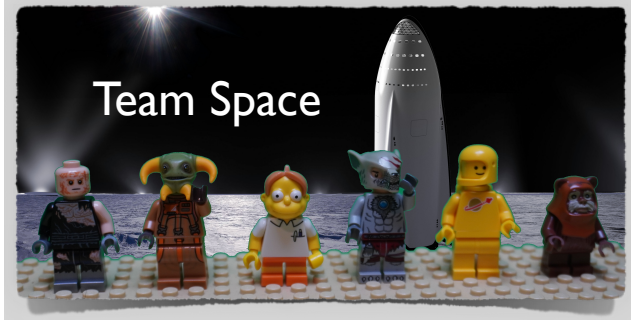

Less


CERTIFIED
Less
PRACTITIONER

# Maximizing Dependencies with Interdependent Teams









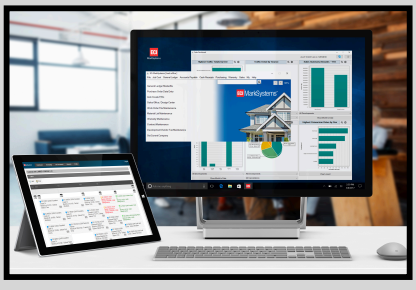








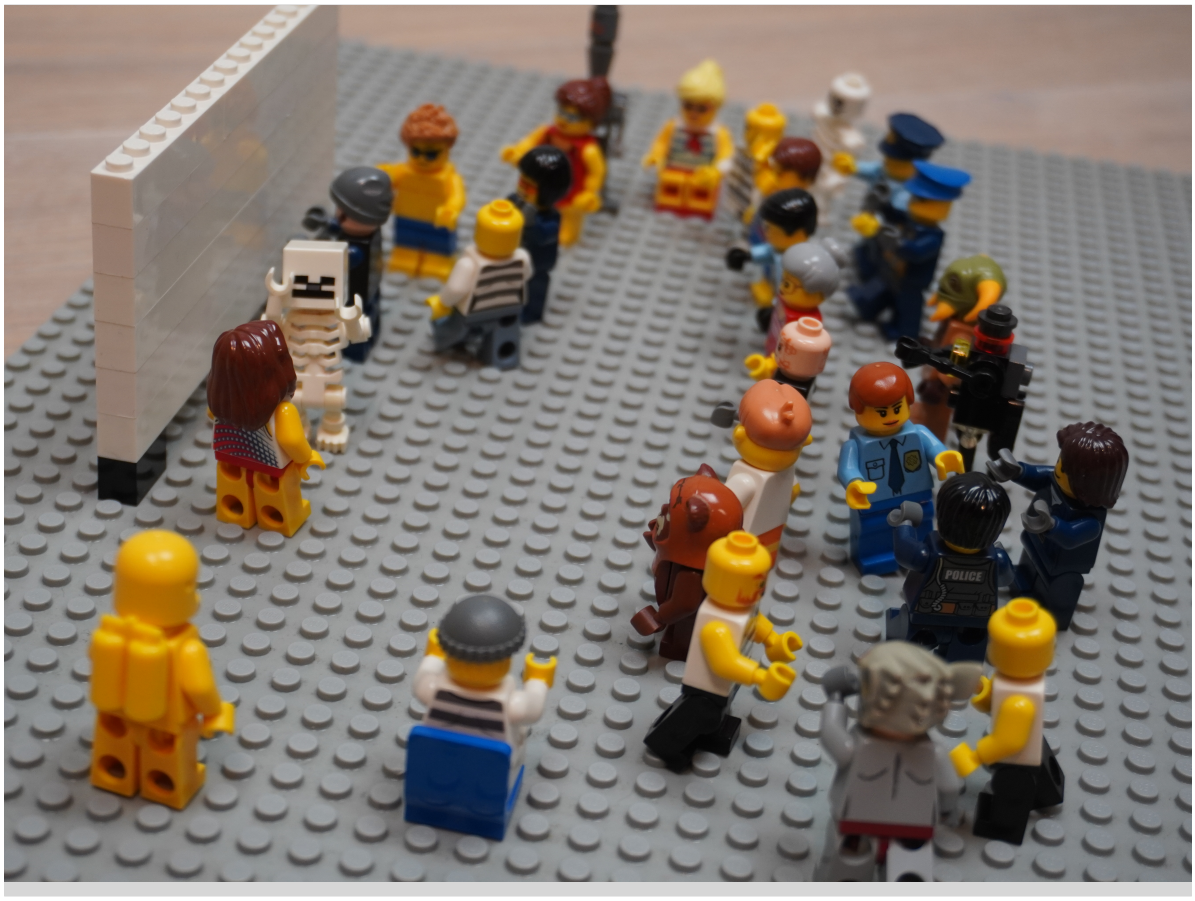

# Features

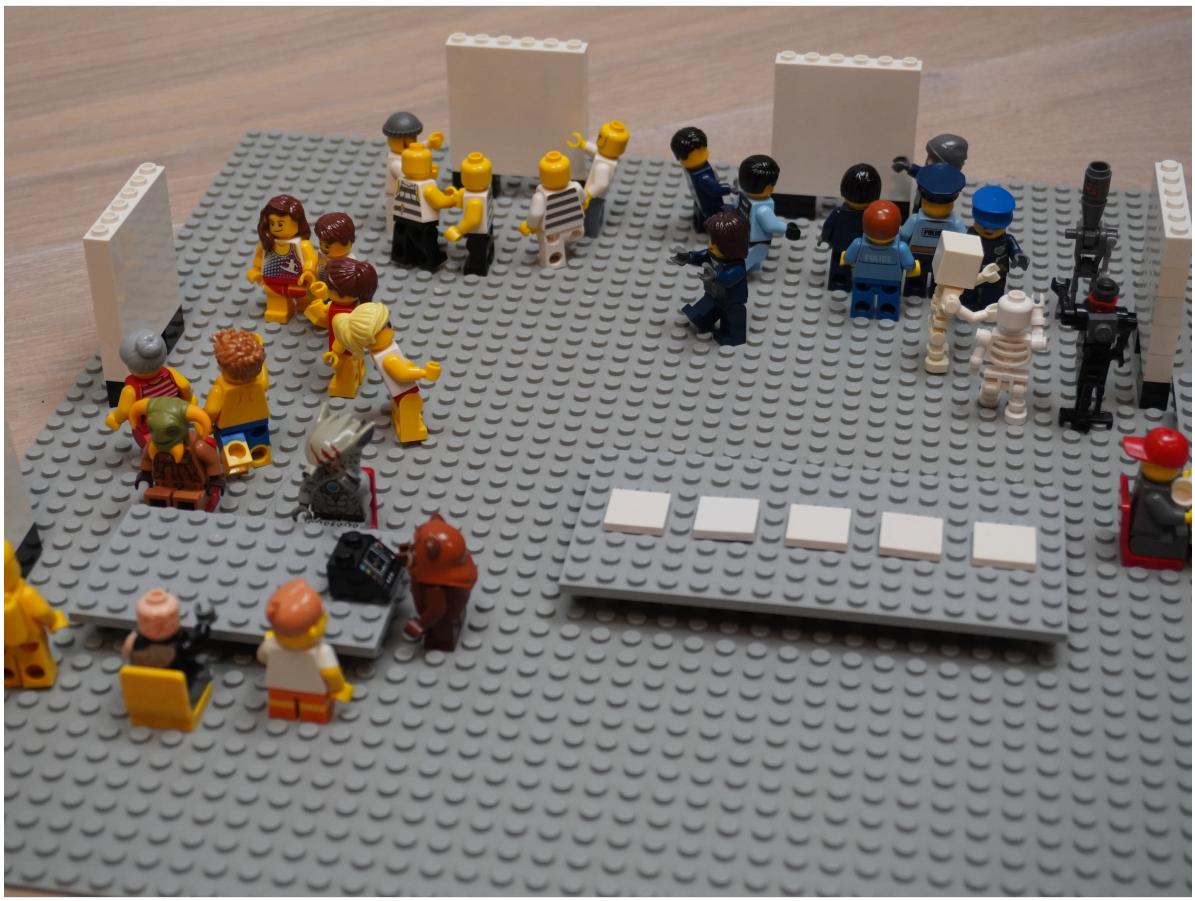


## Top of Product Backlog

- Viewing courses with direct link
- Course registration
- Basic integration with External Scheduler
- Migrate main page to New Tech
- Course registration with invoice
- External Scheduler support for timezones
- Migrate menus to New Tech
- Course registration through payment provider
- Cancelling registration without payment




POLICE

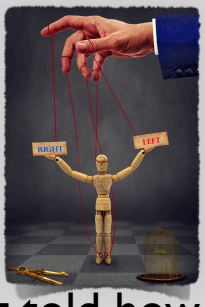






Enablers

# Two Journeys




Being told how to work



Self-managing Team



Programmer



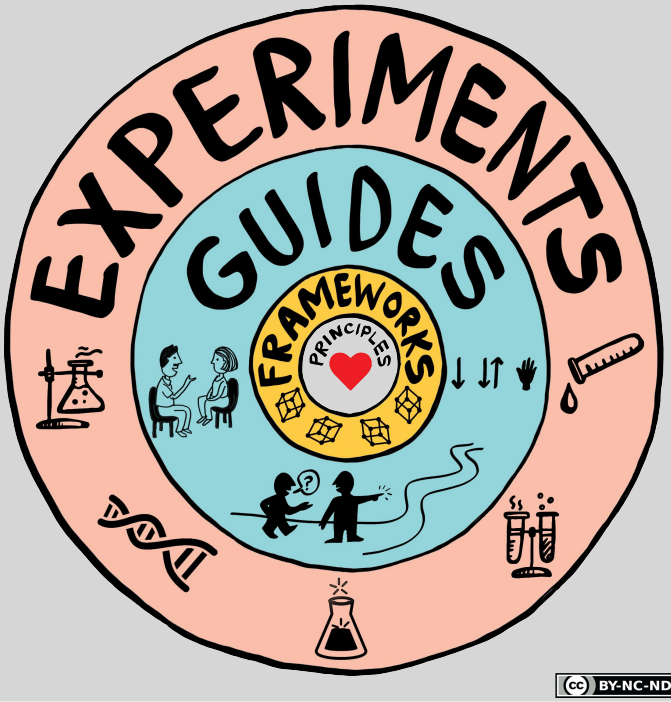
Product Developer

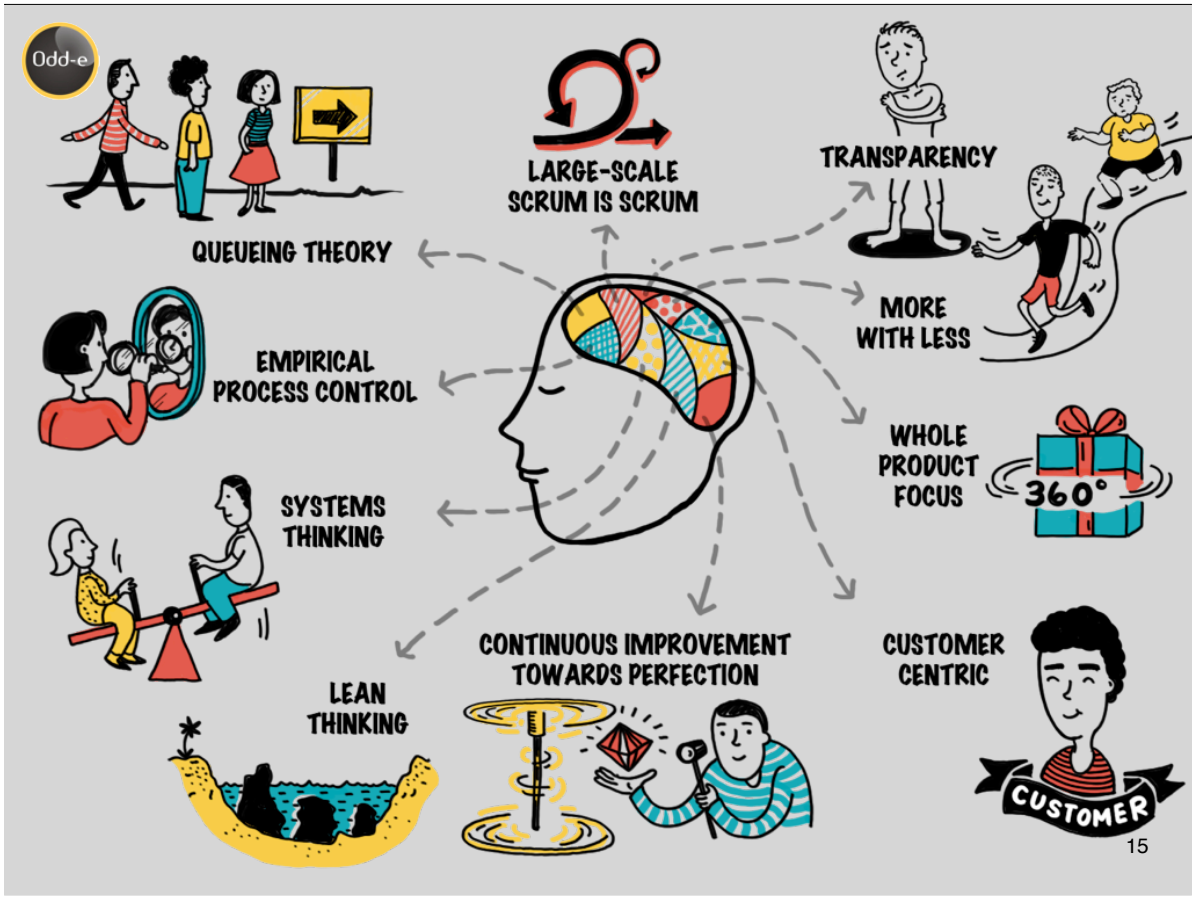


BY-NC-ND

Odd-e

# LeSS



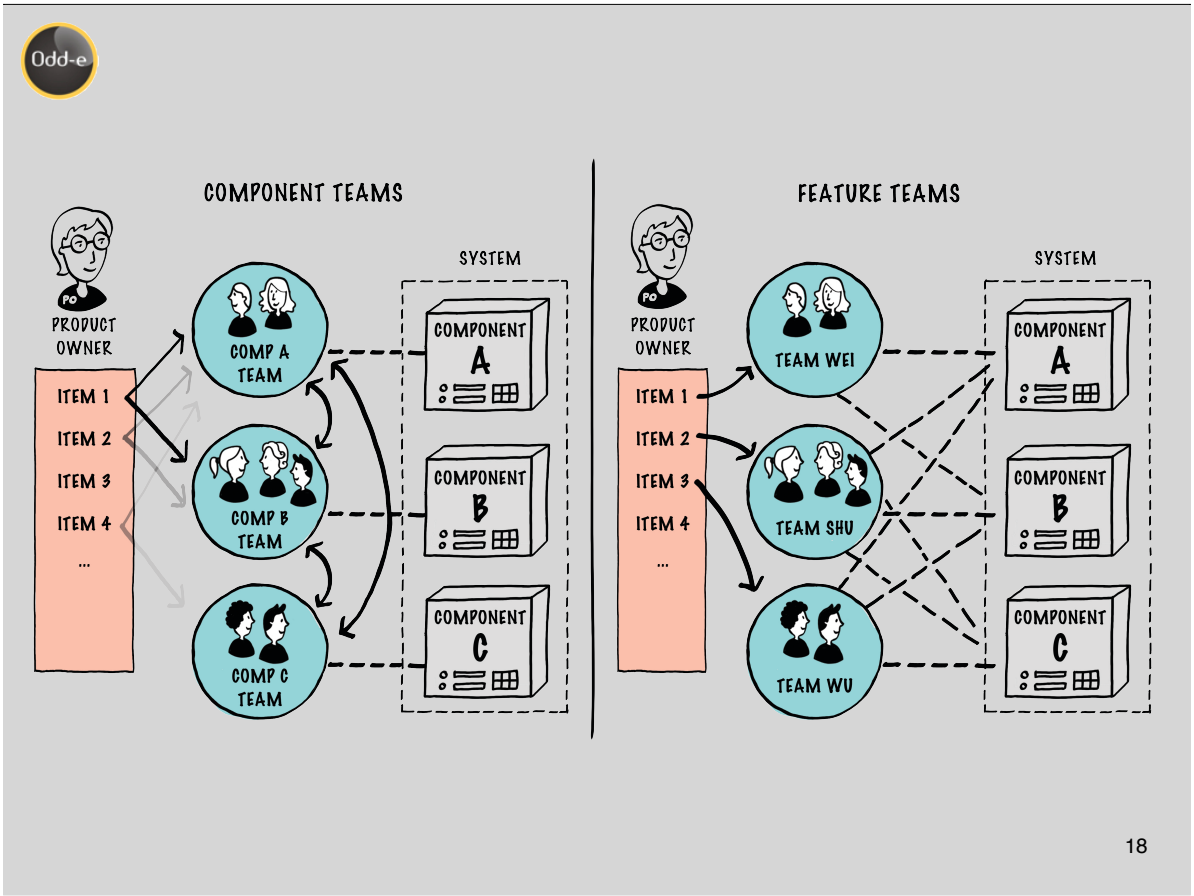



# Whole Product Focus



# Customer Centric

THE RITZ-CARLTON



# Types of Dependencies



By Bundesarchiv, Bild 183-1990-0414-009 / Wolfried Pätzold / CC-BY-SA 3.0, CC BY-SA 3.0

Asynchronous Dependencies
Interrupt

vs

Synchronous Dependencies
Collaborate



# Interdependent Teams



Independent
Isolated
Teams

vs

Cross-Team Shared Learning



# Maximizing Dependencies



Dependent on

vs

Shared Work



Feature Teams together
**chose** which team works on
which feature
**rather than**
all work for a specific
component always **goes to**
a specific team

# Accidental Specialization

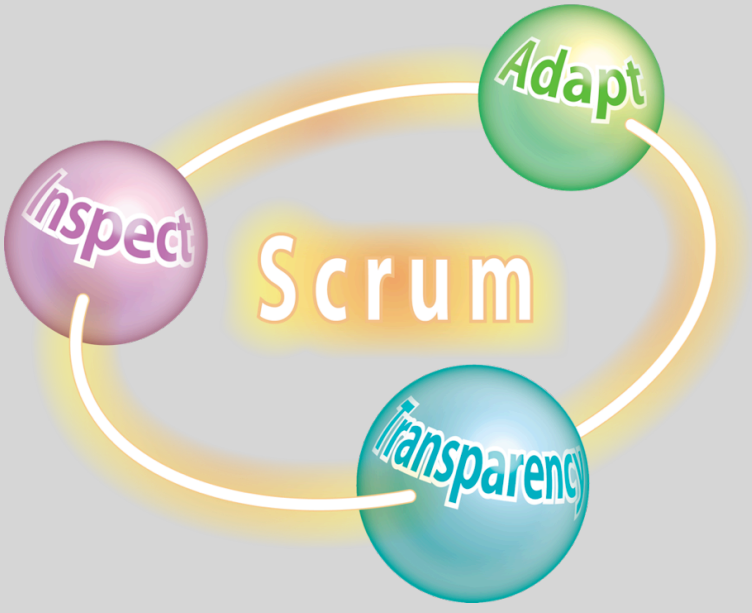


Example

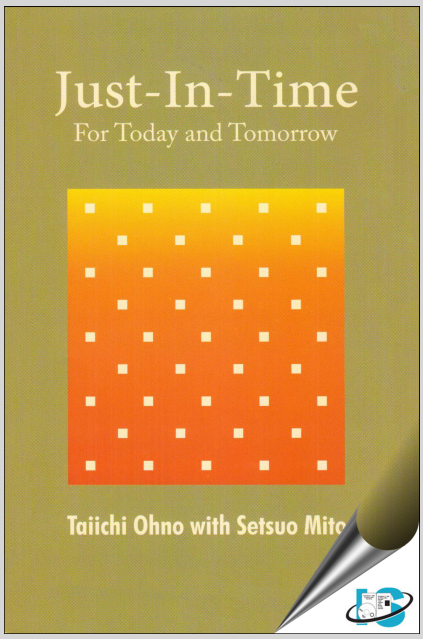
Team HUB likes:

- Features that are complex and touch many components
- Features that are used by lecturers and students
- Features that have a graphical UI

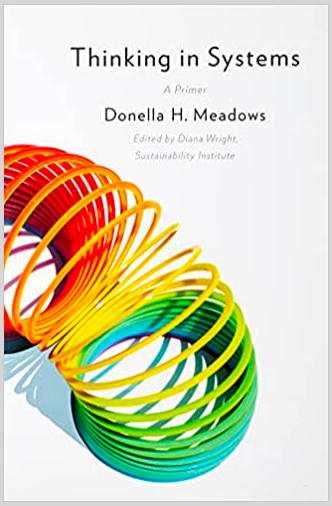
# Large-Scale Scrum is Scrum



# One piece flow




# Avoiding Tragedy of the Commons




- *Educate and exhaust.*

  Help people to see the consequences of unrestrained use of the commons. Appeal to their morality

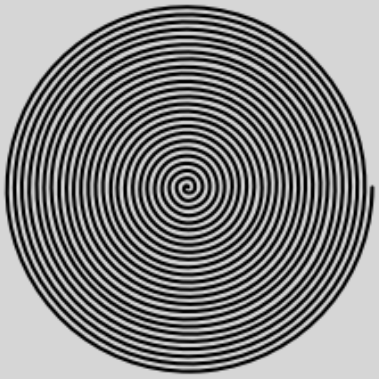
- *Privatize the commons.*

  Divide it up, so that each person reaps the consequences of his or her own actions.

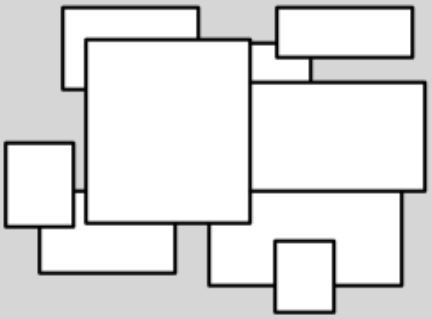
- *Regulate the commons.*

  Mutual coercion, mutually agreed upon.

# Incremental Design



growing



building

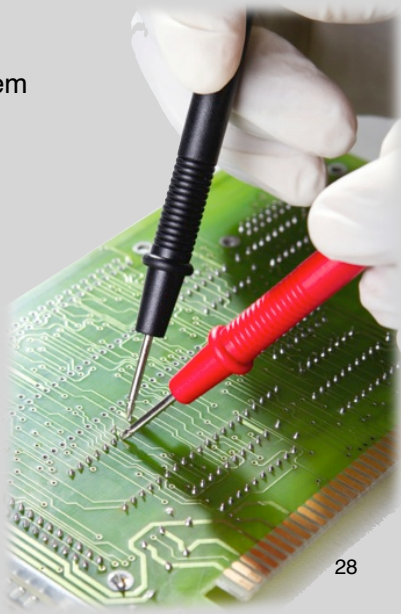
This seems trivial but dramatically impacts how developers do their work.



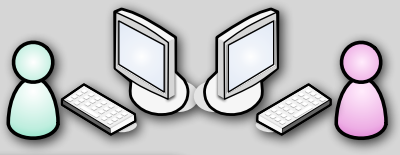
# Continuous Integration

Continuous Integration is a **developer practice**
with the goal to always keep a **working system**
by making **small changes**, slowly growing the system
and **integrating** them at least **daily**
on the **mainline**
typically supported by a **CI system**
with lots of **automated tests**

- Increases transparency
- Increases cooperation and communication
- Enables people to work on same code



# Scenario



Hey! I'm changing the method that conflicts with your changes!

Roger!

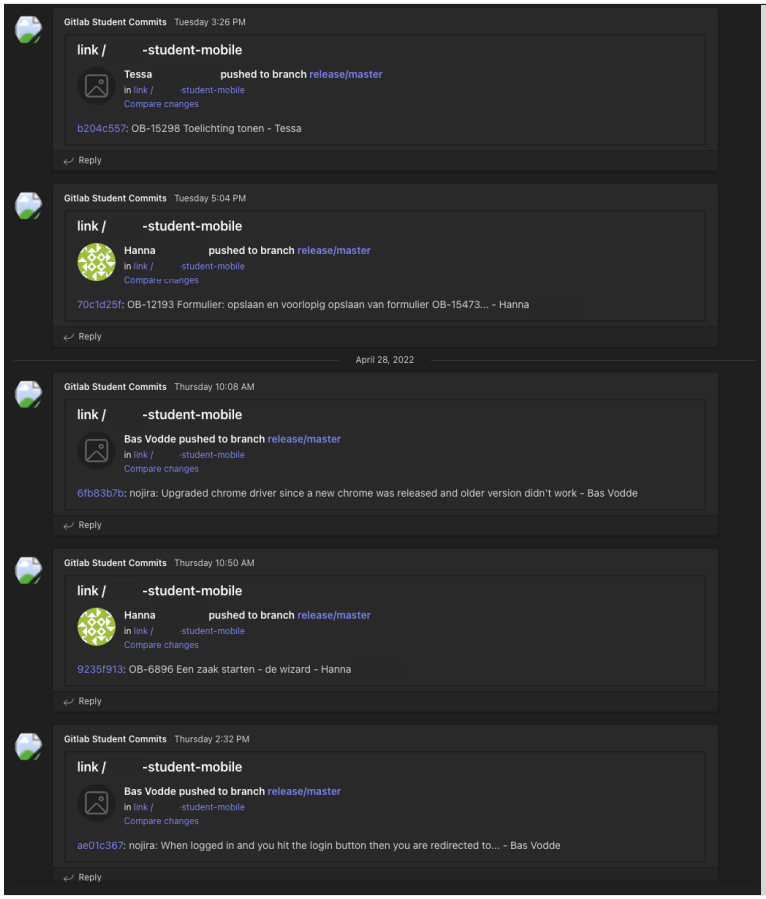
... done and committed!

Ok! I'll update and check the merged changes!
... Looks good!

Great!

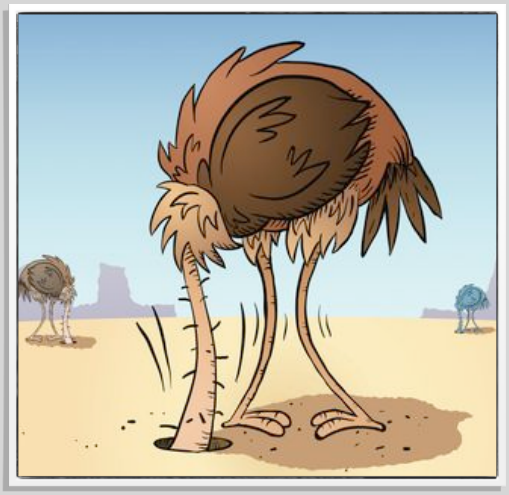


# Communicate in Code






# Avoid branching!








# Mono Repository

contributed articles

DOI:10.1145/2854146

**Google's monolithic repository provides a common source of truth for tens of thousands of developers around the world.**

BY RACHEL POTVIN AND JOSH LEVENBERG

## Why Google Stores Billions of Lines of Code in a Single Repository

This article ou
codebase and de
built monolithic
the reasons the
Google uses a ho
trol system to ho
visible to, and us
ware developers
centralized syste
many of Google's
Here, we provid
systems and wor
sible managing
tively with such
explain Google's
opment" strateg
tems that structu
Google's codeba
software for stati
up, and streamli

**Google-Scale**

Google's monol
tory, which is us
ware developer
the definition o
system, providi
gle-source repo
scaled successfu

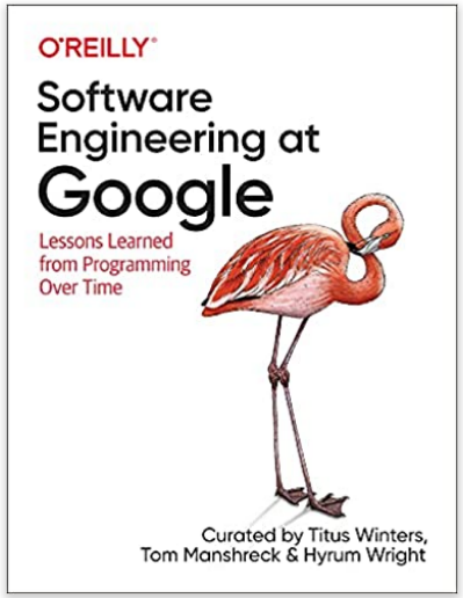
The Google c
proximately one
a history of app
commits spanni
year existence. T
86TB[a] of data, in

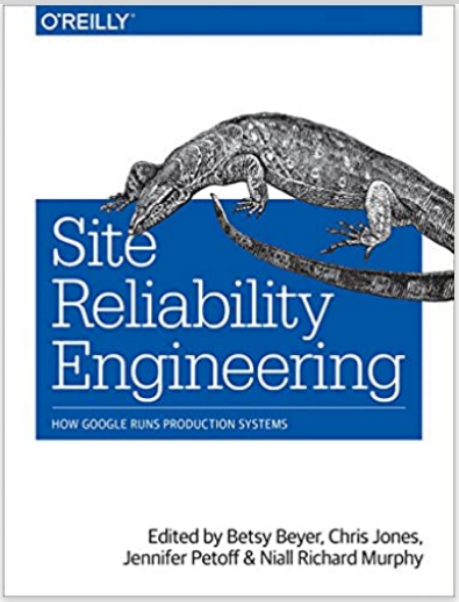
a Total size of uncom
release branches.

# Maximizing Dependencies with Interdependent Teams



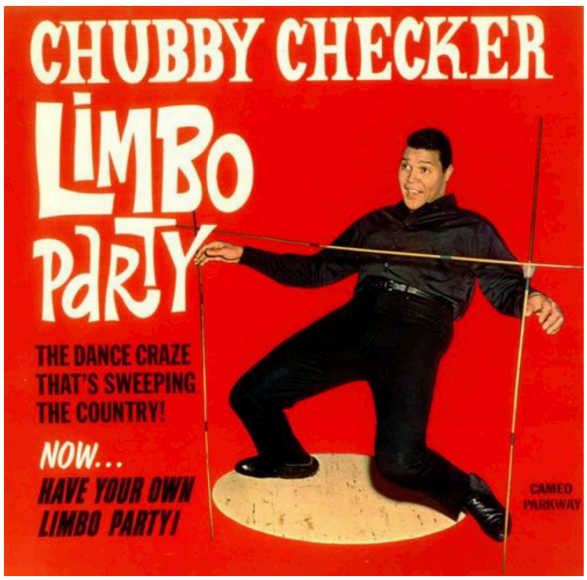
# Google




## Limbo: Scaling Software Collaboration

Kent Beck Jul 11, 2018 · 6 min read



# Micro-services ?

- An architectural style that decomposes the system in services that are independently deployable.
- When using micro-services (and LeSS), avoid:
  - Teams own services (also known as component teams)
  - Repository per services (also known as multi-repo)
- But be aware:
  - Service communication creates additional overhead
  - Additional network communication creates significant additional complexity




# Example Incremental Upgrade (1)

- Refactor all usage of an API that is changing to one place
- Move all usage of changing API again into one component/class. For example this could be called “UpgradeService” and it only exists during the upgrade
- Use a bash script that does the upgrade. It does:
  - Simple renames of changes in the API. Using find and sed.
  - Renames files that are called <file>.upgrade to <file> so that it is possible to have two versions of a file without branching in repository. This is mostly used for configuration but also for e.g. “upgrade service”
  - Deletes all lines that are marked for deletion with a comment. For example “// ION3”
  - Runs an open source upgrade script that does more renames and changes




# Example Incremental Upgrade (2)

- For all incompatabilities, first ask whether we can find a way that works in both versions, if so, refactor it to a different implementation that works for both implementations
- For things that do not exist in the new version, write a dummy wrapper so that we can compile it, it just doesn't do anything
- For things that do not exist in the old version, write an implementation that doesn't do anything in the old version, but does something in the new version
- An example of the above is best visible in the CSS where the changes are dramatic. We add CSS for the new version (which doesn't do anything in the old version, it is just dead code) and then for the old styles, we mark them as "// ION3" so that they will be deleted by the upgrade script






# Example Incremental Upgrade (3)

- People work with 2 checked out versions (same code branch). One version that has run the script and one that hasn't. Try the change in the upgraded version, then manually copy it to the old version to see if it works there also, then we always only commit on the old version checkout (so we never commit the results of the upgrade script)


MORE
with
LESS.