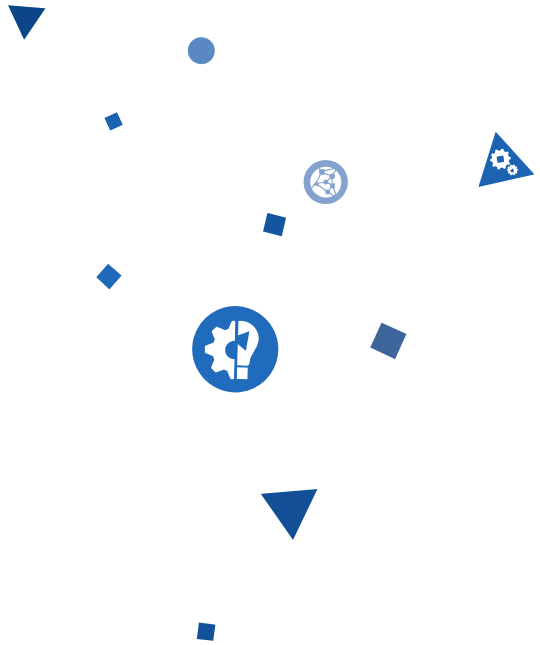


SOLID & ANDROID

SOLID in Android apps development

TRAINING ROADMAP: OVERVIEW

- SOLID
- Clean Architecture
- Example

This training covers major aspects of SOLID. The goal of this training is to give explanation what is SOLID, how it code should be sliced. Also course students will look into the SOLID usage with example.

SECTION 1:
SOLID

REQUIREMENTS TO THE CODE: RULES

- Simplicity
  - Ease of understanding.
- Extensibility
  - Ease of extending functionality.
- Flexibility
  - Ease of adapting to the operating environment.
- Testability
  - Ease of testing from units of code to the entire systems.

CODE CHALLENGES: SIZE & COMPLEXITY

- Problems
  - Most real world problems requiring automation are complex
- Requirements
  - Increasing demand for comprehensive automation
- Solutions
  - Resulting solutions are large and sophisticated

MODULARISATION

- Decomposing large systems into smaller and simpler parts.
  - Functions and procedures.
  - Classes and modules.
- Building the parts.
- Assembling the parts to complete the system.
- To achieve desirable code qualities parts should be interchangeable.

KEY TO INTERCHANGEABILITY

- Coupling
  - Interdependence between different software modules.
  - Low coupling is preferred as it allows flexibility and extensibility.
  - Low coupling via simple and stable interfaces.
- Cohesion
  - Degree to which elements of a module functionally belong together.
  - Higher cohesion reduces complexity of components.
- Higher cohesion and low coupling enables interchangeability

SOLID

- Single Responsibility
  - A class should have a single responsibility.
- Open/Close
  - Open for extension but closed for modification.
- Liskov Substitution
  - Substitution of objects by instances of sub-classes.
- Interface Segregation
  - Many client specific interfaces instead of a big one.
- Dependency Inversion
  - Depend on abstractions instead of implementations.

SIMPLIFICATION WITH SOLID

- Types of coupling
  - Content coupling, Common coupling, Stamp coupling, Control coupling, Data coupling.
- Types of cohesion
  - Co-incidental cohesion, Logical cohesion, Temporal cohesion, Procedural cohesion, Communicational cohesion, Sequential cohesion, Functional cohesion.
- SOLID
  - Five principles of object oriented programming.
  - Aims to deliver extensible and maintainable software

SINGLE RESPONSIBILITY

- A class should have one and only one reason to change.
  - A class should have only one responsibility.
- Promotes high cohesion and low coupling.
  - High cohesion because of focused classes.
  - Low coupling because of dependency on other cohesive classes.
  - The same goes for methods for non-OOP languages

SINGLE RESPONSIBILITY - EXAMPLE

OPEN/CLOSE

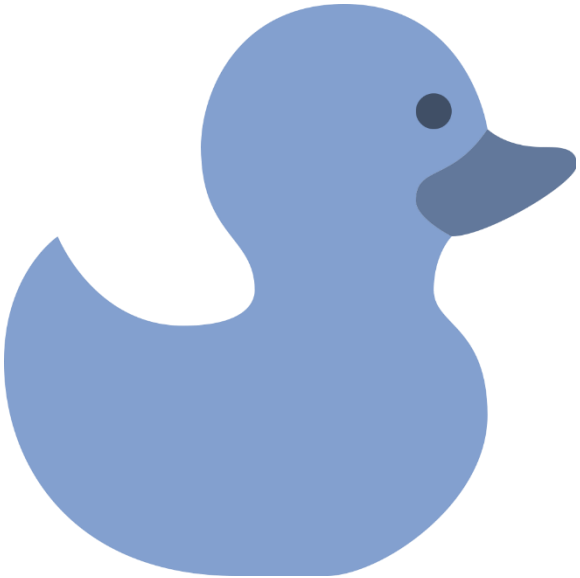
- Software entities (Classes, methods, modules etc.) should be open for extension but closed for modification.
  - Changing functionality does not require modifying existing code.
- Preferred approaches
  - Inheritance
  - Composition of abstractions (plugins)
  - Calling existing methods instead of changing the code (for non-OOP languages)

OPEN/CLOSE - EXAMPLE

LISKOV SUBSTITUTION

- Named after MIT professor Barbara Liskov.
- Functions that use references to base classes must be able to use objects of the derived class without knowing it.
  - Derived classes must be substitutable for base class.
- Caution
  - Should not alter the behavior of the application.
  - Inheritance hierarchies should not violate domain concepts.
    - E.g., a **square** is a **rectangle**, but from OOP's point of view both are **shapes**.

LISKOV SUBSTITUTION - EXAMPLE



INTERFACE SEGREGATION

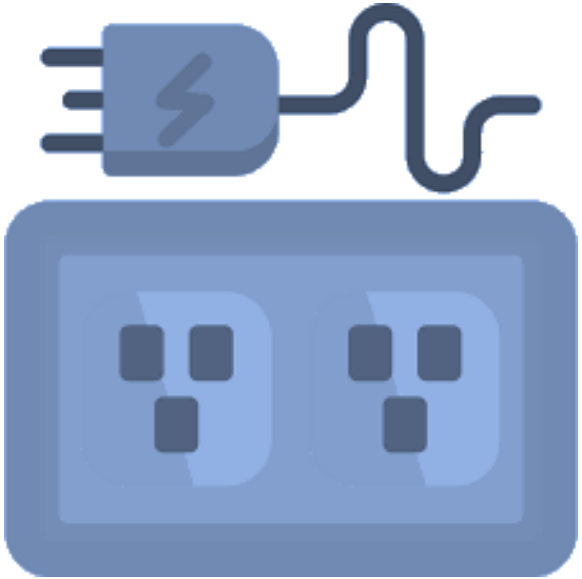
- Multiple fine grained client or domain specific interfaces are better than few coarse grained interfaces.
- Consequences
  - Classes only implement interfaces that are requirement specific.
  - Client classes are exposed only to the functionality that they need.

INTERFACE SEGREGATION - EXAMPLE

DEPENDENCY INVERSION

- High-level modules should not depend on low- level modules.
  - Both should depend on abstractions.
- Abstractions should not depend on details.
  - Details should depend on abstractions.
- Dependencies are a risk
  - Details bind dependents to concrete implementations. This limits flexibility and extensibility.
  - Abstractions provides independence to dependents. Dependents are able to select most suitable implementations

DEPENDENCY INVERSION - EXAMPLE



SOLID VS STUPID

- Single Responsibility
- Open/Close
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

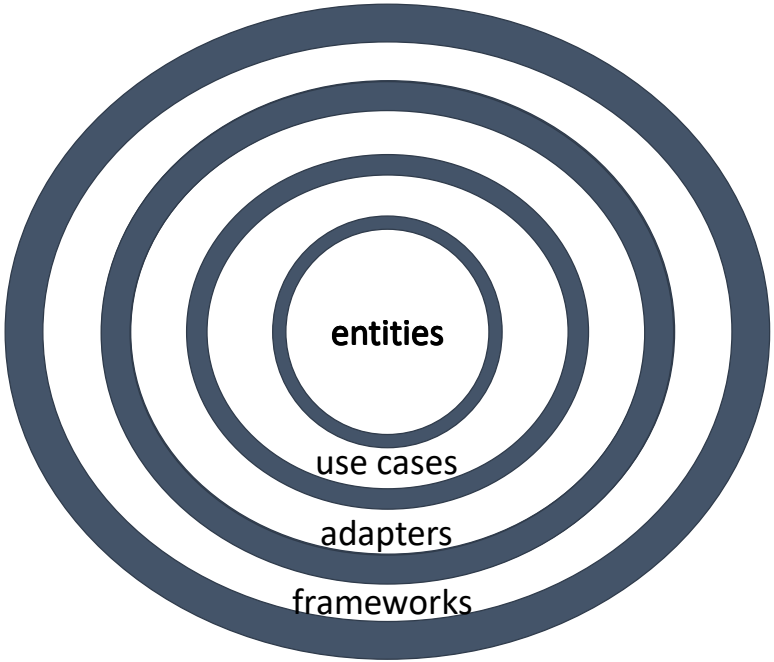
- Singletones
- Tight coupling
- Untestability
- Premature optimization
- Indescriptive naming
- Duplication

SECTION 2:
CLEAN ARCHITECTURE

CLEAN ARCHITECTURE: PRINCIPLES

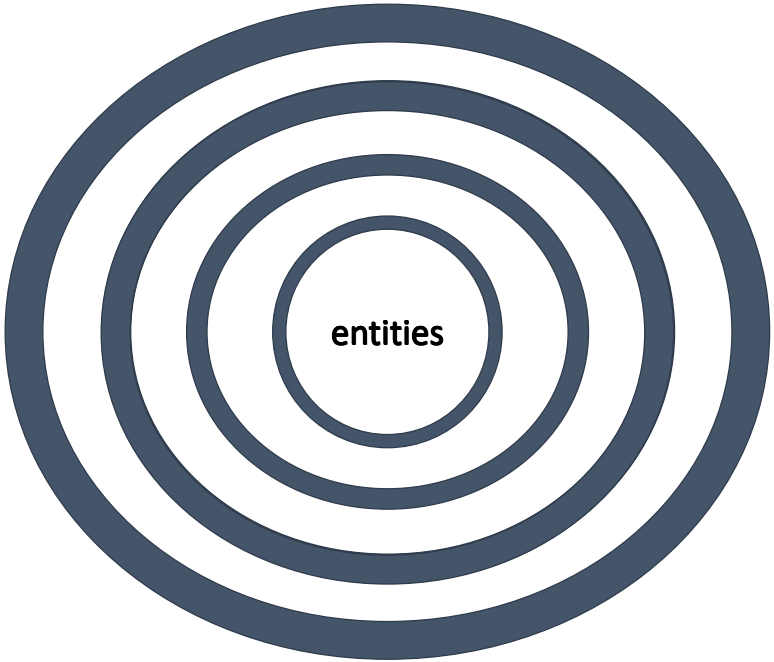
- Independent of frameworks
- Testable
- Independent of UI
- Independent of database
- Independent of external ..{placeholder}..

CLEAN ARCHITECTURE: PARTS



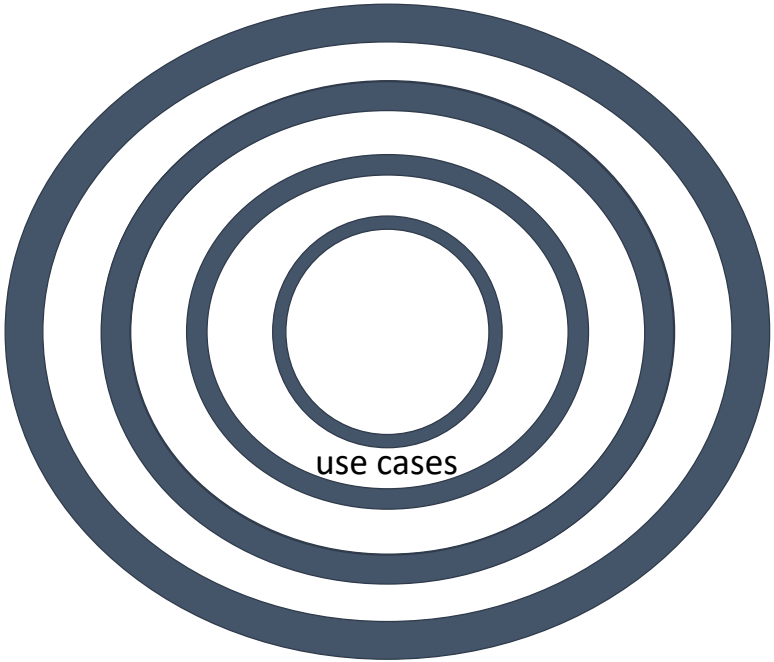
- Dependency rule
  - entities are alone
  - use cases know entities
  - adapters know use cases
  - frameworks know adapters

CLEAN ARCHITECTURE: ENTITIES



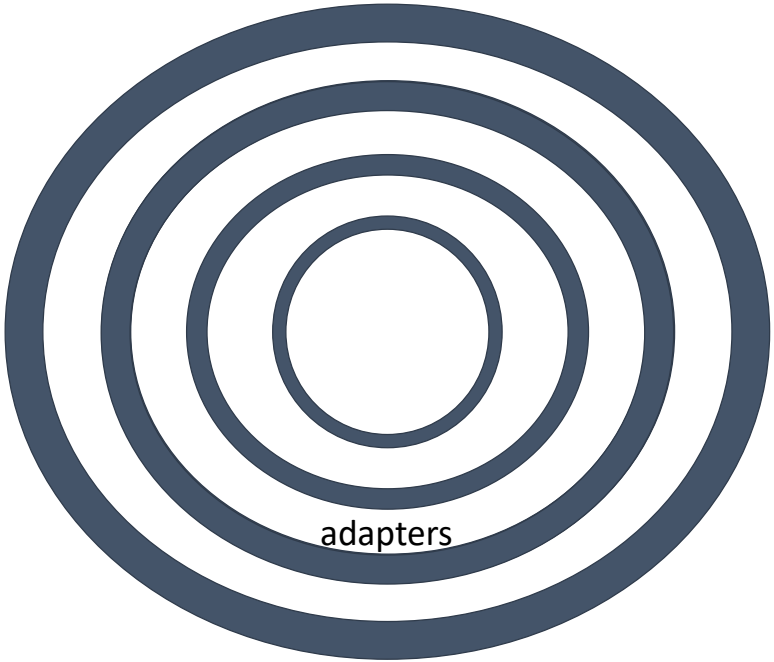
- Entities:
  - enterprise wide business rules
  - could be used by different applications in the enterprise
  - these are the less likely to change when some external changes

CLEAN ARCHITECTURE: USE CASE



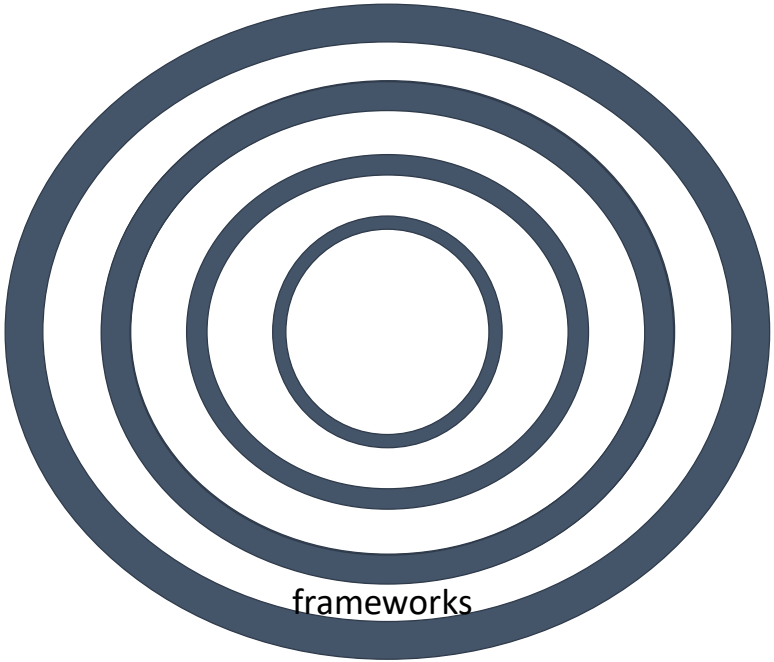
- Use case:
  - application specific business rules
  - they use entities to achieve a goal
  - it will change if the operation of the application changes
  - totally isolated from UI, databases & etc

CLEAN ARCHITECTURE: ADAPTERS



- Adapters:
  - translates data from use case and entities to an external agency
  - converts requests from the outer layers to the inner ones

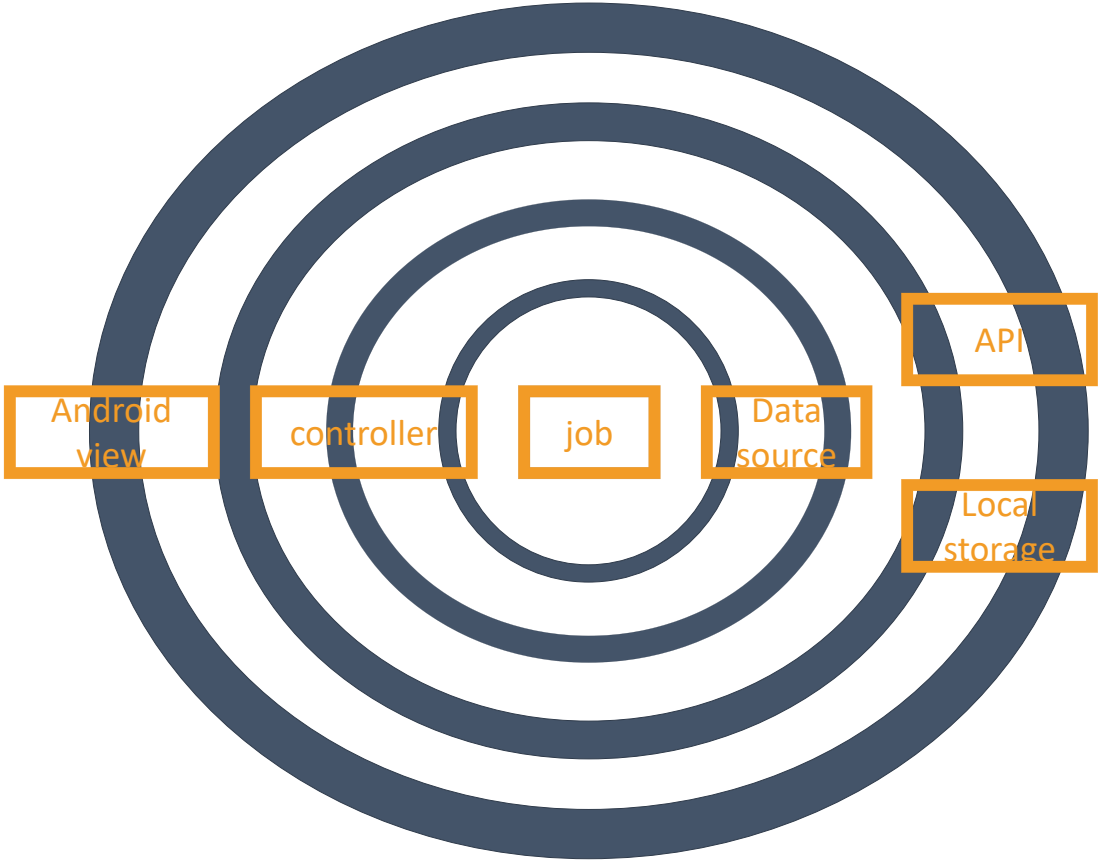
CLEAN ARCHITECTURE: FRAMEWORKS



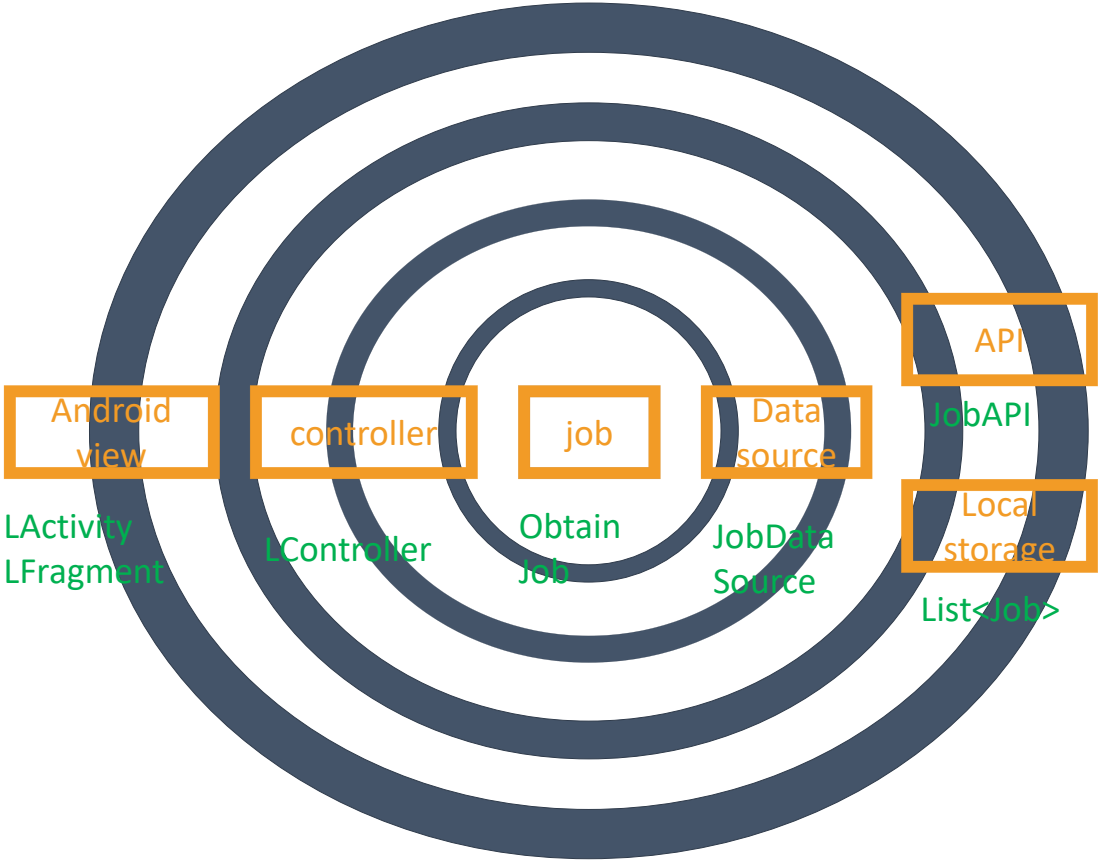
- Frameworks:
  - android screens, web, database, a distributed cache... are details
  - keeping them outside where they can do little harm

SECTION 3:
EXAMPLE

CLEAN ARCHITECTURE: FRAMEWORKS



CLEAN ARCHITECTURE: FRAMEWORKS



CLEAN ARCHITECTURE: FRAMEWORKS