HOW TO BETTER BREAK SOFTWARE

FIXING SEMANTIC VERSIONING

Udo Hafermann
Todor Boev

ABSTRACT

- Semantic Versioning is the current standard for supporting software changes over time
- It also an integral part of the OSGi platform
- Over the years it has become apparent that although Semantic Versioning allows us to express breaking changes, it does not do a good job in supporting the actual transition from the old to the new
- The Semantic Versioning of today favors the parties that are slow to adopt the change, making life hard for everyone else
- We will demonstrate in a straightforward way what happens when we break software and discuss a surprisingly simple approach to fix Semantic Versioning so that it allows all parties (fast and slow) to proceed at their own pace.

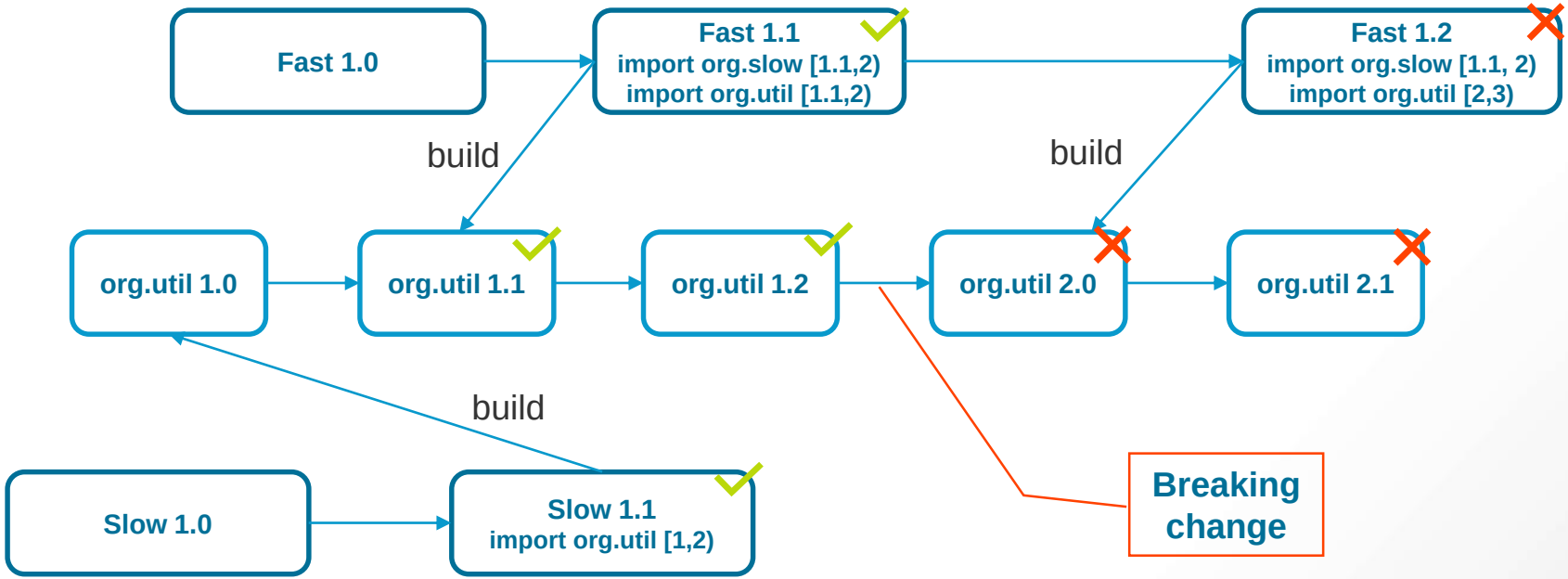
OVERVIEW

- Motivation
- Example
- Proposed convention
- Reasoning
- Revised example
- Findings
- Open ends

MOTIVATION

- We have found ourselves unable to introduce breaking changes even when using Semantic Versioning
- Consider org.osgi.framework;version=2.0
  - E.g. to get rid of Dictionary in APIs
- No good way to move consumers to new universe
- In fact, we have never introduced a breaking change in 20 years
  - Except DMT API, where we introduced a new package, starting at version 2
- JDK has also never broken a package
  - E.g. java.nio as successor to java.io
- E.g. log4j 2 as successor to log4j 1.x

EXAMPLE: CODE EVOLUTION



EXPECTATIONS

- Humans expect that semantic versioning alone solves all issues with software evolution
- How humans expect things to work
  - Introduce relative time by adding *semantic versions* to the names that languages use
  - Change introduced by *accretion*, not *replacement*
  - All built artifacts are there *for all time* for any consumer to bind to
  - Then the modular system can select one version of a name that is compatible with all deployed consumers
  - The module system can force everyone to eventually migrate so I work only against clean code
- How human expectations fail
  - Because *accretion is in fact violated w*hen a breaking change is introduced

BREAKING CHANGE (DEFINITION)

- The meaning of the name changes beyond the understanding of the current consumers. This is why it is called "breaking”
- Q: Why can’t the modular system reliably support a breaking change, even with semantic versioning?
- A: Principle of accretion is violated
  - Breaking change *replaces* the old meaning with a new meaning
  - Code binds to names from other pieces of code that evolve independently
  - Language can't cope: one compilation unit may have to be exposed to both meanings at the same time
  - This goes against the fundamentals of the language
  - No versions in the names/identifiers; no “import org.util.foo@2.0.0 as foo2”
- Modular systems are responsible for composing a consistent namespace
  - OSGi helps via “uses” constraints and class space isolation, but can not fix the fundamental issue (cannot break Java)

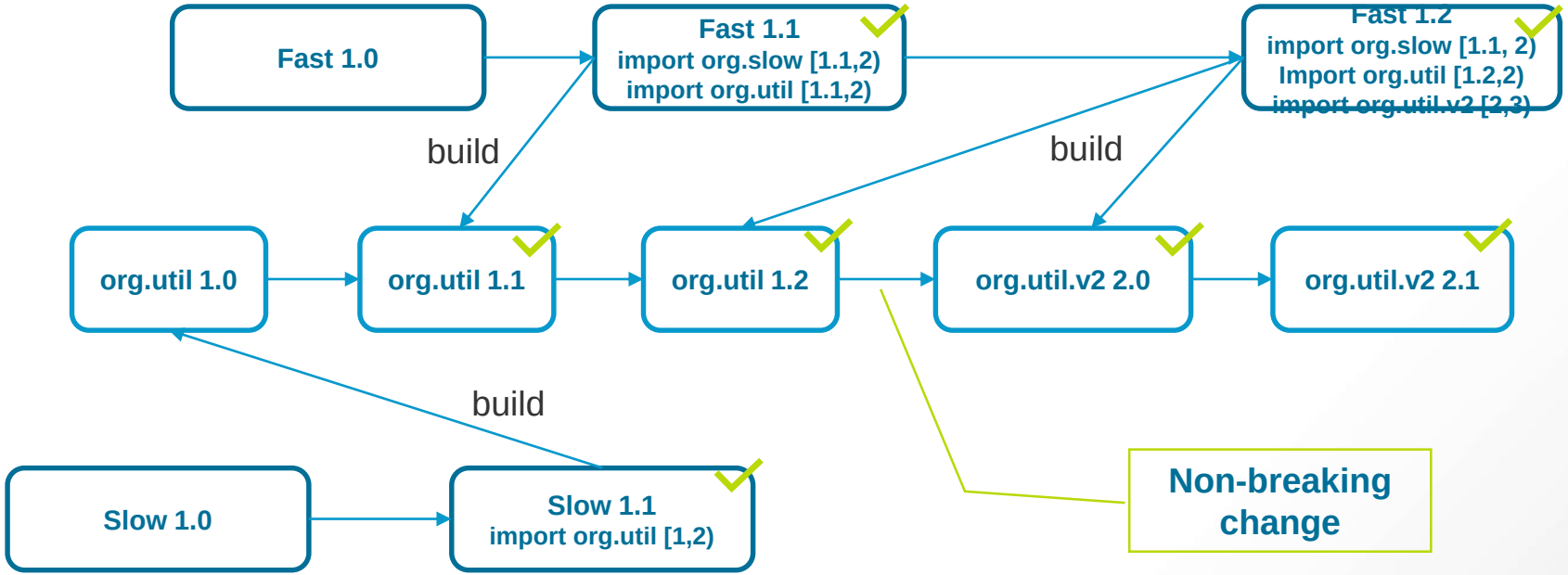
BREAKING CHANGE (SOLUTION)

- Accretion: Breaking change introduces a new name
- It is no longer “breaking”
- If there is need for a compilation unit to use both versions the language/compiler will *force them to*

PROPOSED CONVENTION

1. `org.example.foo; version=0.0.1`
   - The prototype period when the contract is first being defined
   - Package name is reserved
   - Breakage is allowed now and only now
2. `org.example.foo; version=1.0.0`
   - The contract and the package name become locked down
3. `org.example.foo; version=1.0.1`
   - The contract is not changed
4. `org.example.foo; version=1.1.0`
   - The contract is enriched, but not invalidated
5. *`org.example.foo; version=2.0.0`*
   - *ILLEGAL: The original contract is destroyed and replaced by something else*
   - *The relation between old and new becomes too abstract for the machines to cope: language, tools, etc.*
6. `org.example.foo.v2; version=2.0.0`
   - New contract bound to the new package
   - The bundle name and version reflect continuity

EXAMPLE: CODE EVOLUTION



OSGi Alliance

software AG
Freedom as a Service

FINDINGS (PRO)

- Practical migration to new version is possible
- Migration can in some cases be postponed forever
- Adapters can be used to re-implement old on new
  - One compilation unit built against both versions
  - Maintaining the adapter may be cheaper than maintaining the old version
- Maintainer of an API can release an adapter with the change
  - Stateful code needs adapters
  - Only one instance of the state can exist for both versions
  - E.g. Declarative Services and Configuration Admin are effectively singletons
- This addresses “uses” constraint violations
  - Where two incompatible contracts are consumed by one module

FINDINGS (CON)

- Looks odd to some people
- In the case of OSGi introduces additional complexity to the already complex versioning model
  - Closed version ranges still should be used by both consumers (‘[1.1, 2)’) and providers (‘[1.1, 1.2)’)
  - However the upper boundary in the consumer case is never violated
- Harder to track abstract continuity
  - Machines can no longer see the evolution into the new name, unless they themselves enforce the ‘org.example.foo.v2’ convention
  - Humans can choose bad new names that make it hard for other humans to understand the continuity

NEXT

OSGi endorses the idea

So evangelize it and make it palatable

OSGi Alliance

software AG
Freedom as a Service

FEEDBACK