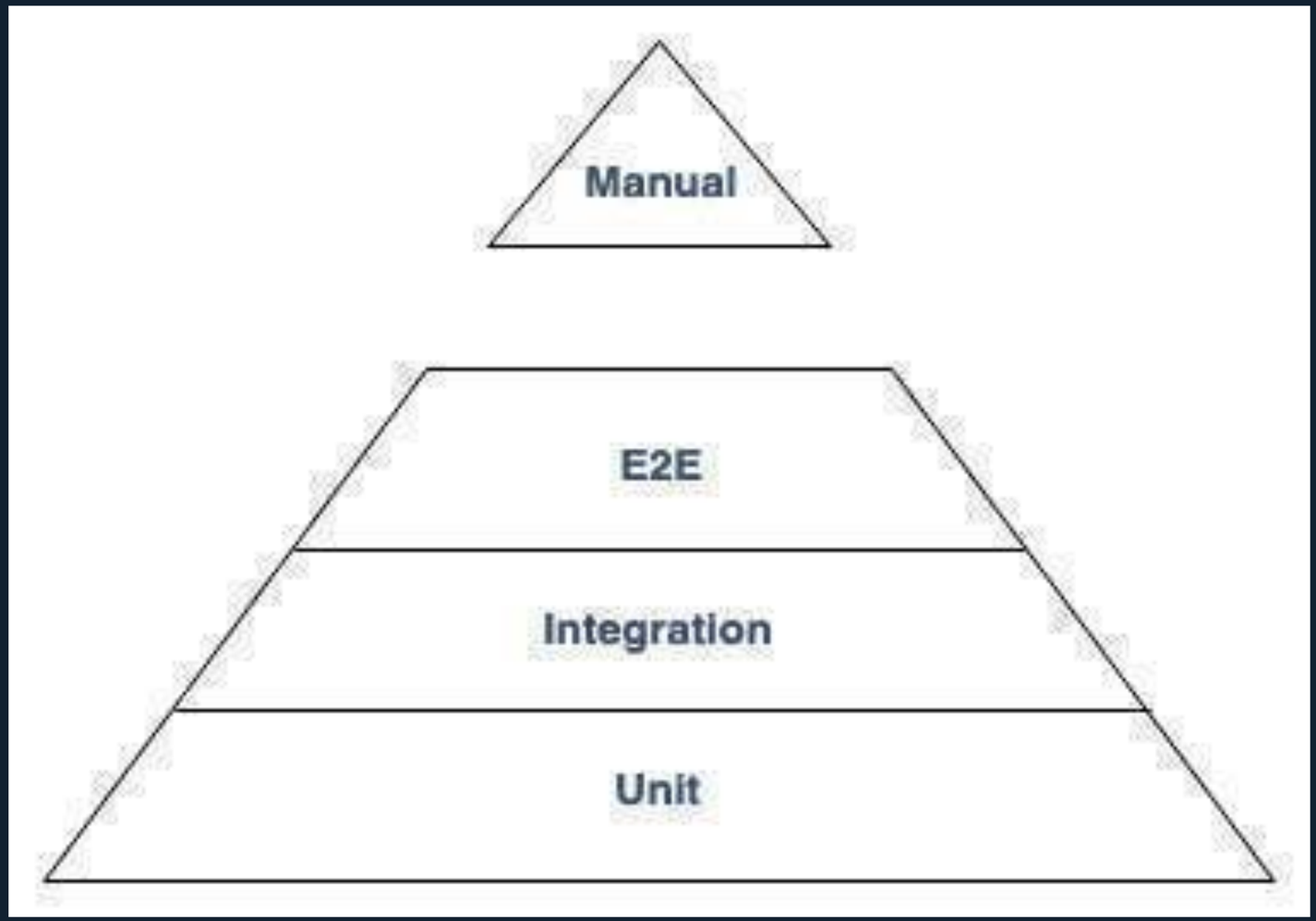
# CLIENT SIDE WEB ENGINEERING
# CLIENT SIDE TESTING

FHS (TMAYRHOFER.LBA@FH-SALZBURG.AC.AT)

# WHY AUTOMATED TESTING

- manual testing is expensive
  - long feedback loop between developer/testers
- helps to find and prevent defects
- allow to change a systems behavior predictably
- find/prevent defects
- gain confidence about quality of the software

# TESTING PYRAMID



# TESTING PYRAMID

- Unit tests
  - lots of small and isolated tests which are fast to execute
- Integration tests
  - some integration tests which test external systems like databases
- E2E
  - few tests which test the whole system

# UNIT TESTING AND TDD

- Test driven development (also known as TDD)
- Type of software development
- Introduced by Kent Beck
  - Author of Extreme Programming

# WHY TDD

- early/fast feedback during development
- Driving the design of our application
  - Testing is a side-effect
- Possibility to refactor
  - Confidence that app is still working
- Break down large problems into small problems
  - Think about edge cases

# TDD TO ME

“Helps me to break down bigger tasks into small steps I can keep in my head”

# TDD

“TDD doesn't drive good design. TDD gives you immediate feedback about what is likely to be bad design. (Kent Beck)
I want to go home on Friday and don't think I broke something. (Kent Beck)”

# WHAT IS TDD NOT

- Silver bullet for clean code
  - it eventually leads to better code
- Replacement for other testing strategies
  - TDD doesn't catch all bugs
  - Helps adding regression tests

# “THE BEST TDD CAN DO, IS ASSURE THAT THE CODE DOES WHAT THE DEVELOPER THINKS IT SHOULD DO. (JAMES GRENNING)”

# TDD INTRO IN 7:26 MINUTES

https://www.youtube.com/watch?v=WSes_PexXcA

# ESSENTIAL VS. ACCIDENTAL COMPLICATION

- Essential complication
  - The problem is hard
  - eg. Tax return Software
  - nothing we can do about
- Accidental complication
  - We are not so good in our job
  - eg. future proofing code

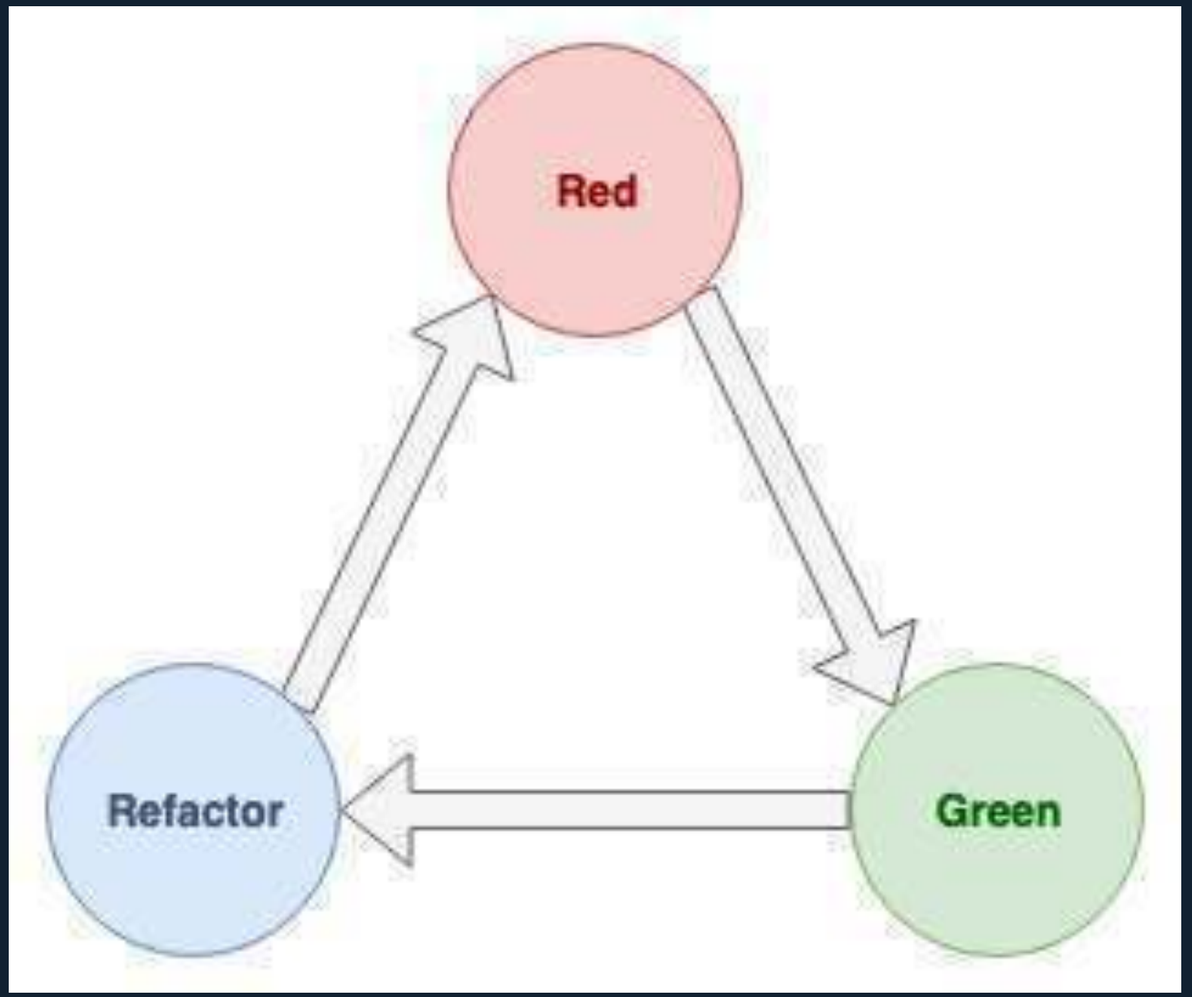
# ACCIDENTAL COMPLICATION

- future proofing code
- cutting corners
  - to get stuff out of the door
  - we're not going to change this anyways
- drives up the cost/development time of a feature
  - mostly the feature isn't complex
  - the way the app is built drives the cost of a

# AVOID ACCIDENTAL COMPLICATION

- Baby steps and TDD
- Refactor a little after every feature/green test
  - clean the kitchen
  - prevents big bang refactoring
  - which are hard to sell to business
- Without refactoring features will take longer

# TDD CYCLE



# TDD CYCLE

- Red: Write a test and watch it fail
- Green: Write just as much code to make the test pass
- Refactor: Clean up

# RED

» Think about the test description

» Descriptions should reflect the behaviour of the program

```
it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

# GREEN

» Write just enough code to make the test pass

  » if there is only 1 product just return 3$

```
function caluculatePrice () {
  return '3$'
};

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

# REFACTOR

» Change the code without changing any of the behaviour

» "Clean the kitchen"

```
const caluculatePrice = () => '3$'

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})
```

# RED (REPEAT)

```
» start with the next test-case

const caluculatePrice = () => '3$'

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product b given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})
```

# GREEN

» start with the next test-case

```
const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})
```

# RED

```
const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})

it('when unknown product is given, throws UnknownProductError', () => {
  expect(() => calculatePrice('productB')).toThrow(UnknownProductError)
})
```

# GREEN

```
class UnknownProductError extends Error {}

const caluculatePrice = (product) => {
  if (product === 'productA') { return '3$' }
  if (product === 'productB') { return '10$' }
  throw new UnknownProductError();
}

it('when product A given, price is 3$', () => {
  expect(calculatePrice('productA')).toEqual('3$')
})

it('when product B given, price is 10$', () => {
  expect(calculatePrice('productB')).toEqual('10$')
})

it('when unknown product is given, throws UnknownProductError', () => {
  expect(() => calculatePrice('productB')).toThrow(UnknownProductError)
})
```

# ANATOMY OF A TEST

» Arrange => test setup

» Aact => call the unit to test

» Assert => verify the result

# ANATOMY OF A TEST

```
it('employeeReport: returns a list of employees ordered by their name', () => {
  // Arrange
  const employees = [
    { name: 'Sepp' },
    { name: 'Max' },
    { name: 'Anton' }
  ]

  // Act
  const result = employeeReport(employees)

  // Assert
  assertThat(result, orderedBy((a, b) => a.name < b.name))
})
```

# WHAT MAKES A GOOD UNIT TEST

- Deterministic
  - randomness hard to test
  - current date or time is hard to test
- Tests should not have any effect on the system
  - changing global state (eg. database)
- no external systems are called
- little test setup

# CODE KATA

- Small exercise
  - to improve programming skills
  - by challenging your abilities
  - and encouraging you to find multiple approaches

# STEPS

» Step 1: Think

» Step 2: Write a test

» Step 3: How much does this test suck?

» Step 4: Run the test and watch it fail

» Step 5: Write just enough code to make it pass

» Step 6: Cleanup

# FIZZ BUZZ (20 MINUTES)

» go to http://tddbin.com/

» or git clone git@github.com:webpapaya/fhs-tdd-starter.git

» https://codingdojo.org/kata/FizzBuzz/

# ISOLATING UNITS UNDER TEST

# RECAP: WHAT MAKES A GOOD UNIT TEST

- Deterministic
  - randomness hard to test
  - current date or time is hard to test
- Tests should not have any effect on the system
  - changing global state (eg. database)
- no external systems are called

# DUMMY OBJECTS

- Objects which aren't used
  - so that the compiler doesn't complain
  - used to fill parameter lists
  - not relevant for JS but if you're using TS you might need these

```
class Person {
  constructor(public firstName: string, public age: number) {}
  isOfLegalAge(): boolean { return this.age >= 18 }
}

it('when age is 18, Person is of legal age', () => {
  const person = new Person("Max", 18)
  //                        ^^^^
  // this parameter would be a dummy object as it's
  // not relevant for the test

  // ...
})
```

# FAKE OBJECTS

- Objects have a working implementation
  - but take some shortcuts
  - eg. inMemoryDatabases instead of persistent DB
- example fake-local-storage

# STUB OBJECTS

- Predefined return values for testing
- Instead of calling the real API we return a value for testing
- Examples:
  - retrieving geolocation in tests
  - testing edge cases (eg.: database throws OutOfMemory exception)

```
const retrieveGPSPosition = () =>
  Promise.resolve({ lat: 12.12, lng: 14.15 })
```

# SPY OBJECTS

» Are stubs that also record the way they were called

» Used to verify side effects (eg. E-Mail sending)

```
it('sends an email on sign up', () => {
  const username = 'username'
  const password = 'password'

  const sendEmail = jest.fn() // create a spy
  const signUp = signUp(username, password, { sendEmail })

  expect(sendEmail).toHaveBeenCalledWith(username, password);
  //                ^^^^^^^^^^^^^^^^^^^^^
  // verify that the spy has been called
})
```

# CLOCK IN KATA (REST OF THE LECTURE)

» go to http://tddbin.com/

» http://kata-log.rocks/clock-in-kata

# REMEMBER STEPS

» Step 1: Think

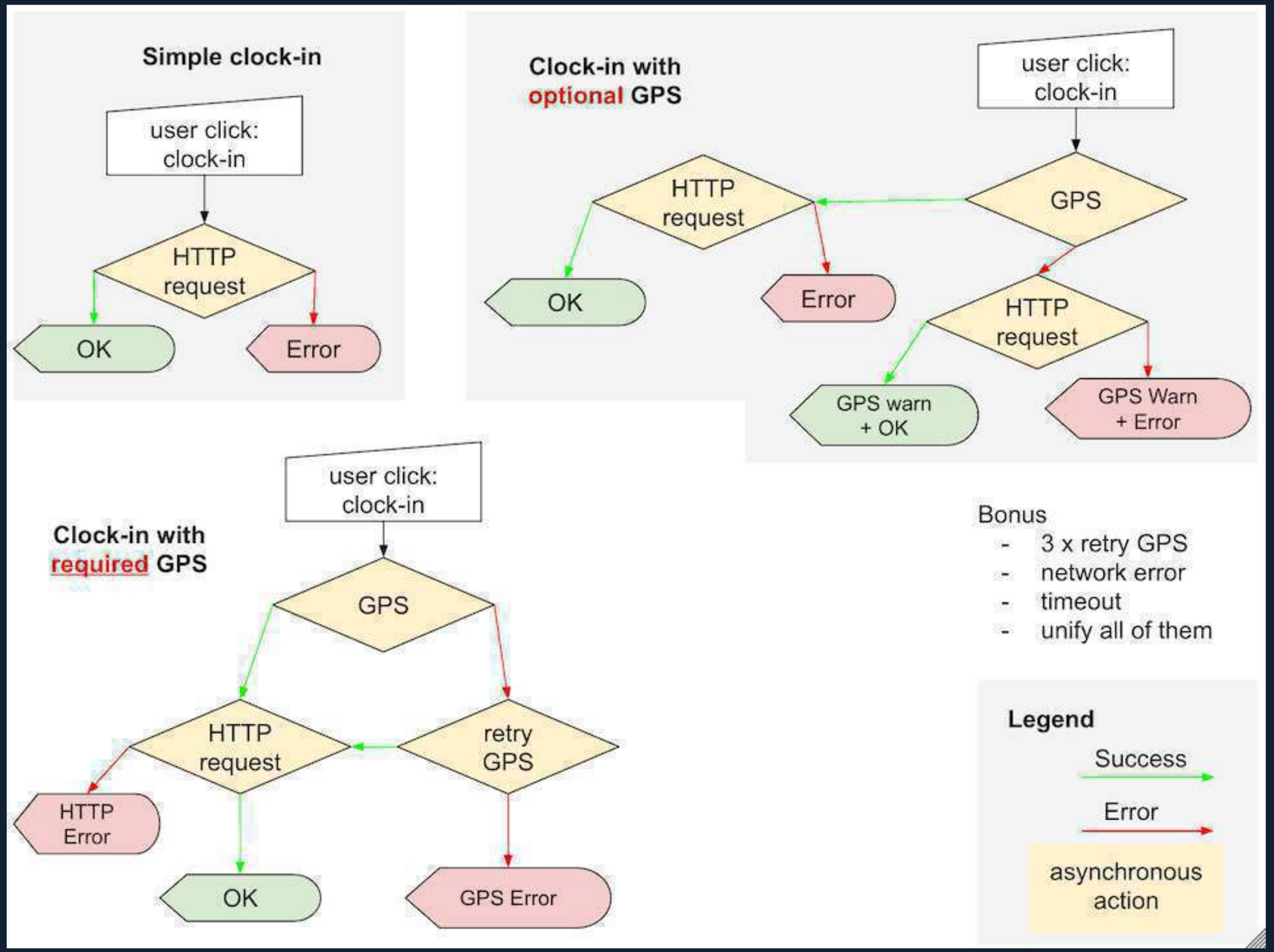
» Step 2: Write a test

» Step 3: How much does this test suck?

» Step 4: Run the test and watch it fail

» Step 5: Write just enough code to make it pass

» Step 6: Cleanup


Simple clock-in
user click: clock-in
HTTP request
OK
Error
Clock-in with optional GPS
user click: clock-in
GPS
HTTP request
OK
Error
HTTP request
GPS warn + OK
GPS Warn + Error
Clock-in with required GPS
user click: clock-in
GPS
HTTP request
retry GPS
HTTP Error
OK
GPS Error
Bonus
- 3 x retry GPS
- network error
- timeout
- unify all of them
Legend
Success
Error
asynchronous action

# HOMEWORK

- Homework:
  - can be done in pairs
  - implement https://codingdojo.org/kata/Bowling/
  - Work in a strict TDD loop
  - commit before every new test
  - write descriptive commit messages
  - try to implement as little as possible in one

# RESOURCES

- Integrated Tests are a Scam
- Is TDD Dead
- Mocks,Stubs,Spys
- Extreme Programming
- TDD