Clean Architecture

SOLID design principle

02-05-2018

Overview

- **1 Building blocks** – SOLID principle
  - *Explanation of the principles and how can it be utilized*
  - *Questions?*

- **2 Clean architecture** – The separation of concerns via layers
  - *Explanation in what the layers means and what it should contain*
  - *Questions?*

- **3 Conclusions**
  - *Pros and Cons*
  - *Questions?*

1 Building blocks
SOLID principle

5 principles of OOP, defined by *Robert C. Martin* aka *Uncle Bob* in *early 2000s*

- ***S*** **- S**ingle **R**esponsibility **P**rinciple (SRP)
- ***O*** **- O**pen / **C**losed **P**rinciple (OCP)
- ***L*** **- L**iskov **S**ubstitution **P**rinciple. (LSP)
- ***I*** **- I**nterface **Segregation Principle** (ISP)
- ***D*** **- D**ependency **I**nversion **P**rinciple (DIP)

See also SOLID cheatsheet – handout

SOLID - Single Responsibility Principle (SRP)

“There should be never more than one reason for a class to change.”

Meaning:

A class should have one, and only one responsibility

```ruby
# BAD

class PlaceOrder
  def initialize(product)
    @product = product
  end

  def run
    # 1. Logic related to verification of
    #    stock availability
    # 2. Logic related to payment process
    # 3. Logic related to shipment process
  end
end
```

```ruby
# GOOD

class PlaceOrder
  def initialize(product)
    @product = product
  end

  def run
    StockAvailability.new(@product).run
    ProductPayment.new(@product).run
    ProductShipment.new(@product).run
  end
end
```

S**O**LID - **O**pen / **C**losed **P**rinciple (OCP)

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

Meaning:

Behavior of classes can be extended, without modification,

we should strive to write code that doesn’t have to be changed every time the requirements change (Inheritance, polymorphism)

```ruby
# BAD

class Logger
  def initialize(logging_form)
    @logging_form = logging_form
  end

  def log(message)
    puts message if @logging_form == "console"
    File.write("logs.txt", message) if @logging_form == "file"
  end
end
```

```ruby
# GOOD

class EventTracker
  def initialize(logger: ConsoleLogger.new)
    @logger = logger
  end

  def log(message)
    @logger.log(message)
  end
end

class ConsoleLogger
  def log(message)
    puts message
  end
end

class FileLogger
  def log(message)
    File.write("logs.txt", message)
  end
end
```

SOLID - Liskov Substitution Principle (LSP)

“Subtypes must be substitutable for their base types”

Meaning:

Any subclass can be replaced by its base class, functions that use references to base classes must be able to use objects of the derived class without knowing it

```ruby
# BAD

class Rectangle
  def initialize(width, height)
    @width, @height = width, height
  end

  def set_width(width)
    @width = width
  end

  def set_height(height)
    @height = height
  end
end

class Square < Rectangle
  # LSP violation: inherited class
  # overrides behavior of parent's methods
  def set_width(width)
    super(width)
    @height = height
  end

  def set_height(height)
    super(height)
    @width = width
  end
end
```

SOLID - Interface **Segregation Principle** (ISP)

“Classes that implement interfaces, should not be forced to implement methods they do not use.”

Meaning: A lot of small interfaces then less fatter interfaces

```ruby
# BAD

class Car
  def open
  end

  def start_engine
  end

  def change_engine
  end
end

# ISP violation: Driver instance does not make use
# of #change_engine
class Drive
  def take_a_ride(car)
    car.open
    car.start_engine
  end
end

# ISP violation: Mechanic instance does not make use
# of #start_engine
class Mechanic
  def repair(car)
    car.open
    car.change_engine
  end
end
```

SOLI**D** - **D**ependency **I**nversion **P**rinciple (DIP)

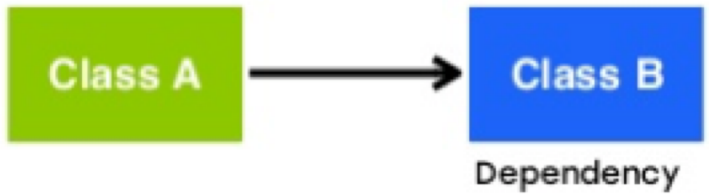
“High level modules should not depend on low level modules rather both should depend on abstraction. Abstraction should not depend on details; rather detail should depend on abstraction.”

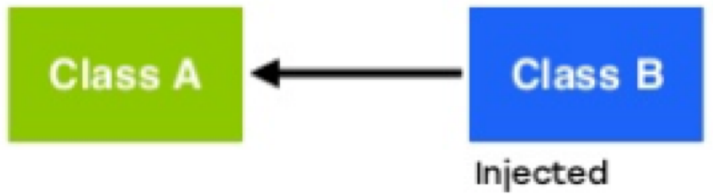
Meaning: by depending on a concept instead of on an implementation, you reduce the need for change

```ruby
# BAD

class EventTracker
  def initialize
    # An instance of low-level class ConsoleLogger
    # is directly created inside high-level
    # EventTracker class which increases class'
    # coupling
    @logger = ConsoleLogger.new
  end

  def log(message)
    @logger.log(message)
  end
end
```

```ruby
# GOOD

class EventTracker
  def initialize(logger: ConsoleLogger.new)
    # Use dependency injection as in closed/open
    # principle.
    @logger = logger
  end

  def log(message)
    @logger.log(message)
  end
end
```

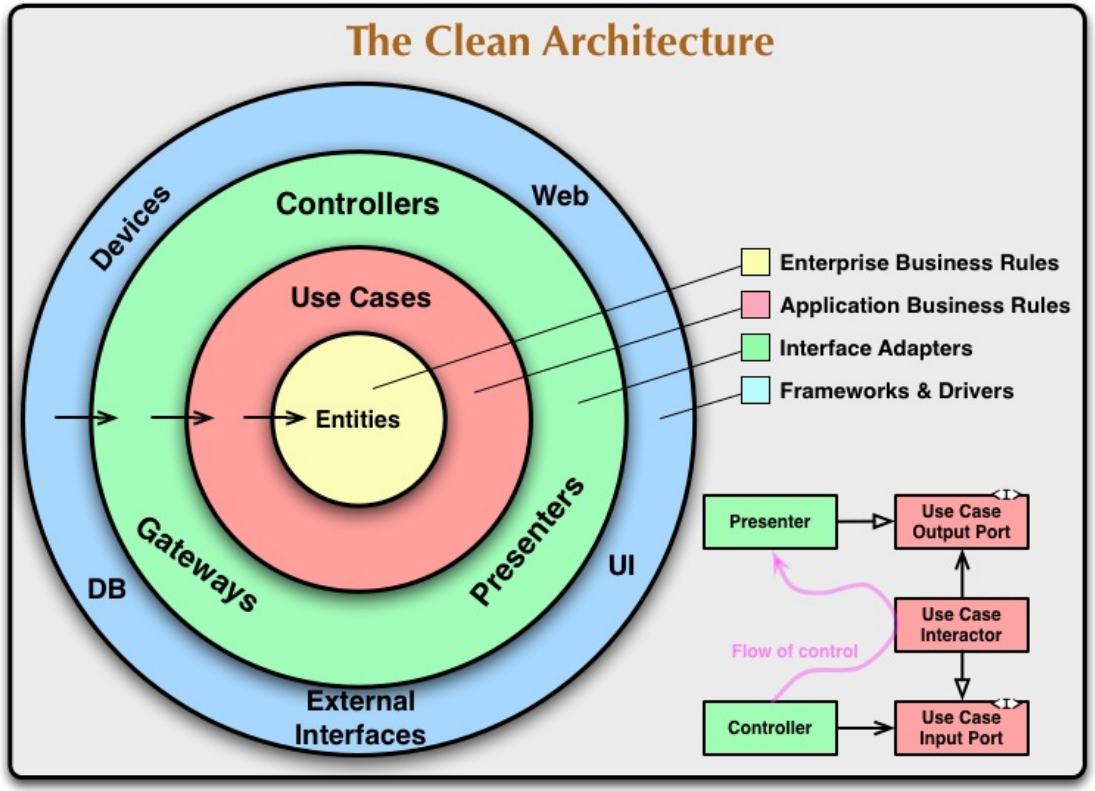
Inversion of control



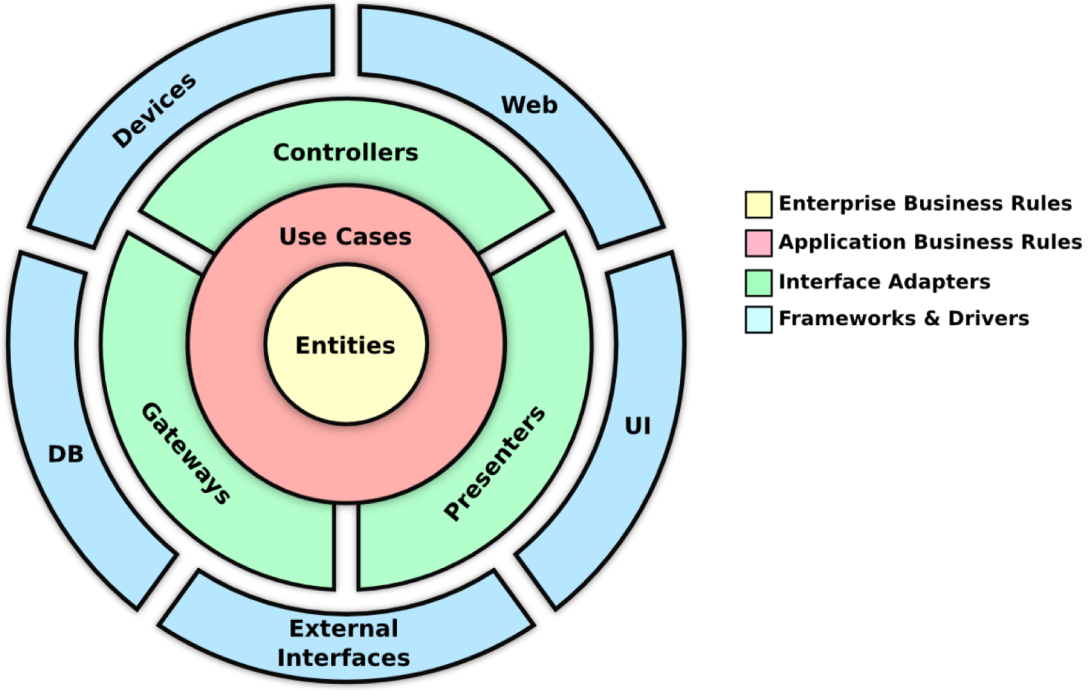


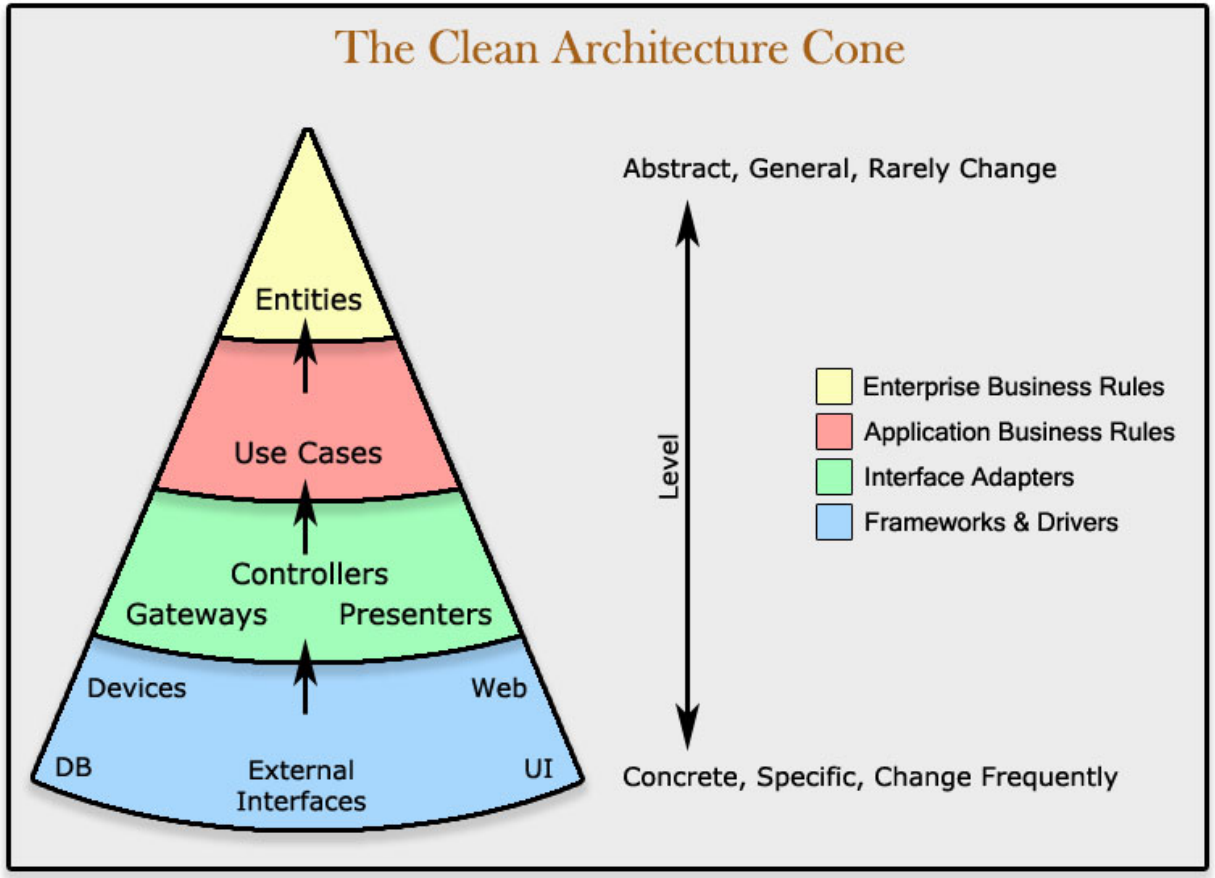
2 The Clean Architecture

- *proposed and evangelized* by *Robert C. Martin*, aka *Uncle Bob*, first mentioned in *2011, popularized in 2012*



The Clean Architecture

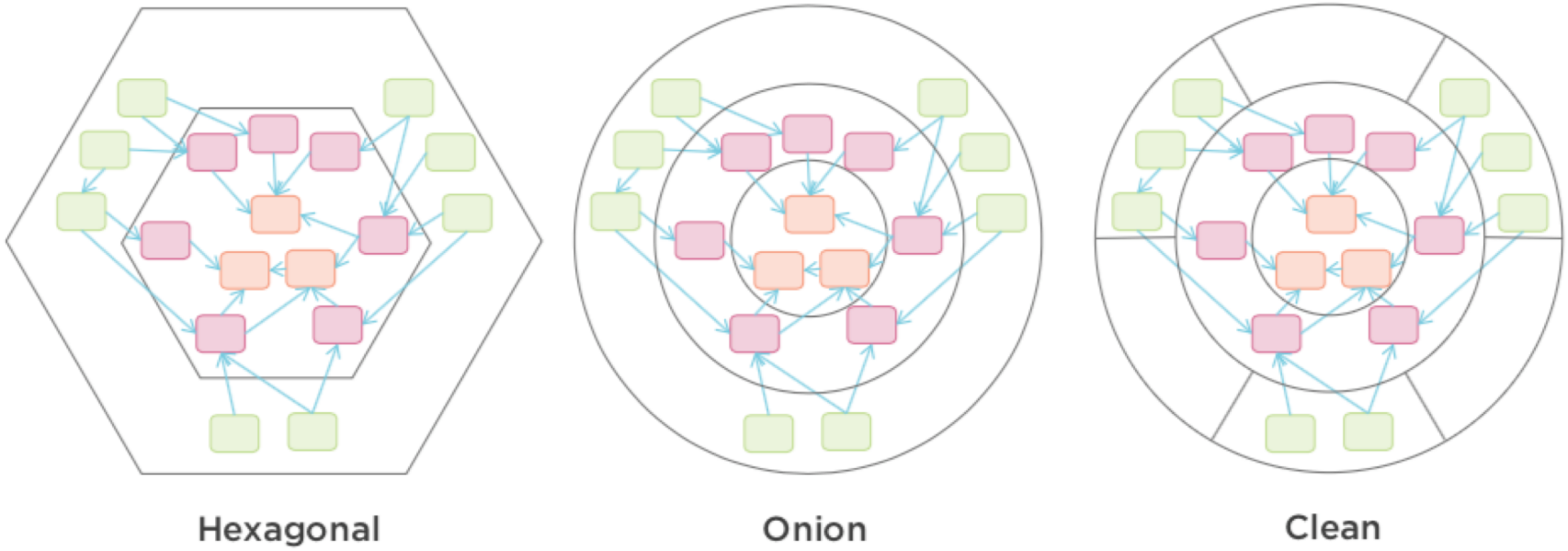




XYZ architecture – It's all the same

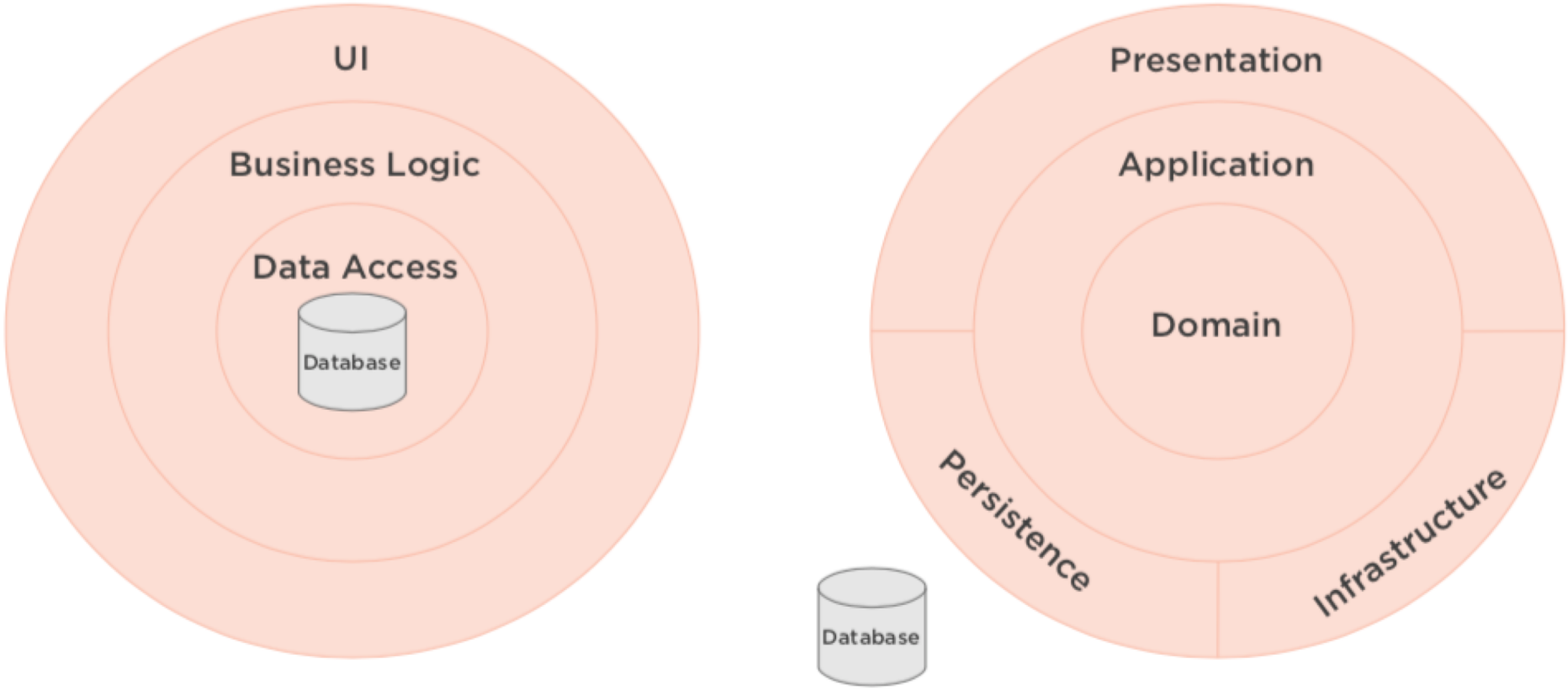
Common objective: Separation of concerns via layers (domain centric), all vary somewhat in their details

Keep in mind that every architecture decision is a trade-off.

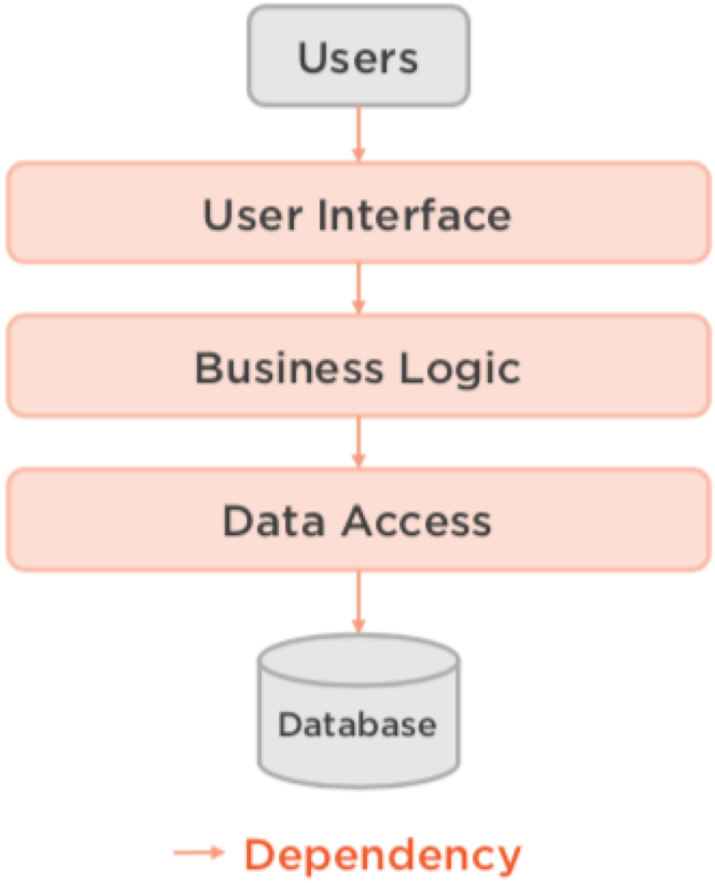


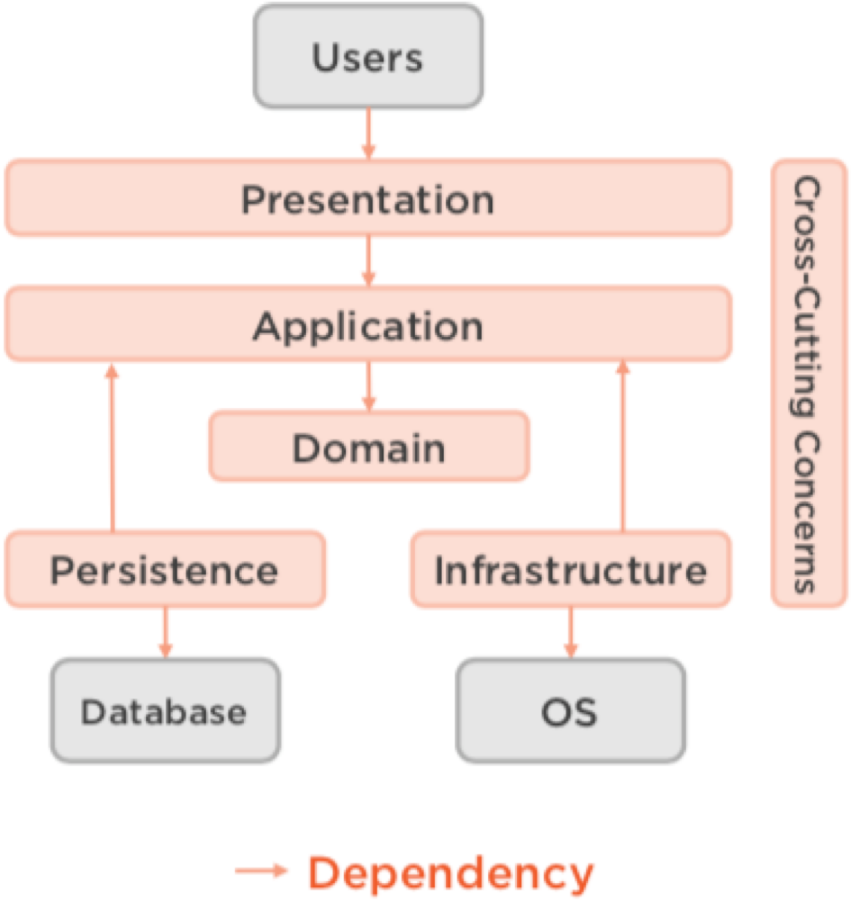
Original Source: http://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/

Data-centric vs Domain-centric Architecture

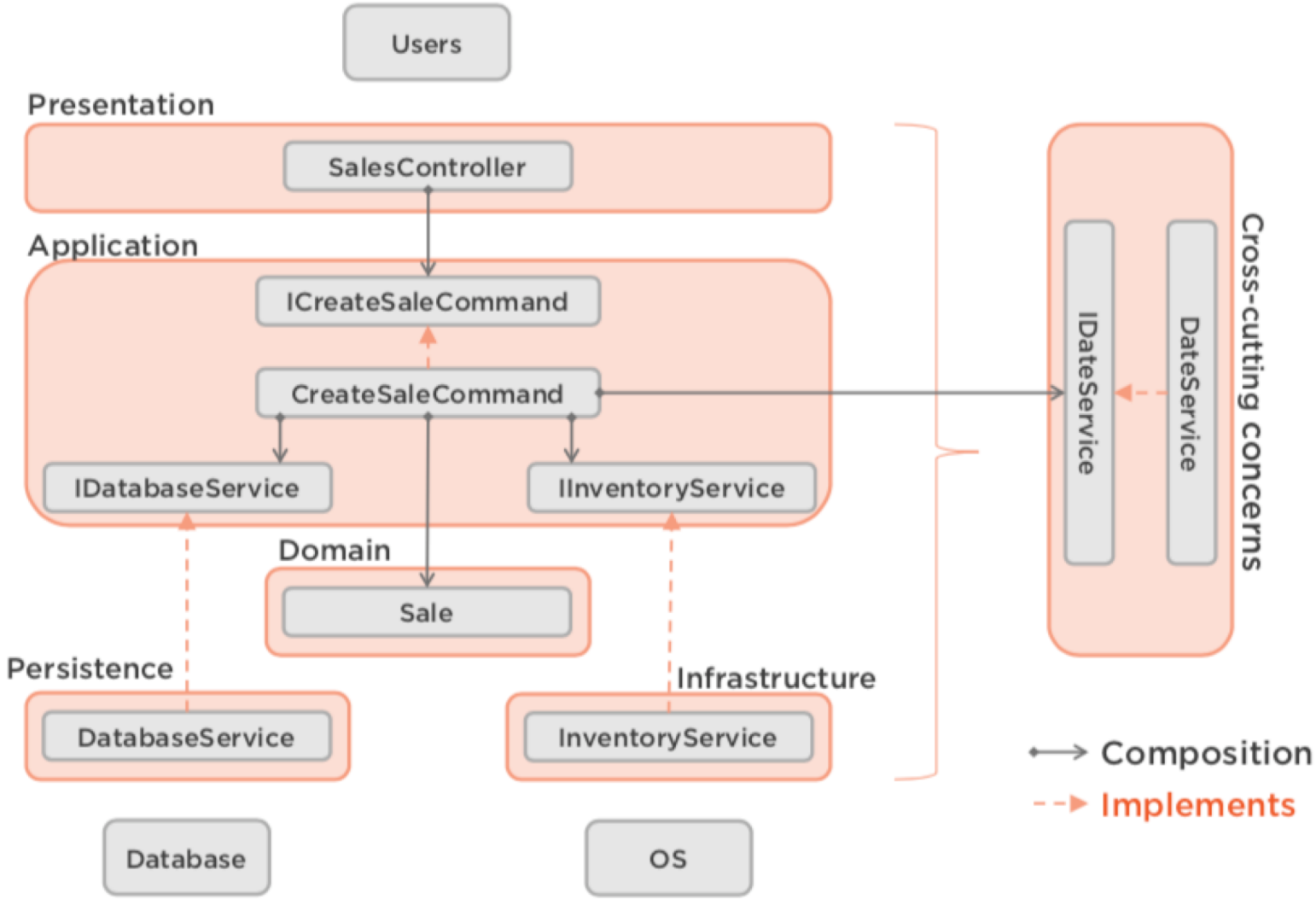


Data-centric vs Domain-centric Architecture

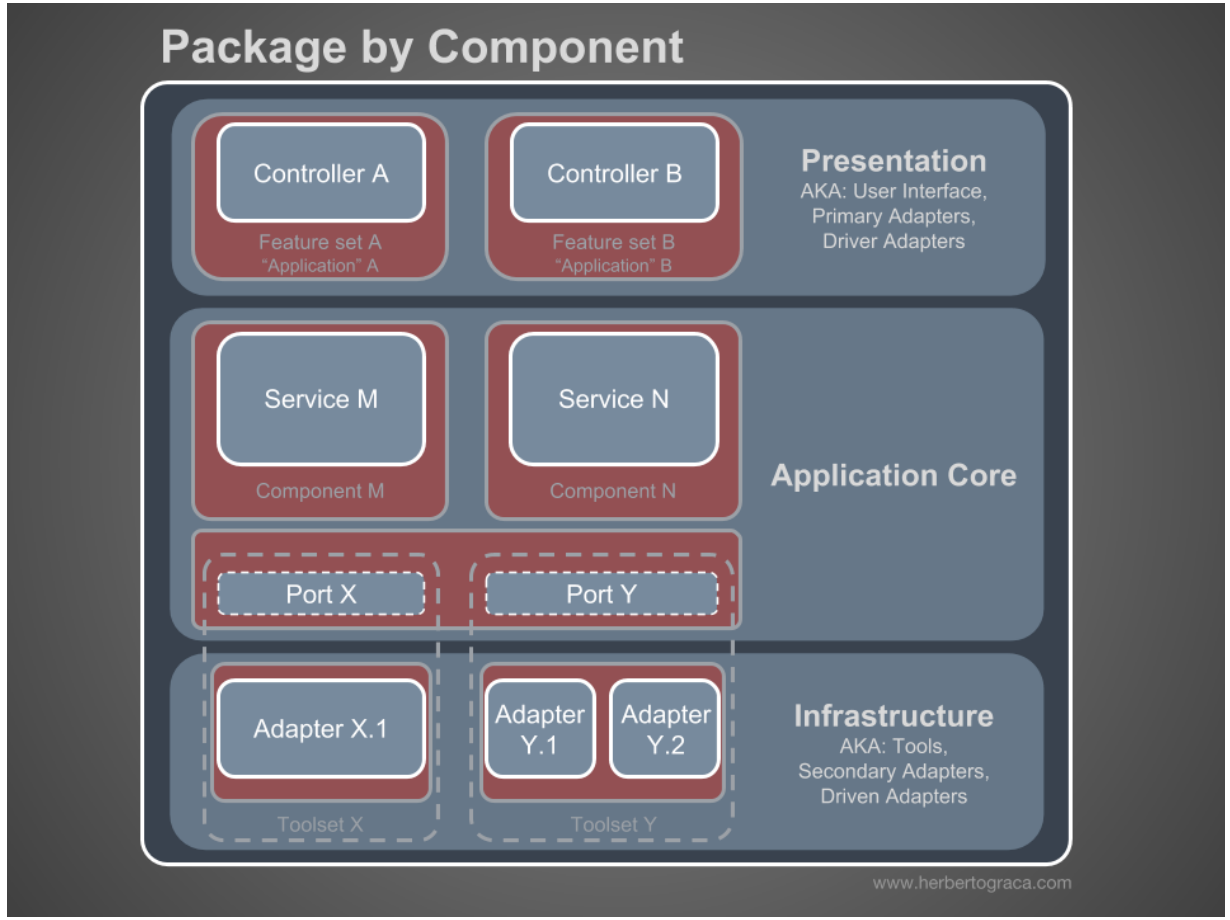


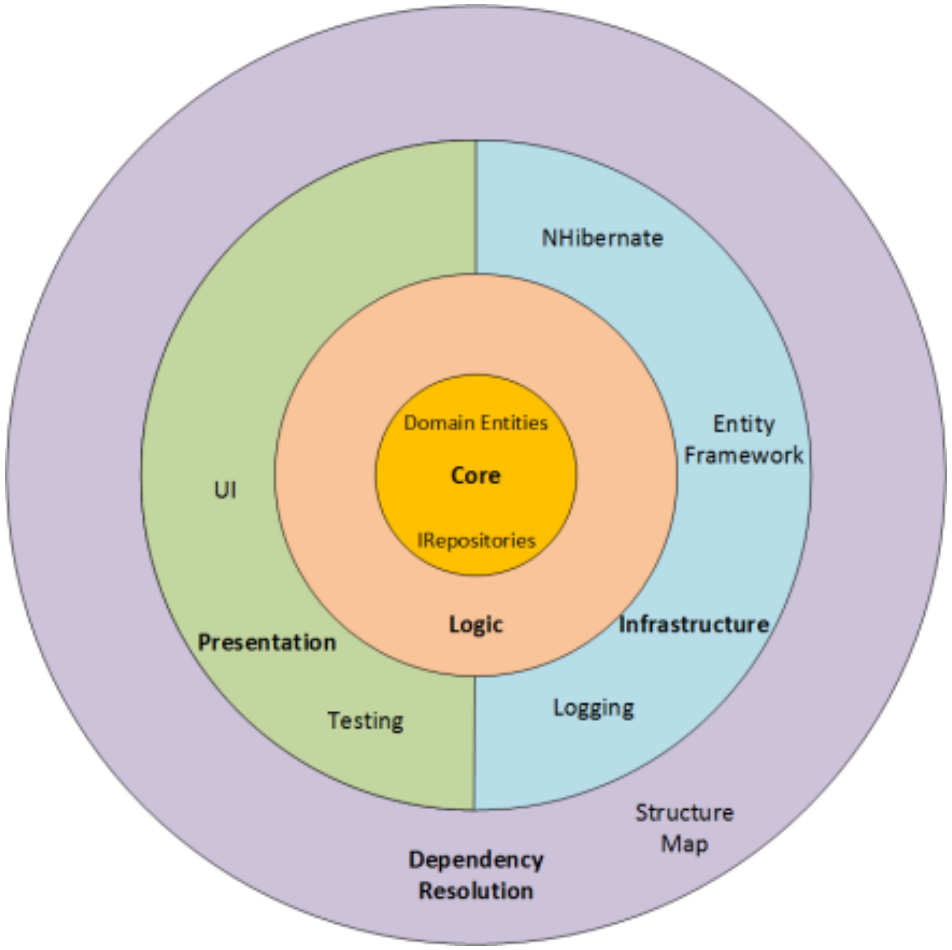


Clean Architecture

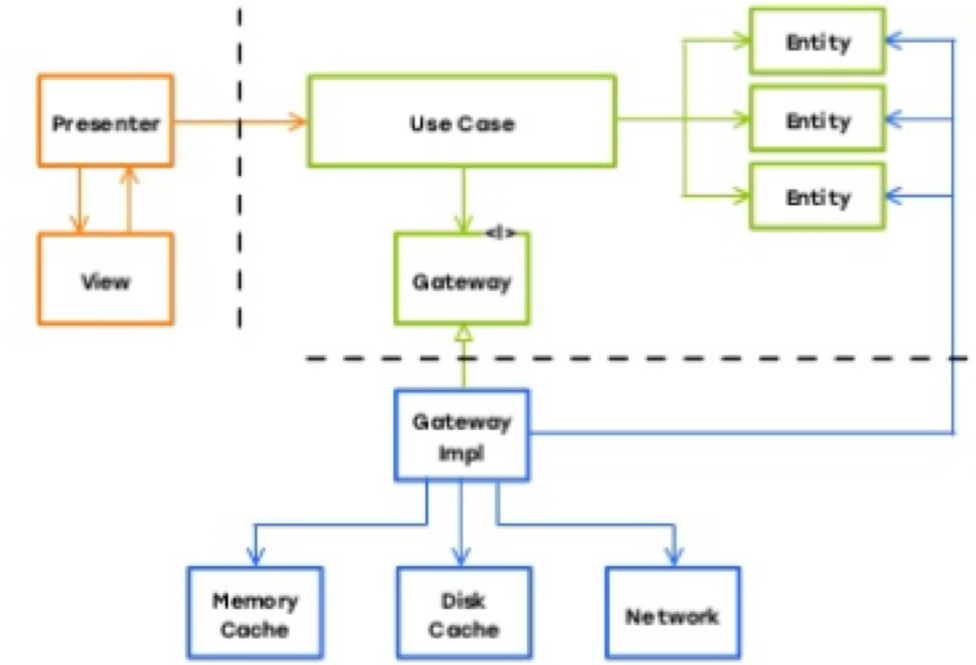


Clean Architecture





Presenters / Gateways



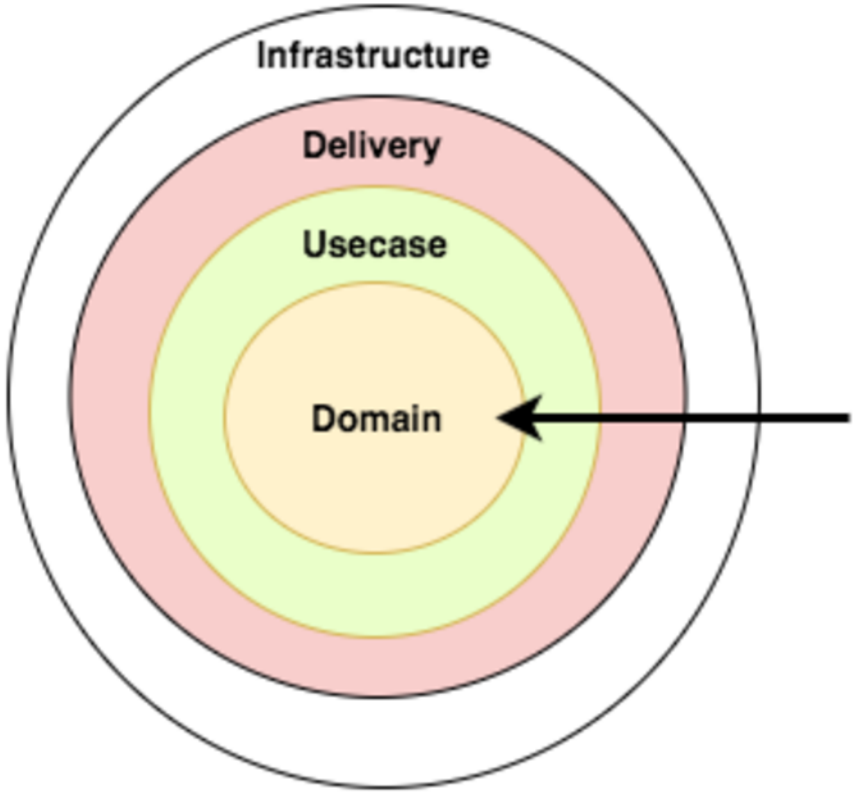
Clean Architecture – Dependency rule

(inversion of control)

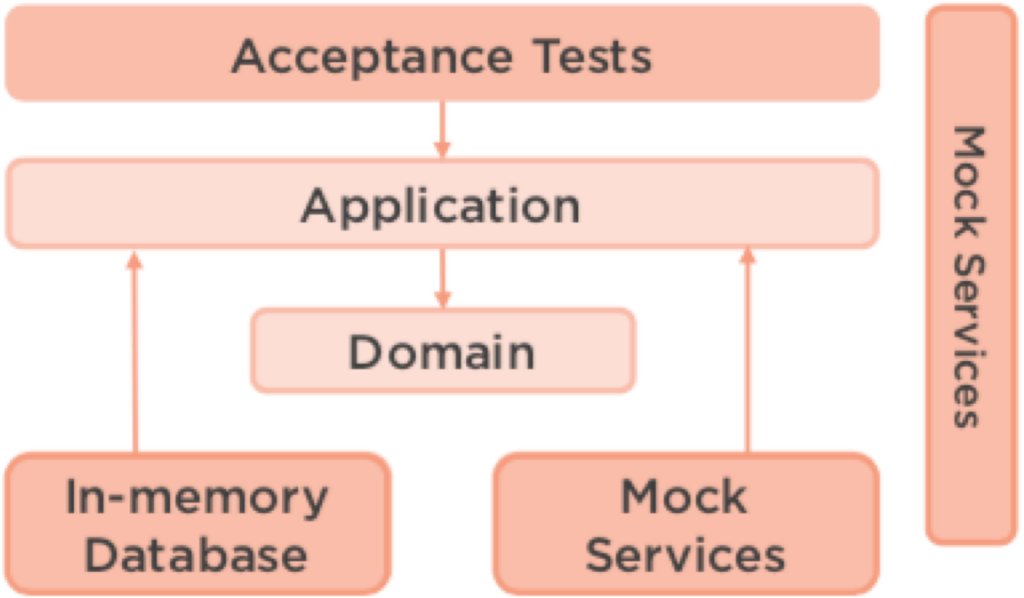
The overriding rule that makes this architecture work is The Dependency Rule.

This rule says that source code dependencies can only point inwards.

Nothing in an inner circle can know anything at all about the implementation of something in an outer circle



Clean Architecture - **Testability in isolation**



3 Conclusion

Cons

- Not wanting to rely on framework usage – the *dependency rule*
- Learning Curve – it’s harder to grap
- Change is difficult
- Requires more thought
- Initial higher cost

Conclusion

Pros

- *interchangeable* Allows us to with a different implementation that conforms to the same interface.
- *Defer decisions until the very end*. Build the logic of your application without worrying what will be the user interface or which database
- *Implement features faster*. Facilitates by the separation of concerns, so you can concentrate on one specific task at a time and you develop faster.