Software development tools

- Free software projects rely on technologies that support the **selective capture and integration of information**. The **more skilled** you are at using these technologies, and at persuading others to use them, the **more successful** your project will be
- Good **information management** is what prevents open source projects from collapsing under the weight of Brooks' Law, which states that **adding manpower to a late software project makes it later**
- Fred Brooks observed that the **complexity of a project increases as the square of the number of participants**. When only a few people are involved, everyone can easily talk to everyone else, but when hundreds of people are involved, it is no longer possible for each person to remain **constantly aware** of what everyone else is doing

Software development tools

- Since almost all communication in open source projects **happens in writing**, elaborate systems have evolved for **routing and labeling data** appropriately; for **minimizing repetitions** so as to avoid spurious divergences; for **storing and retrieving data**; for **correcting bad or obsolete information**; and for **associating disparate bits of information with each other** as new connections are observed
- Active participants in open source projects internalize many of these techniques, and will often perform complex manual tasks to ensure that information is routed correctly. But the whole endeavor ultimately **depends on sophisticated software support**
- If the humans take care to label and route information accurately **on its first entry into the system**, then the software should be configured to make as much **use of that metadata** as possible

Software development tools

- **Technical skills** are essential because information management software always requires configuration, plus a certain amount of ongoing maintenance and tweaking as new needs arise
- **People skills** are necessary because the human community also requires maintenance: it's not always immediately obvious how to use these tools to full advantage. Everyone involved with the project will need to be encouraged, at the right times and in the right ways, to do their part to keep the project's information well organized
- Information management has **no cut-and-dried solution**. There are too many variables. You may finally get everything configured just the way you want it, and have most of the community participating, but then project growth will make some of those practices unscalable

Software development tools

- **Tradeoffs** between the convenience of **those producing** information and the convenience of **those consuming** it, or between the time required to configure information management software and the benefit it brings to the project
- Beware of the temptation to **over-automate**, that is, to automate things that really require **human attention**. Technical infrastructure is important, but what makes a free software project work is care by the humans involved
- Standard set of tools for managing information
  - **Web site**: primarily a centralized, **one-way conduit of information** from the project out to the public. The web site may also serve as an **administrative interface** for other project tools
  - **Mailing lists**: usually the most active communications forum in the project, and the "medium of record"

Tools for managing information

Good mailing list management is not simple at all. It's not just about subscribing and unsubscribing users when they request. It's also about **moderating to prevent spam**, offering the mailing list in **digest** versus **message-by-message form**, providing **standard list** and **project information by means of auto-responders**

- **Both email- and web-based subscription**: when a user subscribes to a list, she should promptly get an **automated welcome message in reply**, telling her what she has subscribed to, **how to interact** further with the mailing list software, and (most importantly) **how to unsubscribe**. This automatic reply can be customized to contain projectspecific information, of course, such as the project's web site, FAQ location, etc

Tools for managing information

- **Subscription in either digest mode or message-by-message mode**: in digest mode, the subscriber receives **one email per day**, containing all the list activity for that day. For people who are following a list loosely, without participating, digest mode is often preferable, because it allows them to **scan all the subjects at once** and avoid the distraction of emails coming in at random times
- **Moderation features**: to "moderate" is to check posts to make sure they are a) **not spam**, and b) **on topic**, before they go out to the entire list. Moderation necessarily involves humans, but software can do a lot to make it easier
- **Administrative interface**: among other things, this enables an administrator to go in and remove obsolete addresses easily

Tools for managing information

  - **Header manipulation**: many people have **sophisticated filtering and replying rules** set up in their mail readers. Mailing list software can add and manipulate certain standard headers for these people to take advantage of
  - **Archiving**: all posts to the managed lists **are stored and made available on the web**; alternatively, some mailing list software offers special interfaces for plugging in an external archiving tool
- **Version control**: enables developers to **manage code changes** conveniently, including **reverting** and "**change porting**". Enables everyone to watch what's happening to the code
  - **Version everything**: keep not only your project's source code under version control, but also its **web pages**, **documentation**, **FAQ**, **design notes**, and anything else that people might want to edit. Keep them right next to the source code, in the same repository tree

Tools for managing information

Any piece of information **worth writing** down is **worth versioning**, that is, any piece of information that **could change**. Things that don't change should be **archived**, not versioned. For example, an email, once posted, does not change; therefore, versioning it wouldn't make sense. The reason versioning everything **together in one place** is important is so people only have to learn one mechanism for submitting changes. Often a contributor will start out making edits to the web pages or documentation, and move to small code contributions later. When the project uses the **same system for all kinds of submissions**, people only have to learn the ropes once. Versioning everything together also means that **new features can be committed together with their documentation updates**, that branching the code will branch the documentation too

Tools for managing information

Don't keep generated files under version control. They are not truly editable data, since they are produced programmatically from other files

- **Browsability**: the project's repository should be **browsable on the Web**. This means not only the ability to see the latest revisions of the project's files, but to go back in time and look at earlier revisions, view the differences between revisions, read log messages for selected changes. Browsability is important because it is a **lightweight portal to project data**. If the repository cannot be viewed through a web browser, then someone wanting to inspect a particular file (say, to see if a certain bugfix had made it into the code) would first have to install version control client software locally. Some version control systems come with **built-in repository-browsing mechanisms**, while others rely on third-party tools to do it

Tools for managing information

- **Commit emails**: every commit to the repository should **generate an email** showing **who** made the change, **when** they made it, **what** files and directories changed, and **how** they changed. The email should go to a **special mailing list** devoted to commit emails, separate from the mailing lists to which humans post. Developers and other interested parties should be encouraged to subscribe to the commits list, as it is **the most effective way to keep up with what's happening in the project at the code level**
- **Use branches to avoid bottlenecks**: branches are valuable because they turn a scarce resource - working room in the project's code - into an abundant one. Normally, all developers work together in **the same sandbox**. People need the freedom to try new things without feeling like they're interfering with others' work

Tools for managing information

There are times when **code needs to be isolated** from the usual development churn, in order to get a **bug fixed** or a **release stabilized**. Branches should not become a mechanism for **dividing the development community**. With rare exceptions, the eventual goal of most branches should be to merge their changes back into the main line and disappear.

- **Singularity of information**: merging has an important corollary, **never commit the same change twice**. That is, **a given change should enter the version control system exactly once**. The revision (or set of revisions) in which the change entered is its **unique identifier** from then on

Tools for managing information

If it needs to be applied to branches other than the one on which it entered, then it should be **merged from its original entry point to those other destinations**, as opposed to committing a textually identical change, which would have the same effect in the code, but would make accurate bookkeeping and release management impossible

- **Authorization**: most version control systems offer a feature whereby certain people can be **allowed or disallowed** from committing in **specific sub-areas** of the repository. Some projects simply use an **honor system**: when a person is granted commit access, even for a sub-area of the repository, what they actually receive is a password that allows them to **commit anywhere in the project**. They're just asked to keep their commits in their area (using the honor system encourages an atmosphere of **trust and mutual respect**)

Tools for managing information

- **Bug tracking**: enables developers to **keep track** of what they're working on, coordinate with each other, and plan releases. Enables everyone to query the status of bugs and record information (e.g., reproduction recipes) about particular bugs. Can be used for tracking **not only bugs**, but also **tasks**, **releases**, new **features**, etc. Bug trackers are also called issue trackers, defect trackers, artifact trackers, request trackers, trouble ticket systems. Classic issue life cycle:
  - **Someone files the issue** - they provide a summary, an initial description and whatever other information the tracker asks for. The person who files the issue may be **totally unknown to the project**, bug reports and feature requests are as likely to come **from the user community as from the developers**

Tools for managing information

Once filed, the issue is in what's called an **open state**. Because no action has been taken yet, some trackers also label it as **unverified and/or unstarted**. It is not assigned to anyone. At this point, it is in a **holding area**: the issue has been recorded, but not yet integrated into the project's consciousness

- **Others read the issue** - add comments to it, and perhaps ask the original filer for clarification on some points
- **The bug gets reproduced** - this may be the most important moment in its life cycle. Although the bug is not actually fixed yet, the fact that someone besides the original filer was able to make it happen proves that **it is genuine**, and, no less importantly, confirms to the original filer that they've contributed to the project by **reporting a real bug**.

Tools for managing information

- **The bug gets diagnosed - its cause is identified**, and if possible, the **effort required to fix it is estimated**. Make sure these things get recorded in the issue; if the person who diagnosed the bug suddenly has to step away from the project for a while (as can often happen with volunteer developers), someone else should be able to pick up where she left off
- **The issue gets scheduled for resolution** - scheduling doesn't necessarily mean naming a date by which it will be fixed. Sometimes it just means deciding **which future release** (not necessarily the next one) the bug should be fixed by, or deciding that it need not block any particular release
- **The bug gets fixed** (or the task completed, or the patch applied, or whatever) - the change or set of changes that fixed it should be recorded in a comment in the issue, after which the issue is **closed and/or marked as resolved**

Tools for managing information

The need for timely reactions implies two things

- **The tracker must be connected to a mailing list**, such that every change to an issue, including its initial filing, causes a mail to go out describing what happened. This mailing list is usually **different from the regular development list**, since not all developers may want to receive automated bug mails
- **The form for filing issues should capture the reporter's email address**, so she can be contacted for more information. However, it should not require the reporter's email address, as some people prefer to report issues anonymously

Tools for managing information

- **Real-time chat**: a place for quick, **lightweight discussions and question/answer exchanges**. Not always archived completely. Many projects offer real-time chat rooms using **Internet Relay Chat** (IRC), forums where users and developers can ask each other questions and get instant responses. The first thing to do is choose a **channel name**. Try to choose something as close to your **project's name**, and as easy to remember, as possible. Advertise the channel's availabity from your project's web site, so a visitor with a quick question will see it right away. Some projects have **multiple channels**, one per subtopic. For example, one channel for installation problems, another for usage questions, another for development chat, etc. When your **project is young**, there should only be one channel, with everyone talking together. Later, as the user-to-developer ratio increases, separate channels may become necessary

Tools for managing information

- **Wikis**: a wiki is a **web site** that allows any visitor to **edit or extend its content**; the term "wiki" (from a Hawaiian word meaning "quick" or "super-fast") is also used to refer to the software that enables such editing. Think of a wiki as falling somewhere between **IRC and web pages**: wikis don't happen in realtime, so people get a chance to ponder and polish their contributions, but they are also very easy to add to, involving less interface overhead than editing a regular web page. If you decide to run a wiki, put a lot of effort into having a **clear page organization** and pleasing visual layout, so that visitors (i.e., potential editors) will instinctively know how to fit in their contributions. Post those standards on the wiki itself, so people have somewhere to go for guidance. Each **individual page or paragraph may be good when considered by itself**, but **it will not be good if embedded in a disorganized or confusing whole**

Tools for managing information

Often wikis suffer from:

- **Lack of navigational principles** - a **well-organized web site** makes visitors feel like they know where they are at any time. if the pages are well-designed, people can intuitively tell the difference between a "table of contents" region and a "content" region. Contributors to a wiki will respect such differences too, but **only if the differences are present to begin with**
- **Duplication of information** - wikis frequently end up with **different pages saying similar things**, because the individual contributors did not notice the duplications. This can be partly a consequence of the lack of navigational principles, in that people may not find the duplicate content if it is not where they expect it to be

Tools for managing information

- **Inconsistent target audience** - to some degree this problem is inevitable when there are so many authors, but it can be lessened if there are written guidelines about how to create new content

The common solution to all these problems is the same: **have editorial standards**, and demonstrate them not only by posting them, but by **editing pages to adhere to them**. In general, wikis will amplify any **failings in their original material**, since contributors imitate whatever patterns they see in front of them. Don't just set up the wiki and hope everything falls into place. You must also prime it with well-written content, so people have a template to follow

Tools for managing development

- Standard set of tools for managing development
  - **Build automation**: is the act of **scripting or automating** a wide variety of tasks that a software developer will do in their **day-to-day activities** including things like:
    - compiling computer source code into binary code
    - packaging binary code
    - running tests
    - deployment to production systems
    - creating documentation and or release notes

Tools for managing development

Advantages of build automation:

- improve product quality
- accelerate the compile and link processing
- eliminate redundant tasks
- minimize "bad builds"
- eliminate dependencies on key personnel
- have history of builds and releases in order to investigate issues
- save time and money - because of the reasons listed above

Tools:

- **Make** - the Unix “make” command is a standard tool to **automate the compilation** of source code trees. It is one example of using automation to reduce barriers to casual contributors.

Tools for managing development

Developers who intend to reuse a component can safely assume that it has a makefile that includes **conventional make targets** like “make clean” and “make install”. Makefiles are essentially programs themselves, so they can be made arbitrarily complex to support various diverse use cases. For example, in addition to compiling code, makefiles can be used to run regression tests. Running regression tests frequently is one key to the practice of frequent releases

- **Ant** - Ant is a **Java replacement** for make that uses **XML build files** instead of makefiles. Each build file describes the **steps** to be carried out to build each **target**. Each step invokes a predefined **task**. Ant tasks each perform a larger amount of work than would a single command in a makefile. Many popular IDEs now include support for Ant

Tools for managing development

Ant is accessible to developers on **all platforms**, regardless of whether they prefer the **command-line** or an **IDE**. Since Ant build files are written in XML, developers are already familiar with the syntax, and tools to edit or otherwise process those files can reuse standard XML libraries

- **CruiseControl** - **Nightly build** tools automatically **compile** a project's source code and **produce a report** of any errors. In addition to finding compilation errors, these tools can build any make or Ant target to accomplish other tasks, such as **regression tests**, **documentation generation**, or **static source code analysis**. These tools can quickly catch errors that might not have been noticed by individual developers working on their own changes

Tools for managing development

Some nightly build tools **automatically identify and notify** the developer or developers who are responsible for breaking the build, so that corrections can be made quickly. Nightly build tools have been used within **large organized communities** such as Mozilla.org and Jakarta.apache.org. Volunteers may be more attracted to projects with **clear indications of progress** than they would otherwise. And the limited efforts of volunteers need not be spent on **manually regenerating API documentation or running tests**. Organized open source development communities use nightly build tools to help **manage dependencies between interdependent projects**.

Tools for managing development

– **Design and Code Generation**
  - **ArgoUML** - ArgoUML is a pure-Java **UML design tool**. ArgoUML closely follows the UML standard, and associated standards for **model interchange and diagram representation**. In addition to being **cross-platform** and **standards based**, it emphasizes ease of use and actively helps train casual users in UML usage. UML modeling tools have experienced **only limited adoption** among open source projects, possibly because of the emphasis on **source code as the central development artifact**. Tools such as ArgoUML that are themselves open source and cross-platform provide **universal access to design models**, because any potential volunteer is able to view and edit the model.

Tools for managing development

If design tools were more widely used in open source projects, UML models would provide substantial support for understanding components **prior to reuse**, for **peer reviews at the design level**, and for the **sharing of design patterns and guidelines** within development communities

- **Torque and Hibernate** - Torque is a Java tool that **generates SQL and Java code** to **build and access a database** defined by an **XML specification of a data model**. It is **cross-platform**, customizable, and **standards-based**. Torque's code generation is customizable because it is **template-based**. Also, a library of templates has been developed to address **incompatibilities between SQL databases**. Hibernate emphasizes ease of use and rapid development cycles by **using reflection** rather than code generation. Projects that have adopted them are able to produce products that are themselves **portable to various databases**

Tools for managing development

Code generation tools can increase the **effectiveness of volunteer developers** and **enable more frequent releases** by reducing the unlikable tasks of **writing repetitive code by hand**, and **debugging code that is not part of the project's value-add**. Code generation is itself a **form of reuse** in which knowledge about a particular aspect of implementation is discussed by community members and then **codified in the rules and templates** of the generator. Individual developers may then customize these rules and templates to fit any unusual needs. Peer review of schema specifications can be easier than reviewing database access code

- **Doxygen, XDoclet and JUnitDoclet** - These code generation tools build on the **code commenting conventions** used by Javadoc to generate API documentation. Doxygen works with C, C++, IDL, PHP, and C#, in addition to Java

Tools for managing development

XDoclet and JUnitDoclet can be used to **generate additional code** rather than documentation. For example, a developer could easily **generate stubs for unit tests** of every public method of a class. Another use is the **generation of configuration files** for Web services, application servers, or persistence libraries. The advantage of using comments in code rather than an independent specification file is that the **existing structure of the code is leveraged to provide a context for code generation parameters**. Doclet-style generators are a **form of reuse** that reduces the need to work on unlikable tasks, and **output templates** can be changed to fit the needs of diverse users. The doclet approach differs from other code generation approaches in that **no new specification files are needed**. Instead, the normal **source code contains additional attributes** used in code generation

Tools for managing development

This is a **good match for the open source tendency to emphasize the source code over other artifacts**

- **Quality Assurance Tools**
  - **JUnit, PHPUnit, PyUnit, and NUnit** - JUnit supports Java **unit testing**. It is a **simple framework** that uses **naming conventions** to identify test classes and test methods. A **test executive** executes all tests and produces a report. The JUnit concepts and framework have been ported to nearly every programming language, including PHPUnit for PHP, PyUnit for Python, and NUnit for C#. It has two key features that address the practices of the open source methodology: **test automation**, which helps to reduce the unlikable task of manual testing that might not be done by volunteers; and **unit test reports**, which provide universally accessible, **objective indications of project status**

Tools for managing development

Frequent testing and **constant assessment** of product quality support the practice of **frequent releases**. Visible test cases and test results can also help emphasize **quality as a goal for all projects in the community**

- **Lint, Checkstyle, JDepend** - The classic Unix command “lint” analyzes C source code for **common errors** such as **unreachable statements**, **uninitialized variables**, or **incorrect calls to library functions**. Adoption and use of these tools is limited, but several projects have started using Checkstyle and JDepend as part of Maven. Analysis tools can also be viewed as a **form of reuse** where **community knowledge is encoded in rules**. Static analysis can **prompt peer review** and help address weaknesses in the knowledge of individual developers

Tools for managing development

– **Collaborative Development Environments**

- **SourceForge** - CDEs allow users to **easily create new project workspaces**. These workspaces provide access to a **standard toolset** consisting of a web server for project content, mailing lists with archives, an issue tracker, and a revision control system. **Access control mechanisms** determine the information that each user can see and the operations that he or she can perform. CDEs also define **development communities** where the **same tools and practices** are used on every project, and where users can browse and **search projects to find reusable components**. CDEs have been widely adopted by open source projects. In particular, a good fraction of all open source projects are now hosted on http://sourceforge.net. CDEs provide the infrastructure needed for universal access to project information: they are **Web-based**

Impact of adopting tools

- The impact of adopting tools
  - Because the **tools are free** and support casual use, more members of the development team will be able to **access and contribute to artifacts** in **all phases of development**. Stronger involvement can lead to **better technical understanding**, which can **increase productivity**, **improve quality**, and smooth hand-offs at key points in the development process
  - Because the “**source**” to all artifacts **is available and up-to-date**, there is **less wasted effort** due to **decisions based on outdated information**. Working with up-to-date information can **reduce rework on downstream artifacts**

Impact of adopting tools

- Because many of the tools **support incremental releases**, teams using them should be better able to **produce releases early and more often**. **Early releases help manage project risk and set expectations**. **Frequent internal releases** can have the additional benefit of allowing **rapid reaction to changing market demands**
- Because many of the tools aim to **reduce unlikable work**, more development effort should be freed for forward progress. **Productivity increases**, **faster time-to-market**, and **increased developer satisfaction** are some potential benefits

Impact of adopting tools

- Because **peer review** is addressed by many of the tools, projects may be able to **catch more defects in review** or conduct **more frequent small reviews in reaction to changes**. Peer reviews are generally accepted as an **effective complement to testing** that can **increase product quality**, **reduce rework**, and aid the professional development of team members
- Because **project Web sites**, accessible **issue trackers**, and CDEs provide **access to the status and technical details of reusable components**, other projects may more readily evaluate and select these components for reuse. Also, HOWTOs, FAQs, mailing lists, and issue trackers help to **costeffectively support reused components**. Expected benefits of **increased reuse** include **faster time-to-market**, **lower maintenance costs**, and **improved quality**