Eclipse Quality APIs

draft 2 – April 15, 2005
Bjorn Freeman-Benson, Eclipse Foundation
with assistance from: Jim des Rivieres, IBM;
Don Schneider, Amazon.com; Tim Wagner, BEA

Audience

This document is written for Eclipse top-level PMC members and especially for the Architecture Council representatives from those PMCs.

The purpose of this document is to outline the Eclipse community's definition of *Eclipse Quality* APIs as well as the process the community uses to ensure that APIs released by the Eclipse projects meet that standard.

The Challenge

High-quality APIs are important to the Eclipse community because Eclipse is about extensible frameworks and exemplary tools – in other words, high-quality components and platforms that the community can use, extend, and build upon. Project teams choose to be part of the Eclipse community for a variety of reasons, but one of the main ones is the association with the Eclipse brand. Eclipse projects have a number of valuable characteristics:

- **Open Source.** To prevent vendor lock-in[1].
- **Reliable Frameworks.** Version to version binary compatibility for the ecosystem to extend and use in additional tools and products.
- **Predictable Schedules.** Consistent, regular, predictable delivery of new releases.
- **Useful Tools.** Configurable, flexible tools that work out of the box.

The community of Eclipse users, extenders, and builders includes you:

- If you are shipping a product on top of Eclipse want plug-ins (components), you want other vendor's plug-ins to work with your product.
- If you are shipping a plug-in for Eclipse, you want your plug-in to work with as many Eclipse-based products as possible.
- If you are assembling components to create a plug-in, you want to know which ones work together.
- If you're an end-user, you want the system not only to continue working, but also to improve in functionality, when you install new plug-ins and updates to existing plug-ins.

Accomplishing all of these high-quality API goals and consequences is a challenge, but one that the Eclipse developers have been, and continue to be, capable of achieving.

[1] Most Eclipse members adopt open source because it prevents vendor lock-in, not because they actually want to read the source code. Vendor lock-in can range from closed protocols and file formats, single-vendor upgrades, abandoning products, and inadequate response to crucial defects. Eclipse's open source nature allows Eclipse members and users to avoid these issues.

Eclipse Framework Developers

Eclipse projects meet the needs of the Eclipse (plug-in using) ecosystem and community by paying close attention to all couplings between components. We design usable APIs that expose functionality to clients in straightforward ways. We write thorough specifications for the APIs telling the clients how to behave and telling the implementers (us) what we've promised. We keep the APIs stable so that clients can use them to build further products and plug-ins. We evolve the APIs in compatible ways so that existing clients are not broken nor are they forced to ship new versions. We use standard release numbering conventions to distinguish API changes from maintenance releases. We use standard naming conventions to delimit what is API from what is not API. [Eclipse Naming] We maintain automated unit test suites that are run as part of all builds to ensure that every implementation we ship conforms to the APIs we have published.

This is what the ecosystem and the community expects of us, and the *platform quality* APIs are the promises that we make to the community.

APIs and Platform APIs

"Eclipse is a kind of universal tool platform." [Eclipse 2001]
A platform is:

- **Useful**: it provides useful services to clients.
- **Open**: it is intended for an open-ended set of diverse clients including clients that have not yet been designed or even invented.
- **Stable**: it provides stability across multiple releases.
- **Growing**: it evolves and improves from release to release in response to clients' needs.

Successful platforms are long-lived, e.g., Win32, Mac OS, IBM 360, Intel x86, and they are long-lived because they try very hard not to break existing clients. Breaking existing clients is an expensive proposition both in direct costs (the effort needed to upgrade the clients), but even more so in indirect costs (the loss of goodwill amongst the community members). Platforms, such as Eclipse, cannot succeed without a vibrant and active support community, and thus it becomes very important to maintain API binary compatibility from release to release.

An API is a *specified and supported cover story* for the code. An API must be visible, it must be thorough, and it must be trustworthy. An API must be able to answer the user's questions without forcing the user to read the source code, nor asking her to run experiments against the implementation.

An API has:

- **A specification.** A description of the cover story and the necessary details. Specifications should be clear about what is defined and what is not defined. Furthermore, specifications should indicate what areas are likely to change in the future (e.g., this two-valued parameter will probably have N-values in the future). A specification is a difficult document to write, and it is more than just a paragraph or two about the component.

- **A test suite.** A set of unit tests to check if an implementation conforms to the specification.
- **An implementation.**
- **One or more clients.** Usually, just having "a client" is not sufficient – an Eclipse-quality API client needs to be developed by a separate team. A client (API user) that is written by the same team that also creates the API implementation usually suffers from information leakage and thus is not a good test of the API. Even better is two or more clients, each developed by a separate team, all of whom communicate only through the API documents.
- **A support promise.** An implicit or explicit promise[2] that the API will be stable from release to release. A *platform quality* API makes a very strong promise about future binary compatibility. Developers should treat releasing a platform quality API as a promise to support that API forever[3].

A *platform quality* API has very strict requirements in these areas. A less strict API between adjacent components in the same release is called a *friends* API [NetBeans]. A friends API is one that takes advantage of the communication outside the definition of the API, for example, between colleagues who work on the same team. A friends API does not have to support arbitrary clients; it just has to support the clients that the implementers co-workers are creating. A friends API does not have to be stable from release-to-release because entire source code of both the implementation and the client is available for simultaneous refactoring.

In short, a friends API is a good mechanism for the separation of concerns and the encapsulation of abstractions that enable us to build large and complex systems. However, a *friends* API stands in sharp contrast to a *platform quality* API.

A platform API must support an open-ended set of clients. A platform API must be stable from release to release. Once a platform API has been released, in conjunction with the support promise, the component team cannot, e.g., change the class, interface, and method names after realizing that more descriptive names would be more helpful. Similarly, the team is forever[4] constrained to work within the framework they have defined. Repairs and improvements must be made without breaking clients, and even that may be hard because unclear, inadequate, or incorrect APIs may have been (mis-)interpreted by clients. If a client legitimately uses an API and a future release "clarifies" that API, the client will see that as a breaking change, all of which will lead to an unhappy client.

[2] Historically, APIs have only had an implicit support promise, but an explicit one is better. Support promises include statements like "this API will be binary compatible for at least three major releases" and "this API will be binary compatible for at least one major release after the API has been upgraded, i.e., interface ISomething will be maintained for at least one major release after interface ISomething2 has been implemented".

[3] The support promise for a *platform quality* API might be only for four or five years, but in the software life cycle, four or five years is effectively "forever".

[4] Where "forever" is defined in terms of the support promise. However, it is useful to mentally approach the situation as "no changes ever".

A platform API tells the clients how to behave. It tells them the assumptions that have been made, the exclusions that have been made, the contract between the caller and callee, and it includes examples of how to use the API. It should contain diagrams of the architecture and where this component fits into that architecture, descriptions of dependencies on other components, etc.

A platform API provides binary compatibility for future releases (as per the support promise) rather than just source compatibility. Binary compatibility is important for resolving the *upgrade problem* – when Eclipse is used as a platform, each of the plug-ins may be upgraded on different schedules. Not only do end users dislike being forced to upgrade all their plug-ins simultaneously, the various projects' and companies' release cycles rarely coincide. Thus, it is important the modules, components, and frameworks be able to be used and reused as binary and without modification.

An *Eclipse quality* API is a platform API and it doesn't happen by accident. An Eclipse quality API is a lot of work, and it is work that has to be started at the beginning of a release cycle. Eclipse quality cannot be added at the end – it has to be built in. The only thing that can be done at the end of a release cycle is determining whether Eclipse quality has been achieved[5].

The Release Review and Eclipse Quality APIs

During the Release Review, as described in the Eclipse Development Process, the Architecture Council representative of the PMC certifies that the release contains "Eclipse quality" APIs. The remainder of this document outlines a process the Architecture Council representative might follow to make that determination[6]. The representative should expect that the Eclipse community, especially his or her peers, will ask about these processes and issues during the Release Review.

Use Senior Technical Talent

The Architecture Council representative certifies that the release contains "Eclipse quality" APIs. He or she does this by ensuring that each API has been reviewed by either himself or herself, or more likely, by the senior developer or team lead of each component. The key is that the APIs must have been reviewed by senior technical people because junior developers simply don't have enough experience. The API reviews must be done by someone who has enough real world experience to make good decisions.

Is It A Defined API?

The first set of issues one should consider is whether the API is well defined. The API, i.e., the specification, should describe where the component resides in the architecture. The specification should describe the dependencies as well as what plug-ins and packages the API resides in. Do the package, class, and method names follow the Eclipse naming conventions? How are the API details separated from the internal details? For example,

[5] Of course, one can always use the lessons learned to commit to doing better in the next release.

[6] Note that the Architecture Council representative is not required to certify that he or she did all this, only that he or she stands behind the work that was done and that the resulting APIs are "Eclipse quality".

are there separate internal packages or are the API and non-API classes mixed in a single package? If they are mixed, is there a naming convention that makes the separation clear?

What is contained in the API? Is that containment a closed system, or do internal classes escape through the API to the harsh glare of the real world? Code is obviously part of the API, but there are many other API items: extension points, XML schemas, file formats, EMF models, database schemas, meta-data, generated artifacts (such as code), … In fact, anything where there is a promise or contract and a commitment by the project to stand behind that promise – any of that is part of the API and must be "Eclipse quality".

Consider meta-data: if version 3 of the component can read version 2 meta-data, but can only write version 3 meta-data, then the fact that using version 3 will modify the meta-data must be considered part of the API and must be specified, tested, and explained. Generated artifacts are similar: if this component generates code, is the generated code API quality itself? Does the API for this component guarantee that future re-creation of the code will be compatible (method names, variable names, integer constants, etc) with the current generated code? If so, or if not, that needs to be specified as part of this API.

What is the story that this API tells? Is it concise and precise story, or is it vague and wandering? Is it clear and understandable? What does this component do? Who are the clients that have helped define this API? And very importantly, is it "a client" or "the client"? In other words, has the API been constructed to support a diverse set of future clients or has it been designed for a single specific client? (Hint: the latter is probably not an "Eclipse quality" API.)

After we consider the overall story, we can examine the Javadoc (Is it complete? Is it better than "sets variable A"? Does it cross-reference the specification?), the unit tests (Are they complete? Are they instructive?), etc. All of this is necessary for a "defined API".

Is It A Good API?

A solid definition is a necessary but not sufficient condition. If the API is well defined, the next thing to consider is the quality of the API. API quality cannot be quantitatively measured which why the "Eclipse quality" certification process relies on the judgment of the top technical leaders (the Architecture Council representative and the PMC).

In addition to the years of experience that the top technical people will apply to the issue of "is it a good API?," there are a few areas that are sometimes overlooked. A good API doesn't have extra unnecessary bits (extra methods, classes, packages, components, …). A good API discusses thread safety and event ordering issues[7]. A good API is not under-specified. In fact, the nastiest problem an API can face is under-specification so we should strive hard to avoid it.

APIs that use abstract classes, or that expect the user to implement the API interfaces rather than just call them, have the additional burden of defining the parent-subclass contract with as much, or perhaps more, rigor than the standard caller-callee API contract – some articles refer to this case as the SPI (Service Provider Interface) [NetBeans]. The

[7] Event ordering is of huge importance to the API clients, but most APIs fail to say anything about it. "Eclipse quality" APIs don't make this error.

clients of the API must be able to create subclasses *without* looking at the source code of the super class. Even though Eclipse is open source, a high-quality API should not require the client to read the source code before creating a subclasses: all the subtle characteristics of "you must call super when overriding this method" and "you must call one of these N initialization methods", etc. must be part of the API.

Good APIs describe where the API is expected to grow in the future. A trivial mistake that all freshman CS students make is forgetting to put a version number in their file formats – a good API does not make that same mistake. An "Eclipse quality" API must leave room for future growth and it must describe to the clients how to accommodate that growth. For example, an API that specifies an integer return as "either X or Y" is unnecessarily restricting its future growth. A better API would say "one of the set of N values, in this release, N is X or Y" provides both room and guidance on growth.

Is It An Acceptable API Change?

As projects continue release after release, the APIs are certain to evolve so we need to make sure that they evolve in a controlled and acceptable fashion. One technique is to introduce parallel APIs (the Interface, Interface2, Interface3 technique) [des Rivieres 2001]. Nevertheless, the fundamental questions of API evolution are: first, did you break any promises, and second, did you weave the new API into the story?

If the new API breaks promises made in earlier APIs, that breakage must be raised as early in the release cycle as possible so that the clients are aware and have time to adapt. Some questions to consider about breaking changes include: What promises were broken? Who gets hurt?[8] Why do they need to get hurt? What other solutions were available, and what were their pain-gain tradeoffs?

Both [NetBeans] and [des Rivieres 2001] have good discussions of what changes are acceptable, and what changes are breaking changes, when evolving APIs and SPIs.

A high-quality API tells a story. It should hang together as a coherent whole and it often includes sequences and examples that explain how it works. When new API is added, the best solution is to weave that change into the story. Branching and options are often used to explain the old solution versus the new solution – something like "the old is a default implementation of the new, more flexible solution". A poor change is just tacked on to the side of the existing story without any attempt to explain to the clients how it fits in with the flow.

Strategies for *Eclipse Quality*

There are many risks to API quality in Eclipse projects, but one of the most common is the contribution of a large body of code. This can happen both early in a project's life as well as later in release N or even N+1. The problem for Eclipse is that the code typically works and works well (or else the project would not have accepted the contribution), but the code typically does not have an associated platform quality API. Rather than just publishing the existing API and committing the project to support those APIs without review, the recommended solution is to contribute the code to org.eclipse…internal…

[8] If you're lucky, there are so few (or none) actual users of the breaking change that nobody gets hurt.

packages[9]. Then, as the appropriately written and reviewed specifications and tests are produced, the now Eclipse quality APIs are moved from the internal packages into the full org.eclipse namespace[10]

There is a conflict between the thoroughness and maturity/stability requirements for platform quality APIs. There is also a conflict between these two requirements and commercial product cycle pressures. For *Eclipse quality,* we must choose maturity over thoroughness, and both over the product cycle. If insufficient front end time was applied to defining APIs in a given release cycle, projects have been known to release *internal provisional* APIs – intended to become an Eclipse quality API, but not having reached that state yet.[11]

Summary

Many issues have made Eclipse successful. One major reason for our success is the emphasis of high-quality APIs for extensible frameworks. The Eclipse technical leaders can maintain this reputation for quality both proactively – by committing significant resources to API development early in their release cycle, and reactively – by ensuring that the APIs being released are "Eclipse quality". Above all, we should keep in mind that "Eclipse quality" is about more than just code: it's about making and keeping explicit, clearly explained, well supported, promises to our users.

References

[Eclipse 2001] http://www.eclipse.org/

[Eclipse 2003] "Eclipse Development Process," http://www.eclipse.org/org/documents/Eclipse%20Development%20Process%202003_11_09%20FINAL.pdf

[Eclipse Naming] "Eclipse Platform Naming Conventions" http://dev.eclipse.org/naming.html

[NetBeans] "How to Design a (Module) API," http://openide.netbeans.org/nonav/tutorial/api-design.html

[des Rivieres 2001] "Evolving Java-based APIs", June 2001, http://www.eclipse.org/eclipse/development/java-api-evolution.html

[des Rivieres 2004] Jim des Rivieres, "Eclipse APIs: Lines in the Sand," EclipseCon 2004, http://www.eclipsecon.org/2004/EclipseCon_2004_TechnicalTrackPresentations/02_des_Rivieres.pdf

[9] Not org.eclipse…internal…api because that would be a contradiction. "API" is public code and "internal" is private code, so it would be impossible to be both internal and API at the same time.

[10] This has the desirable side effect of breaking code that relies on the internal packages. The reason this is desirable is because that code, when it is reviewed, strengthened, and moved to being API, probably now has different behavior than it did as internal code. Thus, clients of the internal code will need to ensure that their use and expectations are still correct.

[11] Projects shall not rely on this escape mechanism repeatedly as that would be counter to the principles of the Eclipse Foundation and the Eclipse open source projects.

[des Rivieres 2005] Jim des Rivieres, “API First,” EclipseCon 2005, http://www.eclipsecon.org/2005/presentations/EclipseCon2005_12.2APIFirst.pdf

[Smith & Kramer] Kevin Smith and Doug Kramer, “Requirements for Writing Java API Specifications”, developers.sun.com, http://java.sun.com/products/jdk/javadoc/writingapispecs/index.html