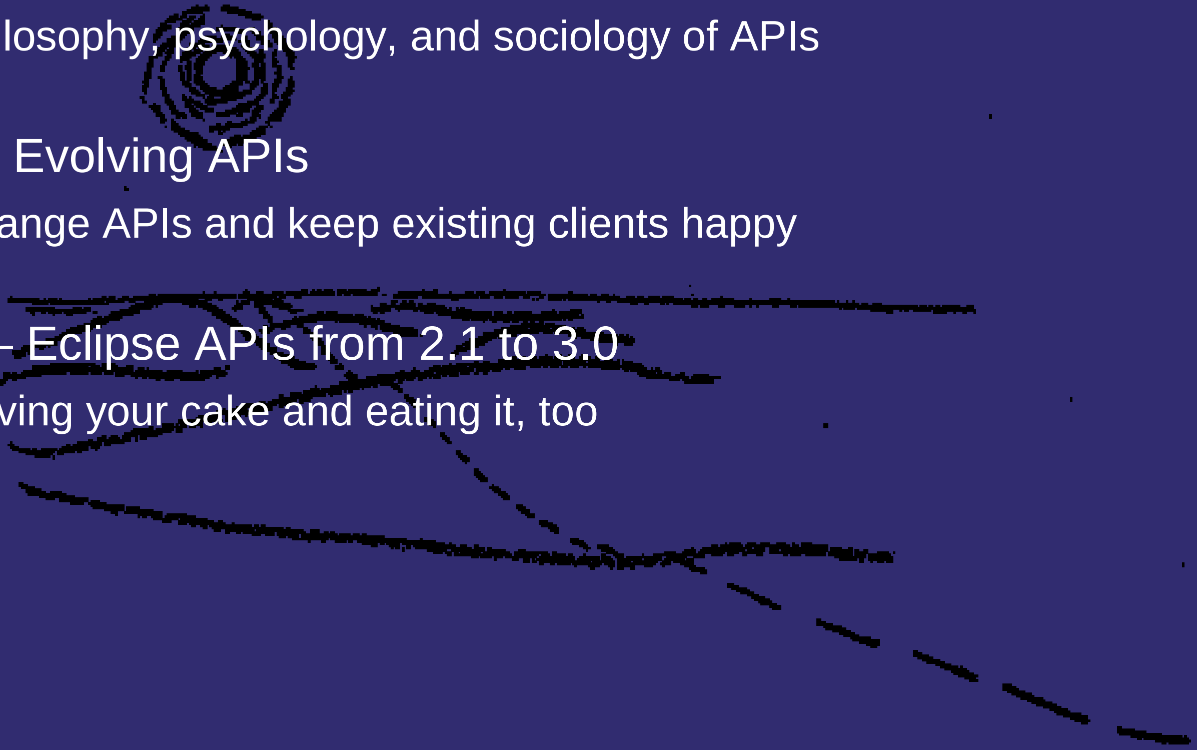
eclipseCON 2004

Eclipse APIs
Lines in the Sand

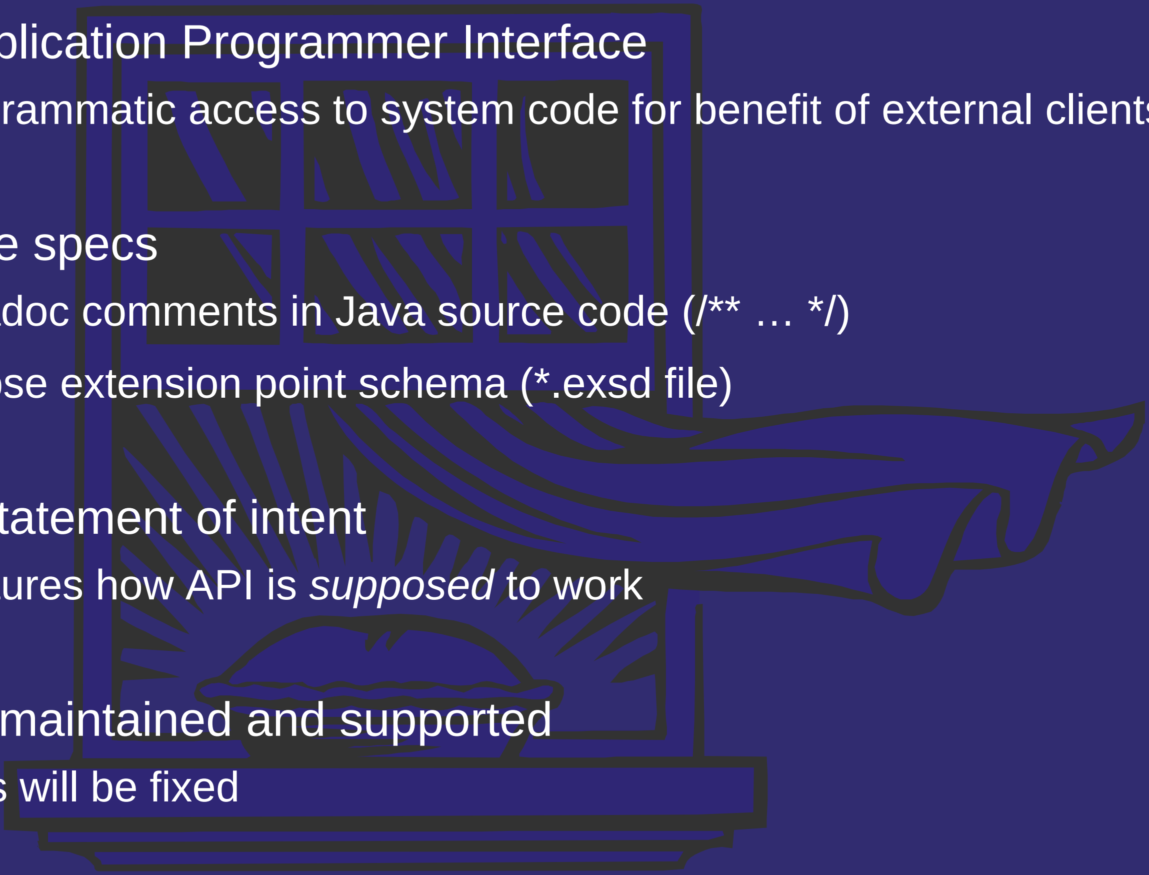
Jim des Rivières
IBM Ottawa
jeem@ca.ibm.com

Outline of Talk

- Part I – Review
  - Philosophy, psychology, and sociology of APIs
- Part II – Evolving APIs
  - Change APIs and keep existing clients happy
- Part III – Eclipse APIs from 2.1 to 3.0
  - Having your cake and eating it, too

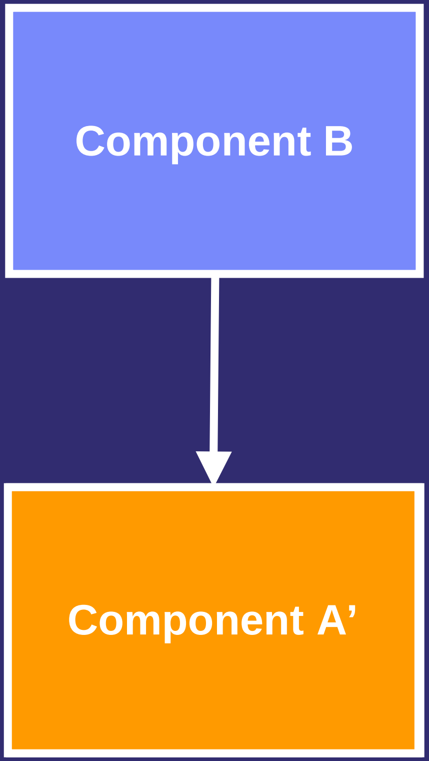


APIs are Specified and Supported



- API = Application Programmer Interface
  - Programmatic access to system code for benefit of external clients
- APIs have specs
  - Javadoc comments in Java source code (/** … */)
  - Eclipse extension point schema (*.exsd file)
- Spec is statement of intent
  - Captures how API is *supposed* to work
- APIs are maintained and supported
  - Bugs will be fixed

Abstract Thought Experiment #1



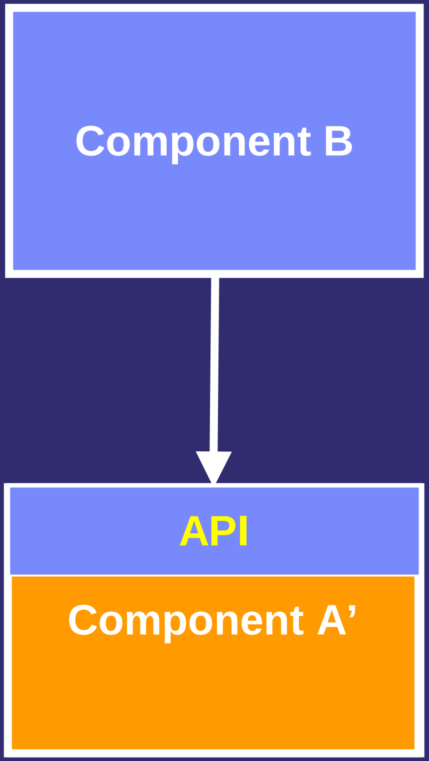
- A written first
- B written with particular A
- A + B are working together

- Now change A independent of B

- A’ + B guaranteed to work?

NO

Abstract Thought Experiment #1



- A written first
- B written with particular A
- A + B are working together
- A has specified API
- A implements API spec
- B lives within A’s API spec

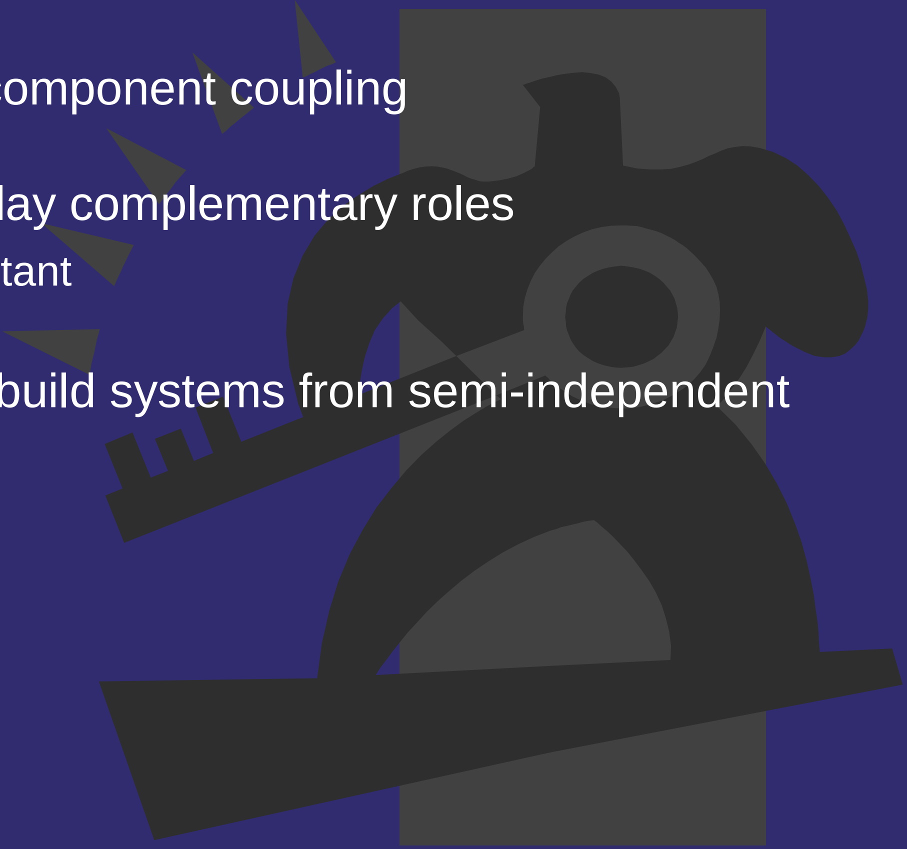
- Now change A independent of B
- No change to API spec
- A’ still honors API spec

- A’ + B guaranteed to work?

YES

API Specs Play a Critical Role

- Component code provides system behavior
- API spec limits inter-component coupling
- Code and API spec play complementary roles
  - Both critically important
- Only scalable way to build systems from semi-independent components



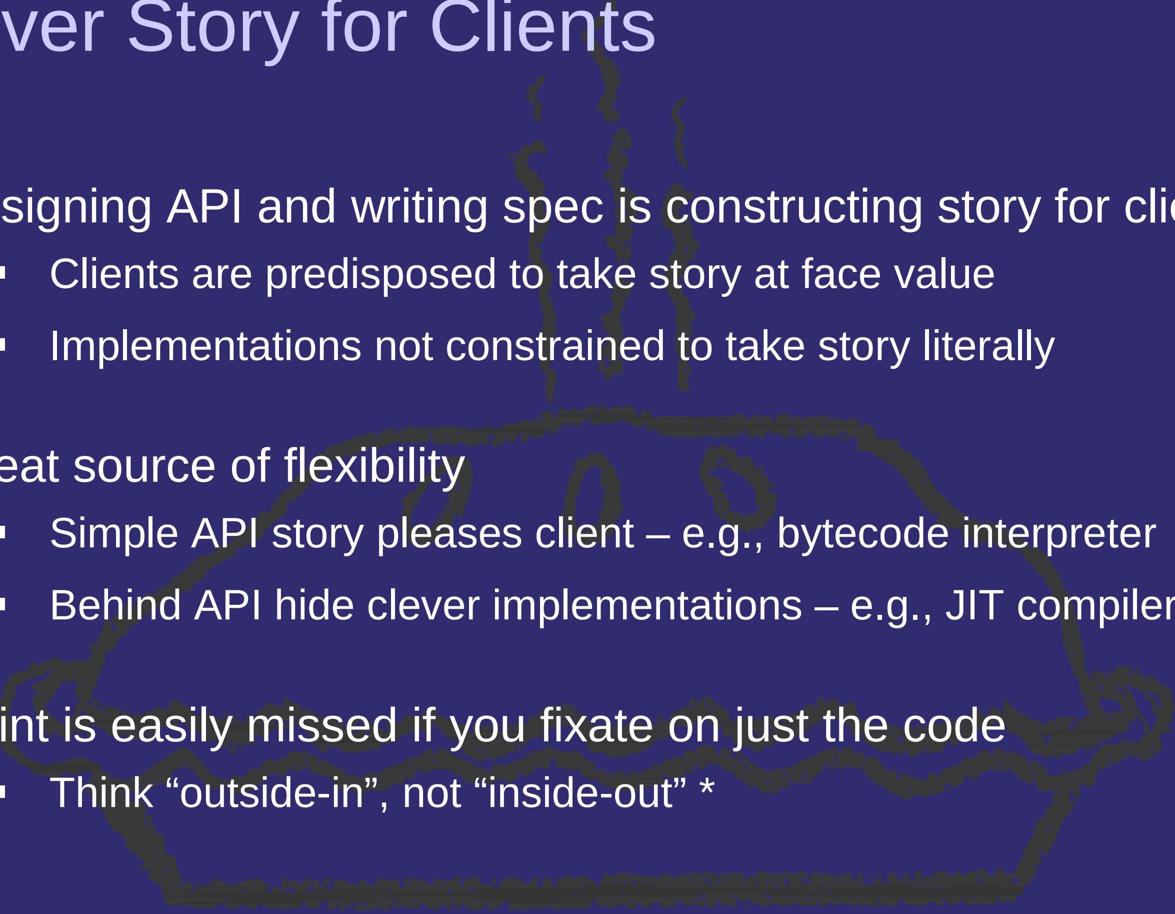
Abstract Thought Experiment #2

- Imagine you have just got some code working
  - No API - nothing declared public
- Task: expose an API for this code
  - By selectively making classes, methods, and fields public
- Extract just the API signatures
  - Erase non-API packages, classes, methods, and fields
  - Erase bodies of API methods
- Your task: write explanation for clients unfamiliar with any of it
  - Explain what every class, method, and field is all about
  - Explain how they are be used to solve client problems
  - Make the story compelling and seamless

Abstract Thought Experiment #2.B

- Now, imagine being handed such an API spec
  - Compelling and seamless story
- Would you believe this story is literally true?
  - Do you really believe that your JVM executes bytecodes? Interesting.
- How could you tell without looking at the implementation?
- Would it bother you if you were deceived?

API is Cover Story for Clients

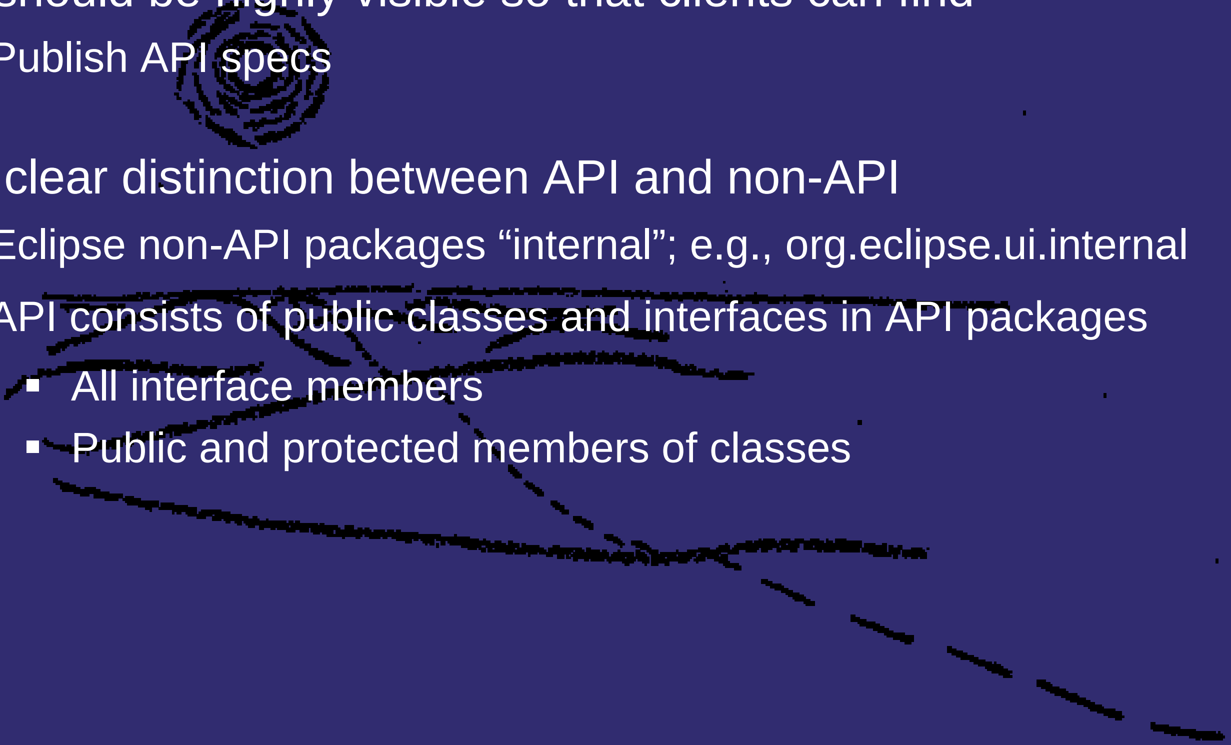


- Designing API and writing spec is constructing story for clients
  - Clients are predisposed to take story at face value
  - Implementations not constrained to take story literally
- Great source of flexibility
  - Simple API story pleases client – e.g., bytecode interpreter
  - Behind API hide clever implementations – e.g., JIT compiler
- Point is easily missed if you fixate on just the code
  - Think “outside-in”, not “inside-out” *

* Jeff Johnson, *GUI Bloopers*

APIs Should be Visible

- APIs should be highly visible so that clients can find
  - Publish API specs
- Draw clear distinction between API and non-API
  - Eclipse non-API packages “internal”; e.g., org.eclipse.ui.internal
  - API consists of public classes and interfaces in API packages
    - All interface members
    - Public and protected members of classes

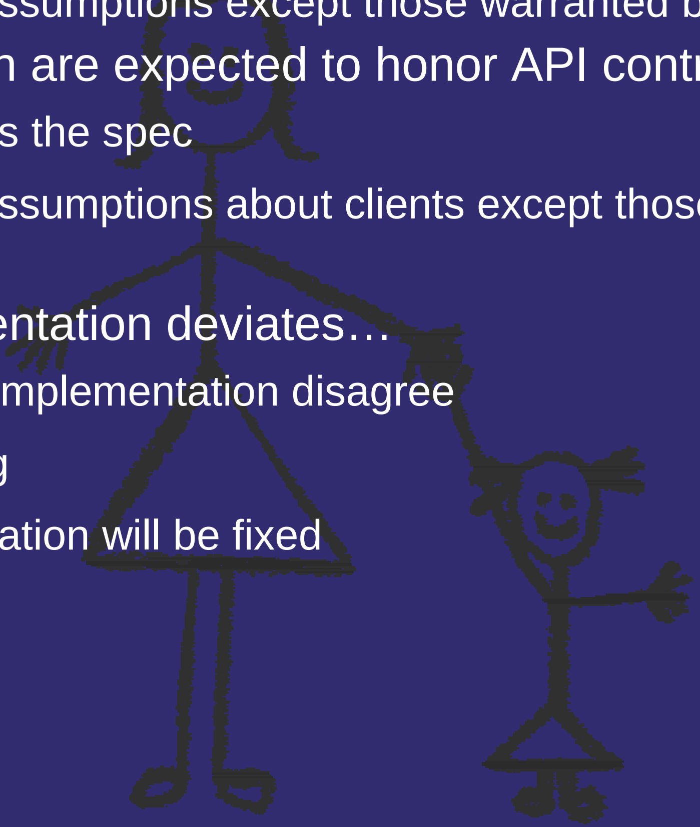


APIs Should be Trustworthy

- Good fences make good neighbors
- Compile time
  - final modifier
- Specification
  - “This interface is not intended to be implemented by clients”
  - Legal values for arguments
- Run time
  - Check argument validity
  - Correct thread
- APIs should draw attention and earn respect

APIs Operate on the Honor System

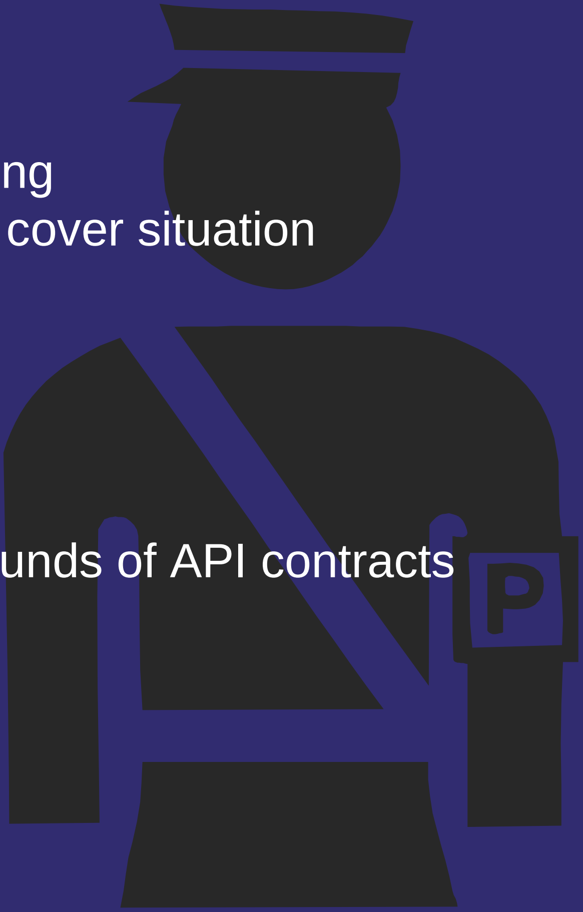
- Clients are expected to honor API contracts
  - Make no assumptions except those warranted by API spec
- Implementation are expected to honor API contracts
  - Implements the spec
  - Make no assumptions about clients except those warranted by spec
- When implementation deviates…
  - Spec and implementation disagree
  - Report bug
  - Implementation will be fixed



Clients May Stray

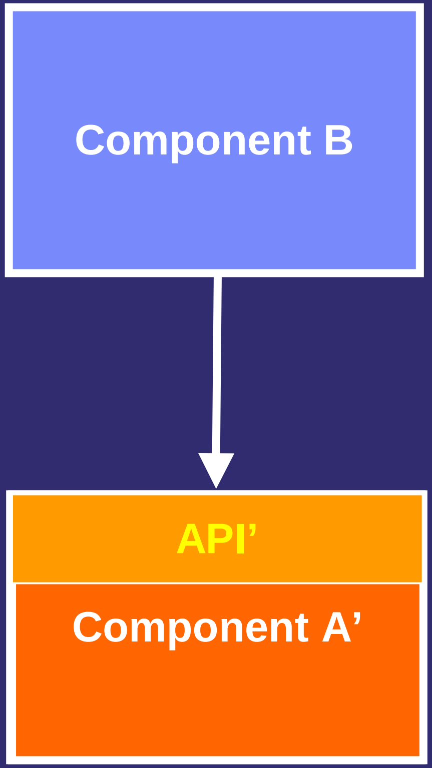
- When client strays…
  - Whether accidentally or deliberately
- Some misuses will be caught
  - At compile time – e.g., private access modifier
  - At run time – e.g., argument checking
- Impossible to prevent all instances of misuse
  - Client implements API interface not intended for them
  - Client references public class in internal package
- Impossible to detect all instances of misuse at runtime
  - Multi-threaded client makes unsafe use of non-thread-safe API

Clients Need to Be Vigilant



- Errant client code may appear to be working
- But may fail since API spec does not fully cover situation
  - Fail in different operating environment
  - Fail as implementation bugs get fixed
  - Fail as implementation improves
- Clients are responsible for living within bounds of API contracts
- There are no API police

Abstract Thought Experiment #3



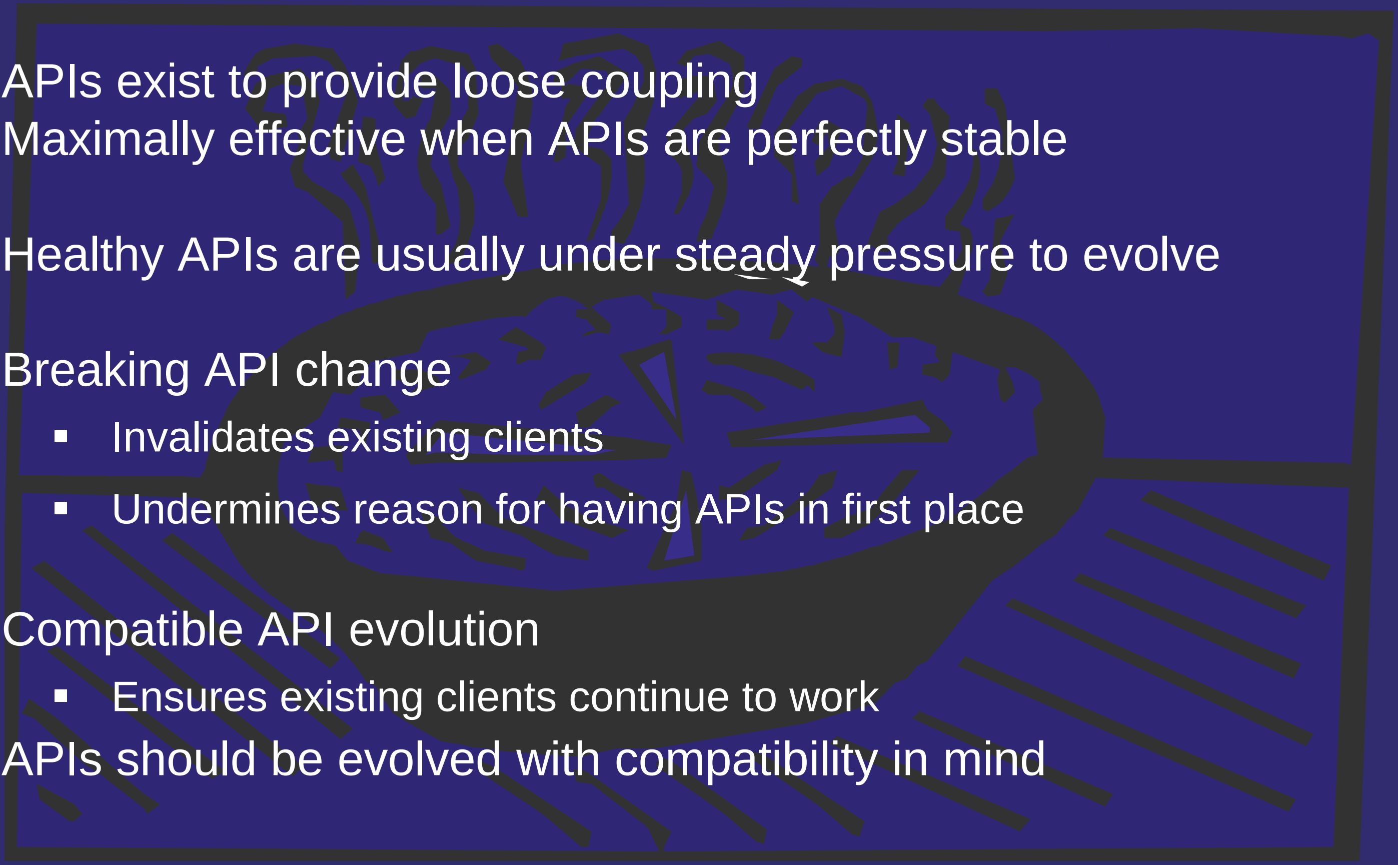
- A provides API
- B uses A’s API
- A and B honor A’s API spec
- A + B work together

- Change A’s API independent of B
- Realign A implementation to match

- A’ + B guaranteed to work?

NO

APIs Should be Stable



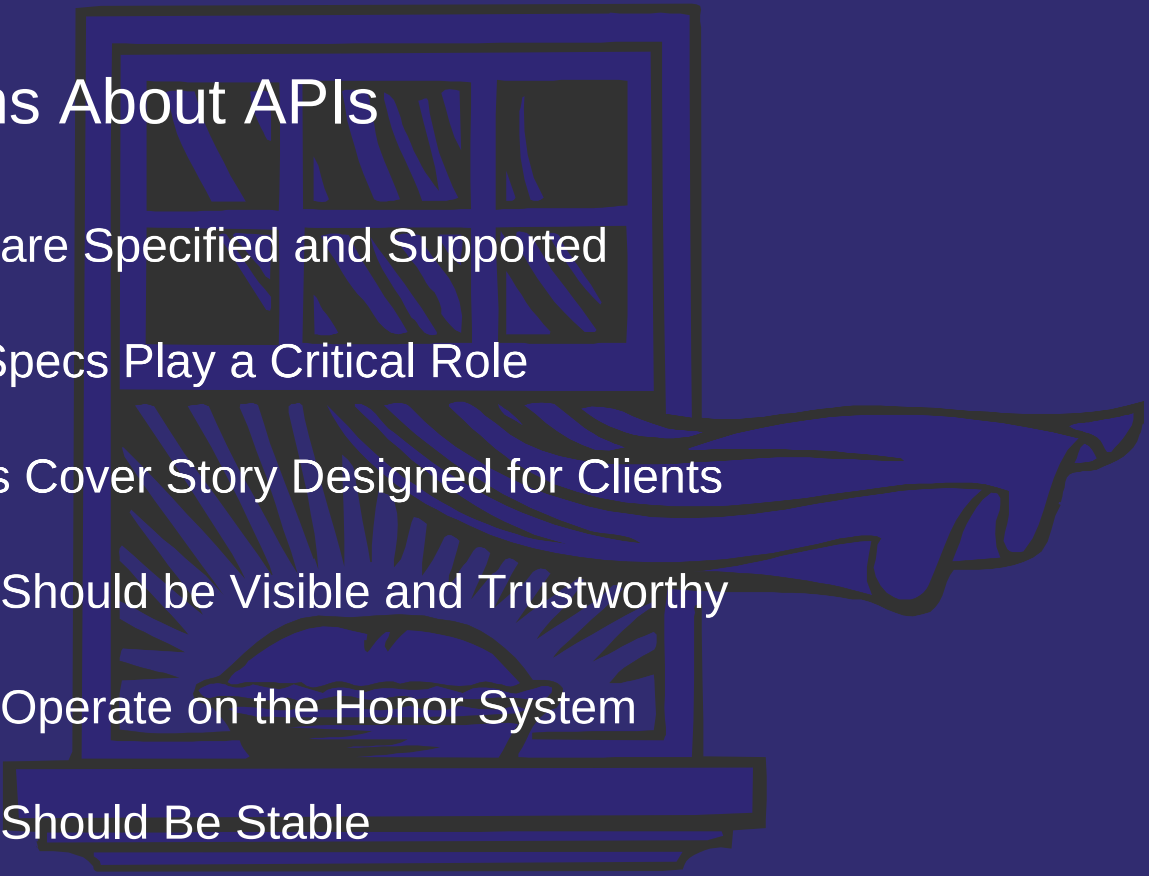
- APIs exist to provide loose coupling
- Maximally effective when APIs are perfectly stable

- Healthy APIs are usually under steady pressure to evolve

- Breaking API change
  - Invalidates existing clients
  - Undermines reason for having APIs in first place

- Compatible API evolution
  - Ensures existing clients continue to work
- APIs should be evolved with compatibility in mind

Review - Summary



Truisms About APIs

- APIs are Specified and Supported
- API Specs Play a Critical Role
- API is Cover Story Designed for Clients
- APIs Should be Visible and Trustworthy
- APIs Operate on the Honor System
- APIs Should Be Stable

Part II – API Evolution

- APIs need to evolve from release to release
  - Changes to API could invalidate existing clients
- Evolve API in compatible ways
  - Preserve as much value as possible across API changes
  - Keep existing clients working
- Two general considerations
  - Contract compatibility – Honor existing API contracts
  - Binary compatibility – Keeping the JVM happy

Contract Compatibility

Before

```
/** Returns the non-empty list of indices. */
public int[] getIndices();
```

After

```
/** Returns the list of indices. The list may be empty. */
public int[] getIndices();
```

Breaks some existing callers

```
int[] d = getIndices();
System.print(d[0]);  // possible array index out of bounds
```

However, existing implementers are fine

```
public int[] getIndices() {
   …; return result; // result is non-empty
}
```

Evolving API Contracts

- API contracts are expressed in API specs
- API contracts promise the client certain things
  - Clients can play multiple roles – e.g., caller, implementer
  - Different roles have different contracts
- Changes to contracts should not invalidate existing clients

Binary Compatibility

Before

```
public void register(String key);
```

After

```
public void register(Object key);
```

Existing calls re-compile as expected

```
register(“foo”);  // no compile error
```

But existing binaries no longer link

```
register(“foo”);  // link error
```

Binary Compatibility DON'Ts for API Elements

1. Rename a package, class, method, or field
2. Delete a package, class, method, or field
3. Decrease visibility (change public to non-public)
4. Add or delete method parameters
5. Change type of a method parameter
6. Add or delete checked exceptions to a method
7. Change return type of a method
8. Change type of a field
9. Change value of a compile-time constant field

10. Change an instance method to/from a static method
11. Change an instance field to/from a static field
12. Change a class to/from an interface
13. Make a class final (if clients may subclass)
14. Make a class abstract (if clients may subclass)

…

Binary Compatibility DOs for API Elements

1. Add packages, classes, and interfaces
2. Change body of a method
3. Do anything you want with non-API elements
4. Add fields and type members to classes and interfaces
5. Add methods to classes (if clients cannot subclass)
6. Add methods to interfaces (if clients cannot implement)
7. Add non-abstract methods to classes (if clients may implement)
8. Reorder class and interface member declarations
9. Change value of a field (if not compile-time constant)

10. Move a method up to a superclass
11. Make a final class non-final
12. Make an abstract class non-abstract
13. Change name of method formal parameter

…

Binary Compatibility

- Java VM has special rules for *binary compatibility*
- API changes should be binary compatible
  - Existing clients should continue to work *without recompiling*
- N.B. Java compiler does ***not*** detect this kind of breakage
- Ref: *Evolving Java-based APIs* http://eclipse.org/eclipse/development/java-api-evolution.html

Adding Methods to API Interfaces

- API interfaces used to hide implementation work well
  - “/** … This interface is not intended to be implemented by clients */”
  - Add new methods to API interface
  - Add corresponding methods to implementing class

```
package org.eclipse.core.resource;
/** … This interface is not intended to be implemented by clients */
public interface IWorkspace {
   …
   public boolean isTreeLocked();  // new
}

package org.eclipse.core.internal.resource;
class Workspace implements IWorkspace {
   …
   public boolean isTreeLocked() {…}
}
```

Avoid API Interfaces that Clients May Implement

- API interfaces *that clients may implement* are problematic
  - Adding method breaks binary compatibility
- Use API class instead of API interface…
  - When client may implement
  - When there is a chance new methods needed in future
  - N.B. converting interface to class breaks binary compatibility

Adding Methods via I*2 Extension Interfaces

- If no choice, add new methods in extending API interface
  - Avoids breaking existing clients that implement

```
package org.eclipse.ui;
public interface IActionDelegate { … }  // original interface

public interface IActionDelegate2 extends IActionDelegate {
   void dispose();   // new
}
```

Usage

```
IActionDelegate d = new IActionDelegate2() {…};

if (d instanceof  IActionDelegate2) {
   IActionDelegate2 d2 = (IActionDelegate2) d;
   d2.dispose();  // call new method
}
```

How to Delete API

- API deletion always breaks any existing clients
- But replacing API with improved version is usually doable

Replacing API Methods

- Add replacement API method
- Deprecate original method
  - Ensure original method continues to work

```
package org.eclipse.jdt.core.dom;
public class Message {
   …
   /** …
    * @deprecated Use getStartPosition() instead
    */
   public int getSourcePosition() {    // rename getStartPosition()
      return getStartPosition();    // forward to new method
   }
}
   public int getStartPosition() {
      …
   }
}
```

API Evolution - Summary

- Evolve API in compatible ways
  - Honor existing API contracts
  - Observe technical rules for Java binary compatibility
- Usually feasible to find way to improve API
  and keep existing clients working without recompiling
- Design APIs with future evolution in mind

Part III – Eclipse APIs from 2.1 to 3.0

- Eclipse 3.0 is major undertaking
  - Need to move Eclipse forward into new areas
- Large number of users of Eclipse 2.1-based products
  - Will be held back if 3.0 does not run 2.1-based plug-ins
- Knew at outset there would be challenges
- Examples of how we're meeting those challenges
  - Xerces
  - RCP Runtime
  - RCP UI

Xerces

- J2SE 1.4 now includes XML support
  - IBM 1.4 JRE includes Xerces XML library
  - Sun 1.4 JRE includes other XML library (not Xerces)
- Cannot include org.apache.xerces plug-in if running on IBM JRE
  - Loads Xerces classes from JRE-supplied library anyway
- Eclipse needs to run on all 1.4 JREs
- Decision:
  - Eclipse 3.0 plug-ins must use J2SE 1.4 XML APIs
  - Drop org.apache.xerces plug-in

Xerces

- Breakage
  - Existing plug-ins that use Xerces library
- We DON'T hide breakage from 3.0 plug-in developers
  - This is story for 3.0 onwards
- We DO HIDE breakage from 2.1 plug-ins at runtime
  - Products shipping on IBM 1.4 JRE
    - Include dummy org.apache.xerces plug-in
    - Allow refs to be satisfied by Xerces in IBM 1.4 JRE
  - Products shipping on other 1.4 JREs
    - Include old 2.1 org.apache.xerces plug-in

RCP Runtime

- Platform Runtime
  - Should provide functionality useful in wide variety of applications
  - Should allows dynamic addition (and removal) of functionality
  - Should be adaptable to many operating environments
- OSGi provides dynamic delivery of managed services
- Decision: Re-host 3.0 Eclipse Platform on OSGi

RCP Runtime

- Breakage
  - Changes to plug-in format
  - OSGi APIs and mechanisms replace many Platform Runtime APIs
  - Obsolete API moved to org.eclipse.core.runtime.compatibility plug-in
- We give 3.0 plug-in developers some options
  - 2.1 plug-in format is still fully supported
  - 3.0 also supports new OSGi-based plug-in format (bundles)
  - PDE can handle both forms (and mixtures)
  - Plug-ins can move to new story if compelling reason to
- We hide breakage from 2.1 plug-ins at runtime
  - Fix prerequisites on start up

RCP UI

- Workbench
  - Should provide functionality useful in wide variety of applications
  - Should be lean
- Requires
  - Shedding IDE biases
  - Severing ties to workspace & resources
- Good news
  - 2.1 Workbench API is 99% free of workspace & resources
- Bad news
  - API methods for opening arbitrary editor on IFile
  - IDE-specific extension points; e.g., org.eclipse.ui.projectNatureImages

RCP UI

- Decision: cut workbench into 2 parts for 3.0

1. Generic workbench
   - Bulk of existing workbench APIs and extension points
   - New APIs for configuring workbench personality
   - Existing org.eclipse.ui plug-in
   - Does not depend on workspace & resources
2. IDE workbench
   - IDE-specific APIs and extension points
   - New org.eclipse.ui.ide plug-in
   - Depends on workspace & resources

RCP UI

- Breakage
  - Extension points in IDE plug-in have different IDs
  - Plug-in prerequisites
  - Some old API methods moved to new classes in IDE plug-in
- We DON'T hide breakage from 3.0 plug-in developers
  - This is story for 3.0 onwards
- We DO HIDE breakage from 2.1 plug-ins at runtime
  - Re-map extension points and fix prerequisites on start up
  - Deleted API methods are more challenging

Deleting API Methods

- Wiegand’s technique to preserve runtime binary compatibility

2.1 API

```
public interface IWorkbenchPage {
  IEditorPart openEditor(IEditorDescriptor ed);
  IEditorPart openEditor(IFile file);  // to delete
}
```

3.0 API

```
public interface IWorkbenchPage
         extends ICompatibleWorkbenchPage {
 IEditorPart openEditor(IEditorDescriptor ed);
}

interface ICompatibleWorkbenchPage {
    // empty
}
```

Mask by alternate declaration (in optional org.eclipse.ui.workbench.compatibility fragment)

```
interface ICompatibleWorkbenchPage {
  /** @deprecated */
  public IEditorPart openEditor(IFile file);
}
```

Eclipse APIs from 2.1 to 3.0 - Summary

- Eclipse 3.0 is evolution of Eclipse 2.1
  - Compatible except in a few areas
- 2.1 plug-ins will need to be ported to 3.0
  - Ref: *Eclipse 3.0 Porting Guide* http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.platform.doc.isv/porting/eclipse_3_0_porting_guide.html
- 2.1 binary plug-ins in the field will work with 3.0
  - Need community help to verify this
  - IMPORTANT to report runtime binary API compatibility problems

API-related Resources

- *How to Use the Eclipse API*, by Jim des Rivieres
  http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html
- *Effective Java Programming Language Guide*, by Josh Bloch
  http://java.sun.com/docs/books/effective/
- *Requirements for Writing Java API Specifications*
  http://java.sun.com/products/jdk/javadoc/writingapispecs/index.html
- *How to Write Doc Comments for the Javadoc Tool*
  http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html
- *Evolving Java-based APIs*
  http://eclipse.org/eclipse/development/java-api-evolution.html
- *Contributing to Eclipse*, by Erich Gamma and Kent Beck
  http://www.aw-bc.com/catalog/academic/product/0,4096,0321205758,00.html
- *Internal Tool* (reports cross-plug-in references to internals)
  http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/jdt-core-home/tools/internal/index.html