Working Software
and how to make it

@antonkeks
Agile Manifesto, 2001

Working Software *over* Comprehensive Documentation
Craftsmanship Manifesto, 2009

Not only working software, but also well-crafted software
The problem is...

Many developers have no idea what

“Working Software”

or even

“Well-crafted Software”

actually means!!!
Манифест, блять!

Мы — сообщество программистов, которые вдоволь намучились с разнообразными методологиями разработки.

Нас уже тошнит и от экстремального программирования, и от Скрама, и от Канбана, и вообще от всего, что напрямую не связано с написанием блять кода.

Мы выступаем за то, чтобы отказаться от всех существующих методологий разработки в пользу одной, самой простой и единственно верной. Эта методология называется «пиши код блять»!

<table>
<thead>
<tr>
<th>Что говорят менеджеры?</th>
<th>Что им нужно на самом деле?</th>
<th>Что мы делаем в итоге?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Продуктивная командная работа</td>
<td>Сотни рабочих часов в неделю</td>
<td>Мы пишем код блять!</td>
</tr>
<tr>
<td>Качественные продукты</td>
<td>100% покрытие кода юнит-тестами</td>
<td>Мы пишем код блять!</td>
</tr>
</tbody>
</table>
Working Software

- Fulfills business goals
- Has value for users
- Easy to use (usability for end-users)
- Easy to deploy, reliable (usability for admins)
- Easy to monitor / audit / trace bugs
- Easy to change / fix / maintain (usability for devs)
- Easy to add features to
- Easy to test (automatically), to keep it working
- Hard to break-in or steal data (security)
- Ideally, no extra work for anyone besides changing code+tests
It is more important for code to be changeable than that it work. Code that does not work, but that is easy to change, can be made to work with minimum effort. Code that works but that is hard to change will soon not work and be hard to get working again.
Another definition

Working software is **fully integrated, tested, and ready to** be shipped to customers or deployed into production.

That doesn't mean you tried it a couple times and it ran without aborting. It means you created unit tests, QA tests, and actually looked at output to prove it works.

[http://www.agile-process.org/working.html](http://www.agile-process.org/working.html)
“Works on my ...” syndrome

Not enough tests or...

Hand-crafted environment  
   == evil || unprofessional

It doesn’t scale without automation
Like in alchemy

There’s always a secret ingredient to make it run...

Some secret profile
Deps in local maven repo
DB should be running on some IP
JNDI resources...
Config params missing
Other rituals/magic needed

Usually the result of either

Lack of CI

or overly complex CI setup

(by those infrastructure specialists?)
“Doesn’t work even on my PC” syndrome

Some developers have no idea how to run their software

...and they still are trying to work on it

Microservices, Maven profiles, config, etc
Avoiding broken software problems

Take total control over your app’s build/environment/infrastructure.

Don’t let infrastructure engineers do it for you (they will introduce more complexity and slowness than needed).

DevOps made being a “sysadmin” unpopular, but now we have e.g. Kubernetes engineers.

Let’s look at the basics...
My computer doesn't work! The hard drive crashed! What do I do?!

Did you back up?

Why? Is it gonna blow?!
12 Factor Apps by Heroku - 12factor.net

I. Codebase
II. Dependencies
III. Config
IV. Backing services
V. Build, release, run
VI. Processes
VII. Port binding
VIII. Concurrency
IX. Disposability
X. Dev/prod parity
XI. Logs
XII. Admin processes
Modern compliance

Git + Docker + common sense

...provide most (but not all) for free

Docker fully controls build/run environments

Nothing undefined will leak in

Docker-compose is a good universal format

Kubernetes is not KISS - don’t depend on it
I. Codebase, II. Dependencies

One versioned codebase, many deploys
Explicitly declare and isolate dependencies
But fear them as hell!
Fewer is better - KISS
No hand-tuned build/run environments!
OS/utilities/system libraries are also dependencies
Dockerfile FROM
III. Config, VII. Port binding

Read config from the environment (vars)

No test/prod/etc configs in git, because there may be many more + security

Docker can map to any external port

Default dev config for good DX

Working docker-compose.yml
IV. Backing services, X. Dev/prod parity

I.e. runtime dependencies

Treat backing services as **attached** resources

Internal/external are the same

No code changes to attach to a different service/db/etc
docker-compose.yml example

services:
  myapp:
    build: .  # will use local Dockerfile
    scale: 2
    environment:
      DB_HOST: db
  db:
    image: postgres:12-alpine
    environment:
      POSTGRES_DB: myapp

Use `docker-compose.override.yml` to map ports for local development
V. Build, release, run

Strictly separate build and run stages

**One build for all** (Except when building for different hardware/platforms)

No code changes in prod - PHP devs still do that!

Avoid shipping the compiler or other tools for security

Don’t build separately for dev/test/prod

Does your Jenkins look like this?

MyApp_build_pipeline_test
MyApp_build_pipeline_prod
FROM openjdk:11 as build
COPY ...
RUN ./gradlew test package

FROM openjdk:11-jre  # or use jlink to build a custom jre
COPY --from=build path/to/**.jar
EXPOSE 8080
CMD java -jar *.jar
VI. Processes, VIII. Concurrency

One or more **stateless** processes for scaling, no even sticky sessions

Quick startup, no local state

Scaling e.g. across multiple machines (horizontally)

Different process types can be scaled independently (web/batch)

External tools should handle crashes/restarts/etc

E.g. Systemd or Docker daemon

No Java app servers!

(which are slow and nightmare-ish anyway)
IX. Disposability

Robustness: Fast startup and graceful shutdown

Easier releases and scaling-up

(No Hibernate or complex Spring setup)

Also, no wasting time during development

For workers: idempotent operation

Robust against sudden death
X. Dev/prod parity

Keep development, staging, and production as similar as possible

Continuous deployment

All envs are really similar

Docker-compose gets you the same backing services
XI. Logs

Treat logs as event streams
Provide context in every message
Debug logs are noise (mostly)
No file management, just stdout
Good for development, flexible to deploy
Machine parseable
XII. Admin processes

Run admin/management tasks as one-off processes

Using the exactly same environment

docker exec -ti <container> script
App should start out-of-the-box
(batteries included)

Right after clone/checkout:

- From command-line
- From IDE, debugger, etc
DX: Developer Experience / Usability

Learn from many open-source projects:

Contributing should be easy

Good IDE, tools/scripts

Start with README (long = something’s wrong)

No waiting for anything

Apple UX vs DX
Good DX example

git clone <repo> && cd repo

cat README*

docker-compose up db -d

./gradlew run

- And it just works!
- Also tests just work and finish within seconds

./gradlew test
Tests should just run

Fast!

From IDE, under debugger if needed

One-by-one if needed

No external dependencies

No manual config

No f**ing extra spring/maven profiles

TDD gets you see your code “works” every minute or so
Fast tests?

Fast is 30 sec for 10 000 test cases

**Unit** tests are fast, they test your code only

You quickly understand why they fail

No Spring context initialization

No real server startup, no http requests

No overuse of Test Containers

Then, have some (also quick) integration tests as well

**Selenide.org** is great for UI tests
DB migrations

And other environment preparation should be automated

NO HAND-EXECUTED SQL SCRIPTS

Should be part of the code to build env from scratch, quickly

Should be done at runtime, not build time

Use Liquibase or Flyway, but not their Maven/Gradle plugins
Fast turnaround/feedback

Know instantly if you have broken anything

You control the project

Enjoy working on it
Complexity

Why do people like complex solutions?

Overengineering everywhere

Too many “moving parts”
  == unreliable

Invest time into simplifying your code/solutions
  It will pay back many times

**First**, you make it work, **then** you make it clean and simple
“Make everything as simple as possible, but not simpler.”
—Albert Einstein
Clean code

Read the book by Uncle Bob

Boy Scout rule:

“Leave the campground cleaner than you found it”

Avoid being negligent

Craft your code

Care for it
Kotlin has shown me the light of how you could be writing code so that it is concise, full of substance, free of tenuous repetition. Now I have an extreme intolerance to any form of boilerplate code. I see boilerplate everywhere, constantly thinking about how I can abstract it.
Framework jail

Be free

Control your frameworks & libraries

Always write your own main() method!

Every dependency should be replaceable

(can you control all those 1500 packages in your node_modules?)
Software Erosion

Bit rot

Uncle Bob:
“Software should get better over time, don’t accept it getting worse”

Active / Dormant

Heroku: Explicit (simple) Contracts to avoid Dormant erosion

Docker will (hopefully) allow you to recreate the build env after many years
Refactoring & Type safety

Constant refactoring prevents active erosion

Type safety enables easy automated refactoring

People avoid things that are not easy

Serious JavaScript development moving to TypeScript

Python 3.5 added type hints

Kotlin is more type-safe than Java, etc.
Well-Crafted Software

- Doing it professionally and skillfully
- Caring about your work and colleagues
- Maintaining it in working order
- Always striving for simpler solution
- YAGNI = “You Ain’t Gonna Need It”
Iterative process

Our customers don't always know the best way how to solve their problem

Help them by demonstrating Working Software frequently