Patterns for Software Release Versioning

KLAUS MARQUARDT

Software developers rightfully focus on the activities needed to polish the software for its first release. The management of future releases, version identification, compatibility checks, and update strategies are typically treated as an afterthought, but insufficiencies or inconsistencies here have all the potential to make your life miserable once the software has hit the market.

These patterns cover practices to identify a particular version, policies to determine version compatibility, and release update strategies. They aim to make the versioning issues explicit, prepare the project for the foreseeable, and help decide what amount of thought to spend. They target the project participants responsible for releases: release manager, project leader, software architect, and product manager.

Categories and Subject Descriptors: D.2.9 [Software Engineering]: Management—Life cycle; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Version control; D.2.11 [Software Engineering]: Software Architectures--Patterns

General Terms: Design, Management

Additional Key Words and Phrases: Release, Pattern, Versioning, Build, Distribution, Installation, Compatibility.

1. MANAGEMENT SUMMARY

To avoid expensive mistakes, you need to know the essential practices for software release versioning. Following these will enable you to update the software as needed, and to solve problems once they hurt.

(1) From the very start, add some version number to your program and data. This can be simple number. Such a number is your start to enable identification, checking, and migration. If you omit this, you lose all opportunity to add these facilities later.

(2) Use an archiving system to store all source code needed to create the software, application data, and tests including test data. Otherwise you will not be able to evaluate bugs reported for some past version.

(3) Uniquely mark each software build that you might show around, that possibly leaves the privacy of the development team. Such a mark is called a “build number”, and it needs a clear relation to your version archive. This is your foot in the door of TECHNICAL RELEASE IDENTIFICATION including bug tracking, compatibility and interoperability testing, and the ability to know what you have given to whom.

(4) If you build software systems with multiple components that need to interoperate, make sure that each component can report its version using an API. Take care that this API is the most stable portion of the entire system, design it to be as long-lasting as your company.

(5) Hand the following pages to your technical team. The patterns below help to explore possible solutions.

2. 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 interoperability of all this code needs to be assured, and the interpretation of its stored data items must be clear. Over time, the installed base of a software application may become significant, and...
each installation needs to be maintained. Bugs are detected and fixed, new features developed, old
versions and data will be migrated, and the installed base becomes inhomogeneous.

Your development team will enter the domain of not just version identification, but of version
interoperation, and change 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?

Most software projects have some ideas and implicit knowledge on versioning. However, changing
team members or key customers, or adding some more complexity than you are used to, can break your
current approach and requires timely 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.

3. ROADMAP
The patterns in this collection cover three key areas of software release management: release purpose,
release identification, and release distribution.

The first set of patterns describes the kind and purpose of software releases.
A FUNCTIONAL RELEASE is a release that is available to end users; this pattern summarizes typical
reasons for a software release.
PATCH RELEASES and SERVICE PACKS update already shipped software, typically to fix a problem.
TEST RELEASES are releases for a limited audience to gather feedback before creating a functional
release.

Several patterns help with the release identification.
TECHNICAL RELEASE IDENTIFICATION is the starting point for version identification, with many known and
common variants.
MAJOR AND MINOR VERSION NUMBER is a very common strategy for a technical identification, combining
two numbers to distinguish the properties of different releases.
MARKETING NAME complements the technical numbering scheme with a name that attracts customers
and gives a quick reference.

For the distribution and installation of new versions several patterns are available. Distribution
comprises shrink-wrapped packages, downloads, and auto installation via internet or digital broadcast.
PULL DISTRIBUTION is the standard where the software user takes care to get the appropriate version.
PUSH DISTRIBUTION becomes useful when timeliness is an essential property of the software or the
specific release, and when administrators need to ensure that many users work with identical software
versions.
EMBEDDED DISTRIBUTION, a euphemism for NO DISTRIBUTION, is the common case with embedded
software systems; still most of them support a MANUAL DOWNLOAD for the technically initiated.
VERSION SERVER is useful where many systems need to be supplied with updates in parallel.

These patterns 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 2002)
describe some of the essential development practices and processes. Especially for patch releases and
partial releases, more policies and practices are documented, for example by (Hohmann 2003). These can
be constructed from and combined with patterns from this collection.

Many of the patterns, or their combinations, have an effect on the business model of the software.
Considerations are what kind of releases and services a customer would be willing to pay for (Kelly 2012) –
and whether customers can be forced to update their installation, to reduce the variety of installations an
administrator needs to know and support.
4. PATTERNS

4.1 FUNCTIONAL RELEASE

A company develops a software product. What status of the software should be delivered to the customer?

Preparing a working software baseline for release requires effort in testing, packaging, marketing, and distribution, and overly frequent releases are costly for the vendor and disruptive for many customers, but customers expect improved software at a convenient innovation rate.

Premature shipping distracts users and may risk a vendor’s business, but late shipment diminish the customer value and the return on investment.

Deliver a software release when a clear consistent set of functions is available and mature.

Many triggers exist that can initiate a new functional release, but the above solution needs to be applied in all situations. Finding the balance between the costs associated to a new release, the revenues expected from early market access, and the revenues at stake when shipping immature software, is a virtue in the middle ground of financial controlling and wizardry. Both, quality and timely support of customer expectations, are essential to business success.

Variant: Innovation Release. In many situations the first product launched can gain the largest market share, so new, innovative functions deserve an immediate release. Often customers have been waiting for an announced functionality and are eager to update their installation — and might even choose to switch to another product if the innovation is overly delayed. However, premature release can have more downsides than gains. The trust into the product decreases, the brand reputation shrinks, and bugfix measures directly diminish the financial gain. An INNOVATION RELEASE works best with a defined test for maturity, specified well in advance so that the release team cannot compromise the test criteria.

Variant: Improvement Release. The world is not perfect, and neither is an INNOVATION RELEASE. While some release does not contain any real bugs, some of the new features are not usable in the foreseen way or demand a change in the program’s workflow to increase acceptance. Such an IMPROVEMENT RELEASE may be worth its costs by keeping the customers satisfied and improving the reputation compared to the market competition.

Variant: Cleanup Release. Complexity can trigger a release when previous releases had needed many fixes, or there are many different versions of the software running in different locations and need still be supported. The financial controlling is involved whenever the current complexity of software releases and combinations in the field overwhelm the release effort, when most developers are involved in emergency maintenance and the testing of all combinations has stalled all true improvement and innovation. A CLEANUP RELEASE denotes the end of refactoring and redesign in development, and an end to the field support of previous versions. A clear cut towards a stable baseline can help to overcome the maintenance effort.

Major achievements in innovation or improvement become available to the customer.

Premature releases that would increase the cost of ownership over the revenues expected are avoided.

Functional releases need identification, both technically and for reputation and marketing. Apply a TECHNICAL RELEASE IDENTIFICATION, and a MARKETING NAME if appropriate.

The business model of the company may combine software releases with further opportunities to sell services or consultancy (Kelly 2012).
4.2 Patch Release

A versioned software product is already established in the market. How do you distribute updates that solve problems of a current software release, and identify the distributed versions?

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.

Ship updates that can be installed on top of the previous version of the software. Identify each Patch Release by a number that refers to the Functional Release, as there might be several of them necessary over the software’s lifetime.

The detailed version numbers of the updated software items need not be immediately visible to external customers. However, the numbers of all installed patches need to be available to end users that decide about further patches installation. Document the Patch Releases so that the development team can make the mapping and reconstruct the entire source on demand.

Variant: Atomic Patches. Bug fixes that have a limited scope and are independent from other fixes can be shipped and installed separately, in any order to the user’s preference. When some patch cannot remain atomic for technical reasons, consider issuing an Improvement Release, or a Service Pack.

Variant: Successive Patches. Patches are implemented in the quickest way if they may build upon one another and require the previous patches to be installed first. This technique frees the development from constraints, but requires thoughtful administration.

Variant: Service Pack. After multiple patches, a combination of several fixes as a Service Pack eases the users’ administration. For the vendor’s convenience, all support activities could refer to a specific Service Pack number, and all future fixes relate to it. 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 effortless and convenient as possible. Give the Service Pack away 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 your customer relationship management.

Patch Releases are a way to fix problems on the client’s side. Your reputation can change for the better, as you visibly care for the installed base.

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 its version and the latest installed Patch Release or Service Pack.

Patch Releases and Service Packs follow the same considerations and processes as Functional Releases, but at a varying granularity. Their business case is the avoidance of loss in money and reputation.
4.3 TEST RELEASE

A software product under development 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?

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.

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 major version number. For user visibility, a test release is typically called “alpha” or “beta”.

Keep in mind that even TEST RELEASES are actual releases that are visible to some of your key customers. They require attention and care since they shape the image of your application for some of its key customers. Furthermore, they consume much valuable time and need to make good use of it. While a full verification cycle like with a FUNCTIONAL RELEASE might appear too much, some essential elements of the total verification measures need to take place. Finally, TEST RELEASES are worth some marketing as they contribute to the public image of the final software product.

Collect feedback for each TEST RELEASE. Only publish a subsequent test release after the available feedback has been included into the software. Make sure that some test users stay in the test release distribution list throughout all test releases, so that you have a consistent feedback on your ability to react.

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

TEST RELEASES are distinguished from FUNCTIONAL RELEASES meant for broad publication. They can be tested in a realistic setting by a limited number of users, without risking proliferation through unawareness.

Include some MARKETING NAME element into the TEST RELEASE to make it obvious to accidental users that this version is not a fully functional and supported version.

4.4 TECHNICAL RELEASE IDENTIFICATION

In many maintenance scenarios, software project participants and end users need a means to refer to a particular release. How do you identify that release?

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

Any release that might possibly leave the privacy of the development team needs identification for further reference.

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 expectations or promises.

Selecting names can be cool, but many kinds of humor do not scale for different (team) cultures.

Any number may do, but you need a sense of which version is newer than another.
Identify the release by a unique number. Increase the number with every release. Link the version number to your source configuration management system.

Make sure that each release gets its own number. Do not slip even in exceptional circumstances, like one time creation for your very special customer, or a trade show presentation. Also ensure that test and trial releases are versioned – preferably every build gets some unique identification.

In a system consisting of multiple versioned software items, each item maintains its own version number. Keep track of the items and numbers, their release status and their compatibilities. For combinations of items released together, maintain a Bill Of Material (Salecker and Schütz 2004) for packaging and package installation.

Make this software version number visible before the download or on the shipping media, and in the software itself at run time. In interoperating systems, each software portion should support an API function to retrieve its version number programmatically.

Each release is uniquely identified without confusion. Each release found in some place can be referenced and reconstructed in source code. 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 2002).

The reliable release identification prevents you from receiving reports of bugs without reference to a reproducible version, or risk malfunctions in the field without a possible removal.

Build numbers reference a snapshot of the development process, without any implications beyond that the software has been build. This build process reference can be exhibited and exchanged in any situation.

The version information does not exhibit purpose or contents, diminishing the potential that some user takes offense.

Plain numbers are boring. They can hardly be used for marketing purposes.

Creating a build number and embedding it into the software adds to the complexity of your tool suite.

MAJOR AND MINOR VERSION NUMBERS are a convenient way to provide technical identification. In addition to this mostly feature driven versioning scheme, maintain a numerical build number and store this number within the software itself.

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.

The following version numbering conventions denote common industrial practice. Though not each of these suits every project, most projects use many conventions from this list.

*Combine three numbers plus build number.* Four pieces of information can conveniently transport compatibility, features, fixes, and SCM reference. Any mixing or implication would enable misunderstandings.

*Major number indicates key feature sets.* The first number of a version numbering scheme indicates the feature set of the release, and a major advancement to previous releases of the same software.

*Major number indicates compatibility.* The first number of an X-Y-Z-Build Number scheme indicates the compatibility of this release to other releases of the same software, or to other components within the software system.

*Minor number indicates features.* Further advances that are added to an already shipped releases are indicated at the second level, the minor version number. This would be features that the software vendor lists at a detailed description but still in the user manual.

*Minor number indicates bug fixes.* When using only a two number scheme, MAJOR AND MINOR NUMBER, or when key features are visibly changed, the minor number also indicates bug fixes. In this case, the minor number is not expressive by itself.

*Third number indicates bug fixes.* Releases that only differ at the third numbering level are expected to be identical in user visible behavior, but higher numbers indicate a patch level or bugs fixed.

*Version number includes build number.* In addition to a two- or three-number-version name, an automated build number can be added to ease the technical reference to the SCM system.
Build number is tool generated. The build number is created automatically during the build process, and not manually chosen. It could be an integer that is calculated from date and time, but mostly it is a plain number incremented with each build.

Numbers do not contain dates. Visible dates in a version number could change while the development team is still busy creating that particular release. Furthermore, few systems want to indicate to their users how old the installed version actually is.

Test releases have MARKETING NAMES. Their name indicates the test level, e.g. alpha or beta. In addition, the test releases also have a numerical identification. The major and minor numbers are chosen as with the intended final version, but test releases have a smaller third number.

4.5 MAJOR AND MINOR VERSION NUMBER

How could the version identification convey hints on key features or compatibility?

You need a way of identification that is quickly graspable for the end user, and that conveys some information about the release.

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.

Identify all versioned items with a major and a minor version number. Use unsigned integers for both.

Follow the unstated rules of software release numbering in industry:
(1) Start with a major number 0 for test and evaluation releases;
(2) The first version for public use, that receives support, is called 1.0;
(3) Increase the major number with major achievements, or to indicate incompatibility;
(4) Increase the minor version number with each FUNCTIONAL RELEASE;
(5) When you increase the major number, the minor number becomes 0.

The two numbers you use are more expressive than a single number or a short name, and still convenient for quick reference by the end user.

Applying the unstated rules, advances and corrections are announced and interpreted correctly. The MAJOR AND MINOR VERSION NUMBER strategy is a tricky beast despite its popularity. It evokes associations with respect to advances or compatibility (Marquardt 2006) that are interpreted differently from different perspectives and software creators.

This numbering scheme has an implicit element of marketing to it. For technical reference, it is often accompanied by two more numbers. For alternatives to express marketing aspects, see MARKETING NAME.

Using the major and minor version numbering scheme is mostly linked with the MAJOR NUMBER COMPATIBILITY (Marquardt 2006).
4.6 Marketing Name

How can you brand a release for marketing and give positive hints to potential customers?

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

Associations with a software release shall be positive, but a major-minor combination is not necessarily creating trust, especially with very low version numbers.

The goal is to signal major achievements, but marketing is not interested to signal minor advances or corrections.

Use a marketing name for shippable versions. Similar to consumable products, the marketing name aims at emotions, not at technical detail.

The marketing name typically becomes clear sometime during the development project. Often the initial project name is used, sometimes a marketing initiative is started to find a brand name. The name needs certification by an accepted authority – depending on the culture of the users’ community, this may be a product manager or key development guru.

Make sure that each marketing name has a match into an actual software version stored in configuration management. 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.

Names can be more expressive than numbers. The certification separates team and technical issues from marketing issues if necessary. However, you need a mapping between both domains.

Mixing a marketing name with a numerical identification could mislead, when users expect features or compatibility based on that number 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.

4.7 Pull Distribution

How can you distribute new software versions to the software users?

There is a new software version available, but the previous version is in use by many customers.

The vendor would like to limit the number of versions in the field, but the users want to decide on their software updates themselves.

Announce the availability of updated versions to all users, and let them order or download the update on their own schedule.

The producer makes the software available, together with a description and a motivation why the clients would want to update their installation. The software may even contact a vendor server frequently, learn about new versions, and announce them actively to the user.

However, the decision whether, when, and how to migrate to a more recent software version is up to the user or a local administrator. The ownership of the installation is with the client side, not with the producer side.

The vendor can take make an upgrade attractive by maximizing the information (quality and quantity) about the new version, and by minimizing the effort the user has to invest. Occasionally even negative measures are in place: users of a version older than some period run out of support, unless they upgrade to a more recent version. The support policy must legally prepare for this cut-off scenario.
These procedures and practices are frequently used:
(1) Announcements about the software update to registered or online users;
(2) Announcements to mailing lists and in publications;
(3) Announcements given by the installed software itself;
(4) Distribution starts in a one-click fashion from the announcements;
(5) Distribution failure is detected and corrected by the software itself;
(6) Successful distribution leads into installation seamlessly.

**Variant: Automatic installation.** The distributed software comes with installer facilities, or the installation program is already available with the previous software installation. After successful distribution, the installation and activation starts immediately without further notice.

**Variant: User triggered installation.** The installation and activation of the distributed software is a separate step that requires user confirmation. This is the more common variant which leaves the user in control for all important administration steps for his environment.

**Variant: User specific distribution settings.** The system supports user adjustable configuration items that control whether the download / distribution and the installation and activation of updated software versions happens without further user interaction, or with confirmation of individual steps. This is a common setup with PULL DISTRIBUTION when the updates are mostly fixes that need to be applied very timely.

The availability of new versions is known, and measures are in place to remove old versions from the field. The administrative decisions are with the users' side. They can use the new version at their own discretion. The pull model allows for complex update scenarios that require many user interactions. It is the only model that allows that the current installation might be temporarily deactivated.

The PULL DISTRIBUTION is broad applicable, only in domains that require very timely updates a PUSH DISTRIBUTION approach might be superior.

### 4.8 PUSH DISTRIBUTION

**How can you distribute new software versions to the software users?**

There is a new software version available, but the previous version is in use by many customers. The vendor would like to limit the number of versions in the field, but the users want to decide on their software updates themselves.

**Make the updated version availability to all users by distributing it to their environment.**

Make sure that the user installations are typically connected to a distribution mechanism, e.g. the internet or a dedicated VERSION SERVER. The computer storage needs to be sufficiently large so that the entire executable code fits twice into it, in case the PUSH DISTRIBUTION fails to work properly. The automatic distribution should cause minimal disruption of the users' workflow, in particular the system and network performance must not be consumed heavily. This can be achieved by factoring the distribution into a separate execution context (task, thread, process) that runs at a lower priority than the main application. Additionally, the storage for the distributed software can be different from the data storage the application needs to access during normal operation. The installation is often separated from the distribution itself; the combination is mostly implemented using one of the following variants.
Variant: **Automatic installation.** The distributed software comes with installer facilities, or the installation program is already available with the previous software installation. After successful distribution, the installation and activation starts immediately without further notice.

Variant: **User triggered installation.** The installation and activation of the distributed software is a separate step that requires user confirmation. This variant leaves the user in control of the key administrative activities for his environment.

Variant: **User specific distribution settings.** The system supports user adjustable configuration items that control whether the download / distribution and the installation and activation of updated software versions happens without further user interaction, or with confirmation of individual steps. This is a common setup with **PUSH DISTRIBUTION** to allow the user to balance the push policy's advantages and the administrative ownership.

New versions are distributed according to the needs of the software provider. Measures are in place to remove old versions from the field. The key administrative decisions are with the users’ side. They can use the new version at their own discretion. The push model has higher system and workflow requirements than the pull model. However, it allows a mostly undisrupted workflow, where updates are in active use with the next application start.

The **PUSH DISTRIBUTION** cannot be used in environments that need to be controlled by legal requirements, where environment changes need to be tracked and user education needs to be certified.

4.9 **EMBEDDED DISTRIBUTION** *(also known as: NO DISTRIBUTION)*

**How can you minimize the costs and efforts related to software updates?**

- New software versions are created as you gain experience and remove bugs, but previous versions have already been shipped and the installed software is technically hard to reach.
- The old software versions do deliver their expected functionality, but the installations not updated still require some customer support.

**Refrain from distributing software updates.** Individual product versions come with a specific software version installed; new software comes with a new product version.

Leaving out all facilities for software update is a great relief with respect to technical effort, including eventually additional hardware costs, and it reduces the effort in logistics tremendously. The software policy should explicitly state this limitation, or it should be clear from the outset that the software is not meant to be exchanged, like in many embedded devices.

This approach pays off if your costs of ownership, especially the customer support ranging from hotline support to a replacement for a reduced price, is considerably lower than the costs an update (including the technical facilitation) would be.

As soon as the software advances could be useful for future customers, create a new version of the entire product as in **FUNCTIONAL RELEASE**.

**Variant: User Maintained Update.** The firmware of many technical devices could technically be updated, but not initiated by the vendor. Since a failure in firmware update could break the device, no warranty is included with either the device or the firmware. Experienced users may want to use the offered updates in a **PULL DISTRIBUTION** model, and update their products manually.
Where the costs of updating would exceed the costs of other customer support measures, this solution is cost effective without being overly offensive to clients. Old released versions stay in the field and retain their usability and customer value. New releases are FUNCTIONAL RELEASES that provide additional value and come with a price tag for the customers.

4.10 VERSION SERVER (also known as: DISTRIBUTION SERVER)

How can you provide software releases for installation at a later time?

New software releases must be made available to end users, but their machines cannot be reached immediately from the vendors’ servers. The software vendor is interested in a PUSH DISTRIBUTION model, but the choice of administrative measures shall remain with the customer. Some setups require that a rollback to a previous version is possible, or that devices can be updated that happen to be inaccessible at a given time, and the allowance to update may be bound to mandatory process steps like education.

Provide the software at a dedicated server within the customer’s network, that each device can access.

A VERSION SERVER holds any software version that has been released for clients’ use, and provides them to individual devices on their request. The installation is initiated at the individual device, where usually the most recent update is offered. In case of failure during installation, the version server provides a suitable rollback.

The producer needs to care for technical implementation of the communication between distribution server and the individual devices, as well as for the maintenance of the version server. While the first is at the discretion of the vendor, the latter requires an administrative agreement between vendor and customer. However, mostly this is easier to achieve than to enable remote access to each individual device.

Who is the owner of a version server is a subject to the maintenance contract. For a full service contract offer, the vendor may supply this server and take care of its administration. Customers taking care of administration themselves may want to own the server and distribute the software releases themselves.

Having the software version available at a Version Server, does not relief the update manager from coordinating the software installation in all places, and to take care for data migration and a potential temporary operation of mixed versions.

Variant: Embedded Version Server. In a simplified environment, the version server could be included in the device itself. This is useful in scenarios where each device is remotely accessible, and where the user plugs together functional components. These pluggable components may not already come with a compatible software version, and could be updated prior to going operational. An EMBEDDED VERSION SERVER may be combined with a VERSION SERVER in the local network.

Variant: Installation at Startup. When many client machines need to run the same software version, e.g. after changes to a central database schema, it is most convenient when the application start also initiates a software update. This is often implemented using a browser based approach to client installations, where a plug-in detects an update and initiates the download immediately.

The vendor can push software updates into the responsibility of the customer, and make the software physically available to all effected devices. It is a defined point where the responsibility for updates can be passed, while the customer remains in administrative control. A VERSION SERVER allows to indirectly access and update machines that are not connected to the vendor’s server, and it enables dealing with different failure scenarios that could happen during installation.
5. KNOWN GAPS AND OMISSIONS

Many different aspects need to be considered with respect to versions and releases. Several further areas are in scope of the topic, but not included in this pattern collection.

A set of patterns describes approaches for compatibility checking, available in (Marquardt 2006).

MAJOR VERSION COMPATIBILITY declares interoperating applications of identical major version compatible.

WHITE LIST CHECK limits compatibility to those combinations that have been tested and explicitly declared compatible.

BLACK LIST CHECK also limits compatibility by excluding a number of component combinations. It is best combined with one of the above strategies.

Several patterns show how the user can access the version information, and what the best ways for access are during installation and at run time.

INSTALLATION FILE NAME or labeled media are the earliest ways to convey the software version.

STARTUP DIALOG is useful to transport either a marketing name, or major and minor number, or a full set of many identification items including plug-in versions as needed. It helps avoid misunderstanding, but needs to be complemented by some place accessible without timing constraint.

VERSION INFORMATION DIALOG, mostly located in the context of the online help area, can be used to detail version information to the end user in case of a service request.

A small set of practices help automating the inclusion of build identification into the delivered software package.

VERSION DATA FILE: The initial step of an automated build is to retrieve the baseline information, and write it into a file that is used by the application software to display version information. Thus it is ensured that build information is included in the Bill Of Material (Salecker and Schütz 2004), and that the user can access the correct version.

INTEGRATED COMPONENT VERSION NUMBERS: If the software consists of different components, the version numbers need to be collected together either at build time (if possible) or at run time (in pluggable systems).

IMAGE VERSION: While a software build is reproducible anytime from the source control system (except that today’s environments include a time stamp and build machine information), an image includes further steps of installation beyond the mere built, that might be essential for system behavior.

For the distribution and installation of new versions, further patterns are known. Firmware may not be updateable at all (like watches) or prepared for frequent updates with guaranteed fix in case of failure (like in space missions). Installation may require user confirmation and interaction, or take place silently.

SEPARATED BOOTLOADER enables a guaranteed distribution even when the system has become corrupted.

MANUAL INSTALLATION is executed by the user, an administrator, or an educated application specialist.

SILENT INSTALLATION supports a timely activation with least user interaction.

Frequently, the installation of incompatible versions needs to be detected and corrected, if not prevented. Some systems need to prevent incompatible combinations from becoming operational.

MONOLITHIC INSTALLATION: installing an entire system at once avoids all conflicts among different software items. In embedded systems though, the hardware version needs to be considered. Applicability is limited to systems that can be reached within one session, and that have no user replaceable or pluggable parts.

VERSION INFORMATION API: If compatibility of distinct components need to be checked at run time, all components need to provide their respective version numbers. This requires a stable API that remains unchanged with all product releases.

INCREMENTAL INSTALLATION CHECK: when each portion of a software system is installed individually, the installing program can check for the available software versions and determine compatibility. On failure installation is abandoned.
STARTUP COMPATIBILITY CHECK: software systems are started sequentially (as a technical necessity). Each software portion can check for the already started software component versions and determine compatibility. On failure the startup is abandoned, leaving a portion of the system operational.

STARTUP COMPLETION CHECK: a sequential system startup needs to be complemented by a completeness check. Each software portion needs to check whether the started software components are sufficient to remain operational. Otherwise, after a reasonable time a failure of operation is announced. The applicability of this pattern includes temporary disconnections from remote systems.

Many patterns describe solutions to increase the compatibility and interoperation. This set helps to find the right architectural granularity with systems of interoperating components.

MONOLITHIC PACKAGE: the entire software consists of one released component. Release and compatibility checks are the least complex, but incremental updates are not possible.

HOMOGENOUS VERSION PACKAGE. Each public release is adapted so that all contained components show identical version information, even though the development and build process may vary per component.

PROCESSOR IMAGE PACKAGE: The entire software running on one processor is packaged and released as a whole. Compatibility of included components needs to be ensured at build time. At run time, compatibility checks are limited to inter processor communications.

PROCESS AND LIBRARY PACKAGE: Each individual process running on a server or client machine has its own version number. Likewise, the libraries these share also bring their own version number. This approach allows for incremental updates, especially when components have different change reasons and rates, but requires many different compatibility checks – or an application that is trusted without extensive checking.

DATA SCHEME PACKAGE: The structure of the stored and exchanged data is versioned independently. This is useful when different applications have common access to a database, or communicate via predefined data items. Applications using the same data scheme version are considered compatible. Updating applications is simple, upgrading the data definition requires extensive changes.

INDIVIDUAL MESSAGE PACKAGE: The structure of each exchanged data item is versioned independently. Similarly, the interface of each public service is published independently. This is useful in situations where many legacy applications need to remain operable at the same time as newly featured applications arise. Example: telecommunication networks.

Several patterns are known to deal with data that stems from other software versions than executed at the moment. Depending on the usage scenarios, compatibility issues occur back and forth, at run time or at installation time. Data migration patterns have been documented in (Rüping 2010).

Patterns for installation also need to check whether the combination of components (executables and data, and potentially operating systems and hardware items) is approved to become operational. Eventually, such an approval might be revoked in case of severe errors reported. Information about approval validity and a possible installation downgrade to an approved combination needs to be provided.

6. ACKNOWLEDGEMENTS

Special thanks to Tim Wellhausen for his thoughtful and thorough shepherding of these patterns to EuroPLoP 2010. Thanks to Neil Harrison and Wolfgang Zuser for shepherding earlier versions, and to all workshop participants at EuroPLoP 2010, EuroPLoP 2006, and VikingPLoP 2005.

REFERENCES


