Karlsruhe Institute of Technology

Software Engineering II

Prof. Dr. Ralf H. Reussner

Topic 4
Clean Architecture

DSIS – DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
KASTEL – INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY

dsis.kastel.kit.edu

Overview on Today's Lecture

Content

- Motivation: Maintainability
- “Clean Architecture” Style

Learning Goals: Students are able to

- Know the underlying principles of clean architecture styles
- Know the “circles” of the clean architecture style and how they apply in software

Tale of Two Values

- What is more important: **Behaviour** or **Maintainability**?
  - **Behaviour**: The software fulfills the functional requirements to the stakeholder's satisfaction
  - **Maintainability**: Changes are simple and easy to make.

- Which of these two provides the greater value?
  - Is it more important for the software system to work,
  - or is it more important for the software system to be easy to change?

[Martin 2017]

Tale of Two Values (cont'd)

- What is more important: **Behaviour** or **Maintainability**?
  - **Behaviour**: The software fulfills the functional requirements to the stakeholder's satisfaction
  - **Maintainability**: Changes are simple and easy to make.
- Which of these two provides the greater value?

- Simple logical tool of examining the extremes
  - A program that works perfectly but is impossible to change
    → won't work when the requirements change
    → developer won't be able to make it work
    → program will become useless.
  - A program that does not work but is easy to change
    → developer can make it work, and keep it working as requirements change
    → program will remain continually useful.

[Martin 2017]

Fight for the Architecture

- Business managers
  - tend to focus on behaviour
  - are not equipped to evaluate the importance of architecture

→ That's what software developers were hired to do.

- Software developers are responsible to focus on maintainability
  - assert the importance of architecture over the urgency of features.

- A continuous struggle between stakeholders
  - As developers are stakeholders, too

→ Clean code and clean architecture
  - Design principles: SOLID
  - Architecture principles: The dependency rule

[Martin 2017]

Motivation

- Layers of Clean Architecture help to separate different concerns from each other.

Architecture remains technology independent and remains better testable.

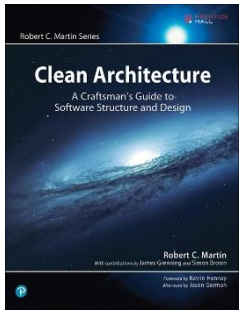
- “The goal of software architecture is to minimize the human resources required to build and maintain the required system.” [Martin 2017]

- “The only way to go fast, is to go well.” [Martin 2017]

Foundations for This Lecture

Clean Architecture

- Core principles of Software Architecture and Design
- Disciplines and practices of professional architects



[Martin 2017]

A warning

- Patterns and principles presented here do not consider other quality attributes than maintainability
- See later lectures for other quality attributes

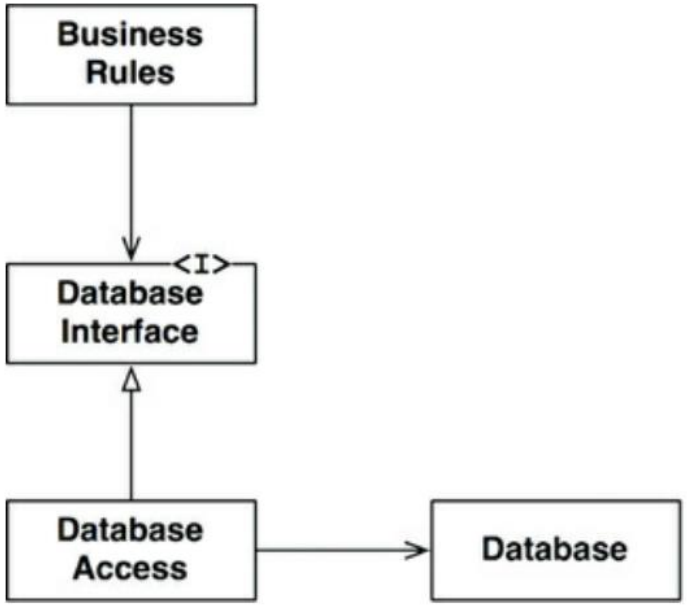
Robert Cecile „Uncle Bob“ Martin:
Programmer since 1970, Founder
and president of Object Mentor Inc.

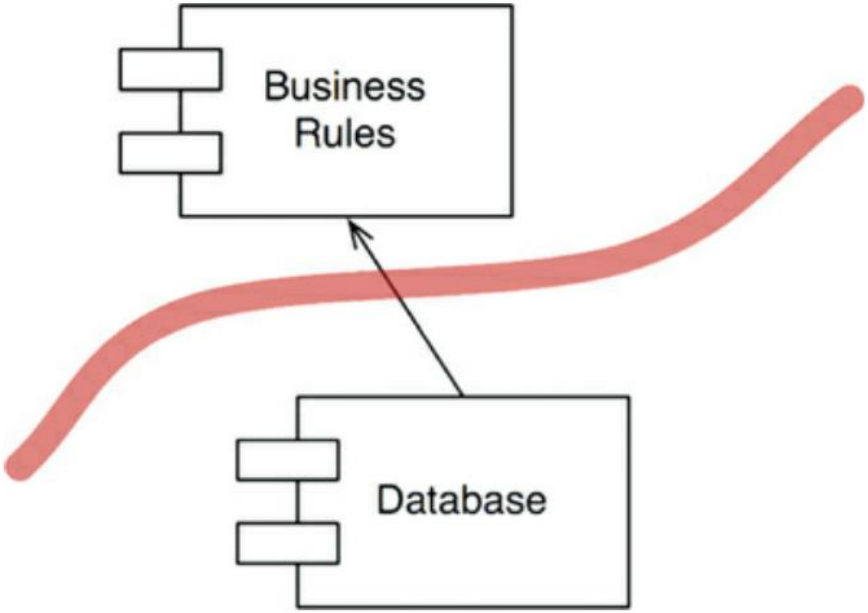
Dependencies in the „Right“ Direction

Principle: Depend in the direction of stability

Put inversely: Any component that we expect to be volatile should not be depended on by a component that is difficult to change.

Example: Business rules should be independent of the way the database is accessed





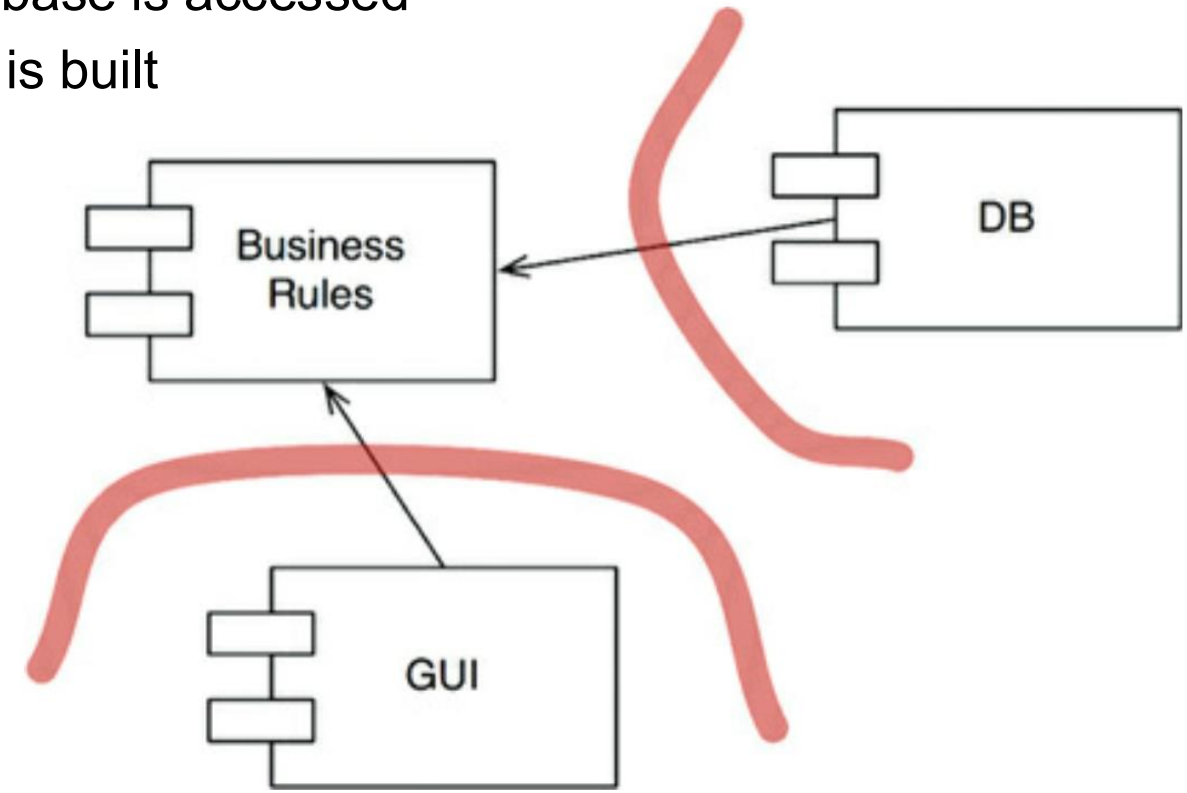
[Martin 2017]

Dependencies in the „Right“ Direction

Principle: Depend in the direction of stability

Example: Business rules should be independent of the way

- the database is accessed
- the GUI is built



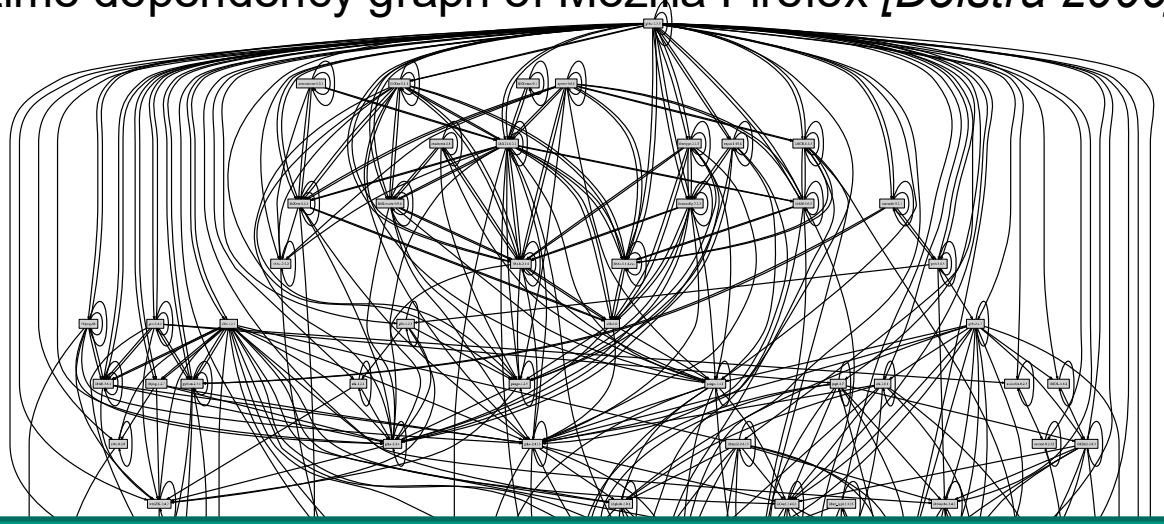
[Martin 2017]

Taming Cyclic Dependencies

Larger systems include multiple interconnected unstable parts

- Hard to enforce *dependence on stability*
- Cyclic *Dependencies* easily occur and hard to spot

Example: runtime dependency graph of Mozilla Firefox *[Dolstra 2006]*



Cyclic Dependency

- Component has a cyclic dependency, iff it has indirect dependency to itself
- Combine all mutual reachable components in a set
  (compare with *Strongly Connected Components (SCC)*)

Taming Cyclic Dependencies

“Although we might serene enough to tolerate cyclic dependencies among a few components within a single package due to carelessness ignorance or special circumstance, we must be steadfast in our resolve to avoid cyclic dependencies among packages.”

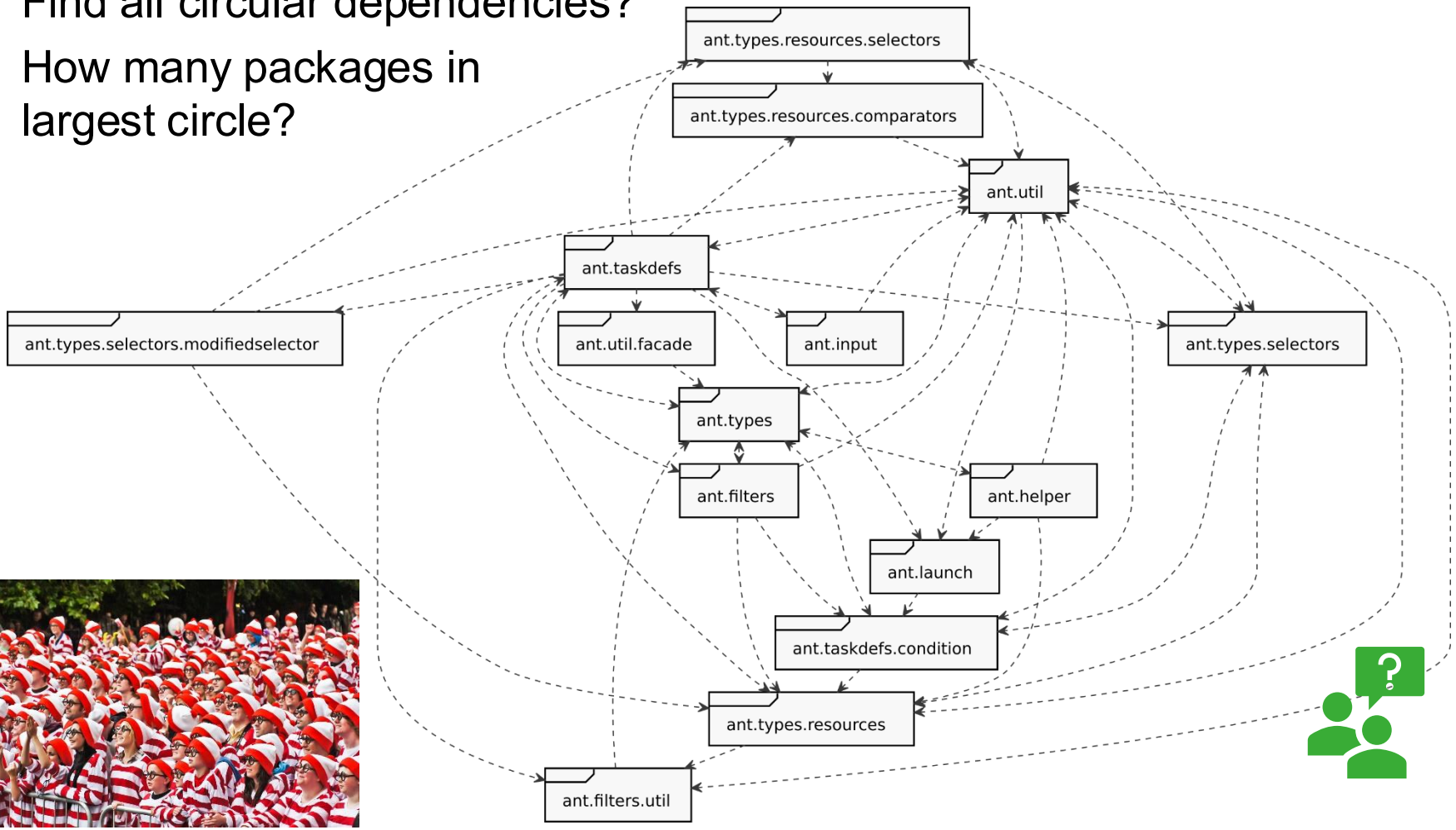
– John S. Lakos [Lakos 1996]

- Changes in any component with a cyclic dependency affects all other components in same cycle
- Drastically reduces maintainability of overall system

Taming Cyclic Dependencies

Example partial package dependencies in Ant *(Version 1.8.4)*

- Find all circular dependencies?
- How many packages in largest circle?

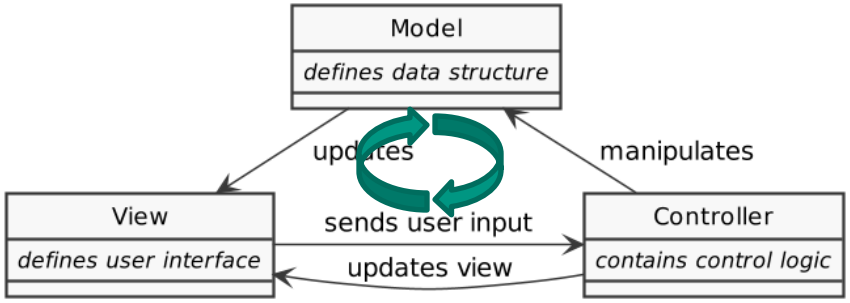


Taming Cyclic Dependencies

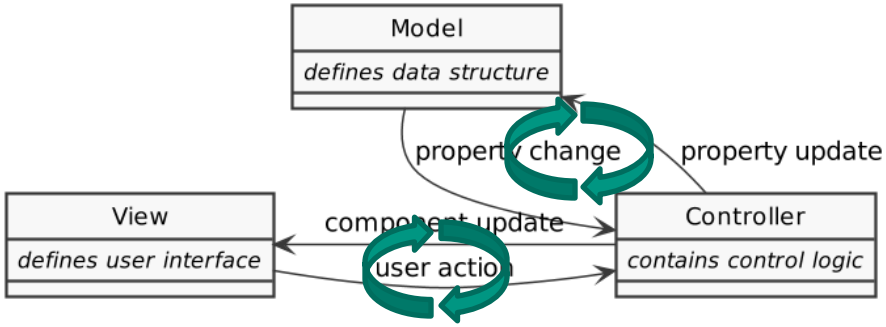
Example MVC or How to (not) Build GUIs

- Recall *Model View Controller (MVC)* design pattern *[Krasner 1988]*

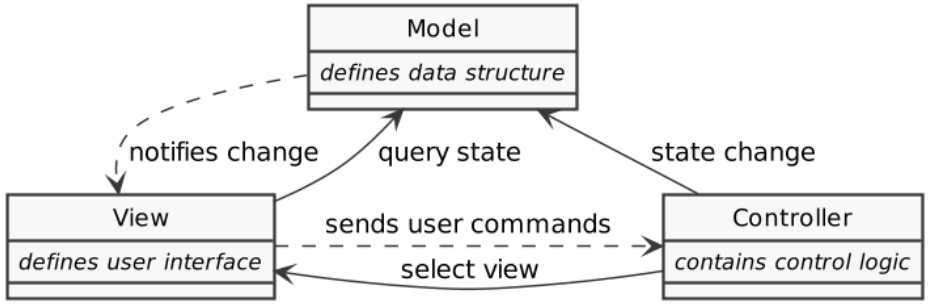
Naïve Variant[1]



Recent Variant[2]



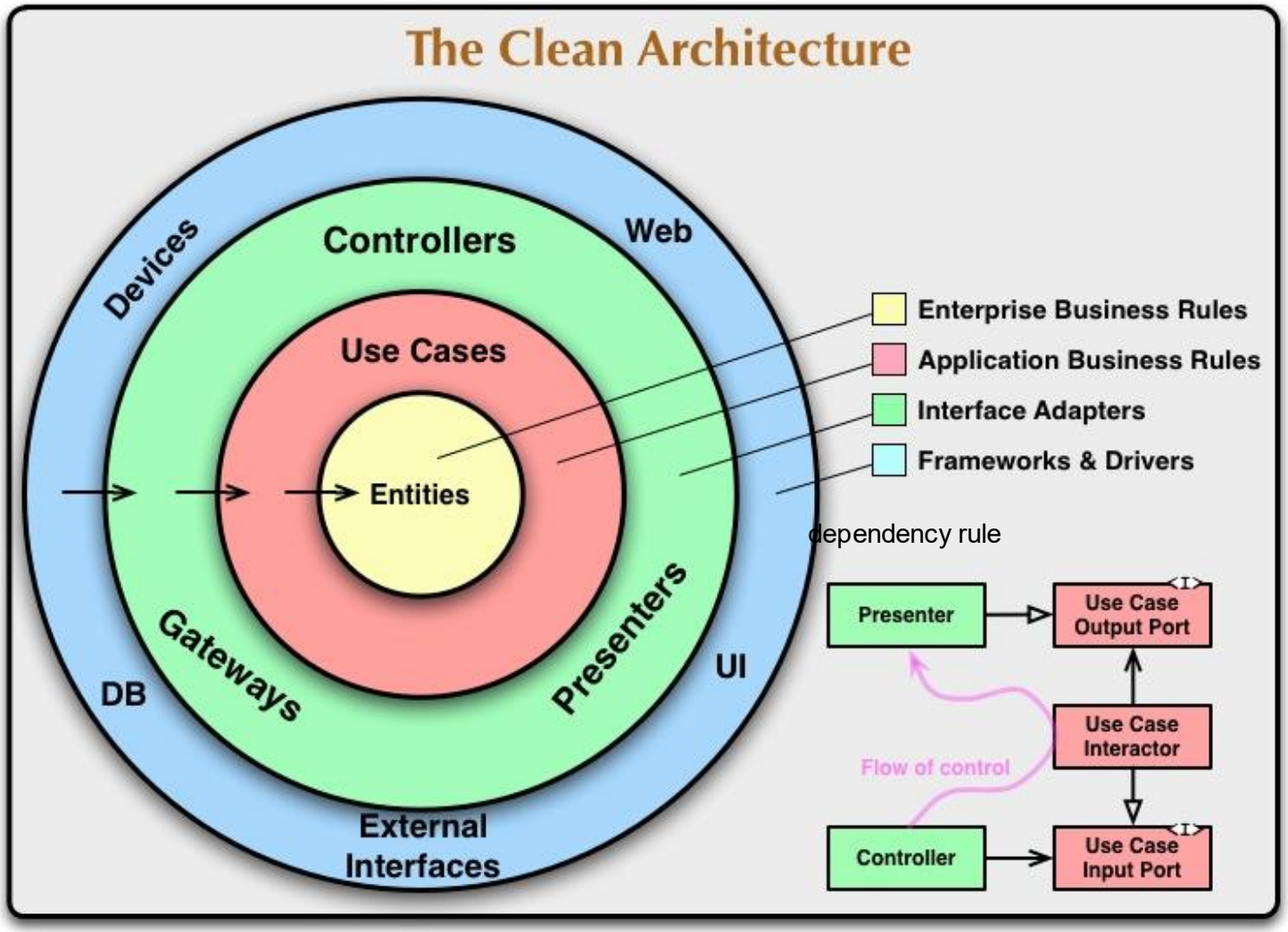
Classic Variant[2]



Dependencies must be resolved with *Dependency Inversion* or *Dependency Injection*

1) https://www.w3school.in/mvc-architecture/
2) https://www.oracle.com/technical-resources/articles/javase/mvc.html

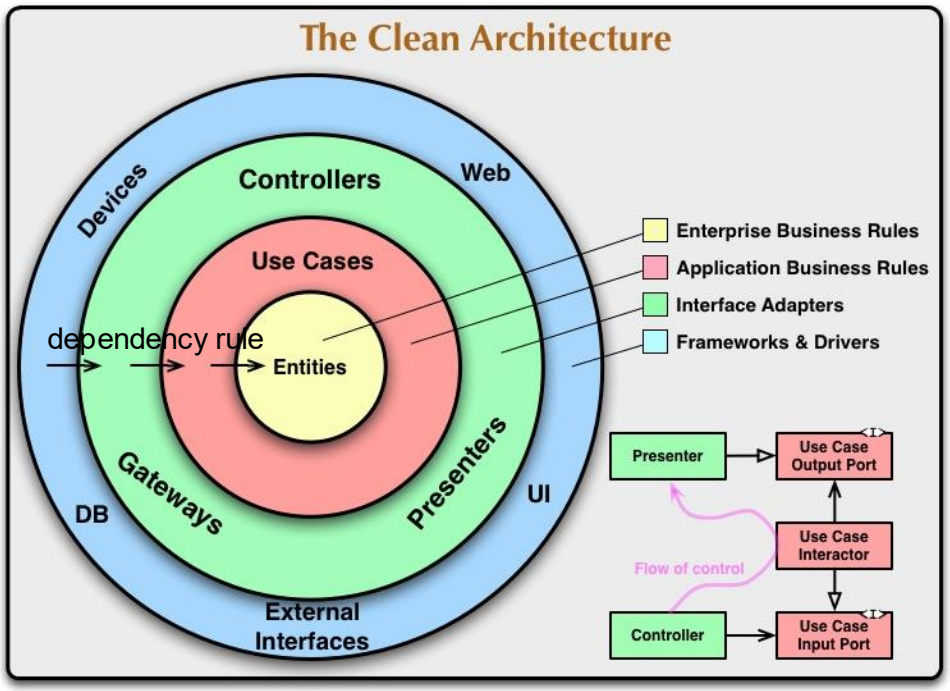
The “Clean Architecture” Style



[Martin 2017]

The “Clean Architecture” Style

- Concentric circles represent different areas of software
  - Further in == higher level the software becomes
  - Outer circles describe mechanisms
  - Inner circles are policies
- “The Dependency Rule”
  - Source code dependencies always point inwards
  - That rule applies even for
    - Functions
    - Classes
    - Variables
    - Data formats



No impacts from outer circles to inner circles! [Martin 2017]

Claimed Benefits of “Clean Architecture”

- Independence of Frameworks
  - No dependencies on feature laden software
  - Use advantages of frameworks with less of their disadvantages
- Testable systems
  - Business rules testable without external elements (i.e. DB, UI)
- Independence of UI
  - Change UI without changing rest of system
- Independence of Database
  - Business rules are not bound to any DB vendor
- Independence of external agency
  - Business rules do not know (and are independent of) the outside world

[Martin 2017]

The “Clean Architecture”: Entities

- Encapsulate business rules
  - Object with methods
  - Data structure set with functions
- Can be used as business object for applications
- Entity comparably stable against external changes

No operational changes of individual applications should affect the entity layer

[Martin 2017]

The “Clean Architecture”: Use Cases

- Contains application-specific business rules
- Implements use cases of the system
  - Defines data flow *from* and *to* entities
  - Defines *collaborations* between multiple entities
- Changes of DB, UI or other externalities are *not* expected to affect UCs
- Layer is expected to be affected by changes to the operation of an application
  - Use cases might change
  - Therefore code of the use case layer might change

[Martin 2017]

The “Clean Architecture”: Interface Adapters

Main concern: data exchange between layers

- Set of adapters
- Convert data
  - From format most convenient for use cases and entities
  - To format most convenient for externalities like DB and web

Example: Contains MVC architecture of a GUI

- Contains views and controllers
- Models are often simple data structures passed from controllers to use cases and back to view

[Martin 2017]

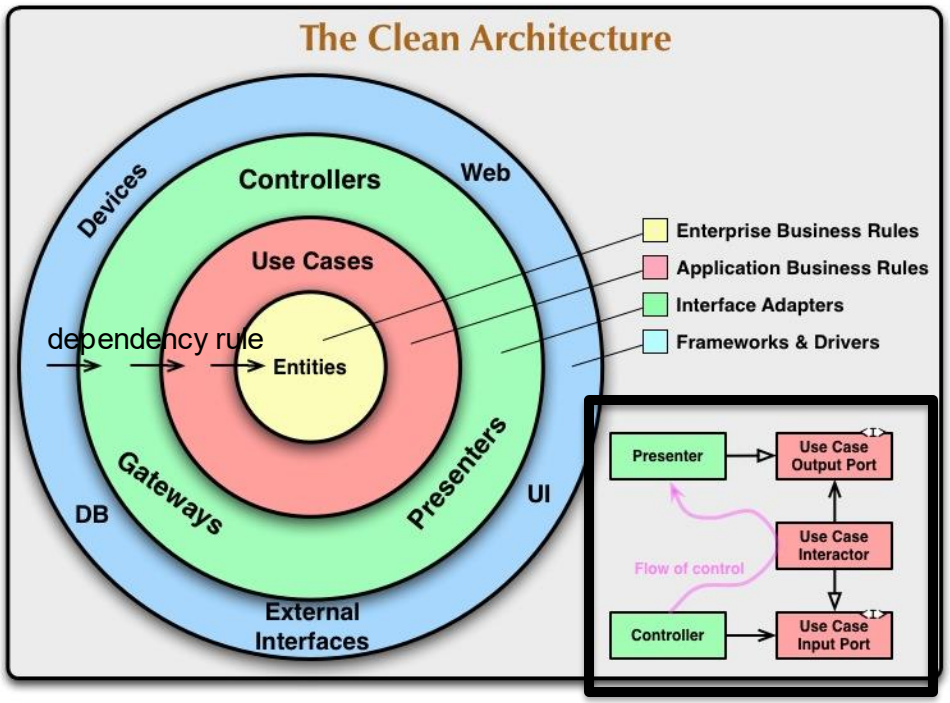
Crossing Boundaries

- Controller and Presenter communicate with use cases
  Controller → Use cases → Executing in presenter
- Source code dependencies point inwards towards use cases

→ Usually resolved with *dependency inversion principle*

Solution in Java:

- Arrange interfaces and inheritance relationships such that code dependencies oppose the control flow where needed

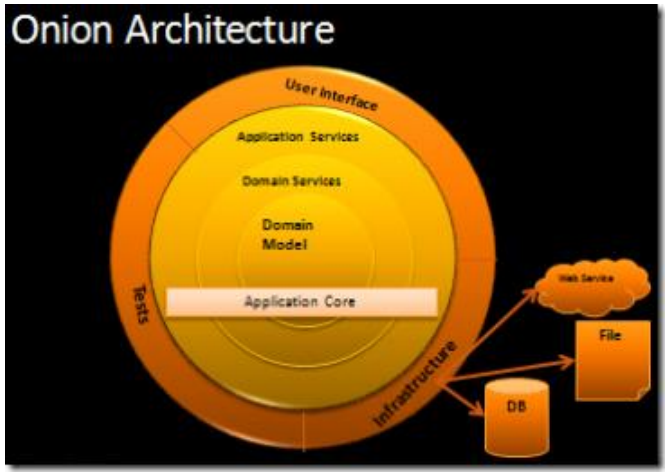


→ Same solution used to across all other boundaries in the system [Martin 2017]

Examples of Clean Architecture Styles

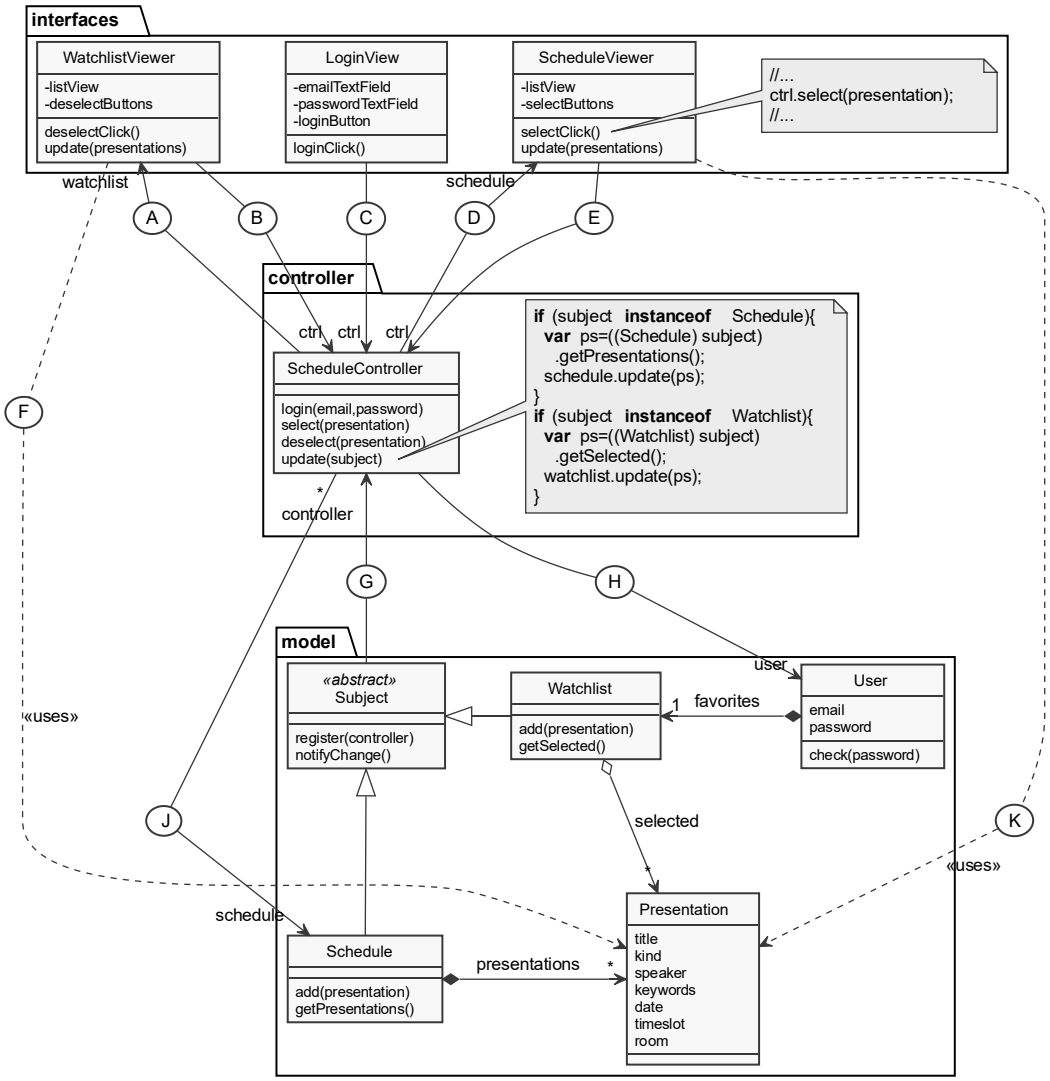
Several Architectural Styles

- Hexagonal Architecture
  - Also known as Ports and Adapters
  - Better testability of core and independent of outside services
  - Services easy to replace
- Onion Architecture
  - Controls coupling, since coupling is the centre
- Boundary/Control/Entity (BCE)
  - Variation of MVC
  - Distribution of responsibility to set of interacting design elements



Common objective: Separation of concerns using layers!

Example Architecture



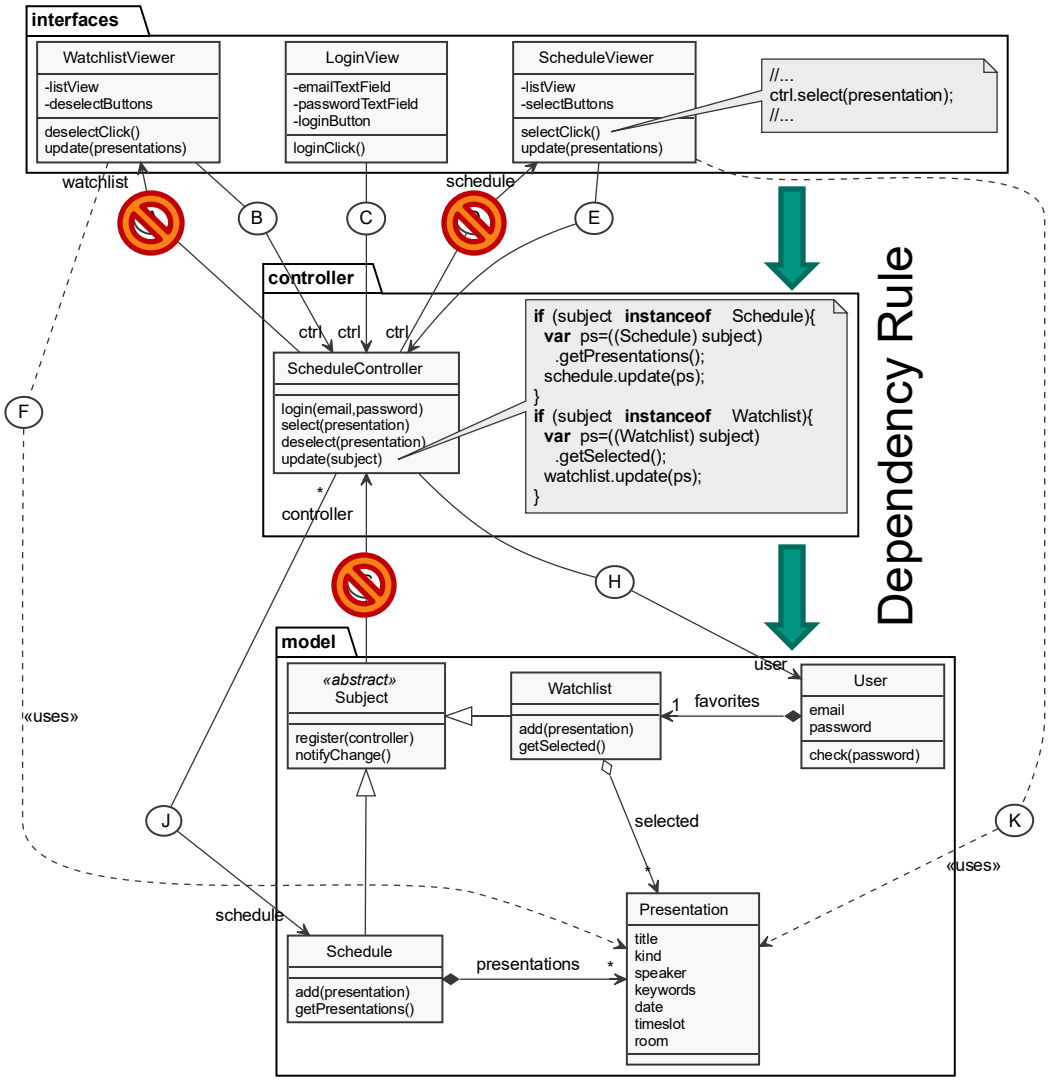
Example:

Conference application with different views

- *Watchlist*
- *Login*
- *Schedule*

Which relations (A–K) the violate *dependency rule*?

Example Architecture



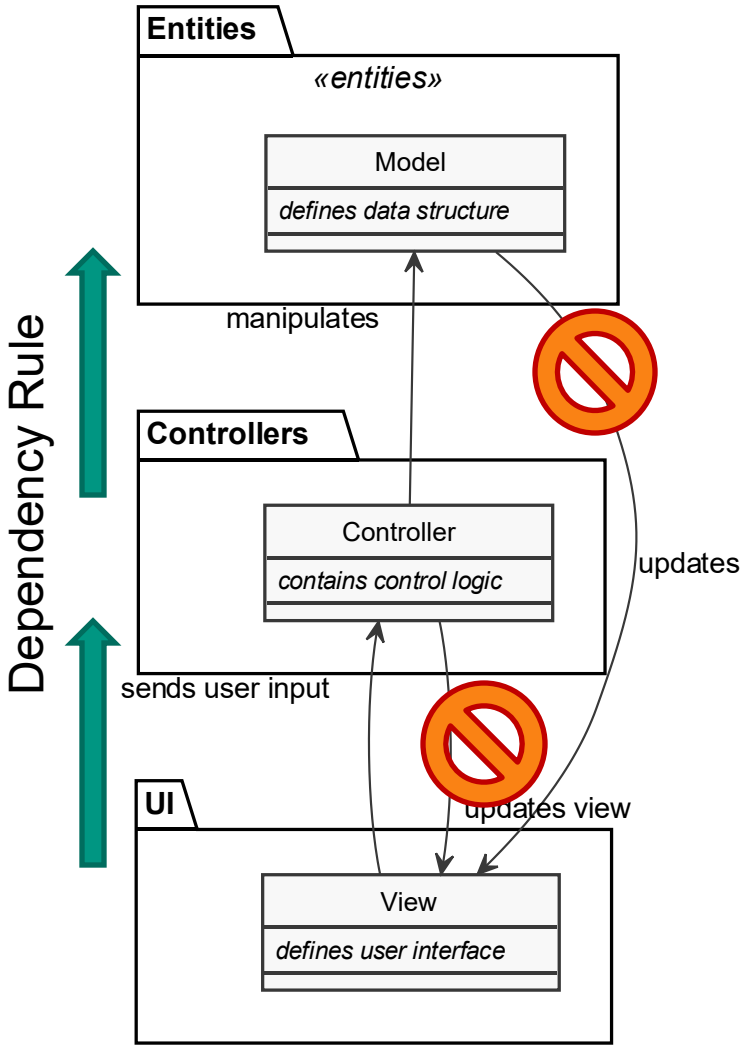
Example:

Conference application with different views

- *Watchlist*
- *Login*
- *Schedule*

Which relations (A–K) the violate *dependency rule*?

Path to Clean Architectures



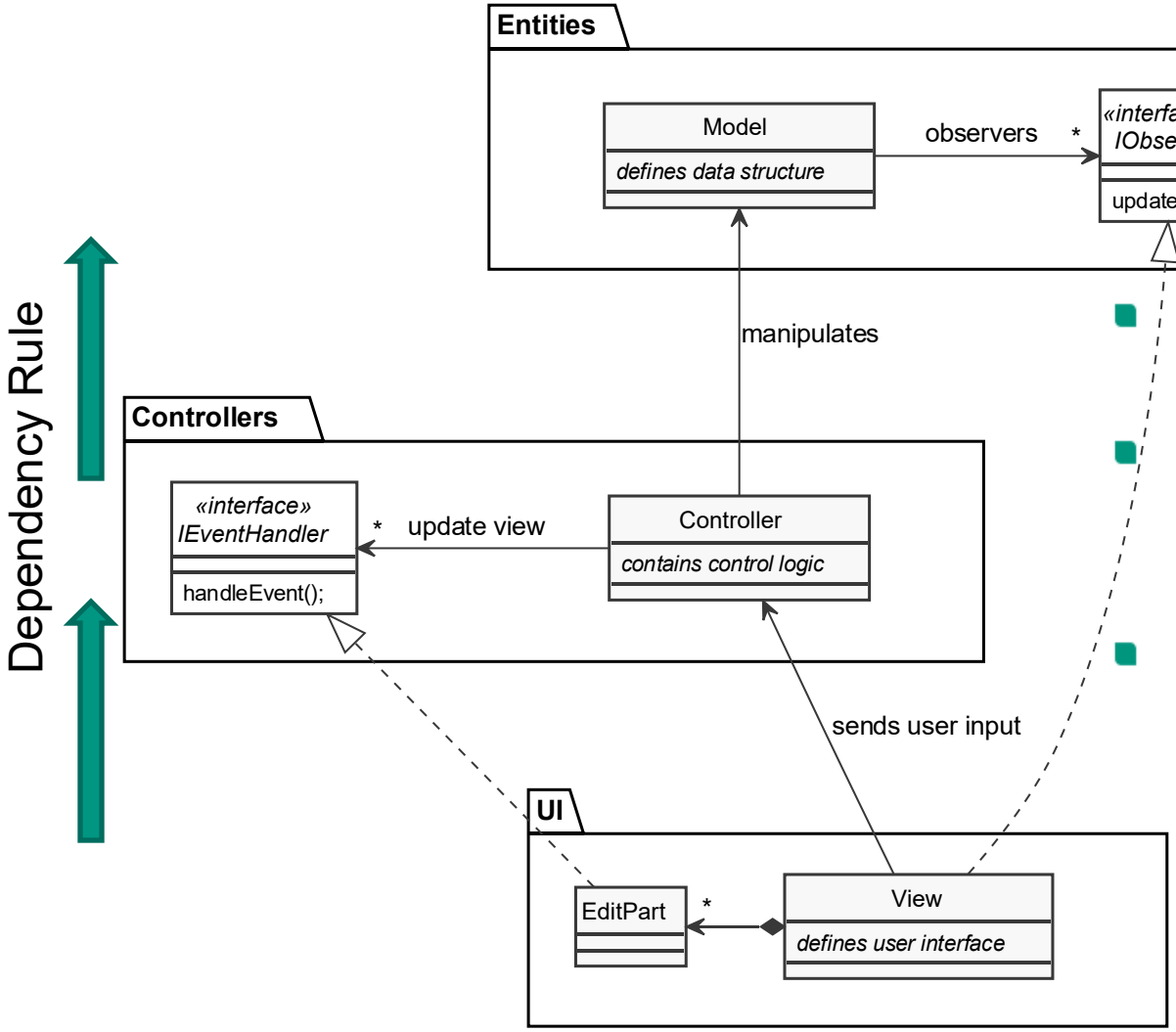
Problem

- dependency in opposite direction of *Dependency Rule*

Solution: Dependency Inversion

- create dependency to abstraction on same layer
- Inherit from abstraction in lower layer (adding dependency to upper layer)

Path to Clean Architectures



- *IObserver* interface as abstraction of *View*
- *IEventHandler* interface to notify *EditParts*
- All dependencies adhere to **dependency rule**

Path to Clean Architectures

Dependency Inversion

- Benefits
  - simplifies **testing** by permitting mock implementations
  - *this is even true, when inner parts are less stable than outer parts. In particular in this case, testability of the inner parts is very helpful. [Reussner]*
  - **decouples** *"boilerplate code"* for wiring
- Drawbacks
  - Requires complex configuration
  - Assumes good (stable, usable) interface abstraction is possible

Others

- *Dependency Injection:* Techniques to resolve abstract dependencies with concrete objects at runtime
- *Inversion of Control (IoC)* container
  e.g. Google Guice, Spring Framework, EJB …

[Martin2002]

Decoupling Modes

- How to distribute a system?
- According to Martin: This decision is one of the options that a good architect leaves open.
- An architecture that
  - maintains the proper isolation of its components, and
  - does not assume the means of communication between those components,
  - will be much easier to transition through the spectrum of threads, processes, and services
  - as the operational needs of the system change over time.

Decouple layers and use cases, then distribution mode can be left open [Martin 2017]

Caveat: This decoupling of distribution and interface design is not always possible (e.g. performance otpimisations of interface to speed up slower remote accesses) [Reussner]

Conclusion

- What we learned –
  - Motivation: Maintainability
  - “Clean Architecture” Style
  - Main principle: Dependency rule

- This lecture does only cover some aspects
  - Martin Fowler‘s and Robert Martin‘s books are worth reading for every serious programmer
  - further design concepts presented in our lecture “Software-Evolution”

References

- [Martin 2017] Robert C. Martin, *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.
- [Lakos 1996] John S. Lakos, *Large-scale C++ software design* Cambridge University Press, p. 217-271, 1996
- [Dolstra 2006] Eelco Dolstra, *The purely functional software deployment model*. PhD Thesis, Utrecht University, 2006.
- [Krasner 1988] Glenn E. Krasner, and Stephen T. Pope. *A description of the model-view-controller user interface paradigm in the smalltalk-80 system*, Journal of object oriented programming 1.3: 26-49, 1988.

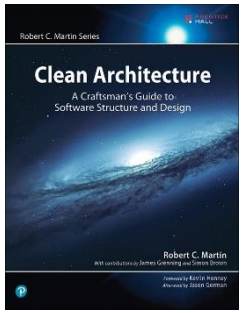


Image Credit

- [Slide 12] William Murphy from Dublin, Ireland - Where's Wally World Record (where you there?) (CC BY-SA 2.0) https://commons.wikimedia.org/wiki/File:Where%E2%80%99s_Wally_World_Record_(5846729480).jpg