Semantic Versioning

In the world of software management there exists a dread place called "dependency hell." The bigger your system grows and the more packages you integrate into your software, the more likely you are to find yourself, one day, in this pit of despair.

In systems with many dependencies, releasing new package versions can quickly become a nightmare. If the dependency specifications are too tight, you are in danger of version lock (the inability to upgrade a package without having to release new versions of every dependent package). If dependencies are specified too loosely, you will inevitably be bitten by version promiscuity (assuming compatibility with more future versions than is reasonable). Dependency hell is where you are when version lock and/or version promiscuity prevent you from easily and safely moving your project forward.

As a solution to this problem, I propose a simple set of rules and requirements that dictate how version numbers are assigned and incremented. For this system to work, you first need to declare a public API. This may consist of documentation or be enforced by the code itself. Regardless, it is important that this API be clear and precise. Once you identify your public API, you communicate changes to it with specific increments to your version number. Consider a version format of X.Y.Z (Major.Minor.Patch). Bug fixes not affecting the API increment the patch version, backwards compatible API additions/changes increment the minor version, and backwards incompatible API changes increment the major version.

I call this system "Semantic Versioning." Under this scheme, version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next.

Semantic Versioning Specification (SemVer)

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1. Software using Semantic Versioning MUST declare a public API. This API could be declared in the code itself or exist strictly in documentation. However it is done, it should be precise and comprehensive.

2. A normal version number MUST take the form X.Y.Z where X, Y, and Z are integers. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 < 1.10.0 < 1.11.0.

3. A special version number MAY be denoted by appending an arbitrary string immediately following the patch version. The string MUST be comprised of only alphanumerics plus dash [0-9A-Za-z-] and MUST begin with an alpha character [A-Za-z]. Special versions satisfy but have a lower precedence than the associated normal version. Precedence SHOULD be determined by lexicographic ASCII sort order. For instance: 1.0.0beta1 < 1.0.0beta2 < 1.0.0.

4. Once a versioned package has been released, the contents of that version MUST NOT be modified. Any modifications must be released as a new version.

5. Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.

6. Version 1.0.0 defines the public API. The way in which the version number is incremented is now dependent on this public API and how it changes.

7. Patch version Z (x.y.Z | x > 0) MUST be incremented if only backwards compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.

8. Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backwards compatible functionality is introduced to the public API. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes.

9. Major version X (X.y.z | X > 0) MUST be incremented if any backwards incompatible changes are introduced to the public API. It MAY include minor and patch level changes.

Tagging Specification (SemVerTag)

This sub-specification SHOULD be used if you use a version control system (Git, Mercurial, SVN, etc) to store your code. Using this system allows automated tools to inspect your package and determine SemVer compliance and released versions.

1. When tagging releases in a version control system, the tag for a version MUST be "vX.Y.Z" e.g. "v3.1.0".

2. The first revision that introduces SemVer compliance SHOULD be tagged "semver". This allows pre-existing projects to assume compliance at any arbitrary point and for automated tools to discover this fact.

Why Use Semantic Versioning?

This is not a new or revolutionary idea. In fact, you probably do something close to this already. The problem is that "close" isn't good enough. Without compliance to some sort of formal specification, version numbers are essentially useless for dependency management. By giving a name and clear definition to the above

ideas, it becomes easy to communicate your intentions to the users of your software. Once these intentions are clear, flexible (but not too flexible) dependency specifications can finally be made.

A simple example will demonstrate how Semantic Versioning can make dependency hell a thing of the past. Consider a library called "Firetruck." It requires a Semantically Versioned package named "Ladder." At the time that Firetruck is created, Ladder is at version 3.1.0. Since Firetruck uses some functionality that was first introduced in 3.1.0, you can safely specify the Ladder dependency as greater than or equal to 3.1.0 but less than 4.0.0. Now, when Ladder version 3.1.1 and 3.2.0 become available, you can release them to your package management system and know that they will be compatible with existing dependent software.

As a responsible developer you will, of course, want to verify that any package upgrades function as advertised. The real world is a messy place; there's nothing we can do about that but be vigilant. What you can do is let Semantic Versioning provide you with a sane way to release and upgrade packages without having to roll new versions of dependent packages, saving you time and hassle.

If all of this sounds desirable, all you need to do to start using Semantic Versioning is to declare that you are doing so and then follow the rules. Link to this website from your README so others know the rules and can benefit from them.

FAQ

How do I know when to release 1.0.0?

If your software is being used in production, it should probably already be 1.0.0. If you have a stable API on which users have come to depend, you should be 1.0.0. If you're worrying a lot about backwards compatibility, you should probably already be 1.0.0.

Doesn't this discourage rapid development and fast iteration?

Major version zero is all about rapid development. If you're changing the API every day you should either still be in version 0.x.x or on a separate development branch working on the next major version.

If even the tiniest backwards incompatible changes to the public API require a major version bump, won't I end up at version 42.0.0 very rapidly?

This is a question of responsible development and foresight. Incompatible changes should not be introduced lightly to software that has a lot of dependent code. The cost that must be incurred to upgrade can be significant. Having to bump major versions to release incompatible changes means you'll think through the impact of your changes, and evaluate the cost/benefit ratio involved.

Documenting the entire public API is too much work!

It is your responsibility as a professional developer to properly document software that is intended for use by

others. Managing software complexity is a hugely important part of keeping a project efficient, and that's hard to do if nobody knows how to use your software, or what methods are safe to call. In the long run, Semantic Versioning, and the insistence on a well defined public API can keep everyone and everything running smoothly.

What do I do if I accidentally release a backwards incompatible change as a minor version?

As soon as you realize that you've broken the Semantic Versioning spec, fix the problem and release a new minor version that corrects the problem and restores backwards compatibility. Remember, it is unacceptable to modify versioned releases, even under this circumstance. If it's appropriate, document the offending version and inform your users of the problem so that they are aware of the offending version.

What should I do if I update my own dependencies without changing the public API?

That would be considered compatible since it does not affect the public API. Software that explicitly depends on the same dependencies as your package should have their own dependency specifications and the author will notice any conflicts. Determining whether the change is a patch level or minor level modification depends on whether you updated your dependencies in order to fix a bug or introduce new functionality. I would usually expect additional code for the latter instance, in which case it's obviously a minor level increment.

About

The Semantic Versioning specification is authored by Tom Preston-Werner <http://tom.preston-werner.com>, inventor of Gravatars and cofounder of GitHub.

If you'd like to leave feedback, please open an issue on GitHub <https://github.com/mojombo/semver.org/issues>.