



Tutorial 15: Java API Design



Boris Bokowski

Eclipse Platform UI committer
IBM Rational Software
Ottawa, Canada

Based on presentation material co-authored with John Arthorne and Jim des Rivieres, IBM Canada

Table of contents

- Introduction (slides 7-24)
- Writing API specifications (slides 25-46)
- Evolving APIs (slides 47-50)
 - Contract compatibility (slides 51-52)
 - Binary and source compatibility (slides 53-71)
 - Techniques for API evolution (slides 72-73)
- API design best practices (slides 74-80)
- API tools (slides 81-84)
- Stories from the API trenches (slides 85-94)
- Legal Notices (slide 95)

Instructor biography

Boris Bokowski is a Software Developer with IBM Rational Software in Ottawa, Canada.

He is a full-time Eclipse committer working on the Platform UI team, a member of the Eclipse Architecture Council, and part of the "API police" for the Eclipse Platform.

Boris has over ten years of Java experience and has worked in many different domains like client-server protocols, compiler implementation, web application servers, and graphical user interfaces.

He holds a PhD in computer science from Freie Universität Berlin, Germany.

Agenda

- Introduction, Example 60 minutes
 - API Specification 60+ minutes
 - API Evolution 60+ minutes
 - API Design Best Practices 20 minutes
 - API Tools 10 minutes
-
- API Stories every now and then as appropriate

Objectives of the course

- Appreciate the role of having strong API specifications.
- Understand different perspectives:
 - Specification, Implementation, Testing, Client.
- Dangers of underspecification and overspecification.
- How to evolve an API without breaking clients.
- Understand the difference between:
 - Contract compatibility,
 - Source compatibility,
 - Binary compatibility.
- API design process best practices.

Abstract

Good and dependable APIs are critical to the long-term success of any larger software project. This tutorial is based on our experience with developing and evolving the Eclipse platform API over the past several years.

It presents best practices for API specification, API evolution, and the API design process. In particular, it gives concrete technical advice on how to design and specify Java APIs, and how to evolve APIs in a way that maintains compatibility.

The tutorial covers source compatibility and binary compatibility of changes to interfaces, classes, generics, and other language features up to Java 5.

Stories from the Eclipse API trenches and the API compatibility quiz will engage participants and make this complex topic accessible.

Introduction

My eyes are dim I cannot see.
I have not got my specs with me.
I have not got my specs with me.

---*The Quartermaster's Song*

What are APIs?

- APIs are interfaces with **specified** and **supported** behavior
- Specified: How is it *supposed* to work?
- Supported: Bugs will be fixed. API contract will be honoured.
- Key for successful componentization
 - What is the public interface of a component?
 - What is its specified behaviour?
- Fundamental for platforms
 - E.g.: Win32, Java SE, IBM 360, Eclipse

Why APIs?

- They are key for successful **componentization**:
 - API spec limits inter-component coupling
 - Only scalable way to build systems from semi-independent components
- Components require API. Clients need to know:
 - What is the public interface of a component?
 - What is its specified behaviour?
 - What kind of support and API contract can I rely on?

Why APIs? (cont.)

- Fundamental for **platforms**.
 - E.g.: Operating systems (Windows, Linux, MacOS), Eclipse, ...
 - Evolve over time, but in a compatible way.
- Make **multiple implementations** possible:
 - E.g.: Servlet API, XML parsing, Browser DOM, ...

API specs

- API specs play many key roles
 - A. Tell client what they need to know to use it
 - B. Tell an implementor how to implement it
 - C. Tell tester about key behaviors to test
 - D. Determines blame in event of failure

Exercise: First baby specs

- **Write Javadoc specs for this API class**

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
public final class Stack {  
    public Stack();  
    public void push(Object v);  
    public Object pop();  
    public Object peek();  
    public boolean isEmpty();  
    public int search(Object o);  
}
```

Question: What do we spec for constructor?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

/**

*** ???**

***/**

public Stack();

Question: What do we spec for constructor?

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
/**
```

```
 * Creates a new empty stack.
```

```
 */
```

```
public Stack();
```

- Initial conditions

Question: What do we spec for methods?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
/**
```

```
*
```

```
*      ???
```

```
*
```

```
*
```

```
*/
```

```
public void push(Object o);
```

Question: What do we spec for methods?

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
/**
```

```
* Adds the given object to the top of this stack.
```

```
*
```

```
* @param o
```

```
*     the object to push; may be null
```

```
*/
```

```
public void push(Object o);
```

- Method specs include
 - Purpose
 - Parameters
 - Postconditions

Question: What do we spec for methods?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
/**
```

```
*
```

```
*     ???
```

```
*
```

```
*
```

```
*
```

```
*/
```

```
public Object pop();
```

Question: What do we spec for methods?

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
/**
```

```
* Removes and returns the object on the top of this stack.
```

```
* The stack must not be empty.
```

```
*
```

```
* @return the object popped off the stack; may
```

```
*         be null
```

```
*/
```

```
public Object pop();
```

- Method specs include
 - Preconditions
 - Results

Question: What do we spec for methods?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

/**

???

***/**

public int search(Object o);

Question: What do we spec for methods?

/ © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */*

*/** Returns the position of the given object on this stack. The position is*

** the distance from the top of the stack or, equivalently, the number of*

** calls to {@link #pop()} required to uncover the object. If the same*

** object occurs more than once, the position returned is that of the one*

** closest to the top. Returns -1 if the object is not present. The method*

** uses {@link #equals(Object)} to compare objects.*

** @return the distance of the given object from the top of the stack, or -1*

** if the object is not present*

**/*

```
public int search(Object o);
```

- Method specs include
 - Other important details

Question: What do we spec for class?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
/**
```

```
*
```

```
*      ???
```

```
*
```

```
*
```

```
*/
```

```
public final class Stack {
```

Question: What do we spec for class?

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
/**
```

```
* Represents a stack (last-in-first-out).
```

```
* <p>
```

```
* This class is not thread-safe.
```

```
* </p>
```

```
*/
```

```
public final class Stack {
```

- Class specs include
 - Purpose
 - General usage

Discussion

Lessons learned

- API is not just public methods

No specs. No API.

API Specification

Appropriate level of specification detail

- Is the specification too specific or detailed, making it difficult to evolve or implement?
 - Gain flexibility by hiding implementation.
 - API is a cover story.
- Is the spec too vague, making it difficult for clients to know the correct usage?
 - Always assume clients are unfamiliar with your code.
 - Tell a compelling and seamless story.
 - Clients may end up relying on implementation details.
- Is the API designed to be implemented or extended by clients?
 - Be aware of different kinds of clients.
 - Keep Client API separate from Service Provider API.

API problems that make life difficult for clients

- Spec too vague
- Completeness (e.g., add but no remove)
- Names (avoid overly long names for elements that are used a lot)
- All interfaces, but no constructor
- Check assumptions about defaults
- Internationalization: specify if strings are visible to the end-user
- Non-functional requirements
 - thread safety
 - progress reporting
 - being able to cancel long-running operations
 - nesting, composeability

API problems that make life difficult for implementations

- Exposing unnecessary implementation details
- Factory methods: specifying that it "returns a new instance" rather than "returns an instance" - prevents future caching/pooling
- Specifying the whole truth, rather than just the truth
- Specifying all failure cases (instead of "reasons for failure include...")
- Specifying that an enum or similar pool of types is a closed set - prevents adding entries later (eg: resource types)
- Specifying precise data structures of return types (HashSet, rather than just Set or Collection)

API problems that make life difficult for implementations (cont.)

Making promises that are hard to keep:

- Over-specifying precision of results (returns the number of nanoseconds since the file was changed)
- Returning a data structure that promises to remain up to date indefinitely (leaking live objects)
- Promises about the order of operations - prevents future concurrency
- Real-time promises

API Contract language

- The language used in an API contract is very important
- Changing a single word can completely alter the meaning of an API
- It is important for APIs to use consistent terminology so clients learn what to expect

API Contract language

- RFC on specification language: <http://www.ietf.org/rfc/rfc2119.txt>
- **Must, must not, required, shall:** it is a programmer error for callers not to honor these conditions. If you don't follow them, you'll get a runtime exception (or worse)
- **Should, should not, recommended:** Implications of not following these conditions need to be specified, and clients need to understand the trade-offs from not following them
- **May, can:** A condition or behavior that is completely optional

API Contract language

Some Eclipse project conventions:

- **Not intended:** indicates that you won't be prohibited from doing something, but you do so at your own risk and without promise of compatibility. Example: "This class is not intended to be subclassed"
- **Fail, failure:** A condition where a method will throw a checked exception
- **Long-running:** A method that can take a long time, and should never be called in the UI thread
- **Internal use only:** An API that exists for a special caller. If you're not that special caller, don't touch it

Advanced Topics

- Subclassing
- Listeners
- Java 5 language features
 - Enums
 - Generics

Specs for Subclassers

- Subclasses may
 - **"implement"** - the abstract method declared on the subclass must be implemented by a concrete subclass
 - **"extend"** - the method declared on the subclass must invoke the method on the superclass (exactly once)
 - **"re-implement"** - the method declared on the subclass must not invoke the method on the superclass
 - **"override"** - the method declared on the subclass is free to invoke the method on the superclass as it sees fit
- Tell subclasses about relationships between methods so that they know what to override

Listeners

- Many opportunities to make mistakes:
 - Listener type: interface or abstract class
 - Event: dedicated type, or just parameters
 - `addListener/removeListener`
- Easy to be inconsistent across multiple instances of this pattern
- Best practices:
 - Listener as interface if just one “`handleEvent`” method, otherwise abstract class to make it evolvable
 - Use dedicated event type (e.g. class with fields) because you may want to add additional fields later
 - Use `equals()` when comparing listeners in `add/removeListener`
 - Allow adding listeners while notifying, but don't call newly added listeners

Story: Locked into a mess

- Resource change events run synchronously at the end of operations that change resources
- Resource changes are not allowed during change events
- Uh oh... “locking” the workspace while synchronous listeners are being called is a recipe for deadlock: thread A calls a listener, which tries to acquire some other lock. Thread B owns that other lock tries to modify a resource.
- For example, listener calls syncExec while UI thread is trying to modify a resource
- It would break API to make listeners asynchronous, or to allow resource changes while listeners are being called
- We have designed ourselves into a hole: resource listeners will always be deadlock prone.
- A fancy hack allows the UI thread to run sync execs while blocked, but problem remains for other kinds of locks

Enums

- Enumeration types are a class type with self-typed constants

```
public enum Direction = {NORTH, EAST, SOUTH, WEST};
```

- `Direction.NORTH` is of type `Direction`
- Constants are canonical instance - can be compared with `==`
- Pros for use in APIs
 - More strongly typed than ints
- Cons for use in APIs
 - Less flexible than ints

Generic Types

- Pros for use in APIs
 - Permits strong typing in certain situations that would otherwise be loosely typed
 - More errors detected at compile-time type
 - More convenient for callers
 - More convenient for implementers
 - Dovetail with Java Collections API
- Cons for use in APIs
 - None if done well
- Neither Pro Nor Con
 - Performance

Generification

- Introducing generic types into an existing API
- Possible to preserve compatibility
 - E.g., Java Collections API was generified in 1.5
- Language has special provisions for backwards compatibility
- Raw type – using generic type as if it were not generic
 - `List beatles = Arrays.asList("John", "Paul", "George", "Ringo");`
`// raw`
- Raw types are discouraged – compiler warnings by default
- Compatibility between old and new is based on **erasures**

Erasures

- The compiler replaces type variables so that all parameterized types share the same class or interface at runtime

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
public class Stack<E> {  
    public void push(E Object element);  
    public E Object pop();  
}
```

```
Stack<String> stringStack;
```

```
Stack<Integer> integerStack;
```

Converting Raw to Parameterized Types

- Applies if
 - Raw types (e.g. Collections) appear in your API
 - Conversion is contract-compatible
- Return types: making stronger promises, always possible
 - `public Map getArgs() -> public Map<String, String> getArgs()`
- Argument types: enforcing existing contracts at compile time
 - `public void setArgs(Map m) -> public setArgs(Map<String, String> m)`
 - This is a binary compatible change (erasure is the same), BUT...
 - `Map<String, String>` is not equivalent to “Map with String keys and values”
 - Sometimes not easy to step up to stronger contract
 - For example, it is easy if they create the map themselves, but hard to do if they get it from somewhere else
 - Be careful not to require too much from your clients

Introducing Type Variables

- Applies if
 - Your API is like the collection framework (e.g. container types), or
 - You inherit from / delegate to a type that was generified, or
 - `java.lang.Object` appears in your API but clients need to downcast

- Return types: Relieving clients from having to downcast

```
interface IObservableValue { Object getValue(); }  
-> interface IObservableValue<V> { V getValue(); }
```

- Argument types: Enforcing contracts at compile time

```
interface IObservableValue { void setValue(Object value); }  
-> interface IObservableValue<V> { void setValue(V value); }
```

- Don't overdo it, generify cautiously
- Weigh type safety against complexity
- Be aware of ripple effect
- Problematic: Arrays, Fields

Arrays and Generic Types Are Different

- `String[]` is a subtype of `Object[]`, but `ArrayList<String>` is not a subtype of `ArrayList<Object>`!
- Reason for this: Principle of substitutability
A is a subtype of B if B can be substituted whenever an A is expected

- Consider:

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
public static void someMethod(List<Object> someList) {  
    someList.add(new Object());  
}
```

```
List<String> stringList = new ArrayList<String>();  
someMethod(stringList);           type error
```

- Array types: `String[]` is a subtype of `Object[]`, but you will get an `ArrayStoreException` if you try to store an `Object` in an array that was created as a `String` array

Array Types in API and Generification

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
▪ class ArrayList<E> {  
    ...  
    E[] toArray() {  
        // how to implement this?  
    }  
    E[] toArray(E[] es) {  
        // here you can use:  
        Array.newInstance(es.getClass().getComponentType(),  
size());  
    }  
}
```

Solution:

- If arrays are pervasive in your API (as in Eclipse):
Do not generify types that appear as array component types in your API
- Otherwise, generify everything except problematic cases like the one above

Don't overdo it with generics. What does this print?

/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */

```
static class A<T extends A<T>> {
```

```
    public T ping() {
```

```
        return (T) this;
```

```
    }
```

```
}
```

```
static class B extends A<B> {
```

```
    public B pong() {
```

```
        return this;
```

```
    }
```

```
}
```

```
public static void main(String... args) {
```

```
    System.out.println(new B().ping().pong().getClass().getSimpleName());
```

```
}
```

- A) A
- B) B
- C) Compile error
- D) ClassCastException

Lessons learned

- Who can remember all this?

API design is hard.

Story: When interfaces are not your friend

- We had an unnatural obsession in the tech preview (0.9) to declare all APIs as interfaces
- IPath is a class that represents a series of strings separated by slashes, such as file system paths
- When the user types in a path, we want to validate that their input is legal before creating a path
- But, if you haven't created a path yet, we have no objects on which to call a validation method (interfaces can't have statics)
- Clients ended up writing code like this:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
String s = promptUserForPath();  
if (new Path().isValidPath(s)) {  
    IPath validPath = new Path(s);  
}
```

- You can't do everything with interfaces!

API review checklist

- Introductory sentence summarizing purpose
- Pre-conditions
- Post-conditions
- Capturing of argument and result objects
- Specifying failure
- Side effects
- Concurrency
- Event ordering
- Callbacks
- Is the story complete, compelling, and seamless?
- Do all parts of the API pull their own weight?

References

- *Requirements for Writing Java API Specifications*
<http://java.sun.com/products/jdk/javadoc/writingapispecs/index.htm>
- *How to Write Doc Comments for the Javadoc Tool*
<http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html>
- Josh Bloch, *Effective API Design*
<http://www.infoq.com/presentations/effective-api-design>
- Josh Bloch, *Effective Java Programming Language Guide*
<http://java.sun.com/docs/books/effective/>

API Evolution

- Different situations when evolving an API
- Compatibility
- Techniques for evolving APIs
- Techniques for writing APIs that are evolvable

Different styles of API evolution

<i>open-ended set of Clients</i>	<p>Product development without compatibility promises</p> <p>Clients must adapt to changes, cost is unknown</p>	<p>Platform development</p> <p>Clients will continue to work without being recompiled</p>
<i>clients known</i>	<p>Internal product development</p> <p>Clients will adapt to changes, cost is known</p>	<p>Internal product family development</p> <p>Clients should continue to work when recompiled</p>
	<p><i>lock-step development</i> <i>("screw the client")</i></p>	<p><i>not in lock-step</i> <i>("maintain compatibility")</i></p>

But remember

- Try to get it right the first time.
- Failing that, do not break clients.

APIs should be stable.

Story: If at first you don't succeed...

- ITextViewer: interface for displaying and working with text (controller between document model to StyledText widget)
- Uh oh...
 - Clients may implement this interface
- ITextViewerExtension: For changing the event mechanism in 2.0 without breaking existing clients
- ITextViewerExtension2: auto-indent, text hovers, etc in 2.1
- ITextViewerExtension3: mapping between document and widget regions in 2.1
- ITextViewerExtension4: text presentation listeners, focus moving in 3.0
- ITextViewerExtension5: rewrite of ITextViewerExtension3 for 3.0
- ITextViewerExtension6: hyper-link decoration and undo in 3.1

Compatibility

- **Contract** – Are existing contracts still tenable?
- **Binary** – Do existing binaries still run?
- **Source** – Does existing source code still compile?

Contract compatibility

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
/**  
 * Returns the current display.  
 * @return the display; never null  
 */  
public Display getDisplay();
```

After:

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

- Not contract compatible for callers of `getDisplay`
- Contract compatible for `getDisplay` implementors

Contract compatibility

- Weaken method preconditions – expect less of callers
 - Compatible for callers; breaks implementors
- Strengthen method postconditions – promise more to callers
 - Compatible for callers; breaks implementors
- Strengthen method preconditions – expect more of callers
 - Breaks callers; compatible for implementors
- Weaken method postconditions – promise less to callers
 - Breaks callers; compatible for implementors

Compatibility quiz

- Is the code snippet a binary compatible change?
- Is it source compatible?
 - For clients?
 - For implementers?

Compatibility quiz #1

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo() {  
        System.out.print("Yes");  
    }  
}
```

After:

```
public class Test {  
    public void foo() {  
        System.out.print("Oui");  
    }  
}
```



- Binary compatible
- Method bodies do not affect binary compatibility

Compatibility quiz #2

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo() {}  
    public void bar() {}  
}
```

After:

```
public class Test {  
    public void foo() {}  
}
```



- Not binary compatible
- Not source compatible
- Deleting methods is a breaking change

Compatibility quiz #3

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo() {}  
}
```

After:

```
public class Test {  
    public void foo(int flags) {}
```




- Not binary compatible
- Parameters are part of the method signature

Compatibility quiz #4

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo(String s) {}  
}
```

After:

```
public class Test {  
    public void foo(Object o) {}   
}
```

- Not binary compatible
- Source compatible


Compatibility quiz #5

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo(Object o) {}  
}
```

After:

```
public class Test {  
    public void foo(Object o) {}  
    public void foo(String s) {}  
}
```



- Binary compatible
- When source references are recompiled they may be bound to the new method
- Will cause errors in source references with a null argument, such as `test.foo(null)`

Compatibility quiz #6

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Super {  
    public void foo(String s) {}  
}  
  
public class Sub extends Super {  
    public void foo(String s) {}  
}
```

After:

```
public class Super {  
    public void foo(String s) {}  
}  
  
public class Sub extends Super {  
}
```



- Binary compatible
- A different method will be called at runtime when method “void foo(String) is invoked on an object of type “Sub”

Compatibility quiz #7

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public static final int x = 5;  
}
```

After:

```
public class Test {  
    public static final int x = 6;  
}
```



- Not binary compatible
- Constant values that can be computed by the compiler are in-lined at compile time. Referring code that is not recompiled will still have the value “5” in-lined in their code

Compatibility quiz #8

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public static final  
        String s = "foo".toString();  
}
```

After:

```
public class Test {  
    public static final  
        String s = "bar". toString();
```



- Binary compatible
- Constant value cannot be computed at compile-time

Compatibility quiz #9

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
package org.eclipse.internal.p1;  
public class Super {  
    protected void foo(String s) {}  
}  
  
package org.eclipse.p1;  
public class Sub extends Super {  
}
```

After:

```
package org.eclipse.internal.p1;  
public class Super {  
}  
  
package org.eclipse.p1;  
public class Sub extends Super {  
}
```



- Not binary compatible
- Protected members accessible from an API type are API
- Is this valid if class Sub is final or says clients must not subclass?

Compatibility quiz #10

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class E extends Exception {}  
  
public class Test {  
    protected void foo() throws E {}  
}
```

After:

```
public class E extends Exception {}  
  
public class Test {  
    protected void foo() {}  
}
```



- Binary compatible
- There is no distinction between checked and unchecked exceptions at runtime
- Not source compatible because catch blocks in referring methods may become unreachable


Compatibility quiz #11

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class A {  
    public void foo(String s) {}  
}  
public class C extends A {  
    public void foo(String s) {  
        super.foo(s);  
    }  
}
```

After:

```
public class A {  
    public void foo(String s) {}  
}  
public class B extends A {}  
public class C extends B {  
    public void foo(String s) {  
        super.foo(s);  
    }  
}
```



- Binary compatible
- The super-type structure can be changed as long as the available methods and fields don't change


Compatibility quiz #12

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
}
```

After:

```
public class Test {  
    public Test(String s) {  
    }  
}
```



- Not binary or source compatible
- When a constructor is added, the default constructor is no longer generated by the compiler. References to the default constructor are now invalid
- You should **always** specify at least one constructor for every API class to prevent the default constructor from coming into play (even if it is private)
- A constructor generated by the compiler also won't appear in javadoc

Compatibility quiz #13

Before:

```
/* © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */  
public class Test {  
    public void foo() {}  
}
```

After:

```
public class Test {  
    public boolean foo() {  
        return true;  
    }  
}
```



- Not binary compatible because the return type is part of the method signature
- Source compatible only if the return type was previously void

Lessons learned

- It is very difficult to determine if a change is binary compatible
- Binary compatibility and source compatibility can be very different
- You can't trust the compiler to flag non-binary compatible changes

Evolve once, check twice.

Binary compatibility DONT's for API elements

- Rename a package, class, method, or field
- Delete a package, class, method, or field
- Decrease visibility (change public to non-public)
- Add or delete method parameters
- Change type of a method parameter
- Add or delete checked exceptions to a method
- Change return type of a method
- Change type of a field
- Change value of a compile-time constant field
- Change an instance method to/from a static method
- Change an instance field to/from a static field
- Change a class to/from an interface
- Make a class final (if clients may subclass)
- Make a class abstract (if clients may subclass)
- ...

Binary compatibility DO's for API elements

- Add packages, classes, and interfaces
- Change body of a method
- Do anything you want with non-API elements
- Add fields and type members to classes and interfaces
- Add methods to classes (if clients cannot subclass)
- Add methods to interfaces (if clients cannot implement)
- Add non-abstract methods to classes (if clients may implement)
- Reorder class and interface member declarations
- Change value of a field (if not compile-time constant)
- Move a method up to a superclass
- Make a final class non-final
- Make an abstract class non-abstract
- Change name of method formal parameter

...

Evolving Generified API

Add type parameter	Breaks compatibility (unless type was not generic)
Delete type parameter	Breaks compatibility
Re-order type parameters	Breaks compatibility
Rename type parameters	Binary compatible
Add, delete, or change type bounds of type parameters	Breaks compatibility

References

- Gosling, Joy, Steele, and Bracha, *The Java Language Specification*, Third Edition, Addison-Wesley, 2005; chapter 13 Binary Compatibility
http://java.sun.com/docs/books/jls/third_edition/html/binaryComp.html
- *Jim des Rivieres, Evolving Java-based APIs*
<http://www.eclipse.org/eclipse/development/java-api-evolution.html>

Techniques for evolving APIs

- Create extension interfaces, use naming convention (ITurtleExtension, ITurtle2)
- Wiegand's device
- Deprecate and try again
- Proxy that implements old API by calling new API

Story: Wiegand's device

- In Eclipse 3.0, wanted to refactor the monolithic org.eclipse.ui plug-in into RCP and IDE portions
- Some APIs tangled generic RCP concepts with IDE concepts
- Example: generic API for workbench pages allowed opening editors on an IFile, which is an IDE concept because it's linked to building, etc
- John Wiegand came up with a technique that allowed **deletion** of API methods while maintaining backward compatibility
- Source compatibility was broken, but we were willing to live with that

Story: Wiegand's device

2.1 API

/ © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */*

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

3.0 API

/ © Copyright 2007 IBM Corp. All rights reserved. This source code is made available under the terms of the Eclipse Public License, v1.0. */*

```
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);  
}
```

Techniques for enabling API evolution

- Use abstract classes instead of interfaces for non-trivial types if clients are allowed to implement/specialize
- Separate service provider interfaces from client interfaces
- Separate concerns for different service providers
- Mechanisms for plugging in generic behavior (IAdaptable) or generic state, such as getProperty() and setProperty() methods

API design best practices

- Good APIs don't just appear overnight
 - Significant design effort
- Good APIs require design iteration
 - Feedback loop involving clients and implementers
 - Improve API over time
- Components build upon the APIs of other components
 - Need collaborative working relationship between teams

Before you begin

- Have and agree on common API guidelines, e.g.:
 - Eclipse Naming conventions
<http://dev.eclipse.org/naming.html>
 - How to Use the Eclipse API
<http://www.eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html>
- Have someone in charge of the API early on
- Have well-defined component boundaries and dependencies
 - E.g., Core vs. UI

Work with a client

- APIs exist to serve the needs of clients
 - Where are those clients? What do they need?
- Important to work with actual clients when designing API
 - Designing APIs requires feedback from real clients who will use it
 - Otherwise risks crummy API that real clients cannot use
- Find a primary client
 - Ideally: adjacent component, different team, same release schedule
 - E.g., JDT UI is primary client of JDT Core
- Work closely with primary client
 - Listen to their problems with using API
 - Watch out for lots of utility classes in client code symptomatic of mismatch between API and what client really needs
 - Work together to find solutions

Development cycle, releasing an API

- Important to distinguish between
 - On-going development
 - Released API
- In each development cycle, opportunity to work with well-known clients on new API
- Release cycles can be short or long
- Release cycles of API, implementation, and Client may be:
 - The same (lock step development)
 - Different (more likely!)

- What kind of compatibility promise do you make?

Across components, use API, not internals

- Component provides API for arbitrary clients
 - API exists to tame inter-component coupling
- Client components are expected to use API “to spec”
 - Not depend on behavior not covered in API spec
 - Not depend on internals
 - Foolish to make exceptions for close friends
 - Close friends don’t point out flaws
 - Sets bad example for others

Minimize disruptive API changes

- Breaking API changes are very disruptive to clients
 - Non-breaking API changes also cause work for clients
 - Work required to port to use new/improved APIs
- During release cycle
 - Schedule API work for early in cycle
 - Whenever possible, preserve API contracts
 - When not possible, coordinate with clients to arrange cut-over plan
- After release has shipped
 - Evolve APIs preserving API contract and binary compatibility

API Freeze

- Before the freeze (normal development)
 - API may change at any time
 - Make sure you co-ordinate with clients though
 - New API added is provisional by definition
- At the API freeze
 - Everything is now real API or internal code
 - Use @since tags to mark new API introduced in this cycle
- After the API freeze
 - Not making changes to the API provides stability
 - Important for converging towards release, lowers risks
 - In practice, exceptions are needed

Story: What is your preference (part 1)?

- In Eclipse 1.0, JFace had an API for storing and presenting preferences (IPreferenceStore).
- Small handful of non-API plug-ins had adhoc APIs for storing preferences
- In Eclipse 2.0, wanted to add ability for primary feature to override default values of arbitrary preferences, and wanted a generic mechanism to import/export all preferences
- Added `org.eclipse.core.runtime.Preferences` API, which is essentially a copy of the JFace API, but in a non-UI plugin so all plug-ins can access it
- Added compatibility layer for clients calling JFace API, and for the various ad-hoc preference mechanisms elsewhere
- Added compatibility code to read old preferences from disk if the client loads a 1.0 workspace

Story: What is your preference (part 2)?

- In Eclipse 3.0, the preference API's limitations were holding us back. It only allowed two “scopes” for preferences: default values, and explicitly set values.
- Want the ability to store the same preference in multiple scopes: per workspace, per project, per configuration, etc
- `java.util.prefs` and `org.osgi.service.prefs` have the notion of hierarchical preference stores in multiple scopes
- Added `IEclipsePreferences` API that extends OSGi preferences to add capabilities such as preference listeners
- Added a compatibility “scope” for accessing values stored the old way
- Added multiple layers of bridging code to adapt clients of the two old preference APIs to the new API
- Multiple layers of compatibility code to load data stored by the two old preference APIs

Story: Trying to be everything to everyone

- In Eclipse 1.0, we wanted tight integration between development tools and repositories, but not to any repository in particular
- But, if X tools are built on Eclipse, and there are Y repositories, it requires X*Y pieces of integration code
- Instead, created a “unified” VCM API that all tools could talk to without requiring knowledge of any particular repository
- It turns out, there isn’t much common ground between different kinds of repositories
- Even for the basic concepts they use different terminology (edition vs revision, catch up vs. update, release vs. commit, etc)
- The common API didn’t have first class integration for any repositories
- Users were confused by subtle changes in semantics and terms
- In Eclipse 2.0, changed to a minimal common team API and first class integration between platform and repository
- But, no integration between various tool stacks and particular repositories (for example, JDT and CVS are not aware of each other)

API Tools

- Work in Eclipse PDE Incubator to provide API tools
- Four general categories of tooling:
 - API Comparison
 - Component version checking
 - Usage discovery
 - Usage validation
- http://wiki.eclipse.org/index.php/PDE_UI_Incubator_ApiTools

API Comparison

- Create XML-based snapshot of the API of a given component
- Produce a report on API changes between two snapshots
- Identifies potentially breaking changes (not perfect, there are various corner cases)
- Uses API difference analysis to suggest appropriate version number changes

Uses for API Comparison

- Catch breaking API changes early
- Helps in writing migration documentation for clients in cases where breaking changes are necessary
- Useful as input for New & Noteworthy, API documentation

(End of presentation)

Version numbers

- major.minor.service.qualifier
- From release to release, the version number changes as follows:
 - When you break the API, increment the major number
 - When you change the API in an (binary) upwards compatible way, increment the minor number
 - When you make other changes, increment the service number
 - The qualifier is changed for each build

Stories

- If at first you don't succeed...
- What is your preference?
- Locked into a mess
- Trying to be everything to everyone
- Wiegand's device
- When interfaces are not your friend

Legal Notices

- IBM and Rational are registered trademarks of International Business Corp. in the United States and other countries
- Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both
- Other company, product, or service names may be trademarks or service marks of others