

↘ Inner Source

Collaborative development across organisations and locations

Auke Jilderda
Principal Collaboration Consultant
auke@collab.net, +31 402 364 205

Objectives

- Today's talk aims to outline **why** and **how** to organise collaborative development that is distributed across organisational and geographical boundaries
- We will focus on the high level overview because the key to success of the approach is in the combination of aspects. Each aspect can be implemented in various ways and we can touch on those in detail another time.

This material is based on

- Early theoretical work at Philips Research
- Hands-on roll out of those ideas:
 - opportunistic approach, Philips wide and
 - planned approach, at a product family program at Philips Medical Systems
- More roll outs at CollabNet, e.g. Electronic Arts

➤ Introduction & Aim

- In today's product development, the majority of the (software) technology parts of a product is commodity to the producer, e.g.
 - mobile phone with Symbian or Linux operating system
 - computed tomography scanner with DICOM as standardised data format
 - consumer electronics such as televisions, video recorders, MP3 players using TCP/IP (as opposed to for instance HAVI) as network stack
- What is differentiating today is commodity tomorrow
 - there are two ways to handle commodity: 3rd party or open source; today's industry almost only focusses on the former
- Software commodified hardware ('80ies)
 - The rise of Microsoft & the fall of IBM
- Open source commodifies software
 - The rise of LAMP and SaaS and (Google) & the fall of MS Windows (and Microsoft)
 - For SaaS, value moves to data set, rather than functionality (e.g. Google, Amazon, e-Bay)

➤ Introduction & Aim (cont'd)

This commodification of software poses three challenges to a software engineering organisation: Teams need to

- Collaborate across organisations, local cultures, departmental cultures and heritage, and various maturity levels
- Gradually and continuously shift its focus, keeping it on the parts that are differentiating to the business
- Take a fundamentally different approach to managing requirements, change requests, and enhancement requests for commodity technology

Today, we will

- Outline the trends driving the commodification
- Explain how to transform your internal engineering to enable commodification
- Explain why and how to move technology from inside your organisation into open source, including how that impacts your requirements management and roadmapping for that technology

➤ Outline

- Trends in software engineering
 - the world is flat(ening)
 - commodification of software
 - shift from in-house development to integration and assembly
- Key challenge
- Collaborative development
 - evolutionary roles
 - openness
 - decentralised ownership & control
 - patch rather than workaround
- Development environment
- Examples

↘ TRENDS IN SOFTWARE ENGINEERING

↘ The World is Flat(ening)

In the software industry, it becomes common practice to “outsource” (or “offshore”, “nearshore”, “onshore”, “farshore”, “right-shore”) parts of development, e.g. to

- Reduce costs
- Align with governments
- Reorganise after recent acquisitions or mergers

As a consequence

- Teams differ dramatically because of their location & heritage
 - in mindset
 - in local culture, e.g. Indian vs French vs German vs Californian
 - in corporate/departmental culture, e.g. a recently acquired small company vs a department that has been part of the corporation for decades
 - in approach and processes
- These differences are often complementary & Good™!
 - e.g. some are better suited to evolving and maintaining mature components, some better to innovating new components

➤ Commodification of Software

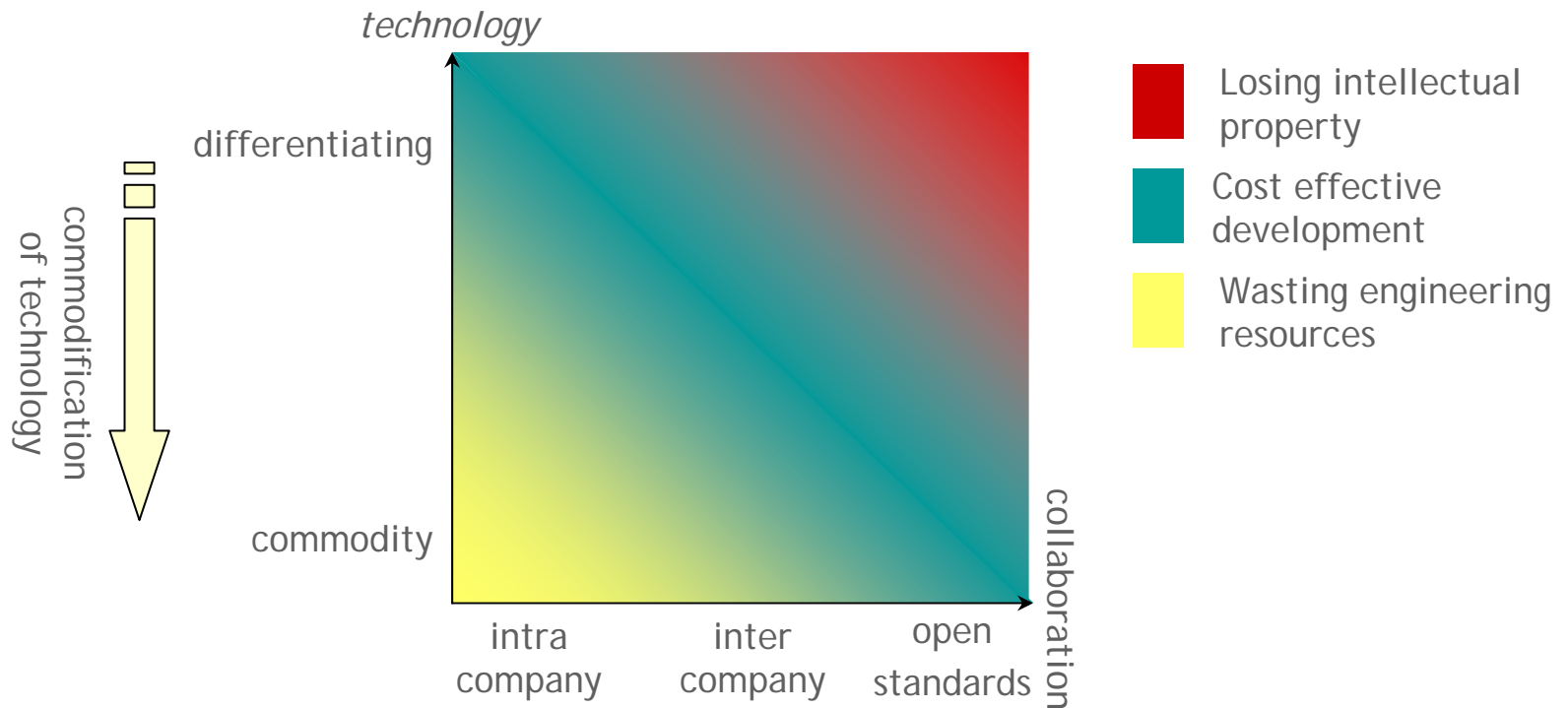
Markets demand

- Increased integration of products
- Improved reliability and quality
- Reduced time-to-market

This drives commodification of software which, in turn, drives

- A shift from in-house development towards assembly of parts that have been developed by others, e.g. via sub-contracting
- A shift towards coalitions or collaborations
 - e.g. open platforms such as the Symbian operating system

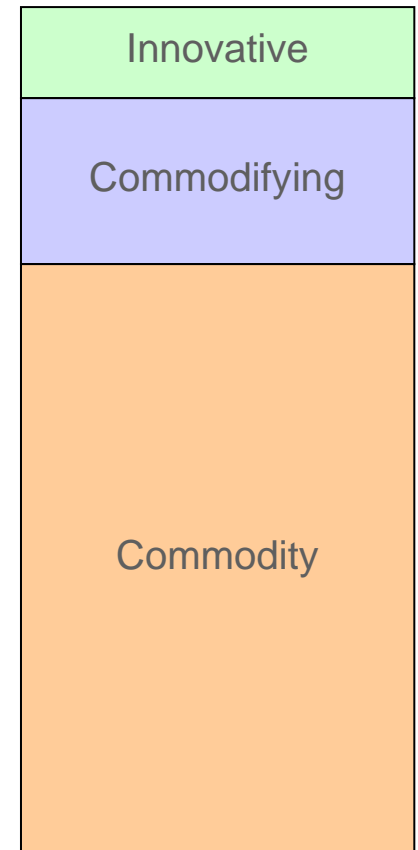
Commodification of Software (continued)



➤ Shift Towards Assembly

Building a product requires assembly of parts at different maturity levels:

- **Innovative**
 - needs agility and speed
 - unstable, re-use requires ability to adapt asset
 - **Commodifying (e.g. a product family)**
 - needs traceability and quality
 - somewhat more stable; re-use still requires ability to adapt asset
 - **Commodity**
 - typically obtained from 3rd parties or open source
 - stable enough for re-use as-is
- Collaborating teams have different needs and foci and standardising processes and approaches is neither feasible nor productive.
- Re-use requires the ability to adapt an asset



➤ Key Observations in These Trends

- Collaborative development is here to stay
 - it is neither a coincidence nor a temporary phenomena but a direct consequence of globalisation
- The teams needing to collaborate are heterogeneous in many aspects
 - collaborative development succeeds only if it allows, or even leverages, the heterogeneity
- Re-use of assets requires the ability to adapt those assets

➤ What does it mean for the daily practice?

- Teams have to focus on the differentiator in the software stack
 - This gives the competitive advantage
- The importance of change requests or fixed release dates can differ
 - For some parts of the software stack, maturity becomes a more important criteria
 - New features of commodified parts of the software should only be released when stable
- The key factor is to slice and dice the software stack in an optimal way
 - Similar to the „Golden Cut“ – there is no algorithm
 - For a first assesment also include holy cows and think outside of the box

➤ What does it mean for the daily practice?

- Everything with limited external interaction should be considered first
 - Helps to control things
- More than any other initiative this needs to be backed and supported by the management team!
 - When going for an Inner Source approach you will face political and psychological resistance

➤ Setting the stage

- You are working for a large utilities company. In your role of a chief architect it's your responsibility to define the future setup of your development group
- Just assume for the moment you have no limitations when making your decision

➤ Exercise #1 – Analyzing the software stack

- Do a triage into innovative, commodifying and commodity
 - CRM system
 - Standard customer and vendor management
 - PDE (Plan Definition Engine for product managers)
 - This software is used to turn new products into billable offerings
 - Bill analysis system
 - Creates metrics and statistics out of customer billing data
 - Billing for water/electricity/gas
 - Paperbased and e-bills
 - Consumption
 - Statistics about used resources
 - Printing
 - General printing purposes for all kinds of documents
 - Service and Vacation planning
 - HR system for your services teams
 - Purchasing and vendor management
 - Support for your purchasing department incl, contract management
- Copy the content of the DVD on your laptop

↘ INNER SOURCE

➤ Aim

To create a scalable eco-system in which teams can efficiently develop and maintain the diversities on others' components that they need

How:

- Introduce
 - flexibility in different timing & priorities between teams
 - flexibility in (start/change/stop) collaborations between teams across departmental and geographical boundaries
 - ability to adapt another's component to one's needs
- Leave flexibility to teams w.r.t. their internal processes

➤ Approach

The approach essentially

- Deploys key aspects of Open Source software engineering
 - open within community scope
 - explicitly defines decentralised ownership
 - patch rather than work-around
- Without changing or replacing existing mechanisms, e.g.
 - commitments & responsibilities
 - roadmaps
 - escalation mechanisms
 - shared architecture

↘ Evolutionary Roles

Flavours of users:

- User
 - uses the software as is
 - occasionally tweaks configuration settings or files a defect
- Contributor
 - much beta testing
 - files bugs and enhancement requests including patches
- Committer
 - contributes regularly
 - has write access to repository
 - has a binding vote

↘ Evolutionary Roles (continued)

Typically, a person's role in a project evolves

- User > contributor > committer
 - only some users evolve to contributor and some contributors to committer
 - but there is no committer that not once has been a user
- Today's user can be tomorrow's contributor, so treat him as such:
 - Enable & nourish this evolution by creating the right mindset and team spirit: enable by being open; nourish & stimulate by giving credit quick and publicly and being forgiving for mistakes

↘ Openness – Ease of Access

- Provide read-only access to *all* information relevant to development
 - source code
 - defect reports
 - enhancement requests
 - developer discussions and decisions
 - documentation, such as requirements & design
- Allow users to submit defects & enhancement request
- Access is not a true/false matter; access has to be easy! For instance,
 - require only a web browser to browse information
 - key documentation and a single point of entry to guide users to information

➤ Openness – Key Documentation

Three key documents to guide users and contributors:

- **Readme:** what is it
 - project description
 - who is involved
- **Install:** how to install it
 - run-time dependencies
 - installation instructions
- **Contribute:** how to build and contribute
 - build time dependencies
 - build instructions
 - how to contribute

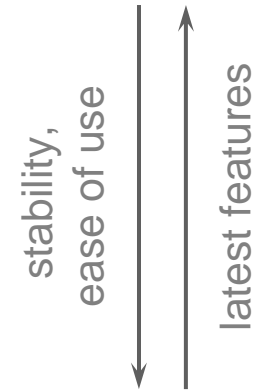
Make them easy accessible:

- in the root of the source tree + on the project home page
- plain text
- concise and to the point
- cover only the typical case(s), not exceptions and rare cases

➤ Openness – Release Early, Release Often

Provide a range of releases:

- Bleeding edge
 - eases developer discussions
 - targets developers
- Snapshots, daily or weekly
 - typically automated builds & regression tested
 - enables beta testing and patching
 - targets contributors
- Major & minor releases
 - targets end users



- Enables different users to work with the version that is most suitable for them

➤ Openness – getting started

- Start with a phased approach
- Phase 1 – consolidate the existing information
 - For the parts of the software stack identified as a commodity expose the documents, source code etc. to the rest of the world
 - Make sure you have a search engine in place
 - Tools should only be replaced when necessary, for example because they allow an easier access to the information
- Phase 2 – work publicly exposed
 - All email traffic for teams working on a commodified part should become publicly accessible. Allow also interaction between user and developer.
 - Expose project plans and feature lists early in the process
 - Even the bleeding edge of the development lines should become visible
 - One important aspect is to monitor and if necessary moderate the communication
 - Culture of Merciness

➤ Openness – getting started (cont.)

- Phase 3 – repeat the previous phases for commodifying components
 - Ensure that you've reviewed the access model
 - Might need more limitations for IP reasons
- Phase 4 – Drive the first projects to a patching model
 - Once you've identified components with a lot of contribution and activity outside of the core project team, try to identify first opportunities for a direct contribution
 - It is key that you cross this gap sooner or later as this is one of the key contributors for improvements
- The duration of each phase can not be clearly determined
 - The adoption of change differs
 - Projects will move with a different pace
 - Ensure that every success get's clearly communicated as this helps you in identifying and supporting champions

➤ Decentralised Ownership & Control

Given team A develops asset and team B needs to adapt it:

- Team A owns asset
 - maintains asset
 - decides whether to accept or reject a patch
- Team B owns patch
 - maintains patch
 - acceptance into asset transfers ownership to team A
 - decides whether or not to submit the patch
- Allow, and actively enable, patching to
 - provide flexibility in timing and priorities between teams
 - increase efficiency (more efficient than workaround)
 - increase amount & quality of feedback
- Optionally, disallow forks if existing management hierarchy already fills this role

➤ Decentralised Ownership & Control (continued)

This definition of ownership is carefully chosen to

- Balance stimulating convergence & allowing divergence
 - ensures only necessary diversities are created
 - patches typically live limited time before either being accepted into main line or fade away
- Keep approach scaleable
- Based on the experienced results a successful adoption of a decentralised ownership & control fosters re-use
 - Design for Re-use often fails
 - Although components are interesting for re-use missing information and the capability to tweak the component for an even better fit leads to a re-write from scratch
 - Decentralised ownership and control is mandatory, but not sufficient

➤ Patch Rather Than Workaround

Options to implement similar functionality

0. Use as is

1. Patch

- a) accepted into the main line
- b) rejected from main line

2. Work-around or glue code

3. Fork

4. Re-develop from scratch

- In large enterprises, current practice defaults to 2
- Balance amount of effort against amount of control required

➤ Patch Rather Than Workaround

Options to implement similar functionality

0. Use as is

1. Patch


- a) accepted into the main line
- b) rejected from main line

2. Work-around or glue code

3. Fork

4. Re-develop from scratch

increasing amount
of duplicate effort
& control



- In large enterprises, current practice defaults to 2
- Balance amount of effort against amount of control required

➤ Patch Rather Than Workaround

Options to implement a function

0. Use as is

1. Patch

- a) accepted into the main line
- b) rejected from main line

2. Work-around or glue code

3. Fork

4. Re-develop from scratch

- In large enterprises the defaults to 2
- Balance amount of effort and amount of control required

The 'holy grail'
of software
engineering

Minimal effort to
get where you
want to be

Worst option:
wasting effort
without gain

Invest effort
to gain control

Invest effort to
gain control and
avoid legacy

➤ Patch Rather Than Workaround (continued)

- Define and adhere to clear guidelines (not rules!) for accepting patches, such as
 - clean design
 - easy to maintain
 - needed by broad set of users
 - fits architecture
- Always provide the reason for rejecting a patch

Summary

- Be open
 - provide read-only access
 - provide key documents to guide users
 - release early & often
- Evolve your users into contributors and committers
- Define and adhere to strict ownership & control
 - decentralised
 - scalable
- Patch rather than workaround
 - making re-use feasible

Successful collaboration is about allowing (short term) divergence while stimulating (long term) convergence

➤ Exercise # 2 – Defining access and control

- Try to define a permissions matrix
 - Keep the following dimensions in mind
 - Level of commodification
 - External/internal team member
 - Project phase
 - Discipline
- How would you split up and organize your core development team?
- Again, there is no right or wrong – just a radical or conservative approach
- Please get prepared to discuss the consequences



How to deal with requirements and project plans

➤ Managing requirements for commodified software

- Successful initiatives moved to a democratized model
 - All members of the community are allowed to raise requests
 - Requests are converted into requirements by the committers
 - Everybody can contribute to the analysis/design efforts
 - Committers are moderating and reviewing the discussions and contributions
- Light toolsupport
 - No fat client based toolset
 - Internet-based tracker is the preferred choice
 - Support for voting
 - Depending on the charta either all members or the committers can vote for requirements to be implemented
- The overall consideration has to be the stability of the commodified software

➤ Managing requirements for commodified software

- Another important aspect for reviewing and deciding is the expected effort
 - To make a justification you have to have a realistic effort estimate and business benefit
 - A benefit can be a strategic advantage or a tactical win
 - Commiters have to ensure that high-demand requirements do not exceed the capacity of the core team
 - „Don't block the funnel“

➤ Managing requirements for commodifying software

- No need to change the current process immediately, but you should get ready for pushing the trigger
 - The decision criterias have to be aligned over a short to midterm period with the commodified software
 - Open the requirements process to a broader audience – you need to identify core committers
 - Over time, the power to decide needs to turned to the broader team
- When to start with the transition process?
 - No scientific approach yet
 - If the number of functional enhancements exceeds 30% of the total number of requirements, it's too early
 - If you see a lot of contribution on the mailing lists or around requirements, this a first indicator for commodification
 - If you see rising contribution (patches, tests) outside of the core team, you are likely in the middle of a commodification

➤ Why is it beneficial to move towards a community?

- A good foundation for answering the question is a book from James Surowiecki, „The Wisdom of Crowds“
- Key to the answer are the reasons why a large group of people is smarter than an elite few
- The four basic ingredients are
 - Diversity of opinions
 - Independence
 - Decentralisation
 - Aggregation
- Interestingly, successful Open Source communities have build a charta that supports all of them

➤ Why is it beneficial to move towards a community?

- Diversity of opinions
 - Establishes a good system of control and balance, eliminating the extreme
 - Stimulates discussions
 - Might lead to an evolutionary competition, as the right to fork is an essential foundation of OpenSource
- Independance
 - Reduces the pressure on teams responsible for a component
 - Releases are not made because of interdependencies
 - Again, if necessary a fork is a valid option; commodified components offer a high level of freedom

➤ Why is beneficial to move towards a community?

- Decentralisation
 - Reduces the risk and vulnerability of a bottleneck
 - Good example: Apache foundation
 - Questionable example: Linux
 - Group inside of a community can compete for the best solution
- Aggregation
 - Makes a community more powerful than a limited team
 - You have 100%+X resources (core team and contributors)
 - Ensures that a community becomes self-corrective if necessary as all activities and contributions are visible

➤ Planning releases

- Release plans for commodified software should be made in an OpenSource way
 - Voting sets the topics
 - Quality of the changes is the final criteria
 - Committers need to foster the development process
 - Sharing releases early in the development cycle is again a best practice
 - Going for heartbeat releases seems to be the most successful approach
 - Users can rely on the fact that they get a new release
 - Openness of the development progress helps to manage expectations
- Release plans for commodifying software can again be done as usual
 - One attribute for requirements should be „Increases reusability“ , give them priority whenever possible.
 - Prepare the users in case you switch from a feature driven release model to a release date driven approach

➤ The release process for commodified software

- Before focusing on the release you have to establish a working model during the development phase
- To achieve the best results a review of the existing SCM processes is essential
- Based on the observations it is more important to organize the version control in line with Open Source principles
- Linking work results back to the requirements is the only real requirement we have seen

➤ Basic Branching Strategies

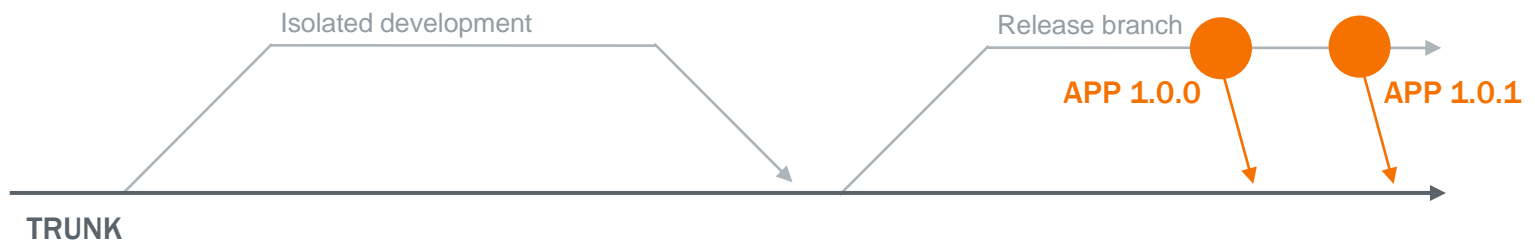
- Unstable Trunk – serial release model
- Stable Trunk – parallel release model
- Agile Release - delayed release definition model

Questions to answer:

- When do you branch and merge?
- What is on each branch?
- Use cases

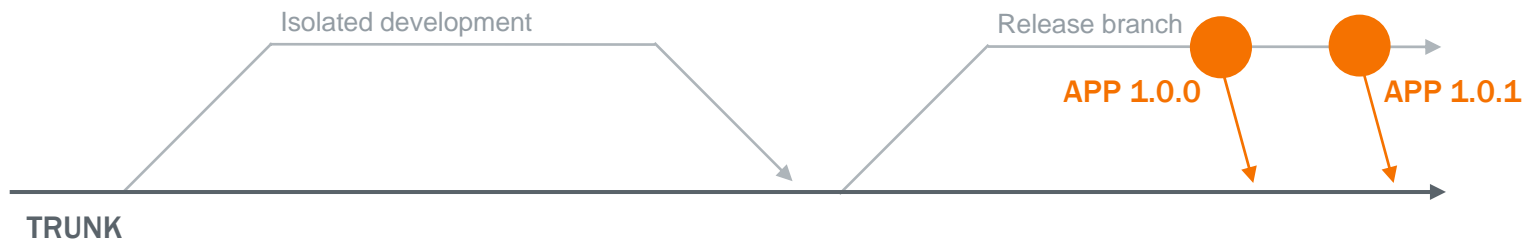
➤ Unstable Trunk

- When to branch:
 - At the feature complete point for release branches
 - At the point work needs isolated from general development
- When to merge:
 - At the commit point for the next release
 - At release points for both production and bug fix releases



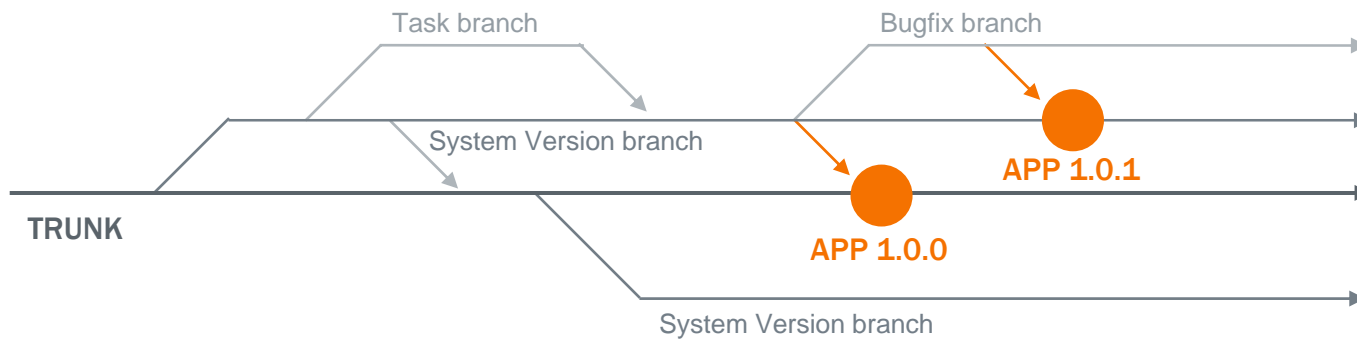
Unstable Trunk

- What's on the branch
 - Trunk - ongoing development of next general release
 - Release branches - formal promotion process
 - Isolated development branches - work that is either experimental or not accepted for inclusion in the next release
- Use cases
 - Waterfall, serial development (e.g., Subversion itself)
 - Single customer with small team (e.g., internal infrastructure projects)



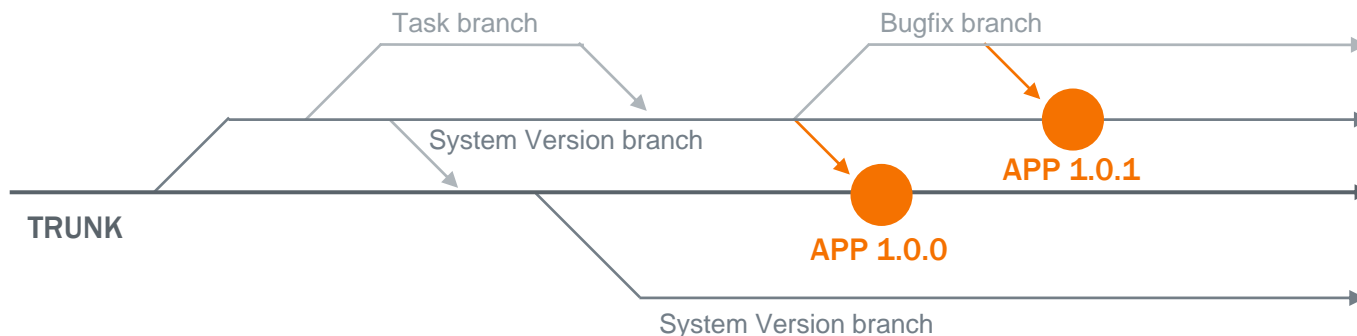
➤ Stable Trunk

- When to branch:
 - At the release definition point for system version branches
 - At the point work needs isolated from general development
- When to merge:
 - At the start of system version branches
 - At the release points both production and bug fix releases



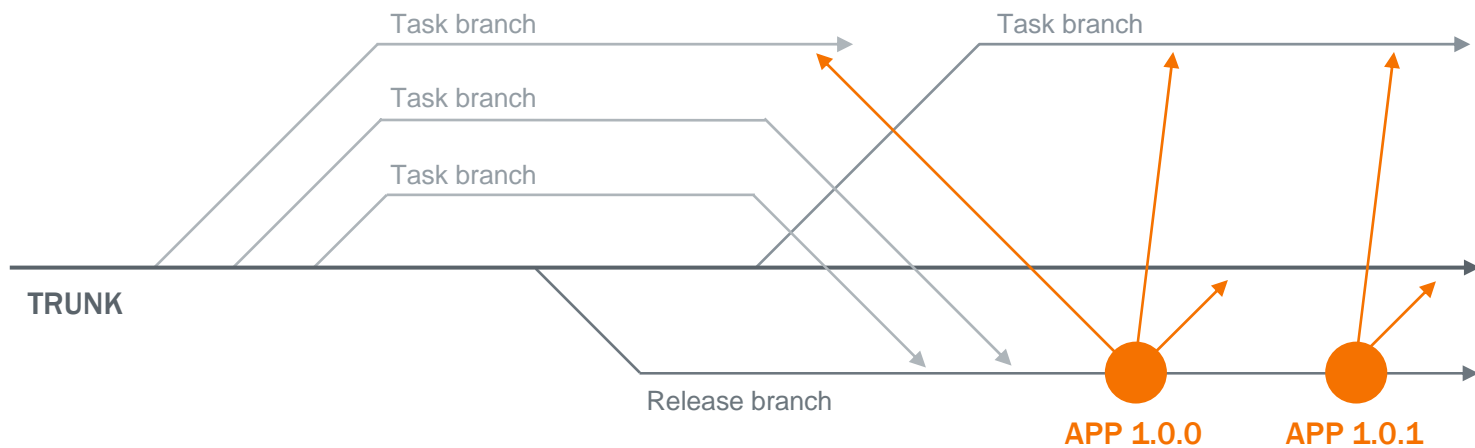
➤ Stable Trunk

- What's on the branch
 - Trunk – stable points of releases and branch starting points
 - Release branches - reflect release development process including formal promotion process
 - Task branches - work that requires long periods of instability
- Use cases
 - Waterfall, overlapping development (e.g., most major software products)
 - Larger teams



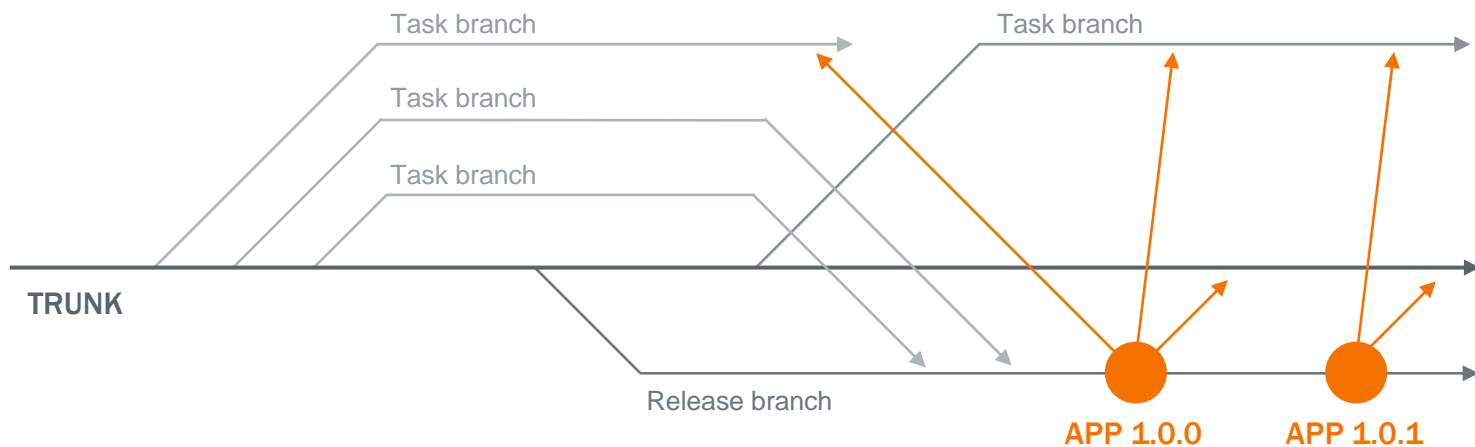
Agile Release

- When to branch:
 - At the release definition point for system version branches
 - At the task definition point for clear units of work
- When to merge:
 - At the definition of releases
 - At release points both production and bug fix releases

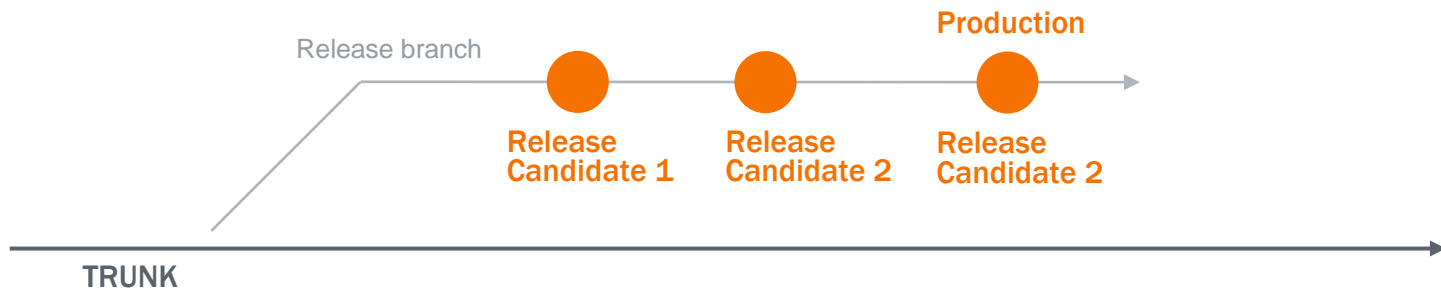
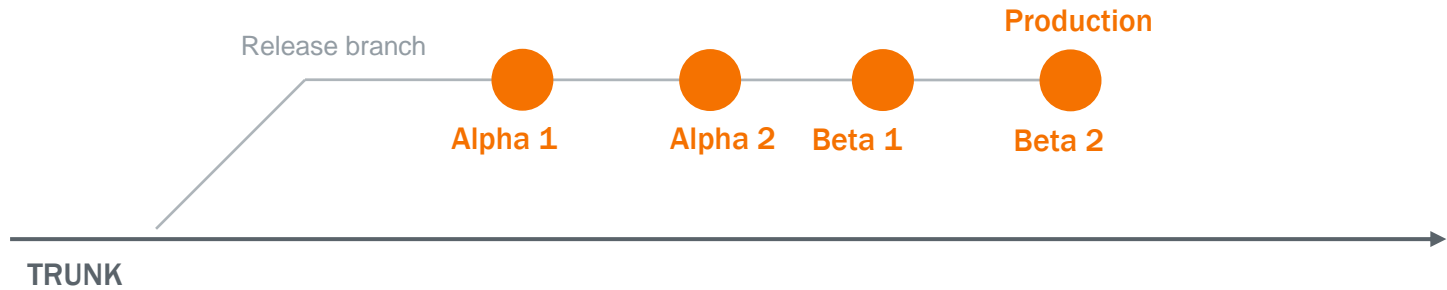


Agile Release

- What's on the branch
 - Trunk – stable points of releases and bug fixes
 - Release branches - reflect release development process primarily the formal promotion process
 - Task branches – definable units of work
- Use cases
 - Agile development and release (e.g., Internet applications)



➤ Promotion Models



➤ Example: the Greenhouse approach

- Requirements are no longer assigned to a fixed release or date
- The community starts to work on multiple activities
- The final decision what makes it in the release is done very late in the development cycle
- You are „harvesting“ what is mature – not what the calendar says ...

➤ Moving from „commodifying“ to „commodified“

- One of the most tricky questions is when is the right time to pull the trigger and declare an object as commodified
- One key criteria is the definition of clearly defined interfaces
 - Elimination of dependencies that are out of control
- Use opportunities to set industry standards
 - Controlling standards is a huge competitive advantage
 - Ready from the very beginning
 - Controlling architectural changes limits the capability of competitors
 - Examples: BlueRay – HD-DVD, Dicom

➤ Moving from „commodifying“ to „commodified“

- Componentization is a key success factor
 - Supports IP governance
 - Independant release cycles are key, otherwise roadblocks occur
 - If a software is not componentized it is not a good candidate for an external collaboration
- Be sure you have the right support for your OpenSource flavour
 - Communities of individuals
 - Look at successful Open Source communities
 - Communities of enterprises
 - Tools must have a higher support for role-based access control
 - Right approach for setting standards in verticals
 - Example: DrK initiative Open Adaptor

➤ Exercise #3 – Managing traceability

- Work in your group to define a traceability chain
 - Pick one commodified component and a commodifying component
 - What information would you track starting with a requirement, an associated task and the artifacts (code, documents etc.)?
 - What gateways would you establish along the development lifecycle
 - This, of course, implies a short description of a lifecycle
 - What timeline and milestones would you set along the commodification?
 - Be ready to present your results!

➤ DEVELOPMENT ENVIRONMENT

Development Environment

Primary functionality areas:

- Version control
- Tracking
 - defects
 - enhancements
- Information
- Communication
 - asynchronous
 - synchronous
 - announcements
- File sharing

Access to assets:

- *Easy accessible from anywhere at anytime*
 - archived
 - searchable
 - referential stability (ability to refer to an object and this reference staying stable across time)
 - single point of entry
 - user identity (single sign-on)
- *Controlled access*
 - ability to limit access to parts of projects
 - scaleable authorisation scheme, human manageable/preventing human mistakes

➤ Development Environment (continued)

- Availability
 - 24/7 monitoring & support
 - > 99% uptime
 - backup & disaster recovery
- Security
 - security is as good as the weakest link; a firewall at the front door does not suffice, security is an aspect of all parts of a system
 - scalable & manageable access rights & permissions; read-only open while strictly controlled write access
- Performance & scalability
 - WAN capable: the ability for two people to touch the same code at the same point in time, regardless of their location or organisation
 - thousands of users
- Speed of deployment of a project
- (Predictability of cost)

➤ Team Communication

- Start communicating and documenting electronically from day 1
 - Replaces point-to-point communication and makes the decision process transparent
 - Mailing list technology might look outdated, but is actually one of the real powerful vehicles to foster Inner Source
 - Allows asynchronous communication
 - Helps new team members to do a „time warp“
 - Example: OpenOffice
- Use discussion forums for communication outside of a project context
 - Assembles information about technologies, algorithms etc.
 - Supports the adoption

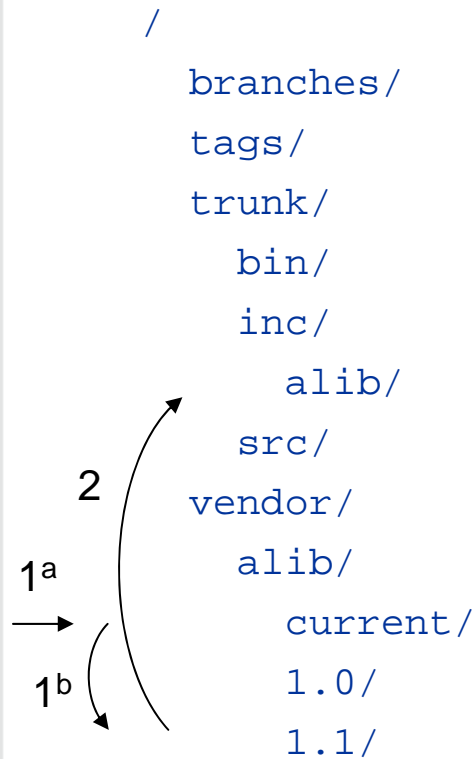
➤ Collaborating with Other Teams: Managing Patches

- Need for teams to collaborate, using (a customised version of) another team's component
 - Inside a repository: branching & merging
 - Across repositories: 'vendor branching'
- There are various (non technical) reasons why working in one repository is not a suitable option. Key challenge for collaborating effectively is how to maintain a set of customizations/patches on another team's component.

Goals:

- Traceability
 - What version are we using?
 - What are our customizations?
- Control over when to introduce another team's release to your team

➤ 'Vendor Branching'



- Organize the tree as follows:

- /vendor/alib comprises successive code drops
- /trunk/inc/alib comprises current version with our local modifications

Standard cycle to introduce a new version from an external team:

1. Bring in the latest code drop from vendor
 - a. Import code drop into /vendor/alib/current, potentially. using `svn_load_dirs.pl`
 - b. Tag code drop, copying into /vendor/alib/<version>
2. At the suitable time for the project, introduce latest code drop

Merge the latest (aka, diff between <version> and <previous version>) into /trunk/inc/alib

Key Points

- Stay agile/make sure you can adapt to changing circumstances
- KISS:
 - Single server, potentially a second server strictly for business continuity
 - Keep authorization scheme simple
- Clear structure
 - Various maturity levels => group component release cycles to maturity level and use system of sub systems to release product
 - Use 'vendor branching' to keep track of the patches on the external components
- Leverage heterogeneity:
 - The outsourced part will be focussed on testing and integrating standard blocks: slower release cycles, more heavy process.
 - The newly acquired and recently acquired part will have much less process and release fast.

↘ THANK YOU