API First

Jim des Rivières
IBM Rational Software
Eclipse Platform committer
jeem@ca.ibm.com

Outline of Talk

- Part I – Introduction
  - Platform quality APIs
- Part II - Best Practices for API Development
  - How to work effectively on components with APIs

Part I - Introduction

- Platforms are…

- Useful
  - Platform provides useful services to clients
- Open
  - Platform intended for open-ended set of diverse clients
- Stable
  - Platform provides stability across multiple releases
- Growing
  - Platform evolves from release to release to better meet client needs

Successful platforms

- Examples of successful platforms
  - Win32
  - Intel x86
  - IBM 360
  - J2SE
  - Mac OS

- Successful platforms last a long time
- They don't break existing clients – compatible evolution
- They keep existing binaries working – binary compatibility

Platform quality APIs

- APIs just among adjacent components in same release (“friends”)
  - Unsupported for arbitrary clients
  - Don’t need to be stable from release-to-release
  - You get chance to correct your mistakes
- Stark contrast to platform APIs
- Platform APIs must support open-ended set of clients and be stable
  - Once API is released there’s no going back
  - Breaking changes are bad news
  - Either extend API in next release
  - Or provide improved API alongside old API

Platform quality APIs are unforgiving

- One chance
  - To tell clients how to use API – the “API specs”
  - To choose helpful class and method names
  - To decide between class or interface
- Once API is in service
  - Constrained to work within framework laid down
  - Repair and improve without breaking clients
- Unclear, inadequate, or incorrect API specs are bad news
  - Confusing to clients
  - Fixing API specs may make things worse

Good specs are key to platform quality APIs

- Good APIs needs good specs
  - Code is not enough (even source code)
- API specs (Javadoc)
  - Describe intended behavior of API element
    - How it's supposed to work
  - How clients should behave to ensure they will not be broken by future changes
- Client code written to spec
  - Every reason to believe it will work with any conforming implementation
- Implementation code written to spec
  - Every reason to believe it will support all conforming clients
- Best (&only) known recipe for long-term survival
  - Allows API to endure, evolve, and satisfy clients that build upon it

Part II - Best Practices for API Development

- 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
- Some ways to work effectively on components with APIs, based on Eclipse Platform Project experience

Before you begin

- Have and agree on common API guidelines
  - *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
  - core vs. UI

Work with API clients

- 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

eclipse CON 2005

API First

- Basic “API First” workflow
  1. Work with primary client to decide what API is wanted/needed (design API)
  2. Write comprehensive API specs
  3. Write API test suites
  4. Implement API

  (Expect numerous iterations within basic workflow)

- Helps ensure APIs are
  - Ready for clients to use (specs and impl. in place)
  - High quality and stable
  - Ready to evolve to meet next requirement

Don’t reveal API too early

- “We shall ship no APIs before its time” *(to paraphrase Orson Welles' old wine commercial)*
- Keep work in internal packages until new API is ready
  - API specs in place
  - API test suite
  - Credible implementation
- When ready, release by moving to API package
- Keeps everyone’s expectations realistic
  - Clients
  - Component owner
- Avoids embarrassing recall of unfinished API just before shipping

Write comprehensive API unit tests

- Unit tests for each API element
- Tests written “to spec”
  - Assumes and tests only what is covered in spec
  - Plays role of idealized client
  - Helps you to keep client view of the API
- Implementer’s safety net
  - Catches stupid mistakes before they can screw over clients
  - Helps working relationship with primary client
- Core components
  - Complete API coverage by automated tests
- UI components
  - Partial coverage by automated tests
  - May required additional manual tests for visual elements

Keep API specs consistent at all times

- You write a story. Someone else changes the story. How do you know whether the story still reads well?

- Pain to detect/remove errors and inconsistencies within specs
  - API spec is more like a book than like code
  - Maintaining consistency require re-reading

- Start with initial set of consistent API specs
- Scrupulously maintain consistency as API spec grows/changes

- Have a “Keeper of the Specs”
  - Responsible long-term for maintaining specs and reviewing spec changes
  - Requires thorough knowledge of API and its history

Keep API tests up to date at all times

- Errors and inconsistencies within tests are found immediately
- But insufficient test coverage is hard to detect/remove

- Start with a set of tests that completely cover API
- Scrupulously maintain tests as API spec grows/changes

- Can be done as part and parcel of changing API
- Or by “Keeper of the Specs” if they keep the tests as well

Funnel all external dependencies thru API

- 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
- Most common form of Eclipse component is single plug-in
  - Internal packages have “.internal.” in name
  - Plug-ins should not reference internal packages of other plug-ins

Focus clients on API

- Encourage clients to discuss/propose API changes in terms of existing API
  - Grounds discussion in reality
- API owner ultimately decides how best to address issue
  - API needs to make sense for all clients
  - Final decision may differ for what was proposed

Tread carefully in vicinity of APIs

- Be especially careful when working on code in component with an API
  - Adding/deleting public method
    - internal class - no problem
    - API class - changes API
- Be careful with renaming/refactoring tools
  - Make sure APIs not affected

Tag API elements with @since

- Scrupulously record which release API element first appeared in
  - In Javadoc with @since tag
    - @since 3.1
  - Elsewhere in words
- During release cycle
  - Distinguishes old API from new API added during current release cycle
    - Still flexibility to change (or delete) newly added API
- After release has shipped
  - Help clients targeting release >= N to avoid API added after N

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

Replace, deprecate, and forward

- Add replacement API in non-breaking way
- Deprecate old API but keep it working
  - @deprecated This method has been replaced by {@link #bar()}.
- For new API added earlier in release cycle
  - Negotiate a short fuse on deleting deprecated stuff
  - Make sure clean up happens well before end of release cycle
  - Schedule API review & polish before too late to fix
    - Reviews often uncover inconsistencies
- For API added in earlier release
  - Deprecation warnings inform of better way
  - API contract and binary compatibility preserved

Outgoing API changes for a component

- Component team works in HEAD stream
- Other component teams compile and run against version as of latest (weekly) integration build
- Keep HEAD compiling and working cleanly
  - Do not release changes that would hose other team members
  - Re-run all unit tests before releasing
- Outgoing API changes require coordination with downstream teams
  - Post “preview version” of component a few days before I-build
  - Allows downstream components to coordinate their changes
  - Use nightly builds to reduce risk for a broken integration build

Incoming API changes affecting a component

- For dependent components, use version as of latest (weekly) integration build
- Team members upgrade to latest I-build each week
  - Laggards slow the team down
  - Puts premium on everyone making each I-build better than last
- Incoming API changes from upstream teams
  - Receive “preview version” of component a few days before I-build
  - Allows component to prepare coordinated changes for next I-build

Staging large changes to existing APIs

- E.g., JDT Core support for new J2SE 5 language features
- Work done in stages, over several months
  - Coordinated with primary client
- Parcels of API changes with stub implementations
  - throw new RuntimeException("Not implemented yet");
  - Allows clients to write and compile code (but not run/test)
- Implementation of API parcels done later
  - Clients can run/test code as soon as API is implemented

Summary - Best practices for API development

1. Work with a primary client of the API
2. Don’t reveal API too early
3. Establish & maintain comprehensive API specs from day one
4. Establish & maintain comprehensive API test suites from day two
5. Funnel all inter-component dependencies thru API
6. Tread carefully when hacking in vicinity of APIs
7. Tag API elements to indicate when first introduced
8. Minimize disruptive API changes
9. Replace, deprecate, and forward
10. Coordinate API changes with clients

API-related Resources

- *Eclipse APIs: Lines in the Sand*
  http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/02_des_Rivieres.pdf
- *How to Use the Eclipse API*
  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/
- *Evolving Java-based APIs*
  http://eclipse.org/eclipse/development/java-api-evolution.html
- *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