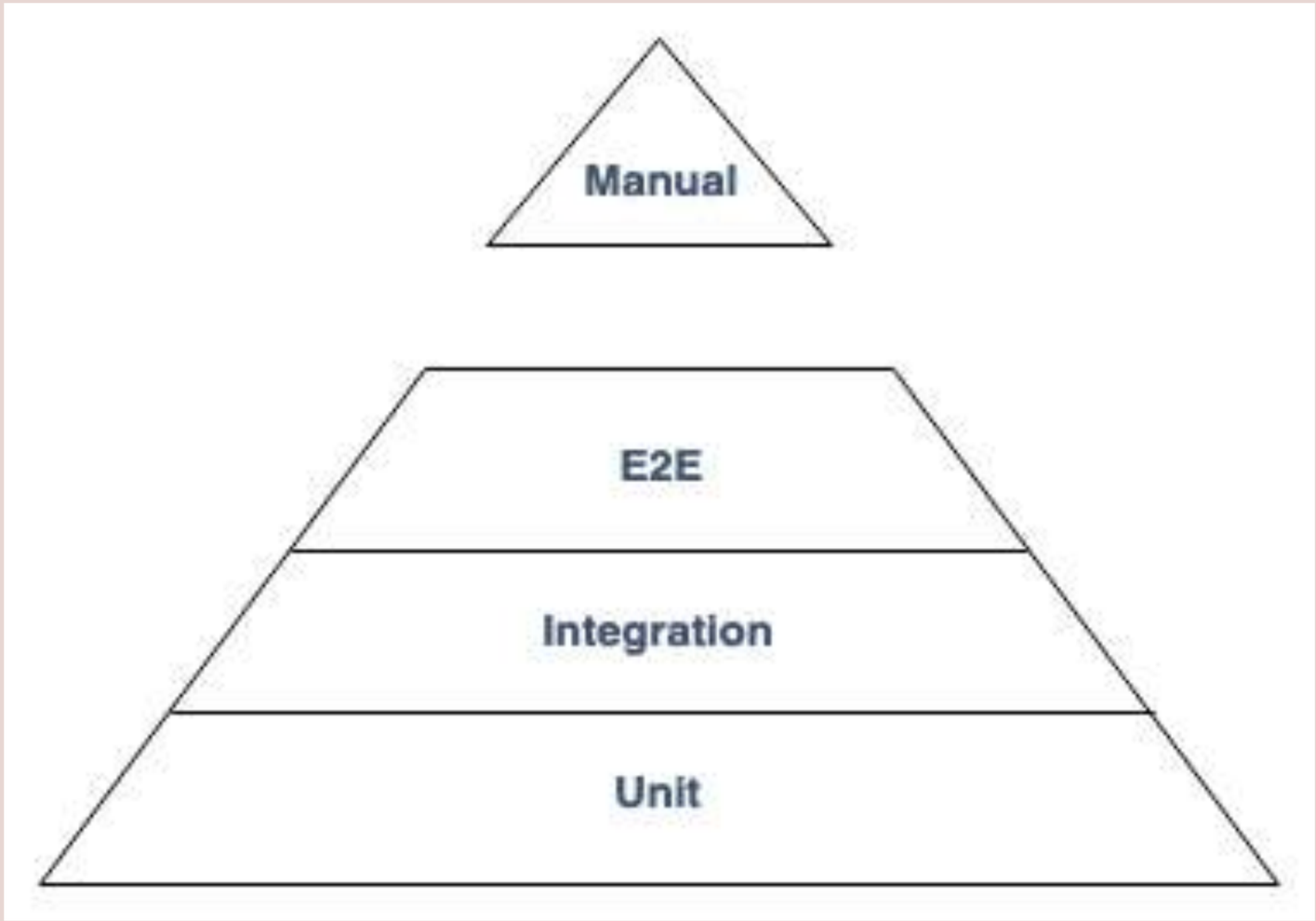
# CLIENT SIDE TESTING

# 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

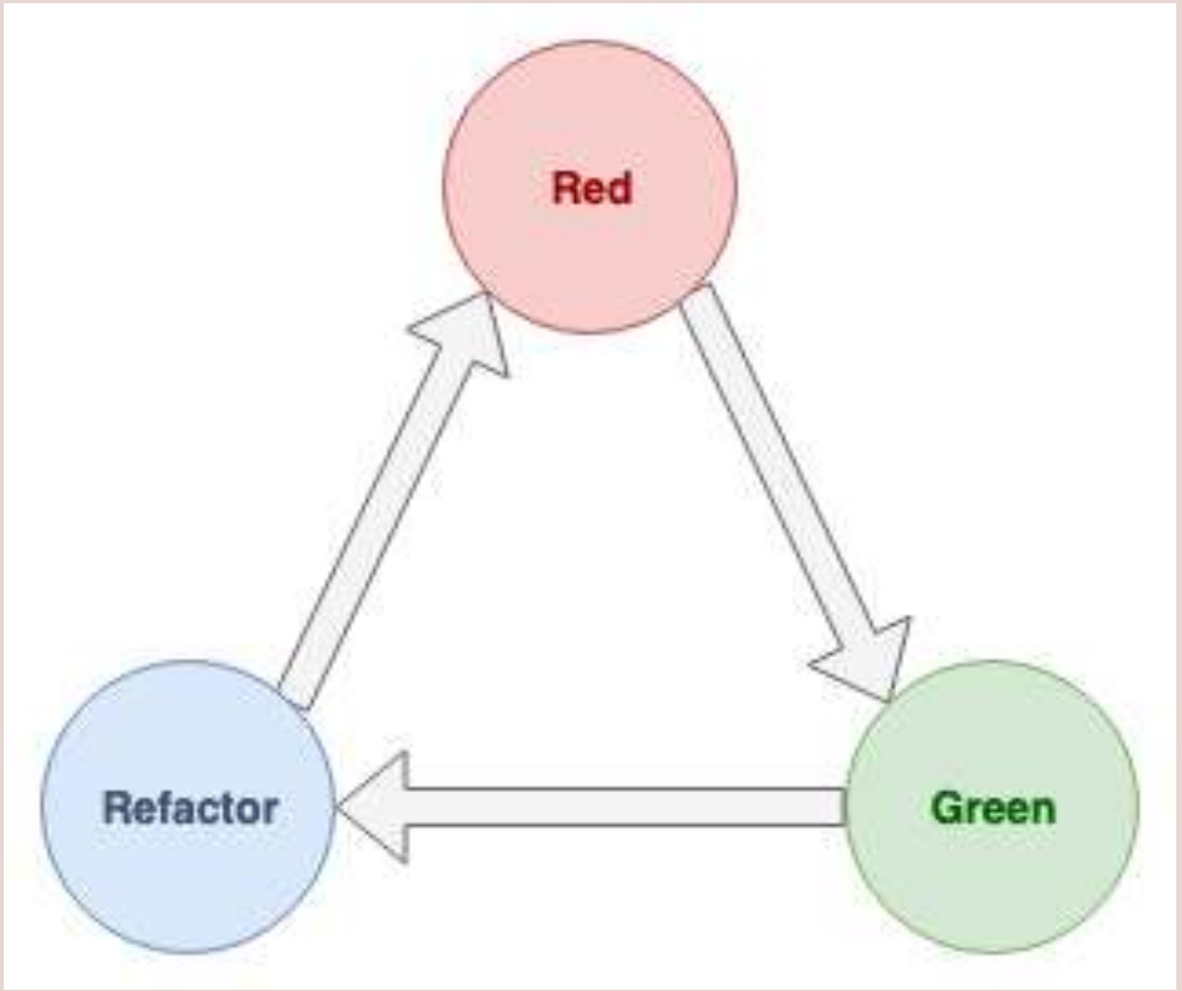
- Silver bullet for clean code
  - it eventually leads to better code
- Replacement for other testing strategies
  - TDD doesn't catch all bugs
  - Helps adding regression tests

# 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

# 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

# RESOURCES

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

# FEEDBACK

» Questions: tmayrhofer.lba@fh-salzburg.ac.at

» https://s.surveyplanet.com/x1ibwm85