TÓPICOS AVANÇADOS DE ENGENHARIA DE SOFTWARE

ADVANCED TOPICS IN SOFTWARE ENGINEERING

Catarina Reis
catarina.reis@ipleiria.pt
reiscatarina
@catarina_reis

Engenharia Informática - 3º ano - 1º semestre
Departamento de Engenharia Informática
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Leiria

EXTREME PROGRAMMING

XP

PEOPLE AND HISTORY

Kent Beck

- 1996 - Project Leader @ Chrysler Comprehensive Compensation System (C3)
- 1997 - JUnit
- "eXtreme Programming Explained: Embrace Change" (1999, 2004)

  Kent Beck and Cynthia Andres. 2004. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.

- 2016 - Technical Coach @ Facebook

GOAL

The goal of Extreme Programming (XP) is **outstanding software development.**

Software can be developed at **lower cost**, with **fewer defects**, with **higher productivity**, and with much **higher return on investment**.

Kent Beck

NO MATTER THE CIRCUMSTANCE YOU CAN ALWAYS IMPROVE.

YOU CAN ALWAYS START IMPROVING WITH YOURSELF.

YOU CAN ALWAYS START IMPROVING TODAY.

Kent Beck

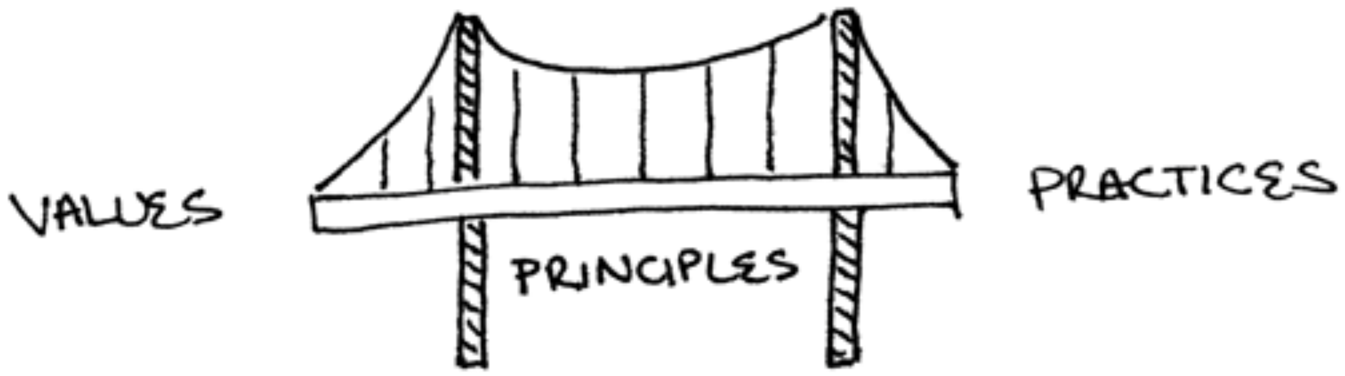
WHAT IS XP?

- XP is lightweight
- XP is a methodology based on addressing constraints in software development, not in any other areas
- XP can work with teams of any size
- XP adapts to vague or rapidly changing requirements
- XP addresses risks at all levels of the development process

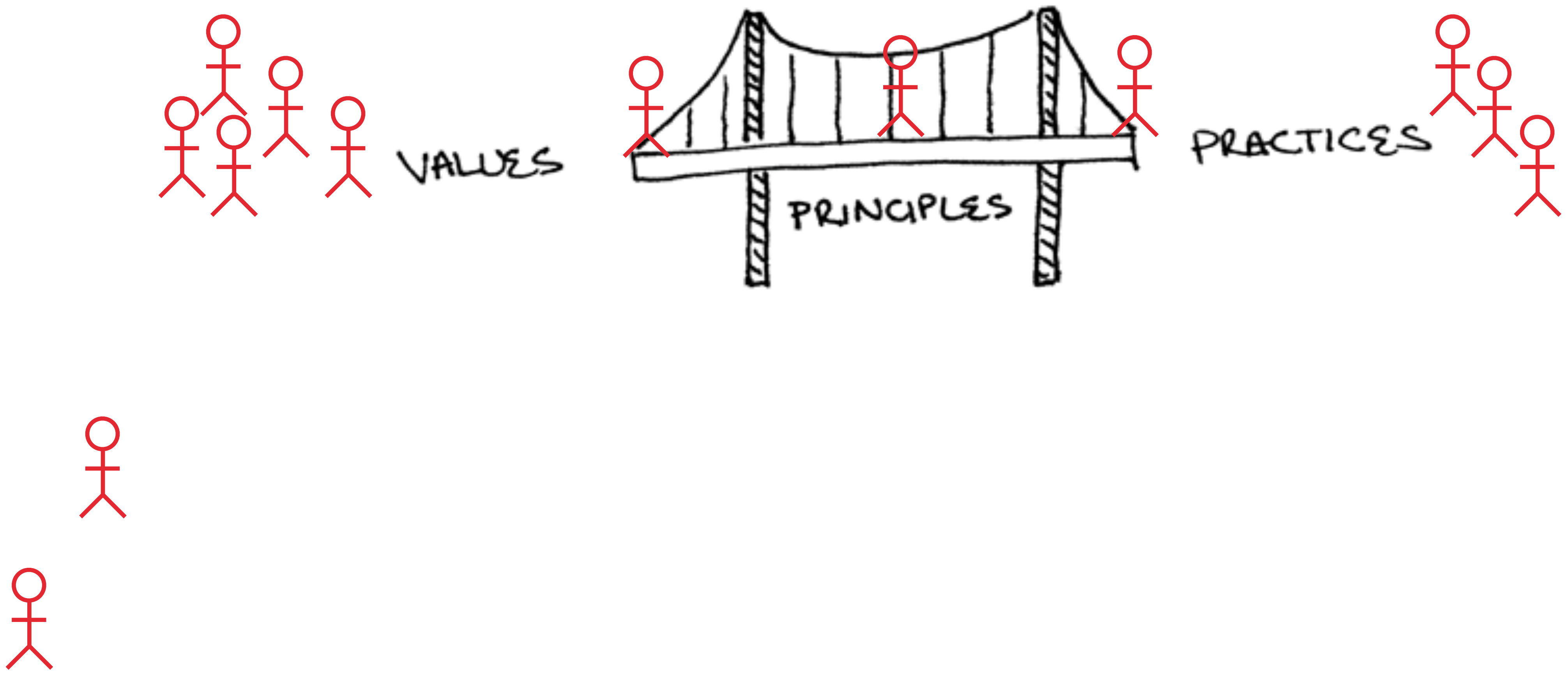
- XP is giving up old, ineffective technical and social habits
- XP is fully appreciating yourself for your total effort today
- XP is striving to do better tomorrow
- XP is evaluating yourself by your contribution to the team's shared goals
- XP is asking to get some of your human needs met through software development

XP INCLUDES

- philosophy of software development based on **values**
- body of **practices** proven useful in improving software development
- set of complimentary **principles**, intellectual techniques for translating values into practices
- **community** that shares values and many of the practices


VALUES
PRINCIPLES
PRACTICES

community



VALUES

COMMUNICATION

SIMPLICITY

Feedback

COURAGE

RESPECT.

PRINCIPLES

PEOPLE DEVELOP SOFTWARE

HUMANITY

- what do people need to be good developers?
  - basic needs
  - accomplishment
  - belonging
  - growth
  - intimacy
- balance between the individual needs and the needs of the team

SOMEBODY HAS TO PAY FOR ALL THIS

ECONOMICS

- time value of money
  - a dollar today is worth more than a dollar tomorrow
  - incremental design defers design investment to the last responsible minute
  - pay-per-use
- option value of systems and teams
  - redeploy the system for several contexts
  - reuse the skills of the team for several projects
  - caution: do not invest in speculative flexibility

EVERY ACTIVITY SHOULD BENEFIT ALL CONCERNED

MUTUAL BENEFIT

- extensive internal documentation of software >> violates mutual benefit
    - write automated tests
    - refactor
    - choose names from a coherent and explicit set of metaphors
- **benefit for:** me, now! me, later! and, my customer as well!

WHEN NATURE FINDS A SHAPE THAT WORKS, SHE USES IT EVERYWHERE SHE CAN

SELF-SIMILARITY

try copying the structure of one solution into a new context, even at different scales

it doesn't mean it will work!

but it's a good place to start!

IN SOFTWARE DEVELOPMENT “PERFECT” IS A VERB NOT AN ADJECTIVE

IMPROVEMENT

- "Good enough"?

  Do the best you can today! But, don't wait for perfection in order to begin!

- Strive daily to bring perfection closer to reality

  Find a starting place, get started and improve from there

TEAMS NEED DIVERSITY

DIVERSITY

- conflict is the inevitable companion
- two different and divergent ideas about a design, present an opportunity
  - programmers should work together and both opinions should be valued
- goal: create the most valuable software possible in the time available

NO ONE STUMBLES INTO EXCELLENCE

REFLECTION

- cycles include time for reflection
- good teams do their work
- **excellent** teams **think**
    - how they are working
    - why they are working

*(**besides** doing their work)*

- and **act accordingly**

THE DAILY BUILD, FOR EXAMPLE, IS FLOW-ORIENTED

FLOW

- steady flow of valuable software
- “big bang” integration –> large chunks of value –> less feedback
- get back to weekly deployment as soon as possible (ASAP)

LEARN TO SEE PROBLEMS AS OPPORTUNITIES

OPPORTUNITY

- what to do? need more time…
- patience can solve a problem by itself…
- turn each problem into an opportunity:
  - to grow and learn
  - and improve

YES, REDUNDANCY

REDUNDANCY

- one solution fails?!?… try another… and another… and another…
- one solution will only evidence the output of another solution?!? …
  - having a final testing phase is redundant?!?
  - only eliminate it after is has shown to be redundant

IF YOU'RE HAVING TROUBLE SUCCEEDING, FAIL

FAILURE

- failure == waste?
- fail instead of talk
- when you don’t know what to do, risking failure can be the shortest and surest road to success

QUALITY ISN'T A PURELY ECONOMIC FACTOR. PEOPLE NEED TO DO WORK THEY ARE PROUD OF

QUALITY

- quality is not a control variable
- projects don't go faster if quality decreases
- projects don't go slower if quality increases
- quality can be measured in defects, design quality and the experience of development
- leads improvements in productivity and effectiveness

CHANGE IS UNSETTLING

BABY STEPS

- big changes = big steps = big fall
- baby steps

overhead of small steps is smaller than the big steps and provides considerable leaps

RESPONSIBILITY CANNOT BE ASSIGNED; IT CAN ONLY BE ACCEPTED

RESPONSIBILITY

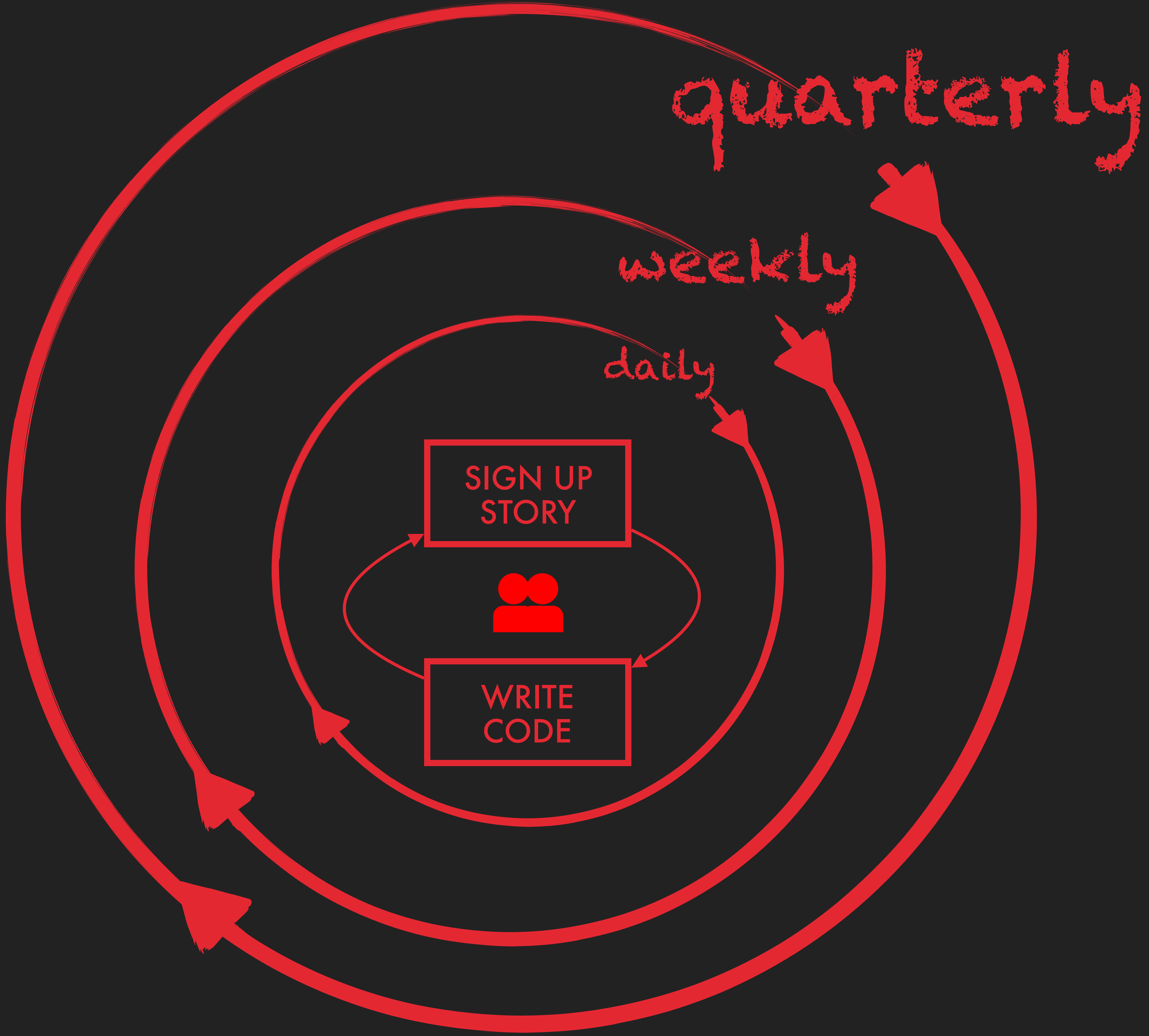
- whoever signs up to do the work
  - estimates it
  - designs it
  - implements it
  - and tests it

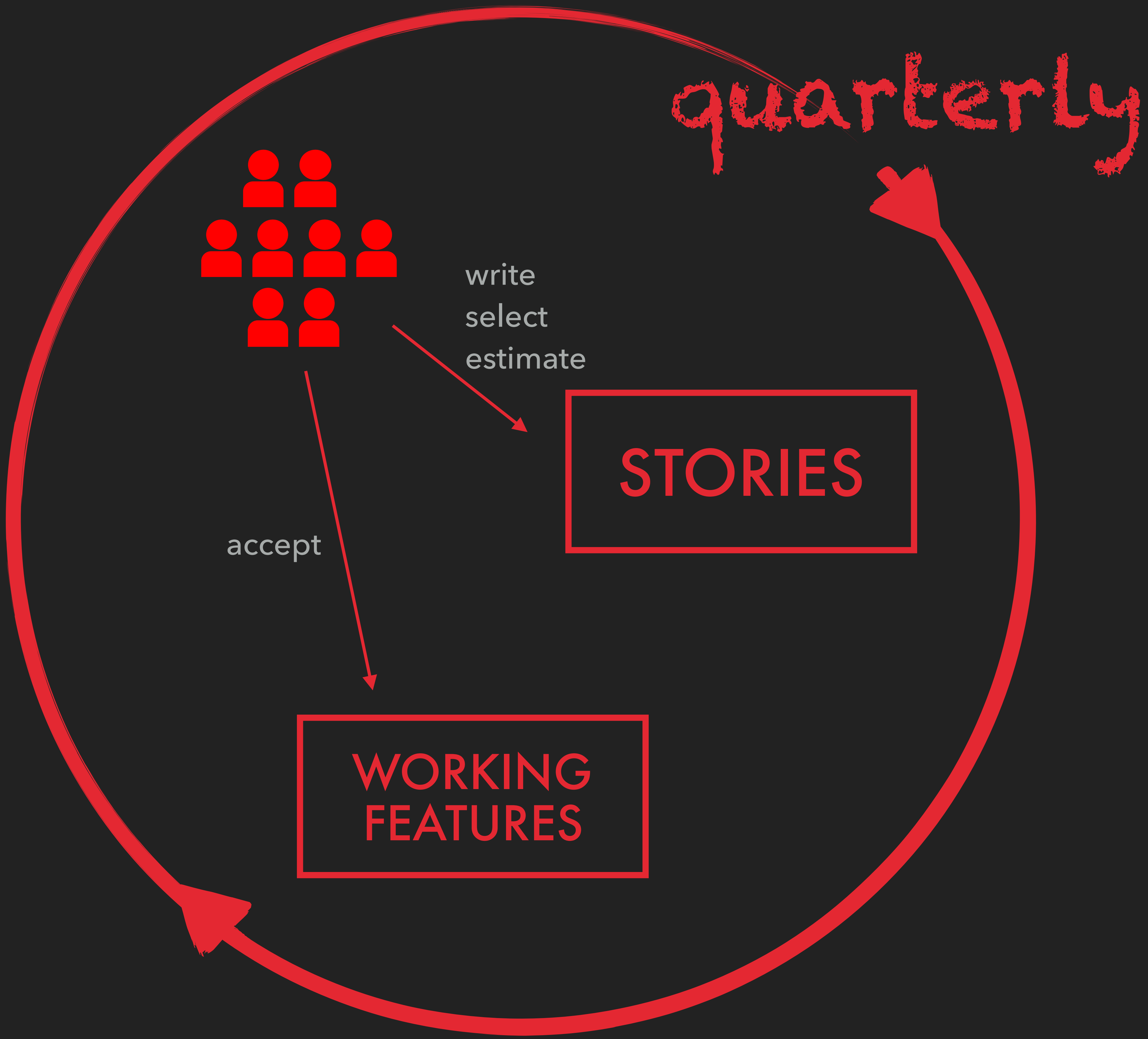
PRACTICES

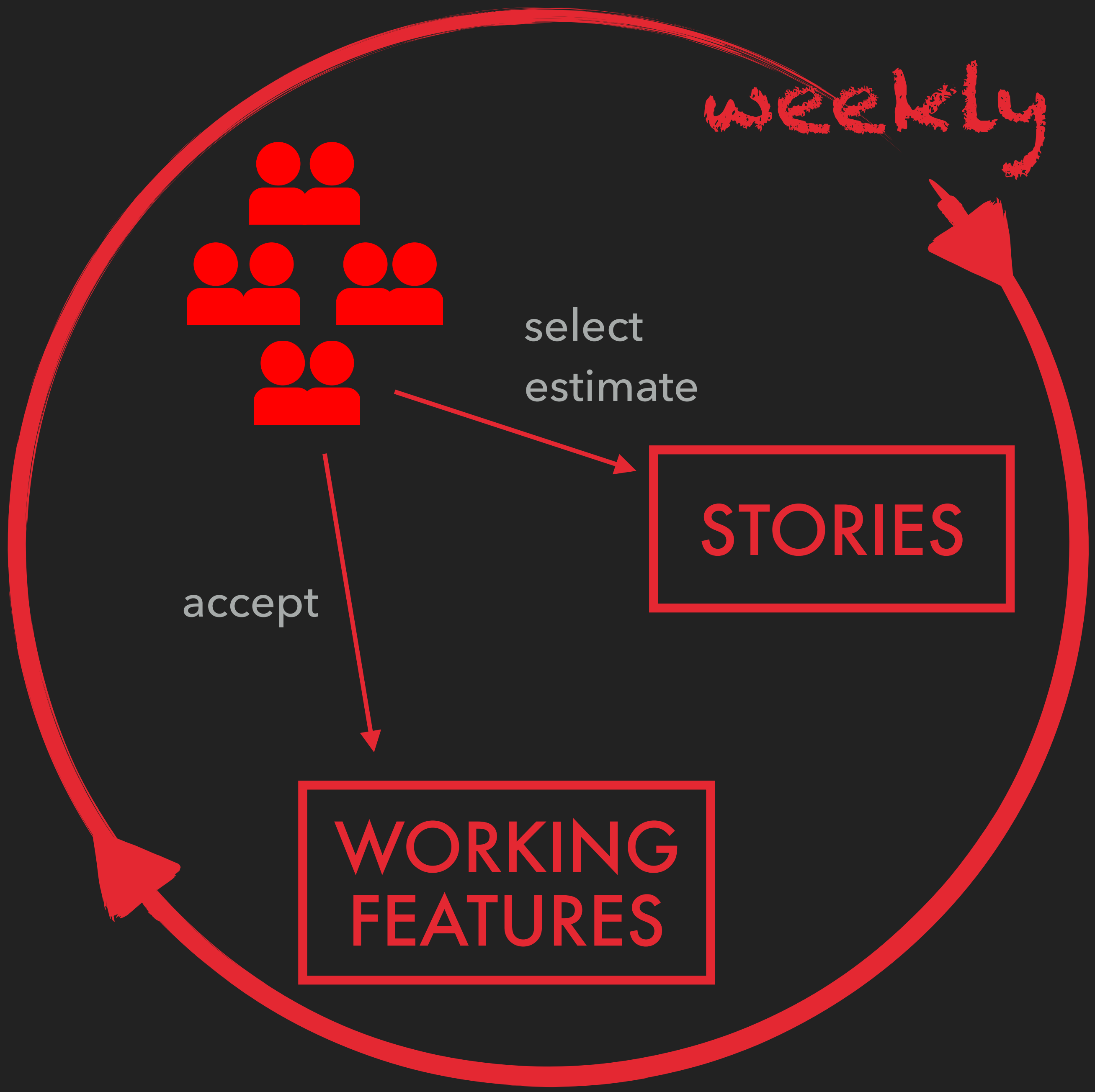
PRACTICES

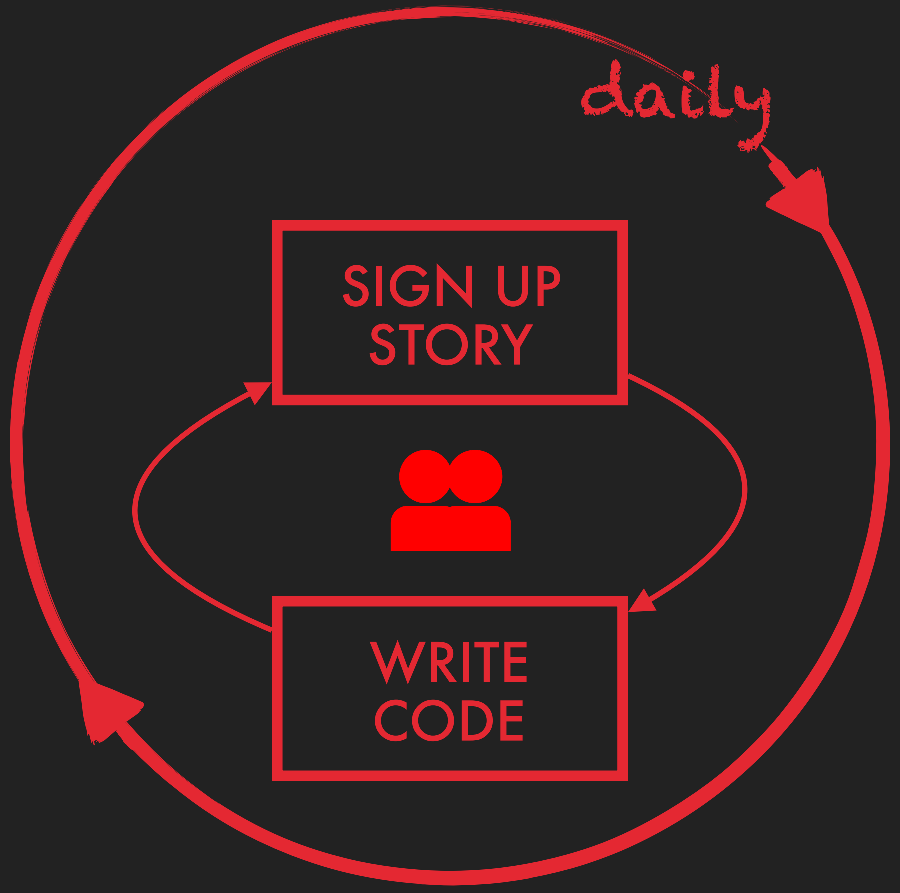
- are context dependent
- stated as absolutes
- a vector from where you are to where you can be with XP
- applying a practice is a choice
- tend to work well together

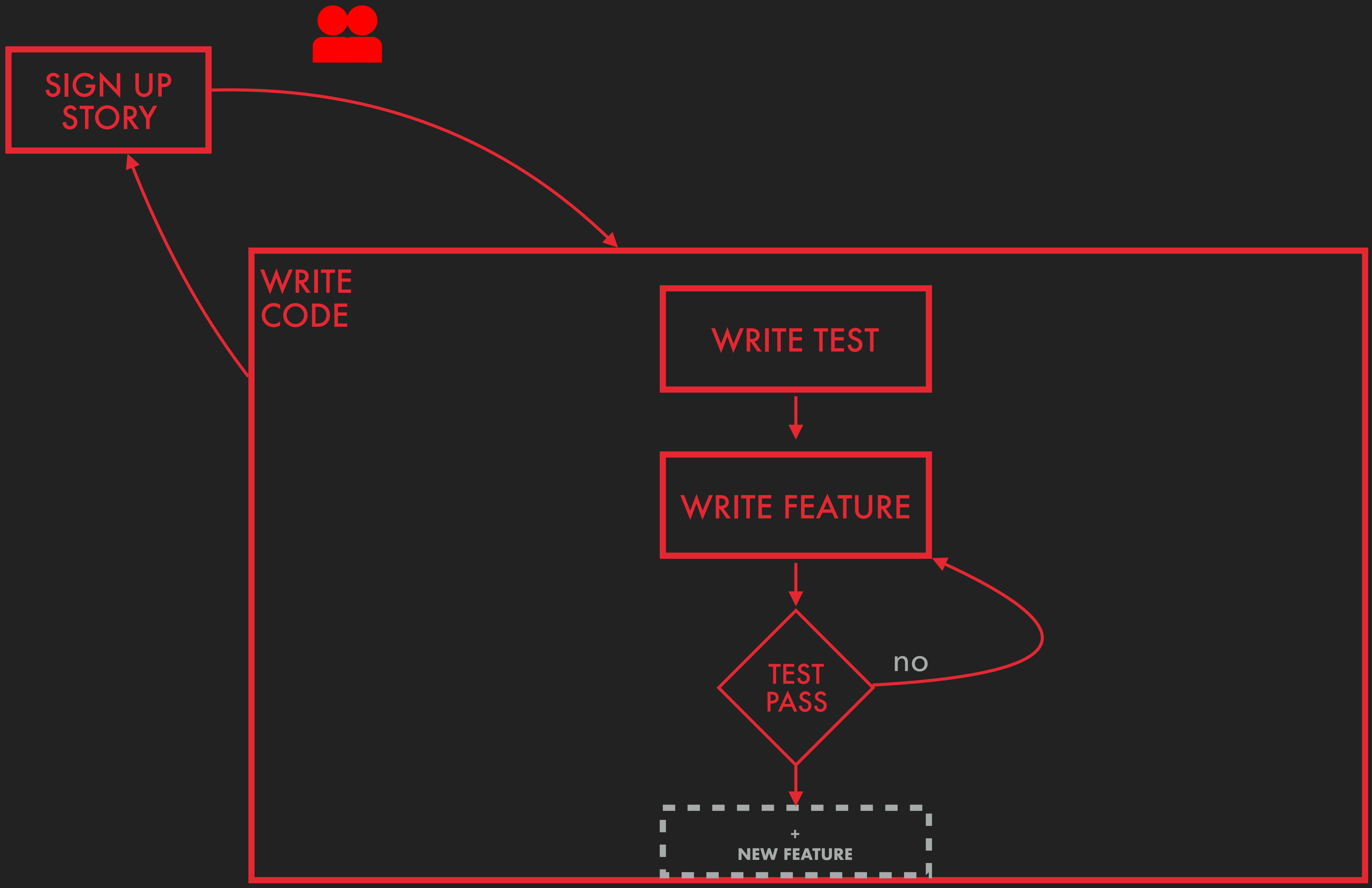
apply 1 and see improvement, apply a compound and see dramatic improvement

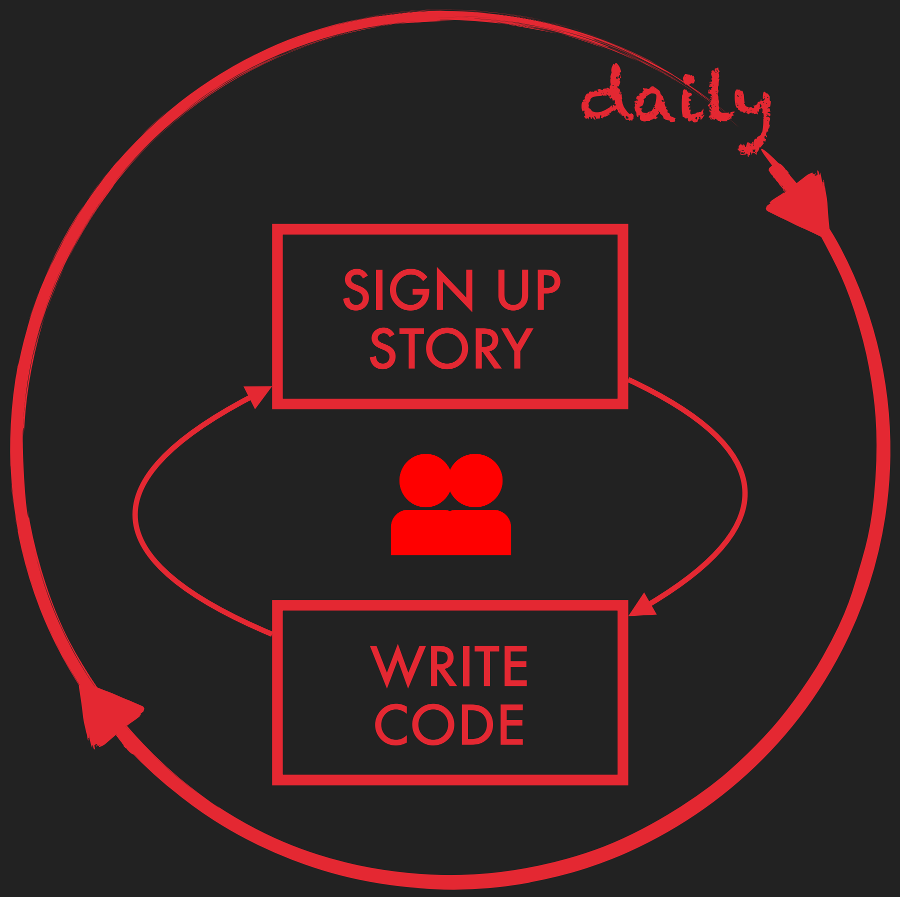

quarterly
weekly
daily
SIGN UP STORY
WRITE CODE

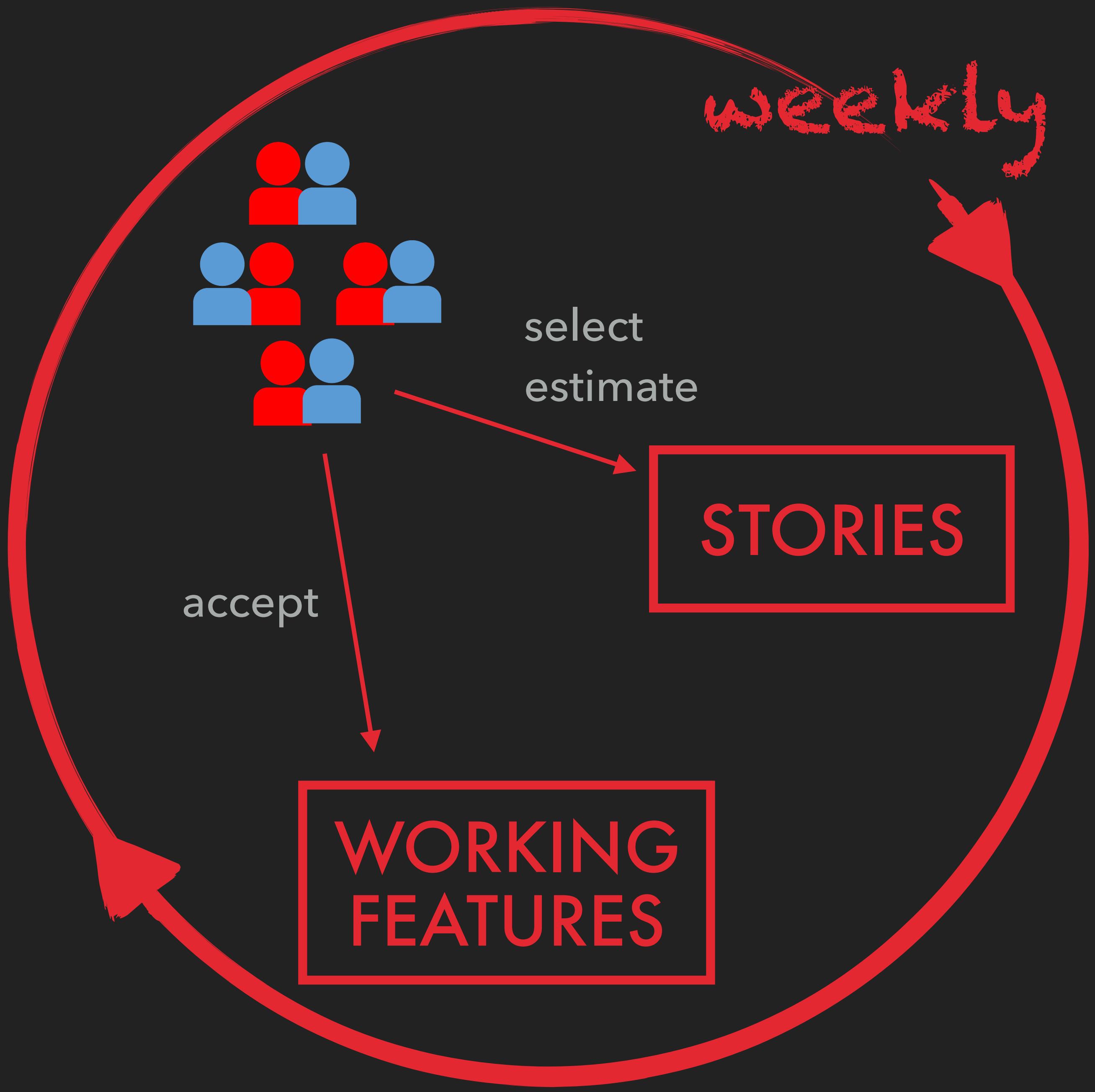

quarterly
write
select
estimate
STORIES
accept
WORKING FEATURES

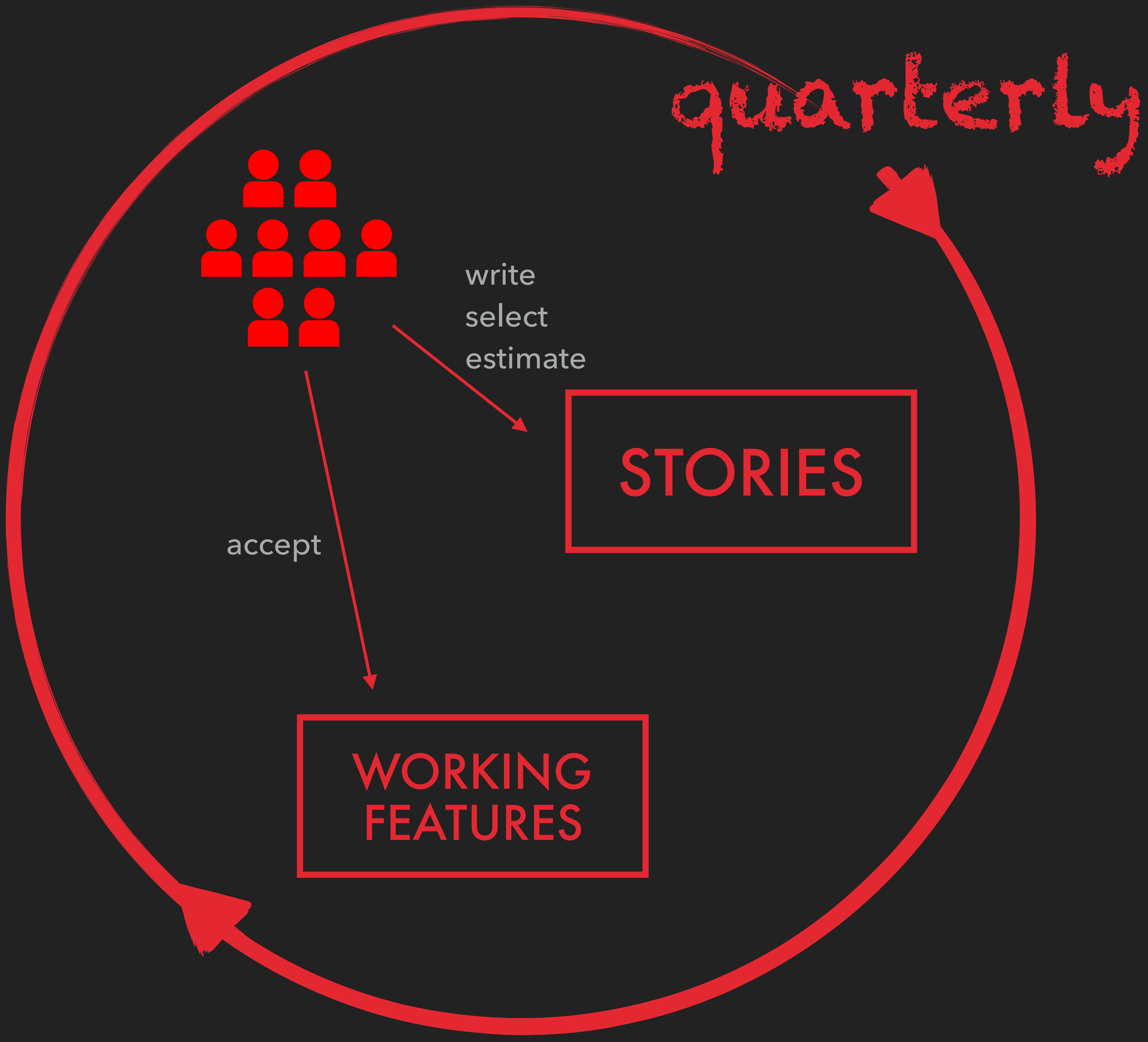

weekly
select
estimate
STORIES
accept
WORKING FEATURES

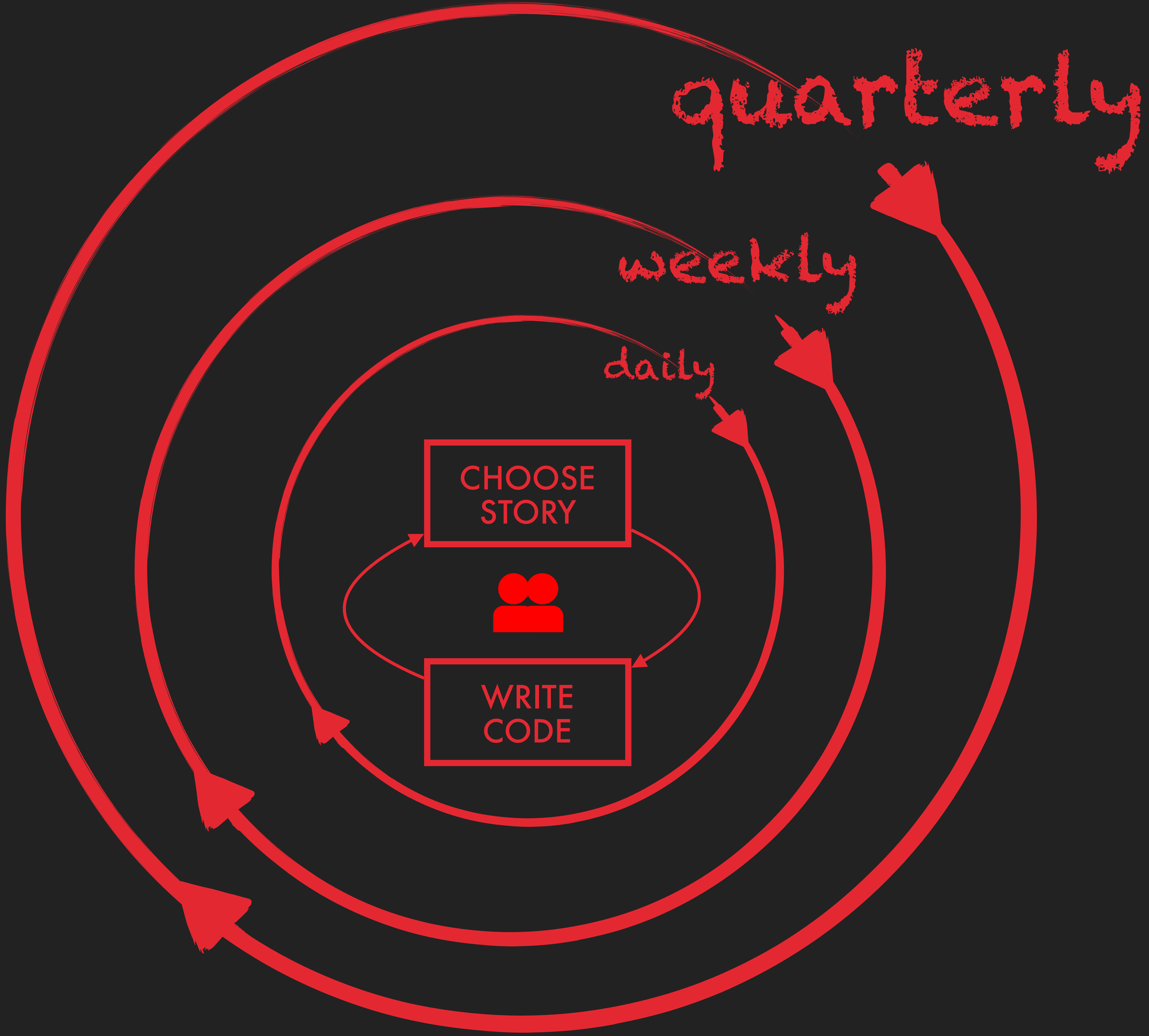

daily
SIGN UP STORY
WRITE CODE


SIGN UP STORY
WRITE CODE
WRITE TEST
WRITE FEATURE
TEST PASS
no
+
NEW FEATURE


daily
SIGN UP STORY
WRITE CODE


weekly
select
estimate
STORIES
accept
WORKING FEATURES


quarterly
write
select
estimate
STORIES
accept
WORKING FEATURES


quarterly
weekly
daily
CHOOSE STORY
WRITE CODE

PRIMARY PRACTICES

SIT TOGETHER

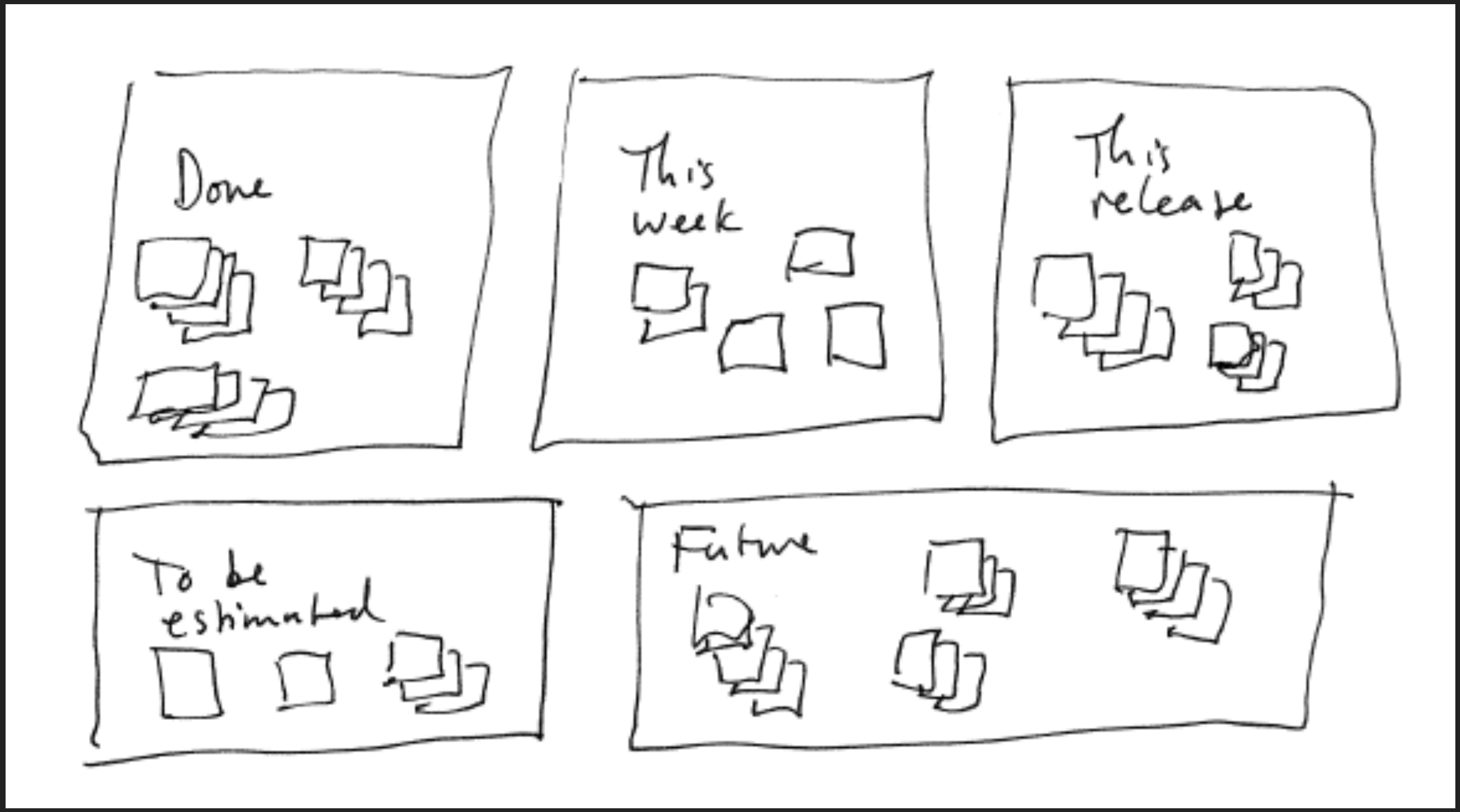
- develop in an open space big enough for the whole team
- communicate with all your senses
  - put a comfy chair in your cubicle
  - conference room - one week trial
  - software engineering labs
- what about multisite teams?
  - the more face time you have, the more productive and humane the project
  - distributed teams need to assess if everything is going well
    - if not, rethink and act >> more travel

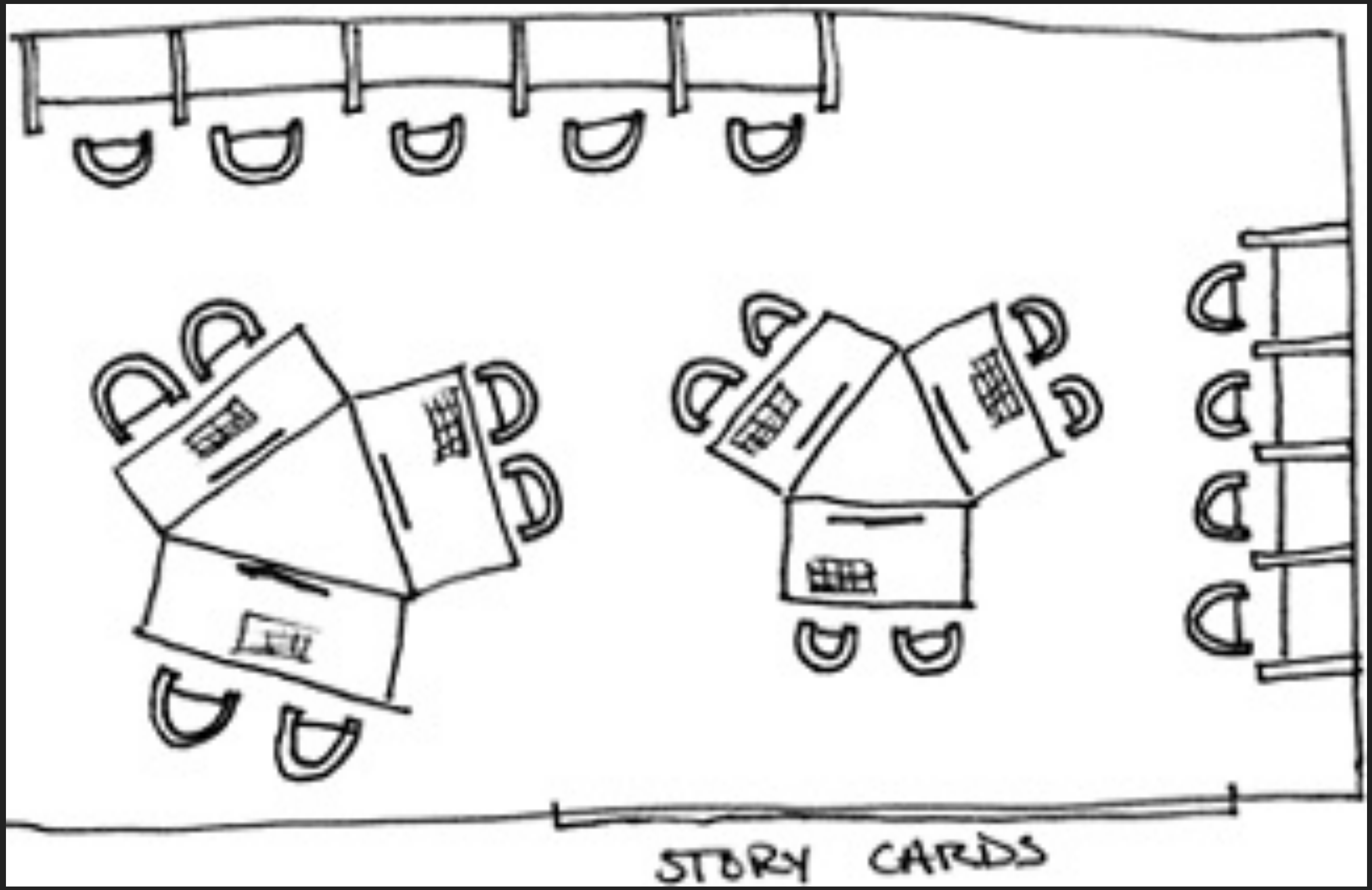
WHOLE TEAM

- cross functional teams
  - people with all the skills and perspectives required for success
- sense of "team"
  - we belong
  - we are in this together
  - we support each other's work, growth and learning
- team size
  - 12 - 150
  - 40% - 60%

INFORMATIVE WORKSPACE

- workspace == about your work
- someone should enter the team space and get the general idea in 15 seconds
- stories on a wall
- water and snacks provide comfort and encourage social interactions
- big visible charts (important and active information that is useful being visible)


Done
This week
This release
To be estimated
Future


STORY CARDS

ENERGIZED WORK

- work only the hours you can be productive and as many as you can sustain
- with enough caffeine and sugar developers keep typing… long past the point where they started removing value from the project
- when sick >> get better 1st
- make incremental improvements

2hour stretch == Code Time

turn off phones and email notifications and simply code for 2 hours

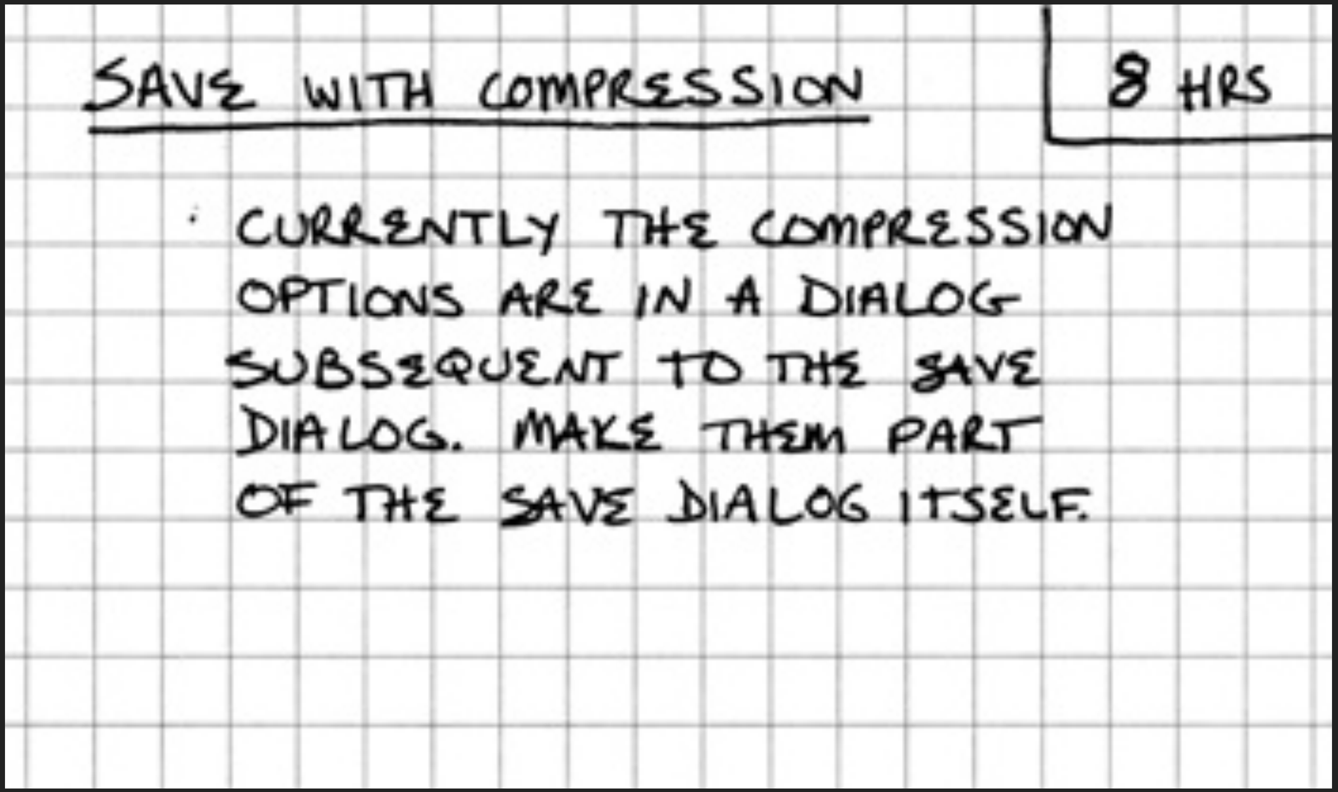
PAIR PROGRAMMING

- write all code with 2 people sitting at 1 machine
- dialog between 2 people simultaneously programming, analyzing, designing and testing
- pair programmers:
  - keep each other on task
  - brainstorm refinements to the system
  - clarify ideas
  - take initiative when partner is stuck
  - hold each other accountable to the team's practices

- can’t work alone? yes, you can, but, bring the design to the table to be implemented with your pair
- tiring but satisfying (bottle of water)
- rotate pairs frequently
- issues?!?

STORIES

- units of customer-visible functionality
- write story >> estimate it
- business and technical perspectives
- short names
- short description/prose or graphical description
- write them on index cards and put them on the wall



SAVE WITH COMPRESSION | 8 HRS

· CURRENTLY THE COMPRESSION OPTIONS ARE IN A DIALOG SUBSEQUENT TO THE SAVE DIALOG. MAKE THEM PART OF THE SAVE DIALOG ITSELF.

- (early estimation)
- how to get the greatest return from the smallest investment

WEEKLY CYCLE

- plan work a week at a time
- meeting in the beginning of the week
  - review the progress to date (actual result versus expected result)
  - customer's pick the week's stories
  - break stories into tasks (sign up and estimate)

- write automated tests that will run upon the completion of the stories
- implement stories and making tests pass
- goal: deployable software at the end of the week (proud team will celebrate progress)
- inspect and adapt
- working weekends is not sustainable

- planning is a form of necessary waste
- break stories into tasks OR write small stories (more work for the customer)
- sign up or pile of tasks

QUARTERLY CYCLE

- plan work a quarter at a time
- identify bottlenecks
- initiate repairs
- plan the theme
- pick a quarter’s worth of stories
- focus on the big picture
- quarter OR season
- good interval for reflection

SLACK

- include minor tasks that can be dropped
- keep your commitments
- include a “slack” in your schedule

10 MINUTE BUILD

- automatically build the whole system and run all the tests in 10 minutes
- software has few certainties
- more valuable than builds that require manual intervention

CONTINUOUS INTEGRATION

- integration == nightmare
  - the longer you take the higher the costs
- more time than programming
- team programming = divide, conquer and integrate
  - async >> check in changes (commit), build and test, report problems by email, sms or a glowing red lava lamp
  - sync >> done after each coding cycle and no more than a couple of hours & wait for the results

TEST-FIRST PROGRAMMING

- write an automated test before changing any code
- problems addressed:
  - scope creep
  - coupling and cohesion
  - trust
  - rhythm
- continuous testing
  - every time you write new code, start by writing the test (regression testing)
- alternatives?

INCREMENTAL DESIGN

- invest in the design on a daily basis
- Barry Boehm - 1960
  - "the cost of fixing defects rose exponentially over time"
- most economical design strategy: make big decisions early and defer all small-scale decision until later
- changes are as costly as defects?
- keep the design investment proportional to the needs of the system so far

- incremental design
    - eliminate duplication
    - simplify the system
    - refactor

GETTING STARTED

- developing software already?
- how and where to start?
  - primary practices are safe
  - start changing 1 thing at a time
  - hard: jump all in and do all the practices, embrace all the values and apply all the principles
- change is not slow, but progress should be measurable

- change starts with awareness
- feelings and metrics
- change starts at home - you can only change yourself

COROLLARY PRACTICES

DON'T TRY THIS AT HOME! OR DO TRY, BUT FIRST, WORK ON THE PRIMARY PRACTICES!

Kent Beck

REAL CUSTOMER INVOLVEMENT

- make stakeholders for your product a part of the team
- they value the product, they pay for it
- they have the needs >> put them in contact with those who can fill those needs
- eliminate waste == **DON'T**
  - develop features that aren't used
  - specify tests that don't reflect acceptance criteria

- objections?
  - 1 size fits 1 customer (and not anyone else)
    - make what the 1 customer needs and wants, the desire of another customer
  - “sausage factory”
    - if they knew… :|
    - fosters trust and encourages continuous improvement, while enhancing relationships

INCREMENTAL DEPLOYMENT

- big deployments >> high risk, and, high human and economic costs
- deploy soon, continuously and empower users - provide them with insurance

TEAM CONTINUITY

- keep them together
- simplifying work relationships and what they achieve is false economy
- large organizations often ignore the value of teams
    - “resources” go back into the pool, once the project is over
- static teams? NO. Moving a person from time to time consistently spreads knowledge and experience.

SHRINKING TEAMS

- when a team grows in capability, keep its workload constante but gradually reduce its size.
- when a team has too few members, merge it with another too-small team (toyota production system)
- figure out how many stories the customer needs each week >> improve development until some members are idle >> shrink the team.

ROOT-CAUSE ANALYSIS

- each defect has a cause
- when found, eliminate both defect and cause
  1. write automated system-level test that demonstrates the defect and the desired behaviour
  2. write a unit test with the smallest scope possible that reproduces the defect
  3. fix the system (the tests should both pass)
  4. figure out the cause of the defect and change to prevent this from happening again

SHARED CODE

- anyone can improve any part of the code at anytime
- collective ownership
- irresponsible? “every man for himself”
- pride of the work done >> pair programming helps >> higher quality

CODE AND TESTS

- only code and tests are permanent artifacts
- generate other documents from the code and tests
- rely on social mechanisms to keep the project history safe

SINGLE CODE BASE

- one code stream
- develop temporary on a branch that only lives for a few hours
- multiple code streams are a waste
- legitimate reasons to multi-code streams

  7 code bases for 7 distinct customers

  - >> config files?
  - >> (re)design

DAILY DEPLOYMENT

- put new software into production every night
- defect rate: no more than a handful per year
- build environment: automated (roll back allowed)
- projects and tasks that take months to become usable?
- barriers:
    - technical
    - psychological or social
    - business related

NEGOTIATED SCOPE CONTRACT

- contracts fix time, costs and quality but require ongoing negotiation of the scope
- reduce risk: sign sequence of short contracts instead of a big one
- “change requests” can be included smoothly in shorter contracts
- doing something just for the sake of being in a contract - when it doesn’t make sense for anyone

PAY-PER-USE

- you charge for every time the system is used
- get feedback to improve
- pay-per-release
  - hurts the customer!
  - hurts: upgrade!
  - hurts: how much money will I spend?
  - subscription model (or support fees)
  - less up-front revenue

THE WHOLE XP TEAM

EXECUTIVE

TESTER

INTERACTION DESIGNER

PROJECT MANAGER

PRODUCT MANAGER

ARCHITECT

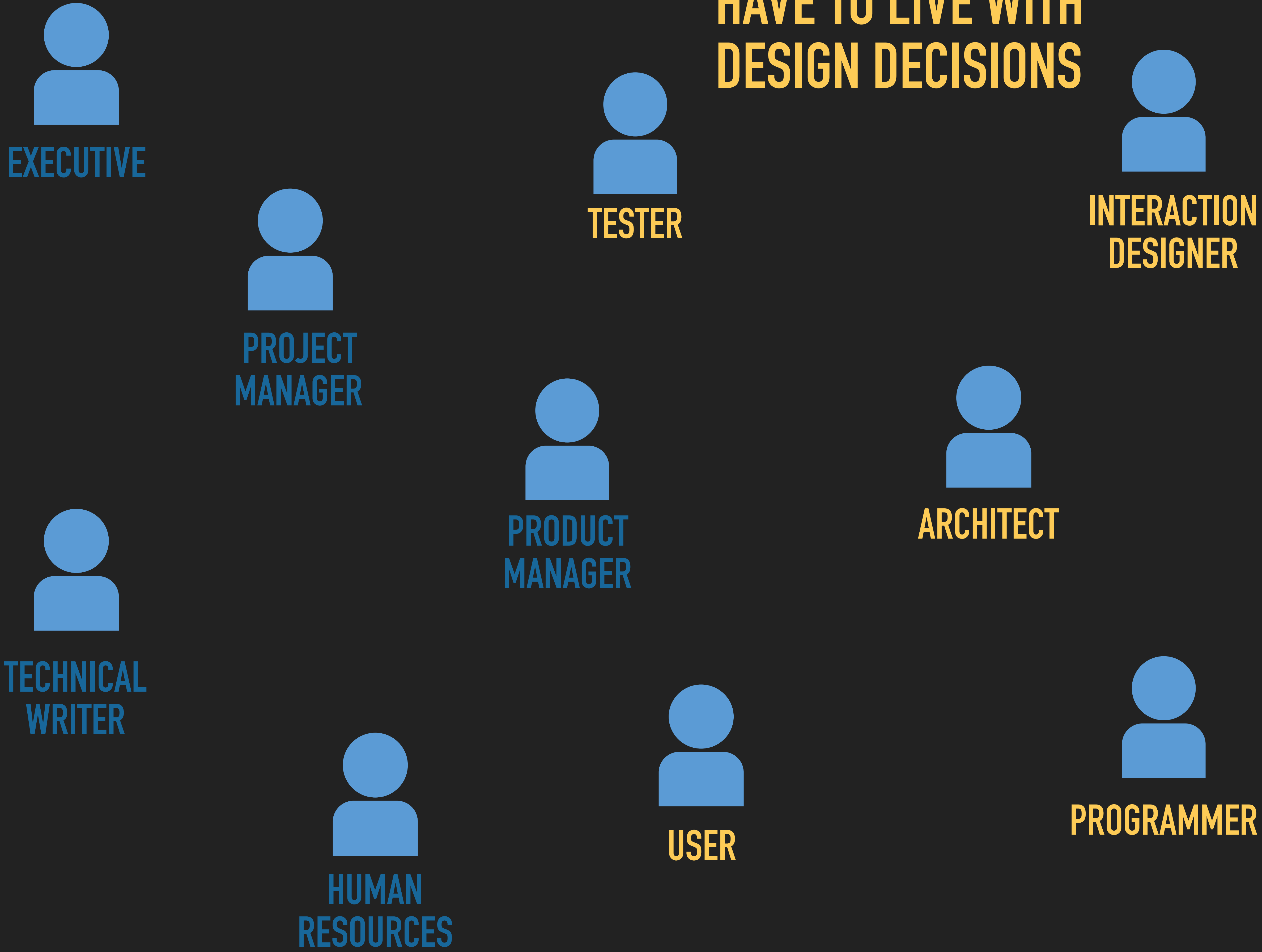
TECHNICAL WRITER

USER

PROGRAMMER

HUMAN RESOURCES

HAVE TO LIVE WITH DESIGN DECISIONS


EXECUTIVE
TESTER
INTERACTION DESIGNER
PROJECT MANAGER
PRODUCT MANAGER
ARCHITECT
TECHNICAL WRITER
HUMAN RESOURCES
USER
PROGRAMMER