Dette er en talk om hvordan man kan jobbe Open Source i tilfeller der man ikke nødvendigvis kan være åpen.

Eller det er en talk om bruk av Open Source som arbeidsmetode.

Eventuelt kan det være en presentasjon om hvordan man kan lykkes med open source prosjekter.

OPEN SOURCE

Eller i bunn å grunn egentlig bare en presentasjon om å jobbe med open source.

Open Source – Hva er det?

- Alle av dere er eksponert for det daglig i en eller annen form. Enten indirekte eller direkte. Det kan være at man bruker det i utvikling, eller at man bare rører det via et produkt.
- Indirekte kan f.eks være bruk av Linux eller macOS som er basert på den åpne kjernen Free BSD, eller annen programvare, HTTP-servere, osv. Open Source er overalt. Men ikke alt er laget på samme måte

Ting som Linux er ting jeg liker å kalle 1. generasjons open source.

Det har vært og er usedvanlig viktig for utviklingsverden og på mange måter har satt et fundament for samfunnet generelt. Det har og en særs myteomspunnet kultur i rundt seg, og har vært tett knyttet opp mot ideologiske overbevisninger og prinsipielle utviklere – og gjerne hackekultur.

Gå enda lengre tilbake i software bransjen, til 50-tallet og 60-tallet, så ser man at programvare ofte var akademiske produkter. Og på så måte ble ofte utgitt i akademiske former og public domain, uten noe lisenser, uten noe tilgangsstyring. Men åpent og tilgjengelig.

Jeg skal ikke gå så mye inn på historie her, det hadde vært sin helt egen presentasjon, men poenget jeg prøver å gjøre er at dette tradisjonelt har vært en type open source.

Men jeg vil og påstå at i de siste 10 årene har det og kommet frem noe litt annet, noe jeg nå kaller open source generasjon 2. Litt bygd på de samme fundament, men kanskje ofte i mindre skala. Med mindre løsninger som brukes av andre utviklere som avhengigheter.

Dette er en generasjon jeg mener startet når terskelen for å bidra til open source prosjekt ble senket. Ikke lenger handlet om ofte tett knytta kjerne teams med lang historie til prosjekter og å måtte sette seg inn i endeløse eposttråder.

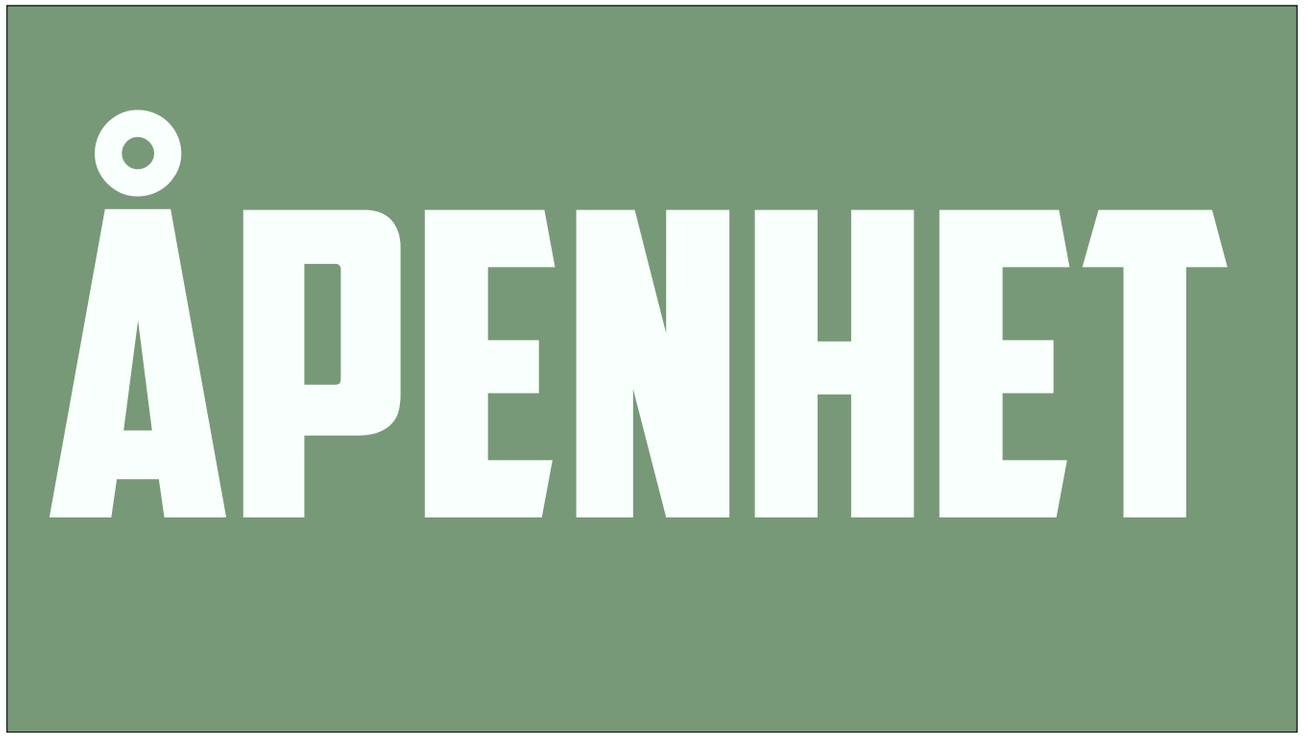
Og mye av det startet med git, Github og Pull Requesten.

På mange måter så er Open Source, etter min mening, kanskje litt misvisende navn. Det virker som at det som skal til er å gi ut kildekoden. Jeg mener Open Source ikke handler om åpen tilgang, men om åpenhet til endring. Evnen til å kunne dele på eierskap og sammen skape noe som er bedre enn det en kunne ha laget. Dette er og i stor tråd med den originale ideologien til Open Source.

Open Source er ikke det samme som disclosed source, men åpenhet som default, som en ryggmargsrefleks. Og dette må til for å lykkes. Vi så f.eks eksempel på det når Apple la ut Swift sin kildekode og den initielle tracktion det fikk i motsetning til når Microsoft open sourced .NET-plattformen. I ettertid har og Swift blitt mer åpen og endringsvillig.

Konsekvensen av Open Source er og prosjektet må følges opp. Du kan ikke kaste kildekoden over i en pølje men mennesker og håpe på det beste, men prosjektet må pleies og opprettholdes.

Open Source handler ikke om lesetilgang, det er en metodikk, en utviklingsmetode og et tankesett. Og dette er prinsipper man kan holde seg til og forbedre sin kodebase, selv om man ikke har muligheten til å gi ut kildekoden. Og det er det vi skal snakke om her i dag. Hvordan gode Open Source-prinsipper er gode prinsipper for alle kodebaser, og hvordan vi kan la Open Source gjøre prosjektene våre bedre.



Men når det er sagt, denne presentasjonen er et slags kompromiss. Jeg mener vi burde bli mye flinkere til å tørre å være åpne, tørre å gi ut kode og oppfordre prosjekteiere til å gi ut kode. I senere år er mange blitt bedre til dette så jeg tror det er på bedringens vei.

Etterhvert som dere kommer ut i arbeidsmarkedet, kan dere merke på at det ikke alltid er like enkelt å få gjennomslag for det man selv tror er rett eller å håndtere all politikk som ofte er tilknyttet avgjørelser.

Denne presentasjonen vil på mange måter og være en liste over egenskaper som må til for vellykket open source prosjekt.

Målet med presentasjonen er å gi konkrete råd for hvilke prinsipper man lære av Open Source prosjekter for å forbedre sine egne prosjekter. For at man skal kunne følge disse prinsippene forutsetter det at man gjennomfører prinsipp nr 1 først, som er å splitte opp prosjekter:

PRINSIPP #1

SPLITT OPP ETTER ANSVAR

I et open source prosjekt løser dette seg selv, men dette er og noe man kan tenke på når man lager et “lukket prosjekt”. Er det noe som kan trekkes ut til en egen pakke? Gjør det. Om mulig, legg det ut åpent. Dette er en forutsetning for å kunne følge de øvrige prinsippene.

Man må kunne identifisere hva er det som kan isoleres, hva er det som er generelt og kan være en egen pakke.

Her snakker jeg ikke bare om biblioteker, men og spesifikk kode for det prosjektet du jobber på. Gjerne og domenespesifikt. Du kan la det leve et eget liv i en egen mappe og kunne følge disse prinsippene. Dette trenger ikke leve i sitt eget github-repo, det kan leve som en del av et monorepo, men semantisk og fysisk skille burde eksistere.

PRINSIPP #2

BRUK TID PÅ README

READMEs vil være inngangen til pakken din. Det er viktig at den sier noe om hva dette er, hva den skal løse og hvordan den løser det. Kom til poenget så tidlig som mulig, uten at det står for mye tekst. Folk er overraskende utålmodige lesere. Gjerne ha et enkelt brukseksempel i første bilde man ser, uten at man må scrolle.

README vil være dokumentasjonen til pakken din og. Det er ofte en fordel å ha dokumentasjonen nære faktisk pakkeinnhold. Dette gjør terskelen for å oppdatere mindre. Det gjør og terskelen for å lese dokumentasjon mindre.

README må og dokumentere om det er noen spesifikke forutsetninger for prosjektet. Hvordan kan det taes i bruk.

Dokumenter også hvordan man kan endre på pakken. Er det noen spesielle ting som må til for at det kjører? Er det noe lokalt oppsett som må gjøres? Selv om det er bare til internt bruk, tenk at det er eksterne konsumenter som ikke vet mye om prosjektet. Sjansen er at X måneder frem i tid, så kommer det noen som ikke vet hva det er og da er dette til stor hjelp.

PRINSIPP #3

HA UTFYLLENDE EKSEMPLER

I tillegg til grunnleggende eksempel for bruk i README, ha forskjellige utfyllende eksempler i egne filer som viser forskjellig bruk om det er noe. Disse burde også være kjørbare med enkel kommando fra README-posisjonen, uten mye oppsett. Automatiser installasjon og kjøring av eksempel via enkel verktøy fra kommandolinjen.

Om mulig og relevant, ha kjørende eksempel på en nettside som man kan åpne uten å laste ned koden – eksempelvis playbooks. Slik kan man se hvordan det er tiltenkt at det skal fungere. Det gjør det og mye lettere å validere om man har forstått ting rett og om man videreutvikler det, har man noe å jobbe med som fungerende eksempel.

PRINSIPP #4

HA DEKKENDE TESTER

Selv om man har eksempler som man kan kjøre, burde pakken ha tester. Og da er det snakk om spesifikke tester, isolert til denne pakken, uten avhengigheter til andre ting. Konkrete tester som validerer denne eksakte pakken.

Ikke bare gjør det at man ser mye mer eksempel på bruk, men det er og en trygghet for videreutvikling og regresjon. Om man kommer inn i et prosjekt hvor det ikke er realistiske, dekkende tester, føles det raskt veldig usikkert å gjøre endringer uten å måtte sette seg inn i hele kodebasen.

Om man ønsker å sørge for at en kodestil blir holdt, kan man og legge inn tester for det. README burde dekke hvordan man skal bruke disse verktøyene. I tillegg, om man bruker Github, kan man bruke Github Markedplace til å integrere opp mot forskjellige tjenester som sier om kodestil divergerer eller tester feiler.

PRINSIPP #5

TENK API & VERSJONERING

Når man har eksterne konsumenter er det viktig å tenke på API-overflate. Både hvordan man designer det (altså hvordan det er å bruke prosjektet), men og dokumentasjon og versjonering av endring.

Vi har en tendens ti å slurve med måten vi designer ting på dersom vi tenker det skal være internt. Men det er ingen unnskyldning. Selv om vi bare skal bruke noe internt, er dårlige API-er en slippery slope til dårlig kvalitet på produktet og vanskeligere vedlikehold. Dette blir lignende broken window theory. Ved å begrense mindre brudd på kvalitet, kan vi være med å minske større og mer seriøse brudd.

Når man endrer koden, må man sørge for å holde orden på om API-et blir endret. Håndter versjoner ut i fra semantic versioning. Er det en bugfiks som ikke endrer API? Patch. Er det en tillegg i API-et men bakoverkompatibelt? Minor. Er det en endring som gjør at det bytter om på bruken for konsumentens del? Major.

API er en sentral del for isolerte pakker. Det er en viktig sukksess-faktor.

PRINSIPP #6
VÆR ÅPEN FOR ENDRING!

Kanskje det viktigste prinsippet av de alle er å kunne være åpen for endring. Alle foregående prinsipper hjelper for konsumering av prosjektet, men er minst like viktig for å senke terskel for å få inn endringer. Det er viktig å tenke på kodebasen som noe under kontinuerlig endring. Bruk Pull Requests aktivt og vis at kode som kommer inn blir merget og arbeidet andre gjør er satt pris på.

En må kunne ha eierskap til prosjektet, men trenger ikke nødvendigvis å ha eierskap til spesifikke kodesnutter. Og det kan være vanskelig for vi føler ofte at koden vi skriver er en del av oss og en forlengelse av våre tanker - og at det dermed blir personlig. Slik er det ikke og det er en følelse vi burde jobbe med å bli kvitt. Det er og viktig å kjenne på det og være respektfull når man vil få inn endringer. Ikke bruk nedlatende termer, men vær diplomatisk i ordbruk og respekter andres arbeid.

Jobb med å dele eierskap for hele prosjektet på tvers av flere mennesker. Ikke min og din kode, men vår kode.

Kode må vedlikeholdes, kode må opprettholdes. Kontinuerlig. En oppnår det bedre med å gjøre innsatsen for å bidra så liten som mulig og vise at man er endringsvillig. Ikke bare blir det enklere for andre å endre, men det blir og enklere for deg.


@MIKAELBREVIK