Patterns for Software Release Versioning

Klaus Marquardt
Email: pattern@kmarquardt.de

Copyright by Klaus Marquardt. Permission granted for the purpose of EuroPLoP 2006

How to version software releases may be an afterthought during development, but they have all the potential to make your life miserable once the software is in production.

This paper covers techniques to identify a particular version, policies to determine version compatibility, and release update strategies. It aims to help the project participants responsible for releases. The affected roles are software architect, release manager, project lead, and product manager. In small projects these roles may be covered by one or two persons.

Introduction

While the software is planned and designed, you think of plans and processes, specifications and delivery dates, architecture and middleware, test and integration, deployment and installation. Versioning releases is often treated as an afterthought. Sure, software is versioned, so what is the point?

There are two points: complexity in size, and in time. All but the smallest systems are a combination of distinct programs or libraries, software developed by different teams and eventually running on different machines. The interoperation of all this code needs to be assured. Over time, the installed base of a software may become significant, and each installation needs to be maintained. Bugs are detected and fixed, new features developed, and the installed base becomes inhomogeneous.

You now enter the domain of not just version identification, but of version interoperation and release management. Which programs can be installed together to function properly, and how is the compatibility checked? And some time more upstream, what is the right granule and time for software items to release?

Luckily, most software systems can survive without deep thoughts on versioning. Implicit and tacit knowledge can bring you so far, but adding just a little more complexity can break your system and requires urgent and careful action. This collection of patterns aims to make the versioning issues explicit, prepare you for the foreseeable, and help you decide what amount of thought to spend when.

Contents

During the course of writing this paper, I found that most of its concepts were less obvious to readers than I thought. The side discussion typically included notes on the kind of systems the readers were building, and what kind of versioning scheme they would use.

This valuable discussion is reflected in the paper structure: it starts with forces to consider, and takes these forces to the different application domains.

Forces

The first thing to consider is the identification of a release. It shall be simple and straightforward, it shall be catchy, it shall reflect the purpose of the release, and it shall contain sufficient information to return to the development environment and possibly restore the entire source code base.

Besides this, software releases and their versioning strategy face some more challenges that are closely related to the application domain.

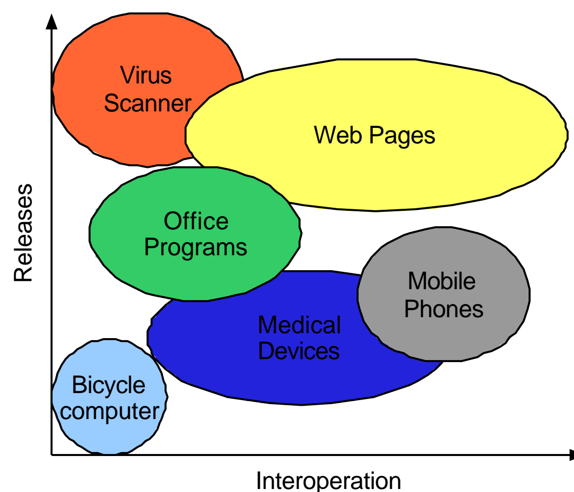
Releases are expensive. They require not just the development effort, but need to be tested, installed, and maintained. The costs for these activities might be much higher than the initial development costs. Thus, a common force is to minimize the amount of releases.

On the other hand, releases are hard to get right. Packing releases is always a compromise of timing, bringing the software to the market timely versus releasing quality software that contains all features and no bugs. A common force is to build many releases, possibly to release each functional enhancement and each bug fix separately.

The other key force to versioning is the amount of interoperation required. Any checks for compatibility are tedious and expensive, especially when there are many different software components involved. Thus a common force is to minimize the amount and the depth of checking to a bare minimum during installation.

However, most software lives in large systems with interfaces to many more software systems. The increased complexity, the combinatorial explosion of different versions coming together, needs to be managed for testing, installation, and at run time. Especially when a fine granular interoperation is required, a compatibility check is best performed with each single invocation to enable valuable functionality whenever possible.

Context



The patterns for release versioning live in different contexts, and their applicability varies. The above graphic spans the spectrum of application domains and shows some examples.

As their second context, the release and versioning strategies needs to consider different phases in the software lifecycle: tests, sales and acquisition, installation, run time interaction, and maintenance. For all situations, the identification of the software version is needed. During installation and at run time, version interoperation needs to be checked.

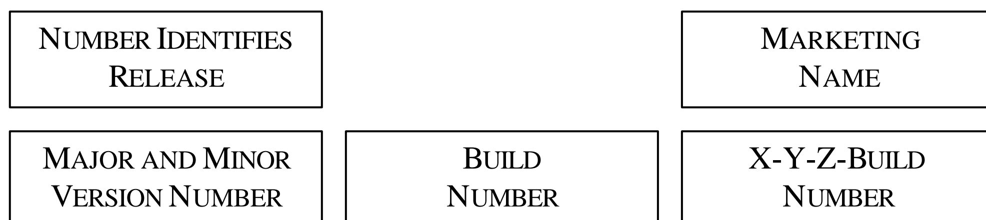
The kind and purpose of the software release is covered by the alternative patterns FUNCTIONAL RELEASE, PATCH RELEASE, and TEST RELEASE. These are a precondition to this pattern collection, listed in the appendix.

Roadmap

The patterns in this collection have relations to the outside world. Release versioning is virtually pointless without the ability to re-construct any version of the installed base. Configuration management patterns [*Berczuk*] describe some of the essential development practices and processes.

Especially for patch releases and partial releases, many more policies and practices are documented, e.g. in [*Hohmann*]. However, these are based on combinations of patterns from this collection.

Different policies of version numbers all base on NUMBER IDENTIFIES RELEASE: MAJOR AND MINOR VERSION NUMBER, BUILD NUMBER, and their combination X-Y-Z-BUILD NUMBER. For the public image, numbers are often replaced or complemented by a MARKETING NAME.



Release versions are also used for compatibility checking, either explicitly or implicitly. The common policies for an automated check at installation or run time are MAJOR VERSION COMPATIBILITY, WHITE LIST CHECK, and BLACK LIST CHECK. They can be combined, or replace each other depending on the software product life cycle.

Further patterns in the domain of distribution and installation need to link with these release versioning patterns. Distribution comprises shrinkware, downloads, and auto installation via internet or radio or digital TV. Installation may require user confirmation and interaction, or take place silently.

Examples

Bicycle computer: The bicycle computer has no software update strategy, the releases in the market will not been seen again for maintenance. Thus, there is no checking involved. A MARKETING NAME is given to the software, or rather to the entire system containing the software. This name, e.g. "BC600", reflects the number of features available to the end user.

Medical Devices: the modules that dosage volatile anaesthetica in the Zeus machine can be plugged by the user. Since they contain electronics and firmware, their version needs to be checked against the Zeus contained software version. During development tests and clinical trials, interoperation was assumed the standard and only versions know to be incompatible were refused. Then, a BLACK LIST CHECK was in place. Before shipment to the end customer, the black list was replaced by a WHITE LIST CHECK to ensure that only combinations that had been thoroughly verified and released for clinical usage, would become operational.

Office Programs: The internal version identification of Microsoft Outlook is irrelevant to the end user, the most visible identification is the MARKETING NAME, e.g. "Outlook 2003". Only in the case of problems, the administrator needs more information. The help display exhibits a number similar to the X-Y-Z-BUILD structure.

Medical Devices: most medical devices are connected to exchange data via a standardized protocol. Since these protocols are enhanced as the clinical knowledge grows, they are versioned using a MAJOR AND MINOR VERSION NUMBER. Protocol changes that violate some of the previously valid assumptions are expressed increasing the major version number. At runtime, compatibility is checked using a MAJOR VERSION COMPATIBILITY check.

Medical Devices: depending on the critically of clinical operation, a MAJOR VERSION COMPATIBILITY based on protocol versions may not suffice. Each end point of the protocol may change specific behavior related with receiving protocol messages, and the entire system behavior also depends on the implementation versions. These are checked using a WHITE LIST CHECK mentioning only the explicitly tested and certified combinations.

Operating Systems: the Microsoft Windows operating system has a prominent MARKETING NAME indicating how recent the release is: "1998", "2003", etc. When PATCH RELEASES become necessary, they are combined into SERVICE PACKS which also use a MARKETING NAME, e.g. "XP service pack 2". A BUILD NUMBER is also available for bug reporting, shown at startup or with a crash report.

Virus Scanner: the Avira Antivir virus scanner distinguishes versions for the search engine, and for the virus definition file. Both are expressed in a X-Y-Z-BUILD style, e.g. "V7.00.00.17", "V6.34.01.189".

Number Identifies Release

Consider a software product or component release.

All users need a means to refer to a particular release. How do you identify that release?

Forces

Selecting names can be cool, just as your project name is, but a sense of humor seldom scales for different (team) cultures.

Each release is created at a particular date, but exposing the date may induce an impression of staleness, even if an unchanged version means quality.

Any number may do, but you need a sense of which version is newer than another.

Solution

Therefore, identify the release by a unique version number. Increase the number with every release.

Make sure that each release gets its own number, and that numbers are not reused. Do not slip even in exceptional circumstances, like one time creation for your very special customer, or a trade show presentation. Also ensure that even test and trial releases are versioned – every release that might possibly leave the privacy of the development team.

Make this software version number visible on the shipping media and in the software itself. The software should support an API function to retrieve its version number.

Improved

Each release is uniquely identified without confusion.

Each release found in some place can be referenced and reconstructed in source code.

The version information does not exhibit purpose or contents, thus diminishing the potential that users take offense.

Consider

From knowing the official release number, your development environment should enable you to re-create the entire sources. Patterns on software configuration management are available for your support [*Berczuk*].

Plain numbers are boring. They cannot serve for marketing purposes.

Combine this with MARKETING NAMES and create a one-to-one relation between the version number, and the combination of MARKETING NAME and PATCH RELEASE / SERVICE PACK.

BUILD NUMBERS are a way to distinguish releases for tests and trials from those releases available to the public.

Major and Minor Version Number

Consider a software product or component release.

A single number to identify a release version can serve as a reference and identification. However, how could you convey hints on key features or compatibility?

Forces You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Any number may do, but you need a sense of which version is newer than another.

You want to signal major achievements, but the product still remains the same, covering the same users' needs.

You want to signal minor advances and corrections, but this shall have a different scope than major achievements.

Solution **Therefore**, identify all versioned items with a major and a minor version number. Use positive integers for both.

Increase the minor version number with each release, except when you increase the major number in which case the minor number starts with 0.[1] Increase the major number with major achievements, or to indicate incompatibility.

The major and minor version number strategy is a tricky beast despite its popularity. It evokes associations with respect to compatibility and advances that may not hold. It is never totally clear when to change a major version number, and what implications this has.

When using this numbering scheme, it is tempting to abuse it as a marketing vehicle. Mixing these will cause confusion, so decide whether you focus on marketing aspects, or on compatibility. Technically, it is best applied when you are serious about compatibility and do not need to market the software directly, such as in embedded systems where the software is just among other ingrediences.

Improved The numbers you use are more expressive now.

Consider Using the major and minor version numbering scheme is mostly linked with the MAJOR NUMBER COMPATIBILITY. Beware that compatibility needs to be checked, and that the amount of possible combinations grows with the square of minor version numbers.

The assumed expression is by convention only, and cannot be extrapolated into the future.

For alternatives to express technical and marketing aspects, see X-Y-Z-BUILD NUMBER and MARKETING NAME.

[1] Hence the proverb: "never buy a version dot-0!"

Build Number

Consider a versioned product that applies MARKETING NAME.

How do you reference the status of the internal development?

Forces

You need to identify a software version, but that identification is not related to features or their marketing.

You distribute versions of the software for testing and trials, but only a few of these will become official releases.

The version identification shall be visible to the end user for reference, but that identification shall not transport any expectation whatsoever.

Solution

Therefore, maintain a numerical build number and store this number within the software itself. This Build Number is kept in addition to a MARKETING NAME or a MAJOR AND MINOR VERSION NUMBER. In a system consisting of multiple versioned software items, each has its own Build Number.

Make this build number meaningless and visible. Meaningless implies that you will not consider this information in a compatibility check, though it will appear in a Bill Of Material [*Berczuk*]. Visible means that any user of the software can see it and reference it in a mail or during a phone call.

Test and trial versions may have a life beyond your expectation, and distribute themselves in unexpected places. Maintain a database with the Build Numbers that left the development team, and document their release status.

Improved

Build Numbers reference a snapshot of the development process, without any implications beyond that the software has been build.

This reference can be exhibited and exchanged in any situation.

Consider

Creating a build number adds to the complexity of your tool suite.

BUILD NUMBERS are most effective when created automatically. The build process should include incrementing that number and linking or packaging it with the software. A new number can be given anytime and should not be based on quality criteria beyond that the build is a complete one. To ensure it does not transport expectations, use a number that is not related to a date.

X-Y-Z-Build Number

This is a variant to and combination of MAJOR AND MINOR VERSION NUMBER, BUILD NUMBER, and MARKETING NAME.

A major-minor version number evokes associations about features and compatibility. However, how could you actually convey this information?

Forces

You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Any number may do, but you need a sense of which version is newer than another.

You want to signal major achievements, but the product still remains the same, covering the same users' needs.

You want to signal minor advances and corrections, but this shall have a different scope than major achievements.

Solution

Therefore, identify all versioned items with a major and a minor version number, plus a patch level and a build number. Use positive integers for all four parts.

Increase the minor version number with each release that enhances functionality, except when you increase the major number in which case the minor number starts with 0. Increase the major number with major achievements, or to indicate incompatibility.

Increase the patch level when you indicate full compatibility and identical functionality, but a change due to bug fixes. Use the build number as a technical reference to your configuration management system.

Improved

The numbers you use are more expressive now.

X-Y-Z-BUILD NUMBER can support both marketing and technical needs, and thus replace or complement a MARKETING NAME.

Consider

Using the major and minor version numbering scheme is mostly linked with the MAJOR NUMBER COMPATIBILITY. Beware that compatibility needs to be checked, and that the amount of possible combinations grows with the square of minor version numbers.

The assumed expression is by convention only, and cannot be extrapolated into the future.

Marketing Name

Consider a versioned product.

A release with a mere version number is boring, it may even evoke negative associations. How can you brand a release for marketing and give positive hints to potential customers?

Forces

You need a way of identification that is quickly graspable, but that still conveys a hint of information.

Associations with a software release shall be positive, but a major-minor combination is not creating trust.

You want to signal major achievements, but marketing is not interested to signal minor advances or corrections.

Solution

Therefore, use a marketing name for shippable versions. This can be an integer number indicating your long history ("10g"), or relating to your intended shipping date ("Windows 1995"). Make sure that each marketing name has a match into an actual software version.

Finding brilliant marketing names is an art which software engineers are typically not good at. Continuity, novelity, or advancements are not expressed in the same wording and domain as base line numbers and compatibility concerns.

To match one view into the other, a simple technical number is sufficient, such as a BUILD NUMBER. The match should not use a MAJOR AND MINOR VERSION NUMBER as this is more complex than necessary and transports information that somebody needs to put in – and neither marketing nor development is interested.

Improved

Names can be more expressive than numbers.

The domains of technical issues and marketing issues are separated.

Consider

You need a mapping between both domains.

Mixing a marketing name with a numerical identification can create a mess, when users expect features or compatibility based on the numbers, that the actual software does not provide.

Combine the MARKETING NAME for major advances with a different strategy to release bug fixes and corrections, like PATCH RELEASE / SERVICE PACKS.

Note that the MARKETING NAME does not need to match the project name, even though the project name may be known to the public prior to the shipping date.

Major Version Compatibility

Consider a system of several components. The application needs to ensure that it checks for the involved versions and only becomes operational with compatible combinations.

Which combination of versioned items can be considered compatible?

Forces

You cannot assume that any combination of software interoperates smoothly, but users implicitly expect compatibility of updates and new versions.

Similar versions can cooperate, but major differences in features will limit the compatibility.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

The compatibility of different software items better be stated explicitly, but the number of possible combinations grows non linear.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

Solution

Therefore, use the MAJOR AND MINOR VERSION NUMBERS for compatibility checking. Assume all versions of the same major version number compatible, regardless of their minor version.

While this assumption is easy to check by the installed software components, it is hard to achieve during development. All possible combinations must be tested, at least in a pair wise approach. Except where legally required, you do not need to test triangle and more complex settings, as their failure probability is very low.

Improved

The compatibility check is technically easy and can happen at run time.

The decision on compatibility is easy to understand and matches the users' expectations.

Compatibility matches to similar features, bug fix releases are considered compatible.

Consider

There is no guarantee that only thoroughly tested combinations of software become operational.

The decision about compatibility is typically not based on thorough testing of all combinations, but on implicit assumption.

The MAJOR VERSION COMPATIBILITY strategy does not scale for very many minor releases, especially when a large number of items is involved. It is often combined with a strategy to limit the number of minor releases to less than a hand full.

White List Check
also known as: Positive Check

Consider a system of several components. The application needs to ensure that it checks for the involved versions and only becomes operational with compatible combinations.

How can your application check for version compatibility, when you cannot rely on the implicit understanding of a MAJOR VERSION COMPATIBILITY?

Forces

You cannot assume that any combination of software interoperates smoothly, but users implicitly expect compatibility of updates and new versions.

Similar versions can cooperate, but major differences in features will limit the compatibility.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

The compatibility of different software items better be stated explicitly, but the number of possible combinations grows non linear.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

Solution

Therefore, maintain a list that includes tested combinations of different items. Check whether the actual versions are listed as compatible. Reject interoperation when the combination is not listed.

The POSITIVE CHECK strategy comes with the understanding that every combination that is not explicitly mentioned as compatible, is assumed to be incompatible. This emphasizes the importance of testing and verification and prevents unintended slips in the release procedure.

The information which versions cooperate can be maintained outside the software and have independent distribution channels. In systems where some parts are movable or exchangeable, the allowance information might not include the actual combinations although they have been tested already. Thus, an update channel for the allowance data is as important as the distribution of the software itself. If fraud prevention is essential, the allowance information needs to be encoded.

Improved

The compatibility check is technically easy and can happen at run time.

Only thoroughly tested combinations of software can become operational.

The release process is not bound to the development but to QA.

Consider

Installations done by end users have a higher risk to refuse operation.

A second type of data needs to be distributed.

Black List Check
also known as: Negative Check

Consider a system of several components. The application needs to ensure that it checks for the involved versions and only becomes operational with compatible combinations.

How do your application conveniently check for version compatibility, when most available versions will be interoperational?

Forces

You can assume that most combination of software interoperate, but you need a way to exclude those combinations that you know to be incompatible.

You can determine the compatibility during development, but a run time check increases the chances to identify conflicts.

You want to be able to incrementally update particular software portions, but you need to test all combinations that can occur.

The compatibility of different software items better be stated explicitly, but for just trying something in your own environment you avoid significant administrative overhead.

Solution

Therefore, assume compatibility, and define those cases where you know that versions do not interoperate. List these failure cases in a negative list, and check for their appearance.

The NEGATIVE CHECK strategy comes with the understanding that every combination that is not explicitly mentioned as incompatible, is assumed to be compatible. This reduces the importance of formal testing and verification and enables an informal release procedure by the developers.

Improved

The compatibility check is technically easy and can happen at run time.

Software to be tested can quickly become operational.

Software known to be erroneous in combination, can be excluded from operation.

Consider

There is no guarantee that only thoroughly tested combinations of software become operational. Continuous partial updates in real life will bring together untested combinations that are not excluded by a black list. Note that installations done by end users have a higher risk to malfunction, possibly even unnoticed.

As the NEGATIVE CHECK strategy can lead to unexpected behavior when not carefully controlled, depending on the usage in the field it needs to be replaced by a more strict checking strategy.

Acknowledgements

Thanks to Wolfgang Zuser for shepherding these patterns to EuroPLoP 2006. Further thanks to Neil Harrison for shepherding an earlier version, and to the workshop participants at VikingPLoP 2005 for their feedback.

References

Berczuk Stephen Berczuk, Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley 2002

Hohmann Luke Hohmann, Beyond Software Architecture: Creating and Sustaining Winning Solutions. Addison-Wesley 2003

Kelly Allan Kelly, Business Strategy Patterns for Selling Knowledge with Products and Services. To appear in: Proceedings of VikingPLoP 2005

Appendix: Patterns for Release Purposes

Functional Release

Consider a company developing a software product.

What status of the software should be delivered to the customer?

Forces Preparing a working software baseline for release requires effort in testing, packaging, marketing, and distribution, but releasing software is the key business of most software companies.

Premature shipping distracts users and may risk a vendors business, but late shipment diminishes the value for the customer and the return on investment.

Solution **Therefore**, deliver a new software release when a major functional gain has been achieved.

Finding the balance between the costs associated to a new release, and the revenues at stake when deferring a release and thus costs, is an art in itself. It requires a combination of market knowledge, and financial controlling. The development team members contribute their knowledge of the time and effort it takes to finalize a release for shipment.

Improved Major achievements become available to the customer.

Premature releases that would increase the cost of ownership over the revenues expected, can be avoided by a cost analysis.

Consider Functional releases need identification. When a combination of numbers is used, increase the numbers to express the amount of novelty. An increase of a major version number indicates not only major improvements, but typically implies incompatibility to past releases.

For marketing purposes, consider using a Marketing Name to improve the mental identification of the new release.

The business model of the company may combine software releases with further opportunities to sell services or consultancy [*Kelly*].

Patch Release

Consider a versioned product that applies MARKETING NAME.

How do you distribute updates that solve problems of a current software release, and identify the distributed versions?

Forces You need to distribute corrections, fixes, and features that are overdue, but marketing fixes is different from marketing new products.

Associations with a software release shall be positive, but admitting bugs in a previous version is painful.

You need a way of identification that is quickly graspable, but that is not confused with new key versions and products.

Solution **Therefore**, ship updates that can be installed on the previous version of the software. Identify each Patch Release by a number, as there might be several of them necessary over the software's lifetime.

The (minor) version numbers of the updated software items need not be visible to external customers. However, document the Patch Releases so that the development team can make the mapping and reconstruct the entire source on demand.

Variant: combine several fixes into a Service Pack and ship further fixes related to this Service Pack. Service Packs reduce installation costs and the variety of installed versions to consider. Subsequent Service Packs should not require one another, but each should include all updates of the previously released Service Packs.

Since you have an interest that your customers install the Service Packs, make the entire process of distribution and installation as painless as possible. Give the Service Pack for free, using multiple distribution channels such as CD or internet download. Do not acquire consumer data that they might not want to give you. Announce each new Service Pack visibly through all channels of customer relationship management.

Improved Patch Releases are an elegant way to fix problems on the client's side.

Your reputation can change for the better, as you actually do care for the installed base.

Consider You need to maintain distinct branches of development to separate fixes from new features for future releases.

To avoid unnecessary installation, the software release should visibly display not only its version but also the latest installed Patch Release or Service Pack.

Patch Releases and Service Packs follow the same considerations and processes, but can have varying granularity. For a terminology, typical Service Packs are large and contain several different patches.

Test Release

Consider a company developing a software product. The product needs to be tested in real life situations.

How can you identify a release intended for testing, and ensure that it will not be confused with the final product?

Forces Developing software is expensive, but shipping software that does not meet the markets needs means spending even more money.

Releasing software is expensive, but only released software has the potential to be evaluated for usability and appropriate functionality.

Solution **Therefore**, prepare a small number of subsequent test releases, and mark them as test releases both for technical identification as well as in a user visible way.

For technical identification it is common to use a MAJOR AND MINOR VERSION NUMBER with zero as minor version number. For user visibility, a test release is typically called "alpha" or "beta".

Improved Functional releases for a broad audience are thoroughly tested.

Market acceptance and financial success is more likely.

Consider Even test releases are actual releases that are visible to some of your key customers. They are worth some marketing as they contribute to the public image of the final software product.

For early test releases, you may want to track where they are tested and used, and prevent unnoticed installation.

Collect feedback for each test release. Only pulblish a subsequent test release after the feedback has been included into the software development. Make sure that a part of the test users stay in the test release distribution list, so that you have a consistent feedback on your ability to react.