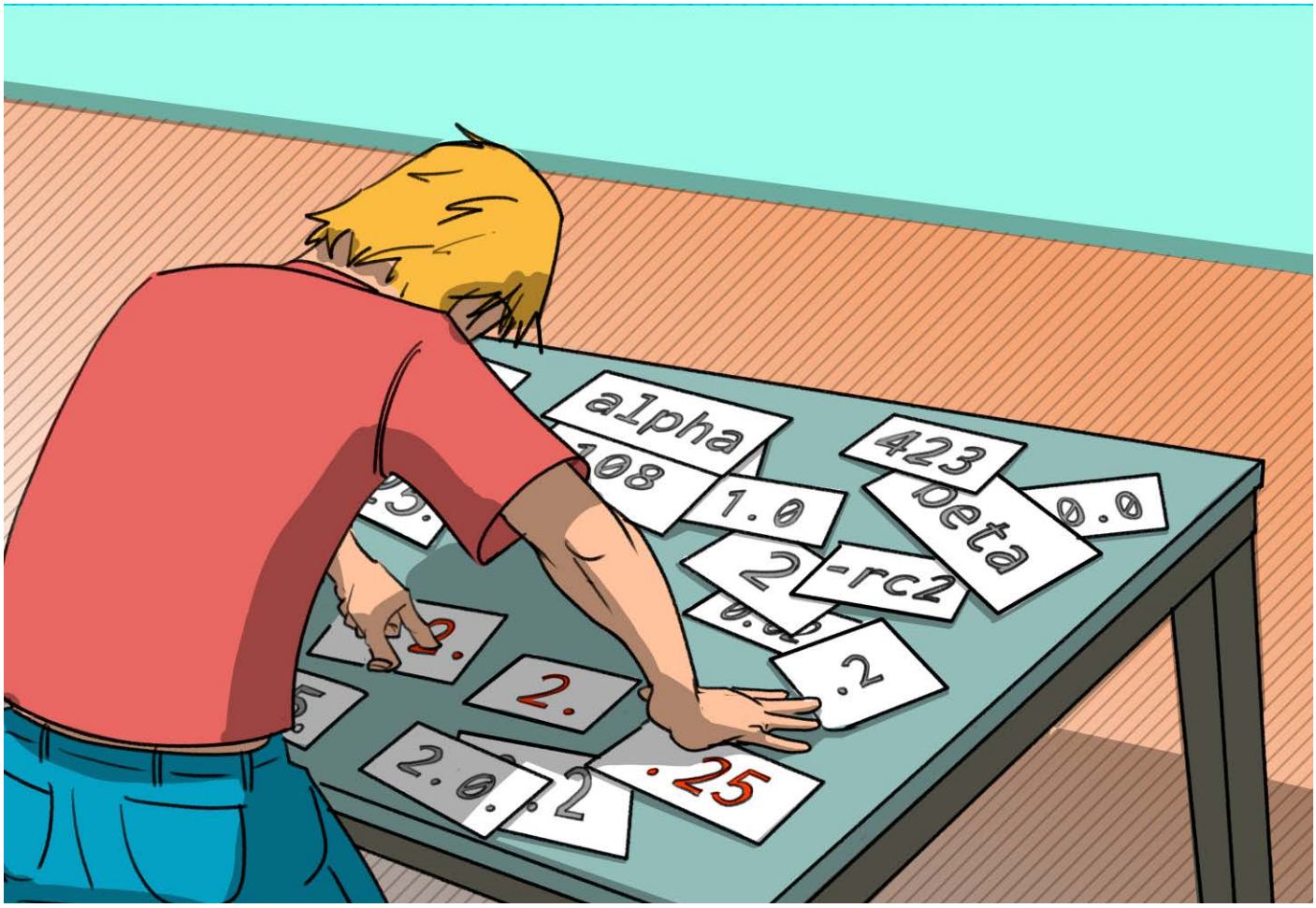


Bild: Thorsten Hübner

Bedeutung 2.0.0

Warum Versionsnummern nicht willkürlich sind

Software ist nie fertig und jedes Entwicklungsprojekt eine ständige Baustelle. Um Veränderungen und Verbesserungen zu dokumentieren, gibt es Versionsnummern. Welchem Schema sollten diese folgen und kommt nach Version 1.9 direkt die 1.10 oder gleich die 2.0? Semantic Versioning liefert klare Antworten.

Von Jan Mahn

Versionsnummern können mehr sein als eine technische Information über den Stand einer Software. Besonders runde Versionsnummern haben eine eigene Ausstrahlung: Als die Apple-Entwickler 2001 bei Mac OS die Version 10 erreichten, müssen sie sich in die Vollkommenheit dieser Zahl verliebt haben. Mac OS hieß von da an Mac OS X (also mit einer römischen 10) und machte fast zwei Dekaden keine großen Sprünge mehr. Von 2001 bis 2019 konnte Apple sich nicht von der 10 trennen und vergab nur die Versionsnummern 10.1 bis 10.15. Erst 2020 war die Zeit reif für einen großen Satz: macOS 11 mit dem Spitznamen Big Sur. Die Karriere der 11 fällt kurz aus, schon ein Jahr später, im Herbst 2021, folgt macOS 12. Auch Microsoft klebte lange an der runden 10 – 2015 brachte das Unternehmen Windows 10 auf den Markt und verkündete damals, dass dies das letzte Windows werden solle. Erst 2021 warf man den Plan über den Haufen und kündigte Windows 11 an.

Schaut man auf die Betriebssystemhersteller, könnte man den Eindruck gewinnen, dass Versionsnummern vollkommen willkürlich vergeben werden und die Marketingabteilung wesentlichen Einfluss auf die Nummerierung hat. Doch die beiden Betriebssysteme sind eher die Ausnahme. Unter Softwareentwicklern hat sich mittlerweile ein Versionsschema durchgesetzt, das wenig Raum für Willkür lässt: Semantic Versioning – also eine Versionierung mit Bedeutung. Dieses Schema

begegnet Entwicklern und Anwendern vielerorts: bei Updates von Desktopsoftware, Serverdiensten und Bibliotheken für sehr viele Programmiersprachen. Das komplette Regelwerk für Semantic Versioning (erschienen in Version 2.0.0) finden Sie über ct.de/ydw9.

Bedeutungsschwer

Versionsnummern nach Semantic Versioning (SemVer) bestehen aus drei Teilen, getrennt durch Punkte – etwa 1.2.5. Die drei Komponenten nennt man „Major“, „Minor“ und „Patch“ und für die Vergabe der Zahlen gibt es klare Regeln. Die erste Regel: Jede veröffentlichte Version (interne Tests unter den Entwicklern gehören nicht dazu) braucht eine neue Versionsnummer. Es dürfen also nie zwei unterschiedliche Entwicklungsstände unter einer Nummer im Umlauf sein, nachträglich wird keine einzige Zeile Code verändert, ohne eine neue Nummer zu vergeben. Führende Nullen in allen drei Komponenten sind nach Semantic-Versioning-Regeln grundsätzlich unerwünscht.

Die erste stabile Version, die produktiv eingesetzt werden kann, bekommt die Version 1.0.0 (zu Beta-Tests später mehr). Ist eine Software erst mal auf dem Markt, wird man früher oder später Fehler feststellen und diese reparieren wollen. Wenn ein Knopf in der Oberfläche nicht so arbeitet, wie man das erwartet hat, oder die Software unter bestimmten Umständen immer abstürzt, ist es Zeit für ein Patch-Release. Patches dürfen nur Probleme lösen, keine neuen Funktionen einbringen. Jede Patch-Version darf aber beliebig viele unterschiedliche Probleme beseitigen, wichtig ist nur, dass die bisherige Funktion nicht angetastet wird. So muss zum Beispiel eine mit der Vorgängerversion einer Software erstellte Datei weiterhin lesbar sein. Die erste reparierte Version erscheint unter Versionsnummer 1.0.1. Fixes kann es so viele geben, wie Bedarf an Reparaturen ist. Nach 1.0.9 muss also nicht 1.1.0 folgen – große Projekte, die regelmäßig aktualisiert werden, könnten problemlos etwa Version 1.0.423 herausbringen.

Neben dringend anstehenden Reparaturen kommen bei den Entwicklern meist auch Wünsche für neue Funktionen an. In einer Desktopsoftware kann das ein neuer Knopf sein, in einem API zum Beispiel neue Endpunkte. Sind solche Änderungen fertig, ist es Zeit für ein Minor-Release, im Beispiel wäre jetzt Version 1.1.0 dran. Sobald die Minor-Version erhöht wird, muss der Patch-Level unbedingt wieder auf 0 gesetzt werden. Zusammen mit neuen Funktionen kann man in einem Minor-Release beliebig viele anstehende Reparaturen erledigen, für die man sonst Patch-Releases veröffentlicht hätte. Wichtig bei der Vergabe von Minor-Releases: Sie müssen stets abwärtskompatibel sein. Die Nutzer bekommen durch ein Minor-Update nur Funktionen hinzu, müssen aber nichts umkonfigurieren, keine Dateien anpassen und sollten nichts vermissen. Minor-Updates sollte man wie Patches stets „gedankenlos“ installieren können, gerne auch automatisiert und nachts. Am nächsten Morgen darf es kein böses Erwachen geben.

Für die Vergabe von neuen Minor-Versionen gibt es noch einen weiteren Grund: Wenn man sich entscheidet, eine bisher existierende Funktion abzukündigen, ist eine neue Minor-Version Pflicht. In der Dokumentation und den Release-Notes wird dann der Hinweis ergänzt: „Folgende Funktion wird in Zukunft abgeschafft. Nutzen Sie sie nicht mehr und stellen Sie sich jetzt um.“

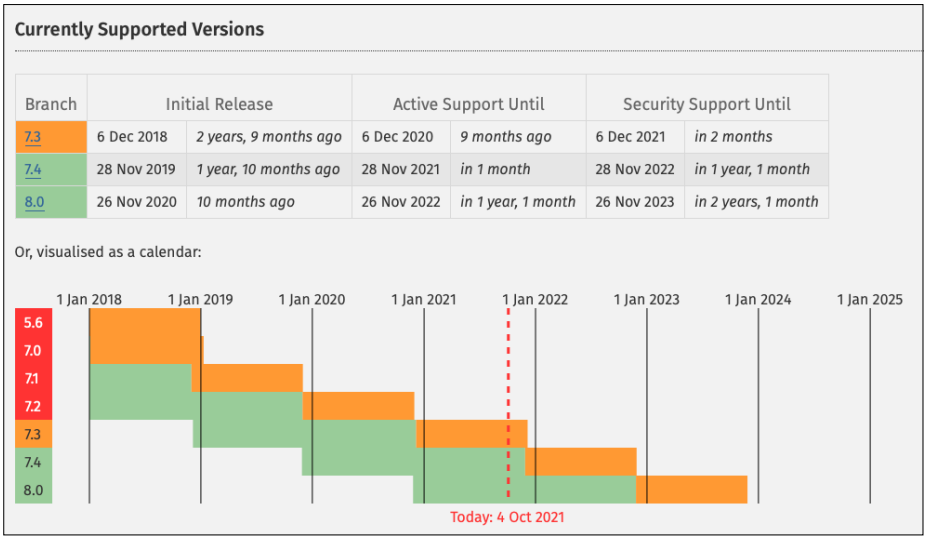
Große Brocken

Wann immer Änderungen „breaking changes“ sind, also nicht mehr abwärtskompatibel, braucht man ein Major-Release. Dabei werden ebenfalls Minor- und Patch-Version wieder genullt, von 1.23.456 stellt man also zwangsläufig auf 2.0.0. Neue Major-Versionen sind der Zeitpunkt, an dem man zuvor abgekündigte alte Zöpfe abschneidet und nach Belieben weitere Funktionen und Reparaturen einbaut. Der Sprung von Version 1.23.456 auf Version 2.0.0 darf beim Anwender durchaus Arbeit verursachen – eine gute Software hilft aber beim Wechsel auf die neue Major-Version. Haben Sie eine Dokumentenverwaltung entwickelt und beim Sprung von Version 1 auf 2 das Dateiformat geändert und nicht nur erweitert, könnte das Programm beim Öffnen einer alten Datei fragen, ob sie automatisch ins neue Format übertragen werden soll. Bei Serversoftware gehört in der Dokumentation ein Kapitel mit einer Umzugsanleitung zum guten Ton (ein sogenannter Migration Guide).

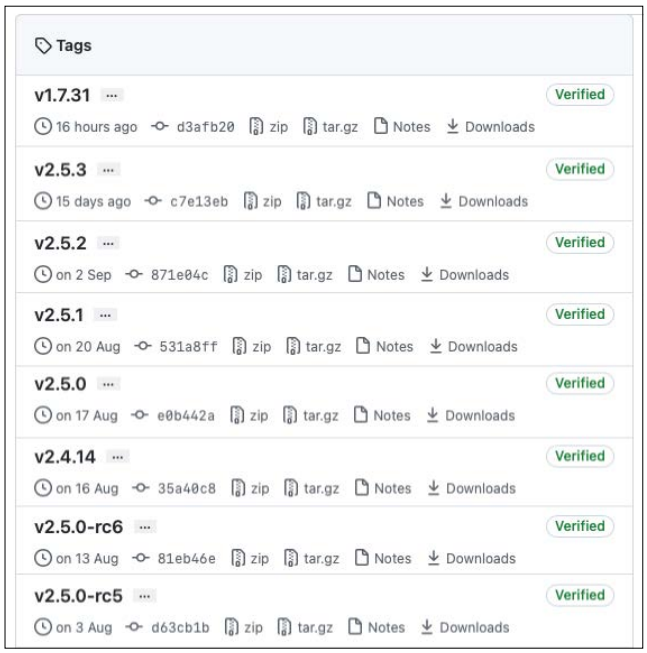
Nicht zwangsläufig endet mit dem Sprung auf eine neue Major- oder Minor-Version die Entwicklung an der Vorgängerversion – die Entscheidung, was weiter unterstützt wird, kann nur das Entwicklerteam treffen und das sollte seine Pläne frühzeitig bekannt geben. Unten sehen Sie einen Screenshot der Lebenszyklen der

c't kompakt

- Semantic Versioning diktiert klare Regeln, wie Versionsnummern vergeben werden.
- Viele große und kleine Projekte halten sich an die Regeln und machen Nutzern das Leben leichter.
- Wenn Entwickler schlampen, ist Chaos unvermeidlich.



Mit dem Erscheinen einer neuen Major-Version muss nicht zwangsläufig Schluss sein mit der Entwicklung an älteren Versionen. Das PHP-Projekt versorgt zum Beispiel mehrere Stränge parallel.



Sobald sich die Entwickler entscheiden, mehr als eine Version zu unterstützen, kann es in der Liste der Releases unübersichtlich werden. Sowohl 1.7.31 als auch 2.5.3 sind aktuelle Versionen – mit jeweils unterschiedlichen Funktionen.

Programmiersprache PHP. Anfang Oktober 2021, als dieser Artikel entstand, bekamen die Versionen 7.3 und 7.4 noch Updates auf Patch-Ebene, während am Versionsstrang von 8 bereits an neuen Funktionen geschraubt wurde (Version 8.1 befand sich bereits in der Testphase). Mehrere Stränge am Leben zu halten, ist durchaus arbeitsintensiv – viele Entwicklerteams veröffentlichen für die alten Versionen nur noch Sicherheits-Patches und keine, die nicht sicherheitsrelevante Fehler tilgen.

In Ruhe getestet

Ganz so einfach wie beschrieben ist der Lebenszyklus von Versionen meist nicht. Zunächst ist es utopisch, dass das Leben einer Software mit einer einsatzbereiten Version 1.0.0 beginnt – in Wahrheit fängt alles mit einem leeren Ordner an und geht mit vereinzelten Fragmenten von halbfertigem Code weiter. In der internen Entwicklung fängt man meist mit Version 0.1.0 an und zählt zunächst die Patch-Versionen hoch, bis die Anwendung etwas Gestalt angenommen hat. Die 0.2.0 kommt, wenn der erste intern gesetzte Meilenstein erreicht ist. Von den strengen SemVer-Regeln kann man zu diesem Zeitpunkt noch etwas abweichen – sie gelten erst ab Version 1.0.0. Auch ohne Major-Releases dürfen sich Details hier jederzeit und ohne An- und Abkündigung ändern.

Eine weitere utopische Annahme ist es, dass ein Sprung von einer Minor- oder Major-Version zur nächsten reibungslos und ohne ausprobieren klappt. Während man Reparaturen auf Patch-Level meist direkt veröffentlichen kann, weil die Eingriffe überschaubar sind, empfiehlt sich für neue Funktionen und große Änderungen eine gründliche Testphase. Dafür braucht man Vorabversionen, die ein Kreis von Freiwilligen ausprobiert. Solche Pre-Releases kennzeichnet man nach Semantic Versioning, indem man eine Zeichenkette mit einem Bindestrich anhängt, etwa 2.0.0-alpha. Wer eine so markierte Version installiert, muss mit Problemen rechnen und sollte sie den Entwicklern melden – das ist der Sinn einer Testphase. Pre-Releases werden untereinander alphabetisch sortiert, strenge Regeln für die Benennung gibt es aber nicht. Erlaubt sind nur ASCII-Zeichen, also keine Umlaute und Sonderzeichen: Nach 2.0.0-alpha kann man zum Beispiel 2.0.0-alpha2 veröffentlichen, später 2.0.0-beta, dann etwa 2.0.0-rc1. „rc“ steht für Release Candidate. In diesem Status sind keine weiteren Funktionen mehr zu erwarten. Verliefen alle Test zufriedenstellend, ist die Zeit reif für die finale 2.0.0.

Im Alltag

Wer Semantic Versioning in freier Wildbahn sehen will, muss nur die Release-Seiten großer Open-Source-Projekte bei GitHub öffnen – zum Beispiel die des HTTP-Routers Traefik unter github.com/traefik/traefik/releases. Als dieser Artikel entstand, waren Version 1.7.31 im alten Entwicklungszweig und 2.5.3 im neuen Zweig aktuell. Version 2.5.0 hatten die Entwickler zuvor gründlich bis 2.5.0-rc6 in Pre-Releases getestet.

Intensiven Gebrauch von Semantic Versioning machen Paketmanager diverser Programmiersprachen. Bei Softwarebibliotheken gelten dieselben Spielregeln wie für fertige Programme.

Paketmanager nutzen typischerweise eine Datei, in der man die Abhängigkeiten der eigenen Software einträgt, bei Npm heißt sie zum Beispiel package.json. Sie liegt auf der obersten Ebene eines Projekts und enthält im Abschnitt `dependencies` die verwendeten Bibliotheken, zum Beispiel folgendermaßen:

```
"dependencies": {
  "express": "4.17.1"
}
```

Damit ist die Abhängigkeit auf den Patch 4.17.1 festgenagelt, von Reparaturen würde man auch bei einem Update per `npm update` nichts mitbekommen. Möchte man Neuerungen auf Patch-Ebene bekommen, schreibt man `~4.17.1`. Die Tilde verrät, dass es mindestens diese Version oder ein höheres Patch-Level sein muss.

Wer alle Minor-Releases mitbekommen möchte, verwendet die Syntax `^4.17.1`. Alle neuen 4er-Releases kommen dann per `npm update`. Das setzt aber etwas Vertrauen voraus: Nicht immer schätzen Entwickler von Bibliotheken ihre Änderungen richtig ein. Was sie für „non breaking“ halten, kann schlimmstenfalls doch negative Auswirkungen haben. In der Vergangenheit führten solche Fehleinschätzungen immer mal wieder zu Frust, der sich in Issues bei GitHub entlud. Bevor Code mit aktualisierten Abhängigkeiten den Nutzer erreicht, sind Tests dringend angeraten. Wer ganz mutig ist, trägt schlicht `>=4.17.1` ein. Anders als Semantic Versioning ist die Syntax von Paketmanagern nicht vereinheitlicht. Composer für PHP verhält sich ähnlich wie Npm, NuGet kennt andere Steuerzeichen. Details finden Entwickler in der Doku ihres Paketmanagers.

Kein Entwicklerteam ist gezwungen, seine Software nach Semantic Versioning zu benennen – das Schema hat aber mittlerweile so große Verbreitung gefunden, dass man gut beraten ist, ihm zu folgen. Sobald sich Entwickler bewusst Gedanken machen müssen, welche Änderungen sie in einer Patch- oder Minor-Version unterbringen, müssen sie auch ihre Entwicklung anpassen und arbeiten meist automatisch mehr im Sinne der Nutzer. *(jam@ct.de)* **ct**

***Semantic Versioning 2.0.0:* ct.de/ydw9**