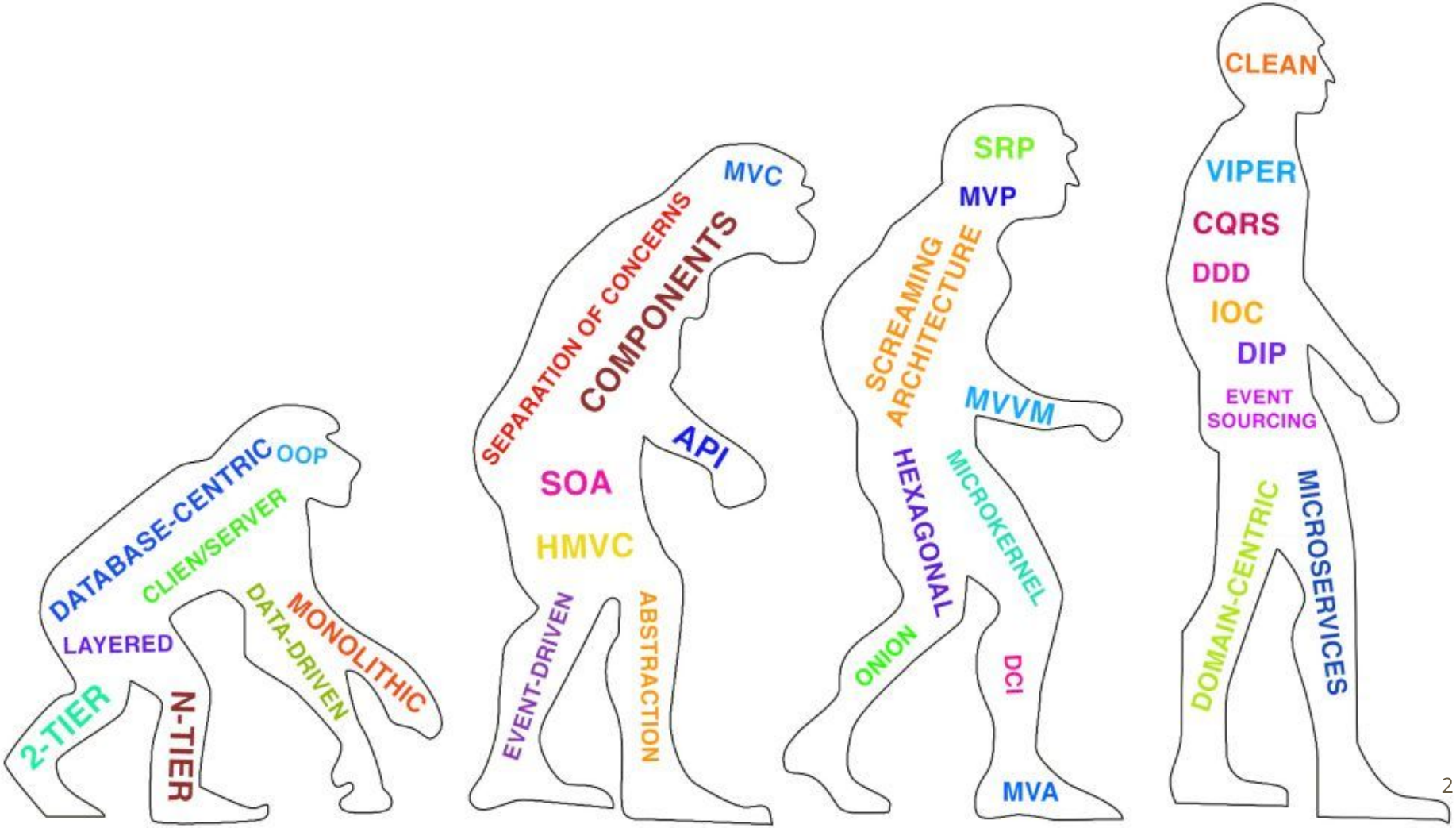
The Clean Architecture

@UINT48


2-TIER
N-TIER
LAYERED
DATABASE-CENTRIC
CLIEN/SERVER
OOP
DATA-DRIVEN
MONOLITHIC
EVENT-DRIVEN
ABSTRACTION
HMVC
SOA
SEPARATION OF CONCERNS
COMPONENTS
API
MVC
ONION
HEXAGONAL
SCREAMING ARCHITECTURE
MICROKERNEL
MVVM
MVP
SRP
DCI
MVA
DOMAIN-CENTRIC
MICROSERVICES
EVENT SOURCING
DIP
IOC
DDD
CQRS
VIPER
CLEAN

Agenda

- From STUPID to SOLID
- What is Clean Code?
- Clean Architecture 💎
  - (Entity/Domain) Layer
  - (Usecase/Service) Layer
  - (Repository/Store) Layer
  - (Delivery/Transport) Layer
  - Infrastructure (Frameworks and Drivers)
- GET YOUR HANDS DIRTY
- Final thoughts

From STUPID to SOLID

- Singleton Invasion
- Tight Coupling
- Untestability
- Premature Optimization
- Indescriptive Naming
- Duplication

- Single responsibility
- Open/closed
- Liskov substitution
- Interface segregation
- Dependency inversion

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

INTERFACE SEGREGATION

Tailor interfaces to individual clients' needs.

DEPENDENCY INVERSION

Would you solder a lamp directly to the electrical wiring in a wall?

What is Clean Code?

Does it work?

Is it clever?

Is it easy to understand?

Is it esthetic?

Is it short?

Is it testable?

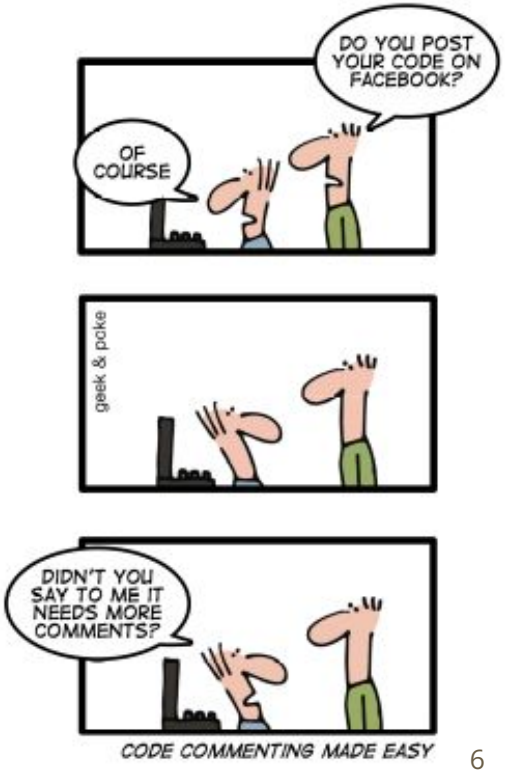
Is it well-structured?

Does it scale?

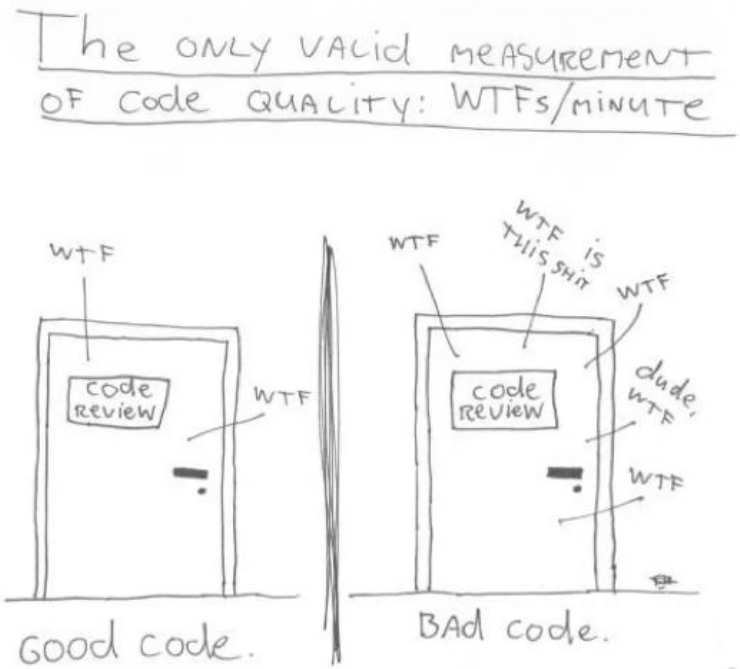
Is it object-oriented?

Does it have explanatory comments?

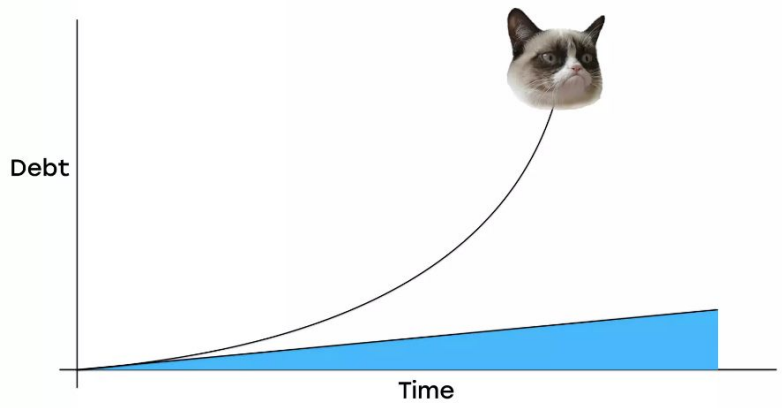
Is it maintainable?



What is Clean Code?



Overly complex code = warning that code is "**bad**" = danger of **bugs**



What is Clean Code?

- Focused
- Single-minded attitude
- Undistracted and unpolluted
- Readable, simple and direct
- Compact and literate
- Contains only what is necessary
- Makes it easy for other developers to enhance it
- Tests should be also clean
- Looks like it’s author cares
- Contains no duplicates
- Foundations are established on tiny abstractions

What is Clean Code?

- Don’t build complex machines (KISS).
- Make the software so simple that there are obviously no deficiencies
- No method should be longer than ~50 lines. If it becomes too long split it to more reusable methods.
- No class should have more then ~10 public operational methods (not including getters and setters). If it becomes too large split the responsibility and state it to reusable classes.
- Getters should not have side effect on the object state.
- The "main" method (or the operational public methods) should be readable almost as human English.
- If you find yourself Copy-Paste a chunk of code, you probably need to create a new method with this code and reuse it.

Clean Architecture

- Coined by Robert C.Martin
- Combination of various ideas
  - Hexagonal Architecture
  - Onion Architecture
  - DCI
  - Screaming Architecture
  - Single responsibility
  - ...
- Clean Architecture (2012)

Hexagonal Architecture
Alistair Cockburn (2005)

Onion Architecture
Jeffrey Palermo (2008)

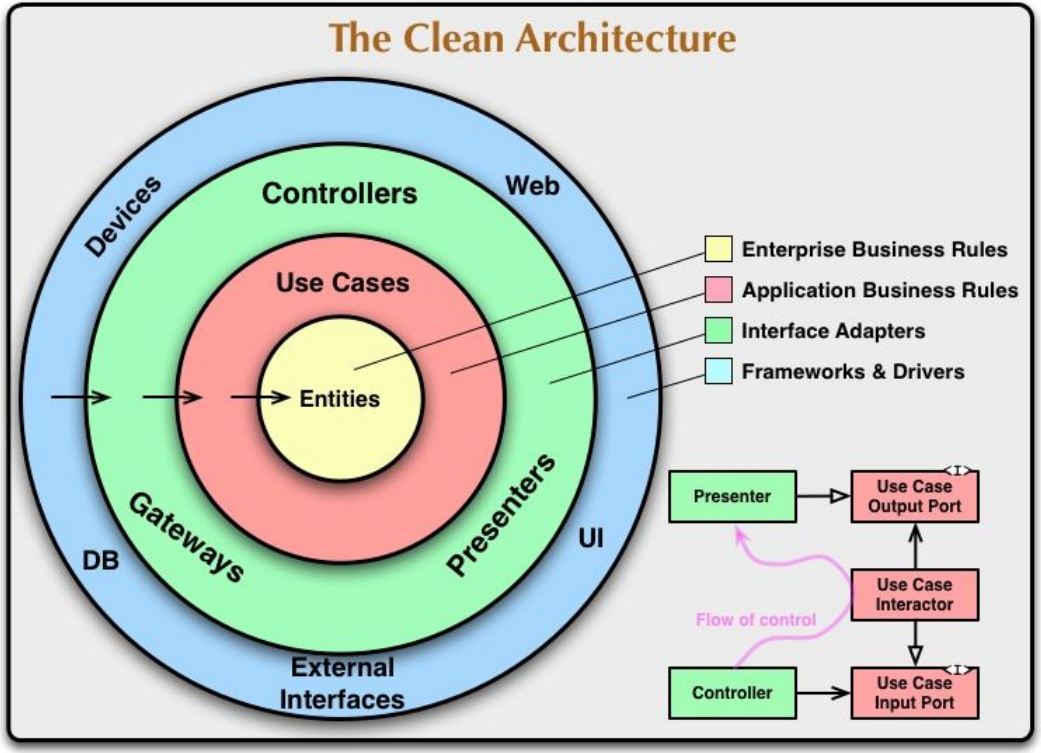
DCI (Data, Context, Interaction)
James O. Coplien (2010)

Screaming Architecture
Robert C. Martin (2011)

Clean Architecture

Clean Architecture combines a group of practices that produces systems with the following characteristics:

- Testable
- UI-Independent
- Independent of Infrastructure (databases,frameworks,libraries)

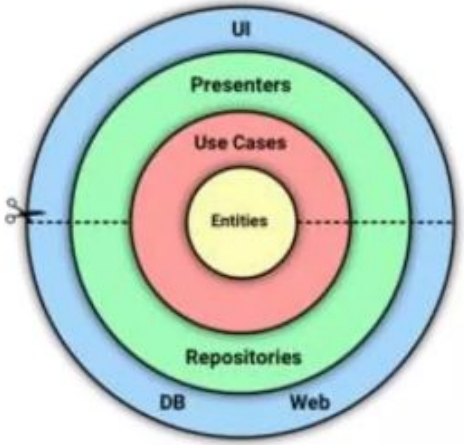


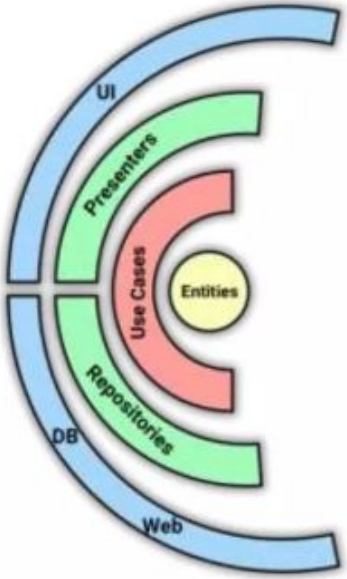
Clean Architecture

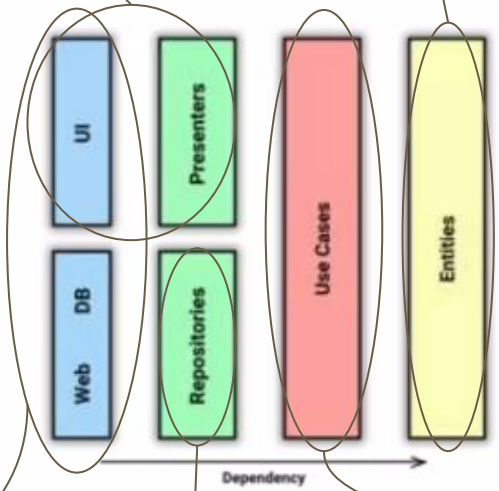
- The main topic in Clean Architecture is Dependency Inversion (SOLID)
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should be depend on abstractions.
  - Source code dependencies can only point inwards!
  - Nothing in an inner circle can know anything at all about something in an outer circle.
- A big part of it is abstracting away implementation details, a standard in technology, especially software.
- Separation of concerns, it exists on several levels. There are structures, namespaces, modules, packages, and even (micro)services.

(Delivery/Transport) Layer

(Entity/Domain) Layer



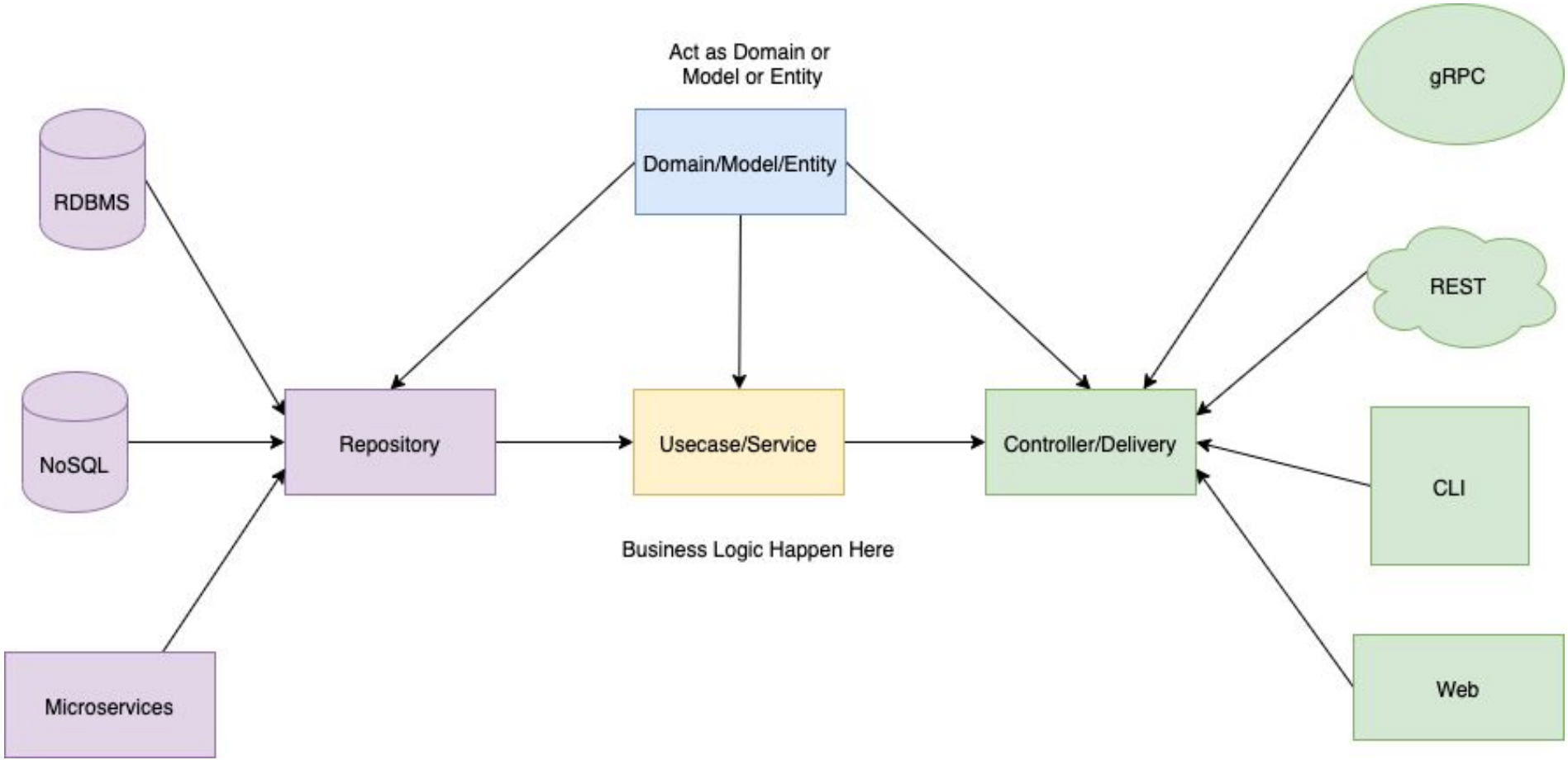




(Usecase/Service) Layer

Infrastructure (Frameworks and Drivers)

(Repository/Store) Layer


Act as Domain or
Model or Entity
Domain/Model/Entity
RDBMS
NoSQL
Microservices
Repository
Usecase/Service
Business Logic Happen Here
Controller/Delivery
gRPC
REST
CLI
Web

(Entity/Domain) Layer

(Entity/Domain) Layer

The Entity layer, also known as the Domain layer, is where the **business entities** are defined. These entities encapsulate the most fundamental business objects. They are the core of the business logic but are kept simple and focused on the business domain.

- **Independence**: Entities are independent of any specific technology or external agency. They are not concerned with how the data is stored or retrieved, or how the application is presented to the user.
- **Stability**: Entities are the most stable part of the system.

(Usecase/Service) Layer

(Usecase/Service) Layer

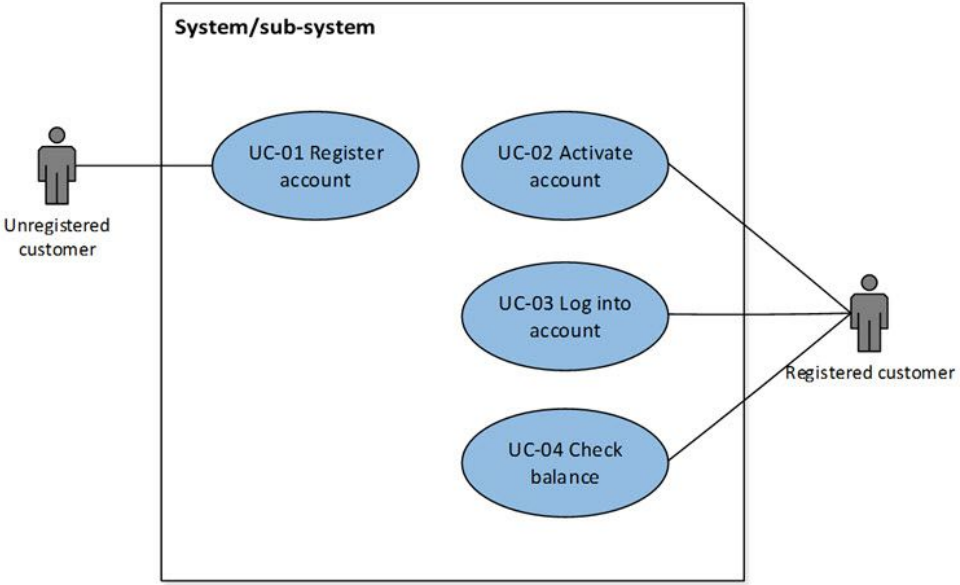
The **service layer** is responsible for implementing business logic (rules). While the service layer will depend on the repository interfaces to interact with the data, it should use the interfaces provided by the repository layer to perform its operations.

Two cases of business rules:

- Business specific business rules
  - For example: creating an account, login process, ...
- Application specific business rules
  - For example: validation, sanitization user inputs, ...

(Usecase/Service) Layer

- **Use Cases**: These are the high-level business operations that the system can perform. They encapsulate the business rules and logic that are specific to a particular action or operation. Use cases are the **heart of the application's business logic**



(Usecase/Service) Layer

The center of your application is not the database. Nor is it one or more of the frameworks you may be using. The center of your application is the use cases of your application - ***Uncle Bob***

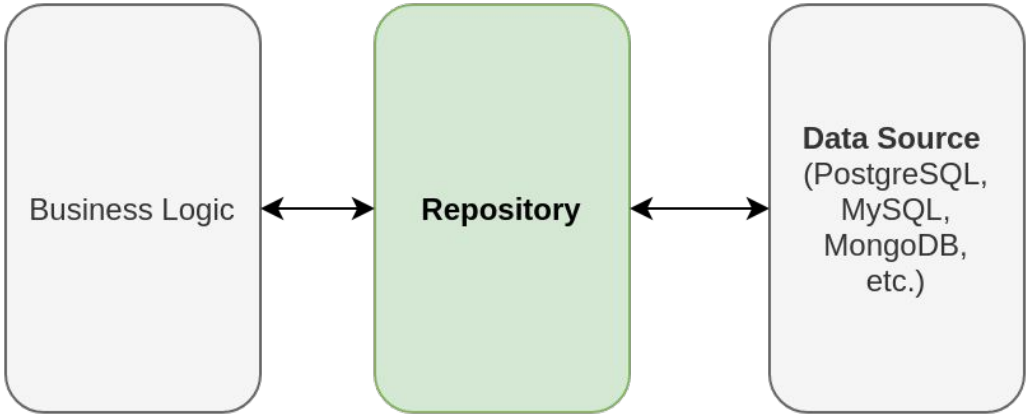
(Repository/Store) Layer

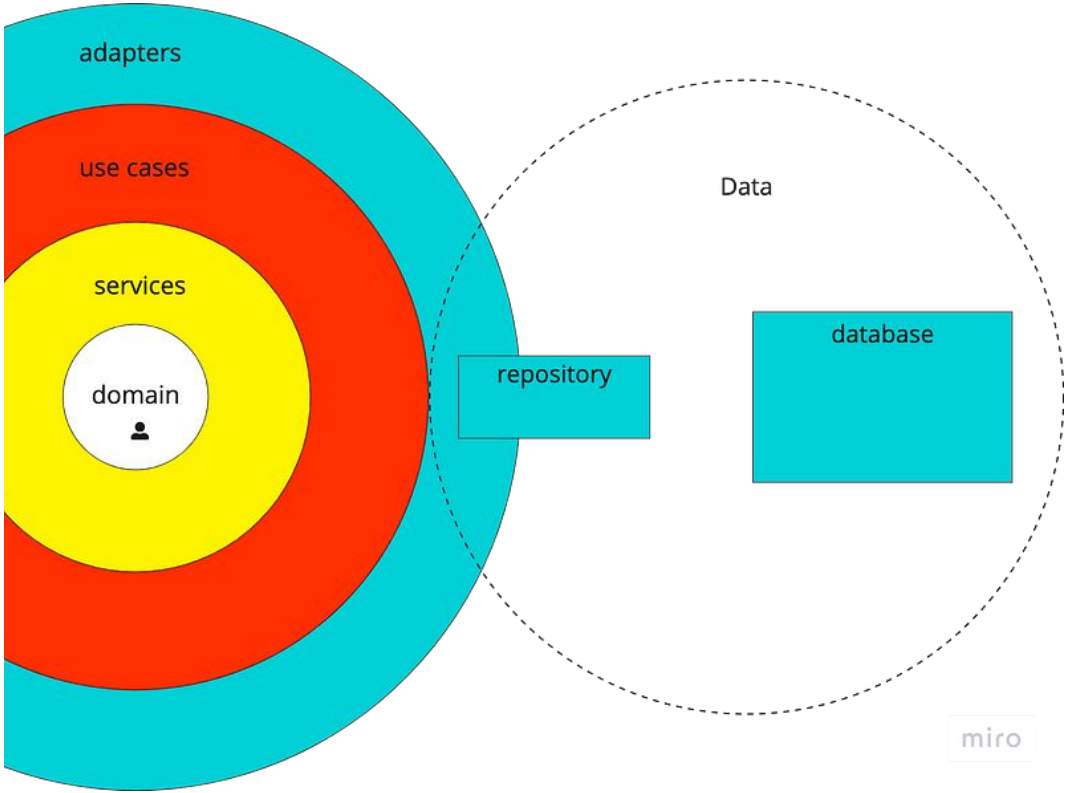
(Repository/Store) Layer

Repository layer is a crucial component that acts as an **abstraction** between the **data sources** (like databases or external services) and the business logic layer (typically the service layer). The Repository layer is responsible for retrieving and storing data, providing a clean and decoupled interface for the rest of the application to interact with data sources.

(Repository/Store) Layer

The Repository layer is part of the outer layers, which are closer to the infrastructure. It is used by the inner layers (like the service layer) to interact with data sources. This layering ensures that the core business logic remains isolated from external concerns.




adapters
use cases
services
domain
Data
repository
database
miro

(Delivery/Transport) Layer

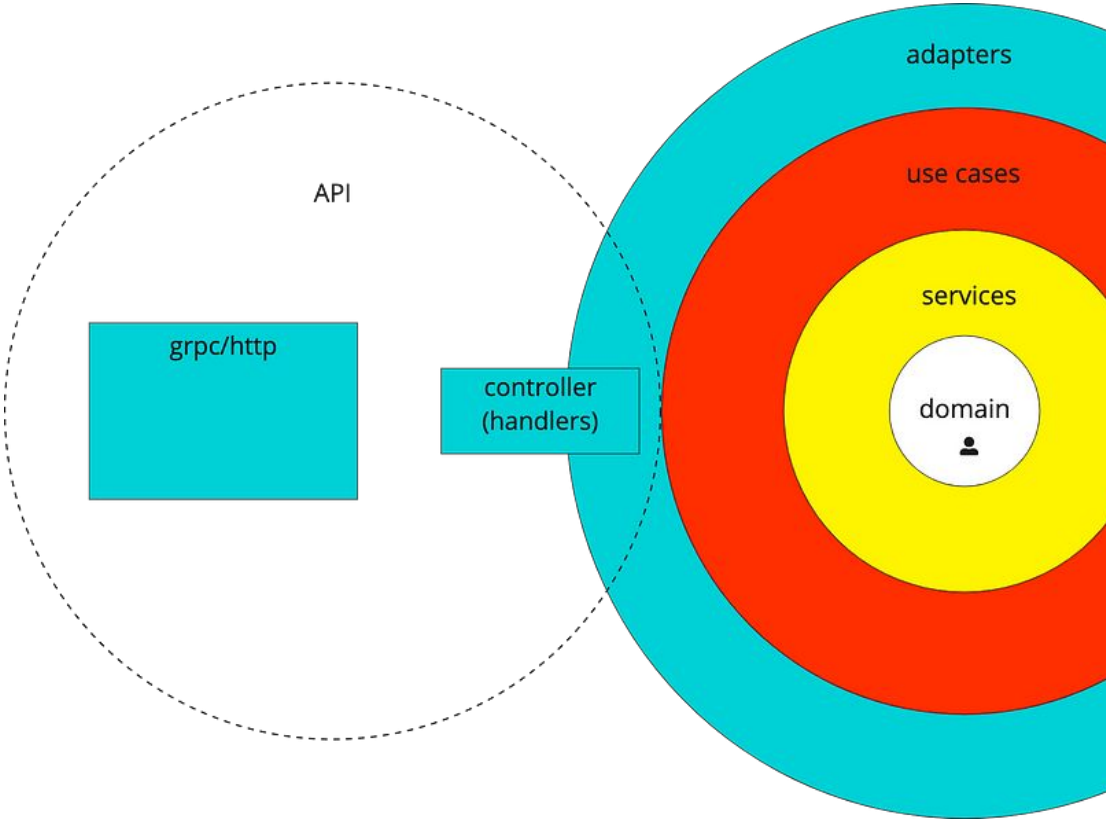
(Delivery/Transport) Layer

The Delivery/Transport Layer is part of the outer layers, which are closer to the infrastructure.

The Delivery Layer, also known as the **Interface Adapters** in Clean Architecture, is responsible for converting data between the format used by the inner layers (like Use Cases/Services) and the format used by the Delivery Layer (like http server or grpc server)

Common technologies used in the Delivery Layer:

- Web / REST
- HTTP
- gRPC
- WebSocket
- CLI / TUI / GUI
- Telegram Bot!
- ...


adapters
use cases
services
domain
API
grpc/http
controller
(handlers)

Infrastructure(Frameworks and Drivers)

Infrastructure(Frameworks and Drivers)

In Clean architecture we INVERT software dependencies for:

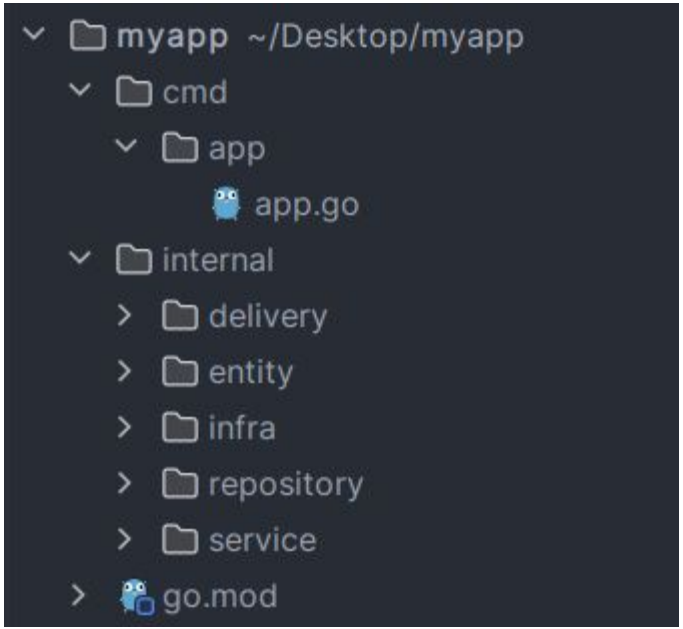
- User Interface
- Database
- External Interfaces
- Web (eg: HTTP Requests)
- Devices (eg: Printers and Scanners)

The frameworks and drivers layer is where all the details go. The web is a detail. The database is a detail. We keep these things on the outside where they can do little harm!

GET YOUR HANDS DIRTY

Project structure

Note: This code is more a **pseudocode** for showing the concept and is not complete and runnable.



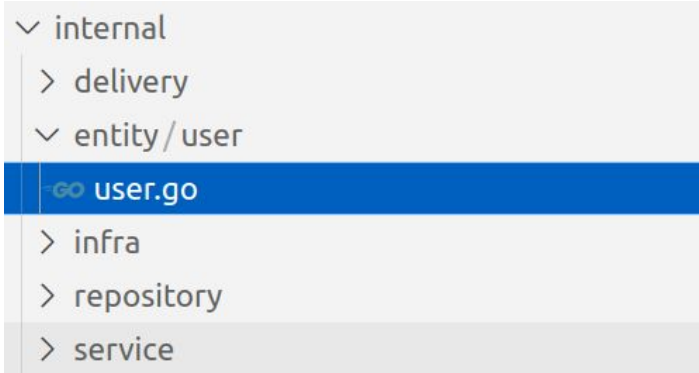
https://github.com/uint48/clean-architecture-presentation

User Entity (Entity Layer)

```go
package user

// role values
const (
    RegisteredUserRole = 1
    NotRegisteredRole  = 0
)

type User struct {
    ID       string
    Username string
    Password string
    Email    string
    IsActive bool
    Role     int
    Balance  float64
}
```



User Service (Service Layer)

userservice.go

internal > service > userservice > userservice.go > ...

```go
package userservice

import (
    "errors"
    "myapp/internal/entity/user"
    "myapp/internal/repository"
)

type UserService interface {
    Register(user *user.User) error
    Activate(userID string) error
    Login(username, password string) (*user.User, error)
    CheckBalance(userID string) (float64, error)
}
```

- internal
  - delivery
  - entity
  - infra
  - repository
  - service / userservice
    - userservice.go

userservice.go

internal > service > userservice > userservice.go > ...

```go
15
16  type Service struct {
17      userRepository repository.UserRepository
18  }
19
20  func NewService(userRepository repository.UserRepository) *Service {
21      return &Service{userRepository: userRepository}
22  }
23
24 > func (s *Service) Register(user *user.User) error {…
32  }
33
34  func (s *Service) Activate(userID string) error {
35      u, err := s.userRepository.FindByID(userID)
36      if err != nil {
37          return err
38      }
39      u.IsActive = true
40
41      return s.userRepository.Update(u)
42  }
43 > func (s *Service) Login(username, password string) (*user.User, error) {…
45  }
46
47 > func (s *Service) CheckBalance(userID string) (float64, error) {…
53  }
54
55 > func validateUser(u *user.User) error {…
65  }
```

User Repository (Repository Layer)

EXPLORER

MYAPP

- .idea
- cmd
- internal
  - delivery
  - entity
  - infra
  - repository
    - mysql
      - user.go
    - user.go
  - service

user.go

internal > repository > user.go > ...

```go
package repository

import "myapp/internal/entity/user"

type UserRepository interface {
    FindByID(id string) (*user.User, error)
    Save(u *user.User) error
    Delete(id string) error
    Update(u *user.User) error
    Get(username string) (*user.User, error)
}
```

EXPLORER

MYAPP

- .idea
- cmd
- internal
  - delivery
  - entity
  - infra
  - repository
    - mysql
      - user.go
    - user.go
  - service
- go.mod
- go.sum

OUTLINE

user.go

internal > repository > mysql > user.go > ...

```go
package mysql

import (…
)

type UserRepository struct {
    db *sqlx.DB
}

func NewUserRepository(db *sqlx.DB) (*UserRepository, error) {…
}

func (r *UserRepository) FindByID(id string) (*user.User, error) {
    var u user.User
    query := "SELECT id, username, password, email, is_active, role, balance FROM users WHERE id = ?"
    err := r.db.QueryRow(query, id).Scan(&u.ID, &u.Username, &u.Password, &u.Email, &u.IsActive, &u.Role, &u.Balance)
    if err != nil {
        if err == sql.ErrNoRows {
            return nil, errors.New("user not found")
        }
        return nil, err
    }
    return &u, nil
}

func (r *UserRepository) Save(u *user.User) error {…
}

func (r *UserRepository) Delete(id string) error {…
}

func (r *UserRepository) Update(u *user.User) error {…
}

func (r *UserRepository) Get(username string) (*user.User, error) {…
}
```

REST Controller(Delivery Layer)

EXPLORER

CLEAN-ARCHI...

- cmd
- internal
  - delivery
    - rest.go M
  - entity
  - infra
  - repository
  - service
- presentation
- go.mod
- go.sum
- README.md

rest.go M

internal > delivery > rest.go > ...

```go
package delivery

import (
    "encoding/json"
    "myapp/internal/entity/user"
    "myapp/internal/service/userservice"
    "net/http"
)

type RESTController struct {
    userService userservice.UserService
}

func NewRESTController(s *userservice.Service) *RESTController {
    return &RESTController{userService: s}
}

func (c *RESTController) Run(addr string) {
    http.HandleFunc("/register", c.Register)
    http.HandleFunc("/activate", c.Activate)
    http.HandleFunc("/login", c.Login)
    http.HandleFunc("/check-balance", c.CheckBalance)

    http.ListenAndServe(addr, nil)
}
```

EXPLORER

MYAPP

- .idea
- cmd
- internal
  - delivery
    - rest.go
  - entity
  - infra
  - repository
  - service
- go.mod
- go.sum

rest.go

internal > delivery > rest.go > ...

```go
func (c *RESTController) Register(w http.ResponseWriter, r *http.Request) {…
}

func (c *RESTController) Activate(w http.ResponseWriter, r *http.Request) {…
}

func (c *RESTController) Login(w http.ResponseWriter, r *http.Request) {
    username := r.URL.Query().Get("username")
    password := r.URL.Query().Get("password")
    u, err := c.userService.Login(username, password)
    if err != nil {
        http.Error(w, err.Error(), http.StatusUnauthorized)
        return
    }

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(u)
}

func (c *RESTController) CheckBalance(w http.ResponseWriter, r *http.Request) {
    userID := r.URL.Query().Get("userID")
    balance, err := c.userService.CheckBalance(userID)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }

    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(map[string]float64{"balance": balance})
}
```

MySQL Database (Infrastructure)

EXPLORER

MYAPP

- .idea
- cmd
- internal
  - delivery
  - entity
  - infra / database
    - mysql.go
  - repository
  - service
- go.mod
- go.sum

mysql.go

internal > infra > database > mysql.go > ...

```go
package database

import (
    _ "github.com/go-sql-driver/mysql"
    "github.com/jmoiron/sqlx"
    "log"
)

type ConnInfo struct {
    Host   string
    Port   string
    User   string
    Pass   string
    DBName string
}

func NewMySQLConnection(c *ConnInfo) (*sqlx.DB, error) {…
}

func CloseMySQLConnection(client *sqlx.DB) {…
}
```

App Entrypoint

EXPLORER

MYAPP

- .idea
- cmd / app
  - app.go
- internal
- go.mod
- go.sum

app.go

cmd > app > app.go > ...

```go
package main

import (
    "log"
    "myapp/internal/delivery"
    "myapp/internal/infra/database"
    "myapp/internal/repository/mysql"
    "myapp/internal/service/userservice"
)

func main() {
    // Hardcoded MySQL connection info because this code is only for learning purposes
    appMySQLInfo := &database.ConnInfo{ …
    }

    appDbConn, nErr := database.NewMySQLConnection(appMySQLInfo)
    if nErr != nil {
        log.Fatalln(nErr)
    }
    defer database.CloseMySQLConnection(appDbConn)

    userRepo, err := mysql.NewUserRepository(appDbConn)

    if err != nil {
        log.Fatalln(err)
    }

    userService := userservice.NewService(userRepo)

    apiController := delivery.NewRESTController(userService)

    // run api server
    apiController.Run(":8080")
}
```

Final thoughts

1. Code is communication!
2. Helpful to practice reading code
3. Important to take time to learn
4. Break up complicated code into manageable chunks
5. Writing clean code is an iterative process
6. No single way to write clean code

- **Clean Architecture: A Craftsman's Guide to Software Structure and Design**

  Robert C. Martin

- https://threedots.tech/post/introducing-clean-architecture/
- https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- https://www.slideshare.net/DmytroTurskyi/the-clean-architecturepptx
- https://www.slideshare.net/slideshow/clean-architecture-148074952/148074952
- https://gocasts.ir