Software craftmanship

Emmanuel CHAFFRAIX

15/05/2020

Sotfware Craftmanship Manifesto

Not only working software,
but also **well-crafted software**
Not only responding to change,
but also **steadily adding value**
Not only individuals and interactions,
but also **a community of professionals**
Not only customer collaboration,
but also **productive partnerships**

Source: http://manifesto.softwarecraftsmanship.org/

Summary – Clean Code

Technical Debt

Acronyms : DRY, KISS, YAGNI, DIE

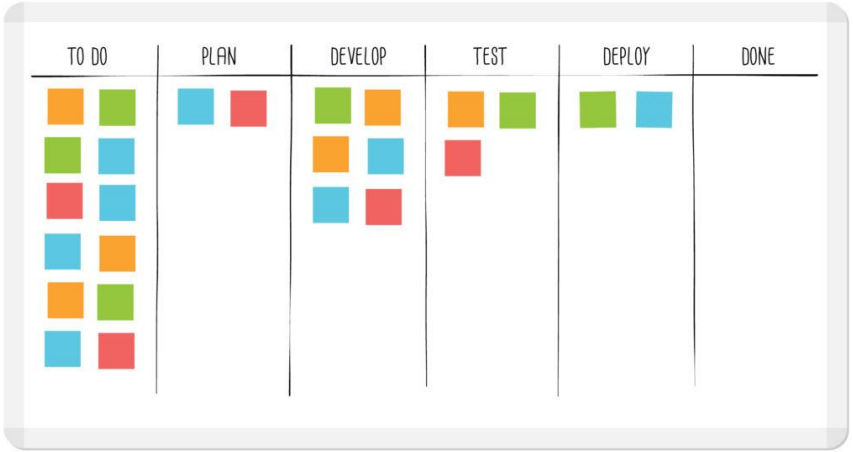
OO and SOLID Principles

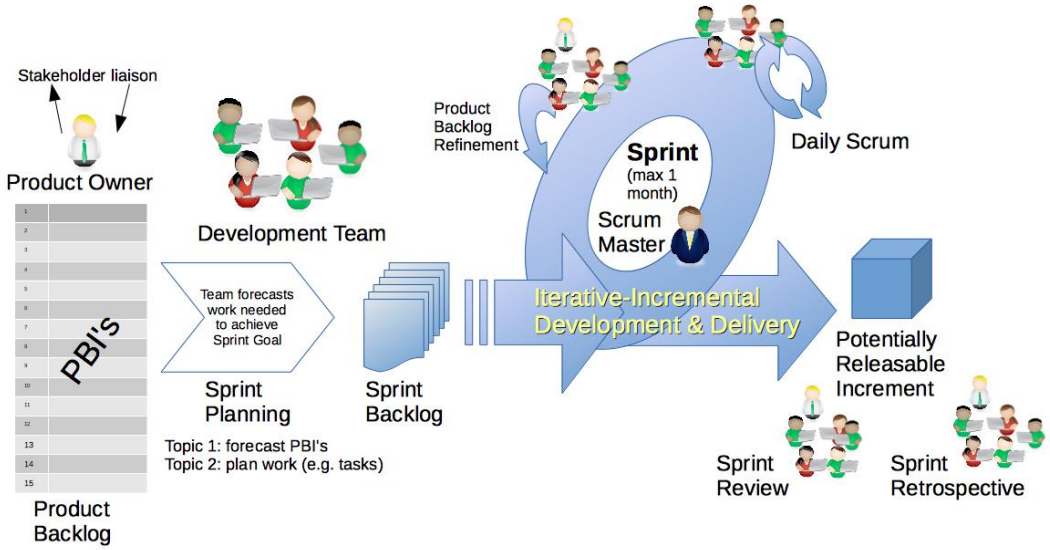
Summary – Be Agile

From V-Cycle to Agile : Build – Measure – Learn

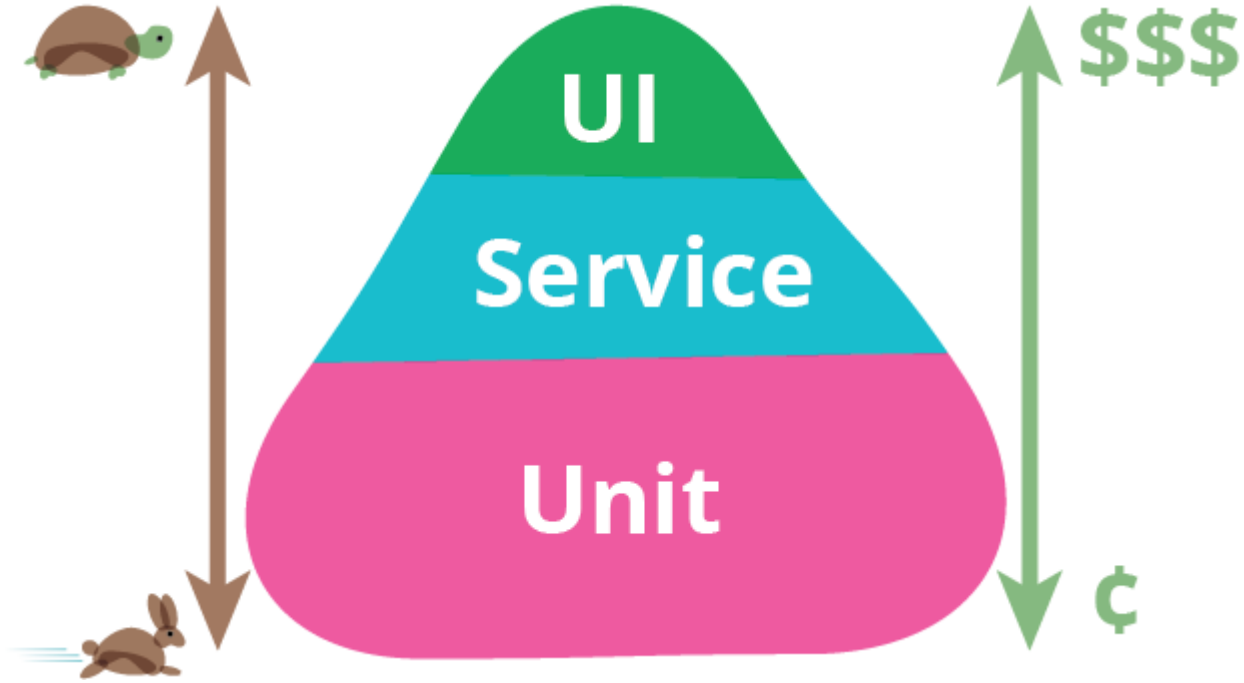
SCRUM : Backlog, Sprint, Product Owner, Scrum Master

KANBAN : Kanban Board, column limits





The test pyramid

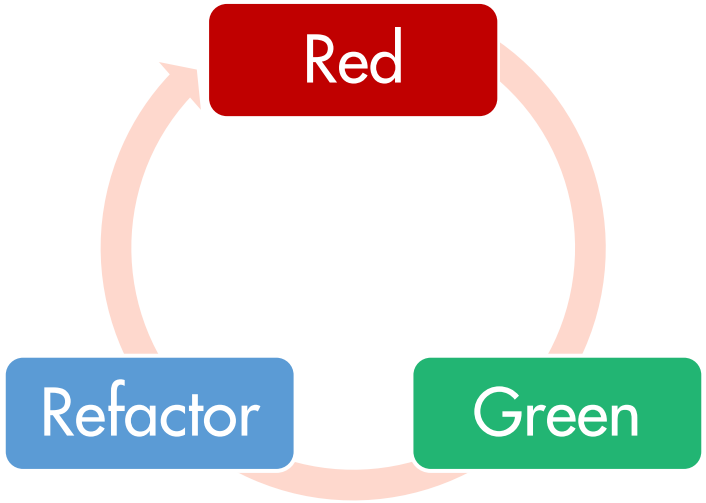


Summary – Think Tests

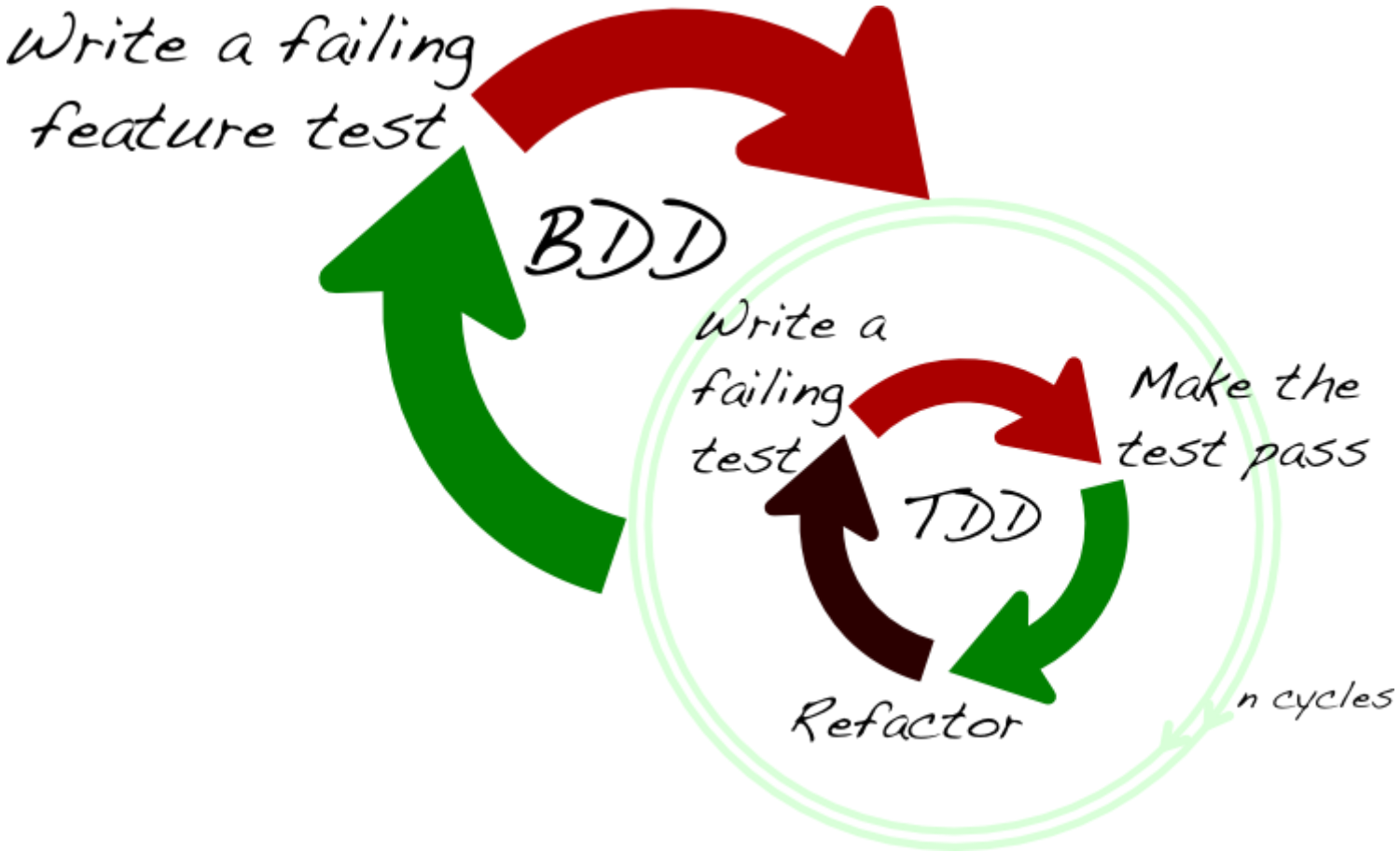
Tests are code that test code

TDD : Test first approach

BDD : Behavior approach



The double loop



Working with others

Avoid :

- missing code files
- Ugly code naming
- Code deletion

Anticipate problems

So rules are required !

- Coding style
- Code reviewal
- Automated process

Software Craftmanship Basics

Quality: clean code, refactoring, tests, simple design

Humility: question yourself, continously improvement

Sharing: pair programming, collective ownership of source code

Pragmatism: understand constraints, adapt !

Professionalism: Treat your client as a partner

Design patterns

15/05/2020

What is a design pattern ?

a general, reusable solution to a commonly occurring problem within a given context in software design

Common problem resolution

A pattern : you know it, you retrieve it

GoF : 24 patterns

Creational

Builder, *Factory method*, Abstract factory, Prototype, *Singleton*

Structural

Adapter, Bridge, Composite, *Decorator*, Façade, Flyweight, Proxy, Delegation

Behavioral

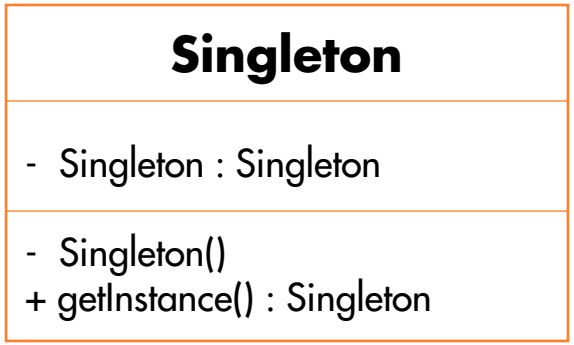
Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, *Observer*, State, *Strategy*, Template method, *Visitor*

Design patterns
Creational

15/05/2020

The singleton

Restricts the instantiation of a class to **one** object



The singleton : implementation

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance()
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

The singleton : thread safe

```
public class Singleton
{
    private static volatile Singleton instance;
    private static readonly object locker = new object();

    private Singleton() { }

    public static Singleton Instance() {
        if (instance == null) {
            lock (locker) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Double-checked locking

The singleton, thread safe

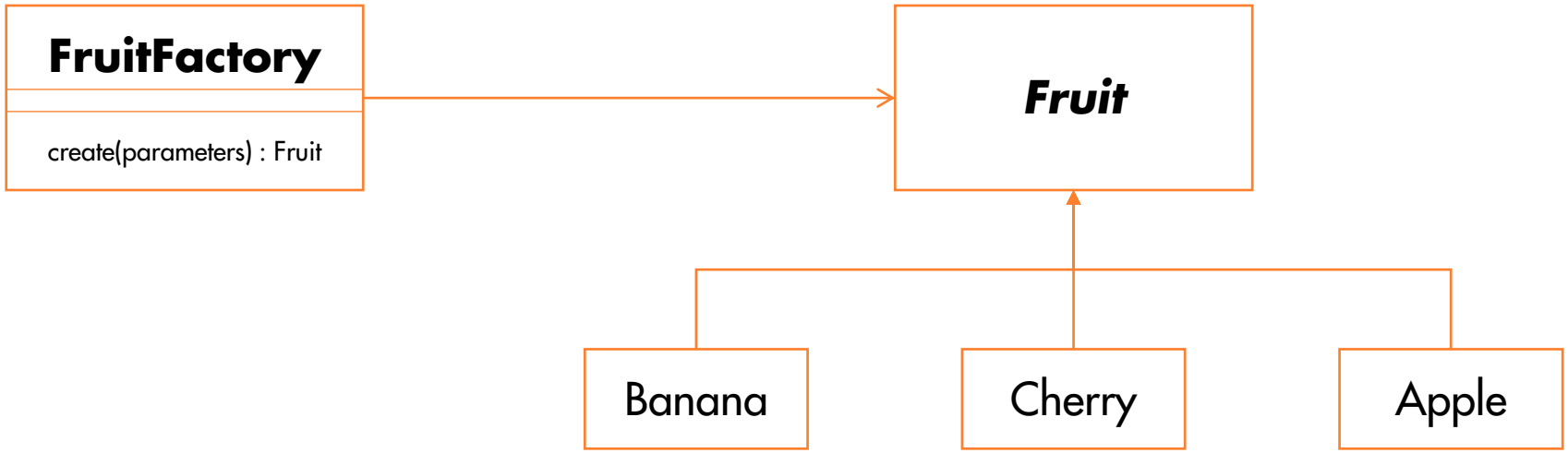
```
public class Singleton
{
    private static readonly Singleton instance = new Singleton();

    private Singleton() { }

    public static Singleton Instance() {
        return instance;
    }
}
```

The singleton, thread safe C++

```
static std::atomic<Singleton*> Singleton::m_instance = nullptr;
static std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
  Singleton* tmp = m_instance.load(std::memory_order_acquire);
  if (tmp == nullptr) {
    std::lock_guard<std::mutex> lock(m_mutex);
    tmp = m_instance.load(std::memory_order_relaxed);
    if (tmp == nullptr) {
        tmp = new Singleton; m_instance.store(tmp, std::memory_order_release);
    }
  }
  return tmp;
}
```

Factory method

Creating objects without having to **specify** the exact **class** of the object that will be created



Factory method

```
interface IFruit {
    int Price { get; }
}

class Cherry : IFruit {
    public int Price { get; } = 75;
}
class Apple : IFruit {
    public int Price { get; } = 100;
}
class Banana : IFruit {
    public int Price { get; } = 150;
}

enum FruitType {
    Banana,
    Apple,
    Cherry
}
```

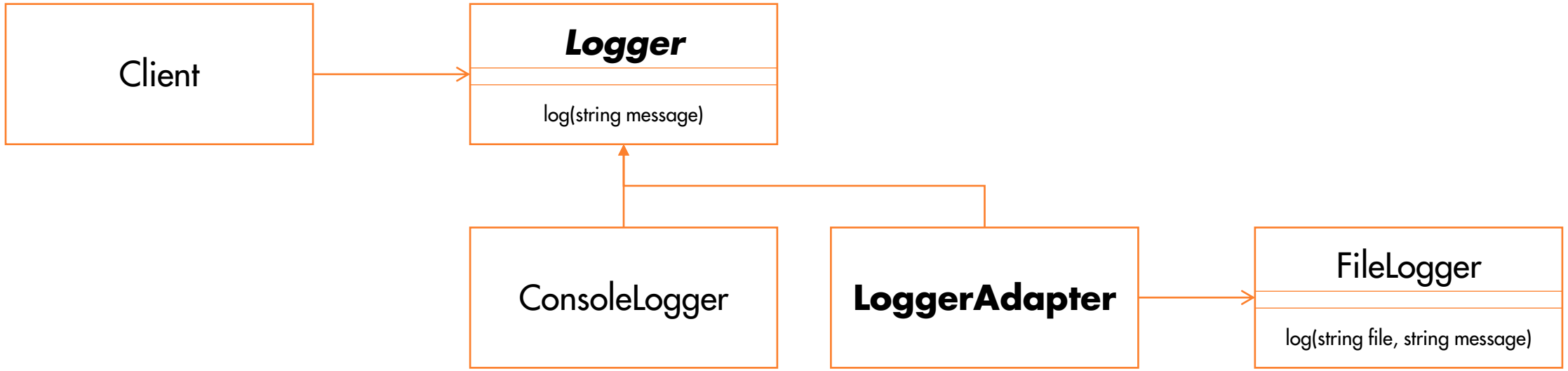
```
class FruitFactory
{
    private Dictionary<FruitType, Func<IFruit>> mapper;

    public FruitFactory()
    {
        mapper = new Dictionary<FruitType, Func<IFruit>>
        {
            {FruitType.Apple, () => new Apple()},
            {FruitType.Banana, () => new Banana()},
            {FruitType.Cherry, () => new Cherry()}
        };
    }

    public IFruit Create(FruitType type)
    {
        if (!mapper.TryGetValue(type, out var result))
        {
            throw new Exception($"No fruit with type {type}");
        }
        return result();
    }
}
```

Design patterns
Structural

15/05/2020

Adapter (aka Wrapper)

Allows the **interface** of an existing class **to be used** as **another** interface



Adapter

```
interface IOAdapter
{
    string Read();
    void Write(string message);
}

class ConsoleAdapter : IOAdapter
{
    public string Read()
    {
        return Console.ReadLine();
    }

    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

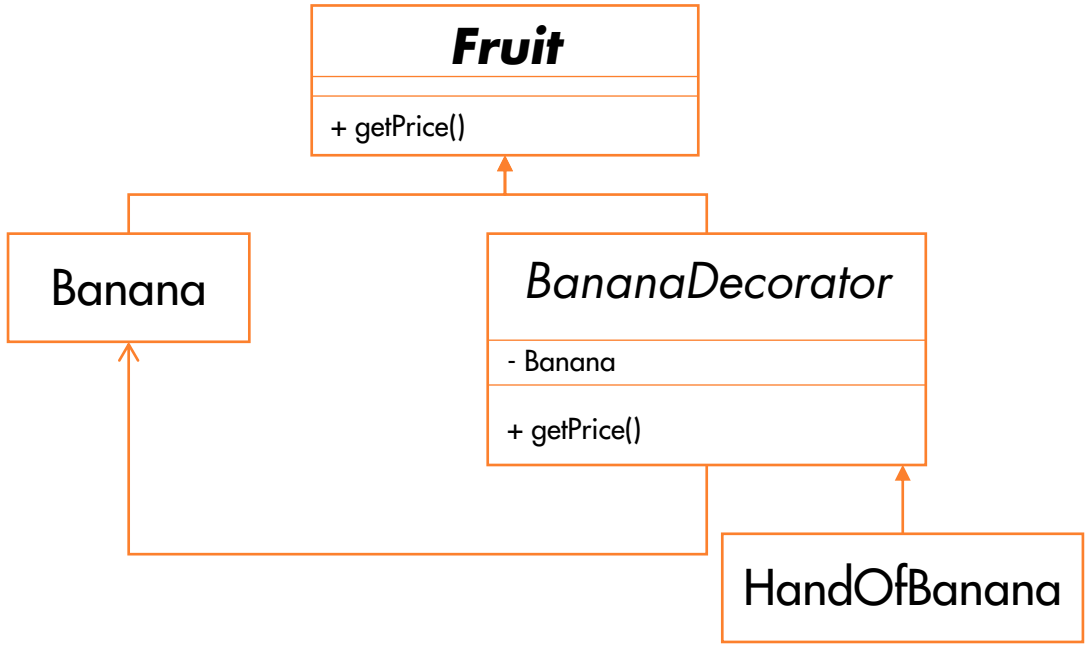
Adapter

```
interface IOAdapter
{
    FruitType Read();
    void Write(string message);
}

class ConsoleAdapter : IOAdapter
{
    public FruitType Read()
    {
        var input = Console.ReadLine();
        if (Enum.TryParse(typeof(FruitType),
                          input, true, out var value))
        {
            return (FruitType) value;
        }
        return default(FruitType);
    }

    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

```
interface IOAdapter
{
    IFruit Read();
    void Write(string message);
}

class ConsoleAdapter : IOAdapter
{
    FruitFactory readonly factory = new FruitFactory();

    public IFruit Read()
    {
        var input = Console.ReadLine();
        if (Enum.TryParse(typeof(FruitType),
                          input, true, out var value))
        {
            return factory.Create((FruitType)value);
        }
        return default(IFruit);
    }

    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}
```

Decorator

Allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class.



Decorator

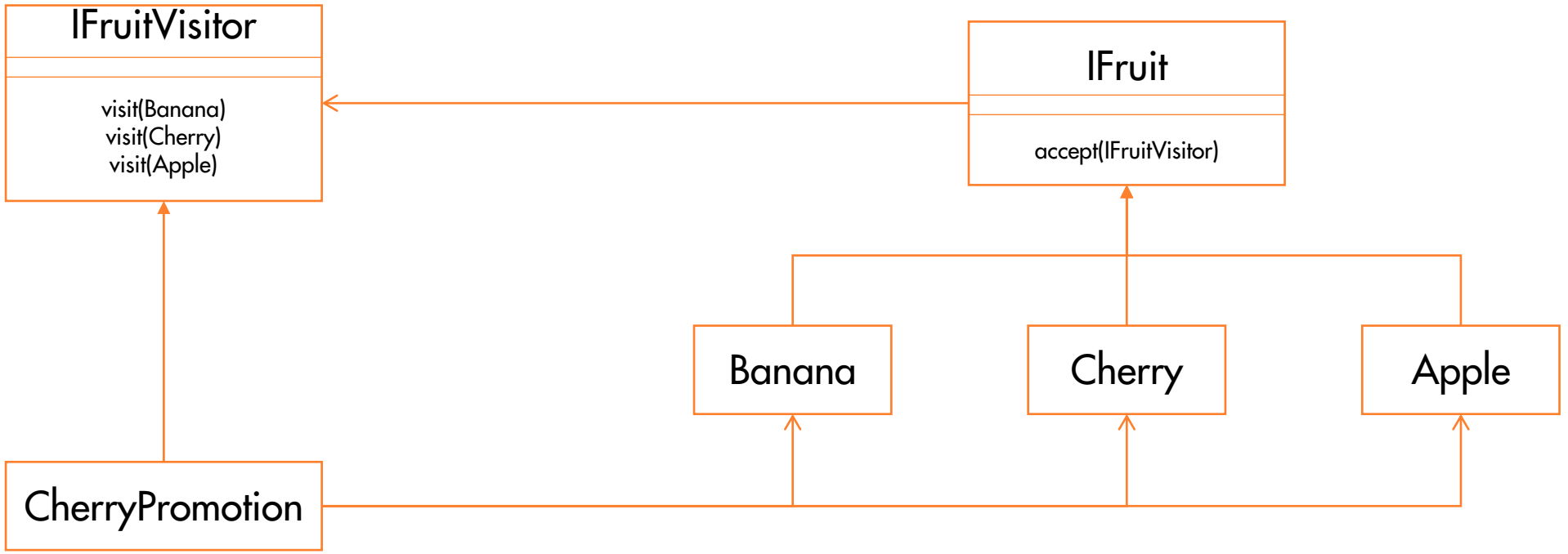
```
public interface IBananaDecorator : IFruit
{
    Banana Banana { get; }
}

public class HandOfBanana : IBananaDecorator
{
    const int factor = 4;

    public HandOfBanana(Banana banana)
    {
        Banana = banana;
    }

    public int Price => Banana.Price * factor;

    public Banana Banana { get; }

    public int Accept(IFruitVisitor fruitVisitor)
    {
        return Banana.Accept(fruitVisitor) * factor;
    }
}
```

Design patterns
Behavioural

15/05/2020

Visitor

A way of **separating an algorithm from an object** structure on which it operates



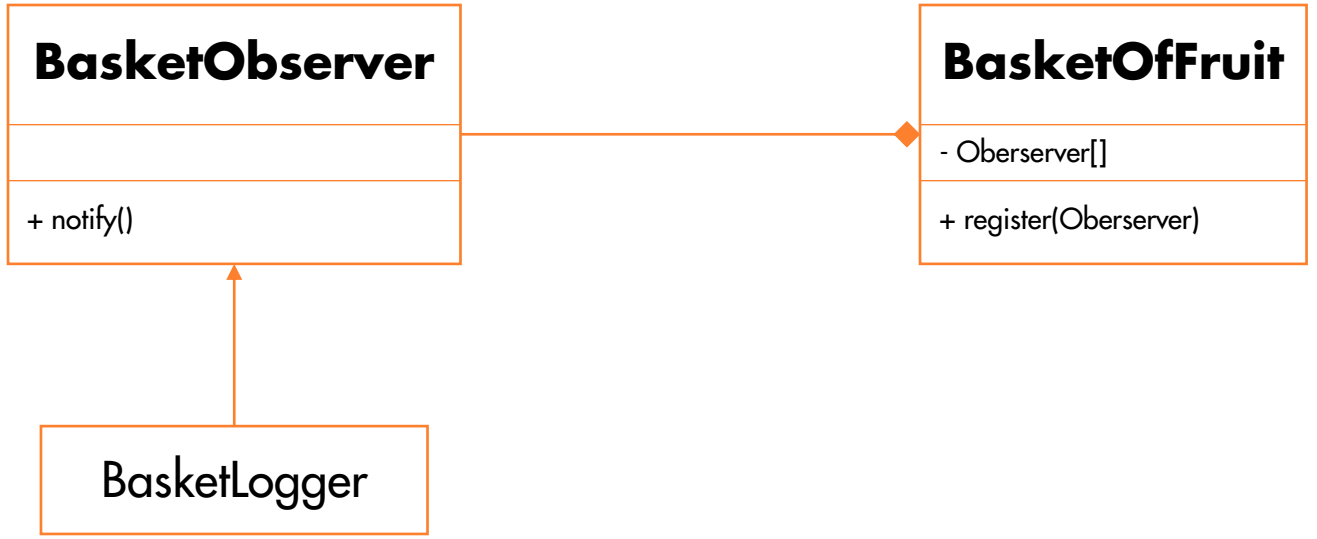
Visitor

```
public interface IFruit
{
    int Price { get; }

    int Accept(IFruitVisitor fruitVisitor);
}

public class Cherry : IFruit
{
    public int Price { get; } = 75;

    public int Accept(IFruitVisitor fruitVisitor)
    {
        return fruitVisitor.Apply(this);
    }
}
```

```
public interface IFruitVisitor
{
    int Apply(Cherry fruit);
    int Apply(Banana fruit);
    int Apply(Apple fruit);
}

internal class CherryPromotion : IFruitVisitor
{
    private int count;

    public int Apply(Apple fruit)
    {
        return fruit.Price;
    }
    public int Apply(Banana fruit)
    {
        return fruit.Price;
    }

    public int Apply(Cherry fruit)
    {
        var reduction = 0;
        count++;
        if (count % 2 == 0) reduction = -20;
        return fruit.Price + reduction;
    }
}
```

Observer

An object, called the **subject**, maintains a **list of its dependents**, called **observers**, and **notifies them automatically** of any state **changes**, usually by calling one of their methods.



Observer

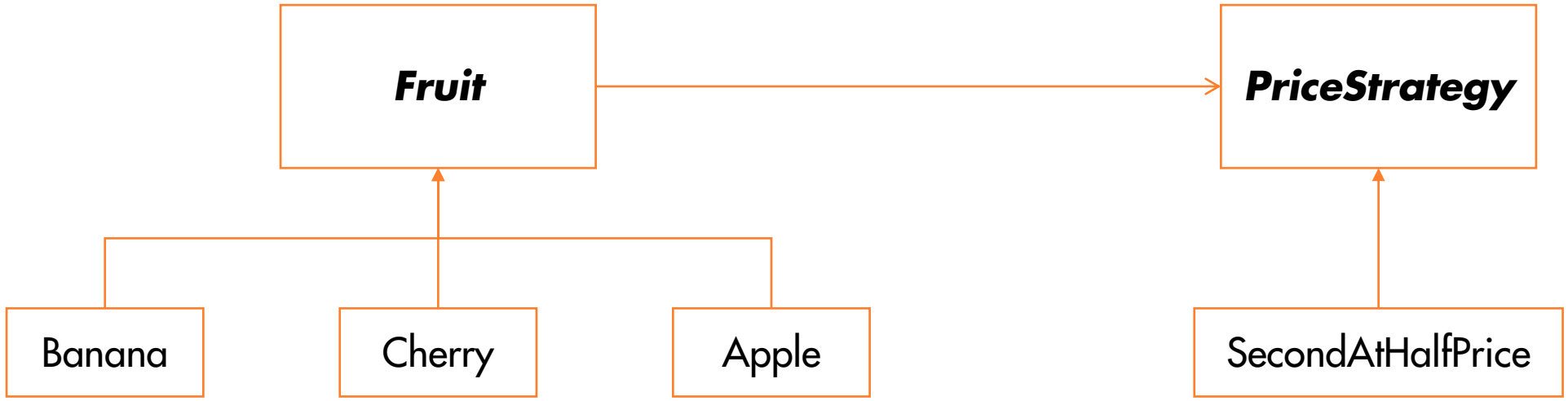
```
public interface IObservableBasket
{
    void Register(IBasketObserver observer);
    void Add(IFruit fruit);
}
```

```
public class ObservableBasket : IObservableBasket
{
    List<IFruit> list = new List<IFruit>();
    private readonly List<IBasketObserver> observers = ...

    public void Register(IBasketObserver observer) {
        observers.Add(observer);
    }

    public void Add(IFruit fruit) {
        list.Add(fruit);
        Notify(fruit);
    }

    private void Notify(IFruit fruit) {
        foreach (var observer in observers) {
            observer.Notify(fruit);
        }
    }
}
```

```
public interface IBasketObserver
{
    void Notify(IFruit fruit);
}

class BasketLogger : IBasketObserver
{
    public void Notify(IFruit fruit)
    {
        ConsoleAdapter.Instance.Write($"Item added : “
                                      + fruit.GetType().Name);
    }
}
```

Strategy

Enables selecting an algorithm at runtime



Strategy

```
public class CherryForStrategy : Cherry
{
    public CherryForStrategy(IPriceStrategy strategy)
    {
        Strategy = strategy;
    }

    public IPriceStrategy Strategy { get; set; }
    public override int Price => Strategy.Apply(base.Price);
}
```

```
public interface IPriceStrategy
{
    int Apply(int defaultPrice);
}
public class SecondAtHalfPrice : IPriceStrategy
{
    private int count;
    public int Apply(int defaultPrice)
    {
        var reduction = 0;
        count++;
        if (count % 2 == 0)
        {
            reduction = -20;
        }
        return defaultPrice + reduction;
    }
}
```

Organize your time

15/05/2020

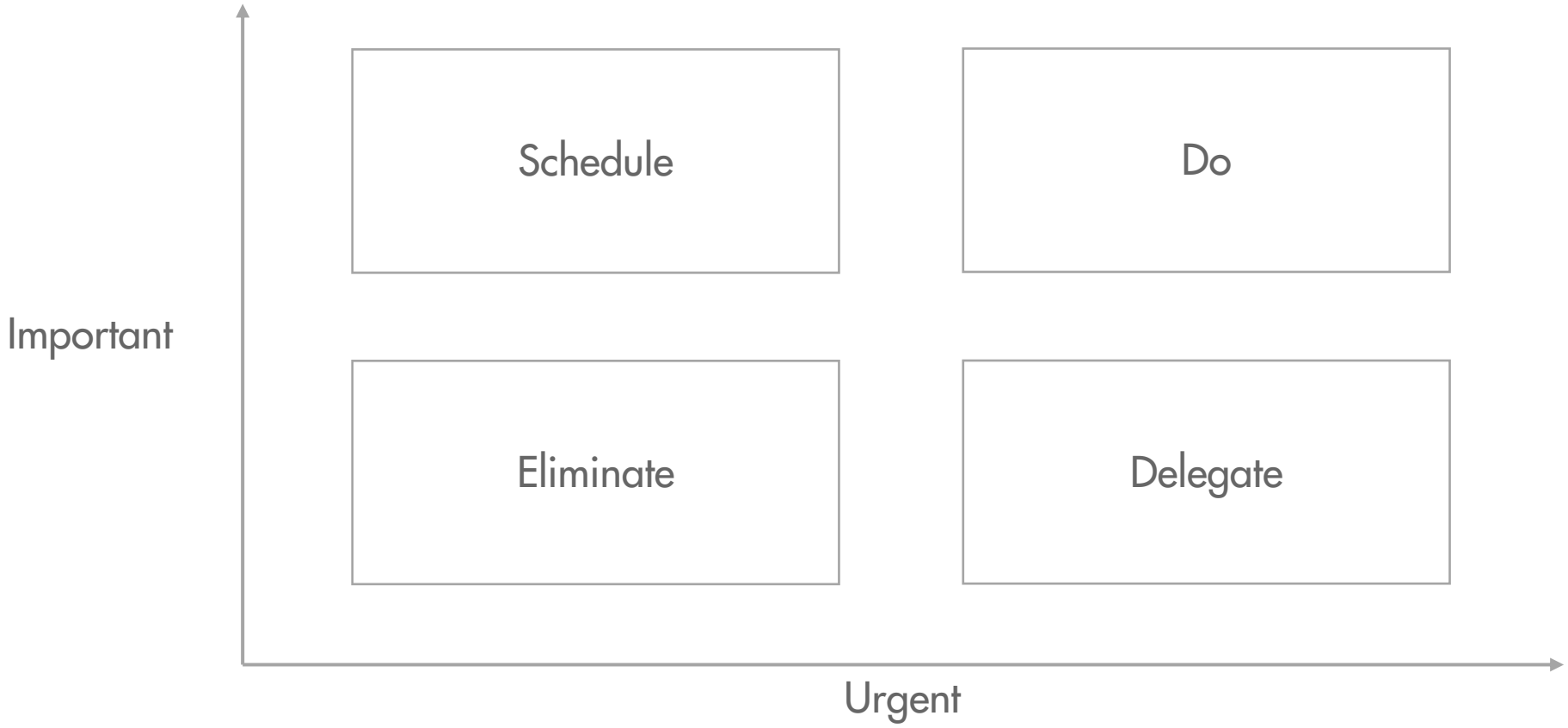
Get Things Done

1. Capture / Collect
2. Clarify / Process
3. Organize
4. Reflect / Plan
5. Engage / Do

Pomodoro

1. Focus on one task during 25′
2. Take a break 5′
3. Repeat 3 or 4 times
4. Take a break 20′

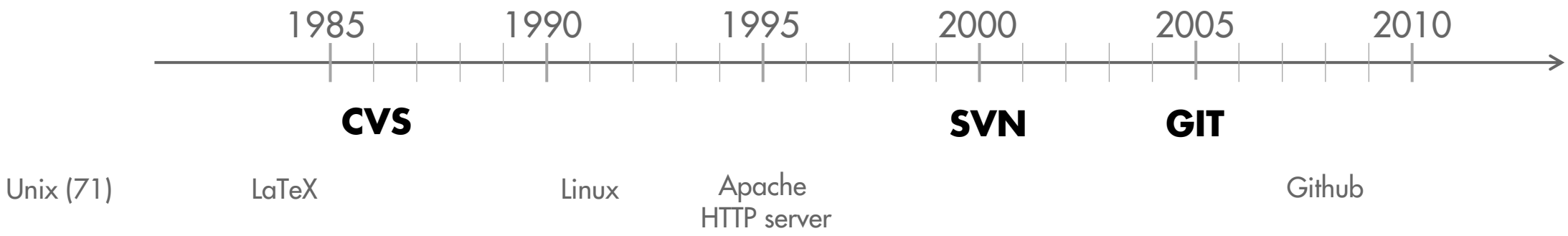
Eisenhower matrix



Organize your code

15/05/2020

Version control system



Version control system

VCS	Revision	Mode
CVS	Per file	Client-server
SVN	Per commit	Client-server
GIT	Per commit	Distributed

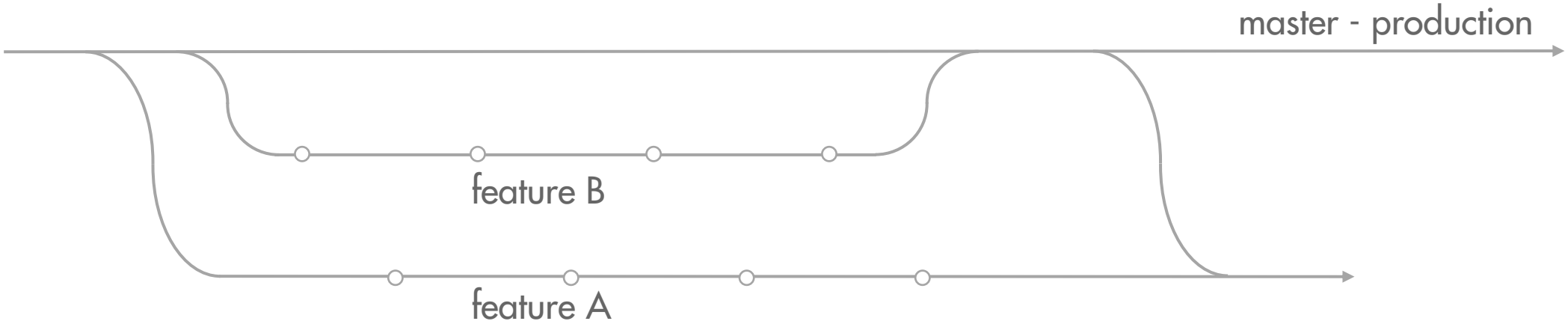
Branching modes

How many productions versions ?

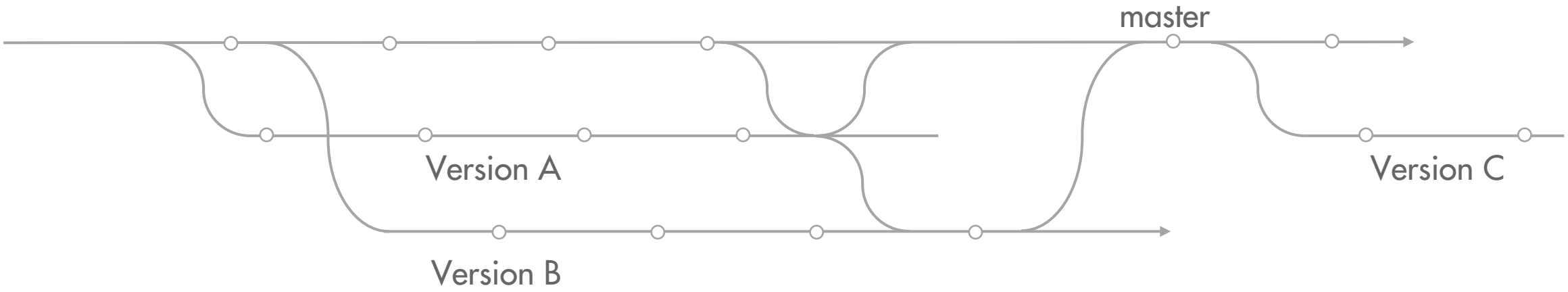
How many developers ?

Which development process ?

Branching mode : Feature



Branching mode : Release



Organize your learning

15/05/2020

Train yourself

Monitory technology

Try them : POC

Be careful of the silver bullet syndrome !

Train with others

Coding dojo / Kata

Pair programming

Code reviews