The Agile Developer & Technical Excellence

Anja Stiedl

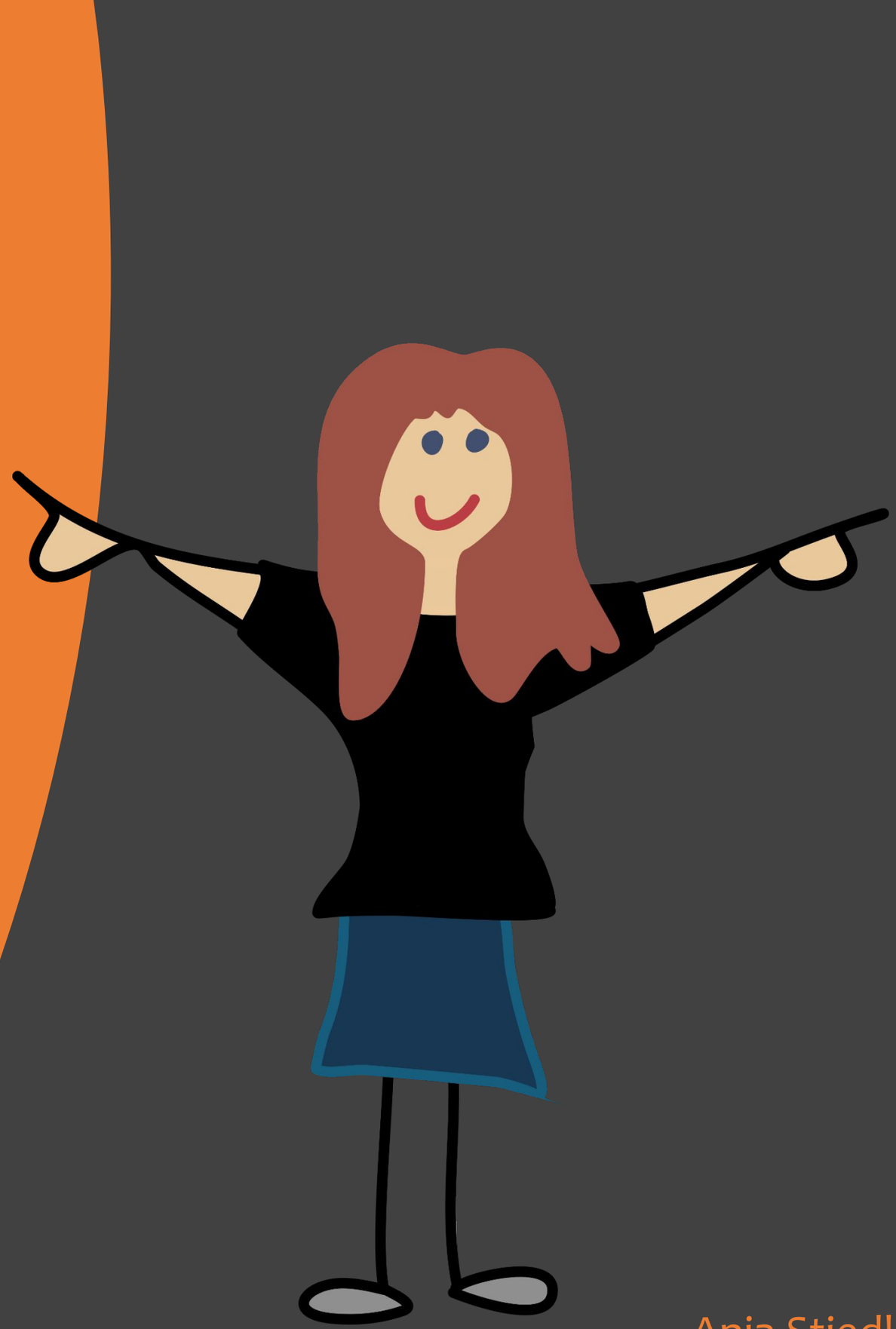
Warning and Disclaimer

my times in Software Development are quite a while ago…

there is a lot of information in here…
some old news, some new news…
and hopefully some interesting…

not all might be applicable easily
for your environment, for your systems,
for your technologies…

let’s figure out together,
what works or
how it could work for us!



Anja Stiedl, 2021

General

From the Agile Manifesto:

(https://agilemanifesto.org/principles.html)

Principle #9
Continuous attention to technical excellence and good design enhances agility.

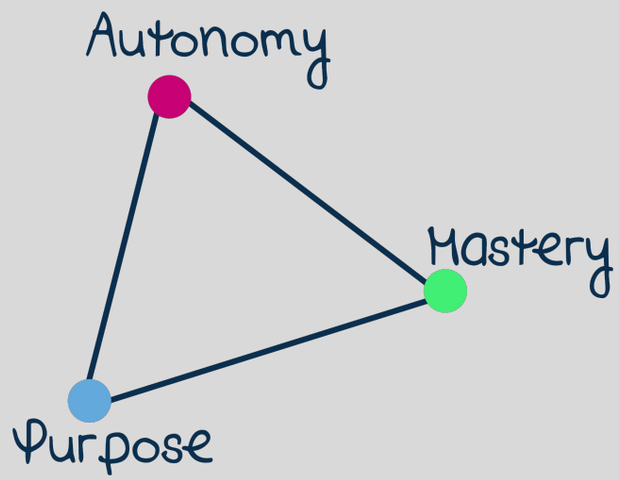
Principle #11
The best architectures, requirements, and designs emerge from self-organizing teams.

Lean:
eliminate Waste!

	Lean production	**Lean SW Development**
T	transportation	= task switching
I	inventory	= partially done work
M	motion	= motion
W	waiting	= waiting
O	over production	= extra feature
O	over processing	= extra processes
D	defects	= bugs
S		= unused skills

Human Factor: Motivation

(Daniel Pink: “Drive”,
youtube: https://www.youtube.com/watch?v=u6XAPnuFjJc)



Knowledge Workers are motivated by:

- Mastery
- Autonomy
- Purpose

Anja's strong opinion:

Tester, Developer & Professionals?

Testers are people who write and run software that tests products.

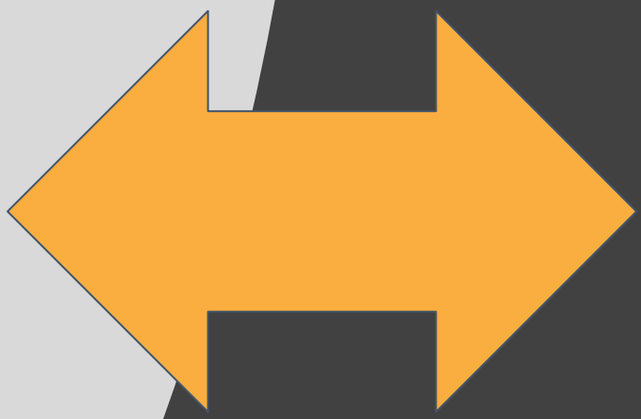
In Scrum we call them both (and others) "Developers".

Software-Developers are people who write professional quality software.

Software-Developers who don't ensure quality of their work, are not working professionally. Testing is one way to do so.

Human Factor: Developing vs. Testing

Testing destroys things



Developing creates things

Self Organizing Teams

what is it?

some aspects:

- all skills to get the job done - E2E, all phases of development, …
- teams take responsibility

- PO&devs plan the work in the iterations:
  - POs plan what to do
  - devs plan how much to do
  - devs plan how-to do it

- the Definition-of-Done defines the quality level to be achieved
- it's a contract between PO & devs

Community of Practise

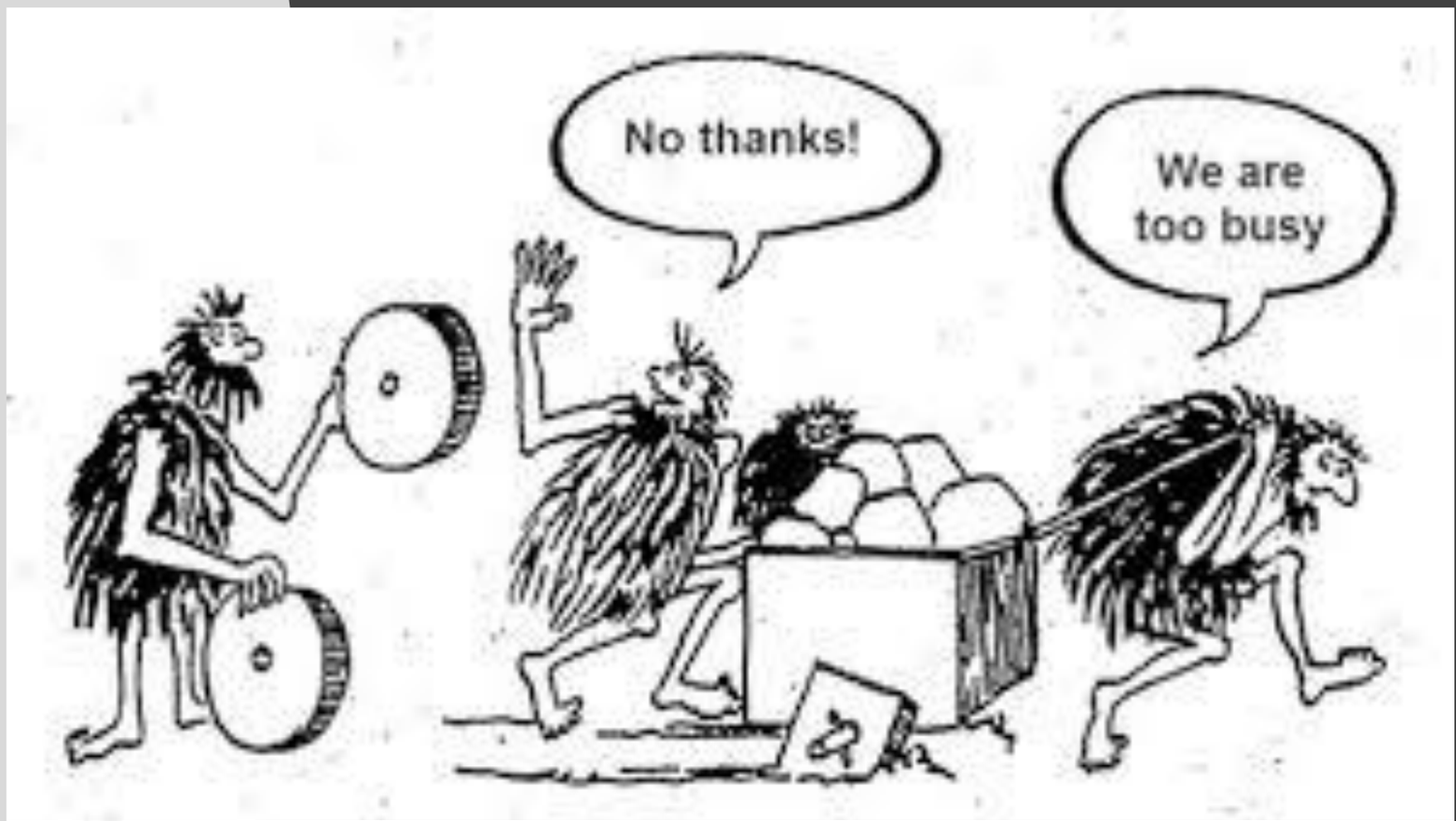
what is it?

A
community of practice (CoP)
is a group of people who "share a concern or a passion for something they do and learn how to do it better as they interact regularly."

[definition from wikipedia]

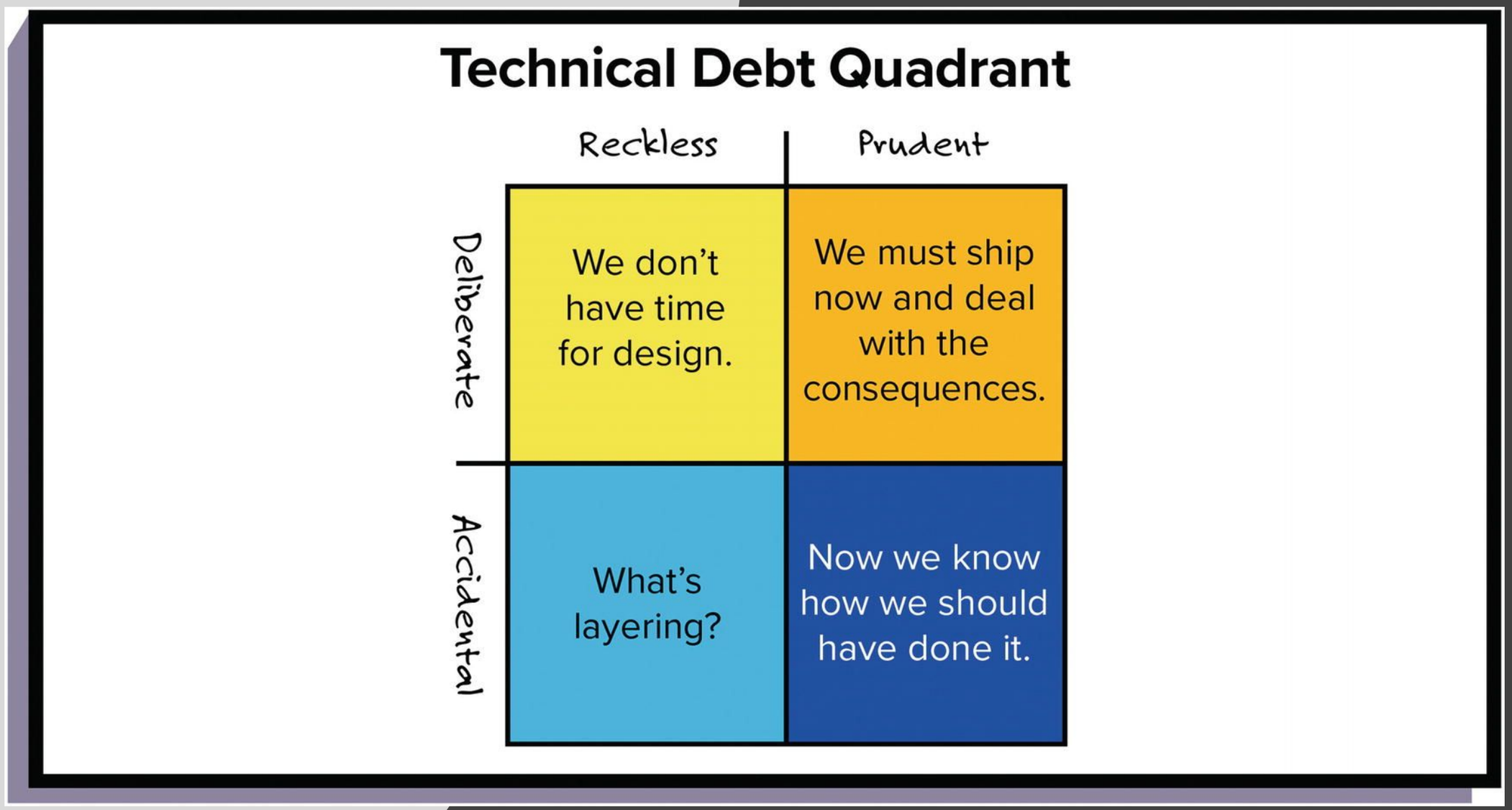
Technical Debt

what is it?



Technical debt (aka design debt or code debt) is a concept in software development that reflects the implied cost of additional rework caused by choosing an easy/limited solution now instead of using a better approach that would take longer.

Technical Debt Quadrant



Build in Software Quality

KISS

Keep it Simple and Stupid!

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

[adapted from Wikipedia]

Don’t Repeat Yourself!

DRY is a principle of software development to reduce repetition of software patterns, replacing it with abstractions or using data normalization to avoid redundancy.

In their book *The Pragmatic Programmer,* Andy Hunt and Dave Thomas apply it to include database schemas, test plans, the build system, even documentation. Besides using methods and subroutines in their code, they rely on code generators, automatic build systems, and scripting languages to observe the DRY principle across layers.

[adapted from Wikipedia]

SOLID

In software engineering, **SOLID** is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable. The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin in his Clean-Code-movement:

The **SOLID concepts** are

- **Single-responsibility principle**: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
- **Open–closed principle**: "Software entities ... should be open for extension, but closed for modification."
- **Liskov substitution principle**: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." See also design by contract.
- **Interface segregation principle**: "Many client-specific interfaces are better than one general-purpose interface."
- **Dependency inversion principle**: "Depend upon abstractions, not concretions.

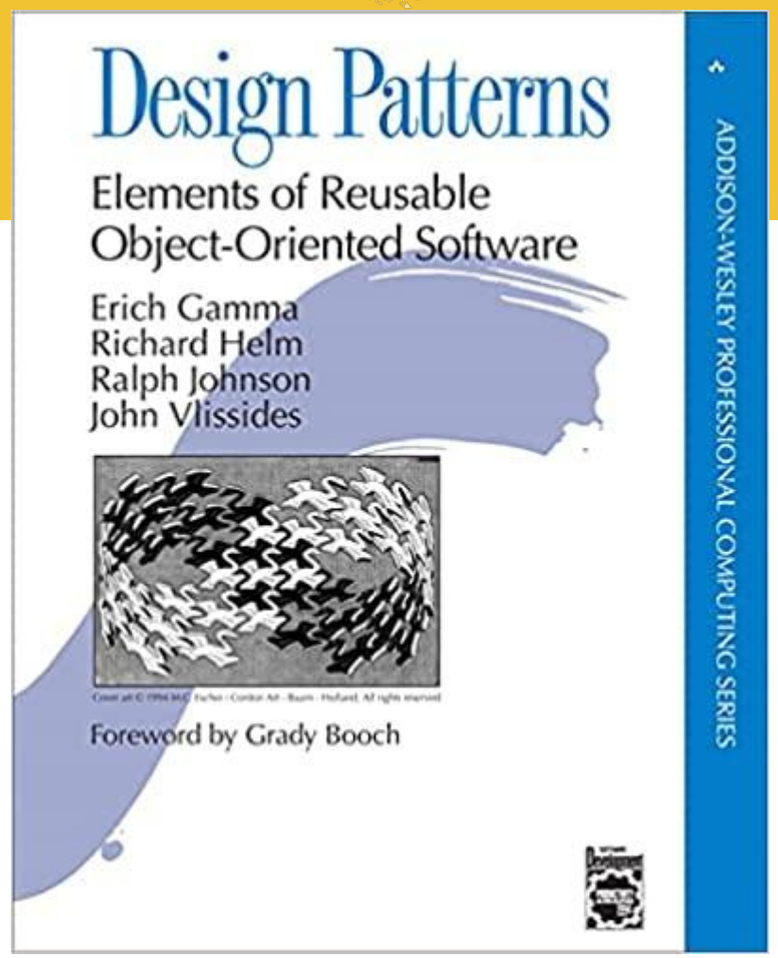
[adapted from Wikipedia]

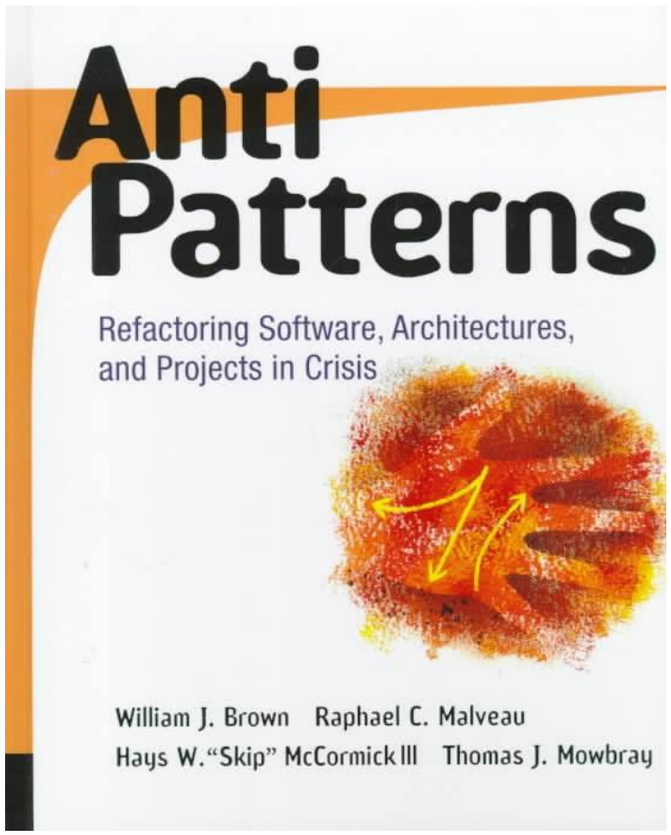
Design Patterns

Design Patterns help to design and write Software in a reusable way.

Capturing a wealth of experience about the design of object-oriented software, four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems. Previously undocumented, these 23 patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves.

Each pattern describes the circumstances in which it is applicable, when it can be applied in view of other design constraints, and the consequences and trade-offs of using the pattern within a larger design. All patterns are compiled from real systems and are based on real-world examples. Each pattern also includes code that demonstrates how it may be implemented in object-oriented programming languages like C++ or Smalltalk.



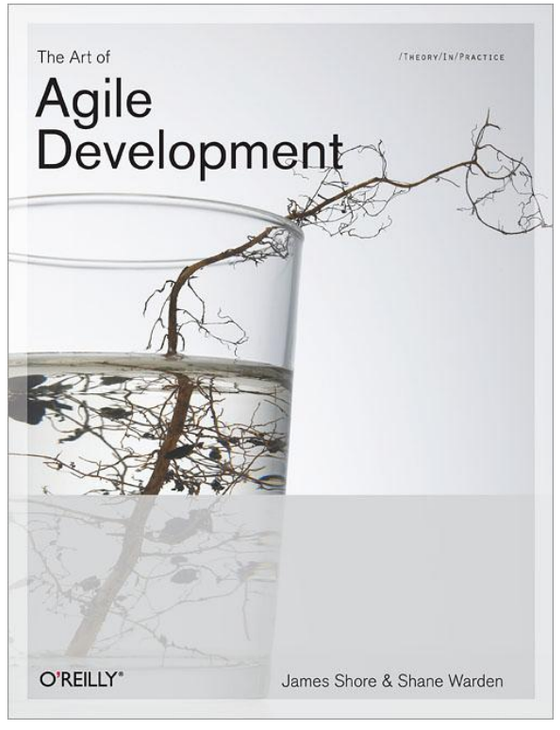


eXtreme Programming XP

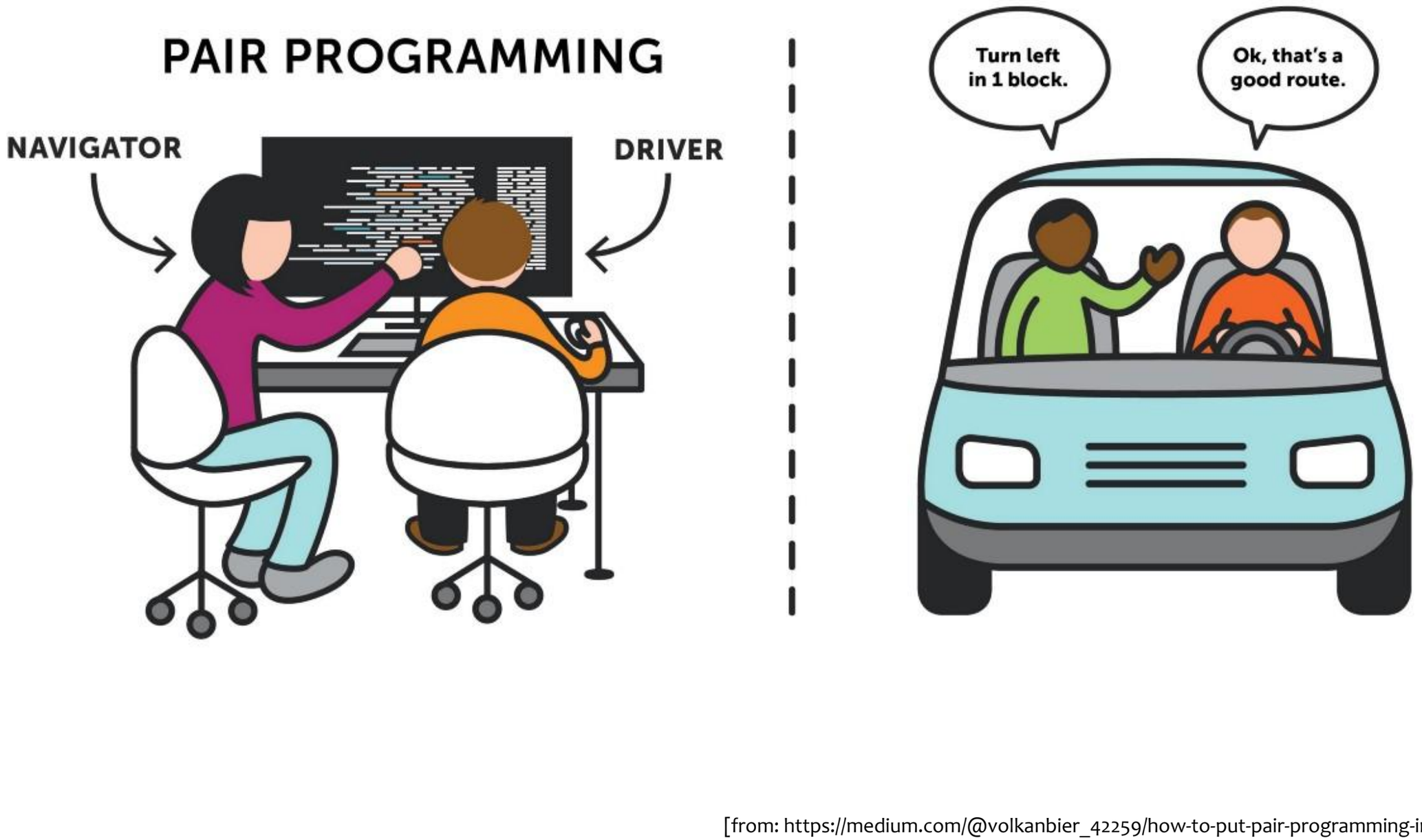
Extreme programming (XP) is a **software development methodology** which is intended to improve software quality and responsiveness to changing customer requirements. ... The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels. [from Wikipedia]

A great overview can be found in the book
James Shore: “The Art of Agile Development”

and on the corresponding website:
https://www.jamesshore.com/v2/books/aoad2



Pair Programming



[from: https://medium.com/@volkanbier_42259/how-to-put-pair-programming-into-action-ce9ebb9d711]

About Pair Programming

Benefits

- ❑ **Pair Quality / Review**
- ❑ **Pair Debugging**
- ❑ **Pair Negotiation**
- ❑ **Pair Learning**
- ❑ **Pair Focus**
  - ❑ **to the inside**
  - ❑ **to the outside**
- ❑ **Pair Courage**
- ❑ **Pair Trust**

Antipattern

- ❑ unequal / asymmetric access to keyboard / display
- ❑ Dominance at the keyboard
- ❑ Pairing-Marriages: no rotation between User Stories
- ❑ Worker-Lazybone-Paires
- ❑ Two Computers
- ❑ Both work on their own task
- ❑ 90% of User Stories 90% „done“
- ❑ People who can't stand each other have to do Pairing
- ❑ Discussions without progress take longer than 10 minutes

Variations of Pair Programming

Mob-Programming

The basic concept of **mob programming** is simple:

the entire team works as a team together on one task at the time. That is: one team – one (active) keyboard – one screen (projector of course).

— Marcus Hammarberg, Mob programming

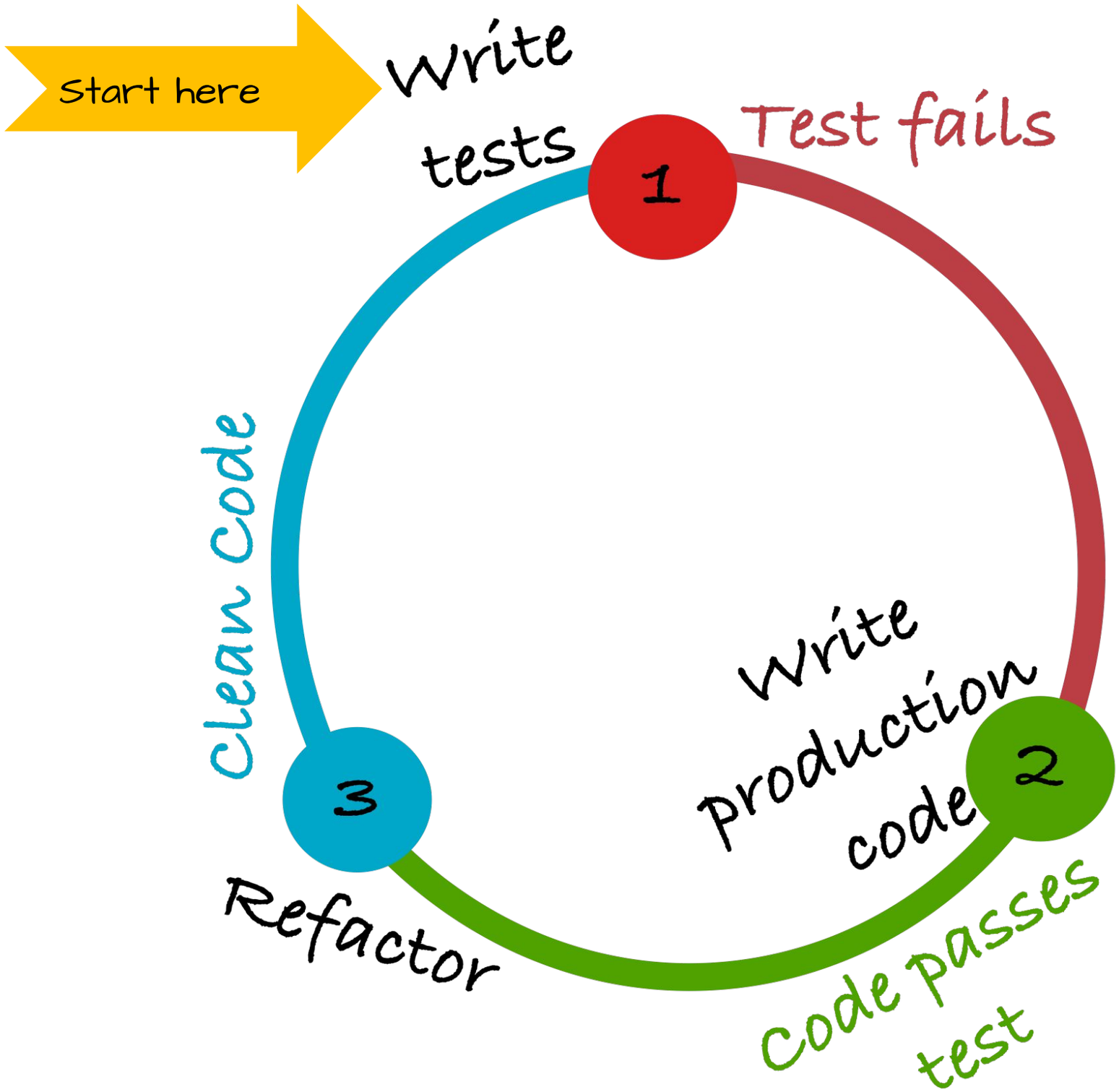
[from wikipedia]

Wolfpack-Programming

By moving the whole development environment to the cloud, we are no longer limited by the number of people who can comfortably fit around a single workstation; suddenly and **entire team of programmers can work together on the same live code base.**

— Julian Fitzell, Helge Nowak

Test Driven Development - TDD



Test-First-Development - is it the same as TDD?

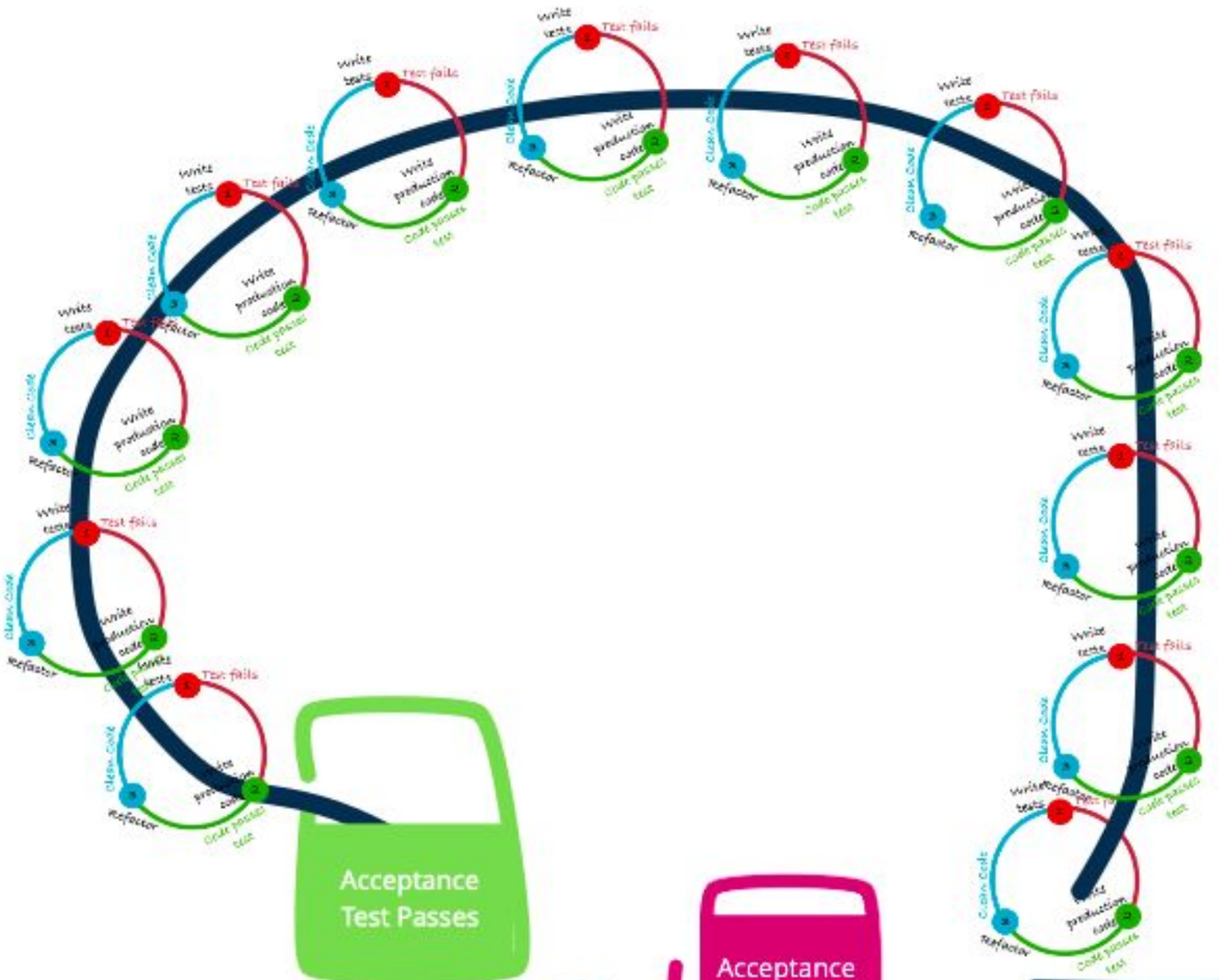
Test-First-Development/TFD and **Test-Driven-Development/TDD** are not synonyms.

Test-First-Development is when all the breaking tests are written first.

Test-driven Development requires only as much test-code written till it fails. That way the tests actually drive the design and increase confidence in the test-suite.

It's not just a semantic difference.

Acceptance Test Driven Development - ATDD



Acceptance Test Driven Development

User Story

for each Acceptance Criteria...

Write Acceptance Test

Acceptance Test Fails

Acceptance Test Passes

demo working Software...

Writing Acceptance Tests / UATs
FIT / FitNesse / Fixture

Fit ("Framework for Integrated Testing" is an engine that processes each **FitNesse** test table, using the **Fixture** Code referred to by that table.

FitNesse is an HTML and wiki "front-end" to Fit. While Fit makes it possible to run test tables, it does not itself provide means of creating those tables or displaying the results of tests. This is where *FitNesse* comes in. *FitNesse* makes it really easy to create, run, organize, annotate, and share Fit tests throughout a software development team.

Interestingly both the wiki and Fit were developed by Ward Cunningham, and you can read about them both on Ward's c2 wiki.

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

- Refactoring lowers the cost of enhancements
- Refactoring is a part of day-to-day programming
- Automated tools are helpful, but not essential

There is
-> no “Refactoring-Sprint”
-> no “Refactoring-Story”

It’s part of day-to-day programming.

Refactoring Catalog 1/2

Change Function Declaration (Add Parameter • Change Signature • Remove Parameter • Rename Function • Rename Method)

Change Reference to Value

Change Value to Reference

Collapse Hierarchy

Combine Functions into Class

Combine Functions into Transform

Consolidate Conditional Expression

Decompose Conditional

Encapsulate Collection

Encapsulate Record (Replace Record with Data Class)

Encapsulate Variable (Encapsulate Field • Self-Encapsulate Field)

Extract Class

Extract Function (Extract Method)

Extract Superclass

Extract Variable (Introduce Explaining Variable)

Hide Delegate

Inline Class

Inline Function (Inline Method)

Inline Variable (Inline Temp)

Introduce Assertion

Introduce Parameter Object

Introduce Special Case (Introduce Null Object)

Move Field

Move Function (Move Method)

Move Statements into Function

Move Statements to Callers

Parameterize Function (Parameterize Method)

Preserve Whole Object

Pull Up Constructor Body

Pull Up Field

Pull Up Method

Push Down Field

Push Down Method

Refactoring Catalog 2/2

Remove Dead Code

Remove Flag Argument (Replace Parameter with Explicit Methods)

Remove Middle Man

Remove Setting Method

Remove Subclass (Replace Subclass with Fields)

Rename Field

Rename Variable

Replace Command with Function

Replace Conditional with Polymorphism

Replace Constructor with Factory Function (Replace Constructor with Factory Method)

Replace Control Flag with Break (Remove Control Flag)

Replace Derived Variable with Query

Replace Error Code with Exception

Replace Exception with Precheck (Replace Exception with Test)

Replace Function with Command (Replace Method with Method Object)

Replace Inline Code with Function Call

Replace Loop with Pipeline

Replace Magic Literal (Replace Magic Number with Symbolic Constant)

Replace Nested Conditional with Guard Clauses

Replace Parameter with Query (Replace Parameter with Method)

Replace Primitive with Object (Replace Data Value with Object • Replace Type Code with Class)

Replace Query with Parameter

Replace Subclass with Delegate

Replace Superclass with Delegate (Replace Inheritance with Delegation)

Replace Temp with Query

Replace Type Code with Subclasses (Extract Subclass • Replace Type Code with State/Strategy)

Return Modified Value

Separate Query from Modifier

Slide Statements (Consolidate Duplicate Conditional Fragments)

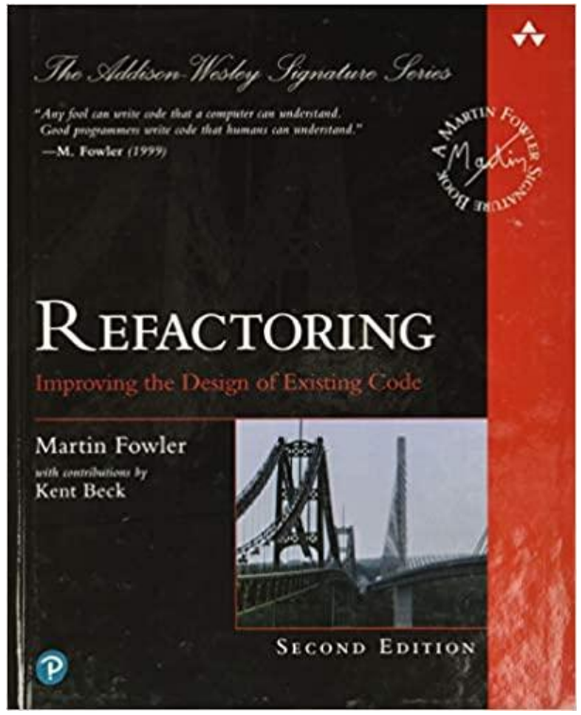
Split Loop

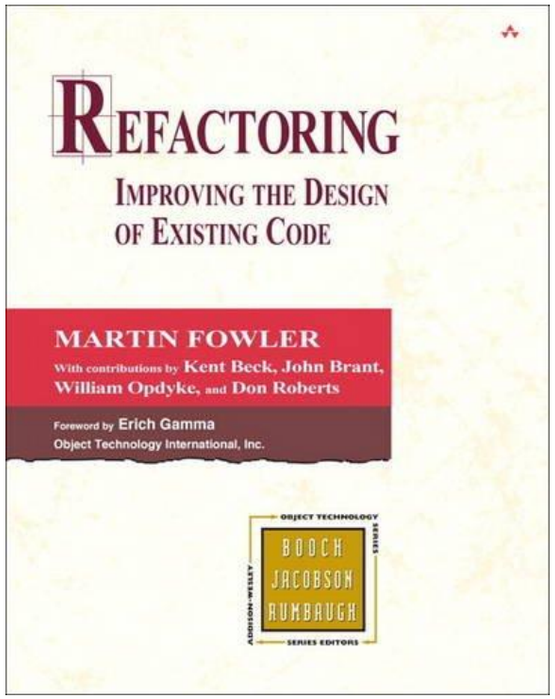
Split Phase

Split Variable (Remove Assignments to Parameters • Split Temp)

Substitute Algorithm

Refactoring - again!





Definition

Martin Fowler gives the following definition of “refactoring”:

noun: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

verb: to restructure software by applying a series of refactorings without changing its observable behavior.

Refactoring isn't another word for cleaning up code - it specifically defines one technique for improving the health of a code-base.

The term "restructuring" could be used as a more general term for reorganizing code that may incorporate other techniques.

Behaviour Driven Development - BDD

Behavioral specifications

Title: Returns and exchanges go to inventory.

As a store owner,
I want to add items back to inventory when they are returned or exchanged,
so that I can track inventory.

Scenario 1: Items returned for refund should be added to inventory.
Given that a customer previously bought a black sweater from me
and I have three black sweaters in inventory,
when they return the black sweater for a refund,
then I should have four black sweaters in inventory.

Scenario 2: Exchanged items should be returned to inventory.
Given that a customer previously bought a blue garment from me
and I have two blue garments in inventory
and three black garments in inventory,
when they exchange the blue garment for a black garment,
then I should have three blue garments in inventory
and two black garments in inventory.

Cucumber / Gherkin

Cucumber is a software tool that supports behavior-driven development (BDD)

Central is its ordinary language parser called **Gherkin.**

It allows expected software behaviors to be specified in a logical language that customers can understand.

As such, Cucumber allows the execution of feature documentation written in business-facing text.

It is often used for testing other software.

It runs automated acceptance tests written in a behavior-driven development (BDD) style.

Example Gherkin

```gherkin
# Comment
@tag
Feature: Eating too many cucumbers may not be good for you
  Eating too much of anything may not be good for you.

  Scenario: Eating a few is no problem
    Given Alice is hungry
    When she eats 3 cucumbers
    Then she will be full
```

(Automated) GUI Testing

#1) Manual Based Testing:

Testers apply their knowledge and test the graphical screen as per business requirements.

#2) Record and Replay:

This is achieved using automation tools and their Record and Replay actions. Test steps are captured in the automation tool during Record and recorded steps are then executed on the application under test during Replay/Playback.

#3) Model-Based Testing:

- Event-based model: Based on GUI events that are to occur at least once
- State-based model: Based on GUI states exercised at least once
- Domain model: Based on domain and functionality of the application

  Following steps are required:

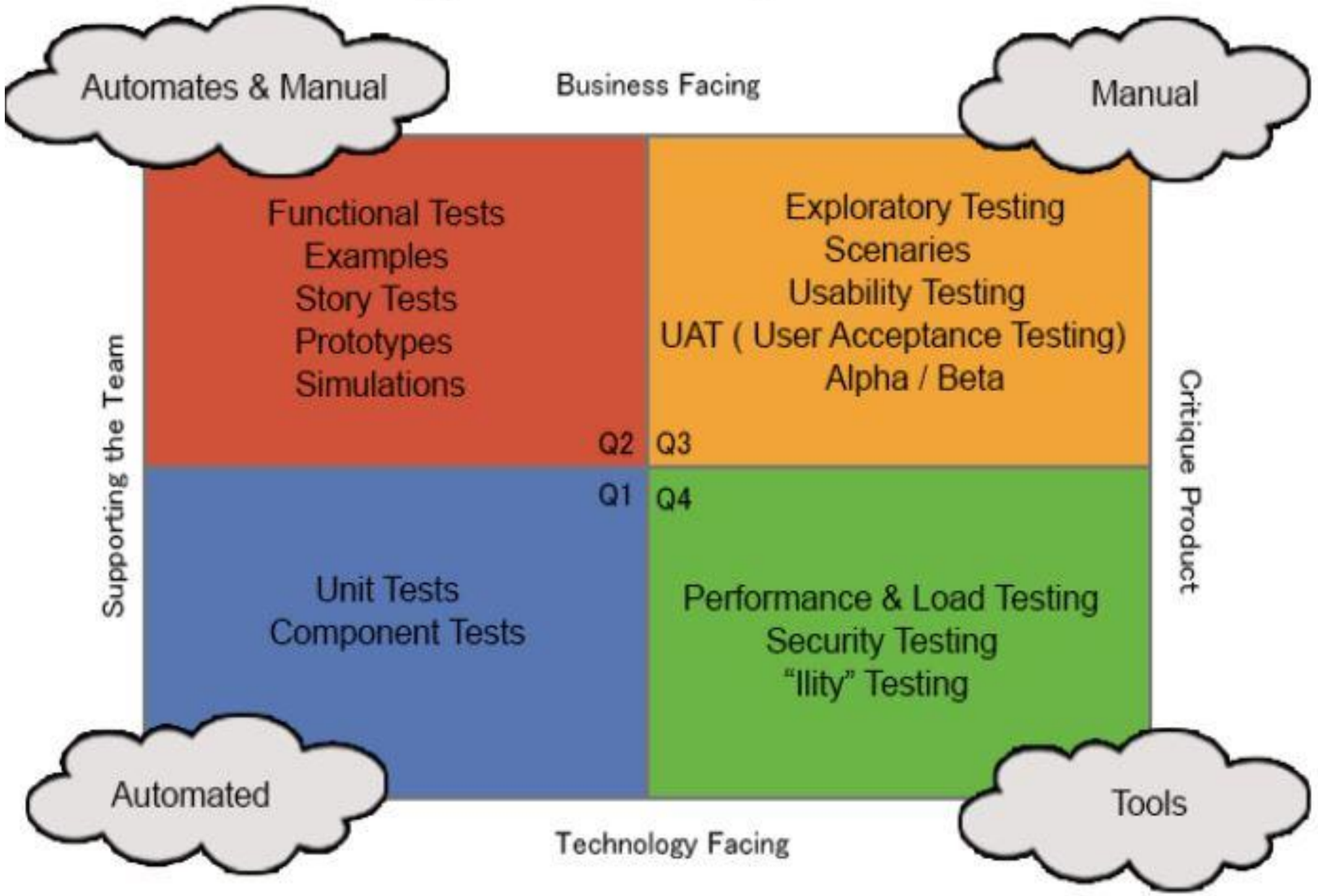
  - Build the model
  - Specify inputs to the model
  - Determine expected outputs
  - Execute tests
  - Compare actual and expected results
  - Decide future actions to be taken

Automated GUI Testing

Selenium is an open-source automation testing tool that supports a number of scripting languages such as Java, Perl, C#, Ruby, JavaScript, etc. It offers testers the flexibility to choose the script language, depending on the application under test. Additionally, Selenium caters to testing requirements by offering high accuracy within a limited time period, saving time and resources.

Apply Quality Test

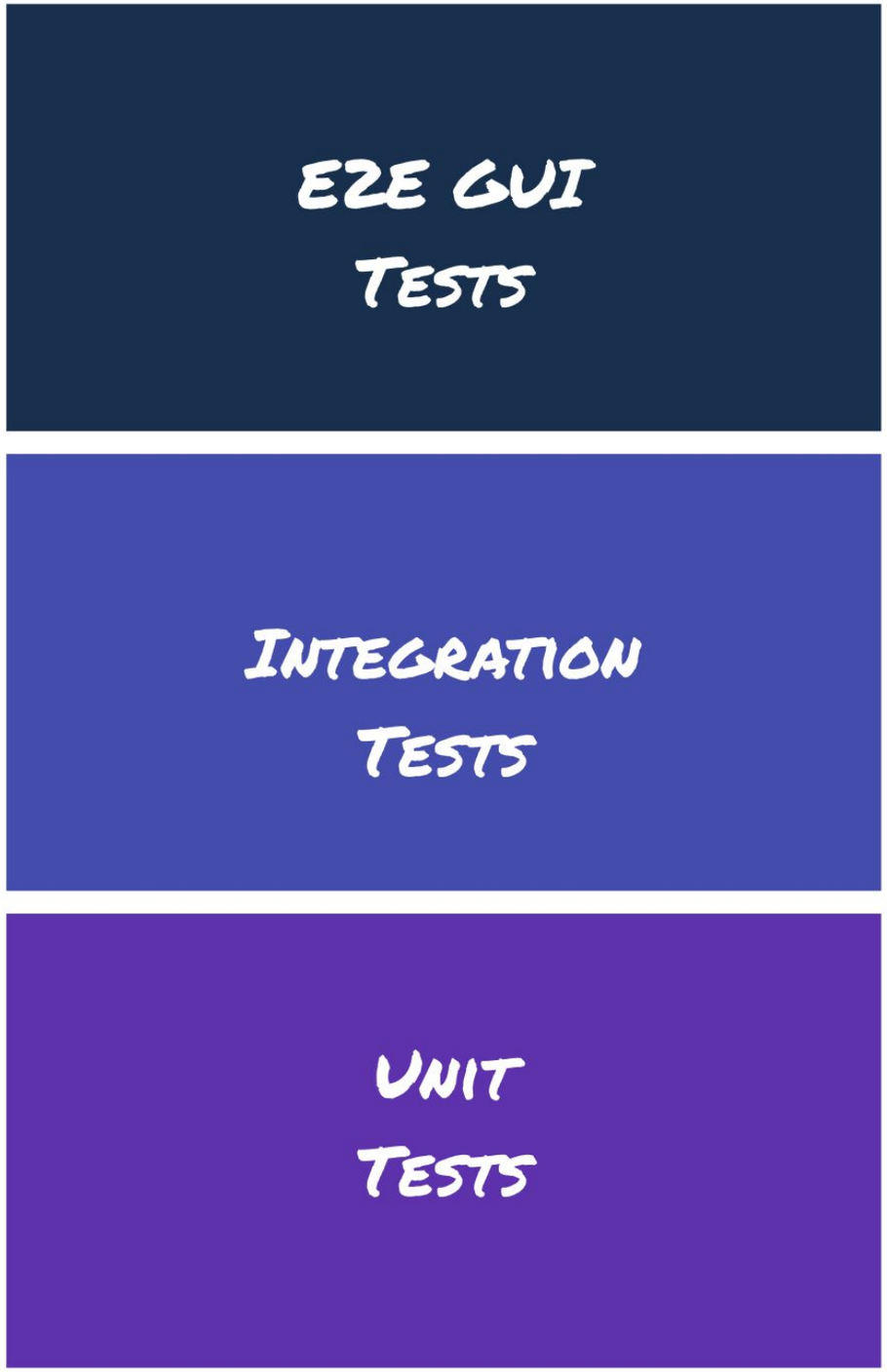
Agile Testing Quadrants



The 7 Software “-ilities” You Need To Know

1. Usability
2. Maintainability / Flexibility / Testilbility
3. Scalability
4. Availability / Reliability
5. Extensibility
6. Security
7. Portability

Agile Testing Triangle



Hard to automate

Easy to perform manually

=> expensive to run!

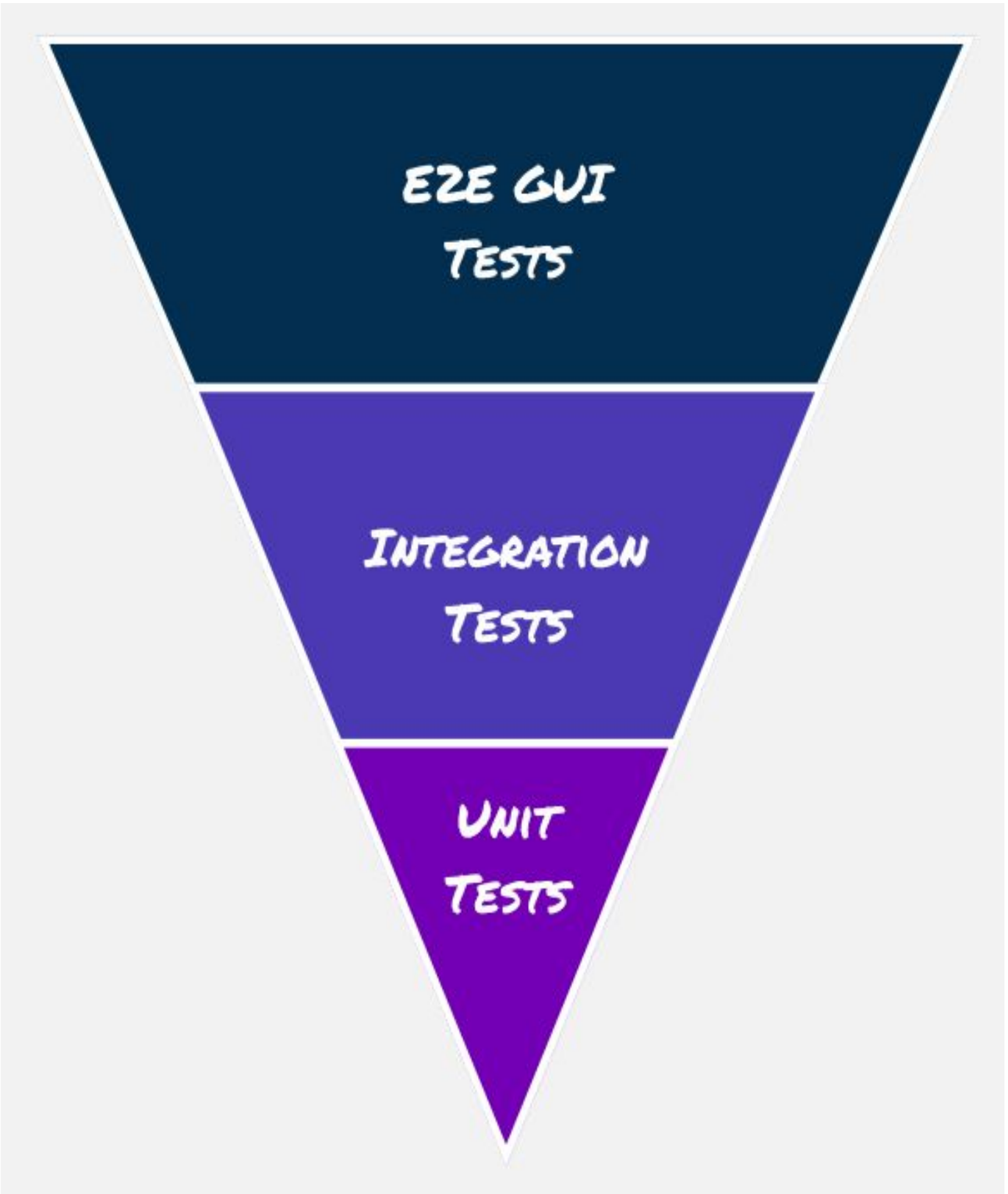
Reality?

Hard to write

Easy to automate

=> cheap to run!

Agile Testing Triangle



Hard to automate

Easy to perform manually

=> expensive to run!

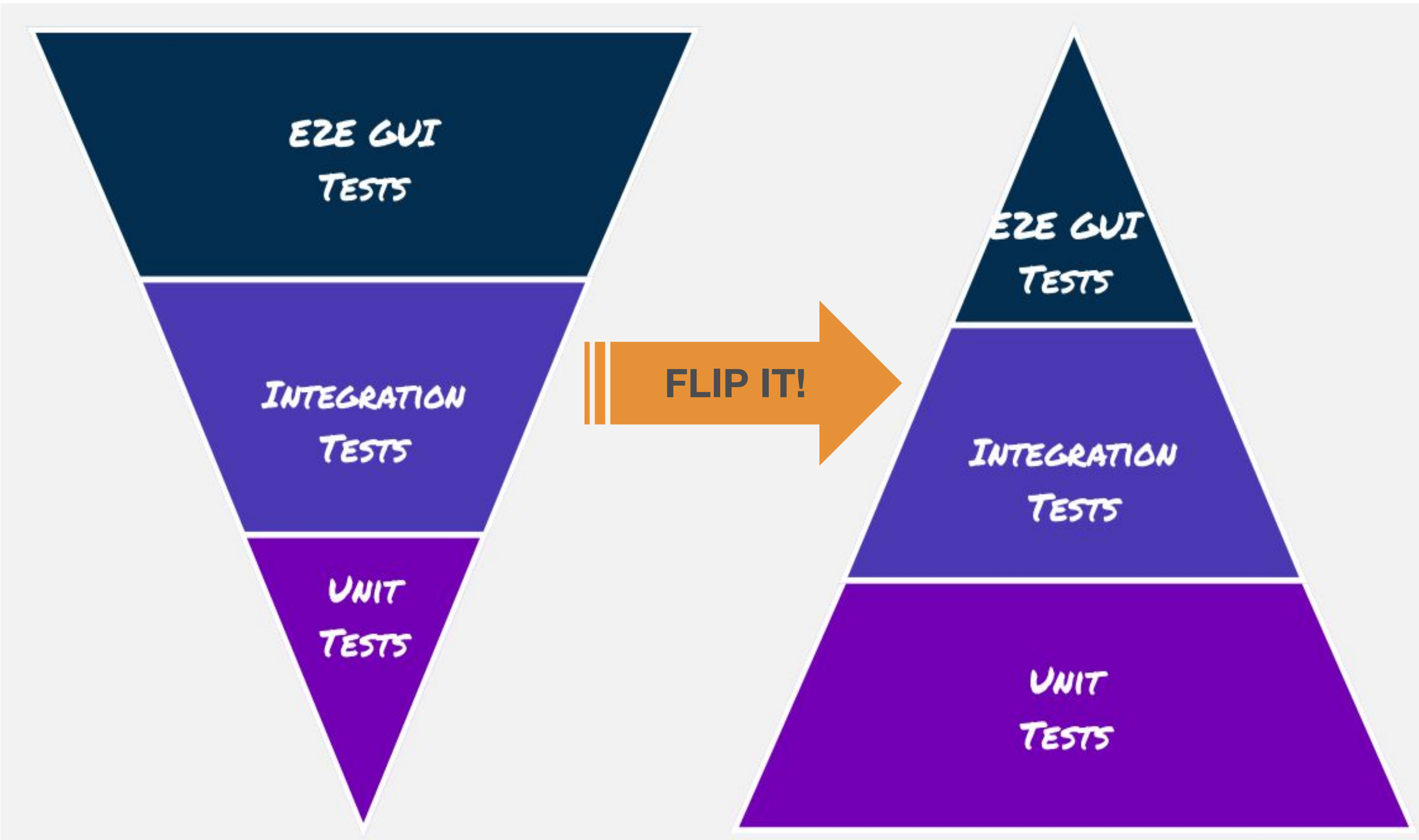
FLIP IT!

Hard to write

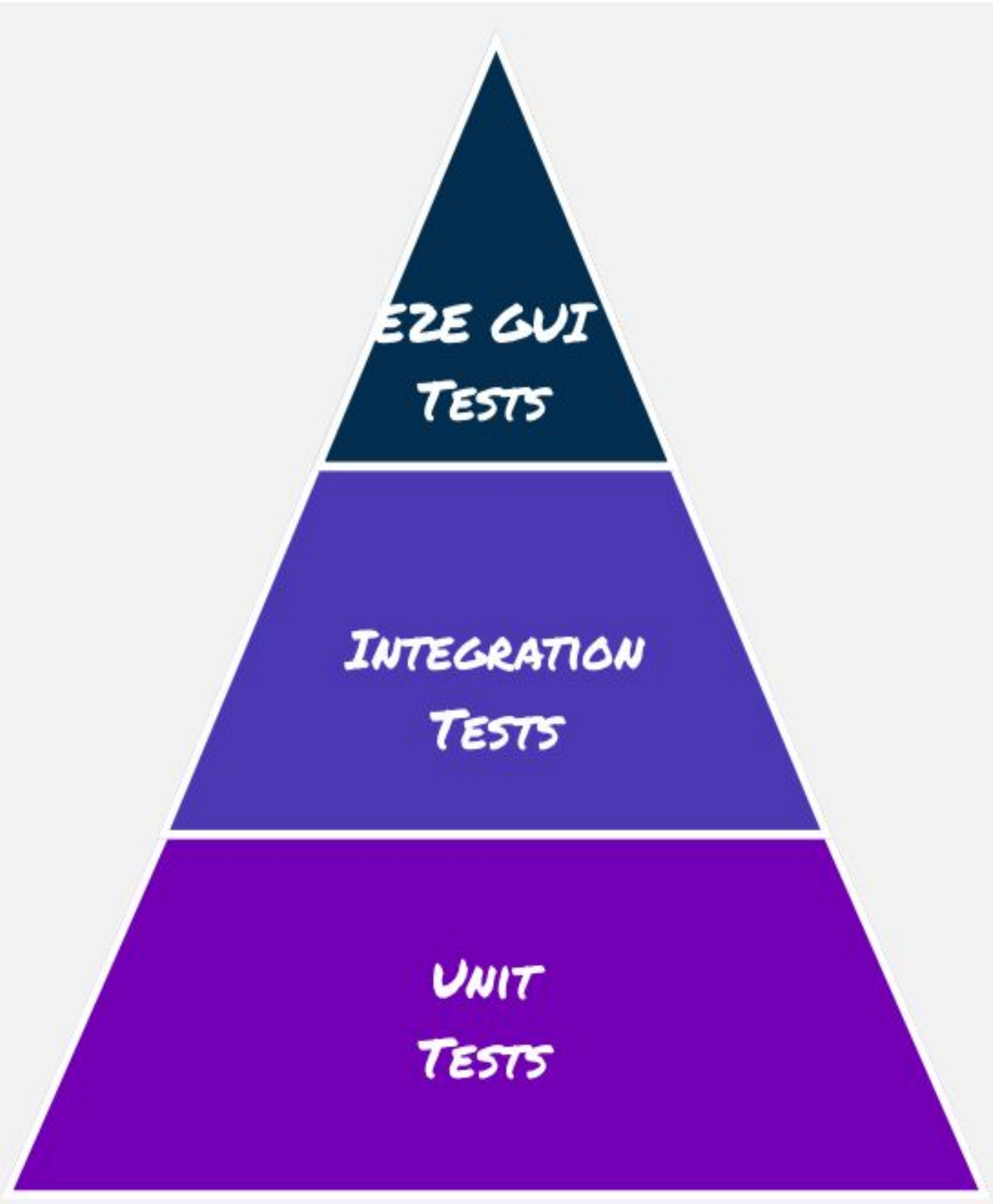
Easy to automate

=> cheap to run!

Agile Testing Triangle



Agile Testing Triangle



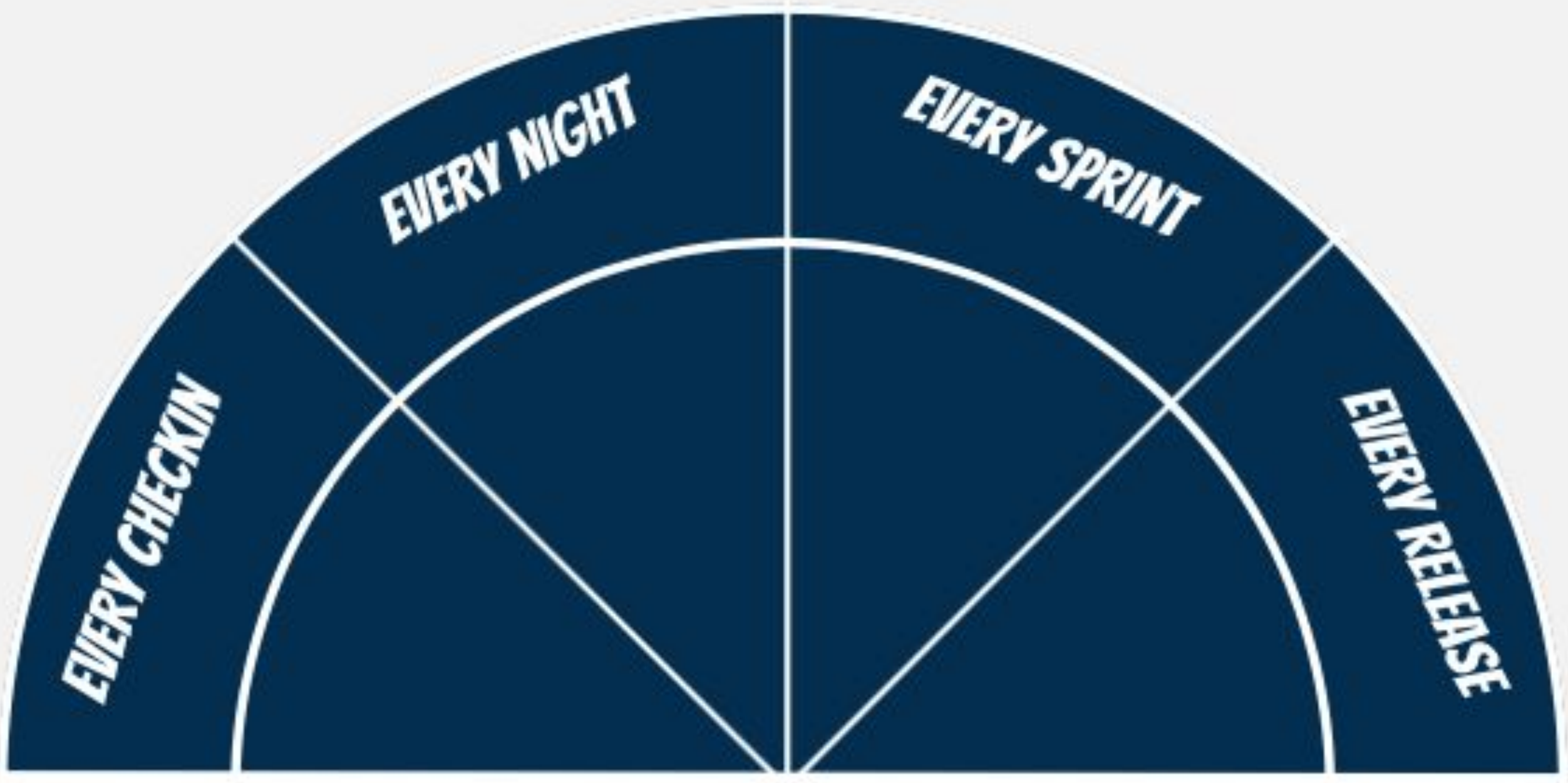
Automation expensive.
=> Automated UI / regression tests
– by using Selenium or QTP
=> Manual testing

=> Automated Acceptance Testing –
by Fit/ FitNesse

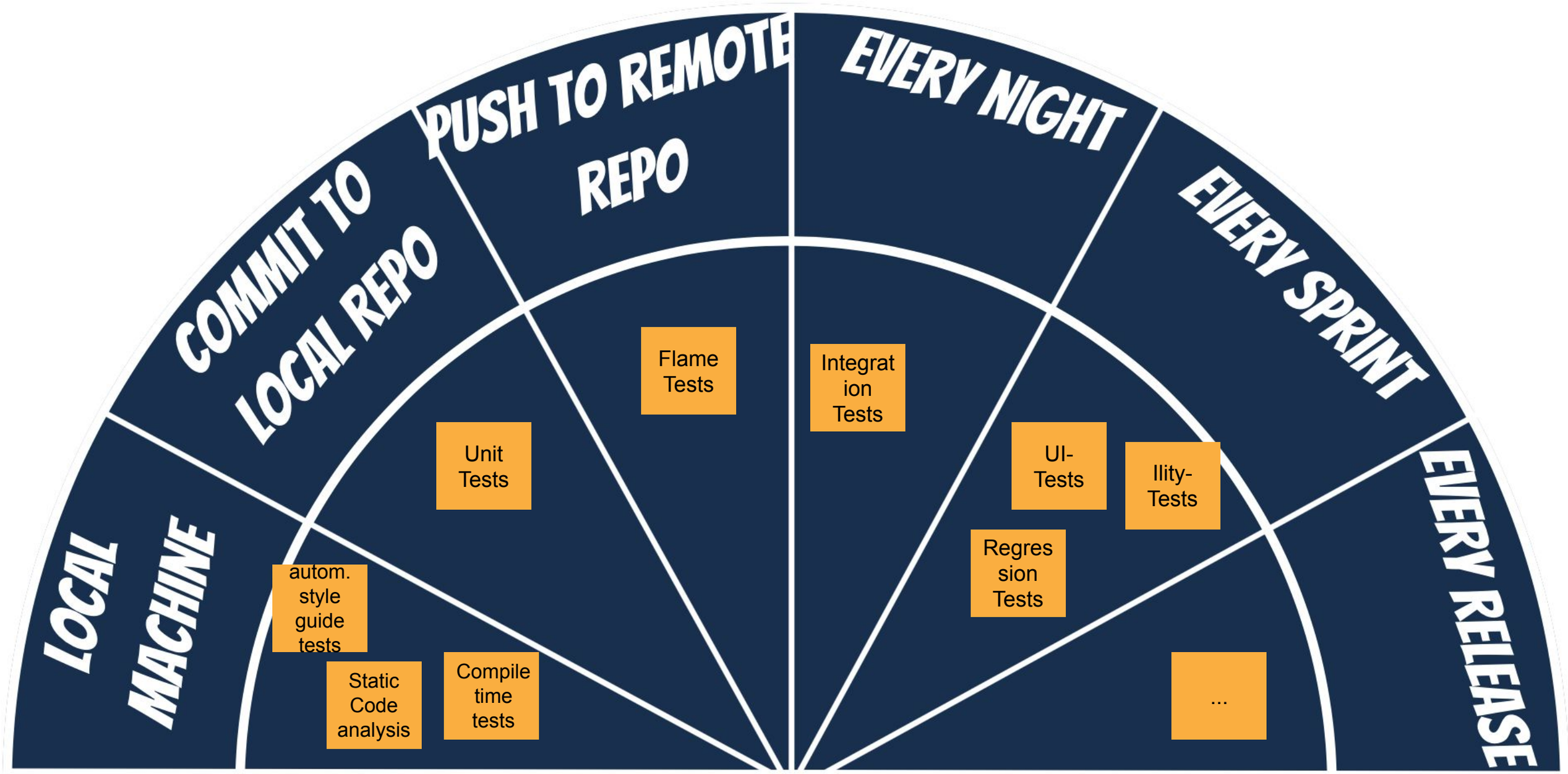
=> Automated Integration Testing

Need to be kept up-to-date.
Need to be run regularly.
=> Automated Unit testing – by
using JUnit… xUnit

Agile Test Radar



Agile Test Radar - just an example



Discussion & Conclusion

Given what we
know, have and want
- what do *we* think will help *us*
as the next step?