To Branch, or Not to Branch …

… that is the question:

Whether 'tis nobler in the mind to suffer
The slings and arrows of ridiculous merges,
Or to sail the master against a sea of troubles.

ein Drama von
René Preißel und Bjørn Stachmann
frei nach
William Shakespeare
oder so ähnlich

Sir René
of the
Golden Master

Sir Bjørn
Baron of
the Branches

To Branch, or Not to Branch ...

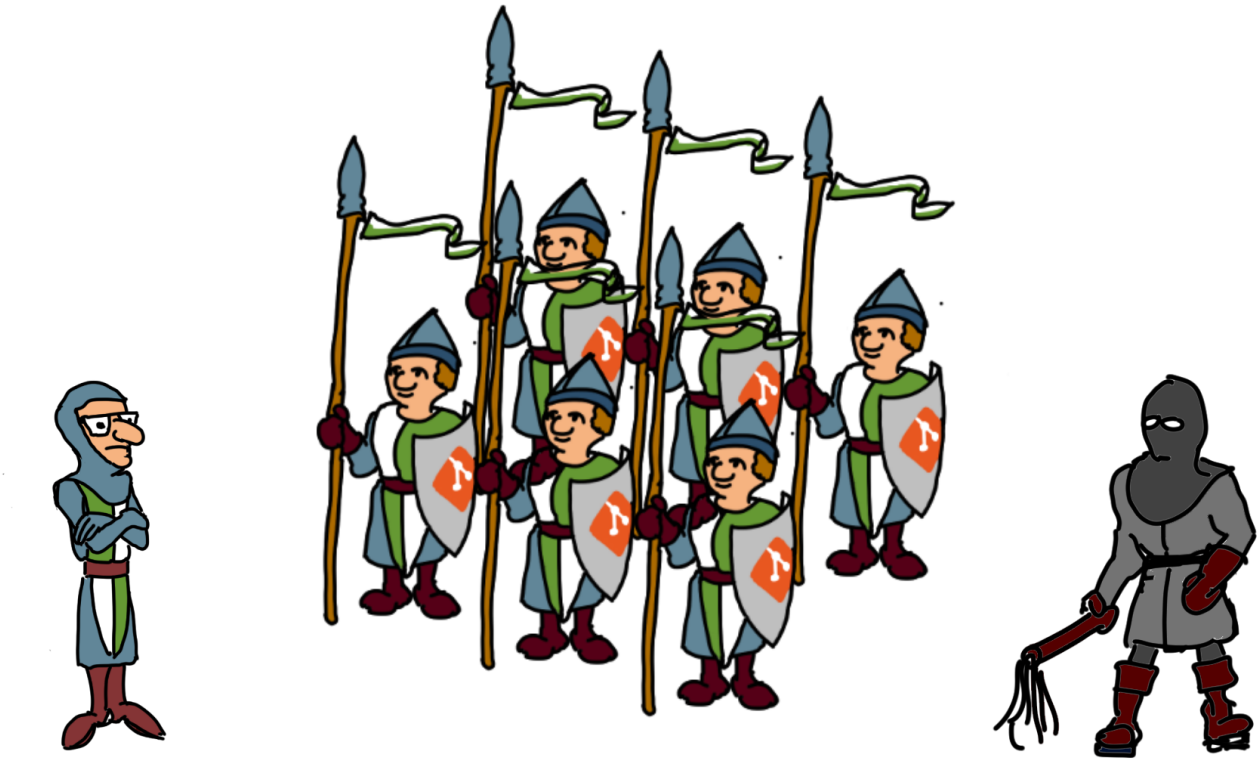
... that is the question:

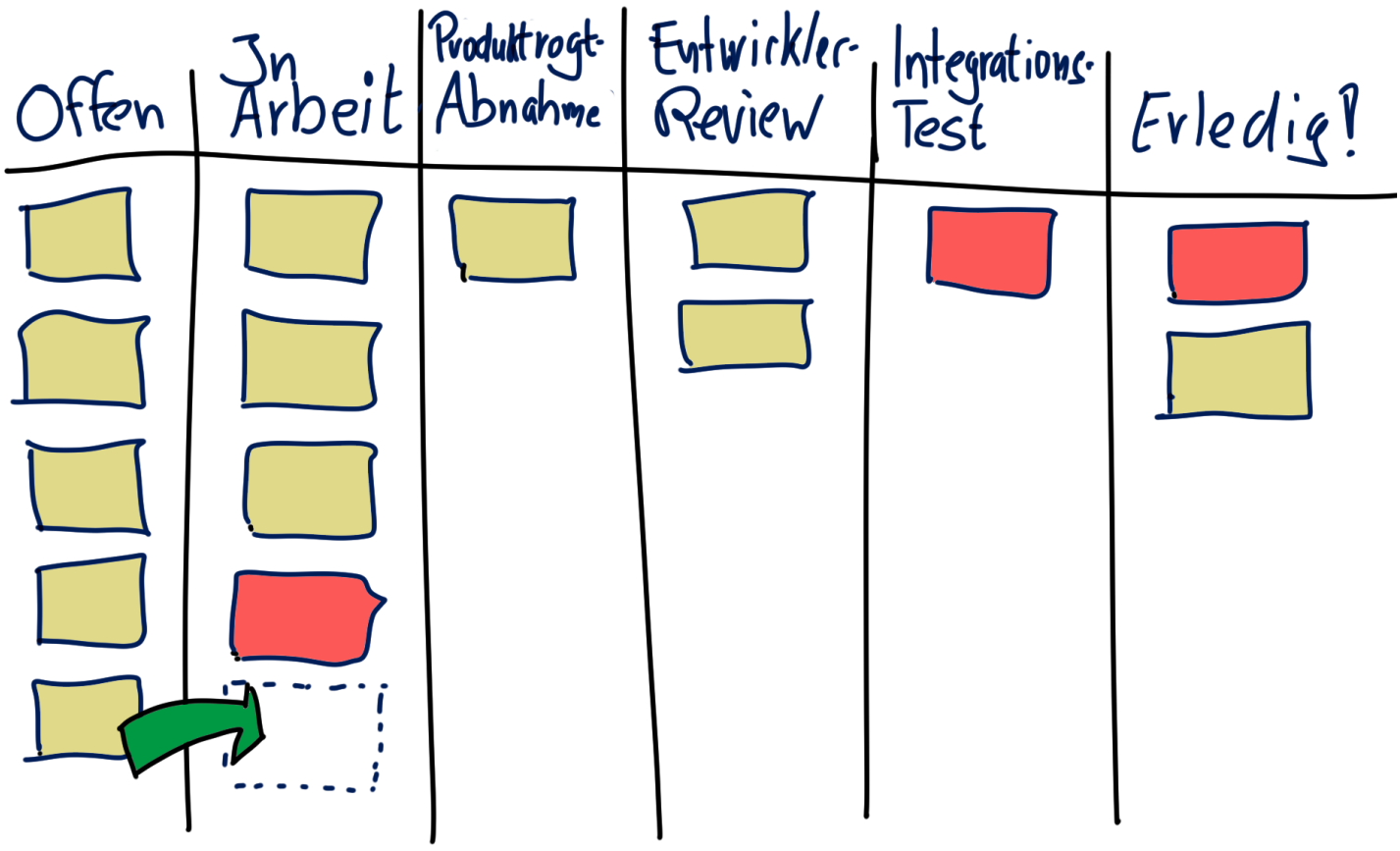
Whether 'tis nobler in the mind to suffer
The slings and arrows of ridiculous outrages,
Or to take the master against a sea of troubles.

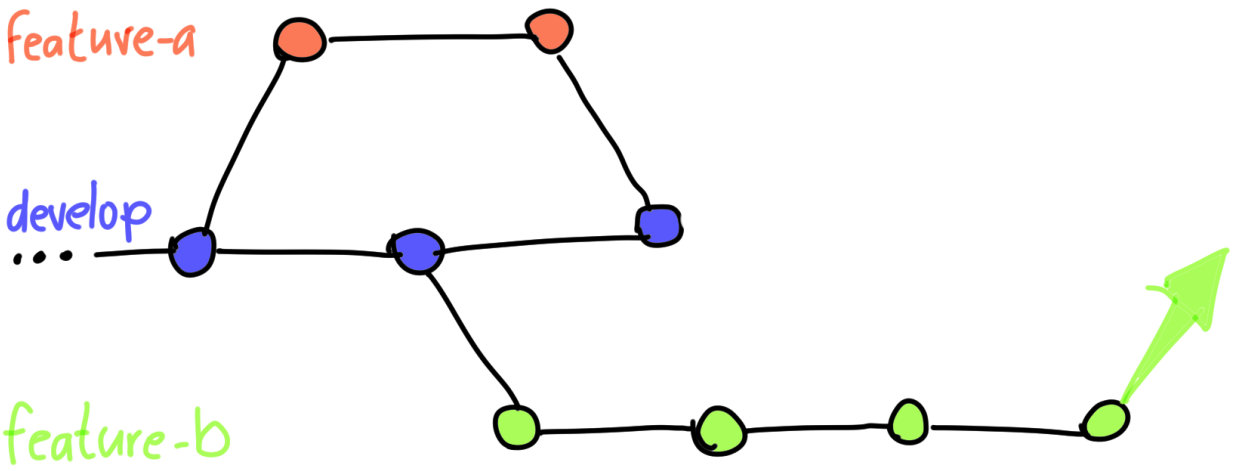
ein Drama von
René Preißel und Bjørn Stachmann
frei nach
William Shakespeare
oder so ähnlich

Mit Git entwickeln:
Wie man es richtig macht.

ein Drama von
René Preißel und Bjørn Stachmann
frei nach
William Shakespeare
oder so ähnlich



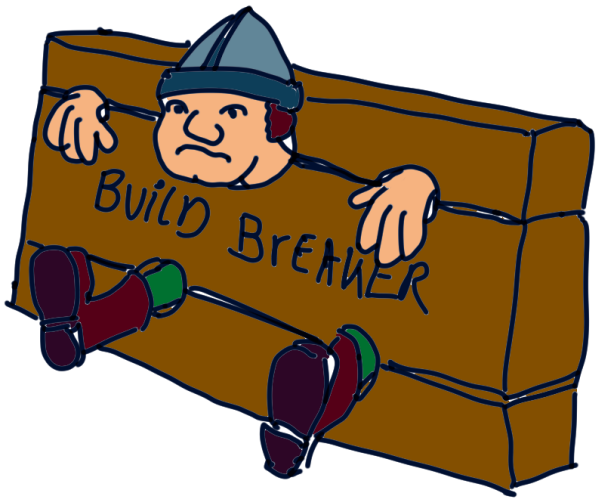

Offen
In Arbeit
Produktrogt. Abnahme
Entwickler. Review
Integrations. Test
Erledig!

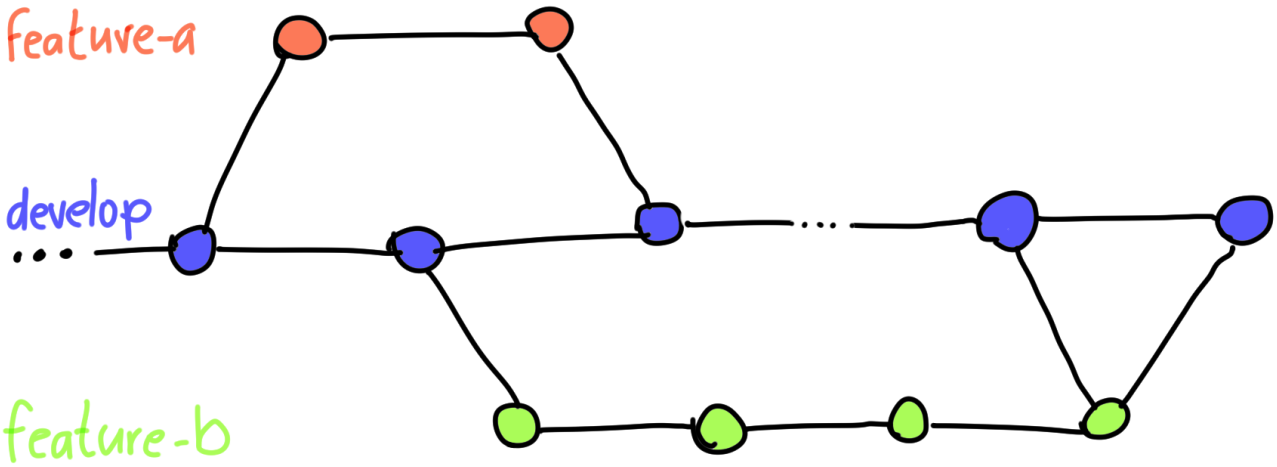


Unsere goldene 1-2-3-Regel

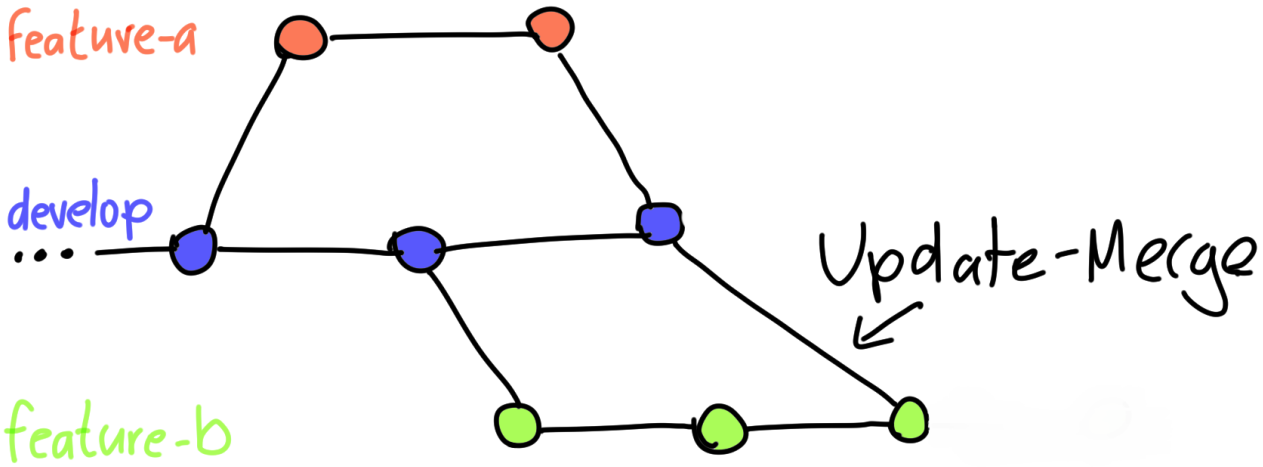
Vor dem Merge

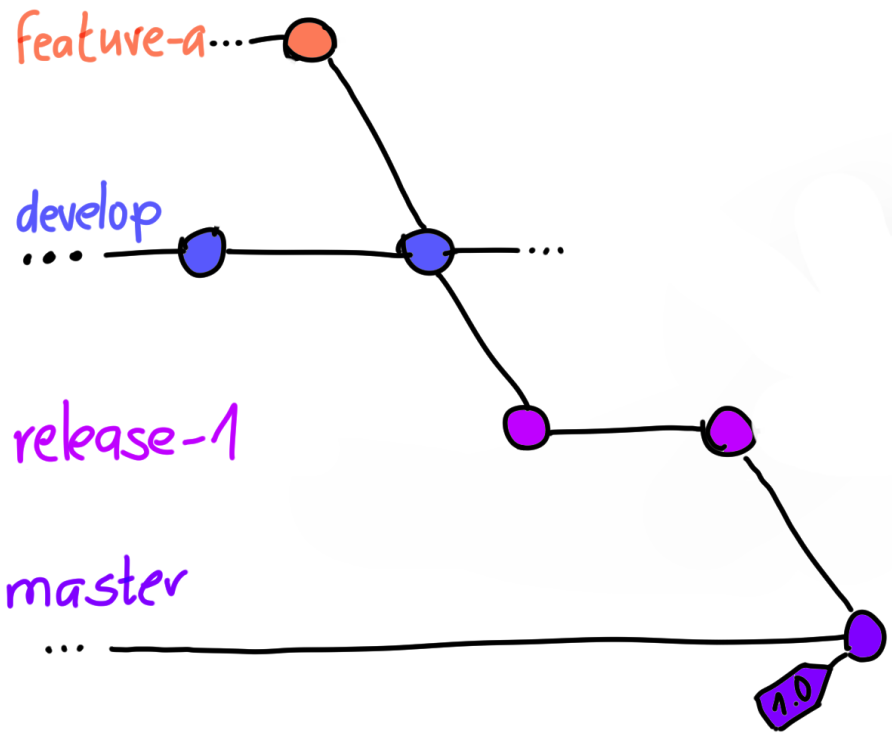
1. Audienz beim Produktvogt
2. Review durch erfahrenen Kollegen
3. Vorab Integrationstest


BUILD BREAKER


feature-a
develop
feature-b

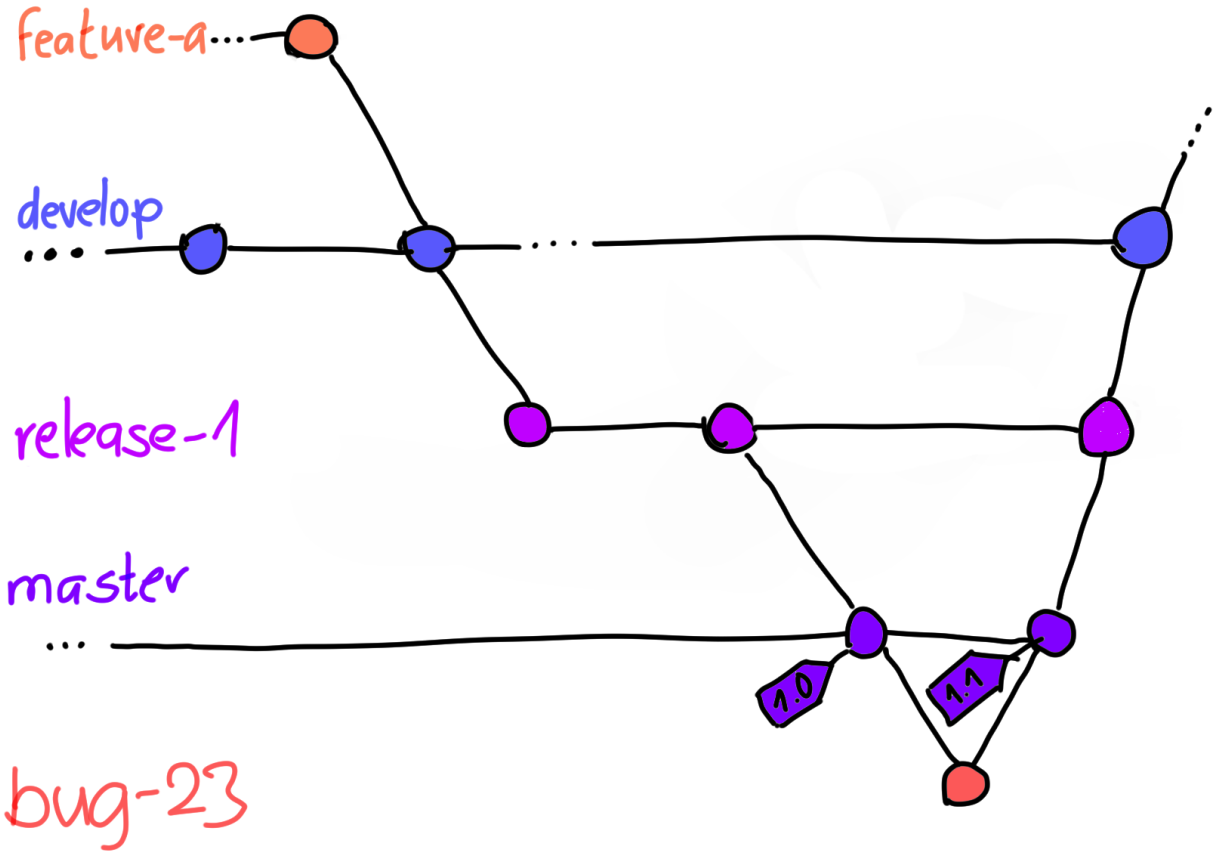
Und wenn ich sehen möchte,
Was die anderen gemacht haben?





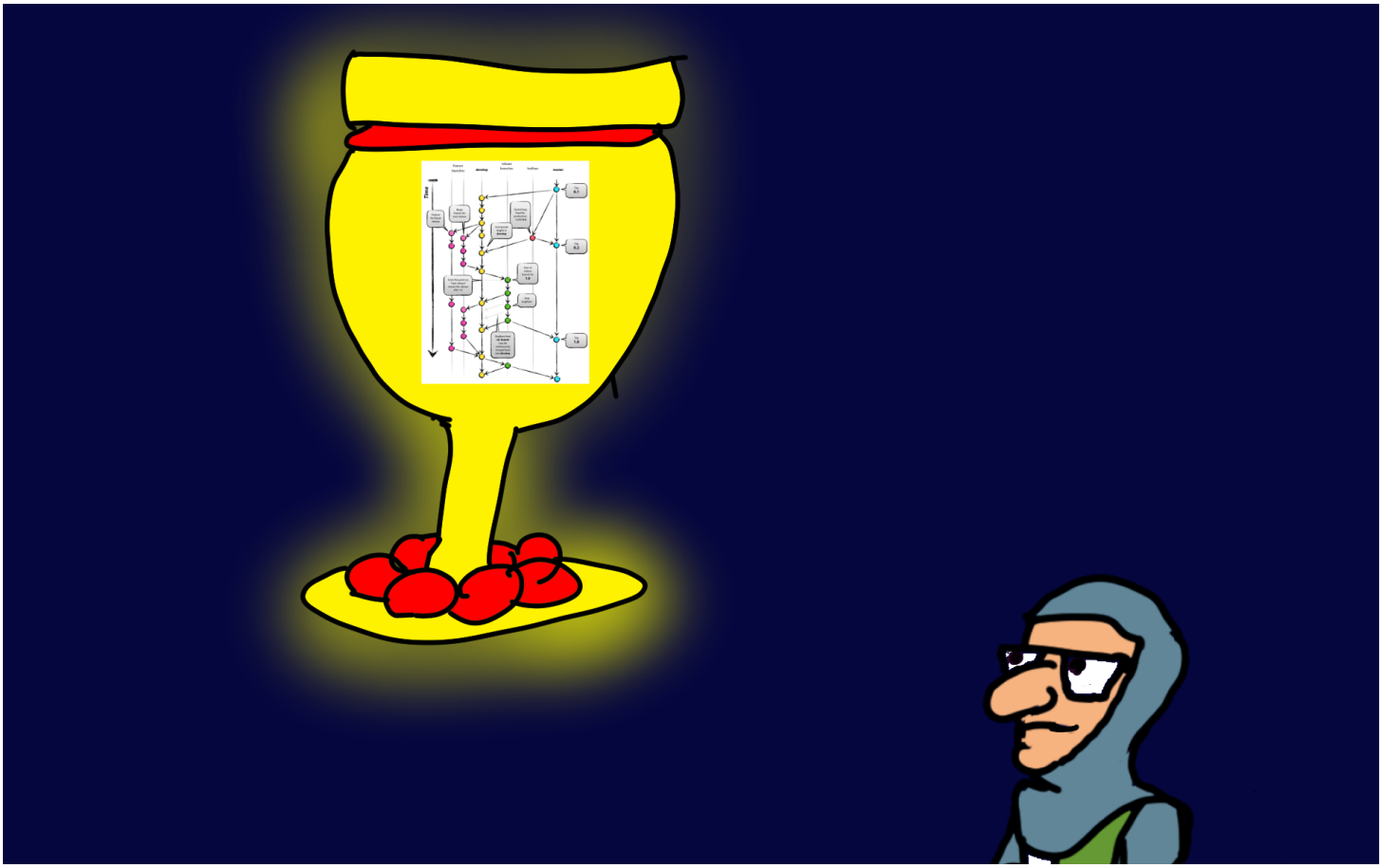
Bugs?

Chuck Norris does not go bug hunting.
"Hunting" would imply the possibility of failure.
Chuck Norris goes bug killing.



Und nun komme der nächste Sprint

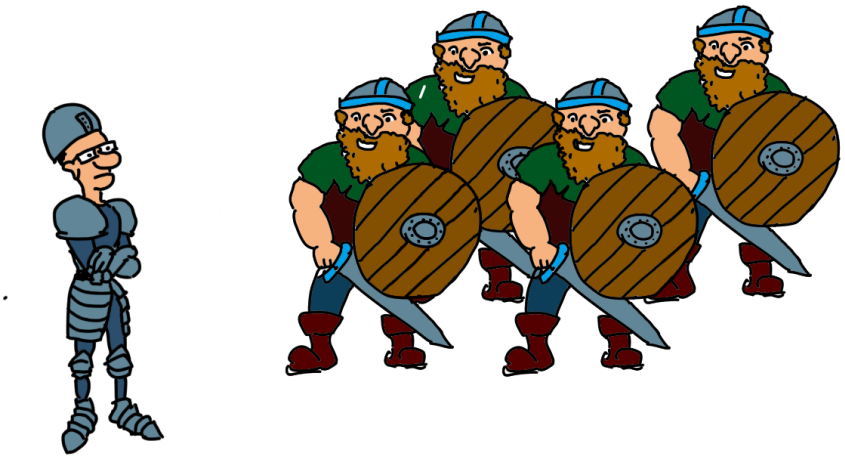
So arbeitet man mit Git!



Daniel Terhorst-North
@tastapod

How to institutionalise and further entrench the insanity that is feature branches (or anything other than trunk-based development: TrunkBasedDevelopment.com).

Friends don't let friends use feature branches or GitFlow. Friends keep all their code on master and use feature toggles.

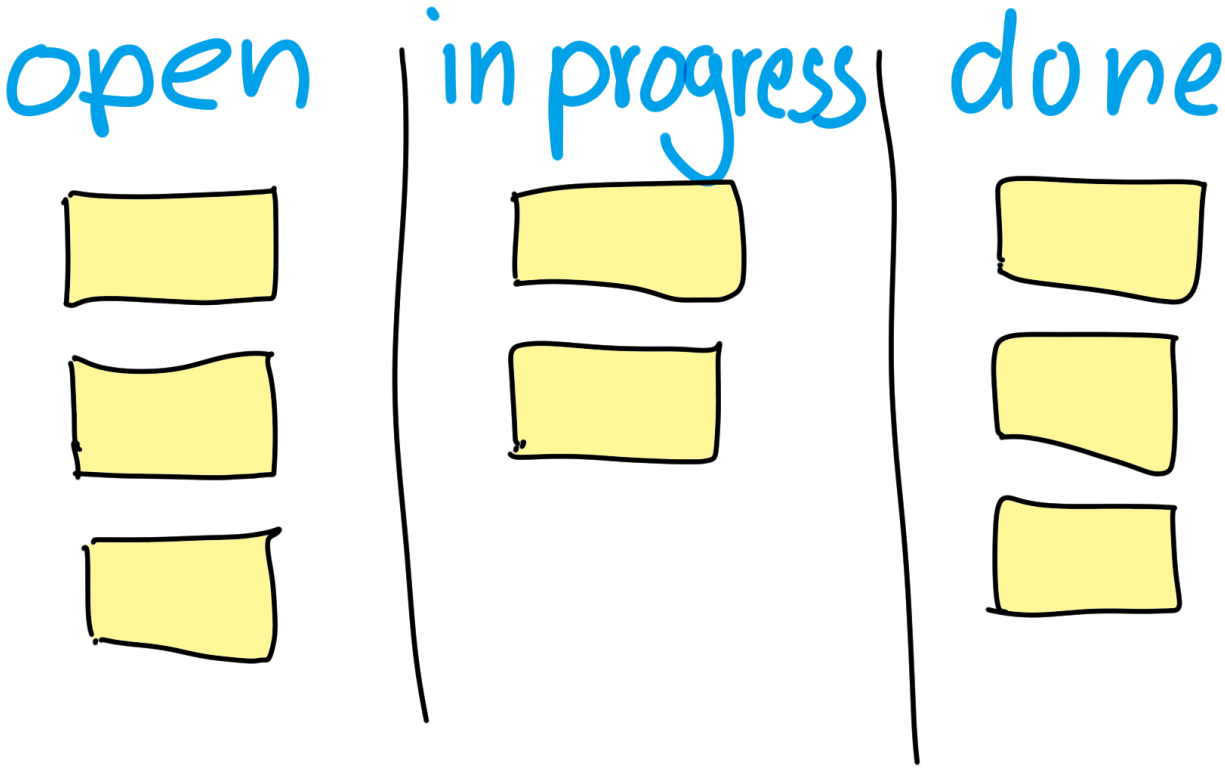


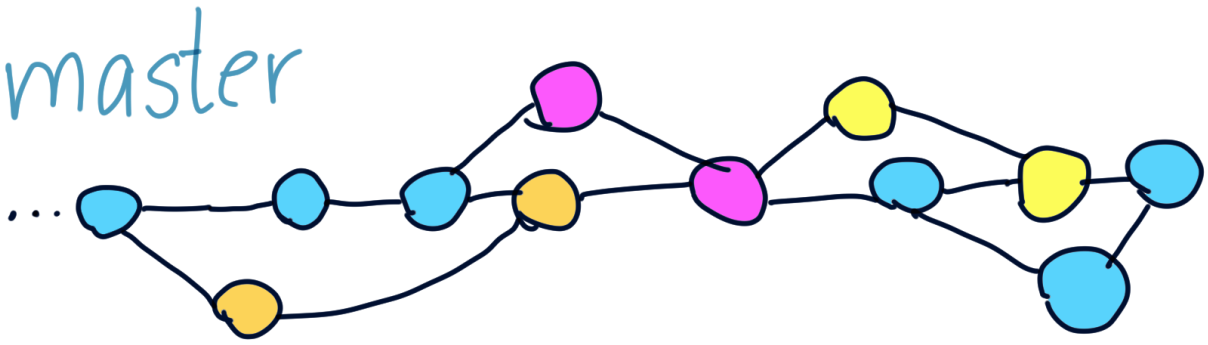
Der `master` ist die beste Version, die es jemals gab.

Jedes neue Commit macht ihn, ein kleines bisschen besser.

Alles unser Wissen und Können steckt darin.

https://trunkbaseddevelopment.com/


open
in progress
done


master
...

Was man kaputt macht,
muss man reparieren.
Und zwar SOFORT!

Bug-Fixing

- `git stash`,
- Bug fixen auf `master`
- `git commit`
- `git pull`, Testen, `git push`
- `git stash pop`, weiter machen

Und wenn ich sehen möchte,

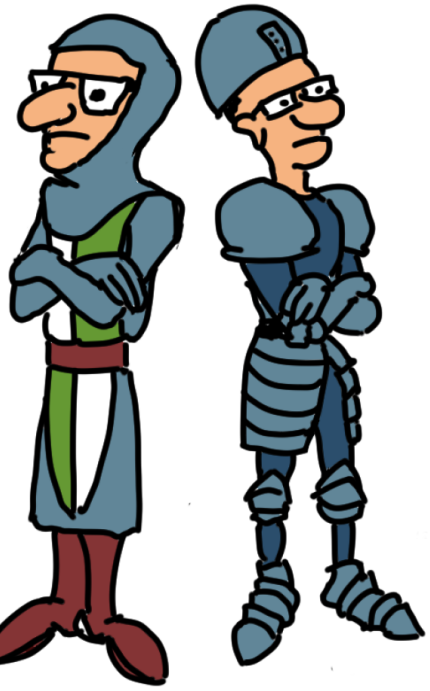
Was die anderen gemacht haben?

Ratet mal ...

```
git pull
```

So arbeiten wir mit Git

1. **Entwickler** ziehen einen Task auf **in Progress**
2. Implementieren (ggf. mit Feature-Toggles)
    - a. `git commit`
    - b. `git pull`
    - c. `git push`, sobald lokale Tests grün
    - d. Falls noch nicht fertig, weiter bei a.
3. Task auf **Done**, weiter bei **1.**

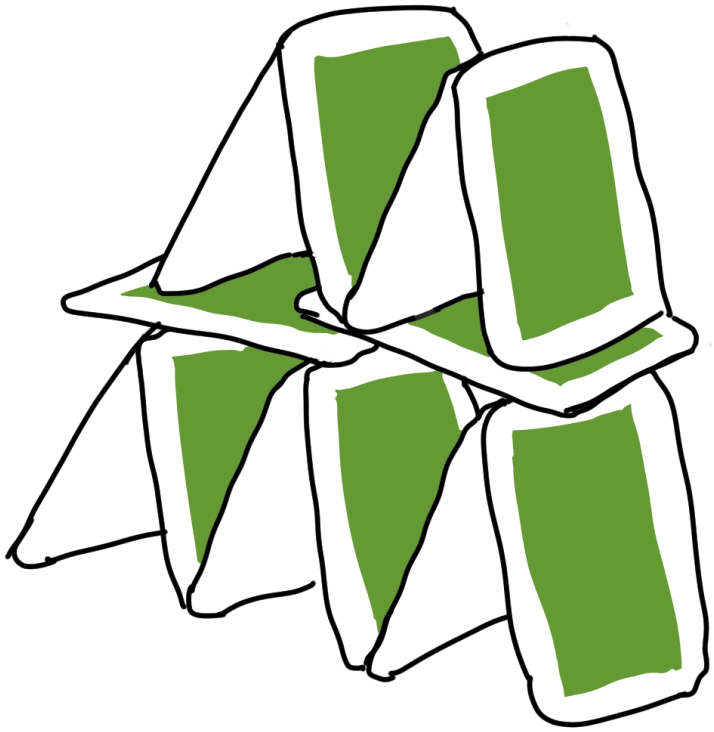


Ungestört arbeiten können

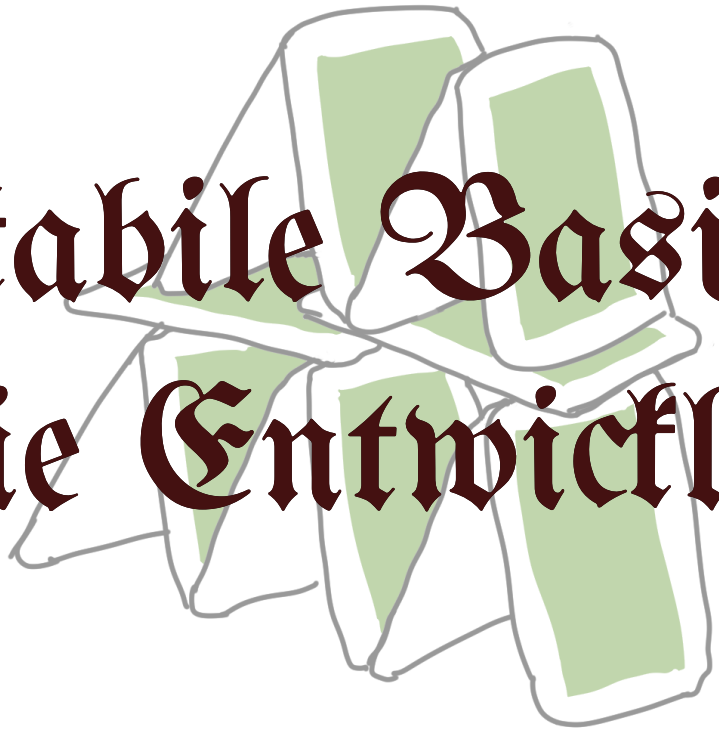
Fazit: Unabhängig arbeiten können.

- Git-technisch ist der Unterschied
  zwischen Feature-Branch und `master`-Klon gering.
- Bei TBD sind
  die Zeiten zwischen Merges meist kürzer(Stunden) als beim Feature-Branching (Tage)
- Kleine Merges: Kleine Probleme
- Große Merges: Große Probleme
- Flow: Entwickler sollten selber bestimmen können, wann sie integrieren.

Sowohl trunk-based als auch beim Feauture-Branching können Entwickler ihren Flow finden.



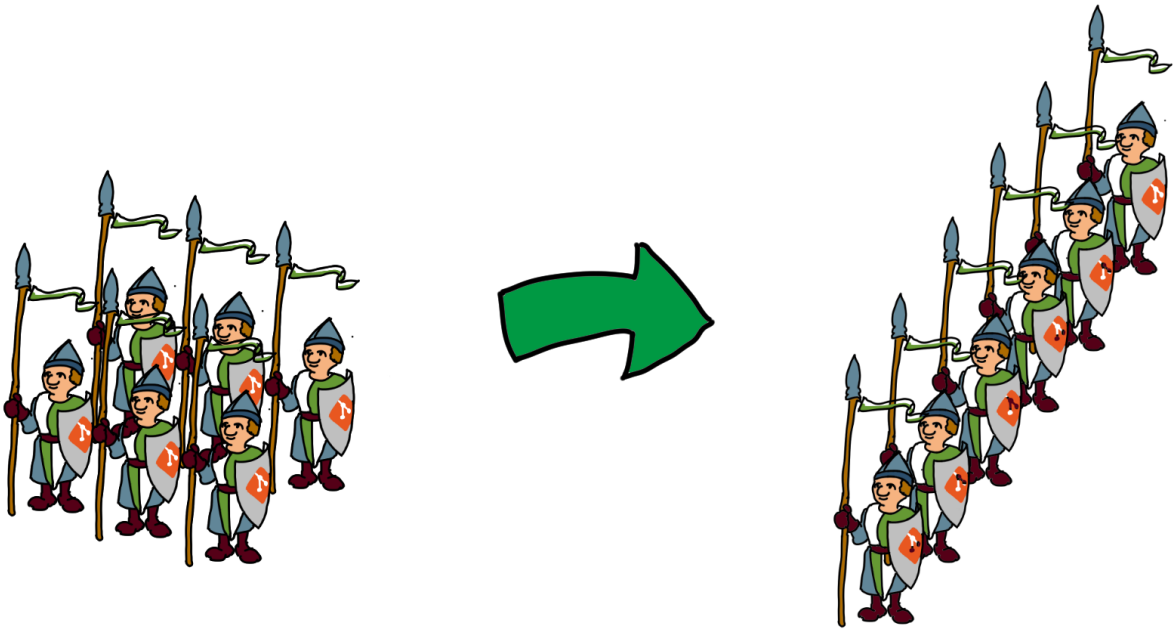
Stabile Basis für die Entwicklung



Fazit: Stabilität

- Ein brüchiger `master` (oder `develop`) nervt.
- Beim Feature-Branching sorgt man mit Reviews und Pre-Merge-Integrationstests
  für Stabilität.
- Beim Trunk-based-Developmen hingegen werden Probleme schneller erkannt und gefixed.
- In kleinen erfahrenen Teams
  wird der Integrationsbranch nur ab und zu mal *rot*.
- Vertrauen statt Kontrolle funktioniert.
- Lange Testsuiten sind ein Killer für TBD.

FB setzt mehr auf Prävention, TBD eher auf schnelle Problembehebung.

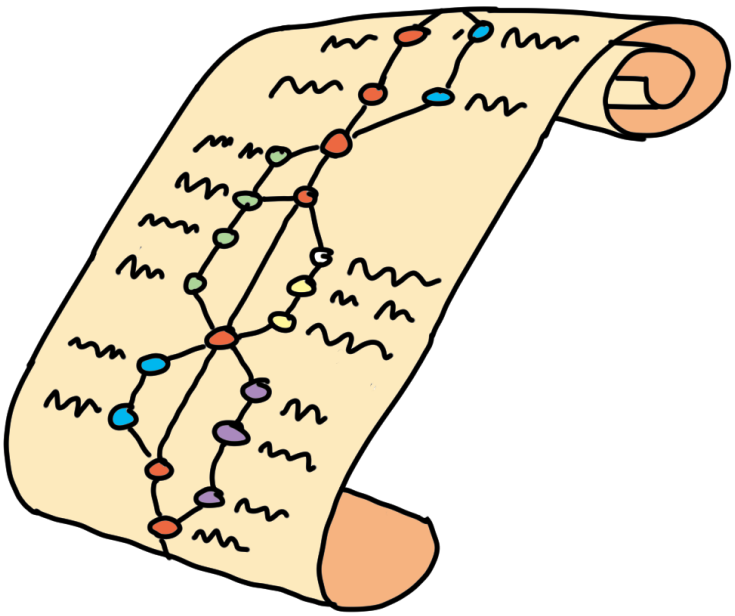


Refactoring

Fazit: Refactoring

- Refactorings betreffen oft viele Dateien
- Je länger die Branches, desto schwerer die Integration nach Refactorings
- Bei Feature-Branching hilfreich:
    - Koordination
    - Kurzlebige Refactoring-Branches
    - Update-Merge-Aufrufe nach Integration des Refactorings

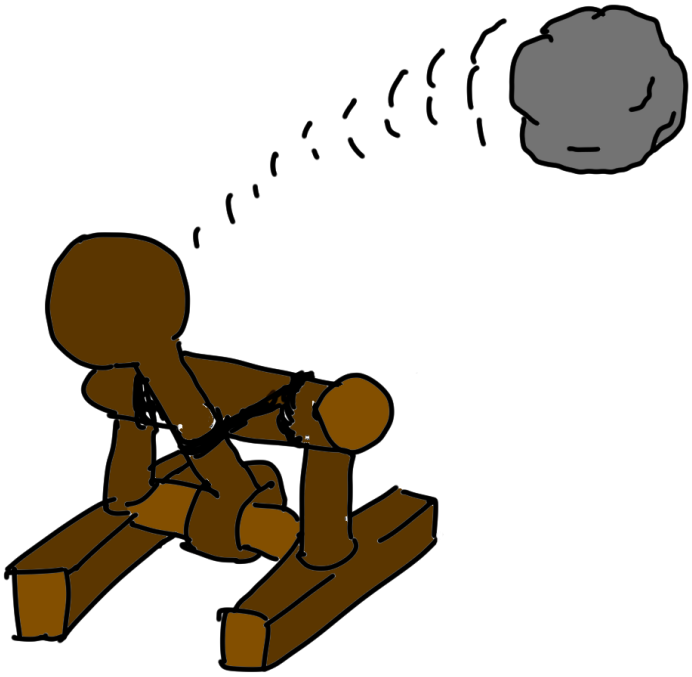
TBD erleichtertert Refactorings



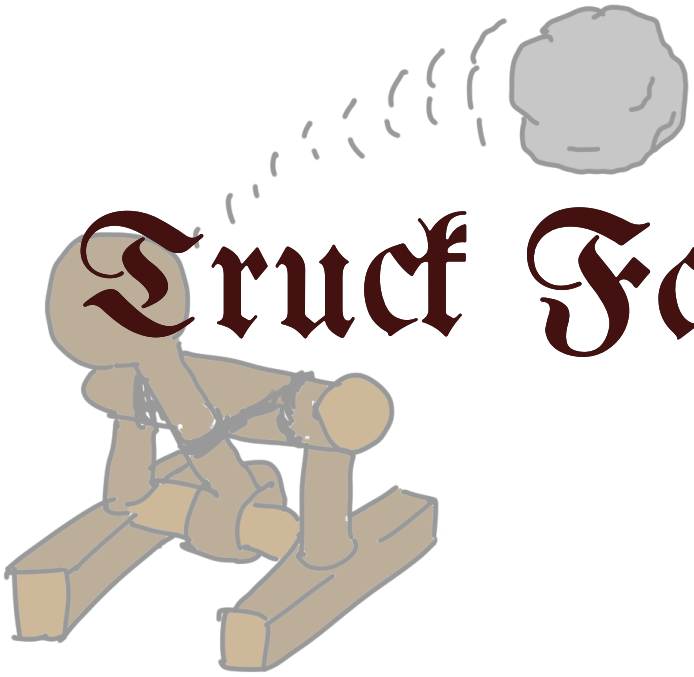
Schöne Historie

Fazit

- Die Historie für einzelne Dateien oder Zeilen ist notwendig für Merges und Fehlersuche.
- Schön muss sie dazu aber nicht sein
- Bei Feature-Branching kann man die Historie nutzen, um die Integration der Features darzustellen.



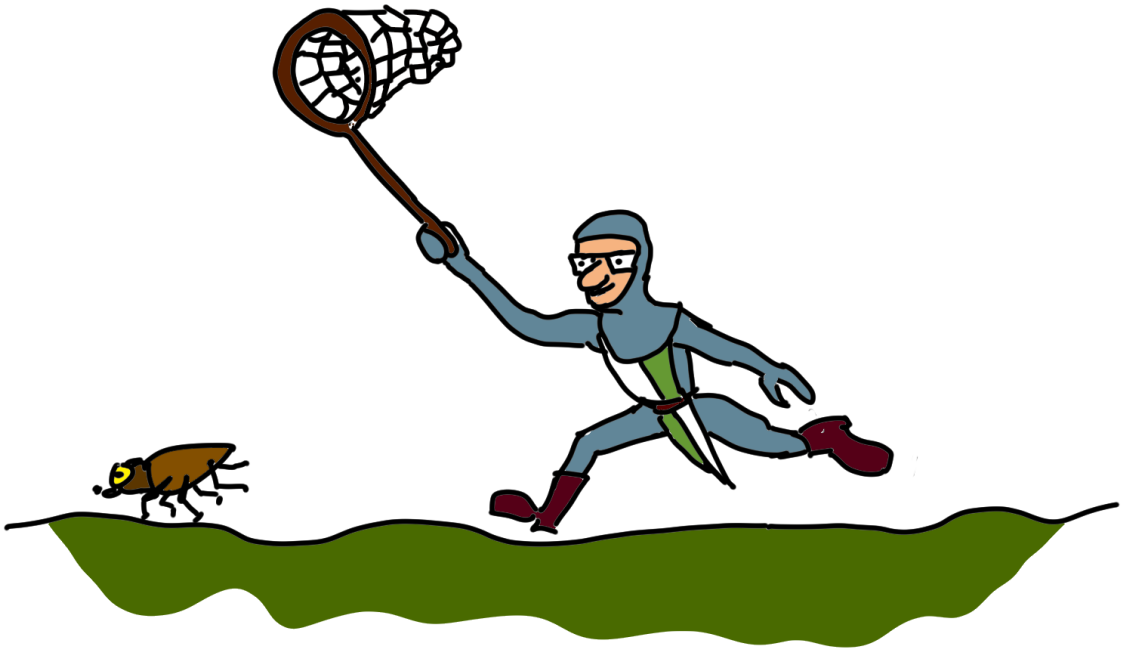
Truck Faktor



Fazit

- Pair Programming:
- Fachlicher Know How Transfer
- Review-Charakter
- Aber kein dokumentiertes Reviewergebnis
- Code Style etc. soweit wie möglich automatisieren

Pair Programming verringert den Truck-Faktor.

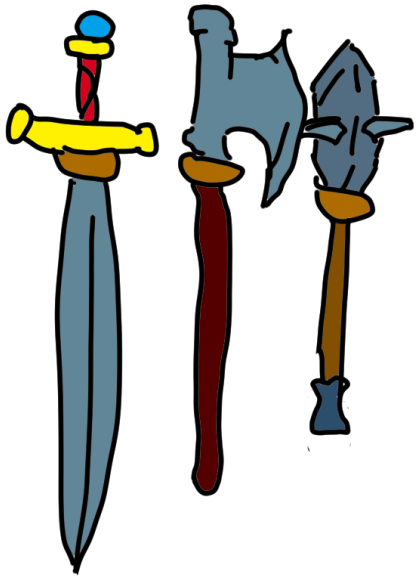


Bug Fixing

Fazit

- Git-Flow ermöglicht sehr zielgenaue Bugfixes
  - Erfordert aber viel Know-How, Arbeit und Disziplin.
- Bei Trunk-Based wird "Forward-Fixing" gemacht
  - Erfordert immer produktionsreifen 'master'
  - Released immer alles
  - Neue Features können durch Toggles deaktivert sein

Feature-Branching ermöglicht zielgenaues Bugfixing. Pragmatisch, ist "Forward-Fixing" oft effizienter.

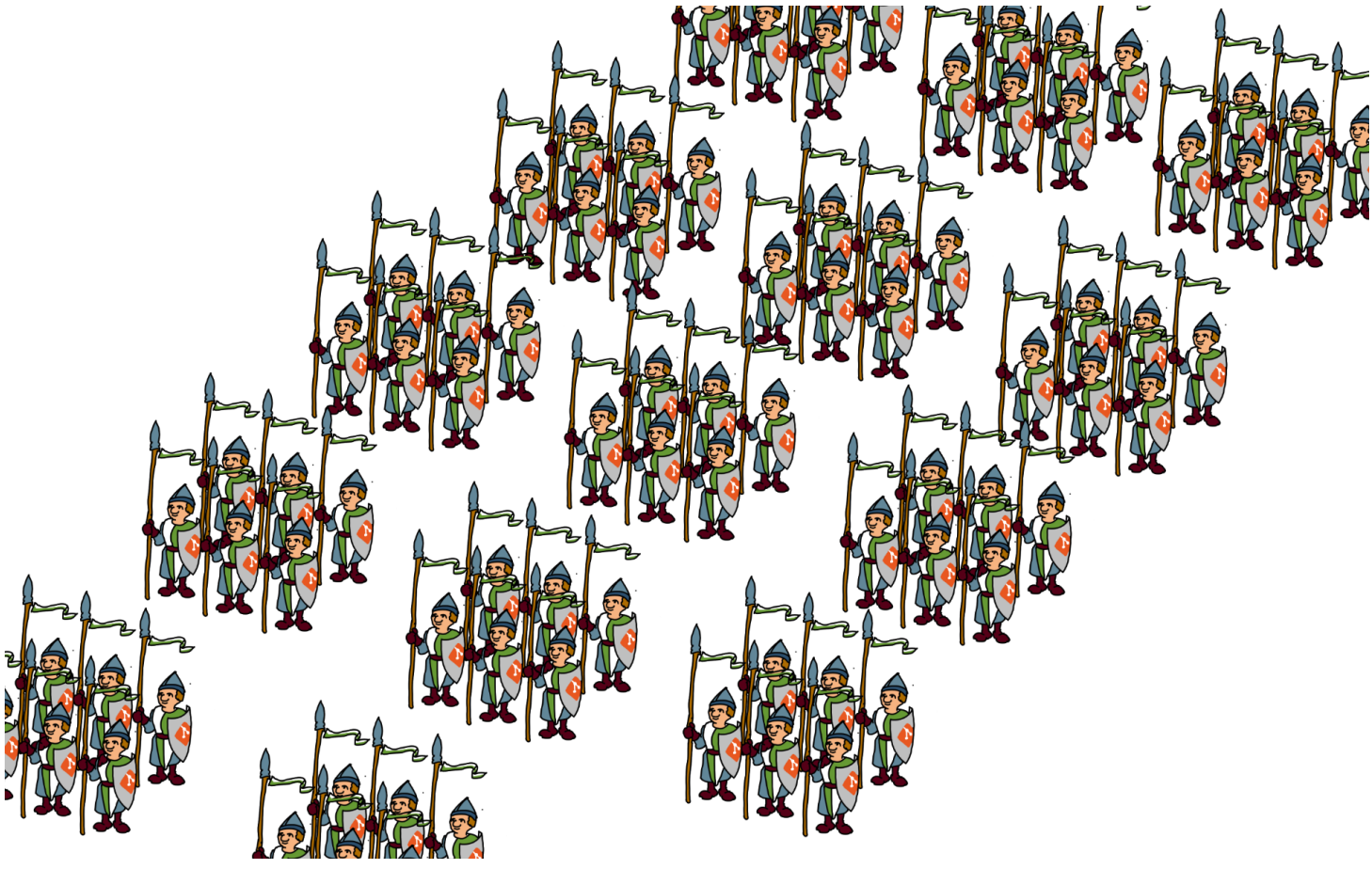


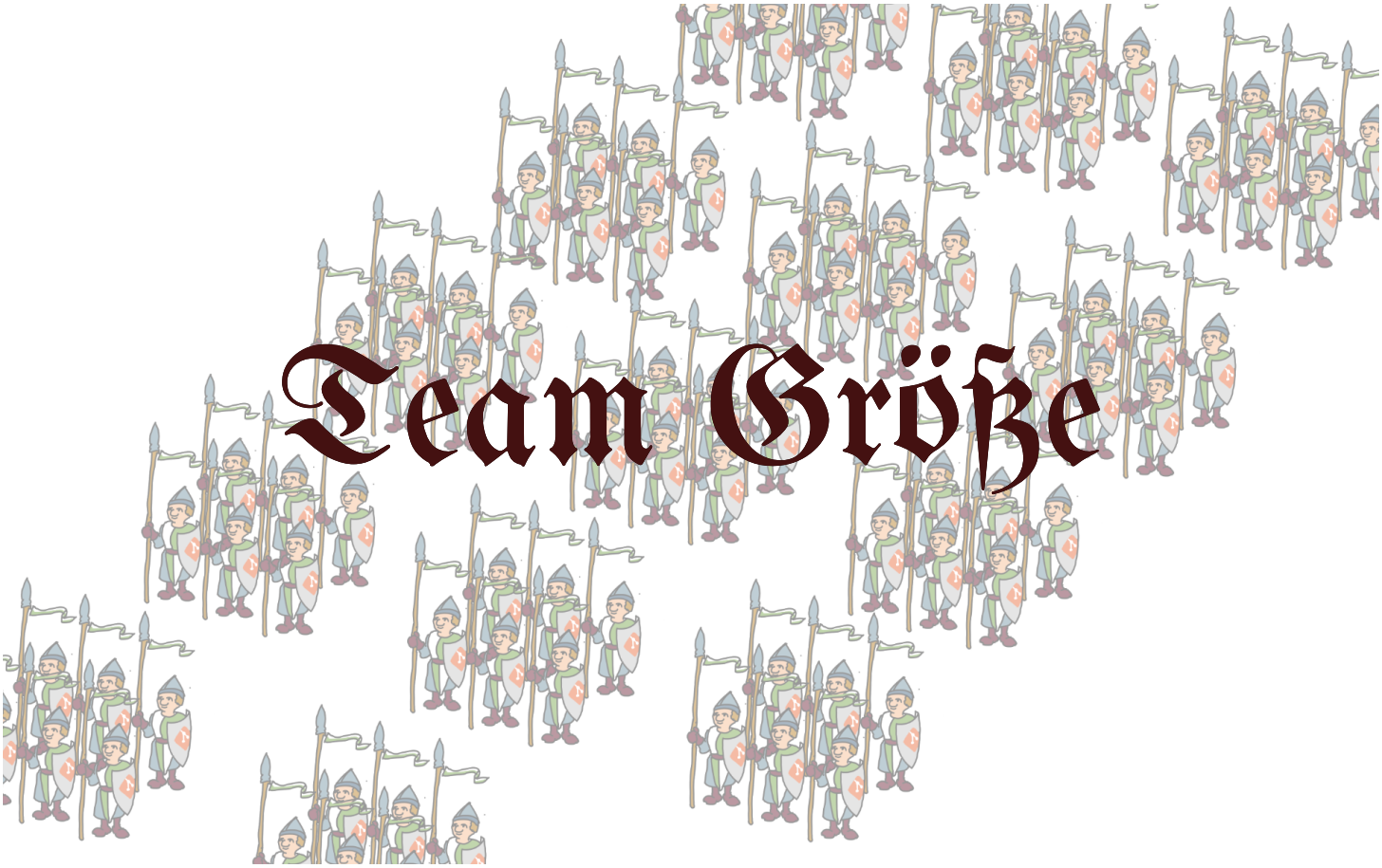
Feature Picking

Fazit

- Feature-Branches geben direktere Kontrolle über die Integration
- Integration kurz vor Schluss ist nicht empfehlenswert.
- Zwischen Feature-Branches sollte nicht gemerged werden.
- Das Teilen von Zwischenergebnissen ist umständlich.
- In TBD akzeptiert man, dass der Code von unvollständigen Features ausgeliefert wird (weggetoggled natürlich).
- Ggf. geben Feature-Toggles dem PO die Kontrolle über Features.

Ein echtes wahlfreies Feature-Picking funktioniert auch beim Feature-Branching nicht wirklich gut.





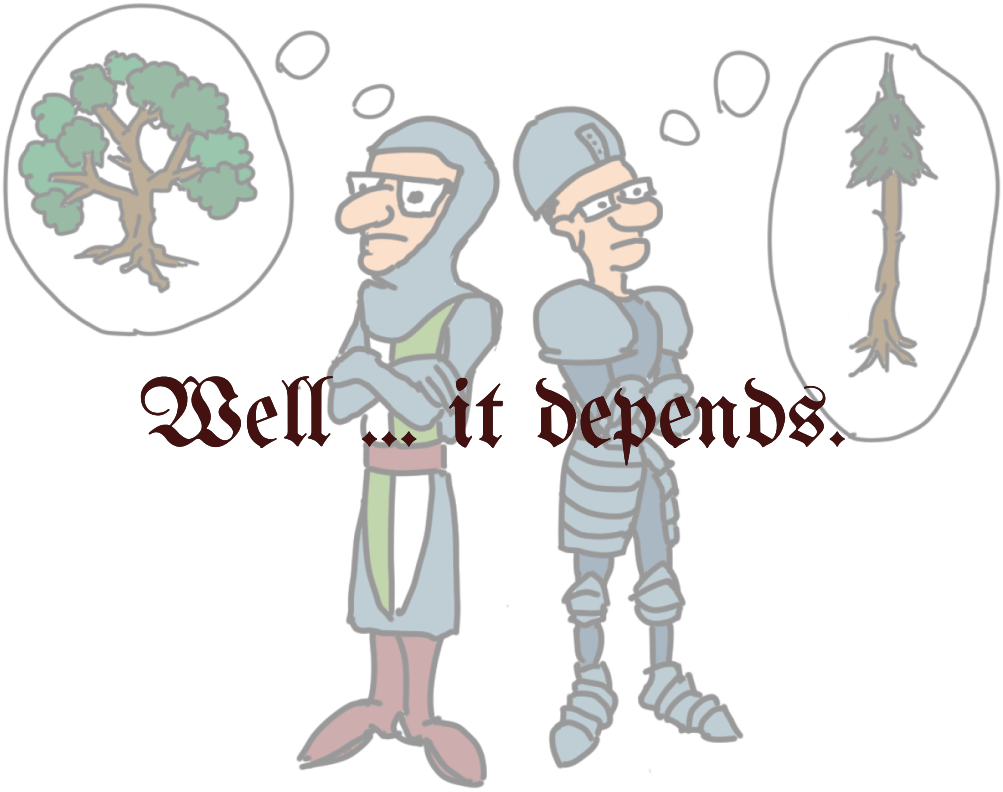
Team Größe

Fazit

Bei vielen Entwicklern in mehreren Teams auf dem selben Repo, Monolithen und Open-Source-Projekten ist der Overhead für Feature-Branching oft gerechtfertigt.

- **Ungestörtes Arbeiten**: Sowohl trunk-based als auch beim Feature-Branching können Entwickler ihren Flow finden.
- **Stabile Basis**: Feature-Branching setzt mehr auf Prävention, Trunk-based-Development eher auf schnelle Problembehebung.
- **Refactorings**: Trunk-based-Developmemnt erleichtert Refactorings.
- **Schöne Historie**: Bei Feature-Branching kann man die Historie nutzen, um die Integration der Features darzustellen.

- **Truck-Faktor**: Pair Programming verringert den Truck-Faktor.
- **Bug-Fixing**: Feature-Branching ermöglicht zielgenaues Bugfixing. Pragmatisch, ist "Forward-Fixing" oft effizienter.
- **Feature-Picking**: Ein echtes wahlfreies Feature-Picking funktioniert auch in Feature-Branching nicht wirklich gut.
- **Teamgröße**: Bei vielen Entwicklern in mehreren Teams auf dem selben Repo, Monolithen und Open-Source-Projekten ist der Overhead für Feature-Branching oft gerechtfertigt.



Sie finden diese Präsentation zum Nachlesen in unserem Blog: Kapitel 26.

```
http://kapitel26.github.io

https://kapitel26.github.io/slides/2019-10-27-to-branch-or-not-to-branch/
```

Drucken / Mit Notizen drucken

René

selbständig ... Softwarearchitekt

... eToSquare ... Entwickler

... Trainer ... Autor

Bjørn

... Senior Expert ... Big Data ...

... Otto... find' ich gut. ... Autor ....