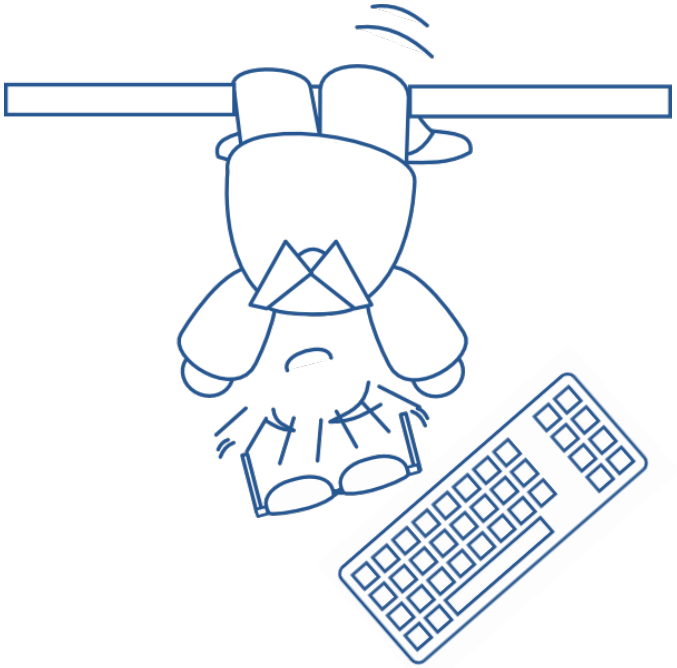
Coding backwards
in order to
to think straight



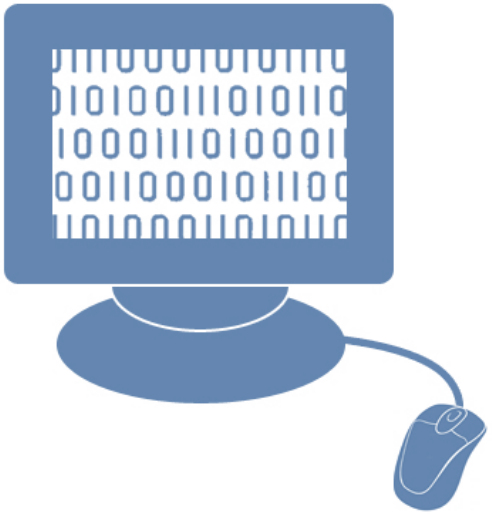
An initiation to

TDD

(Test Driven Development)

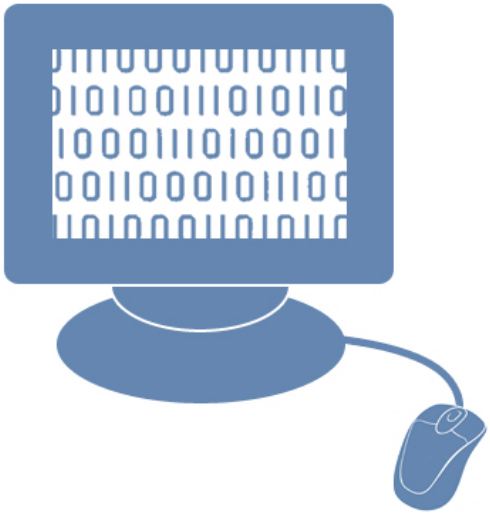
So, TDD ...

... what is that?



It is a practice ...

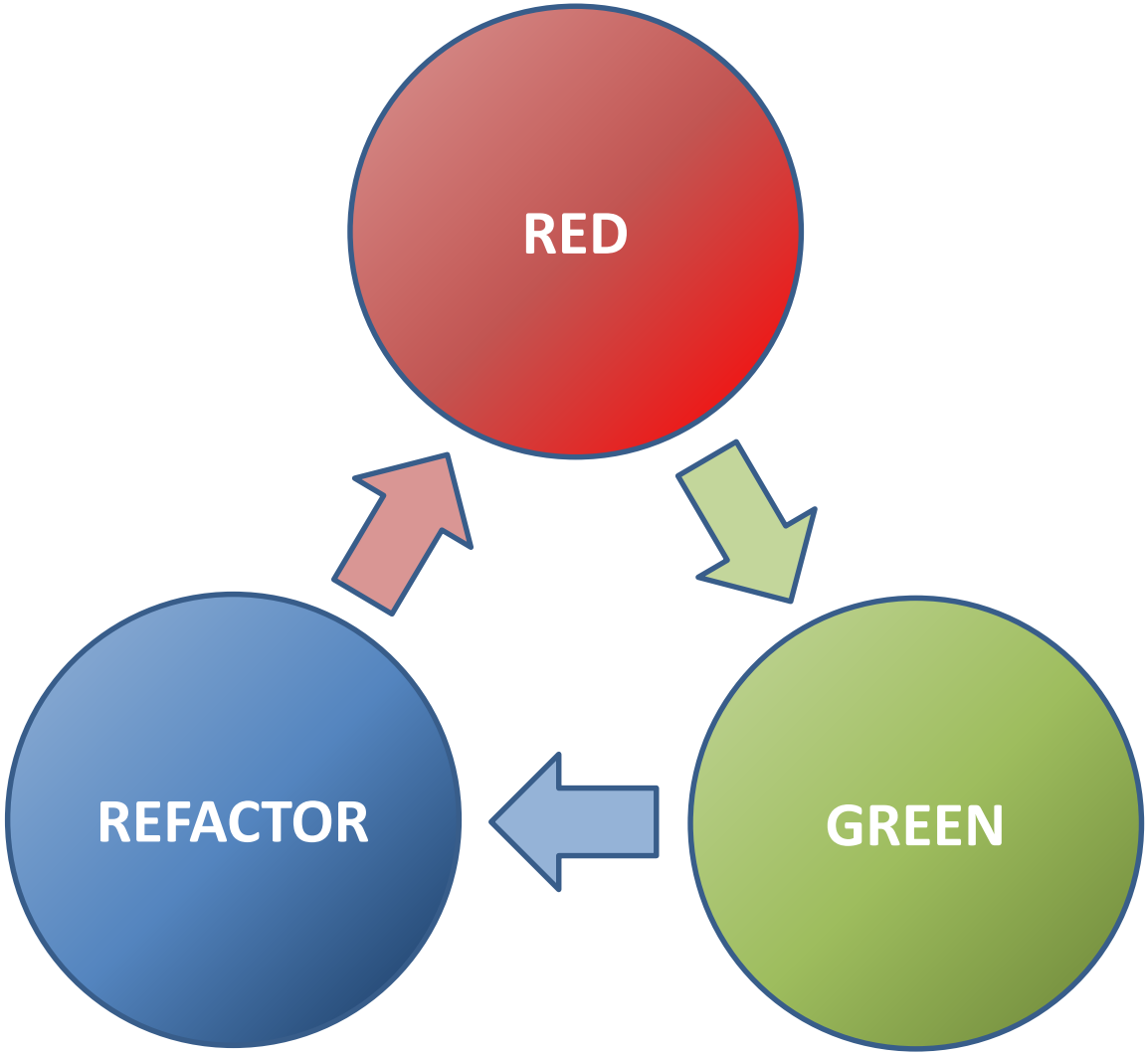
... a development one

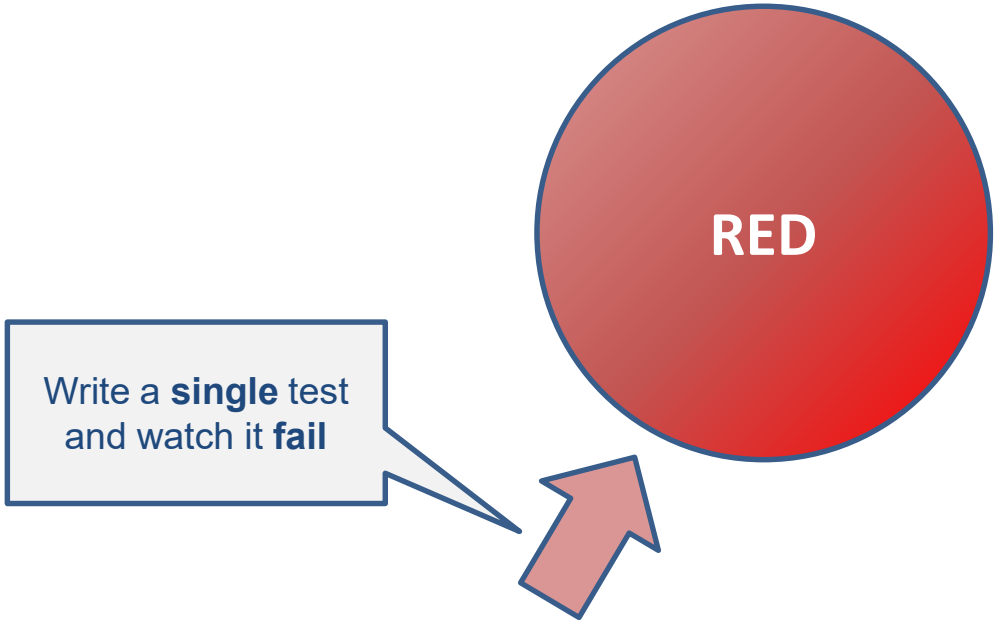


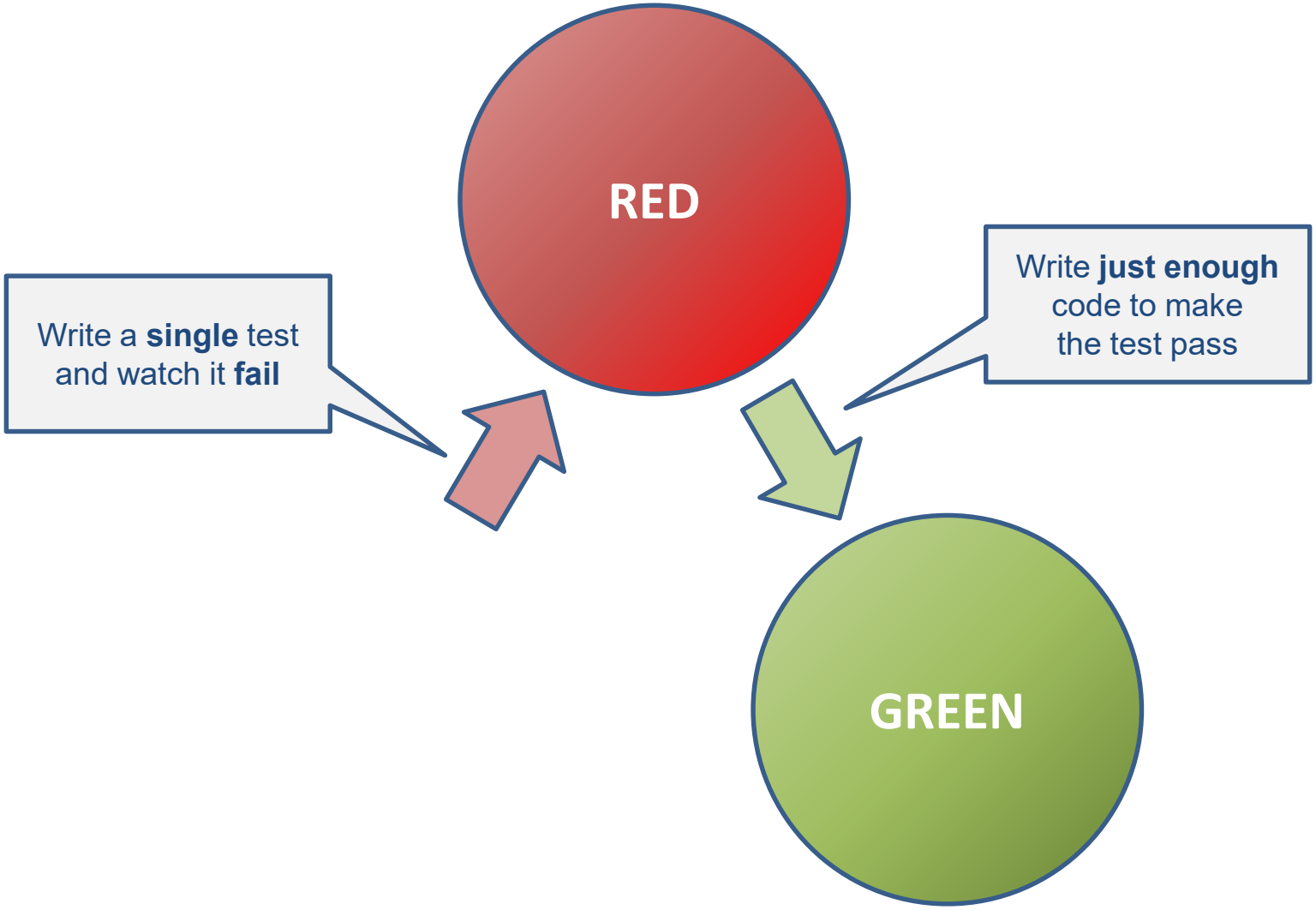
It is a practice …

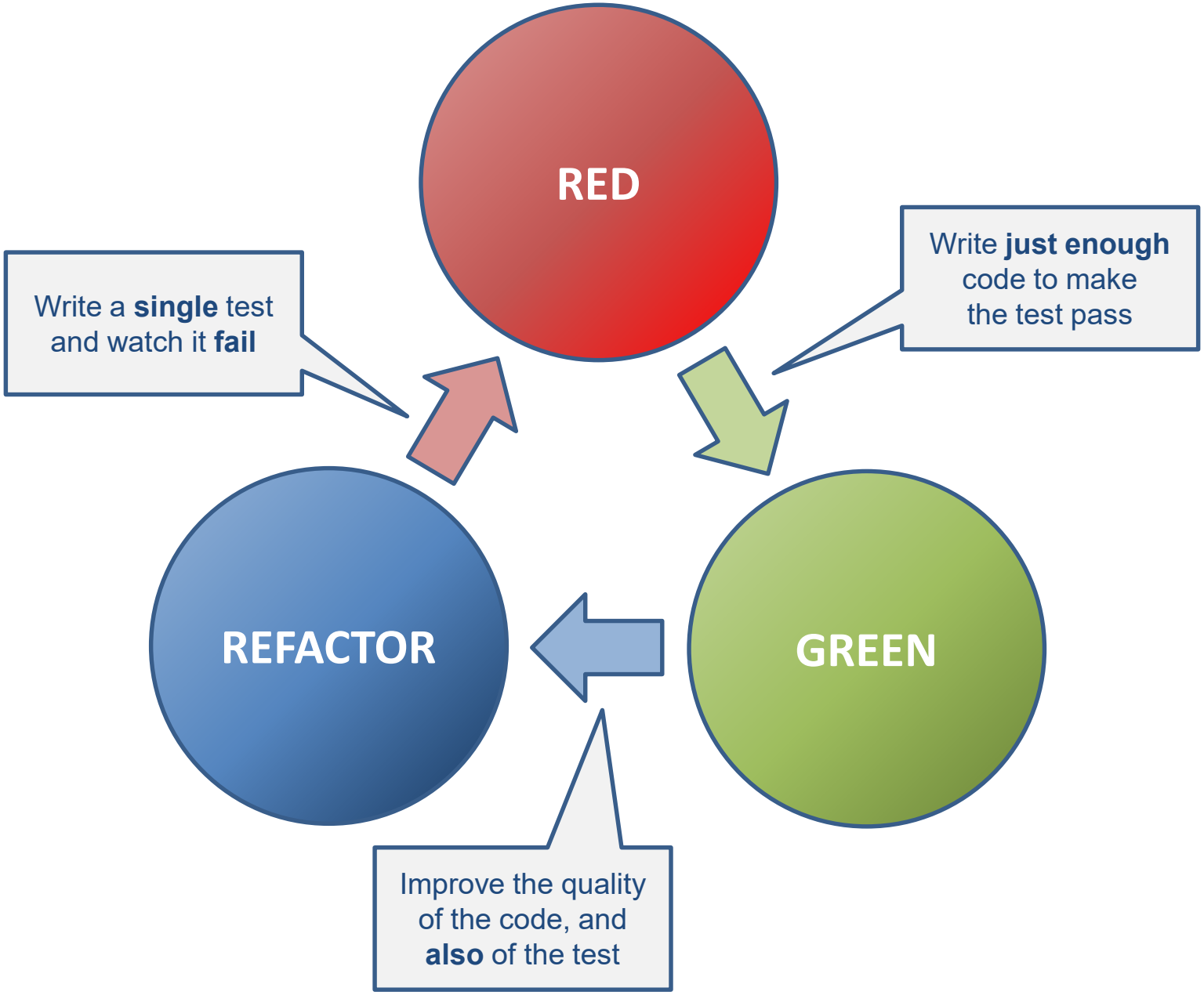
… a development one

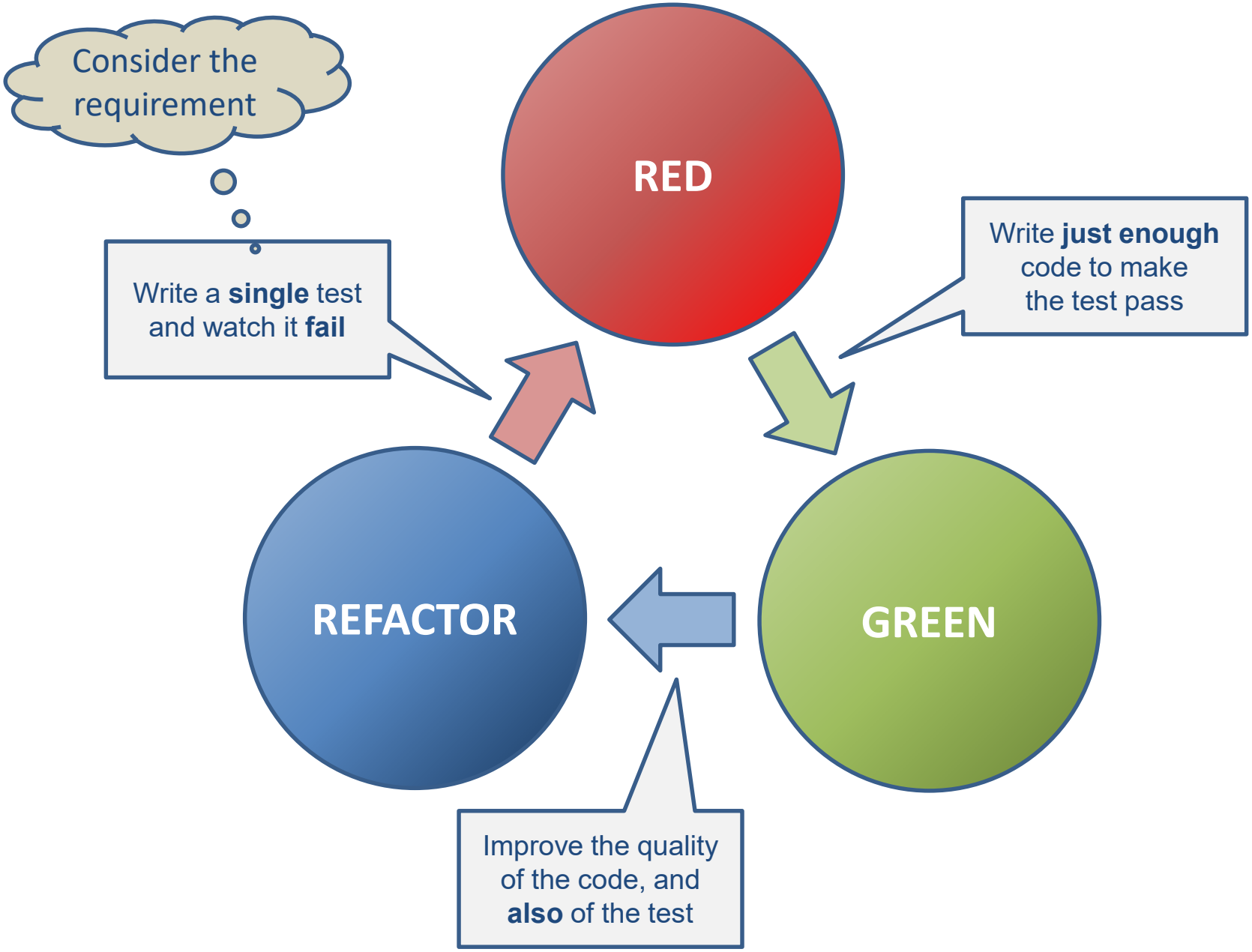
- **It is <u>not</u> a testing practice:**
  - **Its purpose is to create production code**
  - **Test coverage is a welcome side-effect**
- **It is but one of XP practices (which are complementary)**
- **It is only a practice, not a silver bullet, not a dogma**

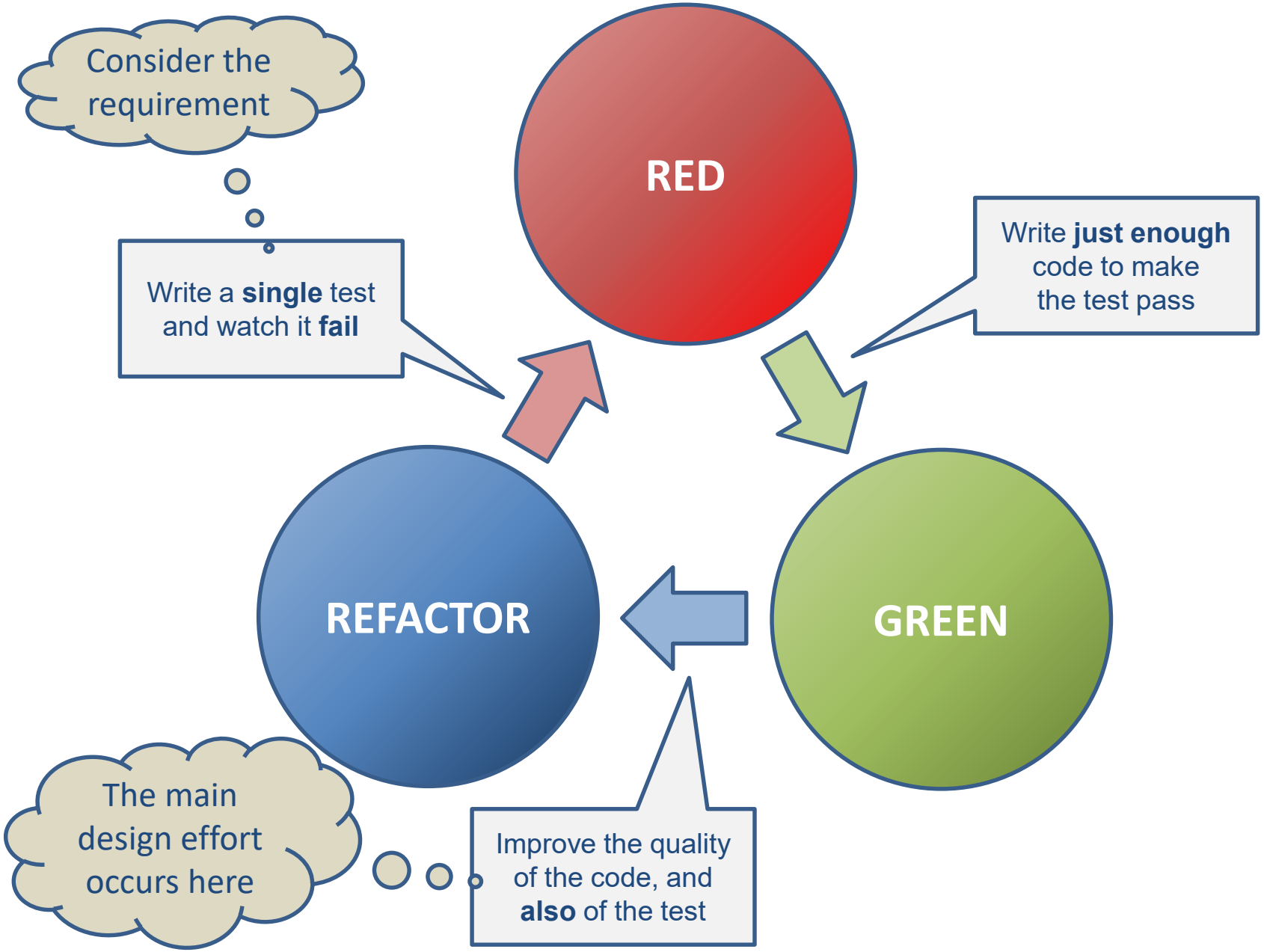

RED
GREEN
REFACTOR


RED
Write a **single** test
and watch it **fail**


RED
Write a **single** test and watch it **fail**
Write **just enough** code to make the test pass
GREEN


RED
Write **just enough** code to make the test pass
GREEN
Improve the quality of the code, and **also** of the test
REFACTOR
Write a **single** test and watch it **fail**


Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
Improve the quality of the code, and **also** of the test
REFACTOR


Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
REFACTOR
The main design effort occurs here
Improve the quality of the code, and **also** of the test

Okay, but ...

... to what end?

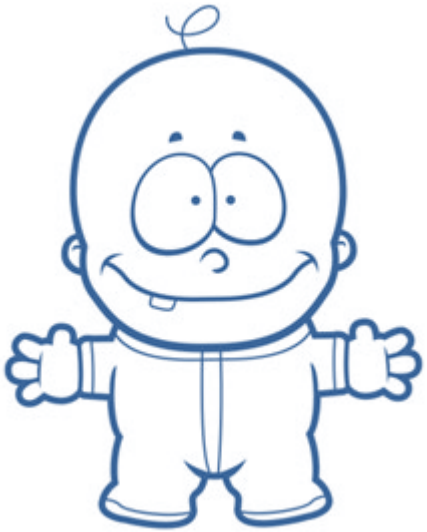
Higher Quality

Higher Quality

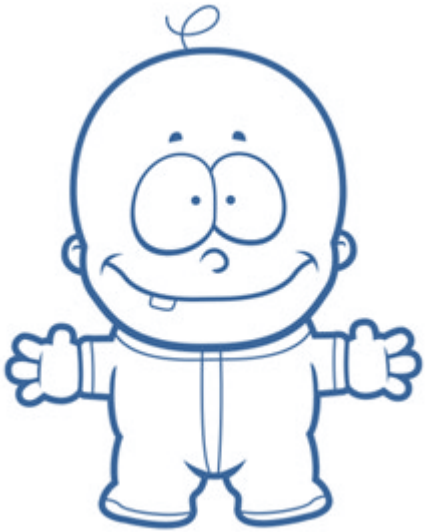
- **Users and business satisfaction**
- **Product relialibility and robusteness**
- **Tolerance to change**
- **Building trust and confidence**
- **Continuous improvement cycle**

Okay, but then ...

... why is it so uncommon?



An Emerging Profession



An Emerging Profession

- **Very little hindsight on the different practices**
- **School curricula are evolving… at their own pace**
- **A high overall complexity of the variables involved**

A « Magnifying » Effect

A « Magnifying » Effect

- **Colliding with existing code can be painful**
- **Discovering problems: design, requirements, organization…**
  - **Accepting these problems and improving**
  - **Not blaming the practice for what it reveals**

Easy from a distance...

... far from being easy

Easy from a distance...

... far from being easy

- **Changing work habits to learn TDD may actually feel like the "Backwards Brain Bicycle" experiment**
- **Conforming to a constrained cycle can be frustrating**
- **Understanding before coding is not that simple**
- **Learning takes time, so there is a cost – and someone has to pay for this, but (usually) no one wants to**

Let’s take a step back

Software is eating the world

Software is eating the world

- **Software is ubiquitous in our personal lives, our society, industry and government**
- **The Fourth Industrial Revolution is upon us, bringing together digital, biological, and physical technologies**
- **Technology becomes embedded within societies and even the human body**

There seems to be a cost

There seems to be a cost

- **Legacy systems, massive failures and 9-digit defects, troubled and cancelled projects, technical debt…**
- **The cost of poor quality software in the US in 2018 is approximately $2.26 trillion**
- **About 60% of the US software engineering work time centers on finding and fixing errors**

So, what can we do?

So, what can we do?

- **Find and fix problems and deficiencies as close to the source as possible**
- **Prevent them from happening in the first place**
- **High-quality software is faster and cheaper to build and maintain than low-quality software**

Okay, but ...

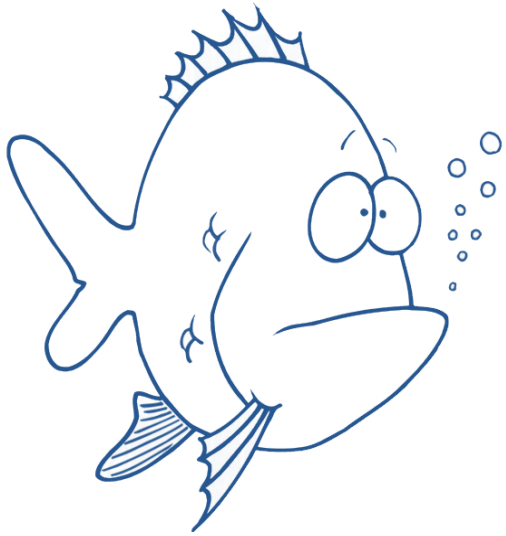
... how does TDD help in all this?

Hacking our brains

Hacking our brains

Evolution left us with many cognitive biases:

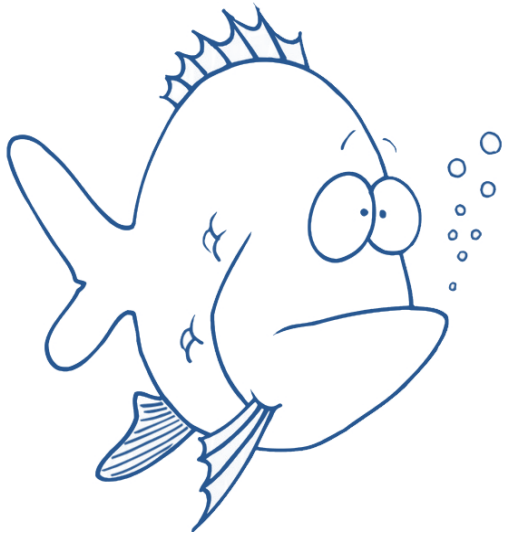
- **Too much information (eg. Confirmation Bias)**
- **Limits of memory (eg. Testing Effect)**
- **Lack of meaning (eg. Bandwagon Effect)**
- **Acting in urgency (eg. Loss Aversion)**



Hacking our brains

versus

Too much information



Hacking our brains

versus

Too much information

TDD is more fit to our attention span:

- **Feedback happening every few seconds**
- **A framework to focus on**
- **Milestones over time and progress**

Hacking our brains

versus

Limits of memory

Hacking our brains

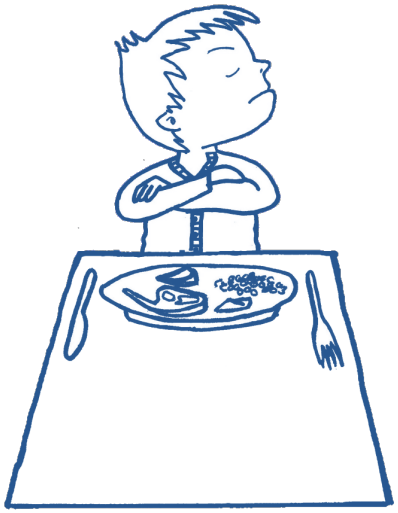
versus

Limits of memory

TDD helps with the learning process:

- **Trial and error**
- **Repetition**
- **Solution emergence**

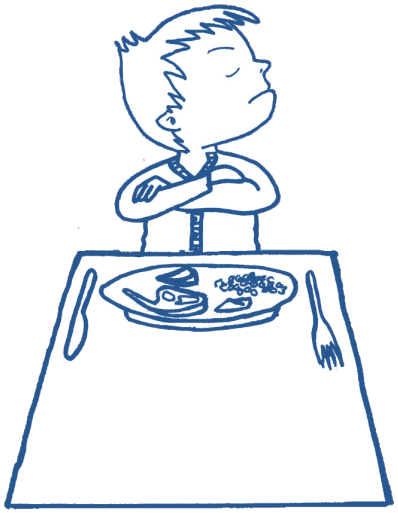
} **Continuous Improvment**



Hacking our brains

versus

Lack of meaning



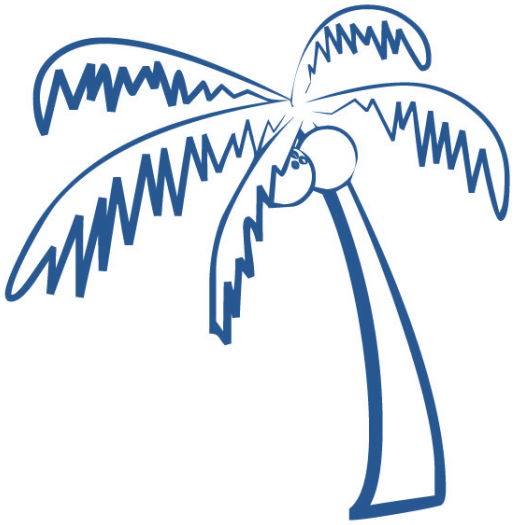
Hacking our brains

versus

Lack of meaning

TDD acts as an operant conditioning process:

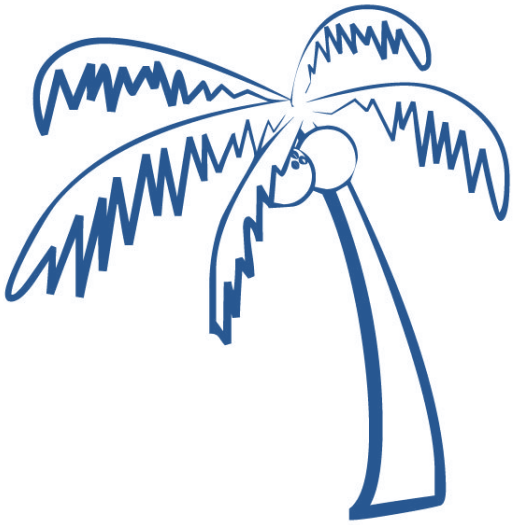
- **« Green » highlights each achievement**
- **« Red » stresses incompleteness**
- **Confidence increases with each cycle**



Hacking our brains

versus

Acting in urgency



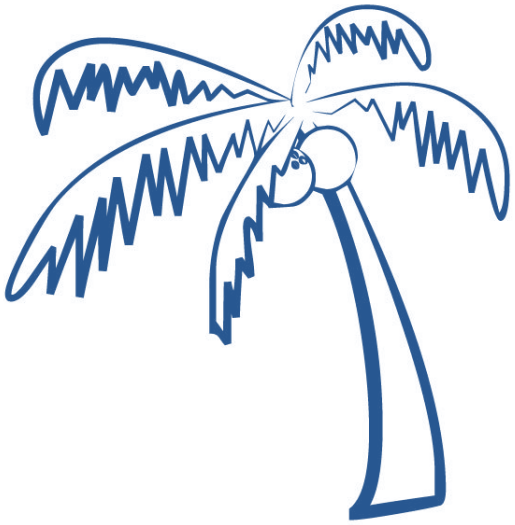
Hacking our brains

versus

Acting in urgency

TDD helps with our natural tendency to procrastinate:

TODO



Hacking our brains

versus

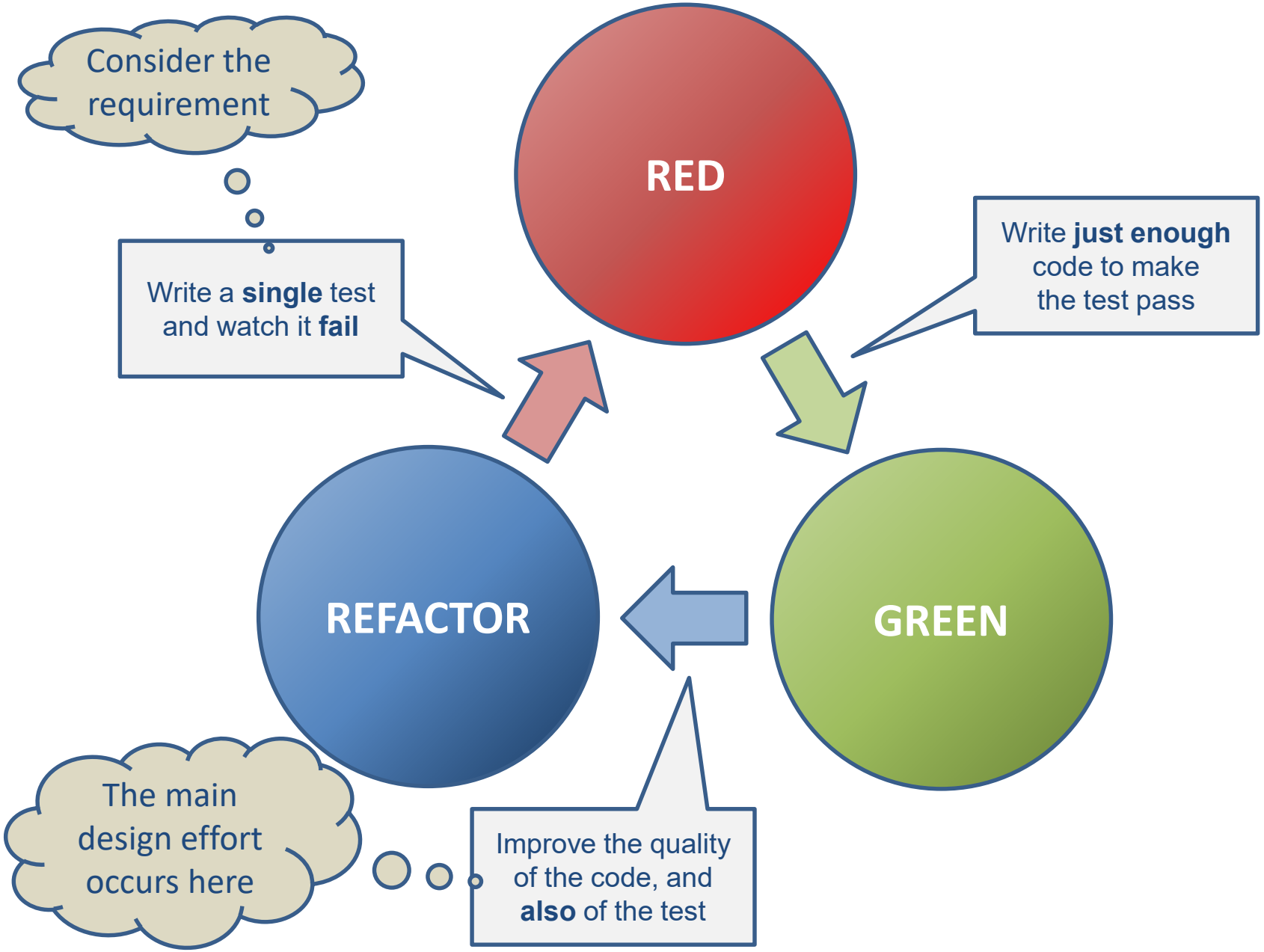
Acting in urgency

TDD helps with our natural tendency to procrastinate:

- **An iterative process, providing steps for progress**
- **Tests cannot be forgotten or postponed**

Let's get back to the point ...

... TDD is therefore:


Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
REFACTOR
The main design effort occurs here
Improve the quality of the code, and **also** of the test

Okay, but, speaking of design...

... what is good design?

Three main categories of Design

Three main categories of Design

- **Architecture**
- **Macro design**
- **Micro design**

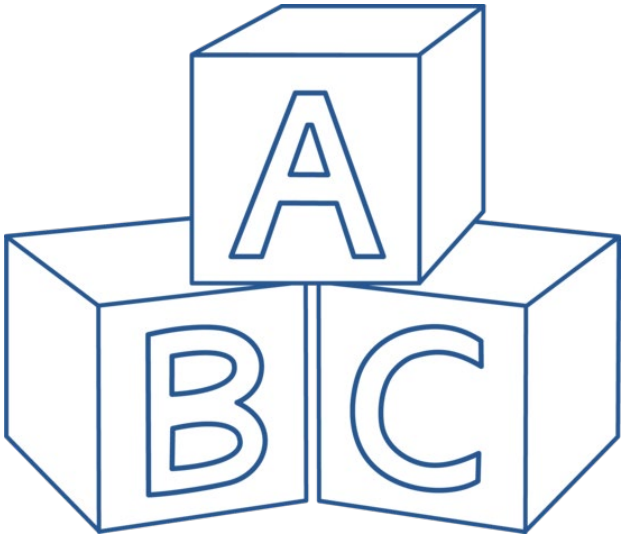
Three main categories of Design

- **Architecture** = **infrastructure, persistence ...**
- **Macro design** = **packages, dependencies ...**
- **Micro design** = **business code, rules ...**

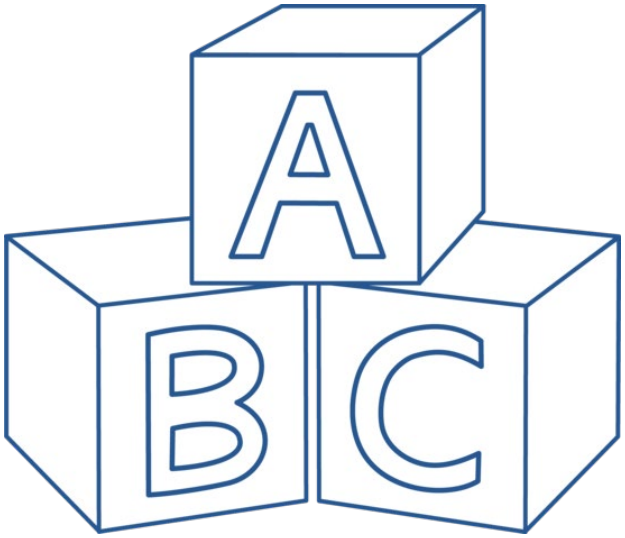
Three main categories of Design

- **Architecture** = infrastructure, persistence ...
- **Macro design** = packages, dependencies ...
- **Micro design** = business code, rules ...

TDD

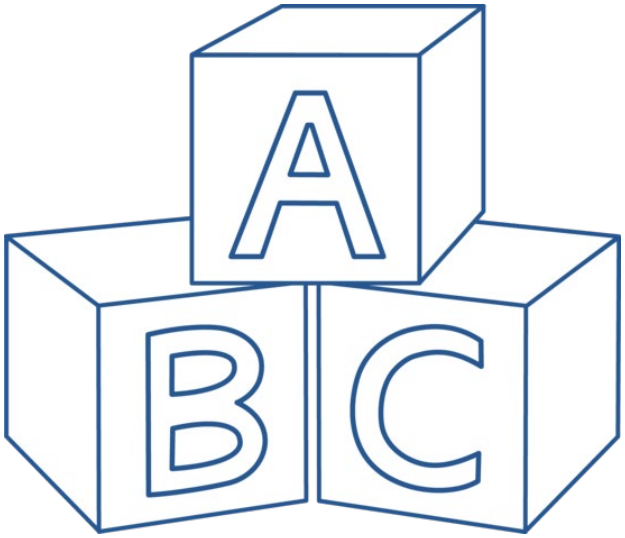


Four Rules of Simple Design



Four Rules of Simple Design

- ✓ **Passes the tests**
- ✓ **Reveals intention**
- ✓ **No duplication**
- ✓ **Fewest elements**



Four Rules of Simple Design

- ✓ **Passes the tests**
- ✓ **Reveals intention**
- ✓ **No duplication**
- ✓ **Fewest elements**

Okay, but ...

... that's not enough!

It's a long way to go

It's a long way to go

- **Design Patterns**
- **SOLID Principles**
- **Clean Code**
- **Domain Driven Design**

...

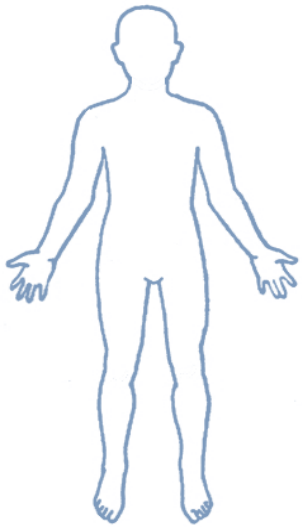
So ...

... how do we get started ?

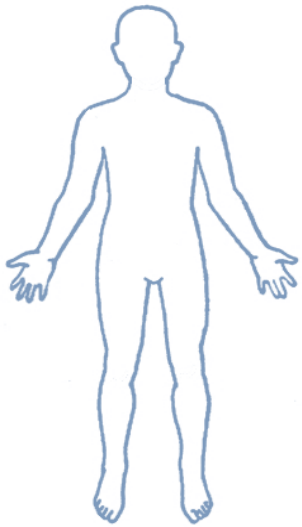
Some guidelines for the TDD cycle

Some guidelines for the TDD cycle

1. **Add one (and only one) new test while in « Green »**
2. **Watch the test fail before coding a solution to make it pass**
3. **Code in order to return as soon as possible to « Green »**
4. **Refactor code or test at any one time, not both**



Anatomy of a Test

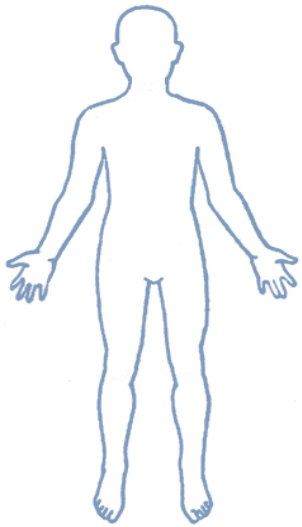


Anatomy of a Test

GIVEN

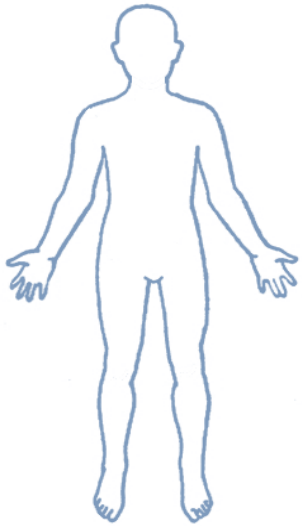
WHEN

THEN



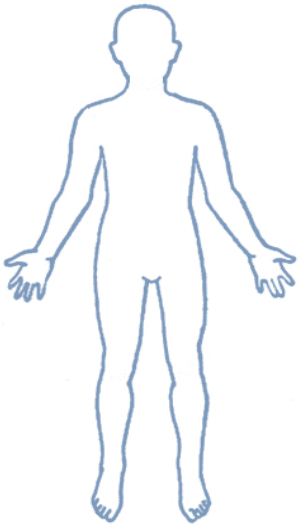
Anatomy of a Test

GIVEN	**Context**
WHEN	**Event**
THEN	**Expectation**



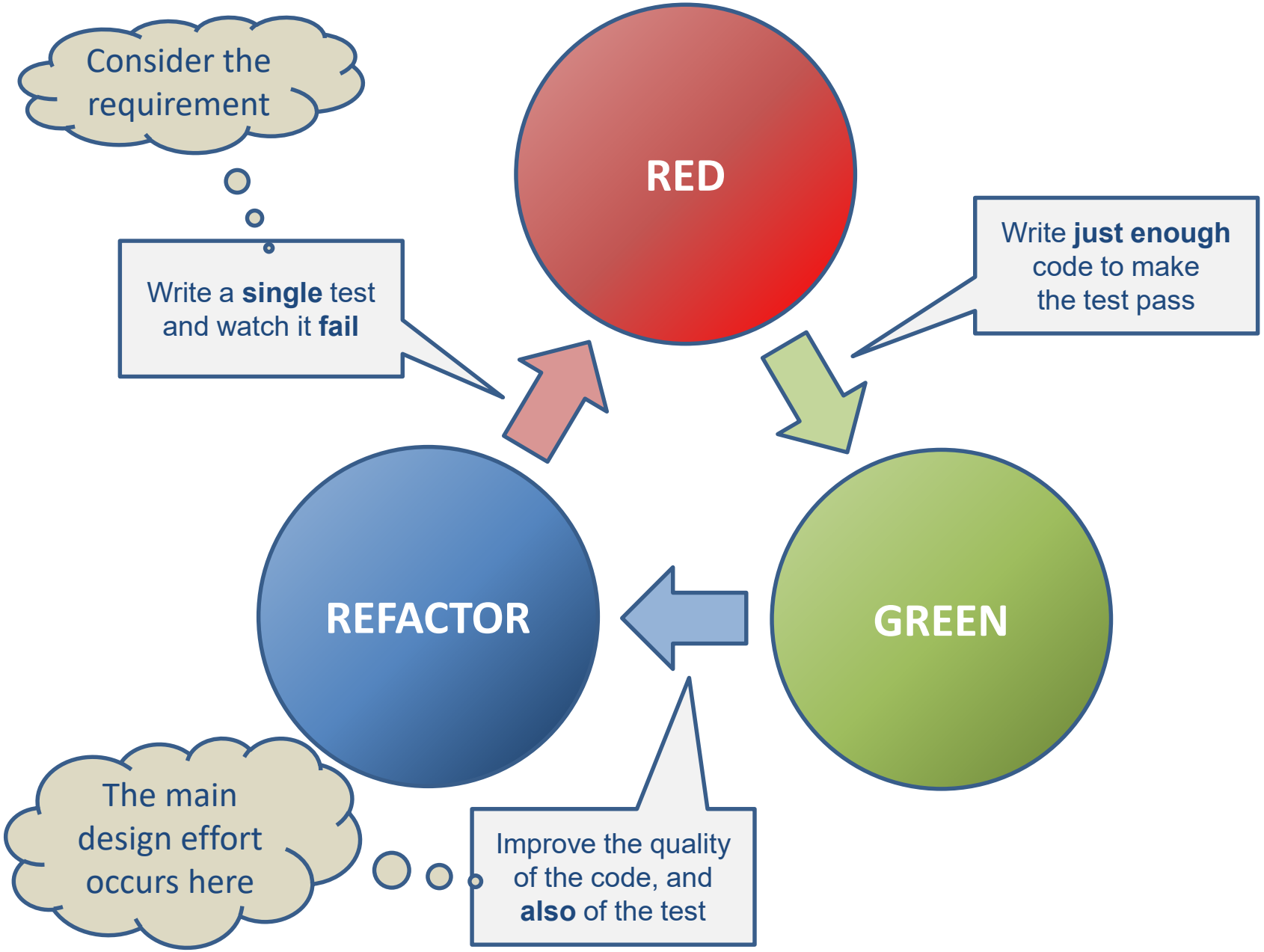
Anatomy of a Test

GIVEN	Context	= states / data
WHEN	Event	= what is being tested
THEN	Expectation	= solution to the requirement



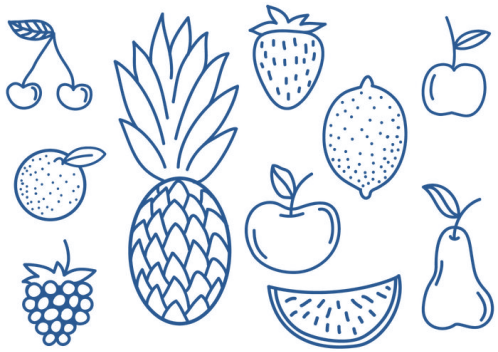
Anatomy of a Test

3	GIVEN	Context	= states / data
2	WHEN	Event	= what is being tested
1	THEN	Expectation	= solution to the requirement


Consider the requirement
Write a **single** test and watch it **fail**
RED
Write **just enough** code to make the test pass
GREEN
REFACTOR
The main design effort occurs here
Improve the quality of the code, and **also** of the test

In the end, the key is ...

... practising!



Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

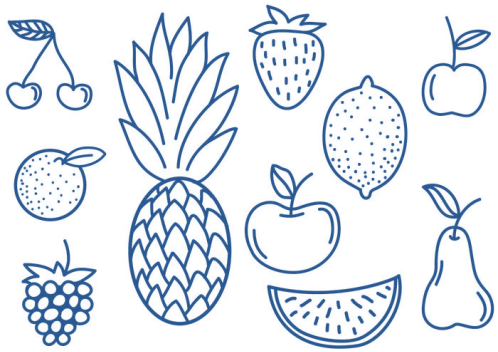
- **1 apple costs 100**
- **1 banana costs 150**
- **1 cherry costs 75**

```
@Test
public void noCheckoutForEmptyCart() {

}
```

```java
@Test
public void noCheckoutForEmptyCart() {

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

```java
@Test
public void noCheckoutForEmptyCart() {

    // WHEN
    int totalAmount = cart.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

```java
@Test
public void noCheckoutForEmptyCart() {

    // GIVEN
    Cart cart = new Cart();

    // WHEN
    int totalAmount = cart.computeTotalAmount();

    // THEN
    assertThat(totalAmount).isEqualTo(0);
}
```

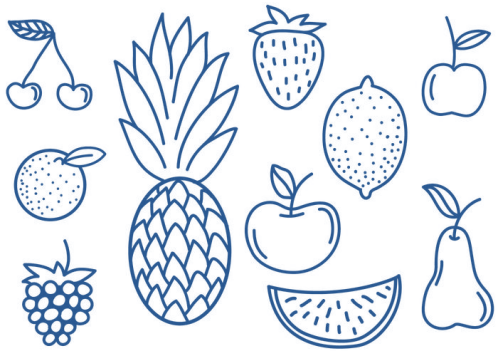


Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

- **1 apple costs 100**
- **1 banana costs 150**
- **1 cherry costs 75**

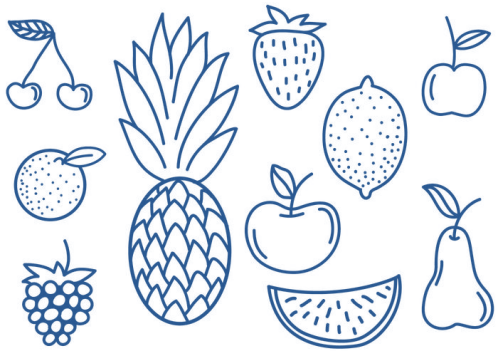


Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **1 cherry costs 75**

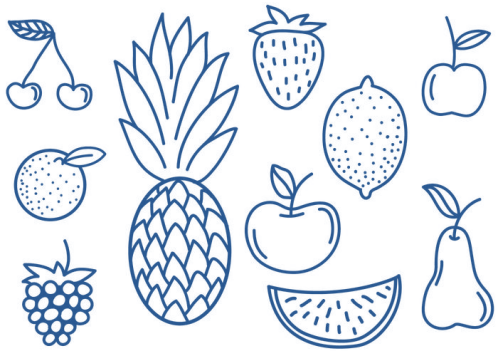


Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **1 cherry costs 75**
- **You have to be able to sell « des pommes » in France**

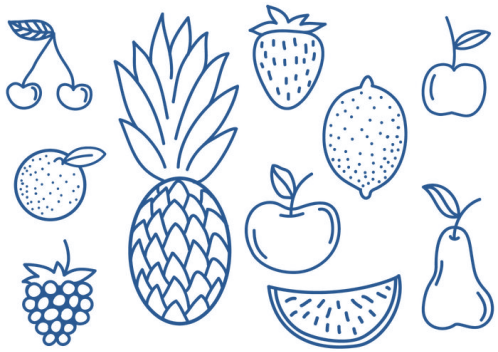


Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **The second banana is half price**
- **1 cherry costs 75**



Fruit Shop

Charge the right amount when the customer goes to the checkout.

Business rules (prices in cents):

- **1 apple costs 100**
- **1 free apple when two bought apples**
- **1 banana costs 150**
- **The second banana is half price**
- **1 cherry costs 75**
- **A loyalty program customer is entitled to a 10% discount**


THANK
YOU