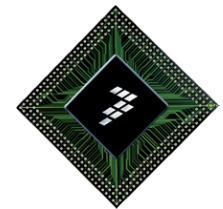


October 9, 2007

# Best Collaboration Practices Using Git

How to improve your odds with Open Source



---

Jon Loeliger

- ▶ Improve Your Odds
  - Betting Against The House
  - Partner With The House
  
- ▶ Playing The Game
  - Open Source Philosophy
  - The New Three R's
  
- ▶ Aces
  
- ▶ Aces and Eights
  
- ▶ Q & A

# Outline: Improving Your Odds

- ▶ The Players
- ▶ Betting Against The House
  - Kernel Source Code Statistics
  - Traditional Development Model
  - Traditional Development Meets Open Source
  - When Betting Schemes Fail
- ▶ Partner With The House
  - Odds In Your Favor

- ▶ The Community
  - Individual Developers
  - Corporations
  - Anyone who participates!
- ▶ Not (necessarily) the nerdy kid down the street
- ▶ Highly skilled professionals

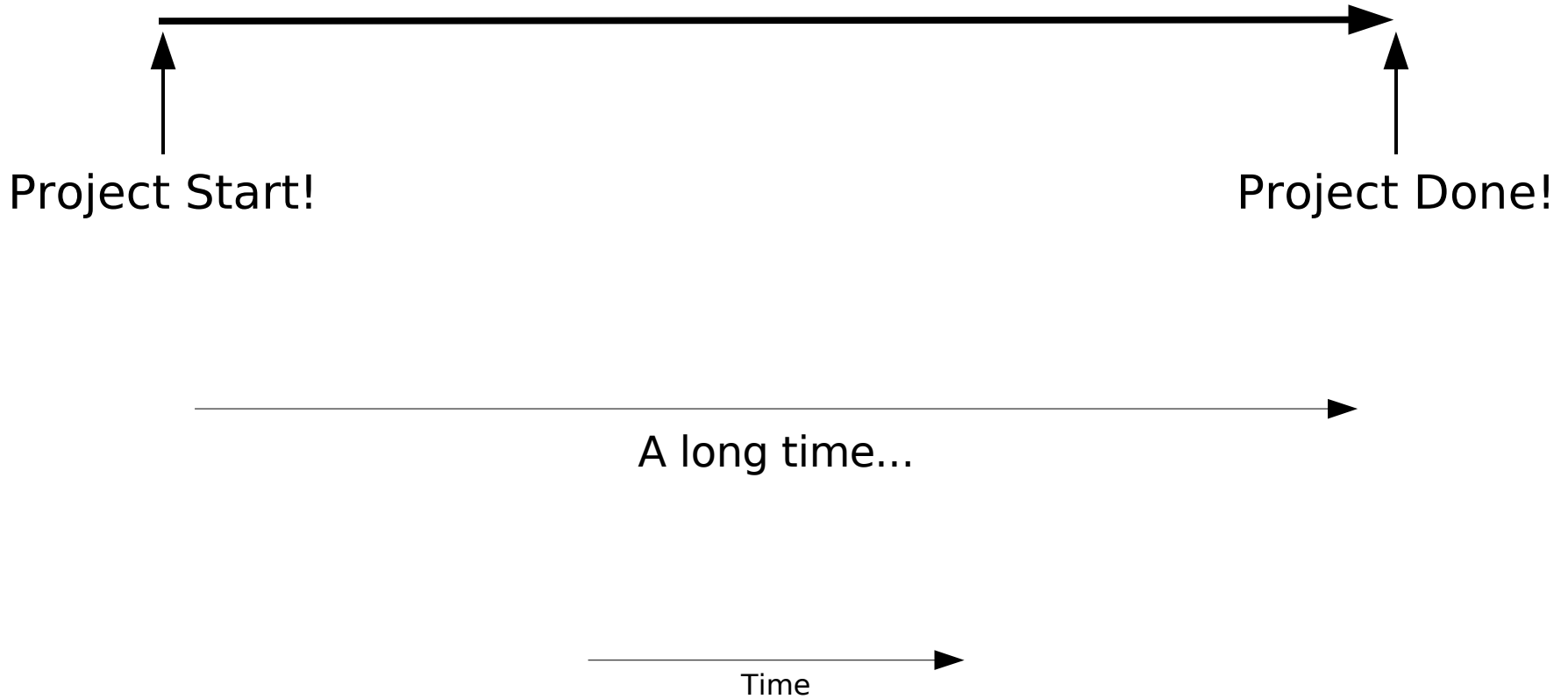
# Betting Against The House

- ▶ Fundamentally, The Community does not care if your code is accepted into mainline or not.
- ▶ “The code is not ready for public consumption yet!”
  - It is lousy
  - It hasn't been announced yet
- ▶ Ignore community feedback...

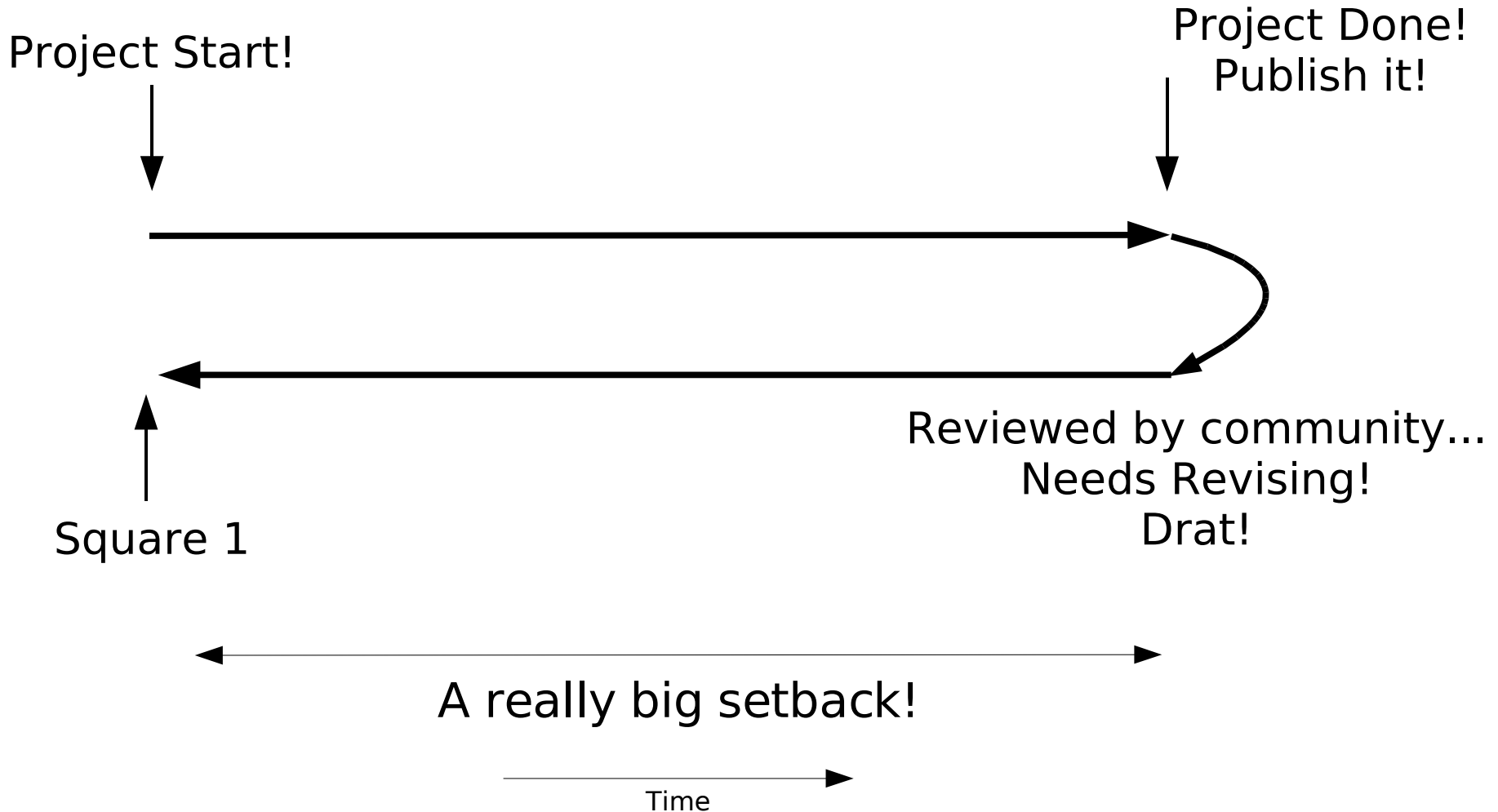
- ▶ Rate of change in code base is high
- ▶ Number of developers involved is high
- ▶ Source: <http://tree.celinuxforum.org/gitstat>

<u>Kernel Release</u>	<u>Number of Developers</u>	<u>Commits per hour</u>	<u>Commits per day</u>	<u>Release Duration Days</u>
2.6.19	836	3.8	91	71
2.6.20	692	3.1	75	67
2.6.21	847	2.9	72	81
2.6.22	883	3.7	89	74
2.6.23				

# Traditional Development Model



# Traditional Development Meets Open Source



# When Betting Schemes Fail

- ▶ Hazards: The odds are against you
  - Rate of *other* patch acceptance into kernel!
  - Large merges are problematic
  - Large patches won't be properly reviewed, or accepted
  - Prone to problems that get left unresolved
  - Patches will accumulate over time; never catch up
  - Holding patches creates more, harder, long-term work
  - Likely to do the same work again in the future

- ▶ The House has a large backing bank behind it
  - Lots of developers
  - Lots of momentum
  - Lots of staying power
- ▶ Corporations have a lot of clout
  - Likely, it is their hardware
  - Or supply Software Development resources
  - Have real money!
- ▶ Summary:
  - We all have a large Vested Interest here

- ▶ Let the System work for you!
  - Exploit The Community and individual developers!
  - Leverage the work of others
  - Avoid duplicating the work of others
  - Different perspectives; better ideas survive long term
  - Accept external expertise
  - Keep up to date!
- ▶ But help too!
  - Offer your expertise
  - Sanity check ideas prior to major development
  - Better code, fewer initial bugs

- ▶ Sense of (joint) ownership
- ▶ Distributed Responsibility
- ▶ Common understanding and direction from the start
- ▶ The sooner your changes get accepted ...
  - ...the sooner you can forget about them!
- ▶ Your changes become the accepted basis
- ▶ Enable help from others

# Outline: How The Game Is Played

- ▶ Open Source Philosophy
- ▶ The Kernel Release Cycle
- ▶ The Patch Cycle
  - Collaborate prior to publishing!
  - What goes into a patch?
  - Reviews
  - Typical review feedback
  - The New Threes R's

# Some Open Source Philosophy

- ▶ Anyone with interest can contribute
  - Add a feature, identify a bug, fix a bug, etc
- ▶ Publish early, Publish often
- ▶ If you want it, why not contribute it?
- ▶ Give-and-take, work with The Community
- ▶ Hierarchy of sub-system Maintainers

# Repository Conventions

- ▶ “What repository should I start with?”
- ▶ “Local” versus “global” repositories
  - Developer versus published in upstream
- ▶ Once “published” it is carved in stone
- ▶ It is just convention that a repository is authoritative

# Why Develop at Top-of-Tree?

- ▶ Target future releases
- ▶ Easier to rebase development incrementally
- ▶ Maintainers always request patches that apply to ToT
- ▶ Upstream maintainers apply patches at ToT
- ▶ Reviewers likely to reject patches for old releases
- ▶ Certain not to duplicate existing work

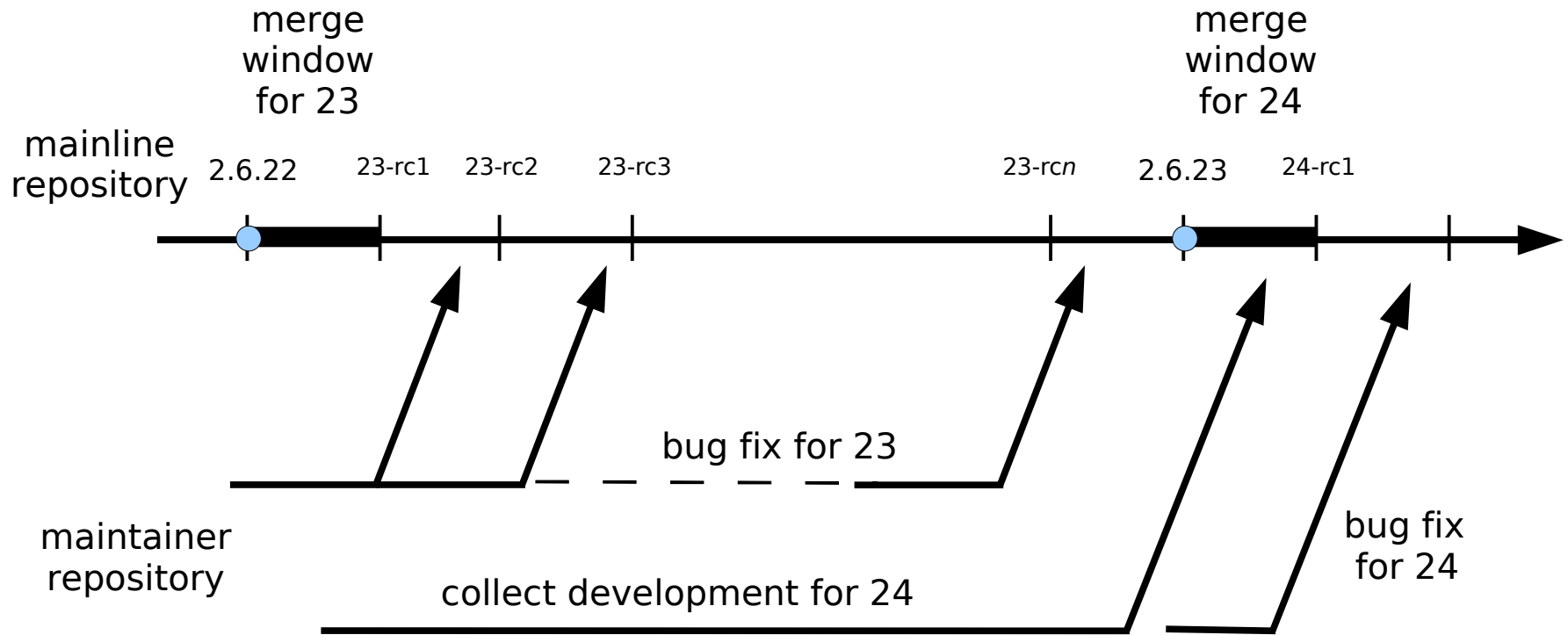
# Keep Current at Top-of-Tree

- ▶ Git makes it easy to keep up-to-date with ToT
  - `git fetch`, `git pull`
- ▶ Slight variations on “ToT” due to different repositories
  - Use the one appropriate for your development
  - Git supports using more than one if needed
- ▶ Development branches can easily be refactored to ToT

# The Kernel Release Cycle

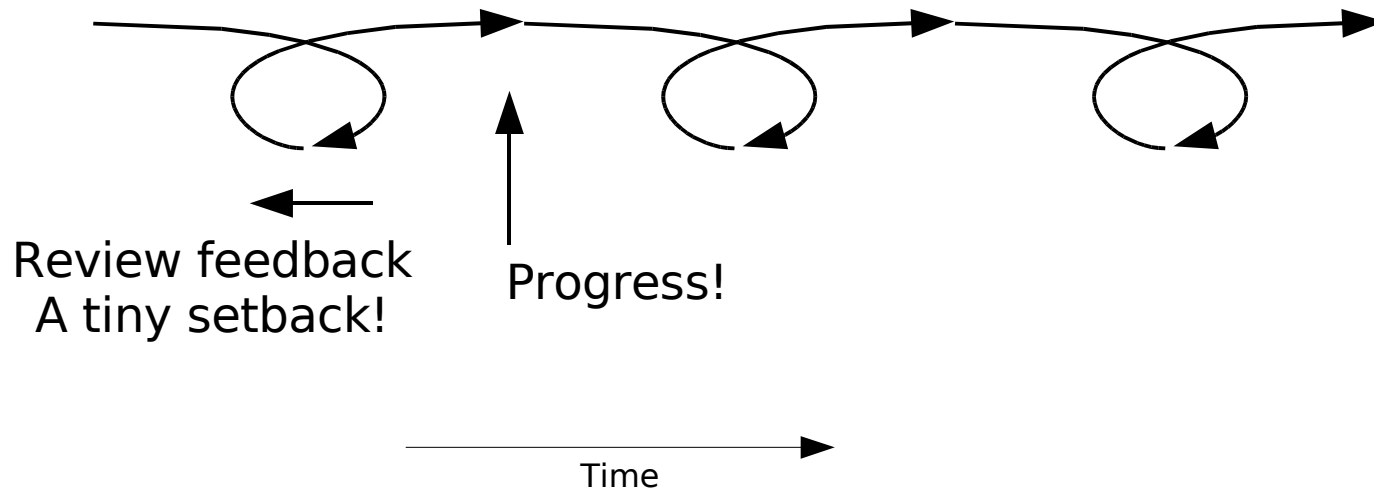
- ▶ Point release tagged and shipped
- ▶ Two week merge window opens
  - Sub-system maintainers push staged patches upstream
- ▶ Arbitrary number of Release Candidates
  - Convention: Whim of some trusted person
  - Bug fixes and stabilization
  - Sub-system Maintainers stage next release patches
- ▶ Sub-system staged patches accumulate at any time!

# Kernel Release Timeline



- 1) Collaborate in advance
  - 2) Implementation
  - 3) Organize patches for Maintainer's Top of Tree
  - 4) Send patches to list, Maintainer for review
  - 5) Receive review feedback from The Community
  - 6) Revise implementation according to feedback
  - 7) Rebase patches to current Top of Tree
  - 8) Re-post revised patches
  - 9) Rinse, lather, repeat until accepted
- ▶ Publishing patches is not the end of the development
    - It is the beginning!

# Patch Feedback Cycle

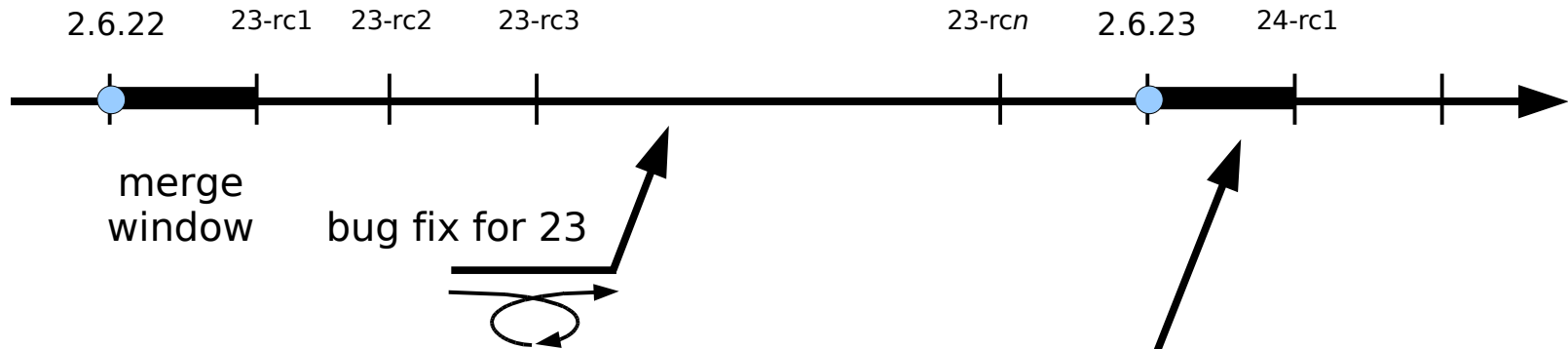


# Git Facilitates Short Patch Cycles

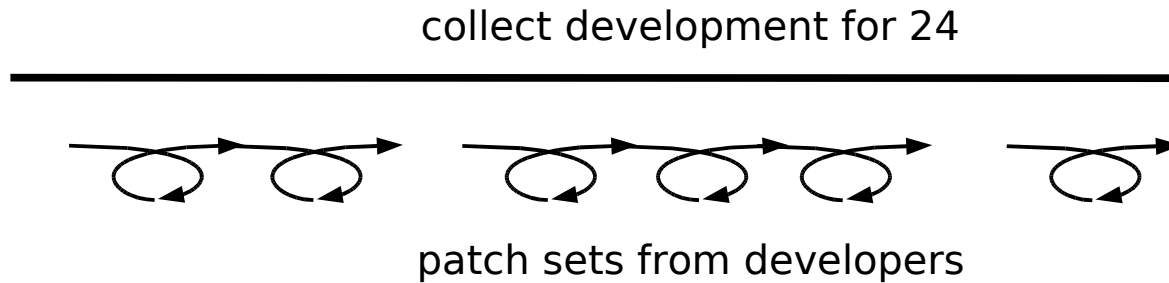
- ▶ Key to cycle time is picking small, concise well-defined and easily understood units of development.
- ▶ Git helps organize concise patch sets in topic branches
  - `git branch`
- ▶ Easy for branches to contain single development topics
  - Short cycles  $\Leftrightarrow$  “early and often” motto
- ▶ Hoarding/holding patches leads to more work
  - Long cycles lead to lost patches
  - Long cycles lead to harder merges later
  - Repeated work

# Combined Development Cycles

## Release Cycle



## Patch Cycles



# Coordinate Prior to Publishing

- ▶ Make an RFC
- ▶ Determine if it is already being worked on
- ▶ Set expectations for developers and Community
  
- ▶ Pick the right basis repository
- ▶ Determine correct upstream maintainer for patches
  
- ▶ Exploit prior expertise and experience
- ▶ Avoid duplicate development
- ▶ Avoid heading down the wrong path

# What is in a patch?

- ▶ Small, self-contained units of development
  - `git commit`
- ▶ Arrange patch sets into “Keep it working” order
  - `git bisect`
- ▶ Something easily reviewed!
- ▶ Follow code and style guidelines
- ▶ Good Change Log entry
- ▶ Follow Sign-off policy
- ▶ Use `checkpatch.pl`
- ▶ Isolate changes into patch sets
  - `git format-patch, git diff`
  - `git commit`

- ▶ Use git's built-in commands
  - `git format-patch`, `git send-email`
- ▶ Send only your changes
- ▶ Use a one line “commit summary” in Subject:
- ▶ Post to correct list and Maintainer
  
- ▶ Use plain ASCII inline text
- ▶ Avoid attachments or encoded text
- ▶ Don't cut-and-paste diffs
  
- ▶ Redmond-based SW is fragile

- ▶ Reviews are the path to mainline acceptance
- ▶ Concise patches are required for a good review
- ▶ Reviewers will:
  - Ask for some improvement
  - Invariably ask to have patches rebased to ToT
  - Invariably need patches to be respun after an review
  - Endlessly nit-pick some stupid typo

# Typical Review Feedback

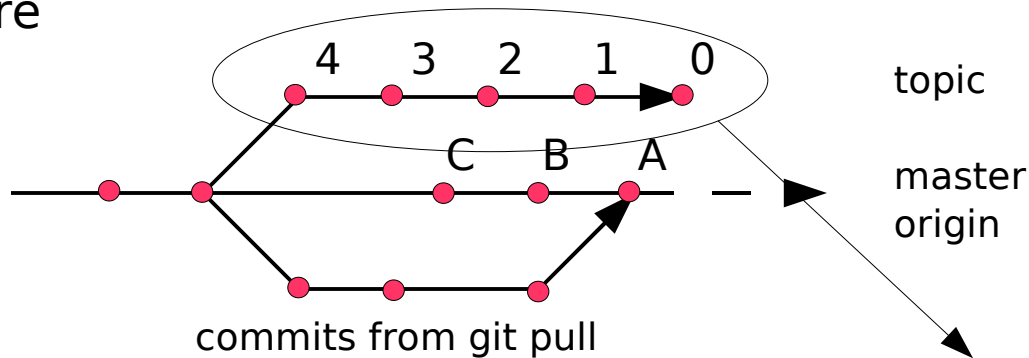
- ▶ Prevent latent bugs from being introduced
  - ▶ Encourage good long-term direction
  - ▶ Provide explanations, request explanations
  - ▶ Help avoid common pitfalls
  - ▶ Suggest better coding practices
  - ▶ Enforce standards
  - ▶ Style, typo corrections
- 
- ▶ Some review issues can be mitigated in advance
  - ▶ Feel free to push back and defend your patch!
  - ▶ Put your ego in check

- ▶ Revise your patches
  - `git cherry-pick`, `git commit --amend`
  - `git gui`, `git rebase --interactive`
- ▶ Rebase your patches
  - `git rebase`, `git fetch`, `git cherry-pick`
- ▶ Repost your patch set
  - `git format-patch`, `git send-email`

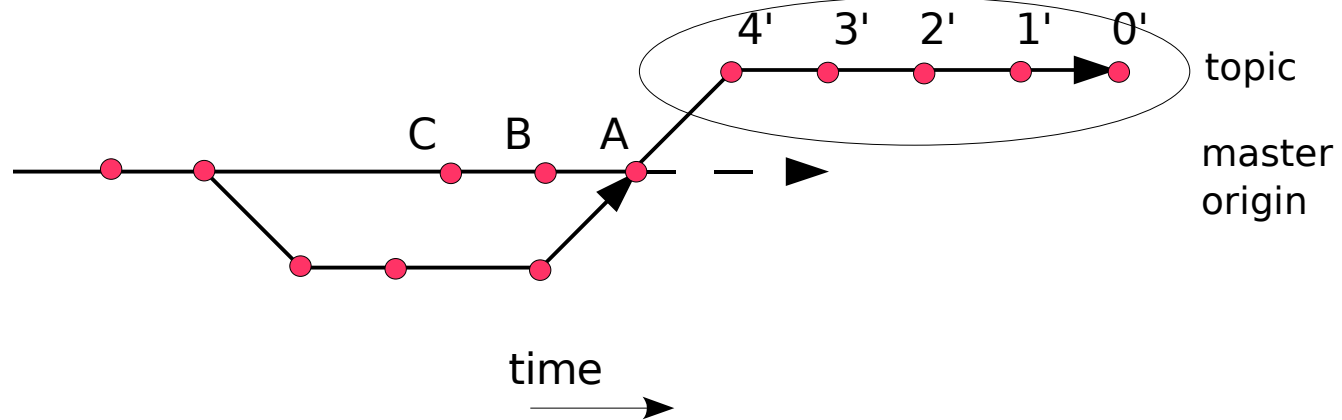
# Rebase: Before and After

- Rebase commits 0 – 4 of `topic` onto `master` branch at A as 0' – 4'

Before



After



- ▶ Participate with The Community
- ▶ Develop at the Top of Tree
- ▶ Post patches early and often
- ▶ Follow code standards, submission guidelines
- ▶ Revise, Rebase and Re-post patches after a review

# Outline: Aces and Eights

- ▶ Don't start with old releases for new development
- ▶ Don't post patches that apply to old releases
- ▶ Don't hold/hoard patches
- ▶ Don't do tarball based development
- ▶ Avoid Redmond based environments

# Outline: Questions And Answers?

▶ Questions?

# Outline: Backup Slides

- ▶ Backup slides

# Git Concepts: Repositories

- ▶ Git repository is self-contained, local and complete
  - Doesn't require external content
  - No centralized repository
  - Includes complete history of every file and change log
- ▶ Git repository is not distributed
  - No: part here, part there, part over there...
- ▶ No git repository is inherently authoritative
  - Only authoritative by convention and agreement

# Git Concepts: Distributed Development

- ▶ Development can be distributed
  - Multiple repositories can create different histories and changes
    - Even if they originated from a common ancestor
  - Multiple repositories with different development can be combined
- ▶ Development can be shared
  - Multiple developers can use and update a common repository
    - No matter if the repository is local or remote
- ▶ Development can be private
  - Revision control your Address List at home

# Git Concepts: Branches in git

- ▶ Cheap, fast and easy
- ▶ Topic/Development Branches
  - Stable, development, bug-fix, testing branches
  - Small development lines, per-feature, per-developer, etc
  - Collect, reorder or organize changes
  - Cherry-pick particular patches
- ▶ Tracking Branches
  - Follow upstream changes in local repository
  - Don't commit to tracking branches
  - Identified as RHS of `Pull: refspecs`
- ▶ Branch names refer to the current branch HEAD revision
- ▶ Branches don't have a “beginning” per se

```
$ git merge-base <original-branch> <branch-name>
```

# Cherry Picking Example

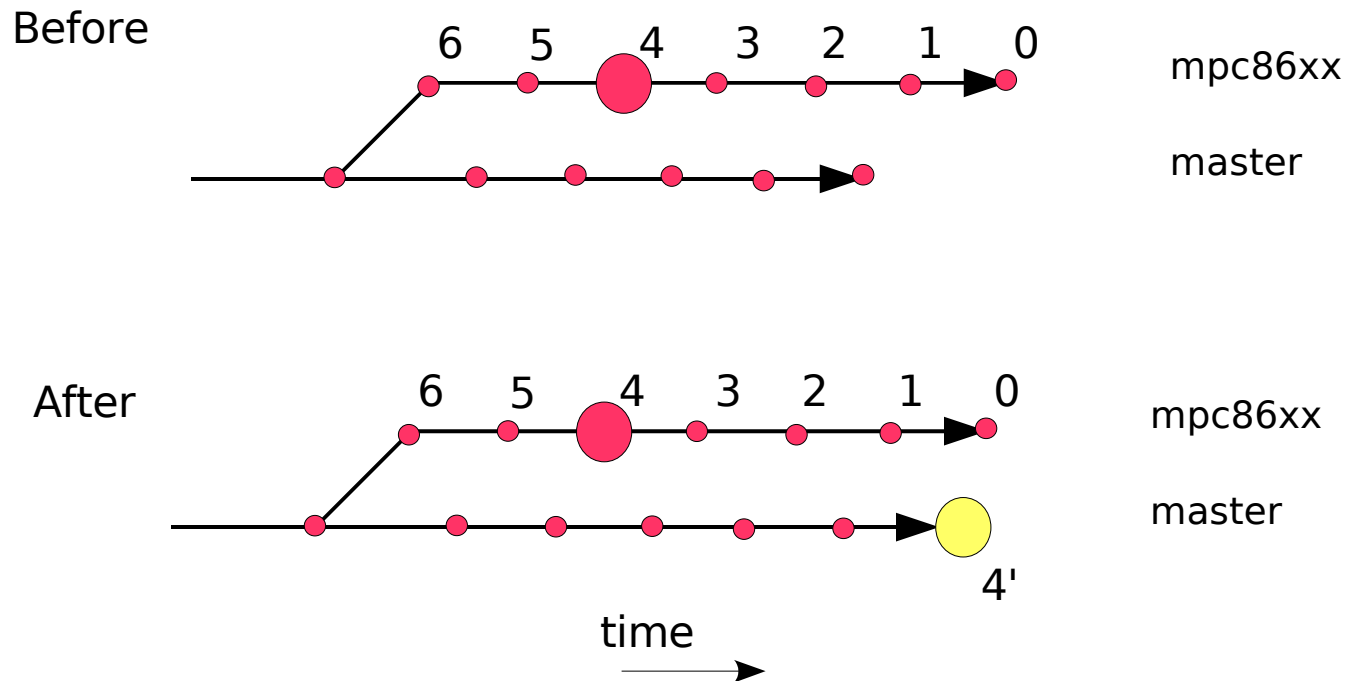
- ▶ You want one or more particular commits from some other branch applied to your current branch
  - Bug fix from some other branch
  - Transfer partial functionality from development branch
- ▶ Get that commit into your repository
  - Already present on a different branch
  - Perhaps using `git fetch` from another repository
- ▶ Cherry-pick it into appropriate branch

```
$ git checkout master
$ git cherry-pick 9ad494
```

# Cherry Picking: Before and After

## Branch Timeline Picture

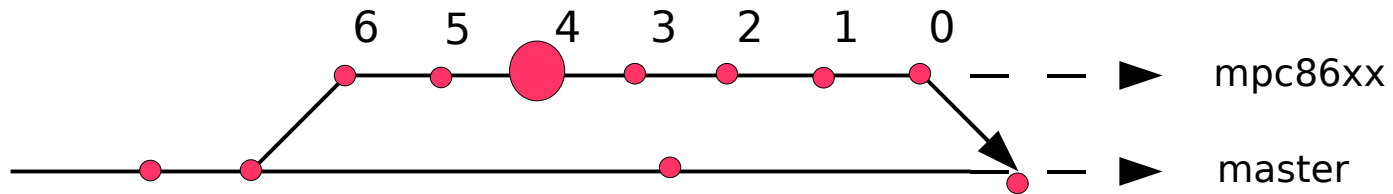
- 4' is a different commit with the same content as 4



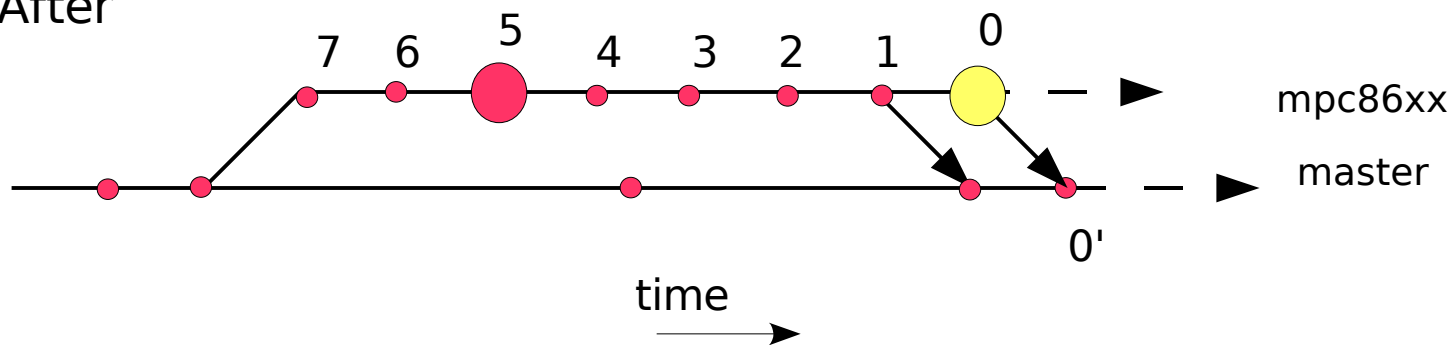
# Revert: Before and After

- ▶ Commit 4 is reverted, creating commit 0 in yellow
- ▶ Commit 0 is merged to master as commit 0'

Before



After



# Rebase Local Changes Example

## ▶ Situation:

- You have done development work on your topic branch
- You pull in upstream `origin` and merge it to `master`
- You don't want to merge `master` into your topic branch
- You don't want to merge your topic into the `master` branch
- No one else has cloned your topic branch!
  - Why? Don't rewrite history
- But you want to send your topic branch patches upstream!

## ▶ This is a job for `git rebase`

## ▶ Rebase Commands

```
$ git checkout topic
```

```
$ git rebase master
```

▶ Creates a series of patches from `topic` to be applied to `master`

▶ May have to resolve conflicts at each step due to merge operation

```
git rebase --continue
```

```
git rebase --skip
```

```
git rebase --abort
```

## ▸ Sources and Documentation:

- `git.kernel.org` -- Git sources, documentation, repositories
- `git.or.cz/gitwiki` -- The Git Wiki

## ▸ Front-ends and Viewers:

- `gitk`, `git-gui`
- Stacked Git: `www.procode.org/stgit`
- Guilt: `git.kernel.org`
- QT Gui viewer: `sourceforge.net/project/qgit`

## ▸ Mail List: `git@vger.kernel.org`

## ▸ IRC: `freenode.net #git`

