The Flow of Change

Branching and merging in the face of agile development, extreme programming, team collaboration, and parallel releases.

Laura Wingerd • Perforce Software • www.perforce.com

Presented at:

SD West, 18 March 2005

Perforce User Conference, 15 April 2005

What we'll cover

- The ideal world vs. the real world
- Codelines and modules
- The "tofu scale"
- The "baseline protocol"
- The "golden rule of collaboration"
- The myth of merging
- Why we don't drive through hedges

Ideas you'll come away with

- How to plan for branching and merging
- How to simplify a complicated branching and merging scheme

Software development in the ideal world

- There are no bugs
- We have all the time in the world
- Schedules never slip
- The first release is perfect
- Customers always upgrade

Let's look at we do make the real-world more like the ideal world

Why look at the ideal world?

1. Because we often get so enmired in procedure that we forget our true objectives. Instead of shooting for our true goals we try to refine and follow procedure.
2. Because the ideal world models simplicity. We can always use more simplicity.

In the ideal world, creating a software product is simply a matter of developing it and releasing it.

We start out with nothing, and over time, a body of code develops.

When we're done, we release what we have.

In the ideal world we make one release

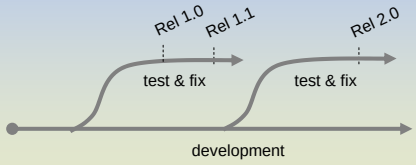
- In the real world one release is never enough
- What we can do about it: make periodic releases

One release is never enough because:
•Customers want more features
•Technology changes
•We didn't meet the requirements exactly
•We didn't understand the requirements exactly
•The requirements have changed

So in the real world we make periodic releases.
This brings us closer to the ideal world by giving us an essentially infinitive amount of time to produce the perfect software.

In the ideal world there are no bugs

- In the real world we need time to stabilize releases
- What we do about it: branch for each release



In the ideal world there are no bugs.
In the real world, there are bugs, and finding and fixing them ("stabilizing" our software) occupies a significant chunk of the schedule.

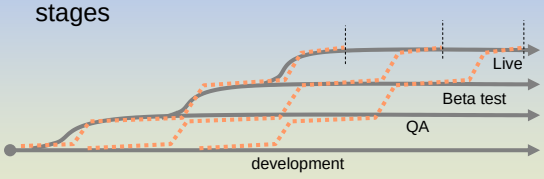
We need to test & fix bugs before, during & after release
- •We can't stabilize and develop in the same codeline. (Development *introduces* bugs!)
- •Unfortunately we can't stop development while stabilizing, because in the real world we have deadlines

So we branch a release codeline.
We fix bugs in the release codeline and make releases from the release codeline.
We merge bug fixes from the release codelines into the mainline.
- •The release codeline evolves toward our ideal-world goal: a stable, bug-free release
- •Development in mainline continues uninterrupted, as it would in the ideal world of no bugs
- •The mainline gets the benefit of the release codeline's stabilization every time a bug fix is merged from a release line into it

•By the way, this helps us achieve another ideal-world goal in the real world:
- •In the ideal world every customer upgrades immediately; in the real world customers are slow to upgrade
- •We can't support customers on old releases with code that's evolved beyond that
- •Branches releases gives us a way to support our recalcitrant customers

In the ideal world we have all the time we need

- In the real world our release cycles can be *very* short
- What we do about it: release in overlapping stages



In the ideal world we don't have to worry about deadlines.
In the real world we have release schedules – sometimes we have *extremely* short release cycles.
•Web content and web-hosted programs are examples of software that have very short release cycles.

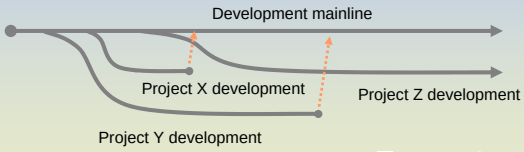
The branch-on-release model doesn't work very well for extremely short release cycles.
(If we're releasing content once or twice a week the number of branches we have to manage would quickly get out of hand.)

Our real-world strategy for short, frequent releases is to deploy content in *stages*.
- •Instead of branching for each release we have a limited number of "reusable" branches through which we shunt code and content as it passes through deployment stages

For example:
- •We branch the development line to a QA line where we can subject it to a barrage of testing. Meanwhile, development continues in the development line.
- •If what's in the QA line passes muster, we branch it into a beta line that is served up on our beta website. That frees up the QA line, and we can copy the latest development into it and start testing.
- •When the beta testing is done, we branch the beta line into the live line, to be served up on our production websites. That frees up the beta line; we copy content from the QA line into it. That, in turn, frees up the QA line; again we copy latest content from the development line into QA for testing.
- •And so on. This allows us to make an unlimited number of releases with no more than three release branches.

In the ideal world all development finishes on time

- In the real world there are unforeseen delays
- What we can do about it: decouple development projects and branch for each project



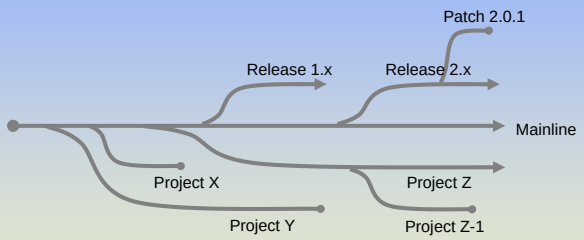
We've been talking about "development" as if it all happens in the same branch.
That would be okay in the ideal world, where all development finishes on time.
In the real world there are unforeseen delays:
- Late/incomplete deliverables can hold up entire releases
- Builds broken by one developer/group can hold up another developer/group

What we can do about it:
- We decouple big deliverables into separate development projects
- We branch development codelines, one for each development project

Here, for example, the mainline is branched into the Project Y dev line, and shortly thereafter into the Project X line. Some developers work on Project X and some on Project Y. Development changes flow to the mainline when done. (We'll see more on this presently.)

Branching into develop codelines brings us closer to the ideal world because:
- Development is not delivered to the mainline until it's buildable and testable. As in the ideal world, the mainline moves forward in increments of buildable, testable changes.
- Development proceeds in each dev branch without being subject to the interim (and likely, broken) changes in the other dev branches.
- If one development project doesn't get finished on time, it doesn't hold up the entire release. We can still release the other finished projects. (This assumes a certain amount of independence between projects, of course.)

The mainline model

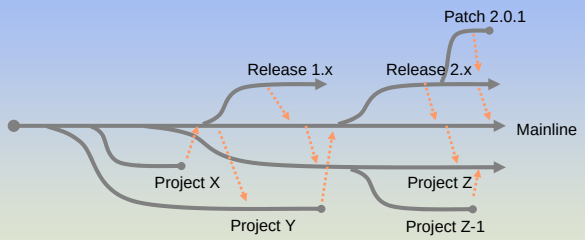


Now we begin to recognize the mainline model software development lifecycle:

•A main codeline forms the trunk from which be branch release and development codelines.

Note that release codelines can branch into patch codelines, and development codelines can branch into sub-project codelines and private branches.

The flow of change

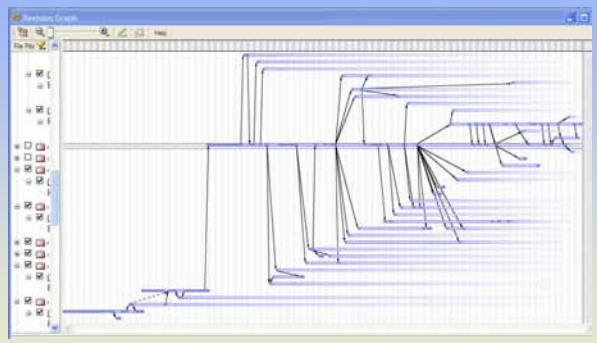


•Changes are propagated from one codeline to another for an obvious reason: we don't want to have to do the same coding over and over again.

•This propagation is the *flow of change*.

•In a simple system like this, it's not hard to track or predict the flow of change.

Chaos?



But what about the real world, with thousands of files and changes?

Here, for example, we see a revision graph of a single file. (This is produced by P4V's Revision Graph feature.)

It shows us the file was branched into a couple dozen codelines and changed probably thirty times altogether.

•But it doesn't tell us where to make the next change.

•Nor does it tell us where to merge the change once we've made it.

Does branching and merging just result in chaos in anything but a very simple system?

Not necessarily…

- Maps, protocols, convention, and etiquette make driving manageable
- Maps, protocols, convention, and etiquette can make codelines manageable

Not necessarily.

Branching and merging are a lot like driving.

Driving is actually extremely complicated but we don't perceive it as such.

Why not?

Because maps, protocols, convention, and etiquette make driving easier.

For example:

- •Drive on the right
- •Signal before turning
- •Stop on read, go on green

"Stop" sign is a shorthand for several protocols we know:

- -Two-way stop
- -Four-way stop
- -Traffic light

There are protocols that make branching and merging easier.

We'll discover some of them in this talk.

[Terminology]

- How is a *codeline* different from a *branch*?
  - *codeline* (a.k.a *stream*) = concept
  - *branch* = implementation
- A codeline's parent is called its *baseline*
  - Other names for *baseline*: backing stream, integration branch, base, origin…
- The *mainline* is the codeline that has no baseline
  - A.k.a. the *trunk*

Before we go on, let's go over some terminology. Terminology is particularly difficult in branching and merging.
•Many vague terms sound the same but vary according to version control system
•There's no standard implementation from system to system
•Many different terms are used for essentially the same thing
For the purpose of this session, let's try and nail down what we mean by "codeline", "branch", "baseline", "mainline", and "module".

Conceptually, a codeline models a version of the *whole system* whereas a branch is a way of implementing a codeline. You could implement a codeline with:
•A full branch – i.e., all files branched from baseline to codeline
•A sparse branch implementation -- some files actually branched and the rest mirroring the baseline

Note that a codeline's parent is typically the codeline it was branched from.
•A codeline's parent can't change but in many version control systems its baseline can.
•In other words, you can "rebase" a codeline to give it a different baseline. (Which you'd do if you wanted change to flow differently, as you'll see in a bit.)

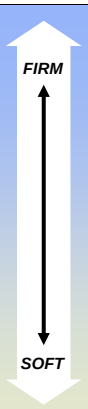
[A codeline by any other name…]

- Streams
- Active development lines
- Feature branches
- Task branches
- Staging codelines
- Private branches

These are some of the many names by which we also call codelines, depending on the version control system we happen to be using.

The "tofu scale"

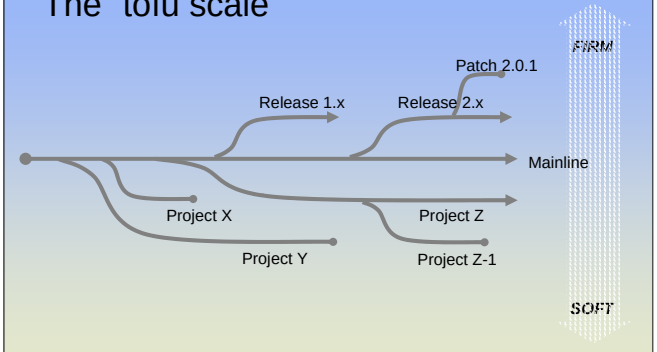
- Firm codelines:
  - Very stable, thoroughly tested, close to release
- Soft codelines:
  - Unstable, barely tested, distant release date



Now lets talk about maps, protocols, conventions, and the rest.

A most useful conventions in mapping codelines is the tofu scale.
It's an assessment of stability, "testedness", and tightness of schedule.
In other words, it measures the risk of change to a codeline.

The "tofu scale"

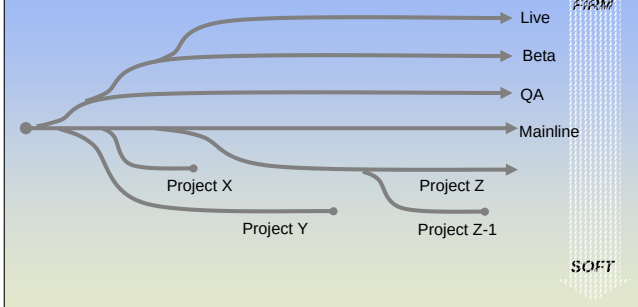


Every codeline has a relative "firmness" with respect to its baseline.
When you draw a codeline diagram you can show the relative firmness of codelines by putting the firm codeline on top and the soft codelines on the bottom.
•The Rel 1.x codeline is firmer than the mainline, for example, because it's subject to extensive system tests – all of which have already been run, by the way -- and it's code that will be soon (or is already) in the customer's hands.
•The Project Z codeline, other the other hand, is softer than the mainline. It's not subject to rigorous system tests, only unit tests.
•The Project Z-1 codeline is even softer than Project Z. It happens to have been branched to support a side-project of Project Z and it doesn't even have unit tests to run.

When codelines are mapped according to the tofu scale we can see at a glance where the risk of change is.
•A change to Patch 2.0.1 or Rel 2.x, for example, would be pretty risky, in terms of schedule and quality.
•A change to Project Z wouldn't be very risky, and a change to Project Z-1 would be least risky.

Note that inferring relative firmness between sibling codelines is a mistake in a two-dimensional graph like this:
•Project X isn't necessarily firmer than Project Y. All we can be sure of is that both of them are softer than the mainline.

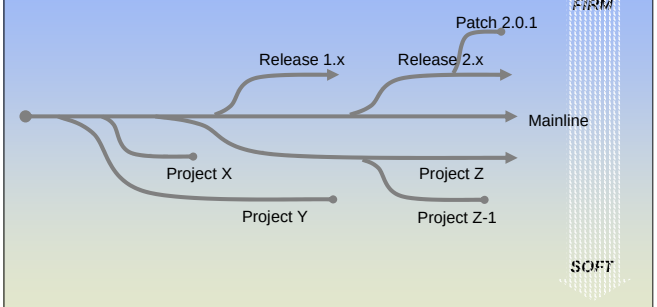
The “tofu scale”



And here we see the staging codeline model plotted on the tofu scale:

- The QA stage is firmer than the mainline
- The beta stage is firmer than QA
- And the live line is the firmest codeline of all in this system.

From timeline…



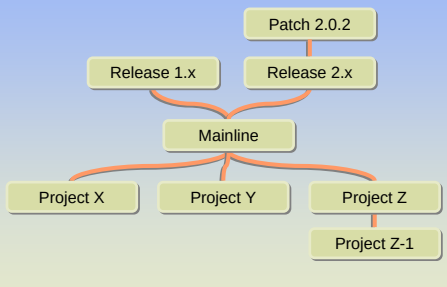
The diagrams we’ve been using are essentially timelines, of course.

We saw a moment ago that one problem with this kind of diagram is that it can be misleading when it comes to the relative firmness of sibling codelines.

Another problem with it is that it’s showing us what happened in the past, not what should happen in the future.

There’s another way to show codelines…

…to baseline map



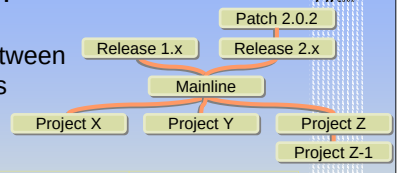
…and that is with a “baseline map”.

A baseline map gets rid of the clutter of history and shows two things:

- The relative tofu rank (and hence the risk of change) of sibling codelines
- How change *should* flow (in the future, as opposed to what happened in the past)

The “baseline protocol”

- Change flows between a codeline and its baseline



When codeline is…	Change flows *to* baseline	Change flows *from* baseline
Firmer than baseline	Continually	“Never”
Softer than baseline	At points of completion	Continually

A baseline map also reveals the “baseline protocol”.

The baseline protocol is this:

- Change flows between a codeline and its baseline.
- In the firm-to-soft direction, the flow of change is *continual*.

- For example:
  - Changes (bug fixes) in the Release 1.x codeline will be merged to the mainline ASAP after they’re checked in.
  - Changes in the mainline will be merged to development codelines ASAP

Thus, a change to a firmer codeline has a stabilizing effect on its softer baseline.

Note that:

- All change flows to the mainline. (Bringing us back to the ideal world…)
- From the perspective of each codeline, the baseline looks like the mainline.

The Golden Rule of Collaboration

- Always accept stabilizing changes
- Never impose destabilizing changes

This protocol, by the way, can be summed up as “The Golden Rule of Collaboration”…

Release codelines

- Flow of change *to* baseline is continual
- Every improvement to a release codeline is an improvement to the baseline

Let’s take a closer look.

What kinds of changes get checked in in a release line?

•Bug fixes and patches. They flow to the baseline continually. (That is, they’re merged to the baseline ASAP after they’re checked in to the codeline.)

•The effect is: every change to stabilize a release codeline has a stabilizing effect on the baseline

Release codelines

- No change flows *to* a release codeline from the baseline!
  - Release codeline gets more and more stable
  - Baseline changes don’t put release codeline at risk

Change “never” flows to a release codeline from its baseline. (We say “never” in quotes because this is a frequently violated protocol.)

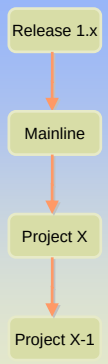
Why no flow from baseline to release codeline?

•The baseline is softer (less stable); change flowing from it to a release codeline would bring destabilization to the release codeline

Note that “flows continually” is the same thing as what we call “continuous integration”.

Development codelines

- Change flows continually from the baselines to development codelines
  - Changes flowing from firmer codelines have a stabilizing effect
  - Development codelines always have latest bug fixes and patches



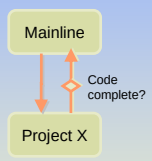
The baseline protocol says that change flows continually to a development codeline from its baseline.

The baseline map shows us the effect of this:

•changes to a release codeline have a stabilizing effect that trickles down to development codelines.

Development codelines

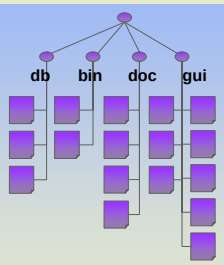
- Flow *to* the baseline from a development codeline is open sometimes, closed others
  - Open when development is “code complete”
  - Closed when development is incomplete or build is broken



Now, in the other direction, from a development codeline to the baseline, the flow of change is *not* continual.
•Development changes only flow to the baseline when they’re able to withstand the baseline’s tests.
•Sometimes we call this “code complete”, but it could be incomplete code, as long as it doesn’t destabilize anything in the baseline. (A better way to say this might be “point of completion”.)
When change *does* flow to the baseline, is it propagated by merging? By copying? We’ll get to this in a moment…

[Terminology]

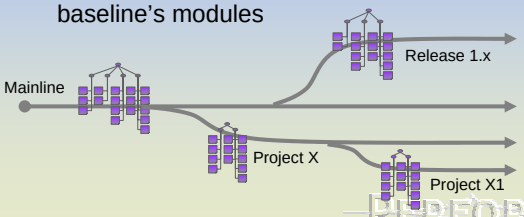
- What’s a *module*?
  - A directory tree of files
  - Corresponds to the set of files needed on local disk to work on a part of the software
  - Also called: *source tree*, *component*, *subsystem*, *folder*



First let’s talk about modules.
What’s significant about modules is that they each have a structure.
•they define relative locations of files within them
•The development tools you use – compilers, debuggers, build tools, etc. – rely on module structures.
•In other words, the root of a module is usually the reference point of tools that operate on files.
Note that modules can be nested.

Modules and codelines

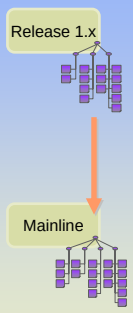
- A codeline is a collection of modules
  - Codelines “inherit” modules from baselines
  - A codeline may contain a subset of the baseline’s modules



A codeline is really a collection of modules.
When we branch a codeline we’re really branching some or all of its modules.

Modules and codelines

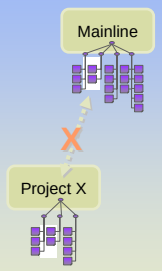
- A codeline is a collection of *relevant* modules
- Change flows between codelines by merging or copying changes between corresponding modules
- Module types to recognize:
  - “Active”, “inactive”, “private”



Each codeline is a collection of *relevant* modules.
When change “flows” between codelines, it’s really propagated by merging or copying files.
Not just any old files, but the files in certain modules.
Let’s recognize, for the sake of propagating changes, three types of modules:
•Modules that will be changed in the course of work in a codeline are *active* modules
•Modules that support building, testing & debugging are *inactive* or *private* modules

“Private” modules

- Will be changed in codeline
- Mimics structure of parent
- There is no flow of change between codeline and the baseline
- Example: “bin” directory



Private modules are branched from the baseline into the codeline. (Or they may be created from scratch within the codeline, but in any case, their structures mimic the structures of their counterparts in the baseline.)

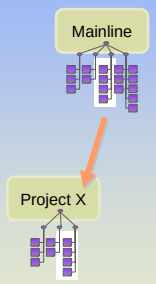
•Private modules will be changed in the codeline. In other words, people will be checking changes in to them.

•However, changes to private modules aren’t merged or copied from one codeline to another.

A typical example is the “bin” directory in a source tree – that’s essentially a private module. Nightly builds in a codeline check files into the “bin” module, but nothing in the codeline’s “bin” module is ever propagated to or from the baseline’s “bin” module.

“Inactive modules”

- Inactive modules support debugging, testing, building
- Change always flows from the baseline to the codeline
  - Inactive modules inherit baseline changes
- Not changed by work in codeline
  - No change flows to baseline



Inactive modules won’t be changed by developers working the codeline. They play a supporting role – that is, they provide the files needed to debug, build, and test the software in the codeline.

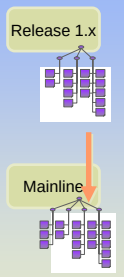
•However, they may be active in the baseline. (That is, they may be changed by developers working in the baseline.)

•When inactive modules change in the baseline, they will change in exactly the same way in the codeline.

•An inactive module in the codeline is essentially a mirror of its counterpart in the baseline.

“Active” modules

- Active modules are the modules we’re working on in a codeline
- Change can occur in both codeline and baseline
- Change always flows to baseline



Finally, “active” modules are the modules we plan to work on in the codeline. They’re the reason we branched the codeline.

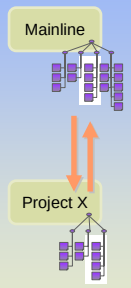
Remember, the real-world codeline is our surrogate for the baseline. In the ideal world, we’d be working in the baseline. But since we can’t work in the baseline, the next best thing is to work in a codeline that looks as much like the baseline as possible.

- •Thus, to cleave to the ideal world, change to active modules in the codeline must flow to the baseline.
- •*When* does it flow to the baseline? According to the baseline protocol:
  - •Continually, if the baseline is softer
  - •At points of completion, if the baseline is firmer

Release codelines are firmer than their baselines, so as active modules change in a release codeline, their changes flow continually to the baseline. (The mainline, in this diagram.)

“Active” modules in development codelines

- Change flows from baseline to codeline
- Change flows from codeline to baseline
- Active modules – but only the active modules – eventually need merging



•And in development codelines, which are softer than their baselines, change in active modules flows from codeline to baseline at points of completion.

•However, active modules can change in the baseline as well. And according to the baseline protocol, change flows continually to a development codeline from its baseline.

•So if active modules are changing in both codeline and baseline, and change is flowing in both directions, this can mean only one thing:

- •Active modules will eventually need merging.
- •(That is, the *files* in active modules will eventually need merging.)

•But interestingly, *only* the active modules need merging.

- •The inactive modules need only be copied
- •the private modules don’t need anything at all.

The myth of merging

- Is merging dangerous?
  - Is *coding* dangerous?
  - Merging is as "dangerous" as coding
    - Can destabilize software
    - Necessitates testing
- Merging can be as safe as coding if done in the right codeline

•We hear a lot that "merging is dangerous". Is it really?

•Well, yes: automated merging can produce incorrect results, and manual merging can produce incorrect results.

•Merging can destabilize software, it can introduce bugs, and it necessitates testing.

•Are these good reasons to prohibit merging? Some would say it is.

•My question is this: Is *coding* dangerous? Of course it is. It can destabilize software, it can introduce bugs, and it necessitates testing.

•But do we discourage coding for any of these reasons? Of course not. We'd never get anywhere in software development if we didn't accept the risk of coding

•And is merging any riskier than coding? No. But as with coding, we only want to do it in the codelines that can accommodate the risk.

•That is, do the coding and the merging in the *soft* codelines.

"Merge down, copy up"

- *Merge* from firm codeline to soft codeline
- *Copy* from soft codeline to firm codeline
- Softer codeline can accommodate merging better than firm codeline can

•The way to make merging safe is to "merge down, copy up".

•Remember, "down" is going from firm to soft codeline. "Up" is going from soft to firm.

•Softer codeline can accommodate merging better than firm codeline can:

•Instability is more acceptable in the softer codeline

•The code in the softer codeline is further from the release date; there's more room in schedule for testing

"Merge down, copy up"

- Release codeline to baseline: merge

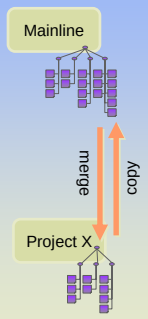
Release codelines, as we know, are firmer than their baselines.

•Thus to propagate change from release codeline to baseline we can go ahead and *merge.*

•The baseline, being the softer codeline of the two codelines, can better accommodate the risk of merging.

"Merge down, copy up"

- Development codeline to baseline:
  - *Merge* from baseline to codeline first, then…
  - Test (compile, proofread, etc.)
  - *Copy* from codeline to baseline



But when we're propagating change from development codeline to baseline:

•We *merge* from baseline to dev first, then…

•We test the merge result (compile, proofread, etc.)

•Having assured a successful merge, we *copy* from dev to baseline

•Now here's where etiquette comes in:

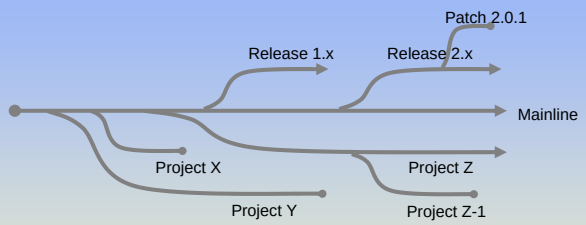
•While a developer is merging down to a development codeline, those of us working in the baseline have to hold of on checking changes in to the baseline.

•We wait politely until the codeline's changes are copied up to the baseline.

•Do we have to wait long?

•No, because remember: change flows *continually* into a development codeline from a baseline. So each merge is a small, incremental merge. Plus, even if the baseline's change was large, only the codeline's active modules will need merging.

Not chaos at all



- State of a codeline and how/when to merge is self-evident

Earlier we asked whether the flow of change is bound to be chaotic in the real world.

Let's look again at our (admittedly simplified) codeline diagram.

If we can count on the protocols, conventions, and etiquette we've just discussed, we can see that, in fact, it's easy to predict:

•Where a change should be made

•The risk of making a change in a given codeline

•How a change flows to other codelines once it's been made.

Why we don't drive through hedges

- You're on the freeway. Your destination is 100 yards from you on the other side of a hedge. The nearest exit is ½ mile away. Do you drive through the hedge to get to your destination?
  - Just as driving through hedges makes the freeway a confusing and dangerous places to drive…
  - Merging changes between arbitrary codelines makes the repository a confusing and dangerous place to check in your code

Meanwhile, here's what to say when a developer asks "Why *can't* I merge a change from my private development branch into the release codeline?"

Remember…

- Tofu scale: firm on top, soft on bottom
- Baseline protocol:
  - Change flows between codelines and their baselines
  - Tofu rank determines flow of change
- Protocol & etiquette of modules
  - Merging happens in active modules
  - Merge down, copy up
  - Be polite when merging is in progress
- Golden rule of collaboration:
  - Always accept stabilizing change
  - Never impose destabilizing change

The things to remember are:

•Use the tofu scale when drawing codeline diagrams

•Respect the baseline protocol

•It often helps to draw a baseline map as well as a timeline diagram

•Respect the protocol and etiquette of modules

•Don't forget the golden rule of collaboration

Laura Wingerd • Perforce Software • www.perforce.com