EclipseSource

Eclipse, OSGi and API Tooling

Chris Aniszczyk (EclipseSource)
zx@eclipsesource.com
http://eclipsesource.com

Who am I?

- Chris Aniszczyk
  - Senior Software Engineer at EclipseSource
  - Plug-in Development Environment (PDE) Co-Lead
  - Been doing OSGi before it was en vogue
  - Committer Representative on Eclipse's Board of Directors
  - Co-author Eclipse RCP Book 2nd Edition
  - I dabble with many things at Eclipse.org
- My blog
  - http://aniszczyk.org
- Follow me on Twitter
  - http://twitter.com/caniszczyk

Overview

- The need for tooling
- Tooling features
- Tooling in action!
- Future work
- Summary
- Q&A

The need for tooling

Remember Kirk's gigantic Modularity diagram?

Define API

- APIs are published contracts
  - Producers specify the contracts
    - Honor contracts release after release
    - Evolve the contracts for enhancements and fix problems
  - Consumers adhere to the contracts
    - Implement the contracts per specification
    - Use provided implementations per contract specifications

The Ideal Release Equation

Developer: ***Compatibility*** (producer)

+

Client: ***Honor API contract*** (consumer)

=

Free migration!!!

The Ideal Never Happens...

Developer: (***features*** + ***deprecation*** + ***Optional extensions*** to existing features + ***Replacement API*** + ***Provisional API***)

+

Client: **Illegal internal references**

=

Migration at a cost

What have we done as developers?

- We use Javadoc™ to document contracts
  - This class is not intended to be instantiated or subclassed by clients.
  - Has no effect if not read
  - Inconsistent wording, placement, and use
- Use `component.xml` to specify APIs (at Eclipse.org)
  - Out of date, not maintained
  - Why? Because there's no tooling and it's separate from the code
- We use package names to categorize as public/internal, but all packages are exported
- We use OSGi package exports (`x-internal` and `x-friends`) to limit visibility, but does not prevent access to internal types
- We use Eclipse's **access rules** to discourage use, but this can be turned off

More things we do…

- Manually examine API changes or use some external tool before a release (aka the API Police)
  - Changes between releases can be significant
  - Validation is time consuming and error prone
  - Lost development time (due to checking and bugs found)

And we still fail...

And we still fail...

- At Eclipse, we have had issues with projects unintentionally breaking API...
- Outside of Eclipse, it highly varies...
  - “Commons collections 3.0 is binary compatible with version 2.1 and 2.0 except for certain methods on one class. As the release was a major version, this is permitted, **however it was unintentional and an error**... the chosen solution is to provide a work around by releasing v2.1.1 and v3.1.”
- Surely you’ve come across API issues, right?

Disclaimer

- This presentation isn't about API design

- API design is still **your problem**
  - Designing quality APIs is tough
  - Ensuring consistent behavior
  - Evolving APIs is tricky

- But there are things that tooling can **help** with…

Good fences...

- Check out some of Joshua Bloch's presentations around API (How to Design API and Why it Matters) and some of the EclipseCon API design presentations...

API Tooling Features

API Tooling to the Rescue!

- Reports
  - Illegal API use
  - Binary incompatibility relative to a baseline
  - Incorrect bundle version numbers
  - Missing or malformed @since tags
  - Leakage of non-APIs types inside APIs
  - Execution Environment validation
- Tightly integrated toolset in the Eclipse SDK
  - Currently limited to Plug-in projects/OSGi bundles
  - Runs as a builder (auto-build, incremental and full builds)
  - Immediate feedback as you develop and use APIs

Specifying API Contracts

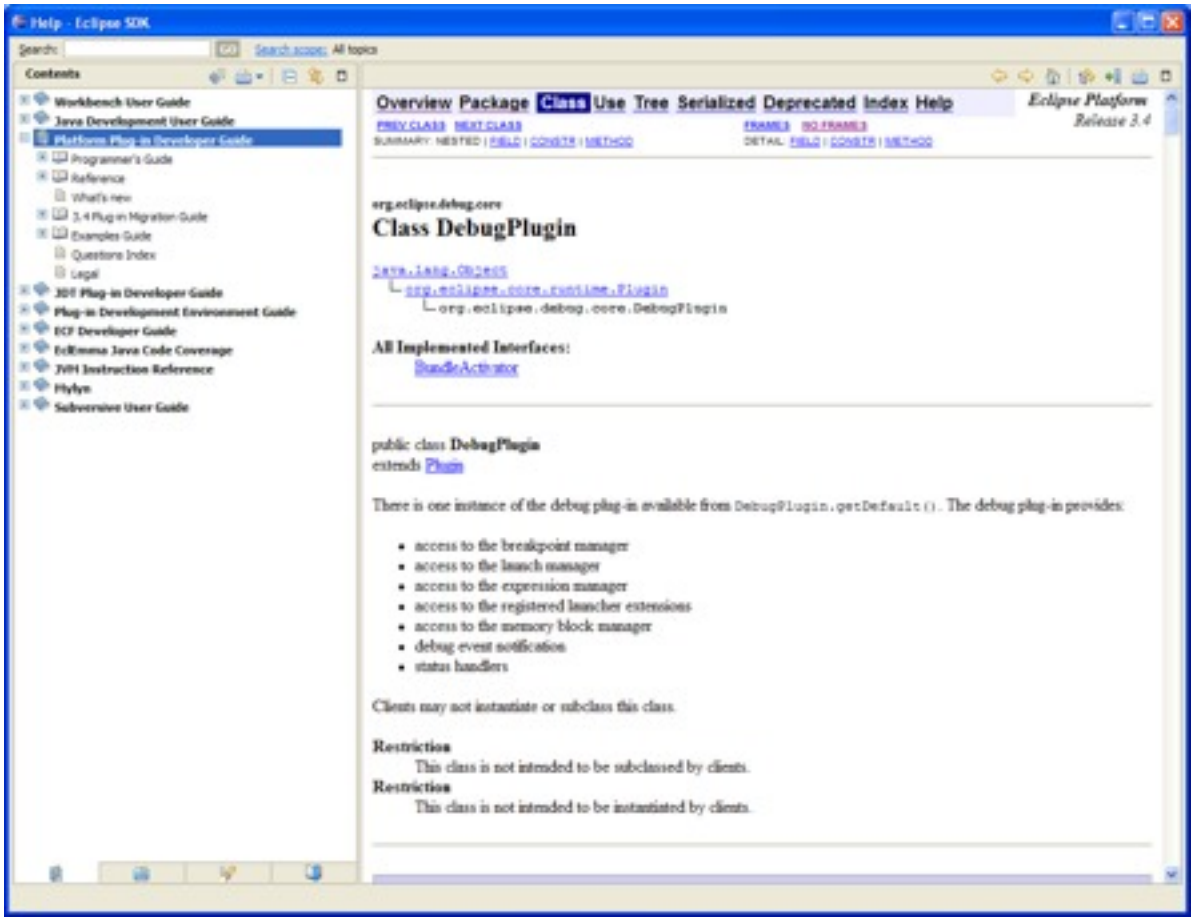
- Use Javadoc tags
  - E.g. *@noimplement, @noextend, @noreference, @noinstantiate*
- Benefits
  - Contracts live with the code for producers and consumers
  - Content assist helps developers
  - Available for projects that are not using 1.5 annotations
  - Restrictions appear in published Javadoc APIs in a standard way
  - Tools can process tags

API Tooling Javadoc Tags

Supported Restriction Tags

	Class	Interface	Method	Constructor	Final Field	Non-Final Field
@noimplement	-	✔	-	-	-	-
@noextend	✔	✔	-	-	-	-
@noinstantiate	✔	-	-	-	-	-
@nooverride	-	-	✔	-	-	-
@noreference	-	-	✔	✔	-	✔

Example of Published API



API Usage

- Once the APIs are specified, the user needs to make sure that he/she is using them appropriately.
- API usage is flagging any kind of **illegal usage** of API: reference to an API that is not supposed to be referenced, implementation of an interface that is not supposed to be implemented, ….
- A consequence of wrong API usage is a potential binary incompatible change error.

Why API Usage Scans?

API producers can see what internals are being used

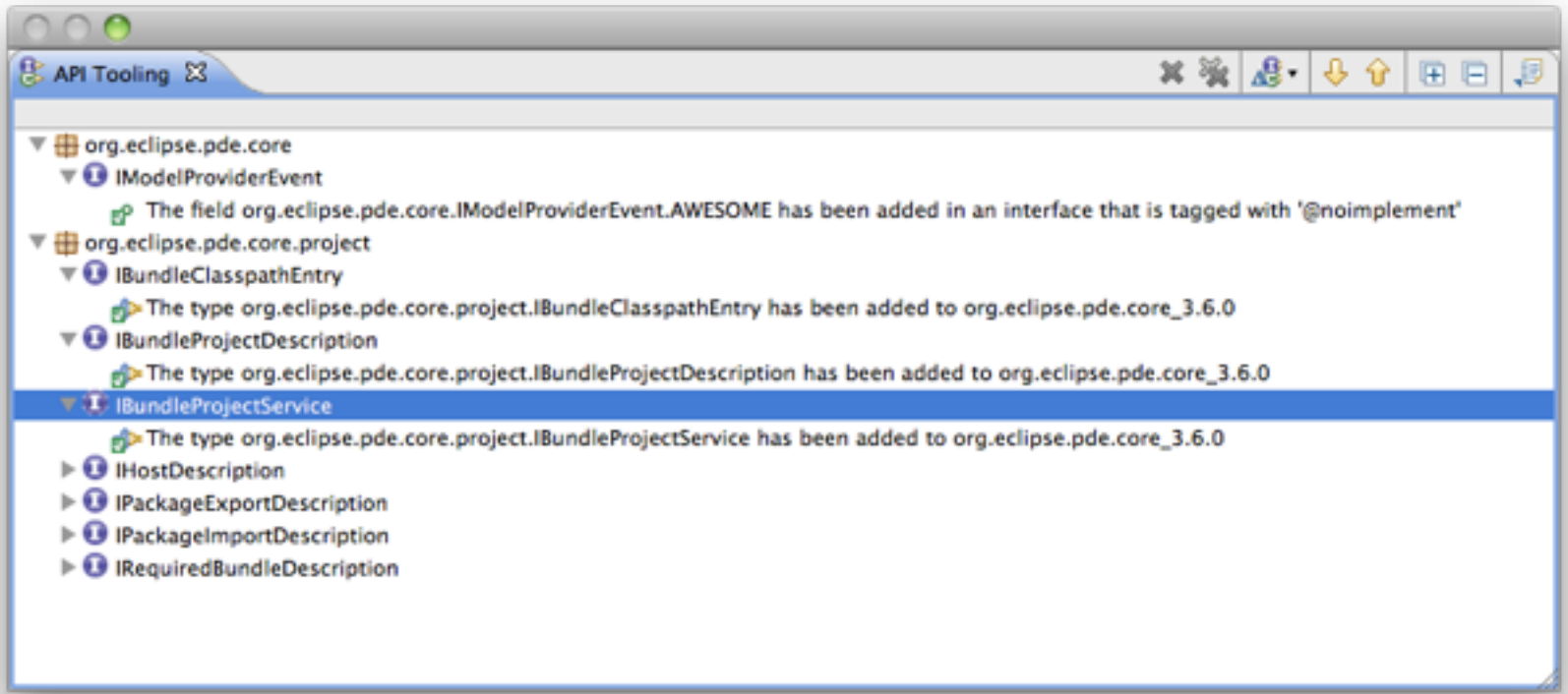
- Inform clients how to use proper API
- Determine what new API needs to be added
- Know what clients will be broken
- Avoid breaking internals until clients have migrated

Validating Binary Compatibility

- Evolving APIs such that they are backwards compatible with existing binaries
  - http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs
  - It is easy to get it wrong
  - Now the tooling takes care of this
- The user simply specifies an API baseline
  - Generally this means pointing to the previous release (N – 1)

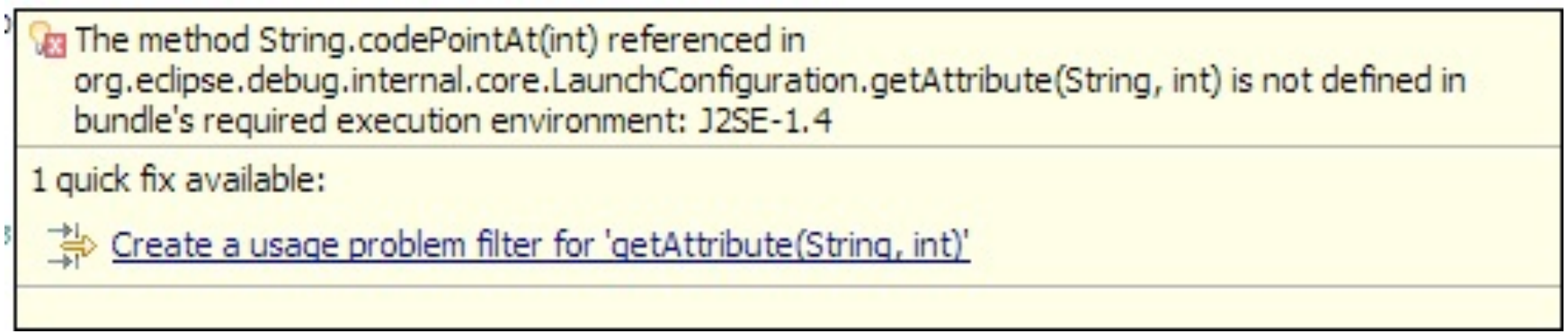
API Comparisons

- What's changed between versions?



System Library Validation

- Validate system library use based on a bundle's required execution environment (BREE)
  - if a bundle claims to run on J2SE-1.4, access to members in J2SE-1.5 and higher are flagged as illegal
  - See OSGi Specification on Execution Environments



Migration Assistance

- *"What will be broken if I move to a new version?"*
- *"Will my plug-in still run on an older target version?"*
- We simply re-resolve the references against X, with the code in X+1
- Of course you can get the same information if you compile, but that requires more setup...

Bundle version number management

- http://wiki.eclipse.org/index.php/Version_Numbering
- The tooling takes care of letting the user know when the minor or major version of a bundle should be changed according to rules described in the document:
  - A new API that is not a breaking change requires the minor version to be incremented
  - A new API that is a breaking change requires the major version to be incremented
- No support for the micro version yet...
  - Technically speaking, this version should be changed as soon as any modification is made in the source code: comment change, method body change,…

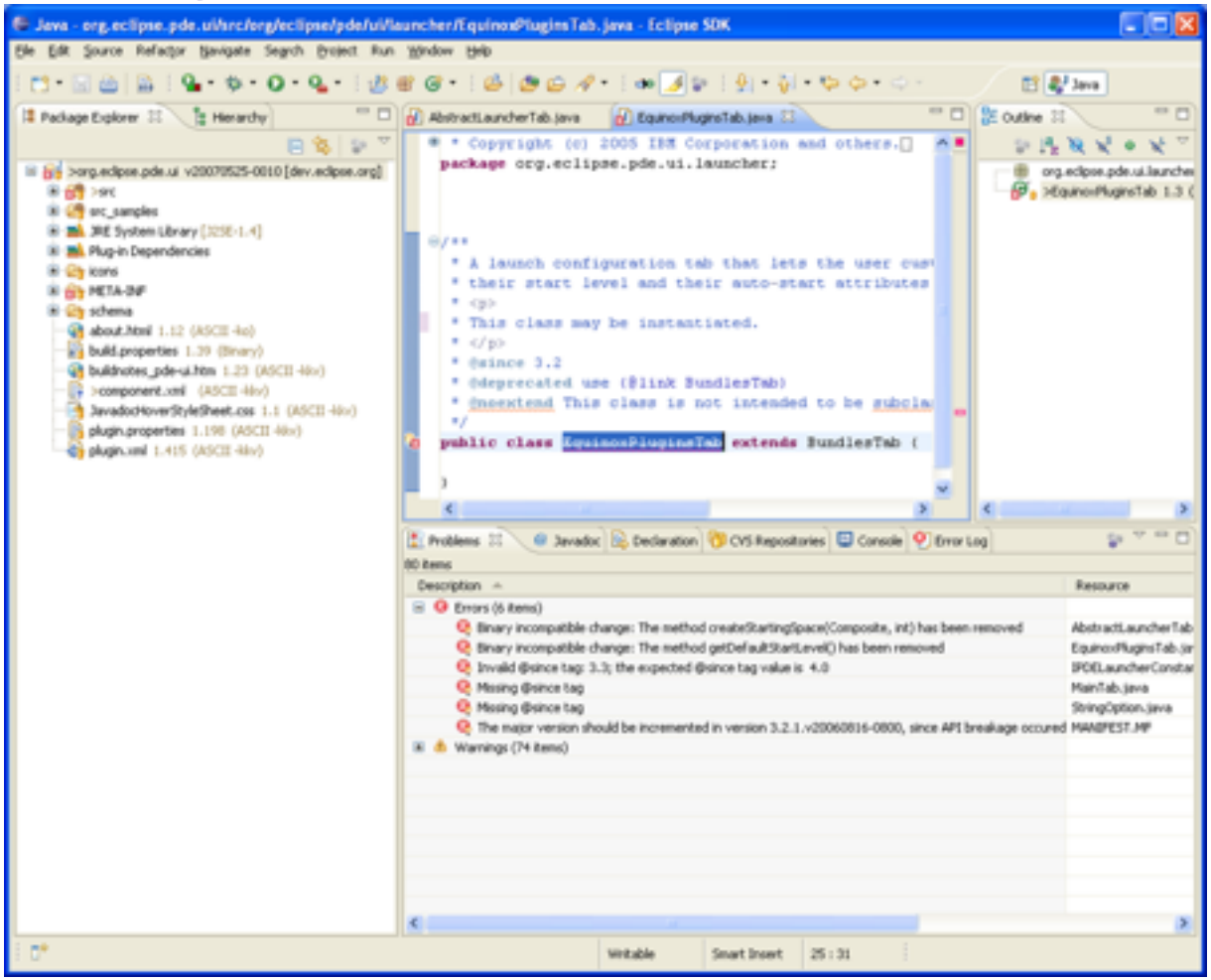
OSGi Versioning 101

- major.minor.micro.qualifier
  - major - An incompatible update; breaking API
  - minor - A backward compatible update; API stable
  - micro - A bug fix
  - qualifier - build date; lexicographic

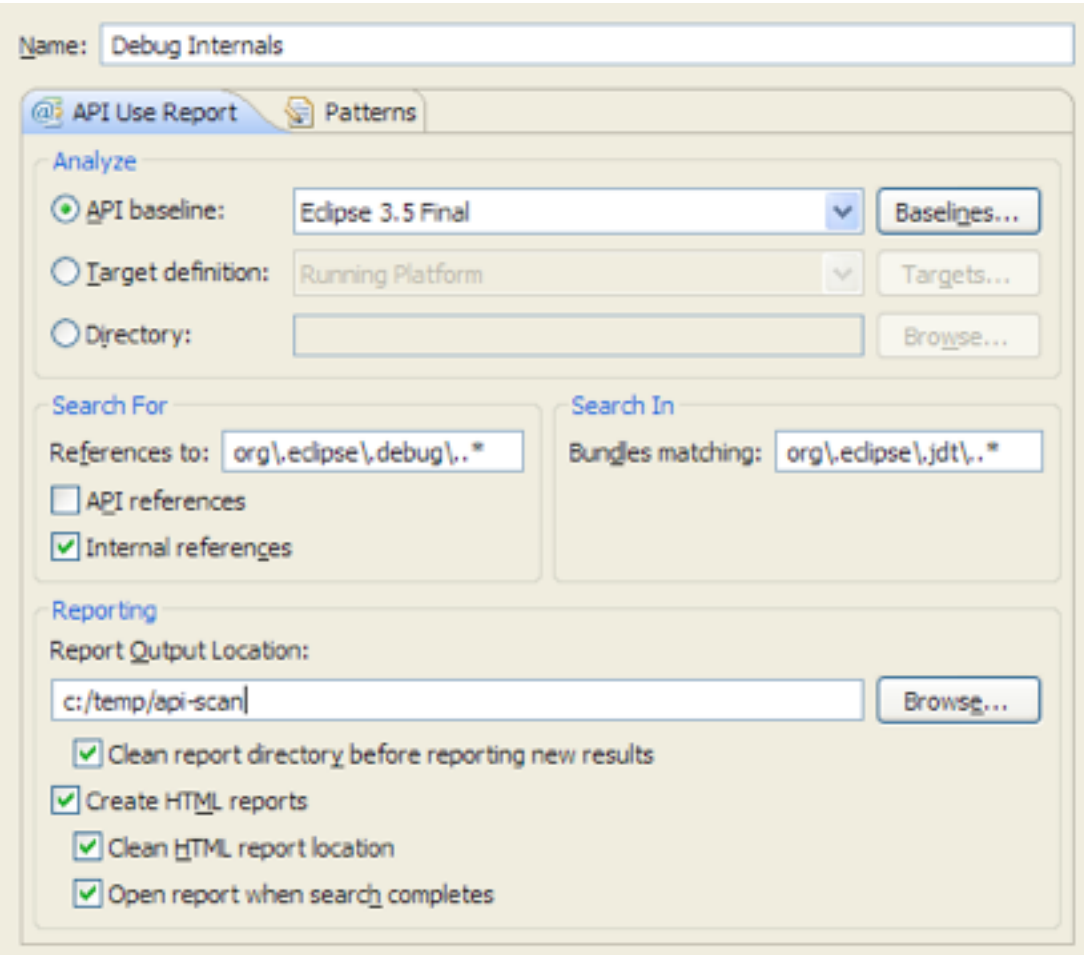
- Versions should encode compatibility at the bundle and package level... aid in evolution...

- They aren't a **marketing number**!

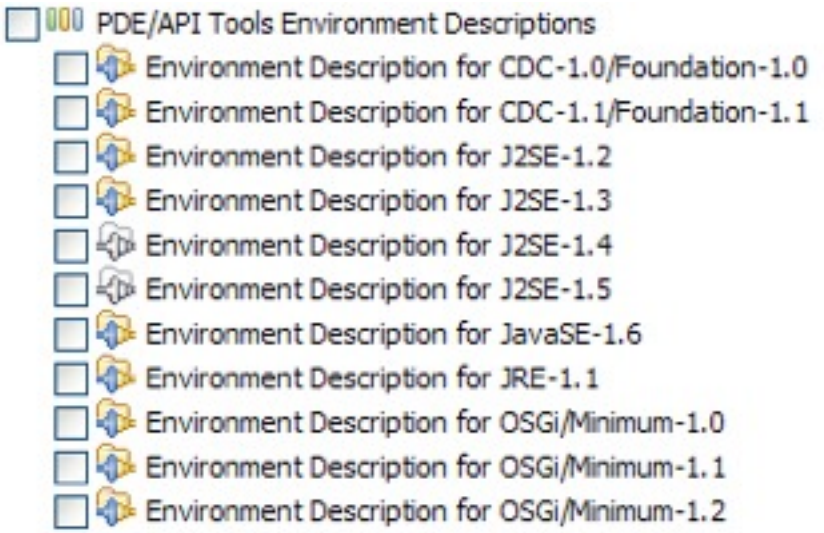
API Tooling in Action (example of bug 191231/191232)

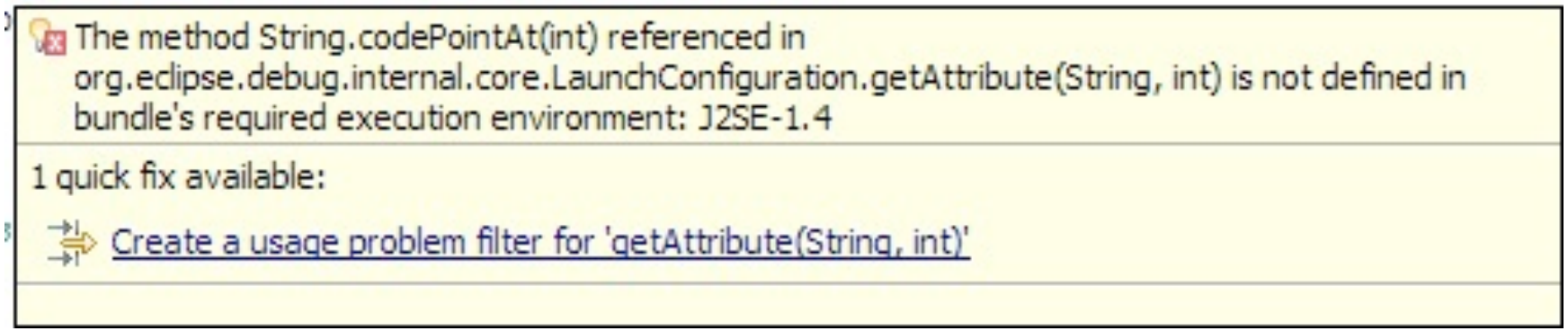


API Tooling in Action (API Usage Report)

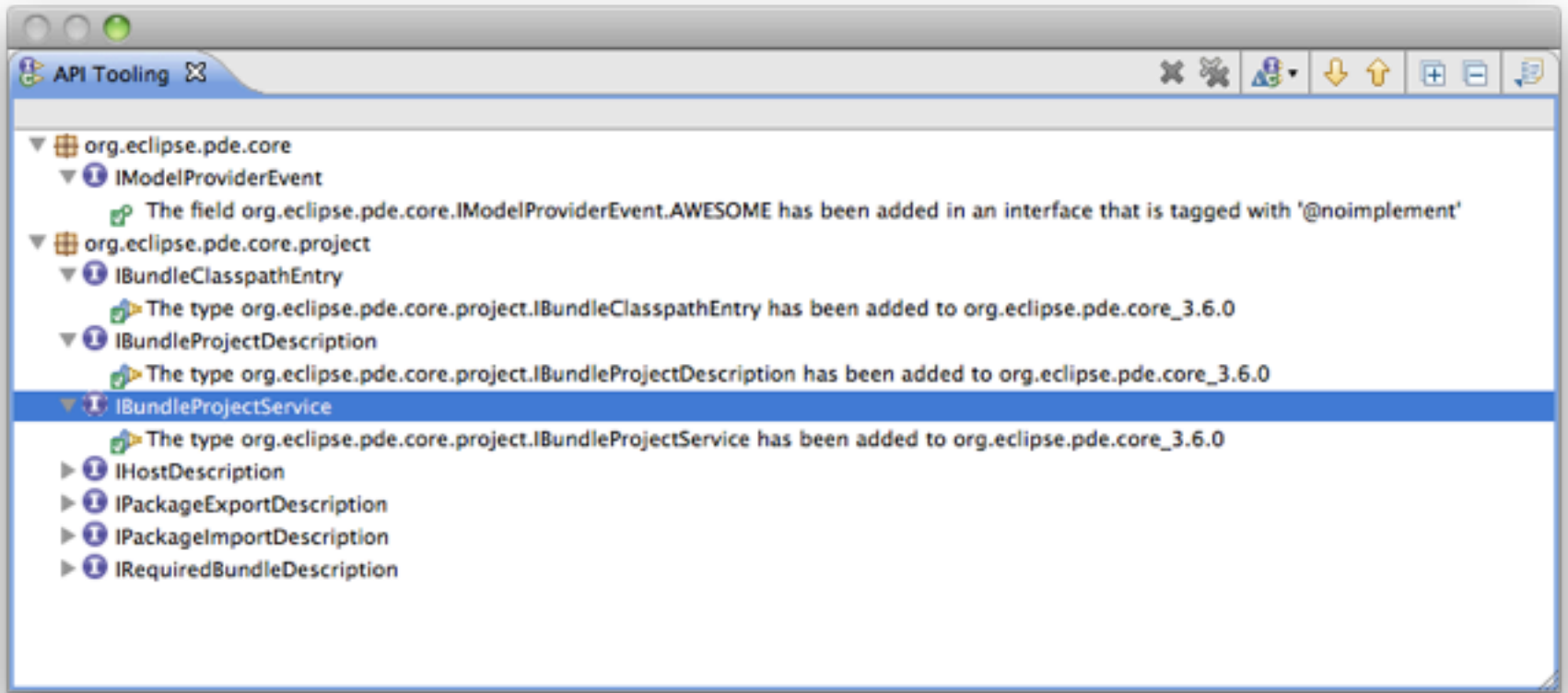


API Tooling in Action (System Library Validation)





API Tooling in Action (API Comparison)



API Tooling Parts

API Profile and API Components

- These are the two major pieces used by the API tooling tools to make diagnosis on the code
- An API profile can be seen like a PDE target platform
- An API profile is a collection of API components
- It is initialized using an Eclipse SDK installation plugins directory (or some other directory of bundles)
- Inside the IDE, a profile corresponding to the workbench contents is automatically created

API Description

- This contains the specific restrictions for all the API types of an API component
- It is kept up-to-date inside the workbench by using resource listeners
- It is exported inside the binary bundles or generated at build time using an ant task.

Filtering of API Problems

- Each component can define a problem filter
- The filtering can be used to remove from reports known breakages.
- For example, an API breakage has been approved by the PMC and you don't want to get it reported for each build.
- The problem filter is also part of the binary plug-in

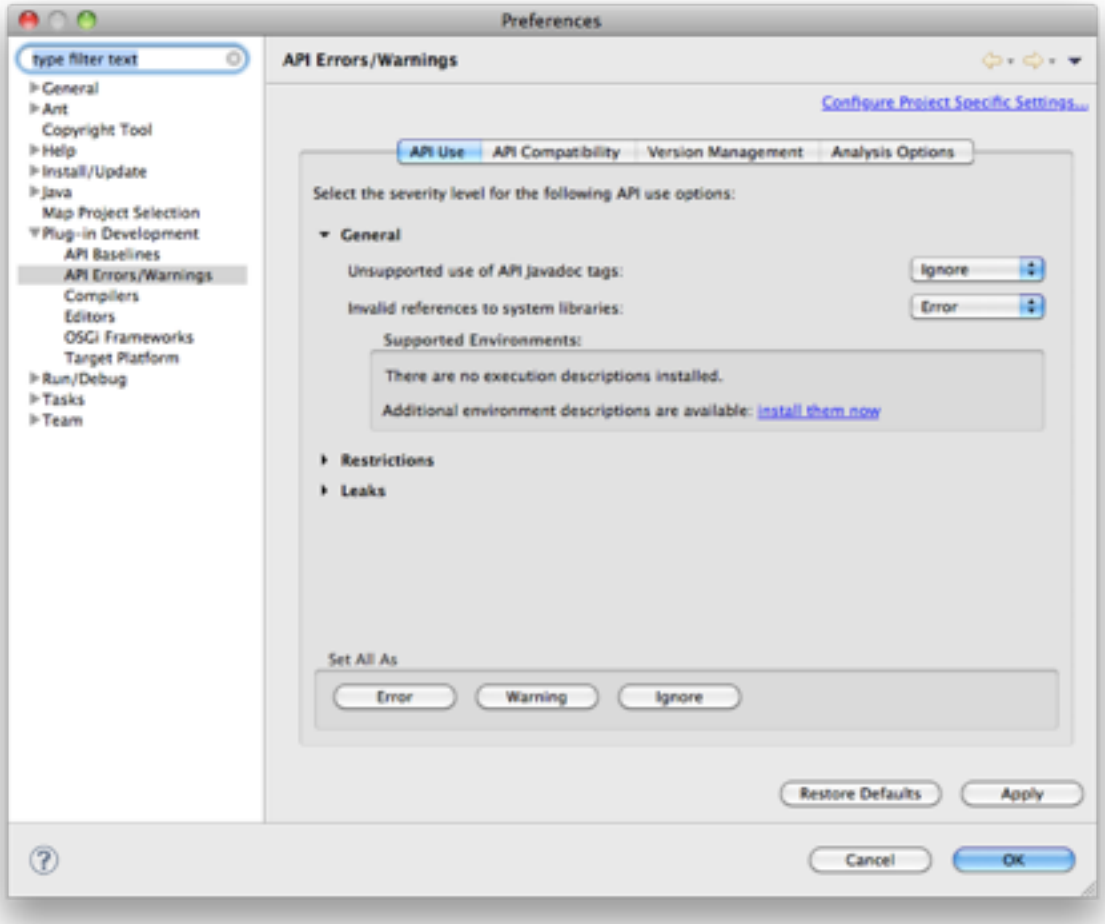
Two aspects: IDE and build process

- Each feature is available from within the IDE or inside a headless build process via Ant tasks
- The IDE support is required to help the Eclipse developer while the code is written
- The build process support is required to provide feedback during the Eclipse build. This also allows other projects to use it inside their builds.

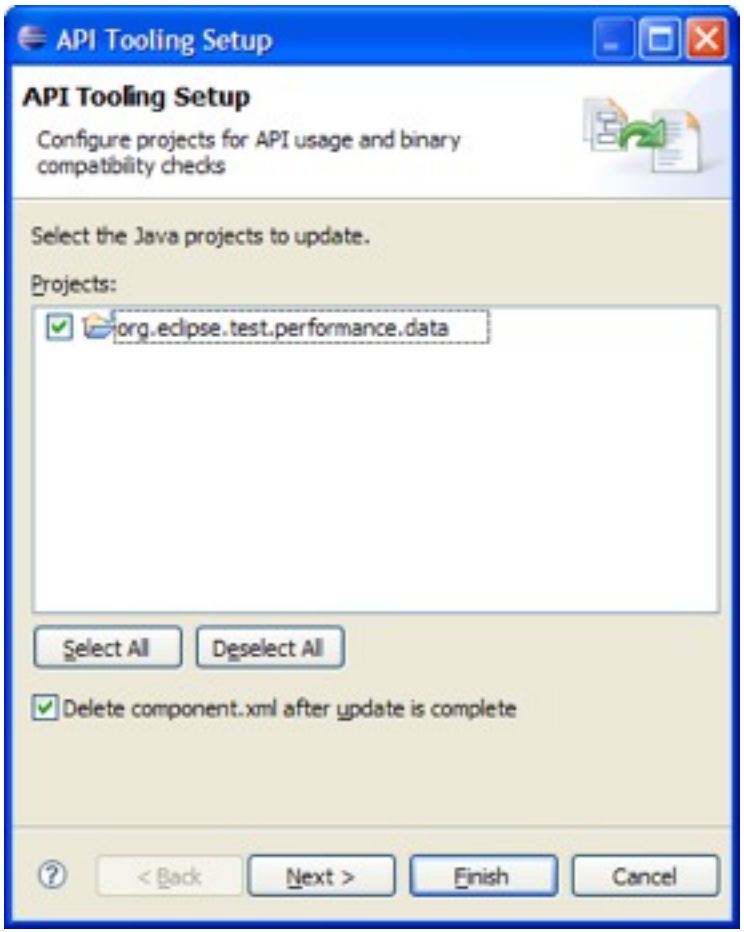
Build support

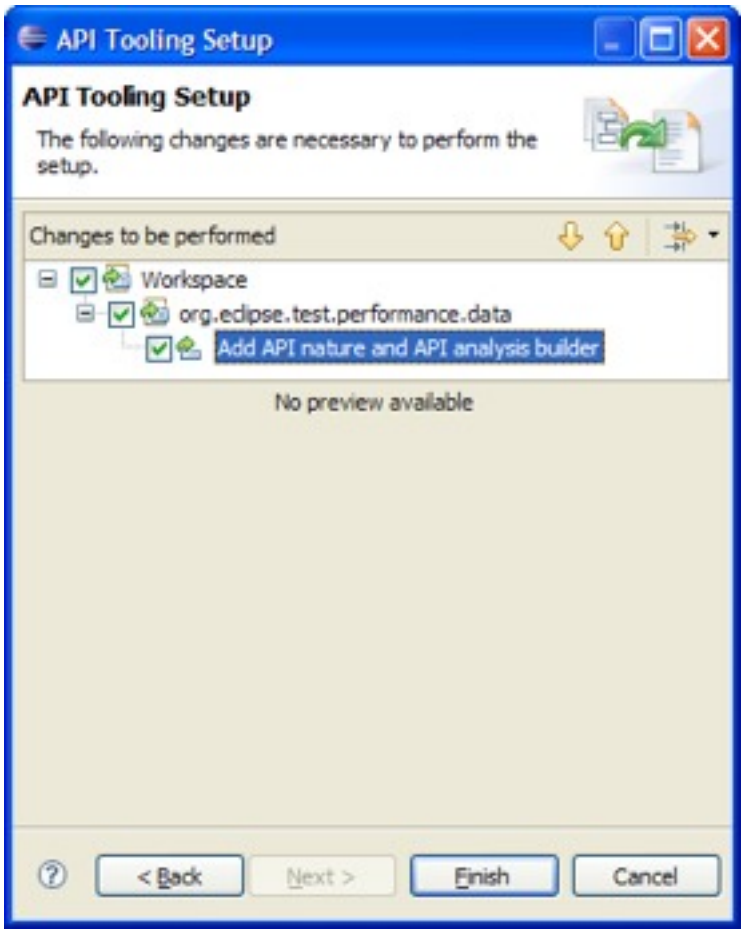
- Addition of new ant tasks:
  - Generation of .api_description file
  - Comparison of SDK drops: binary compatibility, api usage reports
- Integration inside the Eclipse builds (headless mode)
- Integration inside ant build (no Eclipse running)

API Error/Warning Preferences

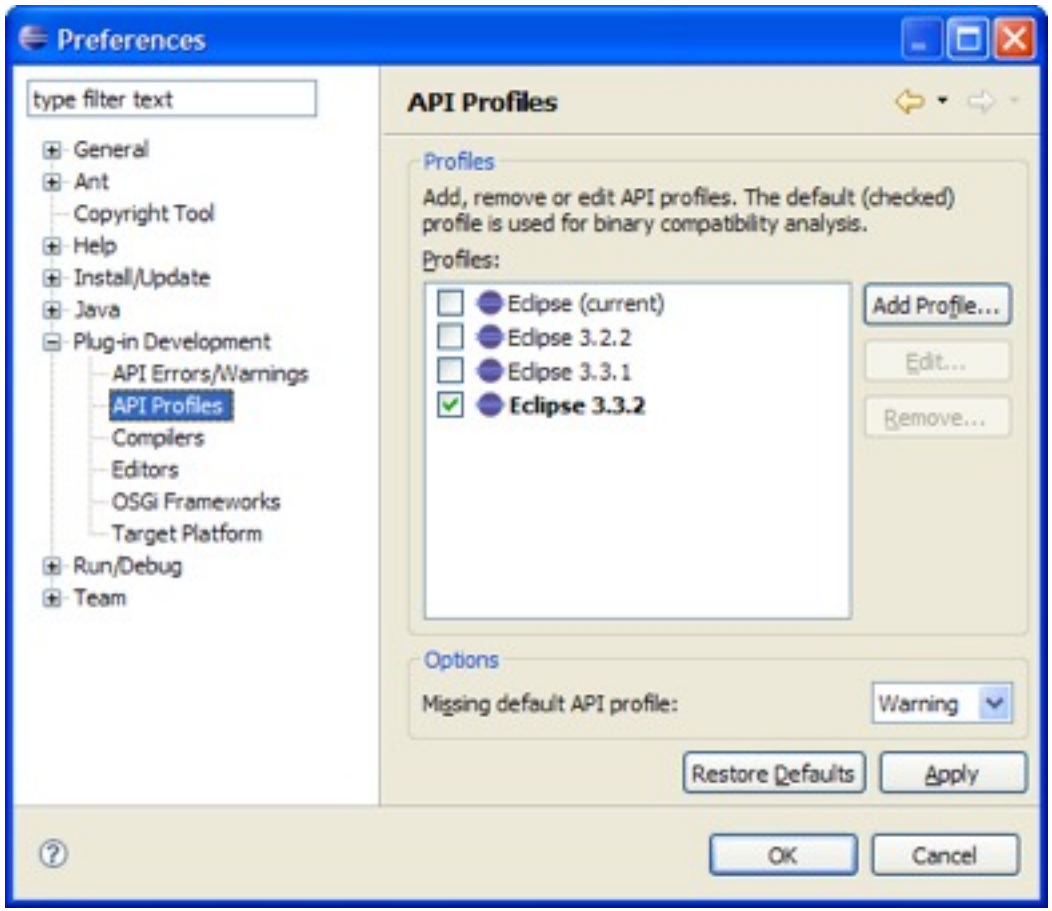


API Setup Wizard

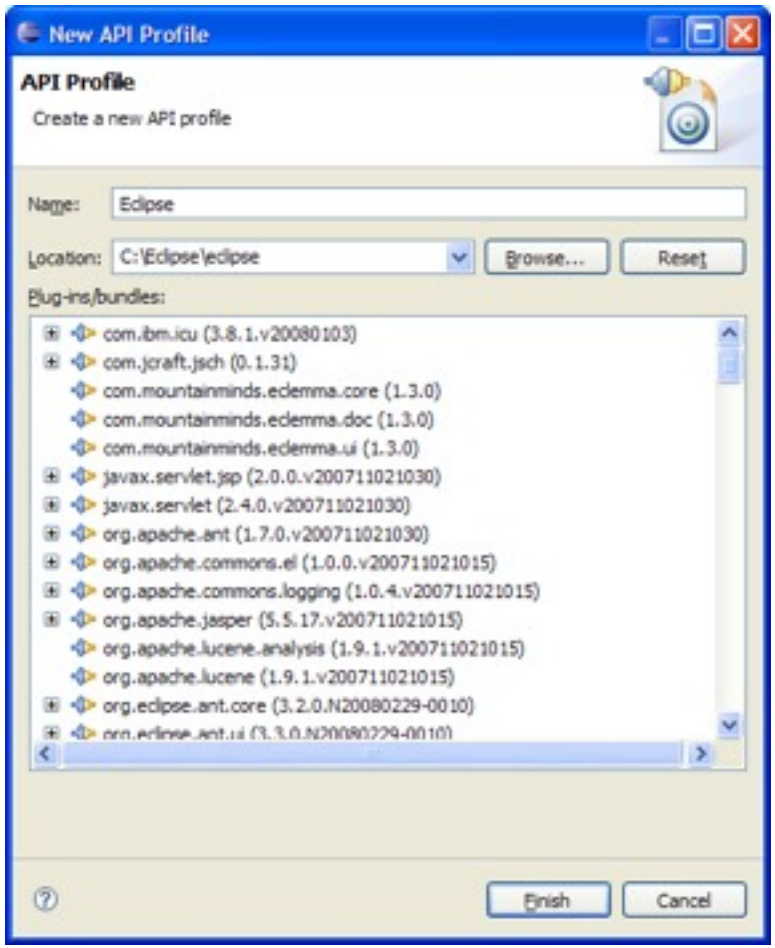




API Profile (Baseline) Preferences



API Profile (Baseline) Wizard



Future work

To be done...

- Handling of package level versioning
- Support extended to support more than just bundles
  - Pure Java™ projects
  - Plug-in extension points
- Improve integration with Eclipse rel-eng build reporting
- Determine compatible version range of required bundles

- And what you might suggest...

Summary

API Tooling Summary

- Help you to define your API restrictions
- Keep a consistent and standard presentation of API restrictions
- Detect binary breakage between a baseline and the current version
- Detect wrong API usage
- Detect wrong @since tags and inconsistent bundle versioning
- Detect Execution Environment problems...
- **Supports more than just Eclipse (hi OSGi folks)!**

Links

- Wiki
  - http://wiki.eclipse.org/Api_Tooling
- Bugzilla
  - https://bugs.eclipse.org/bugs/enter_bug.cgi?product=PDE
- Get Involved
  - https://dev.eclipse.org/mailman/listinfo/pde-dev

Q & A