Applying Open Source Practices to Corporate Software Development

Brian Behlendorf
CTO, CollabNet
Board Member, Apache Software Foundation
Board Member, Mozilla Foundation
OSDL Enterprise Linux Summit 2005

The corporate software development disaster

- > 45% of software development projects in 2003 were cancelled; another 30% are completed late and/or with reduced features.
- The trend is not in a positive direction – failure rates have **risen** the last ten years.
- Source code management practices: reality often defies expectations (and belief).

Some of the more notorious examples:

- U.S. Government failures at the IRS and DoD
  - “There are very few success stories,” said Paul Brubaker, former deputy chief information officer (CIO) at the Pentagon
- AT&T Wireless “Self-Destructs” - CIO Magazine, Apr 15 2004
  - “The story of a botched CRM upgrade that cost the telco thousands of new customers and an estimated $100 million in lost revenue. Hard lessons learned.”

What's behind these failures?

- Slow feedback loops from inception to use.
- High underlying technology churn.
- Poorly documented prior systems and requirements.
- The **growing** difficulty in estimating work.
- Demotivated developers.

Developers as “resources”

- Myth: developers are often seen as commodity, like assembly line workers.
- Reality: the productivity difference between a good developer and a great one is modulo two orders of magnitude. [Brooks]
- Motivation just as important as talent to productivity.
- Corporations usually fail to properly match engineers to projects that inspire them.

Project-oriented mindset kills continuity.

- The lifecycle of a software project doesn't end on a ship date.
- Teams often throw away the development artifacts they created along the way, or place them in obscure places.
- Tight scheduling often means no time to explore what other developers are working on, or clean up one's own code for others.

As a result, for most, software re-use is a myth.

- Some have built asset repositories... with just tarballs of source code and searchable metadata, at best.
- Developers have scant incentive to properly prepare their components for re-use by others, or consider using someone else's work.
- “Forking” is either not allowed (you can't modify this work), or wildly uncontrolled (everyone has their own version).

Software components are not like bricks.

- The fundamental flaw in most of the past discussion about component re-use: components are never finished.
- All software has bugs.
- All software needs adaptation to new platforms over time.
- New requirements can't always be wrapped around or above existing code.
- APIs are conversations, and need to evolve over time.

Thus, the biggest hurdle to re-use is trust.

- So let's say I find an interesting component for my project.
- Who else is using this? What problems have they faced in using it?
- What defects exist? What doesn't the developer want to admit is a defect?
- If I find a defect, who can help me fix it, who do I send my patch to?
- Will there be a patchfix release? A 2.0 release? How do I participate?

What are some Open Source best practices?

- Transparency into the entire process.
- Gradients of access.
- Efficient mapping from developer interest to interesting projects.
- No “architects” who don't cut code.
- Dominant personalities only survive if they can still support a community.
- No whining – need something fixed, then produce some code.
- Forking allowed, but mitigated by teams working together.

One more myth to bust: development predictability.

- Many corporations still harbor the illusion that software development is like a factory pumping out Model T Fords.
- Despite the process experts who tell them the best approaches are more about feedback loops and agility.
- Open Source software gets a free pass on predictability, of course – which has allowed for an interesting experiment.

A different metaphor: the greenhouse.

- Look at the stable of internal projects as if they were plants managed by gardeners.
- Take risks by seeding more than you expect to harvest, perhaps even hedging your bets.
- Set general directions with queues of desired features and bugfixes.
- Weed, train, adjust techniques, harvest at the end.
- Your harvest won't be 100%, but that's OK.
- Make no promises until the harvest is in.

This can break the incentive logjam.

- The greenhouse metaphor encourages the view of software lifecycles as ongoing, long past release.
- In a single environment like a greenhouse, interdependence is implicit, and allows for lightweight and ad-hoc coordination between projects.
- Developers are the gardeners, and being responsible for some plants and admirers of others is the norm.

The importance of slack time.

- A less time-controlled approach allows for more slack time by good developers.
- Slack time in a schedule helps by allowing the developer to:
  - look around speculatively at what other teams have built, and are working on.
  - help others to re-use or refactor code.

How do you roll this out?

- Find a pilot group willing to be the goldfish in the fishtank.
- Allow them to define the initial size of the fishtank to suit their comfort.
- Project should be digital from day one: start with specs, customer requests, any initial artifacts in a single, consistantly viewed space.
- Stay visible in activity and intent throughout the project.
- A “need to know” mentality is cancerous to the project – err on the side of revealing too much rather than not enough.

More roll-out tips...

- Build cross-project teams around certain technologies or standards.
- Provide financial incentives for re-use, and helping others re-use your work.
- Invite others not directly involved, but with stakes in the project, to observe and participate.
- As virtual as all this is, face-time to build trust is essential at the start.

Moving discussions and decision-making online.

- So much knowledge is created, and so many implicit decisions are made, in the ad-hoc conversations between developers, project managers, business owners, and customers.
- Capturing that is essential to re-use, as often code does not speak for itself and documentation and specs are incomplete or conflicting.
- Capturing the debate about a topic allows that discussion to be avoided the next time; or restarted quickly if new data emerges.

Moving discussions online is difficult, but essential.

- Allow – perhaps even require - developers to work from home one or two days every two weeks.
- Work intentionally with one or more people in a remote location.
- It causes everyone to think about their words in a way that anticipates future review by people you don't know – a good discipline.
- Conference calls and in-person conversations still have a role; but reflect them digitally in some way, and allow for participating in decision-making by online parties.

Pitfalls

- Not everyone is proud of their past work – establish an atmosphere of amnesty for the past.
- Personality conflicts are inevitable – either resolve them through management and coaching, or move someone out.
- This is one of the bigger problems the OS community has: time and effort wasted by fruitless argument.

How do you measure success?

- Projects should end up seeing a more graceful and continuous life beyond their release.
- Bringing new developers aboard, even those in distant locales, should be easier.
- Fewer conference calls, less of an oral culture.
- Do developers refer to prior discussions when making decisions?
- Ask the developers themselves... anecdotal evidence can be the most compelling.
- Expect to see lightweight involvement by developers in other projects; and by other stakeholders in theirs.