What is Trunk-Based Development?

What it is...
It is a branching model for software development. Historically, it has also been called “mainline” (see later).

It requires much more concentration and rigor, than making a branch (on the shared source-control server) to suit a whim. Though you could do it without Continuous Integration (CI), as many open source projects do, for enterprise development you have to have CI linked to the trunk, enforcing multiple aspects of “that commit was good”.

In this article, I’m saying nothing about what developers do on their own workstations by way of ‘local’ branching to suit their hour by hour activities. This is all about the shared repo, where multiple developers integrate/merge their daily work for the greater good :)
dominant. Google and Facebook, today, practice a TBD style branching model. If not exactly, then close enough. Either with Google and Facebook publicizing their TBD usage, the mindshare is growing for it. But there are branches!

Yes there are, but concerning releases. “Branch on/for Release” is the strategy. The release branch that will live for a short time before it is replaced by another release branch, takes everything from trunk when it is created. In terms of merges, only cherry-picks FROM trunk TO the release branch are supported. For many enterprises, only bug fixes will be merged. For Facebook (who go live from release branches ten times a week), merges of enhancements happen too, if prioritized by stakeholders.

Nearly everyone agrees that bugs are fixed on the trunk and merged to the release branch, instead of fixed on the release branch and merged to trunk. There is usually a reduced set of people that can commit to the release branch.

Pull-Requests are still branches!
OK, so GitHub pioneered the pull-request as a development workflow. This is quite compatible with TBD, in that a feature/task is marshaled in a place that is not yet on the trunk/master but can be quickly. Normally, code review happens there and CI weighs in automatically with an opinion as to whether the PR branch is eligible to be merged into the trunk/master or not. It everything is right, the the PR is merged back in the master/trunk and then deleted, leaving a smooth trunk/master timeline. Of course, the feature/task branch that’s subject of a pull-request later should be a one-person or one-pair branch, and very short lived (say a day). There are variations too - both ‘forks’ and simple branches of the origin are good pull-requests choices. Only repository read/write permissions guide which should be used.

Obligations for developers
Developers do not break the build with any commit. This requires a lot of discipline, and perhaps why the induction programs of Google and Facebook are lengthy for developers. Rollback/revert of a commit is a strategy to prevent the damage (lost time) from that. More sophisticated companies will use pre-commit verifications. Devs take on habit: prove the commit is good, by synchronizing to the the trunk’s latest revisions, building from root/scratch, double-checking their functional change, then committing. In the early days, including in ThoughtWorks, devs had a “token” to prove that they had not broken the build - nobody else could hold the chicken while they were going through that proving cycle. Rubber Chickens have been used for over a decade for this, but anything will do (thanks to Jez for the link).

Continuous Integration
Continuous Integration, like Jenkins, kicks in for that commit, and runs through a build pipeline building, testing, deploying, testing some more. It may detect failures, and most likely that is because a developer didn’t prove their commit or do the token thing. Another issue might be the developer failed to add a new source file before a commit. This would be easy/quick to remedy, and a situation where “roll forward” would be OK.

Changes that take “too long” to complete
Developers use a technique called Branch by Abstraction (BbA) to ensure that they can complete trickier changes over a longer timespan. I’ve written about that many times. Martin Fowler underlines it as important, and Jez Humble has written about it too. The risk it mitigates, is of the proliferation of branches, and those ‘temporary’ branches not completing on a schedule that has been hoped for.

My own case studies

From 2005, a move from crazy branching towards TBD (a US FX trading bank).
Trunk-Based Development recap
Quick reminder of what TBD is:

- Developers commit to a single trunk more or less exclusively
- Release engineers (or build-cop) create branches, and cherry-pick to branches more or less exclusively
  - Only if a defect cannot be reproduced on trunk, is permission given to fix it on the release branch, and cherry-pick back to trunk.

And if you get the release branch concept, it’s worth remembering:

- Trunk-Based Development means regular developers don’t commit to a release branch.
- Trunk-Based Development means you’re going to delete ‘old’ release branches, without merging them back to trunk.

What is definitely not TBD
Multiple branches that developers commit to. Branches containing the same source files, that is. Refer BbA above - you should be doing it. Often senior devs would claim they have a special case, and want to do it on a branch. The pitfall is the proliferation of branches on the shared source-control server, the length of their ‘temporary’ life, and the difficulty of merging when there are lots of developers and lots of commits to one place or another.

This one aspect is debated back and forth, even by people that like the concept of a trunk. I’m going to put a line in the sand, and say that you should not make branches (on that shared repo) for features regardless of how long they are going to take, and whether they run over release dates. You should do BbA instead.

Not doing a CI pipeline on that single branch
Sure as a personal practice, you could prevent breakage, and many open source teams will argues they are good without CI. But for enterprise-land with tens of developers you need thorough CI.

Manual version number maintenance for dependencies
Dependencies of components for a buildable using can be expressed in a versioned way.

For example Log4J is currently at 1.2.17, and maintained by Apache. You’re not going to pull in their source to your source-control. You’ll depend on a binary, and bake the version number for your own build files (under source control).

For your own stuff, that’s perhaps been built in a different phase of CI, your should not bake in version numbers for specific builds. Borrowing from Maven’s idioms you should instead depend on a moving target under CI. Say ‘OurCommonThings-1.1-SNAPSHOT’, but ensure you build that in a ‘correct’ CI build phase. You’ve no intention of going live like that, but you can’t be Continuously integrating if your not compiling against, and testing with the latest version of everything.

A more hardcore implementation in CI pipeline would use something other than the a controversial ‘1.1-SNAPSHOT’ I suggested above. It (and Maven per se) is a controversial thing in enterprise development. For Subversion or Perforce installations, the repository revision number could be what you use instead - ‘1.1-12345’. Alternatively, a build number might be popular (all CI tools can provide a build number to the scripts they execute).

Not doing CI from ‘root’
In regular configurations, CI should build all your stuff from root/scratch, and not depend on anything built in a prior run. Some of the more sophisticated CI infrastructures (like ThoughtWorks Studios’ Go), have a more provable/fingerprinted way of tactically using pre-build pieces, but regular installations should build from scratch (as fast as possible).
This is another variation of the “latest version of everything” goal.

**Concurrent Development of Consecutive Releases**

Some enterprises work on a series of releases at the same time. They are intending to do dark-deployments using runtime toggles, but also perhaps have build-time switches to subset functionality in the resulting binary, depending on what they want to test in the CI phase. They may have more than one CI pipelines setup for the same trunk, that proves that the “amazon_one_click=true” and the “amazon_one_click=false” alternates build and pass tests. Either one of the two failing is still failure for a commit, and subject to rollback/revert.

You’d only set up pipelines with differing permutations of toggles for releases you’re expecting to put live. If ‘management’ cancel the features of one release (activated by a single toggle), or reorder releases, then you reconfigure the CI pipelines as soon as you can. The dollar and time impact of that ‘re-planning’ is clarified soonest by CI, and the resulting passing/failing view of the trunk for each. You’re never going to test unreasonable toggle permutations.

Incidentally the eXtreme Programming community correctly suggests that consecutive development of consecutive releases is preferable.

**Misnomers**

**Mainline is something else**

OK, so classically “Mainline” is a synonym of trunk, and for trunk-based-development people have been using “mainline” to describe that too. The trouble is that “mainline” also used by the ClearCase community from 1993 and refers to a wasteful and delaying branching model likes so:

This is also a “late” integration design, whereas TBD is a “earliest” integrations, which is one of the critical concepts, and greatest facilitator of cost-reduction during development. The other reality of this branching model, is branches that hang off the release branch, that are supposed to be temporary.

So, in summary, mainline means something else to a lot of software developers.
Feature Toggles
I’ve recently heard people refer to TBD as “Feature Toggles”. Martin Fowler named this long-held practice for the industry. It is often used in conjunction with TBD, but does not have to be. It can be used with any branching pattern, and is perhaps as old as developing software services and putting them live.
At a previous client we talked of build-time toggles too. These, for maven, were profiles like so:

```
# with amazon one click
mvn -p amazon_one_click install

# without amazon one click
mvn install
```

Thus for that client, toggles at runtime were different to toggles that were at build time (Maven profiles), but some could be both of course. As mentioned previously, CI pipelines are going to kicking on commit for reasonable toggle permutations.

Continuous Delivery (and Deployment)
This is the step up from simple TBD and CI usage. Jez has a well known book Continuous Delivery that is essential reading.
Thanks to…

Jez Humble for errata, and a nice quote “branching is not the problem, merging is the problem” (that’s a way of stating one problem TBD is trying to solve)

https://paulhammant.com/2013/04/05/what-is-trunk-based-development/