VŠB TECHNICKÁ
UNIVERZITA
OSTRAVA

www.vsb.cz

SWI 3

David Ježek
VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

References

- Presentations from Jan Kožusznik
- AIELLO, BOB, 2010. Configuration Management Best Practices: Practical Methods that Work in the Real World. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional. ISBN 978-0-321-68586-5. Slaids form jan Kožusznik

Six Functional Areas

- Source code management
- Build engineering
- Environment configuration
- Change control
- Release management
- Deployment

The Traditional Definition of Configuration Management

- Configuration identification
- Change control
- Status accounting
- Configuration audit

Outline of Configuration Management

- Six functional area of configuration management
- Architecture and Hardware Configuration Management
- Compliance, Standards, and Frameworks

Six functional areas

- **Source code management**
- Build engineering
- Environment configuration
- Change control
- Release management
- Deployment

Source Code Management (SCM) - Goal

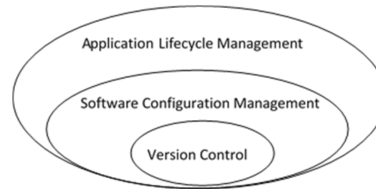
- "Good one starts with making certain that all of your source code is safely locked down and no important source code is lost."
- Another goal is to help improve the productivity of tour entire team – it can import the quality of source code by helping to implement automated testing.
- Provide traceability – one of the most important goal.

SCM - Principles

- Code is locked down and can never be lost
- Code is baselined, marking a specific milestone or other point in time.
- Managing variants in the code should be easy with proper branching.
- Code changed on a branch can be merged back onto the main trunk.
- Source code management process are repeatable, agile and lean.
- Source code management provides traceability and tracking of all changes.
- Source code management best practices help improve productivity and quality.

SCM

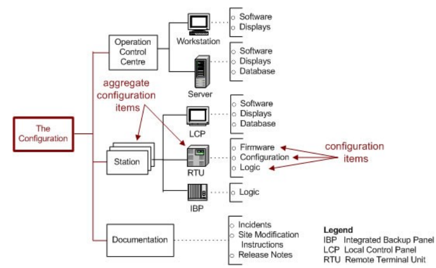
- SCM is sometime referred as software configuration management and sometime source code management.
- Software configuration management is more then source code management.



http://oneettn.tk/configuration-management-software/

Configuration Item(s)

- “Source code or another resulting artifact that make up the system” - Configuration Management (CM) terminology



http://www.chambers.com.au/glossary/configuration_item.php

Configuration Identification

- Identification of configuration items (CI)
- Labelling of CI's
  - Labels must be unique
  - A label usually consists of two parts:
    - Name, including title and number
    - Version
- Naming and versioning conventions
- Identification of baselines

Identify all parts that need to be controlled. What is the smallest part to be configuration managed?
The configuration identification shall reflect the product structure.
All configuration items must be labelled. Use a standard for naming and versioning. If your company does not have a standard, define one!
What is a baseline?

- A snapshot of a configuration at a certain point in time.
- A way to measure where in the development cycle a system really is.

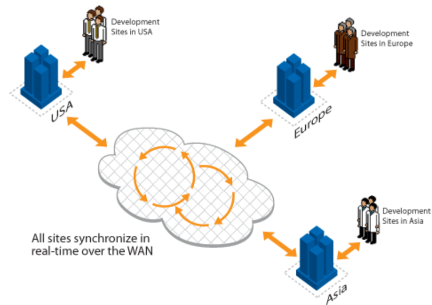
Why baselines?

- A stable point from which new projects (or releases) can be developed.
- To roll back to if changes have caused big problems.
- Possibility to recreate the configuration of the system.
- A base for testing.
- A base for supporting.
- A starting point for more formalized control.

Examples of baselines: functional baseline, design baseline, development baseline, product baseline.

Why it is important

- It gives the tools and processes to manage the configuration items.



http://blogs.wandisco.com/tag/distributed-version-control/

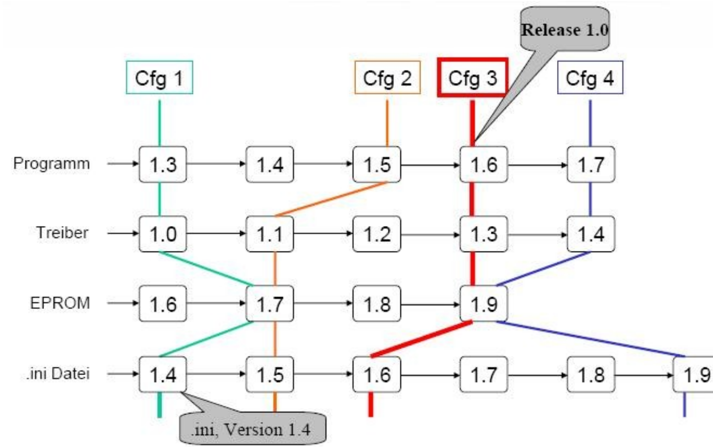
Start with Source Control Management

- Identify goals and requirements for a source code management.
- Organizations often starts by assessing their practice for securing their code and managing changes, baseline, releases, including bug-fixes.
- Start with agile and lean approach.

Core concepts creating baselines and time machines

- Source Code Management is not only “check in”/”check out”.
- Creating a **baseline** – identifying the exact versions of the code for a specif release. This operation has synonymous in CM tools – tagging, labeling, snapshotting.
- Baselines need to be **immutable**.
- Tag named “PRODUCTION” is “float” with current baseline of the code that is in production.

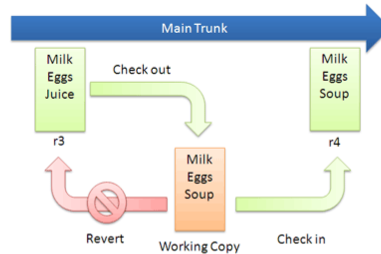
Core concepts baseline



https://wiki.enterpriselab.ch/edu/workspace:modules:swe:s08:deberli:arbeit

Check-in/check-out

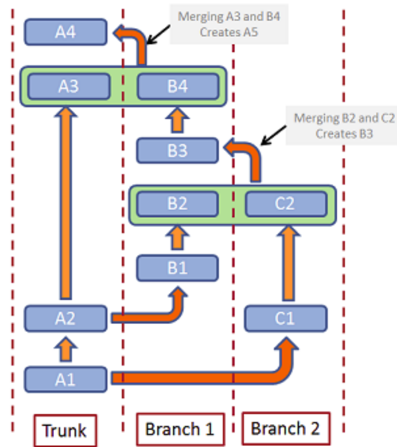
- Checkout:
- Reserved X unreserved
- Check-in
- Different tools may use different terminology: commit instead check-in.
- 2-phase commit (check-in)
- Concept of a private sandbox (workspace) is widely spread.



http://betterexplained.com/articles/a-visual-guide-to-version-control/

Variant management - branching

- Main branch (often called trunk/main)
- More working branches:
- Variants in code, different versions, working on bugfixes.
- Copybranch vs deltabranch



http://www.codeproject.com/Articles/431125/Choosing-a-Version-Control-System-A-Beginners-Tour

Six functional areas

- Source code management
- **Build engineering**
- Environment configuration
- Change control
- Release management
- Deployment

Build Engineering – Goals

- "It is to be able reliably compile and link your source code into a binary executable in the shortest possible time.”
- It includes identifing the exact compile and runtime dependencies and any other specific technical requirements, including compiler switches and dependencies.

Principles

- Builds are understood and repeatable.
- Builds are fast and reliable.
- Every configuration item is identifiable
- The source and compile dependencies can be easily determined.
- Code should be built once and deployed anywhere.
- Build anomalies are identified and managed in an acceptable way.
- The cause of broken builds is quickly and easily identified (and fixed).

Why it is important

- It helps the development team by providing an accurate and repeatable way to compile and link the code in the fastest possible way.
- Agile and Iterative development have highlighted this issue.
- Getting right build also avoids serious problems.

Start with Build engineering

- Start by looking at the existing development build procedures.
- Job of build engineer is to make build scripts more reliable and supportable.
- Evaluate existing tools and processes before starting to improve them.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.
- A few responsibilities
  - The first is that builds must be established that are repeatable,
  - based on an identifiable baseline and that all dependencies are well understood and controlled.
  - Every build consists of and creates configuration items (Cis).
  - The first task of a build engineer is to verify that all executables and essential scripts, documents, and text files are clearly identified.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.

- you need to be able to easily identify the exact version of anything that gets created by the build process
  - all binaries (intermediate code and runtime modules)
  - all configuration files
  - Anything whether they are source, binary, or configuration files.
- In an ideal world, everything should be identifiable with an immutable version ID. In practice, we are used to looking at the About box in a desktop GUI to see the version of the product that we are using. All documentation, including release notes, tutorials, and tech notes, must include version identification so that we know which version of the code they pertain to.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.

- The most basic form of this requirement is to stamp an executable with an immutable version ID (and provide an easy procedure to retrieve the version ID)
  - Build systems using a C++ static char variable with the version ID stamped into the executable.
  - Created JAVA classes to retrieve the version ID and stamp the version ID into the manifest of the JAR, WAR, or EAR file created by the build.
- The key is to make sure that the version ID can be easily traced back to the exact version of the source used to build that executable.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.

- In some cases, we actually stamped the executable with the source code management tool's version label or tag used to build the release. Because we created the build sandbox using this label or tag (and we locked it in the repository), we were reasonably certain that we had all the information that we needed to be able to reliably rebuild the baselined release as required.
- In some cases, we also needed to capture and record the revision of the repository itself (because tags could not be easily locked and developers could conceivably remove the tag and attach it to another version of the code)—after the release was already on its way to QA.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.

- Many builds break because an environment variable was set in the developer's own user account and then completely forgotten two months later when the code was being built for the release to production, most likely using another user account.
- It's not just about source code; all compile (and runtime) dependencies must be understood and controlled. That means that your build scripts should set all required environment variables and confirm that all build dependencies are correctly in place each and every time the build is executed.

Core concepts

- Version IDs and branding of executables
- Immutable Version IDSs
- Stamping In a Version Label or Tag
- Managing Compile Dependencies
- The independent Build
  - All configuration items are rebuild.

- One of the best ways to avoid costly mistakes is to have every release built independently and from the very top of the build structure so that all configuration items are completely rebuilt.
- This is often done by a separate release management team or by an automated build process as in continuous integration (CI).

Core Consideration for Scaling the Build Function

- Selling the independent build
  - sell this verification step as being a good way to guarantee (and literally test) that we did not overlook a compile dependency
- Overengineering the build
- Testing your own integrity
- Organizational choice
  - common for build engineering to be part of the QA group
  - also communicate with the head of systems administration
  - head of the build engineering team be someone who is willing to communicate issues and, if necessary, stop the build from being released (pending resolution of important issues)

Continuous Integration

- Continuous integration (CI) is a popular best practice that refers to attempting a build and deploy of code immediately after a developer commits changes to the source code repository.
- CI is usually done using a software package that makes it easier to monitor the source code management repository for changes and immediately start a build. The results of the build are posted on a dashboard, including the most recent changes responsible for any system outages, including a failed build.

Build Tools Evaluation and Selection

- Make – created in 1977 at Bell Labs
- Ant
- Maven
- Gradle
- Jenkins – continuous integration
- Integrated Development Environments
- "Build engineers face extra challenges when developers know how to build only through their IDEs".
- "Application should never built and deployed to production from within an IDE" - WHY?
- Static Code Analysis
  - A common application of static code analysis is to identify possible security vulnerabilities so that they can be fixed before the code is released. The build engineer is often the only person who can assemble all the code required for a particular release and successfully build the entire system with whatever hooks and modifications are necessary for the static code analysis.
- Include all your stakeholders in tool selection process.

Making a Good Build Better

- Test-Driven Builds
  - Capture also any step that you see developers using troubleshoot the build (e.g. missing particular JAR)
  - building in these tests and creating scripts to automate and check each step of the build
- Trust, But Verify
- Make build and deployment automation in a way that prevents mistakes from occurring:
  - the cockpit of a plane is designed in such a way as to minimize the likelihood of a mistake
  - Design the build so that each step is easy to understand and follow
  - Anticipate what might go wrong and build in tests to verify that the build is successful
- Structure the automation so that one step does not break the whole build
- Use dashboards and reports effectively to communicate build status.

The Role of the Build Engineer

- Software development backgrounds and expert knowledge of the technology and the ability to write code, including Perl, Python, shell scripts, and XML, to create reliable and repeatable builds.
- Plays key role in software development effort.
- Know What You Build – an excellent understanding of relevant build tools and a deep understanding of the application architecture (relevant technologies)
- Partner with Developers
- “Drafting a Rookie”

Establishing a Build Process

- Choose the best practices
- Choose the best tools
- provide training and support
- Establishing Organizational Standards
- Proper use of build tools – naming conventions, templates

Continuous Integration Versus the Nightly Build

- CI – extremely popular, often associated with Agile development,
- Very often, A nightly build is more then sufficient and much easier.
- CI – may trigger many extra builds that will fail → creating unnecessary failed entries on the CI dashboard.
- In some situations, CI causes builds to be executed before previous one has been completed.
- Always go for the lightest process possible – just-in-time process improvement

Conclusion

- Best practices will help to improve productivity and quality
- Build process should be:
- Automated,
- Traceable,
- Fast
- Kept as simple as possible
- Complex technology - try to break a build into manageable parts
- Carefully select your tools

Six functional areas

- Source code management
- Build engineering
- **Environment configuration**
- Change control
- Release management
- Deployment

Introduction

- It refers to identifying, modifying and managing the interface dependencies required for the system to successfully progress from development to QA to production.
- Often called *runtime* dependencies.

Goal

- "It is to always point to the correct runtime resources, such as the QA or production database.”
- "Ultimately, it is establishing and maintaining control as your system makes its way from development to QA to production"

Principles

- Environment configuration dependencies are identified and well understood.
- Environments can be interrogated for their current status.(e.g. ports open).
- Code should be build once with environment configurations changed prior to deployment.
- Environment configurations should be changed in a controlled and predictable way.
- Environment configurations should be documented and understood by all parties.

Why it is important

- It helps to manage both compile and runtime dependencies.
- Common mistake: accidentally to specify production DB when you wanted QA
- It will help release management and deployment.
- Having the flexibility to manage multiple environments means – it can be rapidly deployed and tested.

Start with environment configuration

- Small with environment configuration control – it can quickly grow.
- By getting your key compile and runtime dependencies understood and controlled.
- Eliminate hard coding of all dependencies as soon as possible.

Supporting Code Promotion

- The promotion of code throughout the development lifecycle:
  - Development
  - Quality assurance (QA) - more security controls
  - Production - most secure.
- Environment configuration control helps to make code promotion predictable and repeatable.

Managing configuration

- Configuration of used database.
  - „Which database are you using.“
- Configuration of used external resources.
  - „Did that trade go through.“
- Using tokens to refer to specific resource.
- Centralizing the environment variable assignment.
  - Configuration files
  - Configuration database – instantiate all runtime dependencies during release packaging

Practical Approaches to Establishing a CMDB

- Configuration management database – store IT assets – configuration items
  1) Examine the runtime environment and report back the status of environment configuration values.
  2) Contains the predefined environment configurations
- Identify an then control.
- Understanding the environment configuration

Change control depends on environment configuration.

- RFC may impact the runtime environment
- Configuration management system and specialized configuration management databases help manage environment configuration
- All environment changes need to be:
  - Identified
  - Understood
  - Controlled

Minimize the Number of Controls Required

- Keep environment control process very light in the beginning
  - Too enough controls motivates people to get clever at bypassing the process.
- Mistakes and errors can provide the motivation for improving the process

Managing Environments

- After software is written it is needed an environment to host the release of the code.
- It is recommended automation of the process
- Can test without accidental dropping a trade into production
- A good idea is to have an established procedure to drop and re-create a test environment from scratch – test db and any required resources.

The Future of Environment Configuration

- Existence of interesting tools and frameworks
- Using of virtualization
- Software as a service, cloud computing.
- Always pilot selected approaches.

Conclusion

- Important function (as other disciplines :D)
- Helps to be more productive and avoid a lot of painful mistakes.
- Helps also to develop higher-quality applications by facilitating rapid iterative development.

Six functional areas

- Source code management
- Build engineering
- Environment configuration
- **Change control**
- Release management
- Deployment

Introduction

- It is the most central function in configuration management.
- It is also one of the most underutilized and often misunderstood function.
- 7 types of change control

Goal

- “It is carefully manage all changes to the production environments."
  - part is only coordination (very important).
  - managing changes to the environment.
- It is essential to control which releases are promoted to QA and production.
- Can act as the stimulus to all other configuration management-related functions.

The goal of change control is to carefully manage all changes to the production (and usually QA) environments. Part of this effort is just coordination, and that is very important. But part of this is also managing changes to the environment that will impact all the systems in the environment. It is also essential to control which releases are promoted to QA and production. Change control can act as the stimulus to all other configuration management-related functions, too. This chapter explains how to use change control to manage your configuration management efforts.

Principles

- Changes should be planned and not just last-minute events.
- Changes should be understandable, including their downstream impact.
- Authority of approvals for changes should be established and obtained as appropriate.
- Procedure for emergency changes should be established to cover emergency incidents.
- Change control should assess and confirm that all configuration management process are being followed.

Why it is important

- It can help you to prevent problems that can be costly.
- It can also drive the entire configuration management process.

Change control is important because it can help you to prevent problems that can be costly. Without change control, changes to your production environment will likely result in serious mistakes that can impact your business in a significant way. A number of different types of change control can add value and help your organization run more efficiently. Change control can also drive your entire configuration management process. From guarding changes to your production environment to controlling changes to your processes, change control is important to your entire application lifecycle.

Start with change control

- It is best to start small – consider own goals and priorities. You need to consider implementing seven types of change control.
- By establishing a change control board (CCB) to review and approve all changes to production (or QA) – may include:
  - releases, patches, and runtime configuration changes.

Most people get started with change control by establishing a change control board (CCB) to review and approve all changes to production (or QA). This may include releases, patches, and runtime configuration changes. It has been my experience that it is best to start small and then add additional controls as needed based on risk (for example, potential for mistakes). Change control typically starts small and then grows as needed. As always, start by considering your own goals and priorities. There are seven types of change control that you need to consider implementing.

The seven types of change control

- A Priori
- Gatekeeping
- Configuration control
- Change advisory board
- Emergency change control
- Process engineering
- Senior management oversight

A Priori

- Permission for a change is requested before any actual change to the code is made.
- RFCs are usually created and reviewed by the respective CCB
- It usually refers to changes in the code and most often consist of defining requirements and then actual design of the system
- The role of CM:
  - To track requirements throughout the lifecycle
  - confirm that that all requirement were included in a specific release

Some organizations have a disciplined process whereby permission for a change
is requested before any actual change to the code is made. I have seen defense
contractors that had to describe the changes that they want to make and then
await approval from a government agency before actually writing the code that
implemented the change. In this process, requests for change (RFCs) are usually
created and reviewed by the respective CCB. A priori change control usually
refers to changes in the code and most often consists of defi ning requirements
and then the actual design of the system. The role of configuration management
in this case is to track requirements throughout the lifecycle and confirm that
all requirements were included in a specific release. Many organizations have
a regulatory requirement for tracking requirements, and that often includes a
change control function. Tracking source code changes to requirements is im-
portant, but controlling changes to production are essential, too.

Gatekeeping

- The most common type
- CCB reviews RFC that will impact production (or QA).
- It usually involves giving authority to promote new release.
- Patches to to existing released reviewed by the CCB.
- It generally evaluates whether risks that RFC could potentially impact the production
- CCB is responsible for reviewing the RFC and approving or rejecting the RFC
- CCB will require that all necessary technical experts be present at the CCB meeting – in the practice it is not practical.

The most common type of change control, and usually the first to be implemented, is “gatekeeping” change control where the CCB reviews RFCs that will
impact production (or QA). This usually involves giving authority to promote a new release of the code into production (or QA). Similarly, patches to existing releases are also reviewed by the CCB. This function generally evaluates whether
there is a risk that the RFC could potentially impact the production (or QA) environments. The CCB is responsible for reviewing the RFC and approving or rejecting the RFC. It is common for the members of the CCB to have questions about whether the change requested could impact the production (or QA)
environment. Traditionally, the CCB will require that all necessary technical experts be present at the CCB meeting—although, in practice, this is often not practical. The ITIL framework has made popular the use of a change advisory board (CAB) that consists of experts who can advise on the downstream impact
of a particular change. I discuss how to set up a CAB and why it might need to be separate from the CCB later in this chapter. Closely related are configuration
changes, as discussed in the next section.

Configuration control

- Refers to interface (runtime) dependencies only
- RFC involves a configuration change – CCB reviews and considers the downstream impact.
- Often understanding the interface dependencies often requires specialized expertise.
- Alternative name configuration control board.

When the RFC involves a configuration change, the CCB reviews and considers the downstream impact of the configuration change required.
Configuration
changes can have the same impact as a new release. In practice,
understanding
the interface dependencies often requires specialized expertise and should be
reviewed by a board that contains members who possess this expertise. In
this
case, I believe that the governing body should be called a configuration
control
board. However, there is some confusion in the terminology commonly used
today. Many of the industry standards describe the configuration control board
as governing the configuration of a system in terms of the configuration of the
source code itself instead of environment configuration. In these standards, a
configuration of the code refers to a specific set of versions of the source
code.
I believe that this usage is confusing and a relic of days past when configuration control referred to controlling the version of a Cobol program that was
being promoted on a large IBM mainframe computer. Today, we promote a
packaged release that may contain thousands of configuration Items,
including
binaries, XML, and many other artifacts. I believe that it makes more sense to
use configuration to refer to environment configuration and to use terms such
as
baseline or release to refer to a specific set of code versions that are
promoted as
a release. There are many reasons for this. Most releases are packaged, and
th

Change advisory board

- ITIL – CAB that acts as an expert resource to the change management function
- CCB have access to all required experts to effectively review RCP
- Without the service similar of a group similar to CAB – not understood changes resulting in mistakes and system outages.

I have been very impressed by the itSMF's ITIL framework that places a strong
focus on configuration management in the ITIL section on transition. I discuss
this further in Chapter 14, "Industry Standards and Frameworks." ITIL describes a change advisory board (CAB) that acts as an expert resource to the
change management function. This is the best description that I have seen that
solves the common problem that the folks involved with the process of change
control might not be the most knowledgeable in terms of all the required technical details. It is appropriate that the CCB have access to all required experts
to effectively review requests for change and identify any possible downstream
impacts. Without the services of a group similar to the ITIL CAB, changes could
be made that are not understood, resulting in mistakes and system outages.

Emergency change control

- Emergencies require immediate changes (always occurs)
- CCB cannot meet at any hour of the day or night to authorize emergency changes – focusing on strict adherence to the regular process → production system may being down for an extended period.
- Recommendation: a very senior manager's approval be required for emergency changes – abuse prevention.
- Discussion after the event to understand why an emergency change was required in the first place.

There are always times when emergencies require immediate changes. It is likely
that the CCB cannot meet at any hour of the day or night to authorize emergen-
cy changes, and focusing on strict adherence to the regular process may result
in the company production system being down for an extended period of time.
Any successful change control function must include a well-defined process for
managing emergency changes. I recommend that a very senior manager's ap-
proval be required for emergency changes and that there be discussion after the
event to understand why an emergency change was required in the first place.
I have seen situations in which technology professionals abused the emergency
change control process to bypass the regular change control process. In this case,
you will be successful if you have the support of senior management to ensure
that everyone follows the process in the best way possible.

Process engineering

- PE placed under control of a CCB
- CCB also tasked with communication process change to all affected parties and stakeholders
- Process improvement – organized continuing effort; process CCB can help to manage the process engineering effort on an ongoing basis

Organizations establish processes to run their businesses on a day-to-day basis.
These processes are established, and then the teams affected are expected to
comply with the process. The processes will sometimes need to be adjusted, and
this can have wide-ranging impacts on the entire organization. In this case, the
process engineering should be placed under control of a change control board
that is responsible for reviewing requests for changes to the process. The CCB
for process is also tasked with communicating process changes to all affected
parties and stakeholders. I believe that the best response to a mistake is to reex-
amine existing processes and ascertain whether additional process steps are war-
ranted. Process improvement is an organized continuing effort, and the process
CCB can help to manage the process engineering effort on an ongoing basis.

Senior management oversight

- Provide visibility to senior management and other stakeholders
- The best way to do is with a dashboard that lists
  - The upcoming RFC (including status)
  - Pending approvals
  - Others.
- Many organizations arrange CCBs in a hierarchical fashion.

The change control function should provide visibility to senior management and other stakeholders so that everyone knows the status of upcoming changes and also changes that have been completed (whether successfully or not). The best way to do this is with a dashboard that lists the upcoming RFCs, including their status, pending approvals, and other relevant information. You should also coordinate these efforts with the project management team, especially if your organization has a formal project management office (PMO). Some of my colleagues have pointed out that this function might seem different from the others, and I agree that it is indeed unique. Many organizations arrange their CCBs in a hierarchical fashion to ensure that change control has the proper oversight and control. This function maintains the topmost organizational oversight from a process and change control perspective and is normally only used in larger organizations.

Creating a change control function

- Establishing procedures to review all RFCs and ascertain potential impacts.
- Acting as gatekeeper.
- Most organizations establish a CCB to review and evaluate all RFC.
- Many organizations often handle configuration change control separately from other change control functions.

Change control involves establishing procedures to review all requests for changes and ascertain whether there are downstream potential impacts that might or might not cause a problem. Change control includes acting as a gatekeeper. In this regard, the change control function reviews requests for change and grants permission or rejects the request for change. Most organizations establish a CCB to review and evaluate all requests for changes. We discuss the role of the CCB as is commonly described in many industry standards, including those approved by ISO, IEEE, and frameworks, including Cobit, ITIL, and the CMMI. I also suggest that many organizations often handle configuration change control separately from other change control functions.

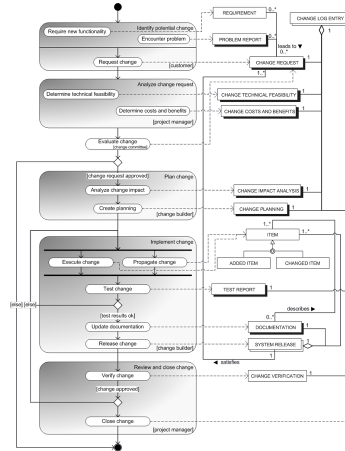
Risks

- Don't forget risks.
- Consider in terms of what might be impacted by a particular change.
- Must take additional steps to mitigate risks.
- Communicate risks to all stakeholders.
- Significant risks communicate to senior management.

Risks are inherent in any major IT effort. The change control process should always consider risk in terms of what might be impacted by a particular change.
This might mean that you will need to escalate a particular request for change to advise others of a problem that could possibly occur. It also might mean that
you must take additional steps to mitigate risks. It should always mean that you
communicate risk to all the stakeholders involved with this effort. In particular, significant risk is one of the items that should be communicated to senior management. It is common for senior management to be interested in change control, and you should consider driving the entire process through change control, too.

Driving the CM process through CC

- Requires considerable commitment and support from senior management
- Well-defined CM policy needed – it spell out the needed compliance with configuration directives.
- Reviewing all the steps required for the release in detail.



The change control process can drive the entire configuration management effort by requiring that all requests for change come with all related entry criteria completed. For example, the RFC to promote the release to QA should also include reviewing the CM plan to make certain that all the configuration management functions are completed correctly. Using change control to drive the CM process requires considerable commitment and support from senior management. There needs to be a well-defined CM policy that spells out the need for compliance with all related configuration management directives. Another example of how change control can drive the release process is reviewing all the steps required for the release in detail. The CCB can recommend that the release process be automated and get release managers to work together better to compare and share release management best practices, including script automation. This can and should include all aspects of configuration management, including source code management, build engineering, environment configuration, release packaging, and deployment.

Entry/exit criteria

- Entry criteria for the CC meeting – a concise description of the requested changes.
- Exit criteria – descriptions of the required tests to verify that the changes are successfully implemented without impacting the other systems.

The entry criteria for the change control meeting should be a concise description of the requested changes. I always require that project managers and development leads provide enough technical details about the change in advance of the meeting that other managers can review the request and ascertain whether there might be some impact on their own systems. The meeting itself is a discussion of possible downstream impacts and whether the change is actually required. The important information is provided for review before the meeting, and the other managers know that they have to participate or else be prepared to handle the consequences of an unexpected change. The exit criteria are descriptions of the required tests to verify that the changes are successfully implemented without impacting the other systems. I liberally use peer pressure to make this effort a success.

Implementing a process can sometimes be a tug-of-war between the project managers and the change management group. The PMs and development managers will insist that they are too busy with real work to be bothered with filling out forms and attending meetings. It works a lot better if you can get the PMs and development managers to view their efforts as being a service to their peers instead. I discuss this further in Chapter 10, “Overcoming Resistance to

After-action review

- Always after the change has been completed.
- Successful – the CCB reviews the completed change and advises that change is completed.
- Problems occur – the after-action review should facilitate discussion of what went wrong; CCB should make plans to avoid problems in the future.
- Discuss problems in an open and honest way.
  - essential for organizations to drive out fear
- Focus on prevention from occurring again.
- Mistakes are often the best catalyst for enhancing organizational processes to prevent mistakes form reoccurring

Change control should always be reviewed after the change has been completed.
This is important regardless of whether the change was successful. When RFCs
are completed successfully, the CCB simply reviews the completed change and
advises that the change is completed. When problems occur, the after-action re-
view should facilitate an open and honest discussion of what went wrong, and
the CCB should make plans to avoid problems in the future. W. Edwards Dem-
ing, widely regarded as the one of the great leaders of process and quality im-
provement, noted that it is essential for organizations to drive out fear. This is es-
pecially true when conducting an after-action review. The team needs to feel safe
that mistakes and problems can always be discussed in an open and honest way.
The focus should be on how to prevent the mistake from occurring again. The
after-action review is sometimes called a post-mortem or, in Agile, a retrospec-
tive. Regardless of the name, it is essential for the organization to discuss what
went well and what needs to be improved. Mistakes are often the best catalyst for
enhancing organizational processes to prevent mistakes from reoccurring.

Conclusion

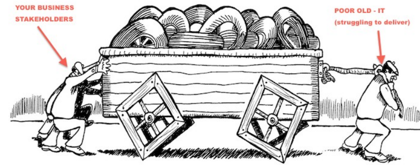
- You can drive all of you CM best practices from within CC
- Start small and implement each of the change control function as needed.

Six functional areas

- Source code management
- Build engineering
- Environment configuration
- Change control
- **Release management**
- Deployment

Introduction

- It is a core function in configuration management that focuses on packaging a system for promotion from development to QA to production.
- Often viewed as being a broad function that may encompass:
  - Source code management
  - Build engineering



Release management is a core function in configuration management that focuses on packaging a system for promotion from development to QA to production. If you are supporting a software production company, “production,” for you, may be shipping the product to the customer, instead of releasing the code to the production (or QA) environment. Whereas release management should focus on packaging the code created during the build process, release management is, in practice, often viewed as being a broad function that may encompass both source code management and build engineering. Release management in a corporate IT environment is slightly different from release management for a software product company—although I have worked in software product companies that still maintained separate QA, integration, and production environments as if they were a corporate IT environment even while shipping the finished product to an end user (or pushing changes via an automated installation process). In this chapter, we focus on defining release management as a function that takes over after the build has been completed and prepares the release for deployment into the desired environment. After a release has been created, it should conform to all the standards set by the release management team. In this chapter, we examine these and other best practices related to release

Goal

- It is to create and maintain a repeatable process for packaging a release.
- It must be clearly defined with little or no chance of error occurring
- It is an automated function that includes creating an immutable ID – embedded into release package
- It should also coordinate any dependencies that might be required for the release to successfully deploy.
- It should be completely traceable with a clearly defined procedure to verify that correct components have been deployed.

The goal of release management is to create and maintain a repeatable process for packaging a release that includes a clear way to identify every component of the release. Release management must be clearly defined with little or no chance of errors occurring. Generally, release packaging is an automated function that includes creating an immutable ID that is embedded into the release package itself. Release management should also coordinate any dependencies that might be required for the release to successfully deploy. Finally, release management should be completely traceable with a clearly defined procedure to verify that the correct components have been deployed into a runtime environment.

Principles

- Release should be readily identifiable with a immutable version ID.
- Release should be packaged with all the dependencies included.
- Release packaging should be automated and designed to avoid human error.
- Release management should be fast and reliable to facilitate iterative development.
- There should be a mechanism to conduct an audit of a release package to verify all of its configuration items.
- The contents of a release should be well understood, including the tracking of requirements.
- Release management should be a source of information on the status of all release, ideally though a release management dashboard.

Why it is important

- RM (release management) provides order to the development process.
- It is first line of defense in making sure that the release is ready to go.
- It often plays the key role in packaging, coordinating, and communicating the status of the release.
- RM is the glue that keeps the development process on track.

Start with release management

- Several places where you could start with implementing a RM function.
  - Creating a release calendar and communicating status – RM is a communications and coordination function.
  - Making certain that releases are always packaged in a reliable way that eliminates any chance of mistakes.
  - Priority for start depends on the specific problems solvable by RM functions.
  - Generally, it should be started be ensuring a reliable way to identify all configuration items and then proceed to automate a release packaging process.

There are several places where you could start with implementing an RM function. Sometimes, you need to focus on creating a release calendar and communicating status and, in this context, RM is a communications and coordination function. I usually start by making certain that releases are always packaged in a reliable way that eliminates any chance of mistakes. You might find that you have specific goals and priorities that will drive where you start with RM. I usually get called into an organization to solve a specific RM-related problem. In this context, my performance is judged based on whether I can solve the specific problem that is adversely impacting the organization. If you have the luxury of starting up an RM function without a specific fire to extinguish, I would say that you should start by ensuring that you have a reliable way to identify all configuration items (also known as configuration identification) and then proceed to automate your release packaging process.

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

Release management concepts and practices

Packaging strategies that work

- Ideally every configuration item should have an embedded immutable version ID that identifies the exact version of the configuration item deployed.
- Embedded version IDs can be traced back to the version labels or tags used to baselines.
- Reasonable and practical approach to this effort – it is not possible to imprint immutable version IDs in every CI
- Package version identification.
- An immutable version ID

A variety of RM concepts and practices are discussed in the following subsections. The focus of release management should be on ensuring that every configuration item (CI) has a unique version ID. This means that every binary has a unique internal stamp that can tell you the version ID of the CI. It is common for developers to proudly point out that the code in the source code management tool has been baselined using a unique version label, tag, or other identifier. But in release management, we have to ensure that all CIs can be identified when they are no longer solely in the version control repository and are also running in a production (or QA) environment.

In an ideal world, every configuration item should have an embedded immutable version ID that correctly identifies the exact version of the configuration item deployed. Release packages typically consist of one or more complete components that can run as a unit. It is true that there may be other dependencies required for the release, but one of the roles of RM is to identify these dependencies and provide a reliable way to manage them. Of course, you want to make sure that the embedded version IDs can be traced back to the version labels or tags

Release management concepts and practices

Sending a release map with the release

- A list of all the configuration items that were delivered as part of a release and developer release notes.
- Includes the size of all configuration items and their date stamps
- release map = bill of materials
- An immutable version ID – embed ID into binaries – branding the executable

The packaged release should always contain a list of all the configuration items
that were delivered as part of a release and developer release notes, product
documentation, and updated help files explaining what is included in the re-
lease. I sometimes call the list of what is included in the release my release map,
because it shows everything that was deployed, including the size in bytes of all
configuration items along with their respective date stamps. Some people call
this a bill of materials. We should also note that date stamps and sizes can actu-
ally be impacted by minor changes or environment issues that do not actually
threaten the integrity of the release. There are more reliable methods that we dis-
cuss, including the use of cryptographic keys. But still the release package itself
should always ship with an immutable version ID that can used to trace back to
the exact version of the source used to build that particular release.

An immutable version ID means that the package can be identified with an ID
that cannot be overwritten either intentionally or by accident. One way to do
this is to embed the version ID into the binary executable at build time. Embed-
ding version IDs into executables was covered in Chapter 2, “Build Engineer-

The ergonomics of release management

- Avoiding human error
- Analyze the reasons for the errors.
- Automate the release packaging effort.
- Understanding the technology.
- Tools from build engineering.

Release management as coordination

- Communicating the status of a release.
- Don't forget the release calendar
- RM and configuration control.

Release management is also a coordination function in that it helps to manage
all the tasks and requirements for a successful release. This may involve coor-
dinating the release itself and all the items that are required for a successful re-
lease. Part of this effort is ensuring that you communicate the status of a release
to all affected parties.

5.5.1 Communicating the Status of a Release

I have seen environments where everyone was doing a great job, but just about
nobody knew that to be the case. The communication within the team was poor
and almost nonexistent to the management above them. Poor communication
results in considerable frustration and can undermine the effectiveness of the en-
tire team. The RM process must, at a minimum, provide visibility into the status
of a release. I always communicate to all stakeholders that a release is planned,
and more important, when it begins to be deployed. Then, I always broadcast
the completion of the RM process along with success of the required smoke tests
that we describe in more detail in Chapter 6.

5.5.2 Don't Forget the Release Calendar

The RM function should also establish a calendar to maintain and

Requirements tracking

- Track requirements from the very beginning of the lifecycle to the final deployment into production.
- To know exactly what is included with a particular release
- To be certain that they do not miss a requirement.
- Many possible ways:
  - A very formal one.
  - Simply keep a list of requirements – in the release notes their delivering and packaging with the completed release is documented.
- Make sure that QA is kept advised of exactly what will be included in each release.

Many organizations need to be able to track requirements from the very beginning of the lifecycle to the final deployment into production. This is often because these organizations have a compliance requirement to know exactly what is included with a particular release. They also need to be certain that they do not miss a requirement. This is sometimes done in a very formal way. Other times, the project manager or development lead will simply keep a list of requirements and then document them in the release notes that are delivered and packaged with the completed release. As a release manager, I have often had to go to the project managers and tech leads to ask for the release notes. I also often take a few minutes to make sure that QA is kept advised of exactly what will be included in each release. Requirements often trigger test cases, and some requirements tracking tools interface with test case management software to generate test cases from requirements. Developing end-to-end support of the software development process is one of the ways that RM adds value to the organization.

Taking release management to the next level

- Many existing industry standards and frameworks provide guidance.
- Using cryptography to sign the code.
- Operating systems support for release management.
- Linux YUM package manager.
- Improving the RM process
- To be open to continuously improving as needs arise.
- RM should be reviewed as part of change control function.

Conclusion

- Pragmatic and realistic approach to establishing RM best practices.
- RM process should meet the needs of the organization in a flexible and reliable way.
- RM process should be:
  - fast,
  - efficient,
  - error free.

Six functional areas

- Source code management
- Build engineering
- Environment configuration
- Change control
- Release management
- **Deployment**

Introduction

- It is final step in the code promotion process

Goal

- It is to promote a release into production without any possible problem occurring.
- It should be like turning on a light switch.
- Enable rollback to previous version.
- To know exactly what is in production and immediately know whether any unauthorized changes have been made – another goal.

Principles

- Promoting a release should be reliable and as simple as possible.
- Promoting a release should be completely traceable with an audit log of all changes.
- Only authorized personnel should be involved with deployment.
- In most organizations, there needs to be a separation of duties between developers and the team that deploys the release.
- Any unauthorized changes should be detect immediately.
- There should a well established procedure for checking the version of a release in production.
- The deployment process should be continuously reviewed and improved as needed.

Why it is important

- You want to make certain you can:
  - Reliably promote a release forward
  - Take a step back and back out a release that was previously deployed.
- Done well – deployment should be a “nonevent”.

Start with deployment

- Automation to stage a release in a shared depot.
- create reliable automation to promote the release and back it out
- Keep deployment as simple as possible – it should be always performed by operations or system administrations team.

Practices

- Staging is key.
- Scripting the release process itself.
- Frameworks for deployment.
- Depot.
- Auditing your release.
- Smoke test.
- Little things matter a lot.
- Communications planning.
- Automation to verify that no changes have taken place.

Deployment should be delegated

- It should be the one function that does not stay within configuration management role.
- Operations team can deploy and fall back to a previous release as necessary.
- The release management team should help establish the deployment procedures and delegate their day-to-day operations.

Conclusion

- It should be the smallest of the CM-related functions. (WHY?)
- Keep the deployment procedure as simple as possible along with being fully traceable.
- Deployment should be performed by the operations team using procedures and automation.

Architecting the application for CM

Introduction

- CM depends on architecture in a number of important ways – often overlooked.
- Achilles' heel of CM – when the app changes → stop working:
  - Source code management
  - Builds
  - Release packaging and deployment

Goal

- It is improve quality and productivity by implementing CM best practices consistent with the architecture.
- Consider requirements of CM during application design and implementation:
  - Embedded ID
  - Rapid and iterative application development.

Why it is important

- Architecture is essential:
  - CM support team needs understand it.
  - CM can provide an essential service to facilitate the development effort by providing tools and process.
- The architecture influences this because the architecture defines
  - What makes up a deliverable unit
  - The communication paths among the units
  - Indirectly the directory structure and other structural aspects of the source code respository
- Modularity leads to decoupling, which adds concurrency to the development process.

Start with architecture

- By evaluating the complexity of the architecture of the application.
- Get ready to create example programs in many different language.
- Identification in reciprocity of help between CM and development.

How CM facilitates good architecture

- Good source code management strategies – baselines, variants – essential for supporting the development of a robust application architecture.
- The architecture itself may need to be designed for CM.
- Configuration management-driven development may produce better systems as Test driven development.
- One way is to provide a framework for organizing code into components, baselines, and snapshot.

CM functions support development

- Source code management
  - Using to facilitate architecture.
  - Training is essential.
  - As service
    - Provide help to developers to use the tools effectively
- Build engineering as service:
  - Developers can rapidly build and test the application.
  - Create a number of build machines usable on an "on-demand" basis.
  - Build can be run from a larger machine and then promoted to the test area.

Conclusion

- CM is impacted significantly by the application architecture.
- Implementing complex architecture is much easier done with CM best practices.
- CM team needs to communicate its requirements to the development organization and technology leadership needs to keep in mind the importance of working with CM team.
- CMDD

Hardware CM

Introduction

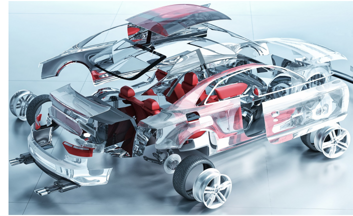
- It is often overlooked and undervalued.
- HW components need to be version controlled just like source code.
- Can't easily check a circuit board into a source code management tool.
- It often miss easy way to confirm the version of the hardware component or the firmware loaded.
- It is needed to have procedure to perform:
  - Configuration identification,
  - Change control,
  - Status accounting,
  - Configuration audits.

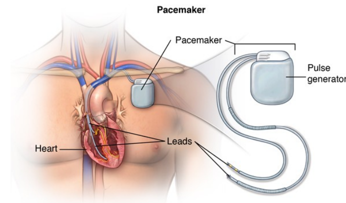
Goal

- Always to know which version of the hardware component is in use.
- To be able to track any changes to the hardware and control changes to the interface and external dependencies.
- To control environment changes that may impact the release management process.

Why it is important

- Getting the wrong version of any circuit chip or hardware could have disastrous consequences:
  - automotive,
  - medical instrument,





Start with hardware CM

- Similar procedures as for software – H**ardware cannot be checked to GIT(yet:-)).**
- Put all design documents under version control as source code.
- All hardware devices need a version ID(as design document).
- The design document and hardware device are configuration items

Best practices

- Version control for design specification – it also includes content of programmable memory.
- Include precise interface – of each component. Changes should be managed through the change control process.
- Understanding dependencies – change of one configuration item impacts another.

Best practices II

- Traceability:
  - All changes should be traced to RFC
  - Every production release must include release notes that indicate exactly what changes are included.
- Deploying changes to the firmware
  - Promoting the changes to firmware to be similar to promoting a release to production.

The future of hardware CM

- It deserves more attention.
- Future standards should support it.
- Hardware CM should also include:
  - Change control,
  - Configuration audits,
  - Tracking the evolution

Conclusion

- It is often overlooked.
- Many technology professionals do not know to handle hardware configuration management.
- Technology professionals need to control changes to HW just like to any other CI.
- Technical issues and requirements must be addressed to handle promoting firmware changes to hardware.
- Versions of hardware configuration items must be controlled.
- It is also needed to analyze and control the interface dependencies for hardware configuration items.

Tools CM

Source code management

- CVS
- SVN
- Team Foundation Version Control
- git
- Mercurial
- Github, Gitlab, Bitbucket – **RELATION TO PREVIOUS?**

Build engineering

- Make
- Ant,
- Maven,
- MSBuild,
- Gradle
- SBT
- Ivy

Environment configuration

- Chef, **Salt**, Puppet, Ansible…
- VMWare, VirtualBox, Parallels Desktop
- Vagrant
- Docker

Change control

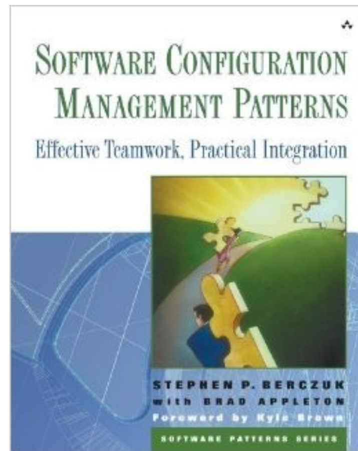
- Mantis,
- Bugzila
- Redmine
- Jira

Release management, deployment

- Continuous integration
  - Jenkins,
  - Bamboo
  - Gitlab CI
  - Team Foundation Server
- Continuous quality
  - SonarQube
  - Squale
  - Kalistick
  - MetrixWare
  - Cast

Software Configuration Management Patterns

Literature



Software configuration management

- SCM practices taken as a whole define how an organization builds and releases products and identifies and tracks changes.

A workspace

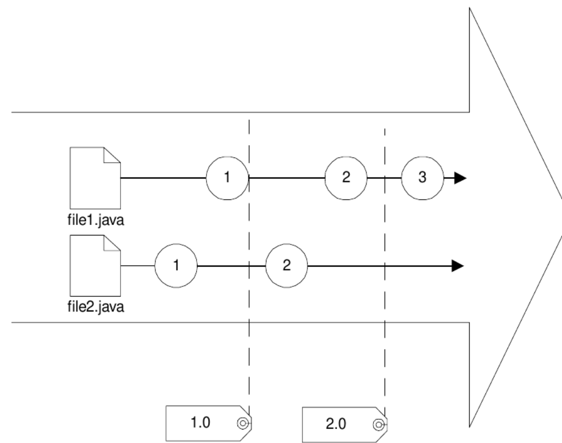
- is a place where a developer keeps all the artifacts he or she needs to accomplish a task.
- can be a directory tree on disk in the developer's working area, or it can be a collection of files maintained in an abstract space by a tool.
- is normally associated with particular versions of these artifacts.
- also should have a mechanism for constructing executable artifacts from its contents
- is also associated with one or more codelines.
- Sometimes is managed in the context of an integrated development environment (IDE)

For example, if you were developing in Java, your

- workspace would include
- Source code (.java files) arranged in the appropriate package structure
- Source code for tests
- Java library files (.jar files)
- Library files for native interfaces that you do not build (for example, .dll files in windows)
- Scripts that define how you build .java files into an executable

A codeline

- is a progression of the set of source files and other artifacts that make up some software component as it changes over time.
- Every time you change a file or other artifact in the version control system, you create a revision of that artifact.
- contains every version of every artifact along one evolutionary path.


file1.java
1
2
3
file2.java
1
2
1.0
2.0

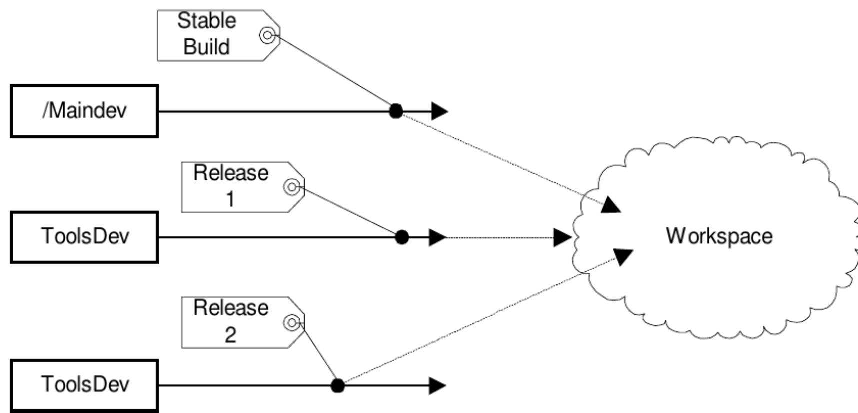

Stable Build
/Maindev
Release 1
ToolsDev
Workspace
Release 2
ToolsDev

Figure I-3. Branching a single file and merging with the trunk

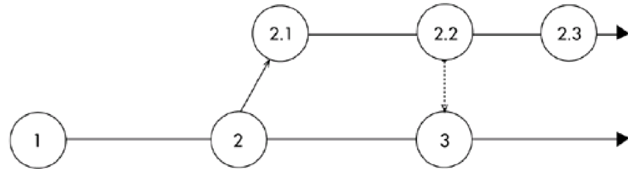
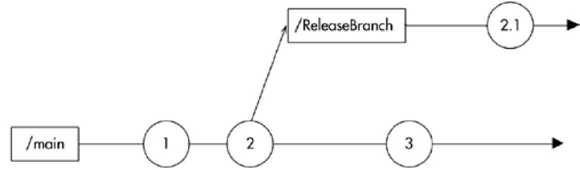


Figure I-4. Branching an entire codeline



Some development organizations take one of these extreme positions.

- Speed is essential, so we will worry about quality and versioning later. Besides, we're small enough that everyone knows what everyone else is doing.
- Quality is essential. We will work slowly, following processes to the letter, regardless of how it frustrates people on the project or reduces productivity. We work on one release at a time.

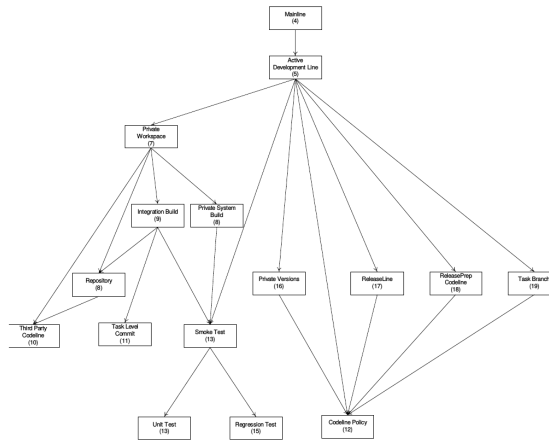
Consider the number of times you have experienced one of the following situations in a software organization.

- "We're in a code freeze. No one may check in any code until the product ships."
- "Just copy the files somewhere. I'll use your version."
- "It works for me! Do you have the correct version of the code?"
- "We use this tool in development, but builds are done with another version control tool. Be sure to keep them in sync!"

Structure of Patterns

- a **title**
- a **picture**
- a **context**
- the **problem**
- a detailed **problem description**
- A short summary of **the solution**.
- A description of **the solution in detail**.
- A discussion of **unresolved issues**

The Pattern Language

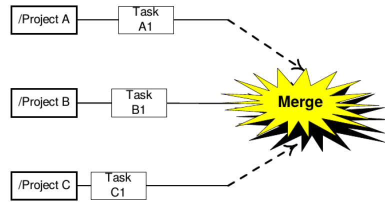


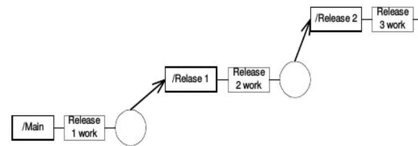
#1 - Mainline

- How do you keep the number of currently active codelines to a manageable set, and avoid growing the project's version tree too wide and too dense?
- How do you minimize the overhead of merging?

#1 – Mainline - II

- a merge can be messy
- staircase branching – it can make it hard to determine where code originated

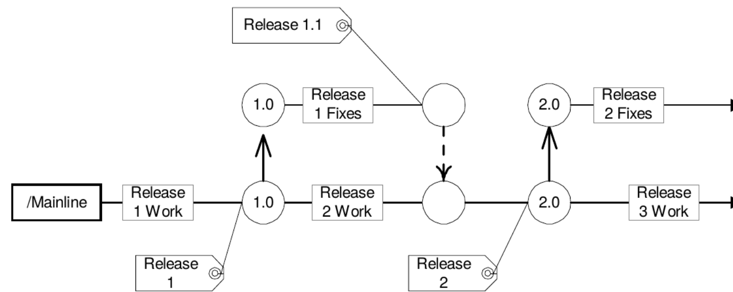




VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

#1 – Mainline - III

- Simplify your Branching Model
  - When you are developing a single product release, develop off of a mainline



The reason for a mainline is to have "a central codeline to act as a basis for subbranches and their resultant merges". The mainline for a project generally starts with the code base for the previous release or version. If you are doing new development, you start with only one codeline, which is your mainline by definition.

Doing mainline development does not mean "do not branch." It means that all ongoing development activities end up on a single codeline at some time.

Don't start a branch unless you have a clear reason for it and the effort of a later merge is greatly outweighed by the independence of the branch. Favor branches that won't have to be merged often—for example, release lines.

Mainline development – advantages

- Having a mainline reduces merging and synchronization effort by requiring fewer transitive change propagations.
- A mainline provides closure by bringing changes back to the overall workstream instead of leaving them splintered and fragmented.

#1 – Mainline - IV

- Unresolved Issues
  - how to keep the mainline usable when many people are working on it - ACTIVE DEVELOPMENT LINE

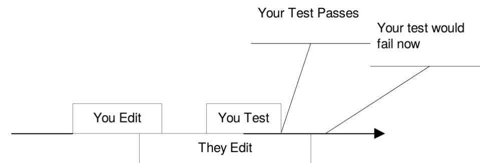
#2 - Active Development Line

- How do you keep a rapidly evolving codeline stable enough to be useful?

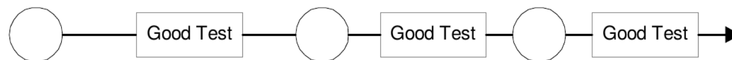
VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

#2 - Active Development Line - II

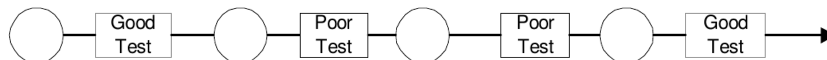
- Long running tests have mixed value



- A stable, bud dead, codeline



- a very active, but very useless codeline



You can prevent changes from being checked in to the codeline while you are testing by using semaphores, but then only one person at a time can test and check in changes, which can also slow progress. Figure 5-2 shows a very stable but very slowly evolving codeline.

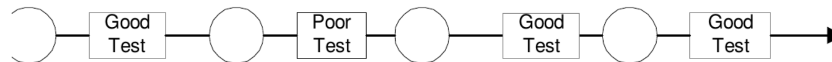
You can go to the other extreme and make your codeline a free-for-all. Figure shows a quickly evolving but unusable codeline.

Institute policies that are effective in making your main development line stable enough for the work it needs to do. Do not aim for a perfect active development line but for a mainline that is usable and active enough for your needs.

An active development line will have frequent changes, some well-tested checkpoints that are guaranteed to be "good," and other points in the codeline that are likely to be good enough for someone to do development on the tip of the line. Figure shows what this looks like.

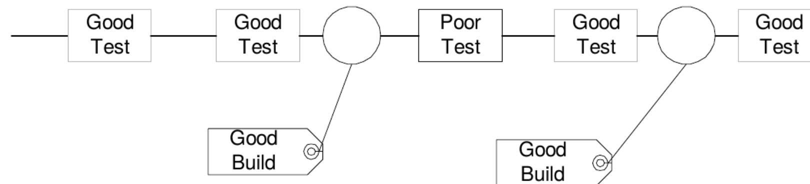
#2 - Active Development Line - III

- Define your goals
  - Institute policies that are effective in making your main development line stable enough for the work it needs to do.
  - Do not aim for a perfect active development line, but rather for a mainline that is usable and active enough for your needs
- An active, alive, codeline



#2 - Active Development Line - IV

- Do an analysis along the following lines
  - Who uses the codeline?
  - What is the release cycle?
  - What test mechanisms do we have in place?
  - How much is the system evolving?
  - What are the real costs will be for a cycle where things are broken?
- Labeling named stable bases



#2 - Active Development Line - V

- Unresolved Issues
  - Once you have established that a ‘good enough’ codeline is desirable, you need to identify the codeline that will be like this - Codeline Policy
  - An individual developer still needs isolation - PRIVATE WORKSPACE
  - When the need for stability gets close - RELEASE-PREP CODE LINE
  - Some long lived tasks may need more stability - TASK BRANCH

#3 – Private Workspace

- How do you do keep current with a continuously changing codeline, and also make progress without being distracted by your environment changing out from under you?

In Active Development Line, you and other developers make frequent changes to the code base, both to the modules you are working on and to modules you depend on. You want to be sure you are working with the latest code, but because people don't deal well with uncontrolled change, you want to be in control when you start working with other developers' changes. This pattern describes how you can reconcile the tension between always developing with a current code base and the reality that people cannot work effectively when their environment is in constant flux.

- Developing software in a team environment involves the following steps:
- Writing and testing your code changes
- Integrating your code with the work that other people were doing

- You can integrate every change team members make as soon as they make it. This is the clearest way to know whether your changes work with the current state of the codeline. The downside of this "continuous integration" into your workspace approach is that you may spend much of your time integrating, handling changes tangential to your task. Frequent integration helps you isolate when a flaw appeared. Integrating too many changes at once can make it harder to isolate where the flaw is because it can be in one of the many changes that have happened since you integrated.

- You can integrate at the last possible moment. This makes it simplest for you, the developer, while you are working, but it means that you may have many outside integration issues to deal with, meaning that it will take longer to integrate at the end.

Isolate Your Work to Control Change

- Do your work in a private workspace, where you control the versions of code and components you are working on. You will have total control over when and how your environment changes.

- Every team member should be able to set up a workspace where there is a consistent version of the software. A concise definition of a workspace is "a copy of all the 'right' versions of all the 'right' files in the 'right' directories". A workspace is also a place "where an item evolves through many temporary and inconsistent states until is checked into the library".

A private workspace

comprises the following:

- Source code you are editing.
- Any locally built components.
- Third-party derived objects that you cannot or do not wish to build.
- Built objects for all the code in the system. You can build these yourself, have references to a shared repository (with the correct version), or have copies of built objects.
- Configuration and data that you need to run and test the system.
- Build scripts to build the system in your workspace.
- Information identifying the versions of all the components in the system.

should not contain the following:

- Private versions of systemwide scripts that enforce policy. These should be in a shared binary directory so that all users get the latest functionality.
- Components that are in version control but that you copied from somewhere else. You should be able to reproduce the state of your workspace consistently when you are performing a task, by referencing a version identifier for every component in the workspace.
- Any tools (compilers, and so on) that must be the same across all versions of the product. If different versions of the product require different versions of tools, the build scripts can address this by selecting the appropriate tool versions for a configuration.

Coding for mainline development

1) Get up to date. Update the source tree from the codeline you are working on so that you are working with the current code and build, or repopulate the workspace from the latest system build. If you are working on a different branch or label, create a new private workspace from that branch.
2) Make your changes. Edit the components you need to change.
3) Do a Private System Build (8) to update any derived objects.
4) Test your change with a Unit Test (14).
5) Update the workspace to the latest versions of all other components by getting the latest versions of all components you have not changed.
6) Rebuild. Run a Smoke Test (13) to make sure that you have not broken anything.

- If your system is small enough, you can simply get source and any binary objects for the correct configuration of all of the product components and build the entire system. You might also consider getting the latest code from the MAINLINE (4) and building the entire system if it does not take too long. This will ensure that the system that you are running matches the source code. With a **good incremental build environment**, doing this should work rather well, allowing for, perhaps, the one time cost of the whole system build.

- If you are working on a multiple tasks, you can have multiple workspaces, each with their own configurations.
- One risk with a PRIVATE WORKSPACE (6) is that developers will work with old "known" code too long, and they will be working with outdated code. You can protect yourself from this by doing periodic Private System Builds and making sure that changes do not break the build or fail the SMOKE TEST (13). (The sidebar "Update Your Workspace to Keep Current"discusses the Workspace Update in more detail.)
- The easiest way to avoid getting out of date is to do fine grained tasks

Repository Pattern

- To create a PRIVATE WORKSPACE (6) or to run a reliable INTEGRATION BUILD (9) you need the right components. This pattern shows you how to build a workspace easily from the necessary parts.

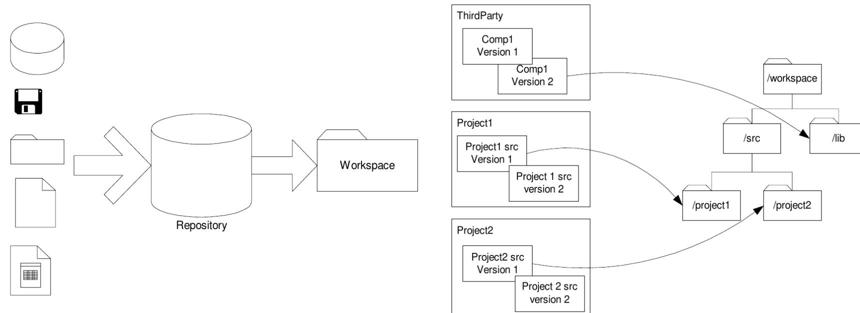
Workspace consists

Some of the things that you need to build and test a software some of the things that you need include:

- The source code that you are working with.
- Components that you are not working with, either as source, or library files.
- Third party components, such as jar files, libraries, dlls, etc. - depending on your language and platform.
- Configuration files
- Data files to initialize your application
- Build scripts and build environment settings so that you can get a consistent build
- Install scripts for some components

One Stop Shopping

- Have a single point of access, or a Repository, for your code and related artifacts.
- Make creating a developer workspace as simple and as transparent as possible.
- Make the mechanism that you use to create a workspace simple and repeatable.
- You should be able to create a workspace that contains artifacts from any identifiable revision of the product, including third party components and built artifacts such as library files.
- The mechanism should also make it easy to determine if there is a new version of an existing element, or a new component that you need when you are working on the tip of a development.


Repository
Workspace
ThirdParty
Comp1 Version 1
Comp1 Version 2
Project1
Project1 src Version 1
Project 1 src version 2
Project2
Project2 src Version 1
Project 2 src version 2
/workspace
/src
/lib
/project1
/project2

Private System Build

- A PRIVATE WORKSPACE (6) allows you, as a developer, to insulate yourself from external changes to your environment. But your changes need to work with the rest of the system too. To verify this, you need to build the system in a consistent manner, including building with your changes. This pattern explains how you can check to see if your code will still be consistent with the latest published code base when you submit your changes.

Think Globally by Building Locally

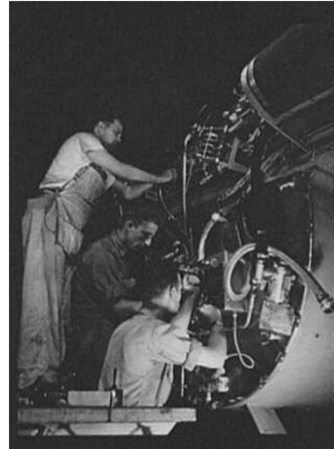
- Before making a submission to source control, build the system using a Private System Build that is similar to the nightly build.

The private system build should have the following attributes:

- Be like the INTEGRATION BUILD (9) and product builds as much as possible, though some details that are related to release and packaging can be omitted. It should at least use the same compiler, versions of external components, and directory structure.
- Include all dependencies.
- Include all of the components that are dependent on the change. (For example, various application executables.)

Integration build Pattern

- Each developer is working in their own PRIVATE WORKSPACE (6) so that he can control when he sees other changes. This helps individual developers make progress, but people are making independent changes in many workspaces that must integrate together, and the whole system must build reliably. This pattern addresses mechanisms for helping to ensure that the code for a system always builds.
- How do you make sure that the code base always builds reliably?
- Some users of the system may not want, need, or be able to build the entire code base. If they are developing software that simply builds on top of another component then they worrying about integration build issues will be a waste of their energy. They really want a snapshot of the system that they know builds.



Do a Centralized Build

- Be sure that all changes (and their dependencies) are built using a central integration build process.

This build process should be:

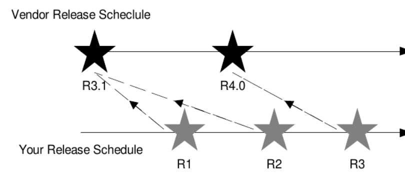
- Reproducible
- As close as possible to the final product build. Minor items, such as how files are version labeled might vary, but it is best if the Integration Build is the same as the Product build. At the end of the integration build, you should have a candidate for testing.
- Automated, or requiring minimal intervention to work. The harder a build is to run, the more even the best-intentioned teams will skip the process occasionally. If your source control system supports triggers, you could have the build run on every check-in.
- A notification or logging mechanism to identify errors and inconsistencies. The sooner that build errors are identified, the sooner they can be fixed.

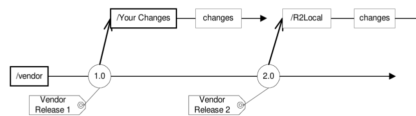
Third Party Codeline - Pattern

- What is the most effective strategy to coordinate versions of vendor code with versions of product code?

Use the tools you already have

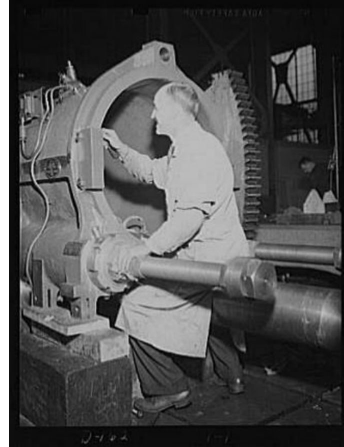
- Create a codeline for third party code. Build workspaces and installation kits from this codeline.





Task Level Commit - Pattern

- An INTEGRATION B UILD (9) is easier to debug if you know what went into it. This pattern discusses how to balance the needs for stability, speed, and atomicity.
- How much work should you do between submission to the Version Control System? How long should you wait before checking files in?
-



Do One Commit per small-grained task

Example of reasonable change tasks are:

- A problem report (but if the problem is a broad problem, it may have two or more check ins associated with it.)
- Changing calls to a deprecated method to use a new API for an entire system.
- Changing calls to a deprecated method for a coherent part of the system.
- A consistent set of changes that you accomplished in a day.

Codeline Policy

- When you have multiple codelines, developers need to know how to treat each one. A RELEASE LINE (17) might have strict rules for how and when to check things in, but an ACTIVE D EVELOPMENT LINE (5) might have less strict rules. This pattern describes how to establish the rules for each codeline to suit its purpose.
- How do the developers know which codeline to check their code into, and when to when to check it in, and what tests to run before check in?

Define the Rules of the Road

- For each branch or codeline, formulate a policy that determines how and when developers should make changes. The policy should be concise and auditable.
- The codeline policy explicitly states the rudimentary policies an organization has about how to conduct concurrent development and how to manage releases. Vance says that “a codeline policy defines the rules governing the use of a codeline or branch” (Vance 1998). In addition to using naming conventions and meaningful codeline names, formulate a coherent purpose for each codeline. Describe the purpose in a clear and concise policy.

Define the Rules of the Road

- The policy should be brief, and should spell out the “rules of the road” for the codeline, including:
  - The kind of work encapsulated by the codeline, such as development, maintenance, a specific release, function, or subsystem;
  - How and when elements should be checked-in, checked-out, branched and merged;
  - Access restrictions for various individuals, roles, and groups;
  - Import/export relationships: the names of those codelines it expects to receive changes from, and those codelines it needs to propagate changes to;
  - The duration of work or conditions for retiring the codeline;
  - The expected activity-load and frequency of integration

Some example of policies for include

- Development codeline: interim code changes may be checked in; affected components must be buildable. (Wingerd and Seiwald 1998)
- Release codeline: software must build and pass regression tests before check-in; check-ins limited to bug fixes; no new features or functionality may be checked in; after check-in, branch is frozen until entire QA cycle is completed.(Wingerd and Seiwald 1998)
- Mainline: all components must compile and link, and pass regression tests; completed, tested new features may be checked in. (Wingerd and Seiwald 1998)

Smoke Test

- An INTEGRATION BUILD (9) or a PRIVATE SYSTEM BUILD (8) are useful for verifying buildtime integration issues.But even if the code builds, you still need to check for runtime issues that can cause you grief later. This verification is essential if you want to maintain a A CTIVE DEVELOPMENT LINE (5). This pattern addresses the decisions you need to make to validate a build.

Smoke Test

How do you know that the system will still work after you make a change?

- You can write tests that target the most critical or failure prone parts of the code, but it is hard to develop complete tests.
- Unstructured and impromptu testing will help you to discover new problems, but it may not have much of an effective yield.
- Rapid development and small grained checkins means that you want the cost of pre-checkin verification to be small.
- The Right Balance

Verify Basic Functionality

- Subject each build to a smoke test that verifies that the application has not broken in an obvious way.
- A smoke test should be good enough to catch "show stopper" defects, but disregard trivial defects(McConnell 1996). The definition of "trivial" is up to the individual project, but you should realize that the goal of a smoke test is not the same as the goal of the overall quality assurance process.
- A smoke test should be:
  - Quick to run, where 'quick' depends on your specific situation
  - Self scoring, as any automated test should be.
  - Provide broad coverage across the system that you care about
  - Be runnable by developers, as well as part of the quality assurance process.

Private Versions

- Sometimes you want to rapidly evaluate a complex change that may break the system while maintaining an ACTIVE D EVELOPMENT LINE (5). This pattern describes how to maintain local traceability without affecting global history unintentionally.
- How can you experiment with a complex change and benefit from the version control system without making the change public?

Private Versions

- A Private History
- Provide developers with a mechanism for check pointing changes at a granularity that they are comfortable with. This can be provided for by a local revision control area, Only stable code sets are checked into the project repository
- There are many ways to implement this. One way is to have an entire PRIVATE WORKSPACE (6) dedicated to a task.
- It is important to make sure that developers using Private Versioning remember to migrate changes to the shared version control system at reasonable intervals.

Release Line

- You want to maintain an ACTIVE D EVELOPMENT LINE (5). You have released versions that need maintenance and enhancements, and you want to keep the released code base stable. This pattern shows you how to isolate released versions from current development.
- How do you do maintence on released versions without interfering with your current development work?

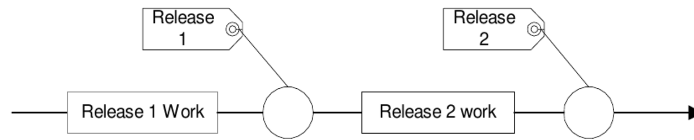


Figure 17-1 .Doing all your work on the mainline

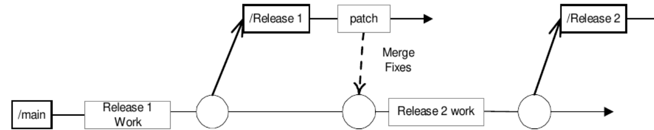
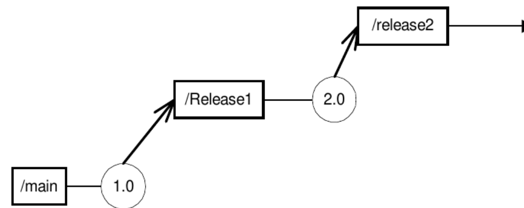


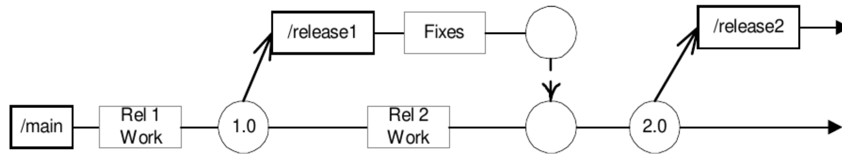
Figure 17-2 .Create a Branch when you Ship

- You can put your new work on a branch, and ship the mainline. You then can merge back. This means that most developers need to merge their work; hopefully the released code won't change too much over time. You may have more than one customer, each with variations of the released software; you may need to keep track of multiple releases that are derived from other releases. You can model this by the staircase structure in Figure 17-3. This structure makes it very hard to figure out what code is common among the releases.



Release Line

- Split maintence/release and active development into separate codelines. Keep each released version on a release line. Allow the line to progress on its own for bug fixes. Branch each release off of the mainline.



Release-Prep Code Line

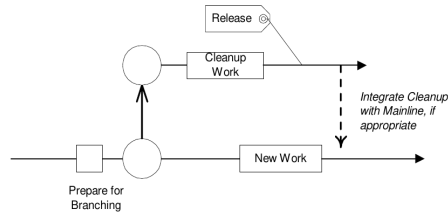
- You're finishing up a release and also need to start continue development on the next release. You want to maintain an A CTIVE DEVELOPMENT LINE (5).
- How do you stabilize a codeline for an impending release while also allowing new work to continue on an active codeline?

Release-Prep Code Line

- There are last minute bugs to fix, details related to installation and packaging and other last minute details to tend to. It is best to not do any major new work on the active development codeline while this clean up is going on, since you don't want to introduce any new problems. You will want to have very restrictive check in and QA policies during this "clean-up" period.
- One solution is to freeze development on the active development line until the release stabilizes.

Branch instead of Freeze

- Create a release-engineering branch when code is approaching release quality. Finish up the release on this branch, and leave the mainline for active development.
- The branch becomes the release branch.

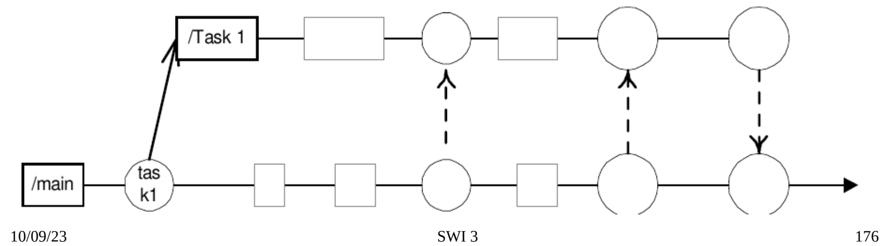


Task Branch

- Handling Long Lived Tasks
- How can your team make multiple, long term overlapping changes to a codeline without compromising its consistency and integrity?

Task Branch

- Use Branches for Isolation
- Fork off a separate branch for each activity that has significant changes for a codeline.



Semantic Versioning 2.0.0 - https://semver.org/

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 backward compatible manner
3) PATCH version when you make backward compatible bug fixes

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

Semantic Versioning

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

2) A normal version number MUST take the form X.Y.Z where X, Y, and Z are non-negative integers, and MUST NOT contain leading zeroes. X is the major version, Y is the minor version, and Z is the patch version. Each element MUST increase numerically. For instance: 1.9.0 -> 1.10.0 -> 1.11.0.

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

4) Major version zero (0.y.z) is for initial development. Anything MAY change at any time. The public API SHOULD NOT be considered stable.

5) Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.

Semantic Versioning

6) Patch version Z (x.y.Z | x > 0) MUST be incremented if only backward compatible bug fixes are introduced. A bug fix is defined as an internal change that fixes incorrect behavior.
7) Minor version Y (x.Y.z | x > 0) MUST be incremented if new, backward compatible functionality is introduced to the public API. It MUST be incremented if any public API functionality is marked as deprecated. It MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes. Patch version MUST be reset to 0 when minor version is incremented.
8) Major version X (X.y.z | X > 0) MUST be incremented if any backward incompatible changes are introduced to the public API. It MAY also include minor and patch level changes. Patch and minor versions MUST be reset to 0 when major version is incremented.

Semantic Versioning

9) A pre-release version MAY be denoted by appending a hyphen and a series of dot separated identifiers immediately following the patch version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Numeric identifiers MUST NOT include leading zeroes. Pre-release versions have a lower precedence than the associated normal version. A pre-release version indicates that the version is unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version. Examples: 1.0.0-alpha, 1.0.0-alpha.1, 1.0.0-0.3.7, 1.0.0-x.7.z.92, 1.0.0-x-y-z.--.

10) Build metadata MAY be denoted by appending a plus sign and a series of dot separated identifiers immediately following the patch or pre-release version. Identifiers MUST comprise only ASCII alphanumerics and hyphens [0-9A-Za-z-]. Identifiers MUST NOT be empty. Build metadata MUST be ignored when determining version precedence. Thus two versions that differ only in the build metadata, have the same precedence. Examples: 1.0.0-alpha+001, 1.0.0+20130313144700, 1.0.0-beta+exp.sha.5114f85, 1.0.0+21AF26D3----117B344092BD.

Semantic Versioning - Precedence refers

- Precedence MUST be calculated by separating the version into major, minor, patch and pre-release identifiers in that order (Build metadata does not figure into precedence).
- Precedence is determined by the first difference when comparing each of these identifiers from left to right as follows: Major, minor, and patch versions are always compared numerically.
- Example: 1.0.0 < 2.0.0 < 2.1.0 < 2.1.1.
- When major, minor, and patch are equal, a pre-release version has lower precedence than a normal version:
- Example: 1.0.0-alpha < 1.0.0.

Semantic Versioning - Precedence refers

Precedence for two pre-release versions with the same major, minor, and patch version MUST be determined by comparing each dot separated identifier from left to right until a difference is found as follows:

1) Identifiers consisting of only digits are compared numerically.
2) Identifiers with letters or hyphens are compared lexically in ASCII sort order.
3) Numeric identifiers always have lower precedence than non-numeric identifiers.
4) A larger set of pre-release fields has a higher precedence than a smaller set, if all of the preceding identifiers are equal.

Example: 1.0.0-alpha < 1.0.0-alpha.1 < 1.0.0-alpha.beta < 1.0.0-beta < 1.0.0-beta.2 < 1.0.0-beta.11 < 1.0.0-rc.1 < 1.0.0.

Other Versioning

Calendar Versioning

- Ubuntu 18.04, for example, was released in April 2018
- Eclipse 2023-06

TeX has an idiosyncratic version numbering system, an unusual feature invented by its developer Donald Knuth. Since version 3.1, updates have been indicated by adding an extra digit at the end, so that the version number asymptotically approaches the number π. As of February 2021, the version number is 3.141592653.

Code Names

- Debian – Toy Story (hamm, slink, potato, woody, ...)
- Eclipse – (Callisto, Europa..., Helios, Indigo, Kepler, Luna, Mars, Neon Oxygen, Photon)

Versioning

Superstition number 13

- The Office 2007 release of Microsoft Office had an internal version number of 12. The next version, Office 2010, has an internal version of 14
- Visual Studio 2013 is Version number 12.0 of the product, and the new version, Visual Studio 2015 has the Version number 14.0
- Roxio Toast went from version 12 to version 14
- Corel's WordPerfect Office, version 13 is marketed as "X3" (Roman number 10 and "3"). The procedure has continued into the next version, X4.

Geek culture

- The SUSE Linux distribution started at version 4.2, to reference 42, "the answer to the ultimate question of life, the universe and everything"

Conventional Commits 1.0.0

The Conventional Commits specification is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of.

- <type>[optional scope]: <description>
-
- [optional body]
-
- [optional footer(s)]
- https://www.conventionalcommits.org/en/v1.0.0/

Conventional Commits

1) **fix**: a commit of the type fix patches a bug in your codebase (this correlates with **PATCH** in Semantic Versioning).
2) **feat**: a commit of the type feat introduces a new feature to the codebase (this correlates with **MINOR** in Semantic Versioning).
3) **BREAKING CHANGE**: a commit that has a footer BREAKING CHANGE:, or appends a **!** after the type/scope, introduces a breaking API change (correlating with **MAJOR** in Semantic Versioning). A BREAKING CHANGE can be part of commits of any type.
4) types other than fix: and feat: are allowed, for example @commitlint/config-conventional (based on the Angular convention) recommends **build**:, **chore**:, **ci**:, **docs**:, **style**:, **refactor**:, **perf**:, **test**:, and others.
5) footers other than BREAKING CHANGE: <description> may be provided and follow a convention similar to git trailer format.

Conventional Commits - GitHub

- Merge Commit: Merge branch '<branch name>'
- Revert Commit: Revert "<reverted commit subject line>"

Types

- **refactor** - rewrite/restructure your code, however does not change any behaviour
  - **perf** - special refactor commits, that improve performance
- **style** - do not affect the meaning (white-space, formatting, missing semi-colons, etc)
- **test** - add missing tests or correcting existing tests
- **docs** - affect documentation only
- **build** - affect build components like build tool, ci pipeline, dependencies, project version, ...
- **ops** - affect operational components like infrastructure, deployment, backup, recovery, ...
- **chore** Miscellaneous commits e.g. modifying .gitignore

Conventional Commits

- **change** - changes the implementation of an existing feature.
- **chore** - includes a technical or preventative maintenance task that is necessary for managing the product or the repository, but it is not tied to any specific feature or user story. For example, releasing the product can be considered a chore. Regenerating generated code that must be included in the repository could be a chore.
- **ci** - makes changes to continuous integration or continuous delivery scripts or configuration files.
- **deprecate** - deprecates existing functionality, but does not remove it from the product. For example, sometimes older public APIs may get deprecated because newer, more efficient APIs are available. Removing the APIs could break existing integrations so the APIs may be marked as deprecated in order to encourage the integration developers to migrate to the newer APIs while also giving them time before removing older functionality.
- **remove** - removes a feature from the product. Typically features are deprecated first for a period of time before being removed. Removing a feature from the product may be considered a breaking change that will require a major version number increment.
- **revert** - reverts one or more commits that were previously included in the product, but were accidentally merged or serious issues were discovered that required their removal from the main branch.
- **security** - improves the security of the product or resolves a security issue that has been reported.
- https://medium.com/neudesic-innovation/conventional-commits-a-better-way-78d6785c2e08

Conventional Commits - GitHub

Scopes - The scope provides additional contextual information.

- Is an optional part of the format
- Allowed Scopes depends on the specific project
- Don't use issue identifiers as scopes

Description - The description contains a concise description of the change.

- Is a mandatory part of the format
- Use the imperative, present tense: "change" not "changed" nor "changes"
  - Think of This commit will <subject>
- Don't capitalize the first letter
- No dot (.) at the end

Conventional Commits - GitHub

Body - The body should include the motivation for the change and contrast this with previous behavior.

- Is an optional part of the format
- Use the imperative, present tense: "change" not "changed" nor "changes"
- This is the place to mention issue identifiers and their relations

Footer - The footer should contain any information about Breaking Changes and is also the place to reference Issues that this commit refers to.

- Is an optional part of the format
- optionally reference an issue by its id.
- Breaking Changes should start with the word BREAKING CHANGES: followed by space or two newlines. The rest of the commit message is then used for this.

Conventional Commits

- feat: allow provided config object to extend other configs
-
- BREAKING CHANGE: `extends` key in config file is now used for extending other config files

- feat!: send an email to the customer when a product is shipped

- feat(api)!: send an email to the customer when a product is shipped

Conventional Commits

- chore!: drop support for Node 6
-
- BREAKING CHANGE: use JavaScript features not available in Node 6.

- docs: correct spelling of CHANGELOG

- feat(lang): add Polish language

Conventional Commits

- fix: prevent racing of requests
-
- Introduce a request id and a reference to latest request. Dismiss incoming responses other than from latest request.
-
- Remove timeouts which were used to mitigate the racing issue but are obsolete now.
-
- Reviewed-by: Z
- Refs: #123

Why Use Conventional Commits

- Automatically generating CHANGELOGs.
- Automatically determining a semantic version bump (based on the types of commits landed).
- Communicating the nature of changes to teammates, the public, and other stakeholders.
- Triggering build and publish processes.
- Making it easier for people to contribute to your projects, by allowing them to explore a more structured commit history.