

**COMSATS INSTITUE OF INFORMATION AND TECHNOLOGY**

**Submitted To: Dr. Waqar Mehmood**

**Submitted By: Muhammad Awais**

**Section: FA-16 BSE – 4B 052**

**Subject: SE 2**

**Date: 13 -03-2018**

**ASSIGNMENT**

**Software Versioning:**

Software versioning is the process of assigning either unique *version names* or unique *version numbers* to unique states of [computer software](https://en.wikipedia.org/wiki/Computer_software). Within a given version number category (major, minor), these numbers are generally assigned in increasing order and correspond to new developments in the software. At a fine-grained level, [revision control](https://en.wikipedia.org/wiki/Revision_control) is often used for keeping track of incrementally different versions of information, whether or not this information is computer software.

**Summary:**

Given a version number MAJOR , MINOR , PATCH increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR , MINOR , PATCH format.

**Introduction:**

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. These rules are based on but not necessarily limited to pre-existing widespread common practices in use in both closed and open-source software. 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.

There are some software which are use for versioning .

**Concurrent Versions System (CVS)**

[CVS](http://www.nongnu.org/cvs/) has been around since the 80s, and has been very popular with both commercial and open source developers.

It is released under the GNU license, and uses a system to let users “check out” the code they are going to work on and “check in” their changes.

Originally, CVS handled conflicts between two programmers by only allowing for the latest version of the code to be worked on and updated. As such, it was a first come, first serve system where the user must publish changes quickly to ensure that other users haven’t beat them to the punch.

Pros:

* Has been in use for many years and is considered mature technology

Cons:

* Moving or renaming files does not include a version update
* Security risks from symbolic links to files
* No atomic operation support, leading to source corruption
* Branch operations are expensive as it is not designed for long-term branching

**Apache Subversion (SVN)**

SVN, or Subversion as it is sometimes called, is generally the version control system that has the widest adoption. Most forms of open-source projects will use Subversion because many other large products such as Ruby, Python Apache, and more use it too. Google Code even uses SVN as a way of exclusively distributing code.

Pros:

* Newer system based on CVS
* Includes atomic operations
* Cheaper branch operations
* Wide variety of plug-ins for IDEs
* Does not use peer-to-peer model

Cons:

* Still contains bugs relating to renaming files and directories
* Insufficient repository management commands
* Slower comparative speed

**Git**

Git is considered to be a newer, and faster emerging star when it comes to version control systems. First developed by the creator of Linux kernel, Linus Torvalds, Git has begun to take the community for web development and system administration by storm, offering a largely different form of control. Here, there is no singular centralized code base that the code can be pulled from, and different branches are responsible for hosting different areas of the code. Other version control systems, such as CVS and SVN, use a centralized control, so that only one master copy of software is used.

Pros:

* Great for those who hate CVS/SVN
* Dramatic increase in operation speed
* Cheap branch operations
* Full history tree available offline
* Distributed, peer-to-peer model

Cons:

* Learning curve for those used to SVN
* Not optimal for single developers
* Limited Windows support compared to Linux

**Mercurial**

This is yet another form of version control system, similar to Git. It was designed initially as a source for larger development programs, often outside of the scope of most system admins, independent web developers and designers. However, this doesn’t mean that smaller teams and individuals can’t use it. Mercurial is a very fast and efficient application. The creators designed the software with performance as the core feature.

Pros:

* Easier to learn than Git
* Better documentation
* Distributed model

Cons:

* No merging of two parents
* Extension-based rather than script ability
* Less out of the box power