract List<Constraint> create(Assignment);
   abstract List<Constraint> create(ConditionalExpression);
   abstract List<Constraint> create(MethodDeclaration);
   abstract List<Constraint> create(MethodInvocation);
   //…
}
```

<as presented in Part I>

Anatomy of a Type Analysis Engine: Type Constraint Generation

```
class TypeConstraintTermFactory {
   // Responsible for "canonicalizing" terms, e.g.:
   //   Flow insensitive => all simple var refs map to same ConstraintTerm
   //   Flow sensitive   => each var ref maps to a different ConstraintTerm

   ConstraintTerm createExpressionVariable(Expression e);       // [e]
   ConstraintTerm createTypeVariable(Type t);                   // t
   ConstraintTerm createDeclaringTypeVariable(IBinding b);      // Decl[b]
   ConstraintTerm createParamVariable(IMethodBinding m,int i); //[Param(m,i)]
   ConstraintTerm createReturnVariable(IMethodBinding m);       // [m]
   //…
}

// General Principle: Save just enough info to locate corresponding AST node
class ParameterVariable extends ConstraintTerm {
   ICompilationUnit fCU; String fMethodKey; int fParamIdx;
   ParameterVariable(IMethodBinding method, int idx, ICompilationUnit cu) {
      fCU= cu;
      fMethodKey= method.getKey(); // DON'T HANG ONTO BINDING!
      fParamIdx= idx;
   }
}
class ReturnVariable extends ConstraintTerm {
   ICompilationUnit fCU; String fMethodKey;
   ParameterVariable(IMethodBinding method, ICompilationUnit cu) {
      fCU= cu;
      fMethodKey= method.getKey(); // DON'T HANG ONTO BINDING!
   }
}
```

Anatomy of a Type Analysis Engine: Type Constraint Generation

```
class TypeConstraintTermFactory implements ConstraintTermFactory {
   Map<Object, ConstraintTerm> fCTMap;

   ConstraintTerm createExpressionVariable(Expression e) {
      Object key;
      switch(e.getNodeType()) {
         case ASTNode.NAME:
         case ASTNode.FIELD_ACCESS:
            key = e.resolveBinding(); // Flow insensitive: all refs map
            break;                    // to the same ConstraintTerm
         default:
            key = new CompilationUnitRange(e);
            break;
      }
      ConstraintTerm t = fCTMap.get(key);
      if (t == null)
         fCTMap.put(key, t = new ExpressionVariable(e));
      return t;
   }
   //… similar methods for creating other ConstraintTerm types…
}
class TypeOperator extends ConstraintOperator {
   private TypeOperator() { }

   static final TypeOperator Subtype   = new TypeOperator();
   static final TypeOperator Supertype = new TypeOperator();
   static final TypeOperator ProperSubtype   = new TypeOperator();
   static final TypeOperator ProperSupertype = new TypeOperator();
   static final TypeOperator Equals = new TypeOperator();
}
```

Anatomy of a Type Analysis Engine: Type Constraint Generation

```
class TypeConstraintCreator { // gen constraints for each language construct
   ConstraintTermFactory fFactory;

   List<Constraint> create(Assignment a) { // [rhs] <= [lhs]
      return new Constraint(fFactory.createExpressionVariable(a.getRHS()),
                            TypeOperator.Subtype,
                            fFactory.createExpressionVariable(a.getLHS()));
   }

   List<Constraint> create(MethodInvocation inv) {
      List<Constraint> result = new List<Constraint>();
      IMethodBinding   method = inv.resolveBinding();
      ITypeBinding     methodOwner = method.getDeclaringType();
      List<Expression> args   = method.getArguments();

      // [rcvr] <= Decl[method]
      result.add(new Constraint(fFactory.createExprVariable(inv.getReceiver()),
                                TypeOperator.Subtype,
                                fFactory.createDeclTypeVariable(methodOwner));
      // [arg #i] <= [Param(method, i)]
      for(int i=0; i < args.size(); i++)
        result.add(new Constraint(fFactory.createExpressionVariable(args.get(i)),
                                TypeOperator.Subtype,
                                fFactory.createParmVariable(method, i)));
      return result;
   }
   List<Constraint> create(MethodDeclaration d) { /* preserve override, etc. */ }
   //…
}
```

Anatomy of a Type Analysis Engine: Constraint Solution

Build Constraint Graph → Initialize Type Estimates → Process Work-List

```
class ConstraintGraph {
   List<Constraint>    fConstraints;
   Set<ConstraintTerm> fAllNodes;
   Map<ConstraintTerm,List<Constraint>> fEdgeMap;

   class TermDecorator implements ITermProcessor {
      Constraint fConstraint;
      void setConstraint(Constraint c) {fConstraint=c;}
      public void processTerm(ConstraintTerm term) {
         addToEdgeList(term, fConstraint);
         fAllNodes.add(term);
      }
   }
   void initialize() { // build graph from Constraints
      TermDecorator decorator = new TermDecorator();
      for(Constraint c: fConstraints) {
         ConstraintTerm lhs = c.getLeft();
         ConstraintTerm rhs = c.getRight();

         decorator.setConstraint(c);
         lhs.processTerms(decorator);
         rhs.processTerms(decorator);
      }
   }
}
```

<as presented in Part I>

Anatomy of a Type Analysis Engine: Constraint Solution

Build Constraint Graph

Initialize Type Estimates

Process Work-List

```
class ConstraintSolver {
   void initializeTypeEstimates() {
      for(ConstraintTerm t: graph.getNodes()) {
         if (t instanceof ExpressionVariable) {
            if (t is a ctor call, literal, or cast)
               setEstimate(t, t.getDeclaredType());
            else
               setEstimate(t, TypeUniverse.instance());
         } else if (t.isConstantType()) {
            setEstimate(t, t.getDeclaredType());
         } else if (t.isBinaryMember()) {
            // don't report OSV smells on binary class
            setEstimate(t, t.getDeclaredType());
         } else
            setEstimate(t, TypeUniverse.instance());
      }
   }
}
```

Anatomy of a Type Analysis Engine: Constraint Solution

Build Constraint Graph
↓
Initialize Type Estimates
↓
Process Work-List

```
class ConstraintSolver {
   void solveConstraints() {
      while (!workList.empty()) {
         ConstraintTerm t = workList.pop();
         for(c: getConstraintsInvolving(t)) {
            lhs = c.getLHS();
            rhs = c.getRHS();
            if (c.getOperator().isSubtype())
               enforceSubtype(lhs, rhs);
            else if (c.getOperator().isEquals())
               unify(lhs, rhs);
         }
      }
   }
   void enforceSubtype(ConstraintTerm l,ConstraintTerm r){
      lhsEst   = getEstimate(lhs);
      rhsEst   = getEstimate(rhs);
      lhsSuper = lhsEst.superTypes();
      rhsSub   = rhsEst.subTypes();
      if (!rhsSub.containsAll(lhsEst))
         setEstimate(rhs, lhsEst.xsectWith(rhsSub));
      if (!lhsSuper.contains(rhsEst))
         setEstimate(lhs, rhsEst.xsectWith(lhsSuper);
   }
}
```

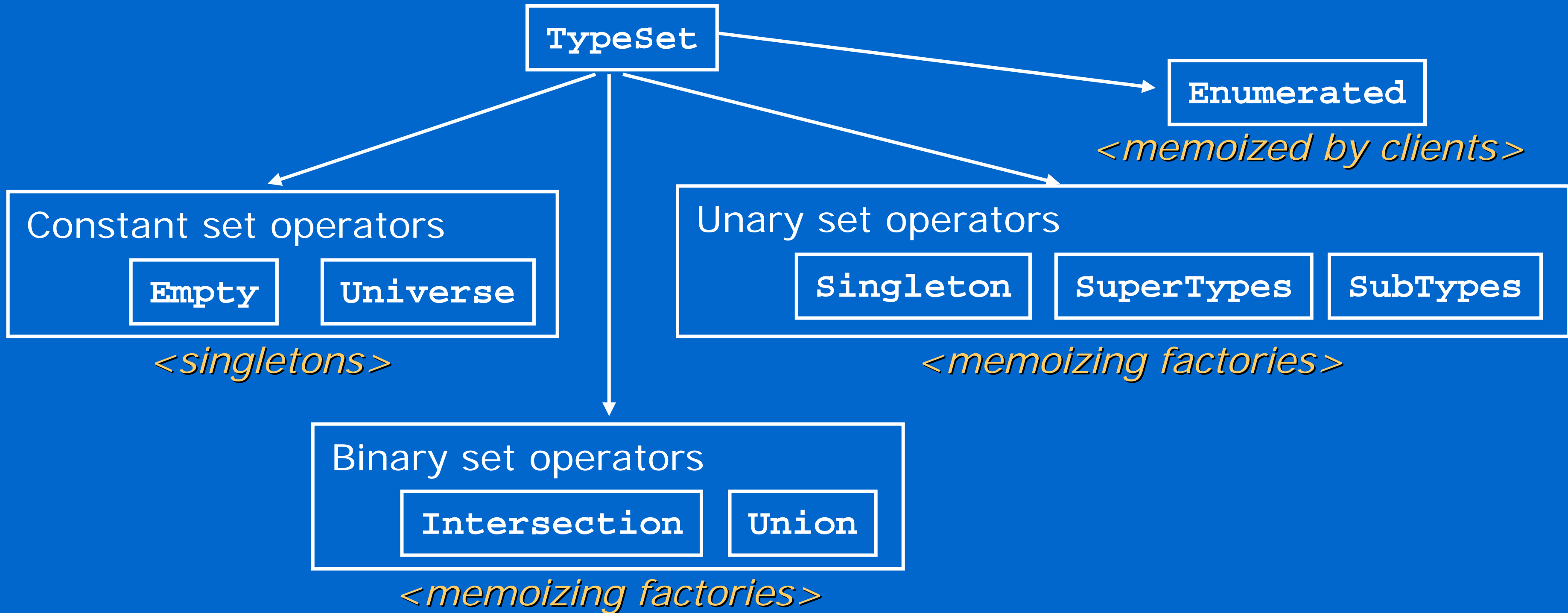
$lhs \leq rhs$

sets monotonically decrease in size!

Anatomy of a Type Analysis Engine: Type Sets

```
abstract class TypeSet { // an immutable "value class" - set of JDT TType's
    // These operations execute in constant time wherever possible
    boolean isEmpty();
    boolean isSingleton();
    boolean isUniverse();

    TType anyMember();

    contains(TType);
    containsAll(TypeSet);

    Iterator<TType>   iterator();  // avoid this as much as possible
    EnumeratedTypeSet enumerate(); // avoid this as much as possible

    // These operations perform algebraic simplifications where possible
    TypeSet subTypes();
    TypeSet superTypes();
    TypeSet intersectedWith(TypeSet);
    TypeSet unionWith(TypeSet);

    TypeSet lowerBound();
    TypeSet upperBound();

    boolean hasUniqueLowerBound();
    TType   uniqueLowerBound();
    boolean hasUniqueUpperBound();
    TType   uniqueUpperBound();
}
```

Anatomy of a Type Analysis Engine: Type Sets



Algebraic simplifications:

```
subTypes(subTypes(S)) = subTypes(S)
subTypes(universe)    = universe
subTypes(java.lang.Object) = universe
lowerBound(superTypes(S))  = S
```

Smell Detection: Adding Markers

```
IMarker createMarker(ICompilationUnit icu, String message,
                     int lineNum, int offset, int length) {
   IResource srcFile = icu.getResource();

   // type ID distinguishes this marker as a smell marker
   IMarker m = srcFile.createMarker("org.pldi2005.smell");

   m.setAttribute(SEVERITY, SEVERITY_INFO);
   m.setAttribute(MESSAGE, message);
   m.setAttribute(LINE_NUMBER, lineNum);
   m.setAttribute(CHAR_START, offset);
   m.setAttribute(CHAR_END, offset + length);

   // client may set additional attributes, e.g. "SMELL_KIND"
   // and "NEW_TYPE" (shown earlier)

   return m;
}
```

Smell Remediation: Quick Fix

```
class OverlySpecificResolutionGenerator
      extends ResolutionGeneratorBase
{
   public IMarkerResolution[] getResolutions(IMarker m) {
      // Examine “SMELL_KIND” attribute of marker to determine
      // whether it’s one of the smells this resolution generator
      // can remediate.
      if (!matchesSmellMarkerKind(“overlySpecific”))
         return new IMarkerResolution[0];

      IMarkerResolution resolution = new OverlySpecificResolution();

      return new IMarkerResolution[] { resolution };
   }
}
```

Smell Remediation: Quick Fix

```
abstract class MarkerResolutionBase implements IMarkerResolution {

   ASTNode findASTNodeForMarker(IMarker m, CompilationUnit unit) {
      int pos = ((Integer) m.getAttribute(CHAR_START)).intValue();
      int len = ((Integer) m.getAttribute(CHAR_END)).intValue();

      return NodeFinder.perform(unit, pos, len);
   }

   void performRewrite(IFile file, ASTRewrite rewriter) {
      // Get an IDocument on the given file, and apply the rewriter to that
      ITextFileBufferManager bufMgr = FileBuffers.getTextFileBufferManager();
      ITextFileBuffer fileBuf = bufMgr.getTextFileBuffer(file.getLocation());

      IDocument doc  = fileBuf.getDocument();
      TextEdit  edit = rewriter.rewriteAST(doc, null);

      edit.apply(doc);
   }

   ICompilationUnit getCUForFile(IFile file) {
      return (ICompilationUnit) JavaCore.create(file);
   }

   CompilationUnit createASTForICU(ICompilationUnit icu) {
      /* Use ASTParser as shown earlier */
   }
}
```

Smell Remediation: Quick Fix

```
class OverlySpecificResolution extends MarkerResolutionBase {
   public String getLabel() {
      return "Make type as general as possible";
   }
   public void run(IMarker m) {
      // Find the CU and parse it into an AST
      IFile                file    = (IFile) m.getResource();
      ICompilationUnit icu     = getCUForFile(file);
      CompilationUnit  astUnit = createASTForICU(icu); // ASTParser

      // Find the node to rewrite
      ASTNode    typeNode = findASTNodeForMarker(m);
      ASTRewrite rewriter = ASTRewrite.create(typeNode.getAST());

      // Create the replacement ASTNode using the qualified name
      // stored in the marker
      String newTypeStr  = (String) m.getAttribute(NEW_TYPE);
      Name   newTypeName = ASTNodeFactory.newName(ast, newTypeStr);
      Type   newTypeNode = ast.newSimpleType(newTypeName);

      // Do the rewrite
      rewriter.replace(typeNode, newTypeNode);
      performRewrite(file, rewriter);
   }
}
```

The End

- That's all, folks!