Software Engineering Best Practices

Workshop

Learning objectives

1. Learn about **Domain Driven Design (DDD)**
2. Improving your code quality with **Test-Driven Development (TDD)**
3. Writing clean code by using **SOLID principles**
4. Increase development productivity by automating releases (**Continuous Integration / Continuous Delivery**).

SOLID Principles

SOLID

SOLID is an acronym for five design principles intended to make software designs more understandable, flexible and maintainable.

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Injection Principle (DIP)

Single Responsibility Principle

A class should have one and only one responsibility.

A class should only have **one reason** to be modified.

toTicketFormat returns a string representation of the product to be printed in a Ticket.

```
3   public class Product {
4
5       private Sku sku;
6       private String name;
7       private Price price;
8
9       public String getName() {
10          return name;
11      }
12
13      public void setName(String name) {
14          this.name = name;
15      }
16
17      public Price getPrice() {
18          return price;
19      }
20
21      public void setPrice(Price price) {
22          this.price = price;
23      }
24
25      public String toTicketFormat() {
26        StringBuilder builder = new StringBuilder();
27        builder.append("Product: ");
28        builder.append(this.name);
29        builder.append("Price: ");
30        builder.append(this.price.getValue());
31        return builder.toString();
32      }
33  }
```

SRP Violation Class is responsible for **both**, the structure of the Product **and** the formatting of it.

```java
public class Product {

    private Sku sku;
    private String name;
    private Price price;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Price getPrice() {
        return price;
    }

    public void setPrice(Price price) {
        this.price = price;
    }
}
```

Product class only knows about the structure of a product.

TicketPrinter.java

```java
public class TicketPrinter {

    public String printTicket(List<Product> products);

}
```

TicketPrinter class is responsible for formatting products into printable strings.

iTJuana.

Open/Closed Principle

Classes should be open for extension

but closed for modification.

ShoppingCart.java

```java
public class ShoppingCart {

  private Customer customer;
  private List<Product> products;

  public double calculateCheckoutTotal() {
    double subtotal = products.stream()
                      .map(product -> product.getPrice().getValue())
                      .reduce(0.0, Double::sum);

    double taxRate;
    switch(customer.getCountry()) {
      case "Mexico": taxRate = 0.16; break;
      case "US": taxRate = 0.08; break;
    }

    return subtotal * taxRate;
  }
}
```

We have to modify this class for every new Country we want to support.

ShoppingCart.java

```java
public class ShoppingCart {

  private Customer customer;
  private List<Product> products;

  @Inject
  private TaxService taxService;

  public double calculateCheckoutTotal() {
    double subtotal = products.stream()
                .map(product -> product.getPrice().getValue())
                .reduce(0.0, Double::sum);

    double taxRate = taxService.getRateForCountry(customer.getCountry());
    return subtotal * taxRate;
  }
}
```

We have made this class **open** for extension. No modification is needed to support more Countries.

```java
public class Addition implements CalculatorOperation {
    private double left;
    private double right;
    private double result = 0.0;

    public Addition(double left, double right) {
        this.left = left;
        this.right = right;
    }

    // getters and setters

}
```

```java
public class Calculator {

    public void calculate(CalculatorOperation operation) {
        if (operation == null) {
            throw new InvalidParameterException("Can not perform operation");
        }

        if (operation instanceof Addition) {
            Addition addition = (Addition) operation;
            addition.setResult(addition.getLeft() + addition.getRight());
        } else if (operation instanceof Subtraction) {
            Subtraction subtraction = (Subtraction) operation;
            subtraction.setResult(subtraction.getLeft() - subtraction.getRight());
        }
    }
}
```

We have to modify this class for every new operation we want to support.

```java
public class Addition implements CalculatorOperation {
    private double left;
    private double right;
    private double result;

    // constructor, getters and setters

    @Override
    public void perform() {
        result = left + right;
    }
}
```

```java
public interface CalculatorOperation {
    void perform();
}
```

```java
public class Calculator {

    public void calculate(CalculatorOperation operation) {
        if (operation == null) {
            throw new InvalidParameterException("Cannot perform operation");
        }
        operation.perform();
    }
}
```

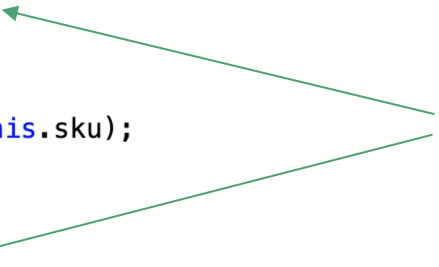
Liskov Substitution Principle

Any derived class should be able to substitute its parent class without the consumer knowing it.

Parent classes should be easily substituted with their child classes without completely changing the behavior.

Service.java

```java
public class CatalogItem {

  private Sku sku;
  private String name;
  private Price price;
  private Category category;

  public getInventory();
}

public class Product extends CatalogItem {

  public int getInventory() {
    return inventoryService.getInventory(this.sku);
  }
}

public class Service extends CatalogItem {

  public int getInventory() {
    throw new RuntimeException("Services do not have inventory");
  }
}
```



Both **Product** and **Service** extend from CatalogItem.

But, services do not have an inventory.

```
3   class Main {
4     public static void main(String[] args) {
5
6       List<CatalogItem> catalogItems = new ArrayList<>();
7       catalogItems.add(new Product("tv"));
8       catalogItems.add(new Product("radio"));
9
10      catalogItems.add(new Service("haircut"));
11      catalogItems.add(new Service("massage"));
12
13      int totalInventory = 0;
14      for(CatalogItem item : catalogItems) {
15        totalInventory += item.getInventory();
16      }
17
18    }
19  }
```

Problem. Services do not have inventory. This will throw an exception.

Inventory.java

```java
public abstract class CatalogItem {
  private Sku sku;
  private String name;
  private Price price;
  private Category category;
}

public interface Inventoryable {
  getInventory();
}

public class Product extends CatalogItem implements Inventoryable {

}

public class Service extends CatalogItem {

}
```

Base class should be as thin as possible.

Additional behavior can be modeled as interfaces that only certain concrete classes implement.

```
3   class Main {
4     public static void main(String[] args) {
5
6       List<CatalogItem> catalogItems = new ArrayList<>();
7       catalogItems.add(new Product("tv"));
8       catalogItems.add(new Product("radio"));
9
10      catalogItems.add(new Service("haircut"));
11      catalogItems.add(new Service("massage"));
12
13      int totalInventory = 0;
14      for(CatalogItem item : catalogItems) {
15        if (item instanceof Inventoryable) {
16          Inventoryable product = (Inventoryable)item;
17          totalInventory += product.getInventory();
18        }
19      }
20
21      System.out.println(totalInventory);
22    }
23  }
```

By casting item into **Inventoryable**, we can call the getInventory() method.

Interface Segregation Principle

A class should not be forced to implement anything that it doesn’t use.

```
11  public interface OrderProcessor {
12    void ship();
13    void refund();
14  }
15
16  public class ProductOrderProcessor implements OrderProcessor {
17    ship() {
18
19    }
20
21    refund() {
22
23    }
24  }
25
26  public class ServiceOrderProcessor implements OrderProcessor {
27    ship() {
28      // services don't need shipment
29    }
30
31    refund() {
32
33    }
34  }
```


ServiceOrderProcessor doesn't need to worry about shipments. So this method is left empty.

```
11  public interface Shippable {
12    void ship();
13  }
14
15  public interface Refundable {
16    void refund();
17  }
18
19  public class ProductOrderProcessor implements Shippable, Refundable {
20    ship() {
21
22    }
23
24    refund() {
25
26    }
27  }
28
29  public class ServiceOrderProcessor implements Refundable {
30    refund() {
31
32    }
33  }
```

More specific interfaces

iTJuana.

Dependency Inversion Principle

High level modules should not depend on low level modules. They should depend on abstractions.

ProductRepository.java

```java

public interface ProductRepository {
    List<Product> getAll();
    void save(Product product);
}
```

CatalogApplication depends on the interface (abstraction), not on the concrete implementation.

CatalogApplication.java

```java
public class CatalogApplication {

    @Inject
    private ProductRepository productRepository;

    public List<ProductInfo> listProducts() {
        List<Product> productList = productRepository.getAll();
        return productList.stream().map(ProductInfoMapper::map).collect(Collectors.toList());
    }

    public void createProduct(CreateProductRequest request) {
        Product product = Product.buildNewProduct(request.getProductName());
        productRepository.save(product);
    }
}
```

InMemoryProductRepository.java

```java
public class InMemoryProductRepository implements ProductRepository {

}

public class MySQLProductRepository implements ProductRepository {

}
```

Concrete implementations can change without impacting depending clients because they adhere to the **interface contract**.

In Summary

- SOLID is guidance to write maintainable code. It requires **spending more time writing the code**, so you can **spend less time reading it** afterwards.
- SOLID principles are principles, not a checklist.
- Always know your trade-offs and use common sense.
- SOLID is your tool, not your goal.

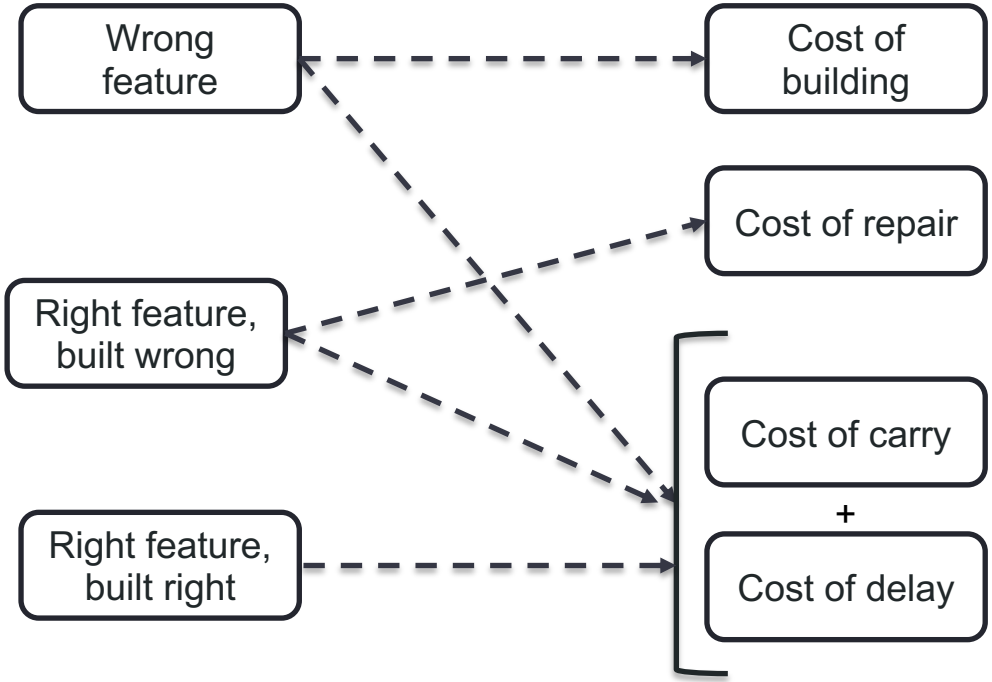
YAGNI – You aren’t gonna need it

- Simple Design
- Don’t add functionality until is necessary
- Is meant to be used in combination with other practices
- Unit Test
- Refactoring
- TDD
- CI

YAGNI - Example

- You’re building the next browsing engine (ITJSearch).
- Metaverse is now officially announced by Meta and is going to change the way we search things in 8 months.
- Start optimizing and adapting the engine for the Metaverse.
- YAGNI Principle is against this.

YAGNI


Wrong feature
Cost of building
Cost of repair
Right feature, built wrong
Cost of carry
+
Cost of delay
Right feature, built right

DRY – Don’t repeat yourself

A single piece of information should be present in only one place and in an authoritative manner in your system.

DRY – Don’t repeat yourself

- Relies on SRP
- Think of code that will be reusable and try to form utility classes
- Keep your code simple
- Divide your logic in smaller pieces

Design Patterns and Architectures commonly used in Android

Design Pattern Definition

Solutions to general problems that most of the developers have faced during object-oriented software development.

Design Pattern Types

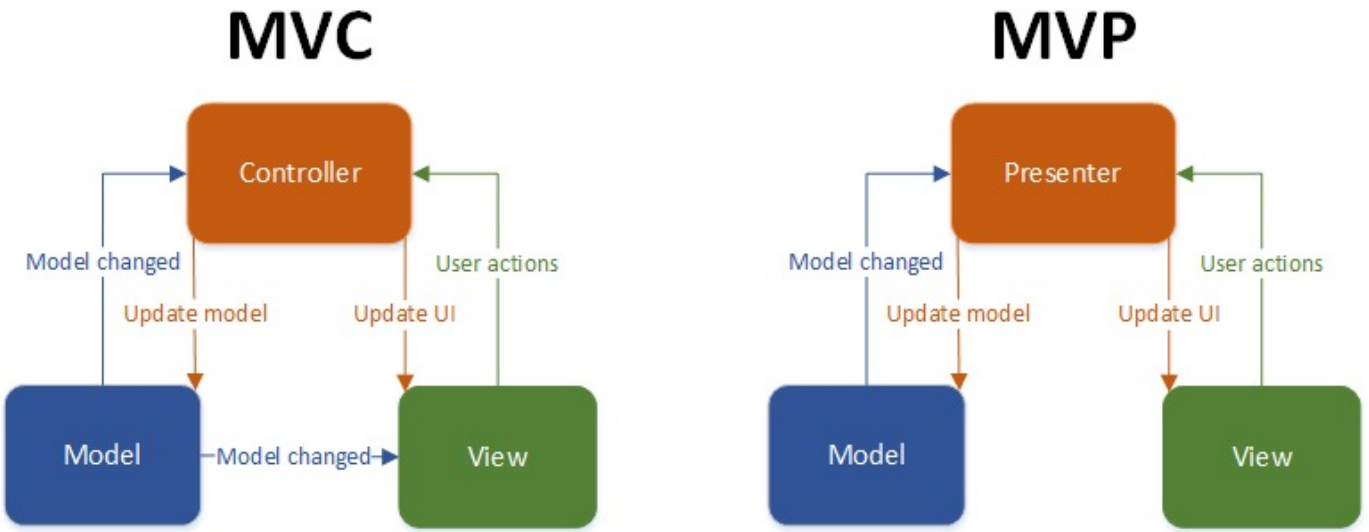
- Creational Patterns
- Structural Patters
- Behavioral Patterns

Design Pattern + Architectures in Android

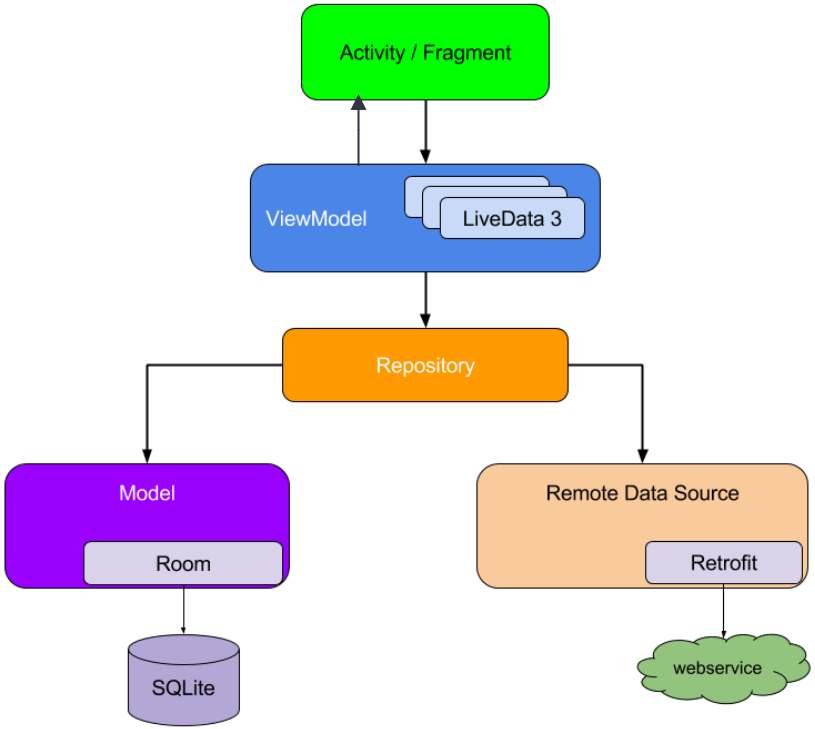
- Creational - Builder, Factory and Singleton
- Behavioral – Iterator and Observer
- Structural – Adapter and Facade

- MVC – Model View Controller
- MVP - Model View Presenter
- MVVM – Model-View ViewModel

Architectures – MVC vs MVP



Architectures – MVVM

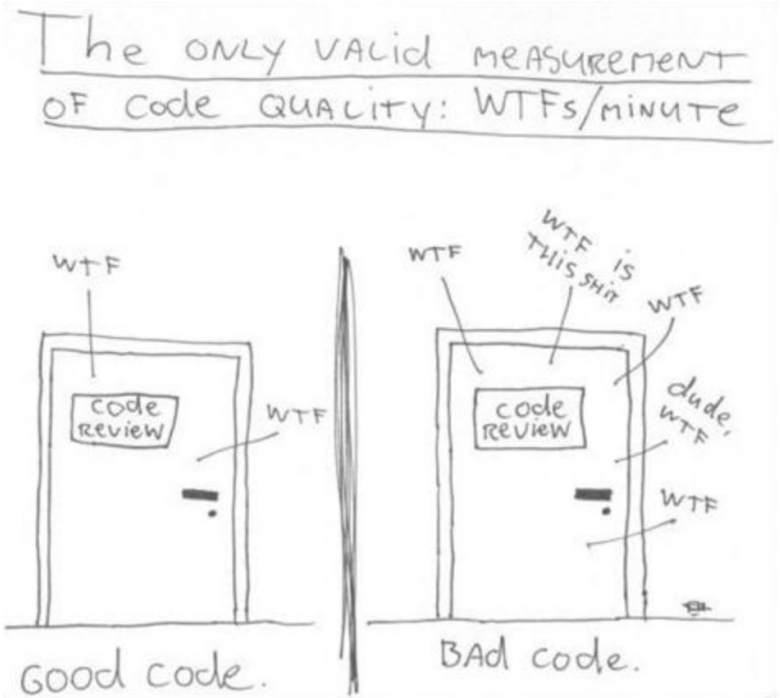


Clean Code

What is Clean Code?

Clean code is code that is easy to understand and easy to change.

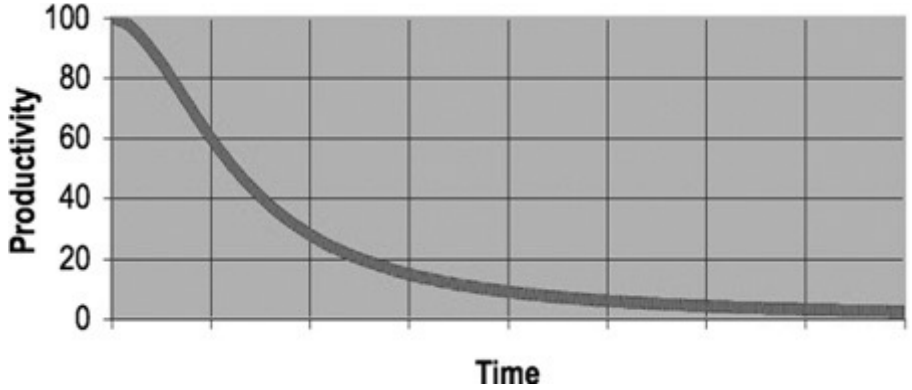
How to measure code Quality?



Images from Robert C. Martin, Clean code book

There will always going to be Bad Code

- Bad code is always to be there. Good code is as good as the requirements are specified.



Images from Robert C. Martin, Clean code book

Use Meaning full names

- Use intention revealing names
- Avoid missinformation

Good	Bad
`int days;`	`int d; // Elapsed days`
`int hours;`	`int h; // Hours`

Good	Bad
`float hypotenuse;`	`float hp;`
`List<String> accounts;`	`List<String> accountList;`

Use Meaning full names

Good

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<>();
    for (int[] cell : gameBoard) {
        if (cell[STATUS_VALUE] == FLAGGED) {
            flaggedCells.add(cell);
        }
    }
    return flaggedCells;
}
```

Bad

```
public List<int[]> getThem() {
    List<int[]> list = new ArrayList<>();
    for (int[] x : theList) {
        if (x[0] == 4) {
            list.add(x);
        }
    }
    return list;
}
```

Images from Robert C. Martin, Clean code book

Use searchable names

- Single letter names or abbreviations are hard to find

```
public double getHp(double b, double a) {
    return Math.pow(b, 2) + Math.pow(a, 2);
}
```

To

```
public double calculateHypotenuse(double base, double height) {
    return Math.pow(base, QUADRATIC_POWER) + Math.pow(height, QUADRATIC_POWER);
}
```

Functions

- 1st Rule of Functions: They should be small
- 2nd Rule of Functions: They should be smaller than that!
- Do one Thing
- One level of abstraction per Function
- Has no side effects
- Command Query Separation
- Prefer exceptions over ErrorCodes

```
public boolean checkPassword(String userName, String password) {
    User user = UserGateWay.findByName(userName);
    if (user != null) {
        String codedPhrase = user.getPhraseEncodedByPassword();
        String phrase = cryptographer.decrypt(codedPhrase, password);
        if ("Valid Password".equals(phrase)) {
            Session.initialize();
            return true;
        }
    }
    return false;
}
```

Comments

- Comments Do not Make Up for Bad Code
- Better explain yourself in code

```
// Check if employee should receive bonus
if ((employee.daysWorkedInCurrentYear >= MIN_DAYS_WORKED_FOR_BONUS &&
        employee.performanceEvaluationRating >= MIN_PERFORMANCE_RAITING_FOR_BONUS) ||
        employee.yearsInService >= YEARS_IN_SERVICE_FOR_BONUS ) {
    // Give bonus to employee
    // ...
}
```

```
    if (shouldEmployeeReceiveBonus(employee)) {
        // Give bonus to employee
        // ...
    }
```

Comments

- Good comments (Some are necessary or beneficial)
- Legal Comments
- Informative Comments
- Warning or Consequences
- TODO Comments
- Javadocs

Objects and Data Structures

- Data abstraction

```
// Concrete Vehicle
public interface Vehicle {
    double getGallonsOfGasoline();

    double getPerformancePerGalon();
}
```

```
// Abstract Vehicle
public interface Vehicle {
    double getKilometersLeft();
}
```

Error handling

- Use exception rather than return codes
- Write your Try-Catch-Finally first (Fail as early as possible)
- Provide Context With Exceptions
- Define Exceptions Classes in Terms of Caller’s needs.
- Don’t pass nulls
- Don’t return nulls

Summary

- Follow Standard Conventions
- Boy Scout Rule (Leave the campground cleaner than you found it.)
- Root Cause analysis
- Don’t make unnecessary comments

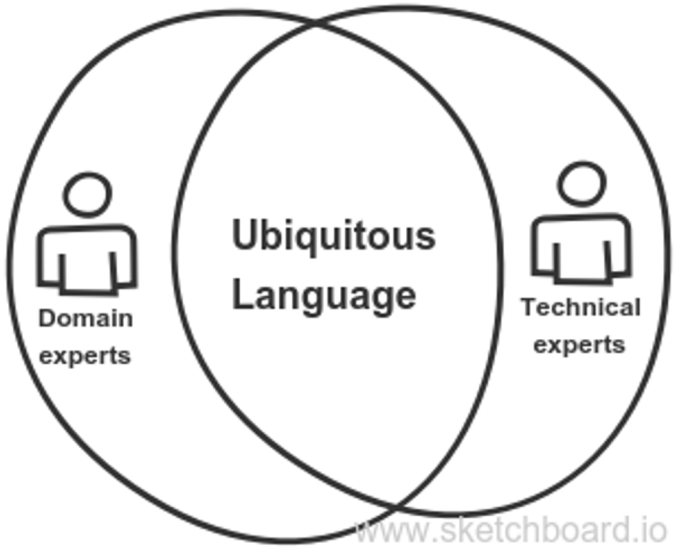
Domain Driven Design (DDD)

Domain-Driven Design (DDD)

Domain-driven design (**DDD**) is software development technique used to build clean and robust applications based on the **deep understanding of the business domain**, requirements and goals.

Ubiquitous Language

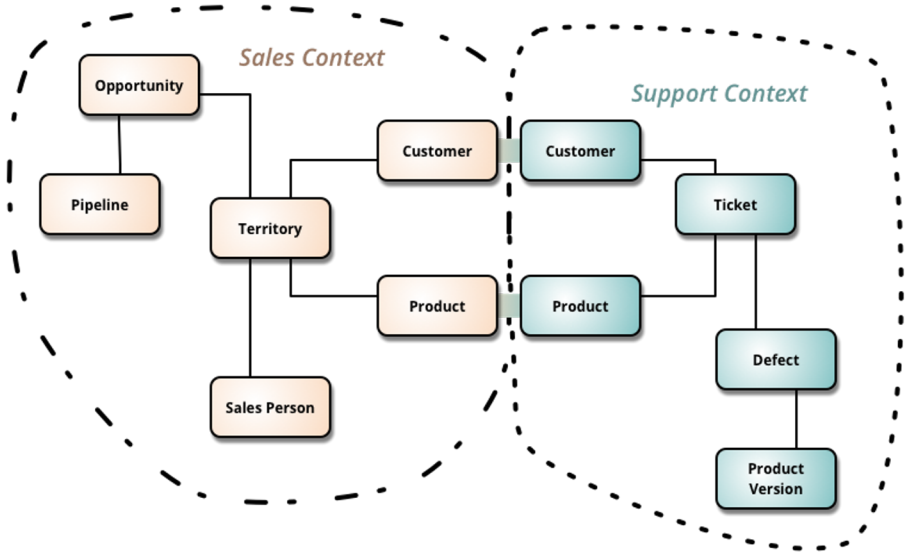
Building up a common, rigorous language between developers, business experts, users, and everyone involved.



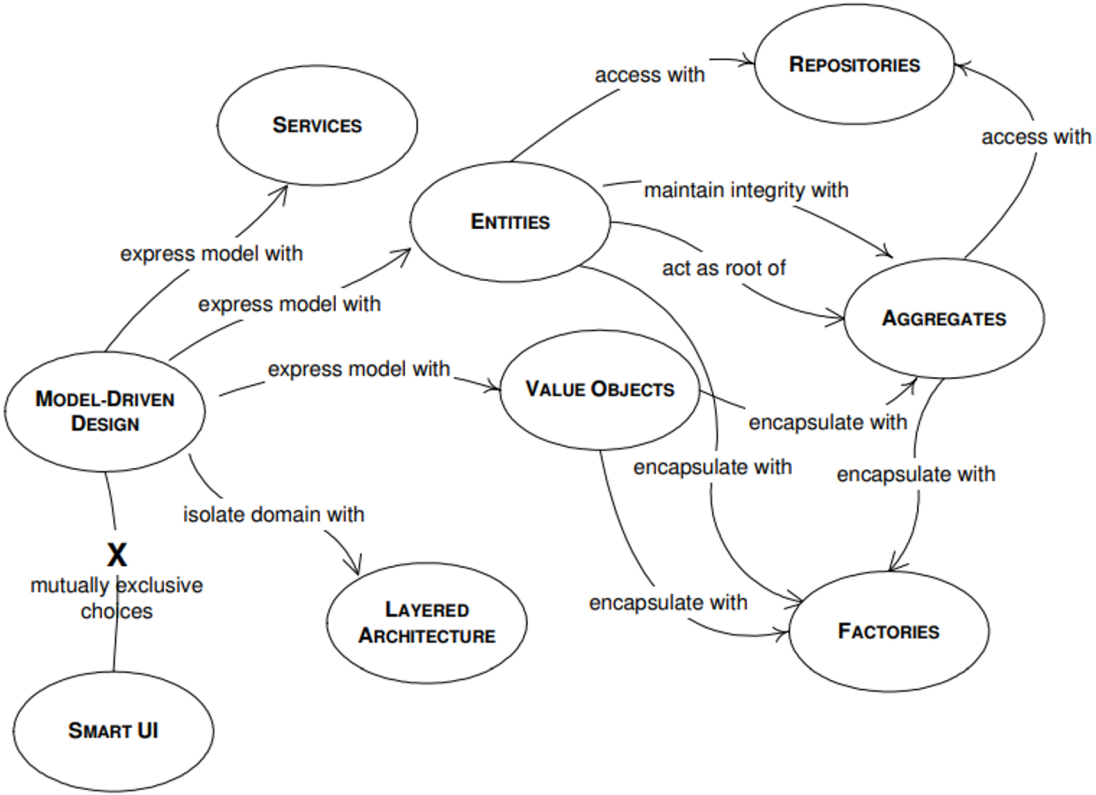
Bounded Context

Bounded Context is a central pattern in Domain-Driven Design, it is the boundary within a domain where a particular domain model applies.

DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.



Building Blocks of DDD

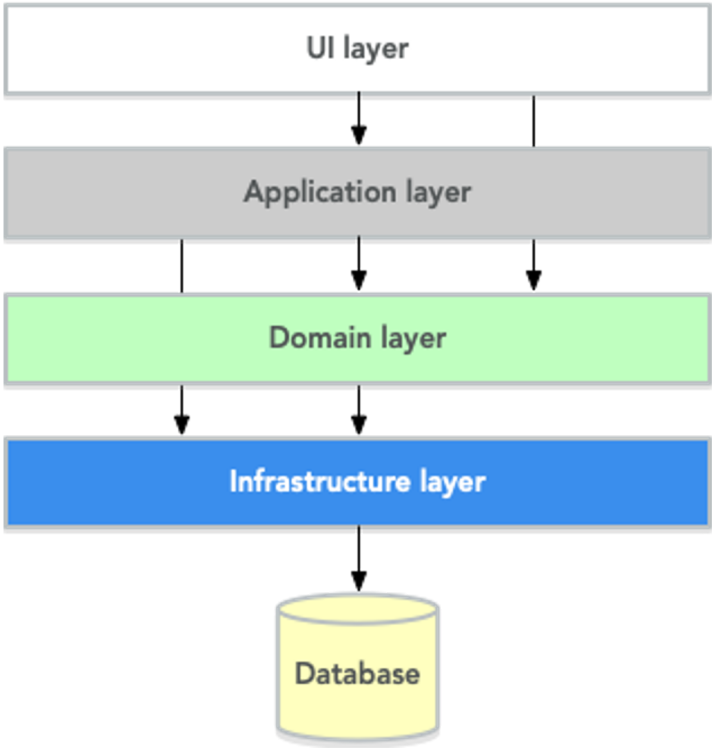


Layered Architecture

Separate your code in cohesive layers.

Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code.

This allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work, and, **to be easily tested**.



Entities & Value Objects

Entities: Objects which seem to **have an identity**, which remains the same throughout the states of the software. It is not the attributes which matter, but a thread of continuity and identity, which **spans the life of a system** and can extend beyond it.

Value Objects: An object that is used to **describe** certain aspects of a domain, and which **does not have identity**. We are not interested in which object it is, but what attributes it has. There are cases when we need to contain some attributes of a domain element.

```
// Entity
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

// Value Object
public class Address
{
    public string City { get; set; }
    public string ZipCode { get; set; }
}
```

Services

Domain Service: An object that **does not have an internal state**, and its purpose is to simply provide functionality for the **domain**.

> *When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless. --* ***Eric Evans*** *Domain-Driven Design*

Infrastructural services are focused on encapsulating the “plumbing” requirements of an application; usually IO concerns such as file system access, database access, email, etc.

Aggregates

Aggregate is a domain pattern used to define object ownership and boundaries.

An Aggregate is a group of associated objects which are considered as one unit with regard to data changes.

Each Aggregate has **is an Entity**, and accessible from outside.

The root **can hold aggregate objects**, can hold references outside object can hold references only to the

```java
package de.mploed.ddd.battery;

import java.util.UUID;

class BatteryRootEntity {
        private UUID serialNumber;
        private CellBlock cellBlock;
        private CellCharger cellCharger;

        BatteryRootEntity(int amountOfCells, int capacityForCell) {
                this.serialNumber = UUID.randomUUID();
                this.cellBlock = new CellBlock(amountOfCells, capacityForCell);
                this.cellCharger = new CellCharger();
        }
```

See example

Factories

Factories is a design pattern which help us deal with object creation. Entities and Aggregates can often be large and complex – too complex to create in the constructor of the root entity.

A Factory Method is an object method which contains and hides knowledge necessary to create another object.

```
public class ImageReaderFactory {
    public static ImageReader createImageReader(ImageInputStreamProcessor iisp) {
        if (iisp.isGIF()) {
            return new GifReader(iisp.getInputStream());
        }
        else if (iisp.isJPEG()) {
            return new JpegReader(iisp.getInputStream());
        }
        else {
            throw new IllegalArgumentException("Unknown image type.");
        }
    }
}
```

Visit reference

Repositories

Repository is a design pattern which **help us deal with storage**.

The purpose of a Repository is to **encapsulate all the logic needed to obtain object references**. The domain objects won't have to deal with the infrastructure to get the needed references to other objects of the domain.

They will just get them from the Repository and the model is regaining its clarity and focus.

The **implementation of a repository can be closely linked to the infrastructure**, but that the r**epository interface will be pure domain model**.

Repositories

Interface / **Domain layer**

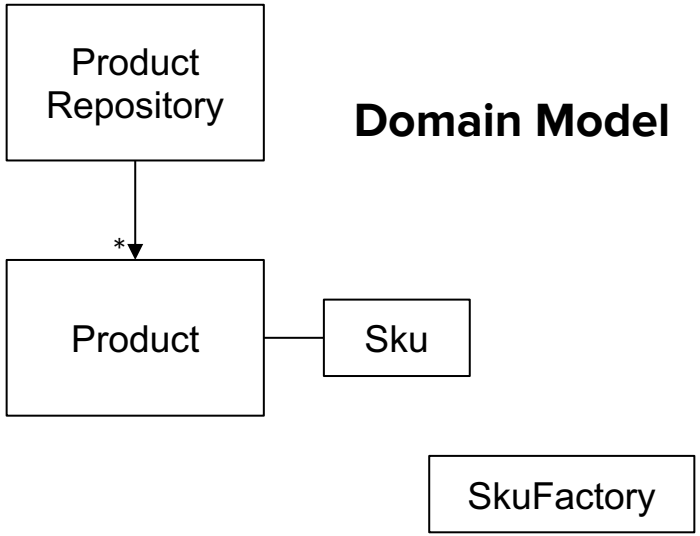
```java
package com.itjuana.library.domain.book;

import com.itjuana.library.domain.author.AuthorId;

import java.util.List;

public interface BookRepository {

    List<Book> findByTitle(BookTitle bookTitle);
    List<Book> findByAuthor(AuthorId authorId);
    Book findByISBN(BookISBN bookISBN);

}
```

Implementation / **Infrastructure layer**

```java
public class BookRepositoryMemory implements BookRepository {

    List<Book> books;

    // This is a dumb and inefficient example
    public BookRepositoryMemory() {...}

    @Override
    public List<Book> findByTitle(BookTitle bookTitle) {
        List<Book> books = new ArrayList<~>();
        // Can you optimize this with the Stream API?
        for(int i=0; i<this.books.size(); i++) {
            if(this.books.get(i).getBookTitle().equals(bookTitle))
                books.add(this.books.get(i));
        }
        return books;
    }

    @Override
    public List<Book> findByAuthor(AuthorId authorId) {
        List<Book> books = new ArrayList<~>();
        // Can you optimize this with the Stream API?
        for(int i=0; i<this.books.size(); i++) {
            if(this.books.get(i).getAuthor().getAuthorId().equals(authorId))
                books.add(this.books.get(i));
        }
        return books;
    }
```

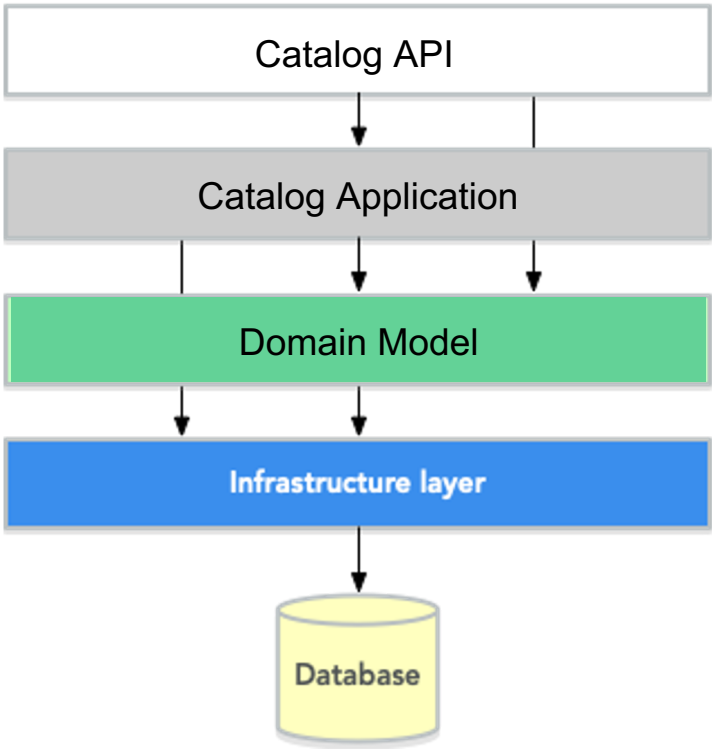
Domain Driven Design

Scenario: A product catalog for an e-commerce website.



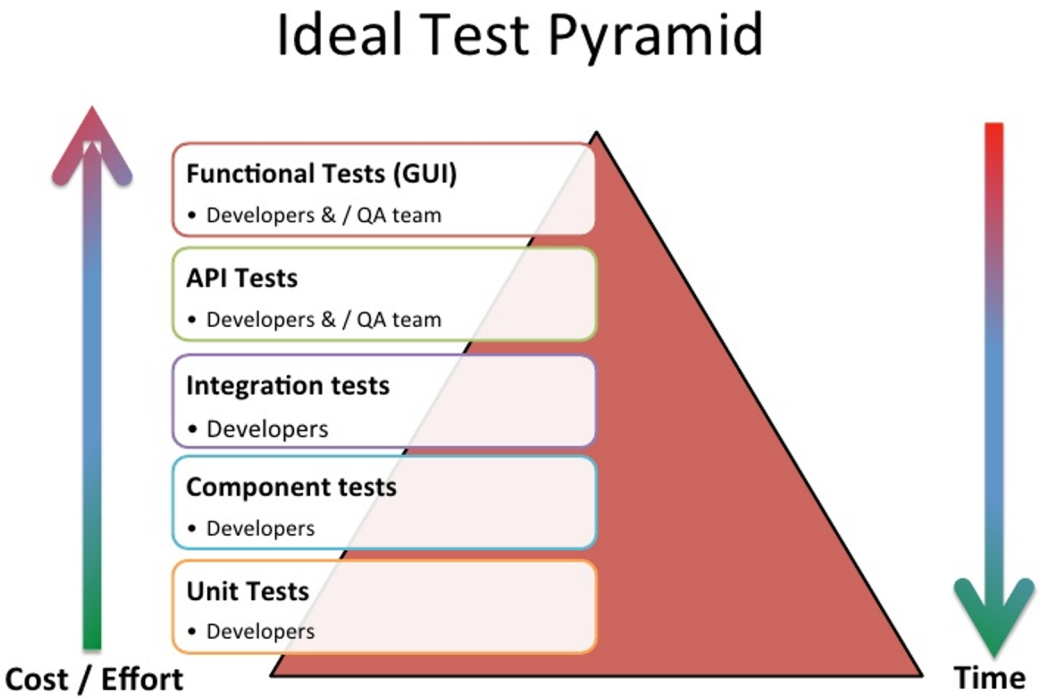
Domain Model

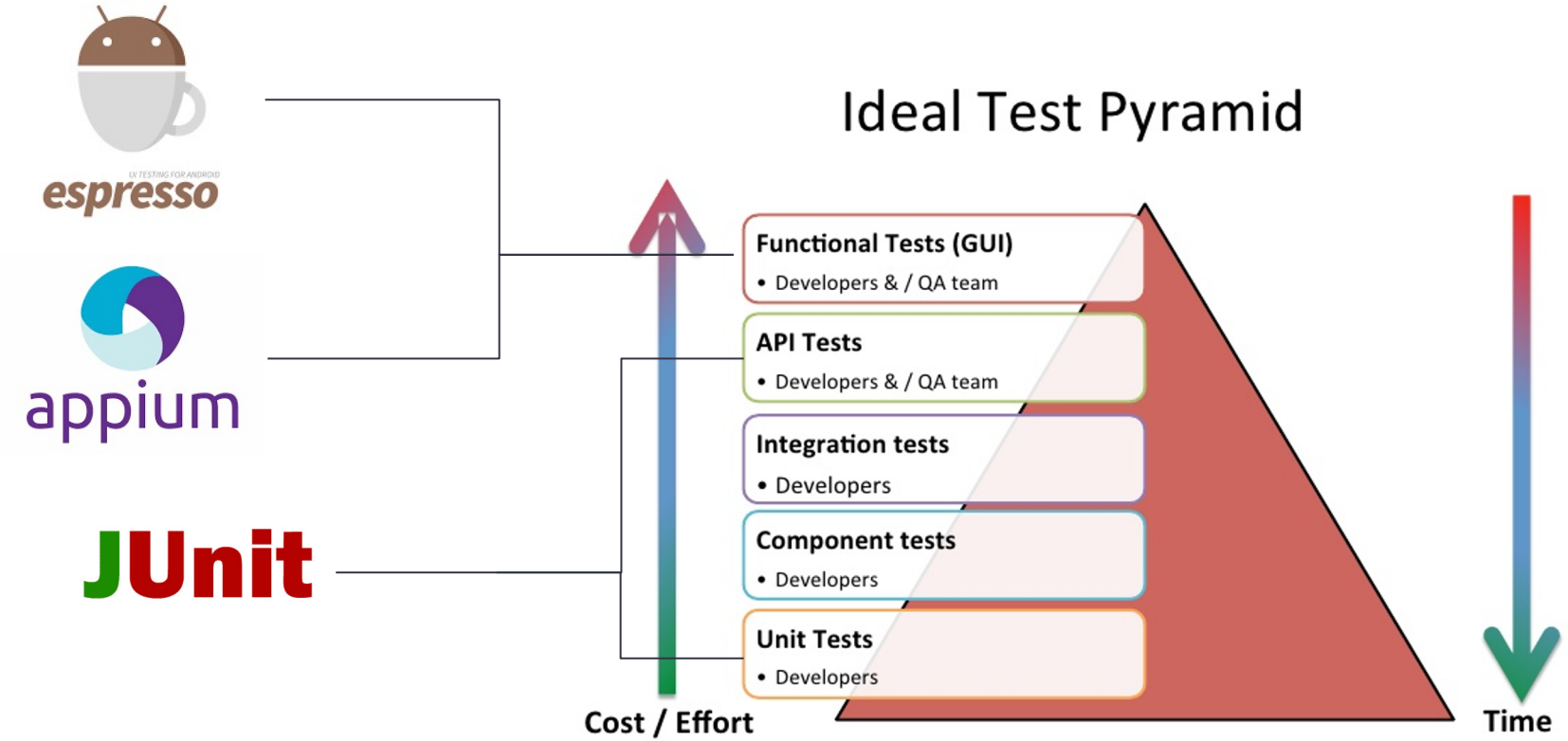
Project Structure



Test Driven Development (TDD)

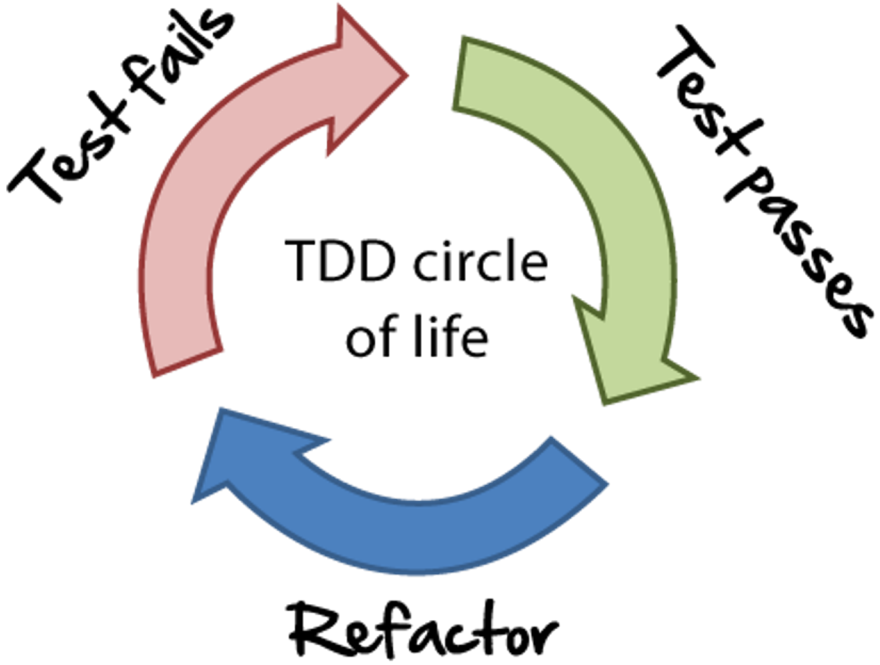
Testing Pyramid




espresso
UI TESTING FOR ANDROID
appium
JUnit
Ideal Test Pyramid
Functional Tests (GUI)
• Developers & / QA team
API Tests
• Developers & / QA team
Integration tests
• Developers
Component tests
• Developers
Unit Tests
• Developers
Cost / Effort
Time

Why do we write unit tests?

Testing-Driven Development in **3 steps**



Test Driven Development - Labs

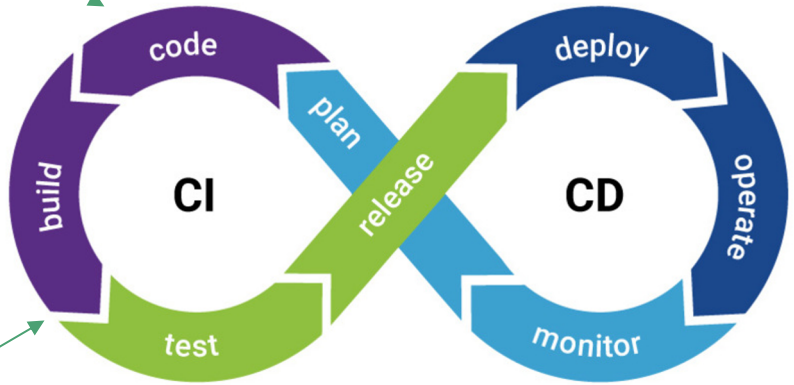
Continuous Integration / Continuous Delivery (CI/CD)

CI/CD is a method to frequently deliver apps to customers by introducing **automation** into the stages of app development.

- Version control
- Peer Code Reviews

Strategies

- Canaries
- Blue/Green



- Compile code
- Unit tests
- Style checkers
- Code metrics
- Create artifacts

- Integration tests
- UI Testing

Optional / Infrequent / Expensive

- Load testing
- Penetration testing

CI

The "CI" in CI/CD always refers to continuous integration, which is an automation process for developers.

Successful CI means new code changes to an app are regularly built, tested, and merged to a shared repository.

CD

The "CD" in CI/CD refers to continuous delivery and/or continuous deployment, which are related concepts that sometimes get used interchangeably. Both are about automating further stages of the pipeline, but they’re sometimes used separately to illustrate just how much automation is happening.

Continuous delivery usually means a developer’s changes to an application are automatically bug tested and uploaded to a repository (like GitHub or a container registry), where they can then be deployed to a live production environment by the operations team. It’s an answer to the problem of poor visibility and communication between dev and business teams. To that end, the purpose of continuous delivery is to ensure that it takes minimal effort to deploy new code.

CD

Continuous deployment (the other possible "CD") can refer to automatically releasing a developer's changes from the repository to production, where it is usable by customers. It addresses the problem of overloading operations teams with manual processes that slow down app delivery. It builds on the benefits of continuous delivery by automating the next stage in the pipeline.

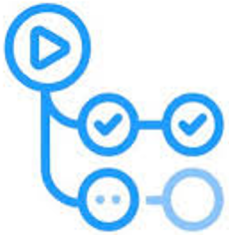
Release & Deploy

Release is about publishing an artifact (jar, docker image, apk) to a binary repository (maven, docker hub, artifactory)

Deploy is about launching the binary on the infrastructure

java -jar build/libs/ddd-ecommerce-0.1-all.jar

Common CI/CD tools

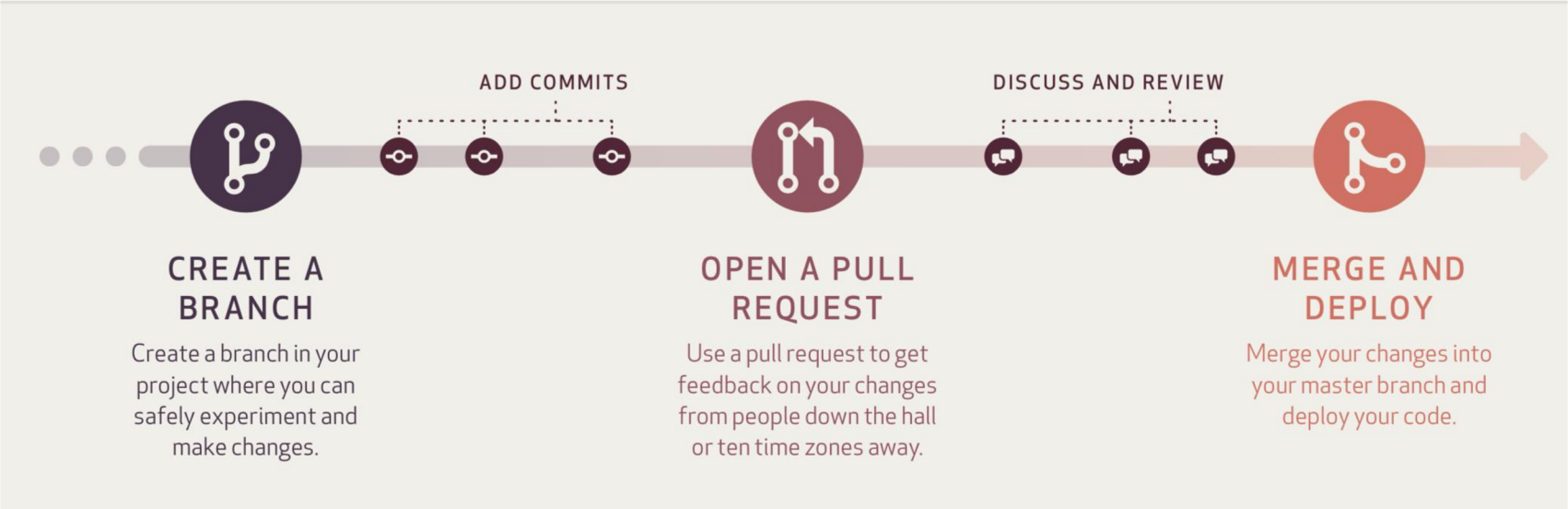


IaaS

In Summary

- **Continuous Integration** is about the **quality** of the code, frequently checked.
- **Continuous Delivery** is about automating the release process to multiple environments, to reduce error human-prone manual steps.

Git Workflow



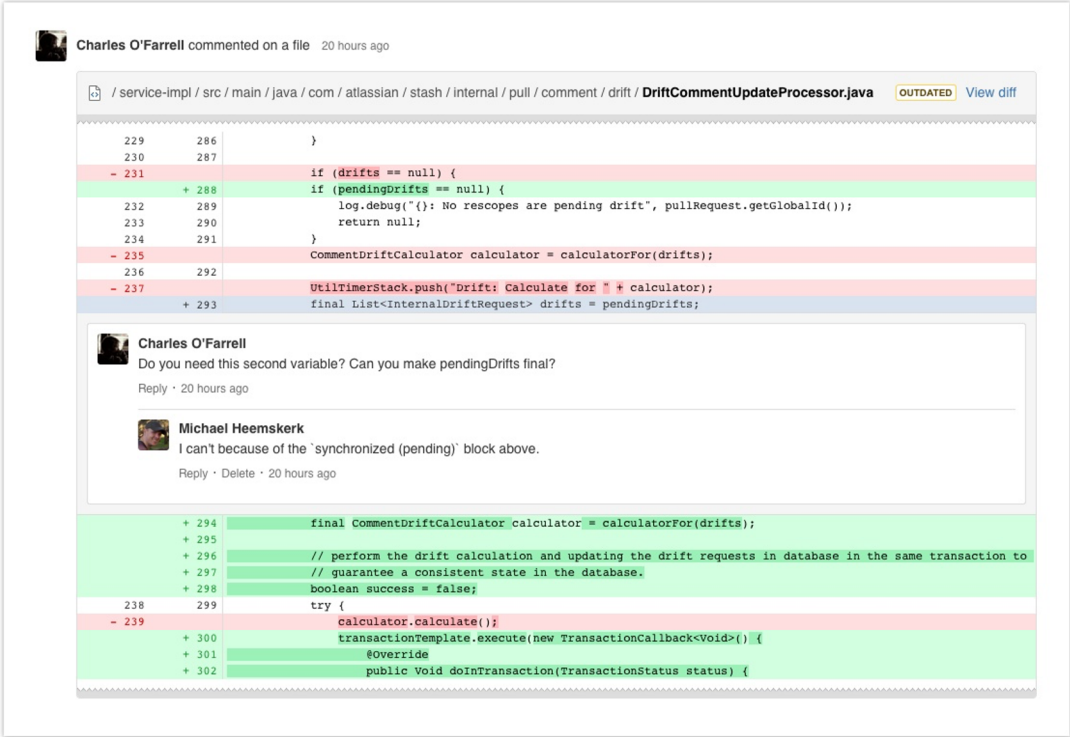
Code Reviews

Rules

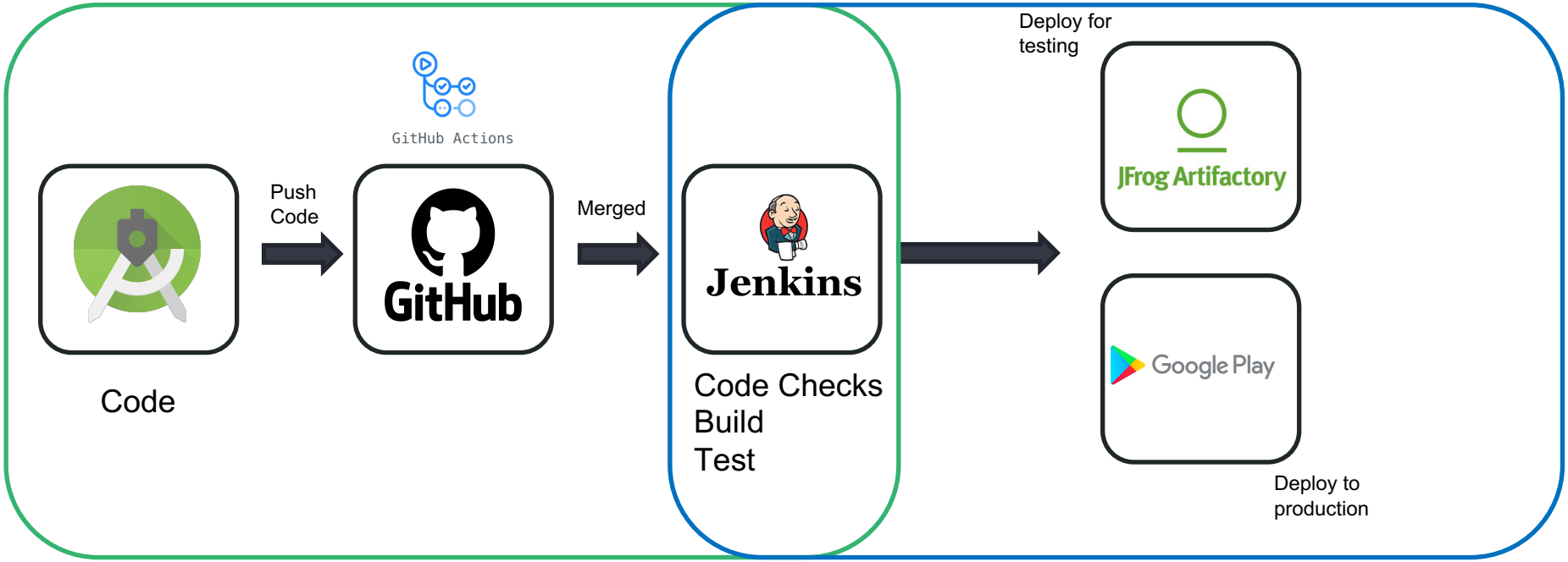
- Author invites all the dev team.
- 2 approvals are enough to merge.

Benefits

- Code quality improves
- Knowledge transfer



Android CI/CD Pipeline



Summarizing

1. The goal of CI/CD is to be ready to release on every commit.
   This doesn't necessarily mean that you will do it.

2. The goal of a pipeline is to prevent a defective release to make it into prod.

3. Encourage Continuous Integration habits. (Commit and push every day, pull continuously, avoid long lived branches, automate builds and tests).

Continue Learning

Clean Code - Uncle Bob - 6 hours of video
https://youtu.be/7EmboKQH8lM

Clean Architecture Book
https://www.amazon.com/Clean-Architecture-Craftsmans-Software-Structure/dp/0134494164

Domain Driven Design Book
https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215

Design Patterns - Head First Book
https://www.amazon.com/Head-First-Design-Patterns-Brain-Friendly/dp/0596007124

Is TDD dead? A debate between Martin Fowler, Kent Beck and David Heinmeier
https://martinfowler.com/articles/is-tdd-dead/