Was ist Inner Source?

Inner Source takes the lessons learned from developing open source software and applies them to the way companies develop software internally. As developers have become accustomed to working on world class open source software, there is a strong desire to bring those practices back inside the firewall and apply them to software that companies may be reluctant to release. For companies building mostly closed source software, Inner Source can be a great tool to help break down silos, encourage internal collaboration, accelerate new engineer on-boarding, and identify opportunities to contribute software back to the open source world.

Inner Source bedeutet

Grundegende Ansätze von Inner Source

Unfortunately, implementation and adoption aren’t prescriptive; one shoe doesn’t fit all. However, there are a few key practices that can be adopted to facilitate a more collaborative development environment.

**Many contributors, few committers**

Typically, in the closed source world, teams have a product owner or technical lead. The remaining developers on the team are regarded as contributors. To provide an optimal collaborative culture, the team must evolve. While the product owner or technical lead doesn’t change, short of being exposed to a (potentially) much larger audience, the real change applies to the rest of the team. They become committers rather than contributors. The difference is subtle and yield greater results. Consider the differences:

A technical lead or product owner is responsible for the overall vision, architecture and technical roadmap of the project. They are responsible for providing final sign-off on feature requests and design documents as well as the overall integrity of the project.

A committer is a key stakeholder in the vision, technical roadmap, and architecture of the project. The lead and all committers must all be on the same page in terms of project direction and vision and be able to communicate that direction and vision to other stakeholders. Committers and leads have permissions required to merge their own code as well as contributions from external contributors.

A contributor contributes to the project, whether it’s code, documentation or tests, but does not have access to merge code directly to the project’s canonical repo or main branch.

**Automation**

The most successful collaborative projects are those that maximize automation. Getting a project up and running consistently between veterans and newcomers alike should be no more difficult than “make build”, “make install”, “make run”, and so on. Nothing further should be required. Projects should also have full automated build and test pipelines in place, giving new contributors confidence in their submissions before code is merged.

**Best practices**

By having a well-defined, company-wide set of best practices, you promote consistency. Consistency lowers the barrier of entry for contributors to hop from project to project. As much as possible, best practices should be based on open source best practices, which helps ensure that:

If they have contributed to open source projects and are familiar with the best practices, new hires have a lower barrier of entry.

The barrier of entry for dependency inspection (debugging, contributions, and so on) is lowered.

Not everyone will adopt open source community best practices for their team. Use what works for you. Best practices to consider for adoption are specific to:

* Project setup (directory structure, common files, and so on)
* Test and static analysis utilities
* Commonly used libraries
* Shared coding standards

By ensuring consistency across projects and teams in these areas, you help ensure minimal overhead for prospective contributors. A developer working on one project in Python should be able to jump to another Python project owned by another team with little to no overhead.

**Marketing and communication**

Code is written. Code is deployed. Code rots. If nobody else knows about it, nobody will actively maintain it.

“Build it and they will come” does not apply to code. It’s more “Build it, market it and attempt to build an active community around it” is a little closer to what’s actually required of a successful, collaborative project. Case in point: Docker. Was it the first containerization technology? Nope. Was it the best marketed and most approachable? Absolutely!

A consistent communication method is a fundamental pillar to the success of inner sourcing. Without a consistent communication method that transcends the team barriers, the effectiveness of inner sourcing efforts is inherently limited by a lack of transparency and open communication. When evaluating collaboration tools, it’s important to define what communication problems you’re trying to solve:

* Are you primarily going to be discussing system design and architecture?
* Are you primarily needing channels to facilitate fighting ops fires?

It’s important to define the actual problems you’re trying to solve as some tools and methods may actually degrade your ability to communicate effectively as a group. For further insight into this, see http://communicationtheory.org/creativity-in-groups/.

VCS != DVCS

VCS (i.e. Perforce, a centralized version control system), typically don’t offer the same out-of-the-box functions that facilitate an inner sourcing model as do DVCS (i.e. git, a distributed version control system). Of course, you can jump through hoops in some VSC to mimic properties of DVCS (for example, setting up code review watches or locking branches), but DVCS workflows are much more suited to flexible development and integration patterns from external contributors (for example, fork and pull request flow using GitHub). Not only that, but out of the box with DVCS, projects are intended to be split into separate repositories, each of which can have a committer list entirely independent of other projects. With this clear separation of administrative controls between projects, it is much easier to adopt external contributors with git than with VCS.

**Documentation**

High quality documentation should be of paramount importance to the team. It should be a first-class citizen, just as important than the shipped code. It effectively makes or breaks your ability to contribute to the project. The more incomplete developer or on-boarding docs are the higher the barrier of entry is. The more incomplete that user docs and API reference are, the harder it is for folks to integrate the project effectively.

Optimally, each project should have a split between developer and user docs. The the former tailored specifically for developers to the project and the latter enable to folks simply using the project. This split allows users and developers to concentrate solely on what’s most important to them. Documentation should be generated from code and code comments where it makes sense to preserve the integrity of the documentation. Developers are much more likely to update comments in line with code than they are to update external docs. Stale documentation can be just as bad if not worse than an overall lack of documentation as it can lead the reader down an incorrect path.

**Generalized solutions**

This is a tricky one. It’s a different approach to thinking and realizing opportunities for generalization, modularity, and reusability in system design. Generally, an immediate knee-jerk reaction of a developer is “Hey, I need feature X in Y project. I’ll just dive into code.” Even if a full design is done, it’s generally for a specific project. In either case, you end up with project-specific code that isn’t modular, extensible or reusable.

Generalized solutions take time to design, implement and test. Having the ability to recognize where you’re making assumptions and validating, they’re the right ones to make or determining an alternate, assumption-free solution can be a hard thing to do and takes a lot of trial and error. However, in the long run, thinking about problems and developing generalized solutions not only benefits you, but others within your organization. Instead of copy/pasting and refactoring, modular code and libraries can be reused by multiple teams and extended upon.

[Source](https://engineering.salesforce.com/inner-sourcing-the-mechanics-c0b1421230fd)

Welche Vorteile bietet Inner Source?

To understand the appeal of Inner Source project management, consider what has made open source software development so successful:

* Programmers share their work with a wide audience, instead of just with a manager or team. In most open source projects, anyone in the world is free to view the code, comment on it, learn new skills by examining it, and submit changes that they think will improve it or customize it to their needs.
* New code repositories (branches) based on the project can be made freely, so that sites with unanticipated uses for the code can adapt it. There are usually rules and technical support for re-merging different branches into the original master branch.
* People at large geographical distances, at separate times, can work on the same code or contribute different files of code to the same project.
* Communication tends to be written and posted to public sites instead of shared informally by word of mouth, which provides a history of the project as well as learning opportunities for new project members.
* Writing unit tests becomes a key programming task. A “unit test” is a small test that checks for particular, isolated behavior such as rejecting incorrect input or taking the proper branch under certain conditions. In open source and inner source, testing is done constantly as changes are checked in, to protect against failures during production runs.

Inner Source differs from classic open source by remaining within the view and control of a single organization. The “openness” of the project extends across many teams within the organization. This allows the organization to embed differentiating trade secrets into the code without fear that they will be revealed to outsiders, while benefitting from the creativity and diverse perspectives contributed by people throughout the organization. Often, the organization chooses to share parts of an Inner Source project with the public, effectively turning them into open source. When the technologies and management practices of open source are used internally, moving the project into a public arena becomes much easier.

Weitere Vorteile von Inner Source:

* Entwicklung wird effizienter – Zeitaufwand und Entwicklungskosten werden verringert
* Bessere Wiederverwendung – Autor und Wiederverwender sind voneinander unabhängig
* Besseres Produkt – höhere Code-Qualität und Innovationsniveau
* Flexibler Einsatz von Entwicklern – keine festen Zuweisungen ist die Kollaboration einfacher und auch mit Externen möglich
* Verbesserte Wissensmanagement – alle Artefakte sind für jeden Mitarbeiter zugänglich; Weiterbildung ist gemeinschaftsbasiert
* Höhere Motivation der Mitarbeiter

Code Ownership

Hierfür existieren 3 Konzepte:

* Strong Code Ownership – Die Code-Basis wird in Modulen aufgeteilt und jedes Modul wird an einem Entwickler fest zugewiesen (Owner). Die Entwickler dürfen nur an ihren eigenen Modulen Änderungen vornehmen. Falls eine Änderung für einen fremden Modul erwünscht ist, muss diese vom Owner durchgeführt werden.
* Weak Code Ownership – ebenfalls hier wird der Code in Modulen aufgeteilt, welche dann an den Entwicklern zugewiesen werden. Der Unterschied besteht darin, dass jeder Entwickler fremde Module bearbeiten kann. Der Owner übernimmt aber die Verantwortung für die Verfolgung der Änderungen.
* Collective Code Ownership – bei diesem Schema wird auf individuelles Modulbesitz verzichtet. Das Team als ganzes ist Besitzer der Code-Basis und Änderungen dürfen von jedem vorgenommen werden.

[Source](https://martinfowler.com/bliki/CodeOwnership.html)

Entwicklung nach dem Inner Source Philosophie

Zusammenarbeit zw. Contributor und Trusted Committer

* Contributor - Contributors may or may not be part of the community. They might be sent by another team to develop a feature the team needs. Therefore, we sometimes also refer to Contributors as Guests or as part of a Guest Team. The Contributor is responsible for "fitting in" and for conforming to the community’s expectations and processes.
* Trusted Committer - The Trusted Committer is always a member of the Inner Source community, which is also sometimes referred to as the Host Team. In this analogy, the Trusted Committer is responsible for both building the house and setting the house rules to ensure their guests are comfortable and can work together effectively. Compared to contributors, Trusted Committers have earned the responsibility to push code closer to production and are generally allowed to perform tasks that have a higher level of risk associated with them.
* Product Owner - Der Product Owner (PO) ist, wie bei den agilen Entwicklungsmethoden, für die Definition und Priorisierung von Anforderungen und deren Implementation verantwortlich. Er kommuniziert oft mit dem Trusted Committer (z.B. um sicherzustellen, dass eine Anforderung oder Beitrag im Endprodukt implementiert wird). Daher wird die Rolle oft von dem Trusted Committer übernommen, vor allem in kleineren Unternehmen.

Unterstützende Software & Werkzeuge

Um die Entwicklung besser verfolgen und analysieren zu können sind unterstützende Tools nötig

* Kommunikation – Damit keine Hürden wegen Unwissen entstehen, muss der Informationsfluss jeder Mitarbeiter erreichen. Der E-Mail-Austausch reicht selbst nicht aus, besonders in geografisch verteilte Entwicklerteams, da viele Beiträge in Echtzeit entstehen. Dazu brauch man Werkzeuge, welche sowohl synchrone als auch asynchrone Kommunikation erlaubt. Das meistbenutzte Plattform ist Slack, es sind aber auch zahlreiche Alternativen verfügbar.
* Ticketsystem – solche Systeme sind bei der Entwicklung von Software zu einer Norm geworden. Sie beinhalten Information für alle Fehler und Bugs, darüber hinaus auch über Backlog, aktueller Projektzustand etc. Die gängigen Werkzeuge sind GitHub Issues und JIRA
* Versionskontrolle – auch ein must-have Werkzeug für jedes Softwareprojekt, hier aber mit einem wichtigen Unterschied. Beiträge zum Projekt erfolgen sowohl innerhalb als auch außerhalb des Entwicklungteams, daher ist die zentralisierte Versionskontrolle mit größerem Bearbeitungsaufwand verbunden. Die verteile Versionskontrolle ist daher die bevorzugte Option, da sie ein gewisser Grad an Unabhängigkeit zwischen Beiträge gewährleistet. Solche Features sind auf das Plattform GitHub verfügbar, wie z.B. forks und pull requests.
* Dokumentation
* Code
* Mailing-Listen und Foren – bei der Zusammenarbeit können immer Unsicherheiten entstehen, daher sind Foren und Mailing-Listen bei kollaborativer Entwicklung beliebt, da sie Diskussion zwischen den Mitarbeitern schaffen können. Foren, wie z.B. StackOverflow, sind sowohl für einfache Fragen als auch für komplizierte Anliegen hilfreich. Die Threads von Mailing-Listen können zeigen, welches Projektteil mehr Arbeit benötigt oder wo sich die Aufgaben aufeinanderstapeln. Diese Threads kann man mit anderen Tools kombinieren, um genaue Kennzahlen für das ganze Projektentwicklungsprozess zu ermitteln, wie z.B. Zeitaufwand von der Idee bis zur Bereitstellung, ausgeführte Iterationen, beteiligte Personen etc.

Werkzeuge

* Communication: Slack, IRC
* Ticket System: GitHub Issues, JIRA, Bugzilla

Resources

* Wiki
* Source code + documentation

Werkzeuge

* Communication: Slack, IRC
* Version Control: Git, SVN
* Ticket System: GitHub Issues, JIRA, Bugzilla

Hilfreiche links:

* [Committer](https://en.wikipedia.org/wiki/Committer)
* [Inner Source Wikipedia](https://en.wikipedia.org/wiki/Inner_source#:~:text=Inner%20source%20is%20the%20use,Tim%20O'Reilly%20in%202000.)
* [Inner Source Commons](https://innersourcecommons.org/)