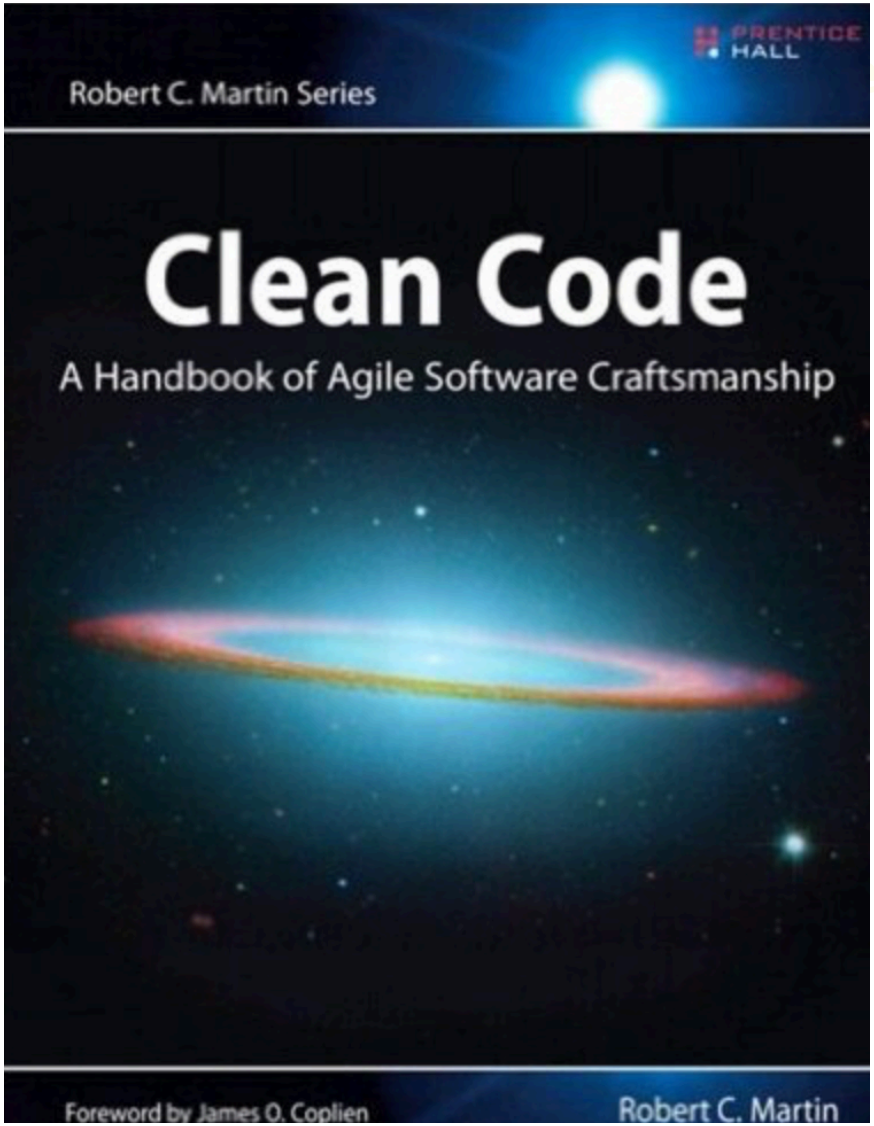
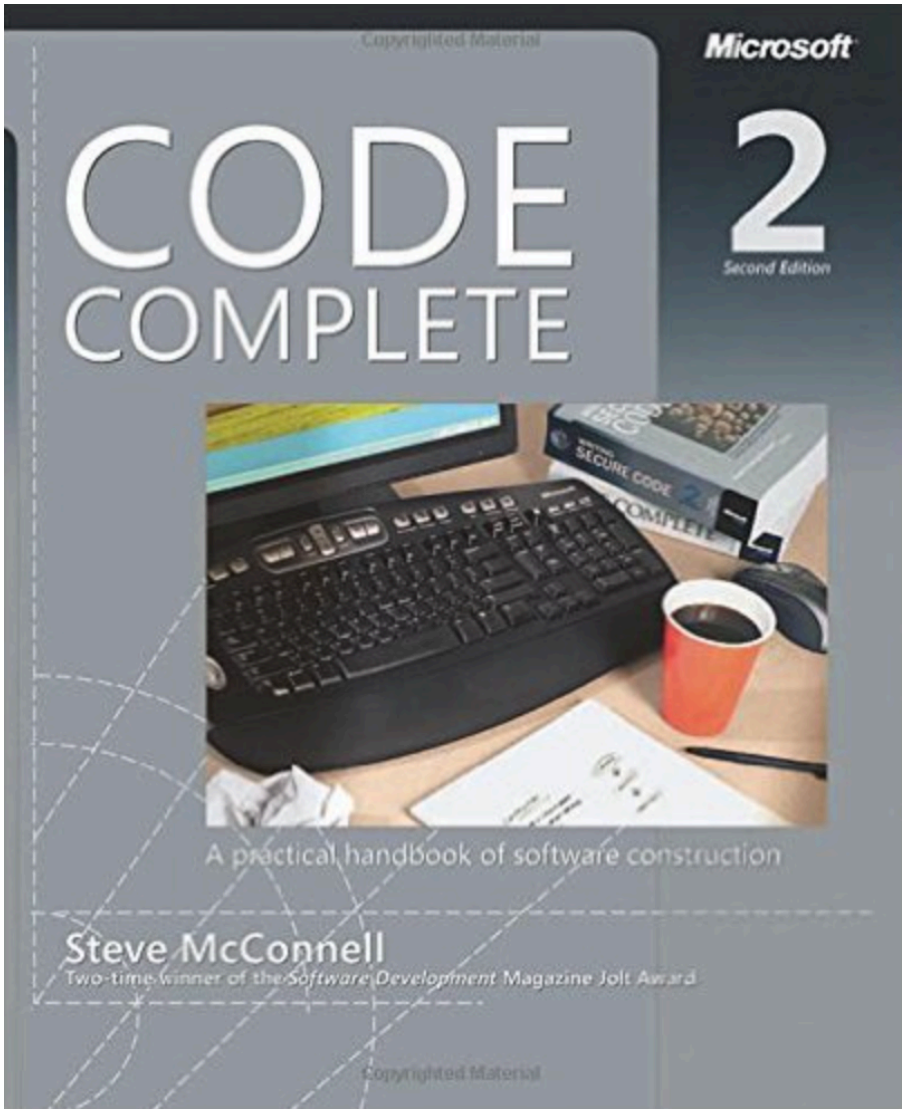
Clean architecture


Volt



tlanego slajdu



Kod który czyta się jak książkę

Use cases

#noFrameworks

Business Driven


JUDGES, 271
I SAMUEL, 115.
II SAMUEL 38
I KINGS, 119
SOLOMON'S
II KINGS, 308
TEMPLE.
NEBUCHADNEZZAR'S
SIX CITIES OF REFUGE.
EHUD KILLS EGLON.
JAEL KILLS SISERA
JEPHTHAH and his DAUGHTER
SAMSON
DAVID AND GOLIATH.
ANCIENT PLOUGH.
YOKES
ARK OF THE COVENANT.
RUTH.
PSALMS of David.
PROVI
ECCLESI
SONG
OF SOLOMON.
CHRONICLES.
II. CHRONICLES.
ELIJAH, THE TISHBITE.
ELISHA, PROPHET IN ISRAEL ABOUT 30 YEARS.
1426 B.C.
NASHON.
SALMON
BOAZ.
OBED.
JESSE.
DAVID.
SOLOMON.
REHOBOAM.
ABIJAH.
ASA.
JEHORAM.
UZZIAH.
JOTHAM.
AHAZ.
HEZEKIAH.
MANASSEH.
AMON.
THE THEOCRACY ENDS and the JEWISH MONARCHY BEGINS
PHOENICIA
PALESTINE
JUDAH
Kingdom of ISRAEL
MESOPOTAMIA and PADAN ARAM
EGYPT
XVIIth DYN. OF THEBES
XXIInd DYN. OF THEBES
XXIst DYN. OF TANIS
XXIIIrd DYN. OF BUBASTIS
XXVth DYN.
EGYPTIAN HISTORY A BLANK
EGYPT FROM THE MONUMENTS.
HEBREW KINGS
THE KHITA OR HITTITES.
BABYLON.
ASSYRIA
BREAK IN BABYLONIAN HISTORY.
FEW ASSYRIAN INSCRIPTIONS KNOWN FOR ABOUT 150 YEARS.
ASIA MINOR.
LYDIA of ASIA MINOR.
MEDIA revolts.
SPARTA.
ATHENS.
MACEDON.
MAGNA GRÆCIA
ROME
CARTHAGE
ALBA LONGA
LATIUM.
CHINA.
11ND DYNASTY, TCHOU.
Historia

Hexagonal aka Ports and Adapters

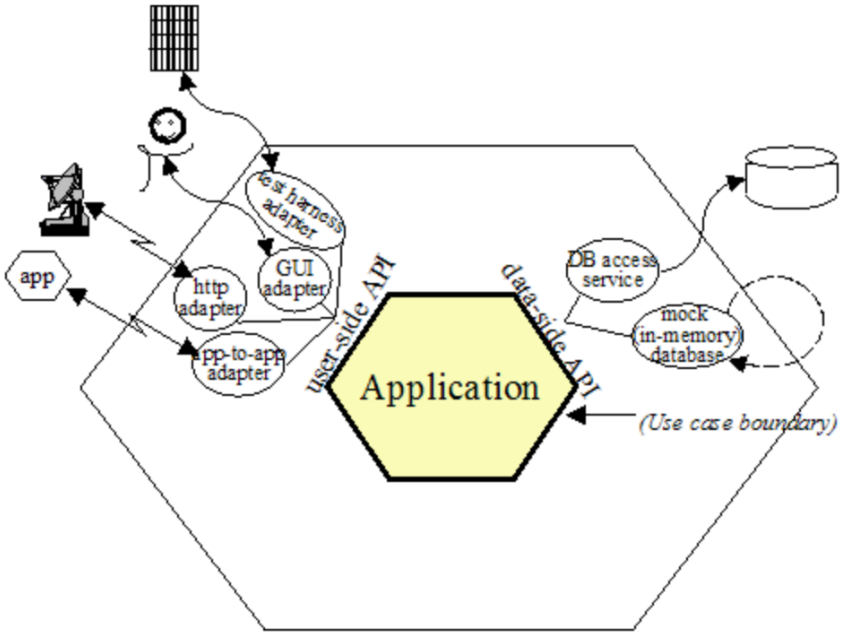
Hexagonal architecture

1/4/2005 |
RA
Content (single or multi vi

Create your application to work without either a UI or a database so you can run automated regression-tests against the application, work when the database becomes unavailable, and link applications together without any user involvement.

(*Japanese translation of this article at* http://blog.tai2.net/hexagonal_architextu
(*Spanish translation of this article at* http://academyfor.us/posts/arquitectura-h

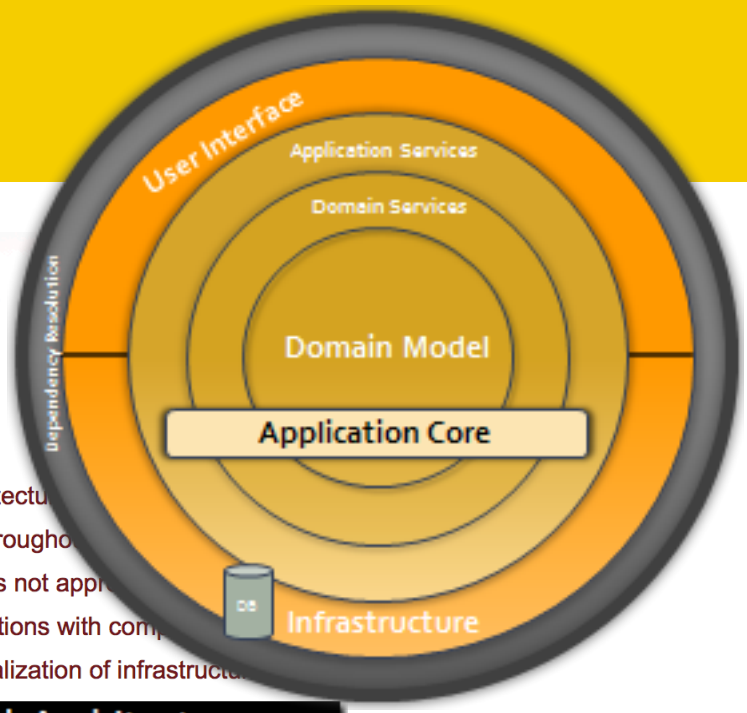
Hexagonal architecture pic 1-to-4 socket.jpg



The Pattern: Ports and Adapters (''Object Structura

Alternative name: ''Ports & Adapters''

Onion Architecture



The Onion Architecture : part 1

29 July 2008

This is part 1. part 2. part 3. part 4. My feed (rss).

I've spoken several times about a specific type of architecture I call "Onion Architectu
to more maintainable applications since it emphasizes separation of concerns throughou
the context for the use of this architecture before proceeding. This architecture is not appr
websites. It is appropriate for long-lived business applications as well as applications with com
emphasizes the use of interfaces for behavior contracts, and it forces the externalization of infrastructu

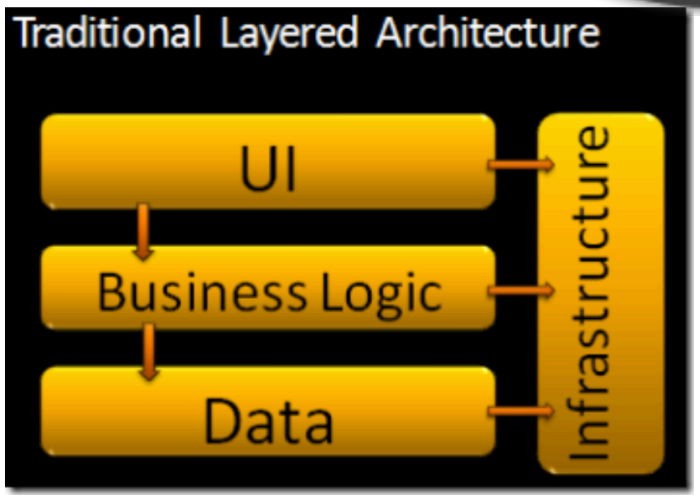


diagram you see here is a representation of traditional layered architecture. This is the basic architecture I see most frequently used. Each subsequent layer depends on the layers beneath it, and then every layer normally will depend on some common infrastructure and utility services. The big drawback to this top-down layered architecture is the coupling that it creates. Each layer is coupled to the layers below it, and each layer is often coupled to various infrastructure concerns. However, without coupling, our systems wouldn't do anything useful, but this architecture creates unnecessary coupling.

Screaming Architecture

Uncle Bob → 30 Sep 2011 + Architecture

Share Tweet G+ Udostępnij

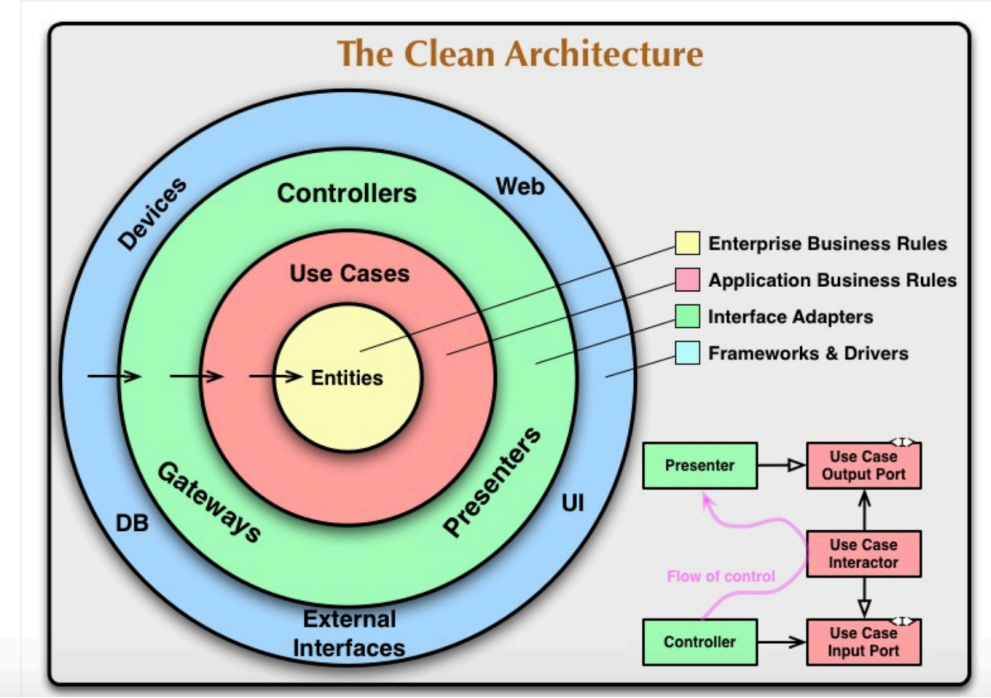
Imagine that you are looking at the blueprints of a buildir prepared by an architect, tells you the plans for the buildi tell you?

If the plans you are looking at are for a single family resid a front entrance, a foyer leading to a living room and perh likely be a kitchen a short distance away, close to the dini area next to the kitchen, and probably a family room clos those plans, there'd be no question that you were looking architecture would *scream*: **house**.

The Clean Architecture

Uncle Bob → 13 Aug 2012 + Architecture Craftsmanship

Share Tweet G+ Udostępnij



Obvious Architecture

Obvious

Obvious is an architecture framework. The goal is to provide architectural structure for a highly testable system that is obvious to understand and where both the front end UI and back end infrastructure are treated as implementation details independent of the app logic itself.

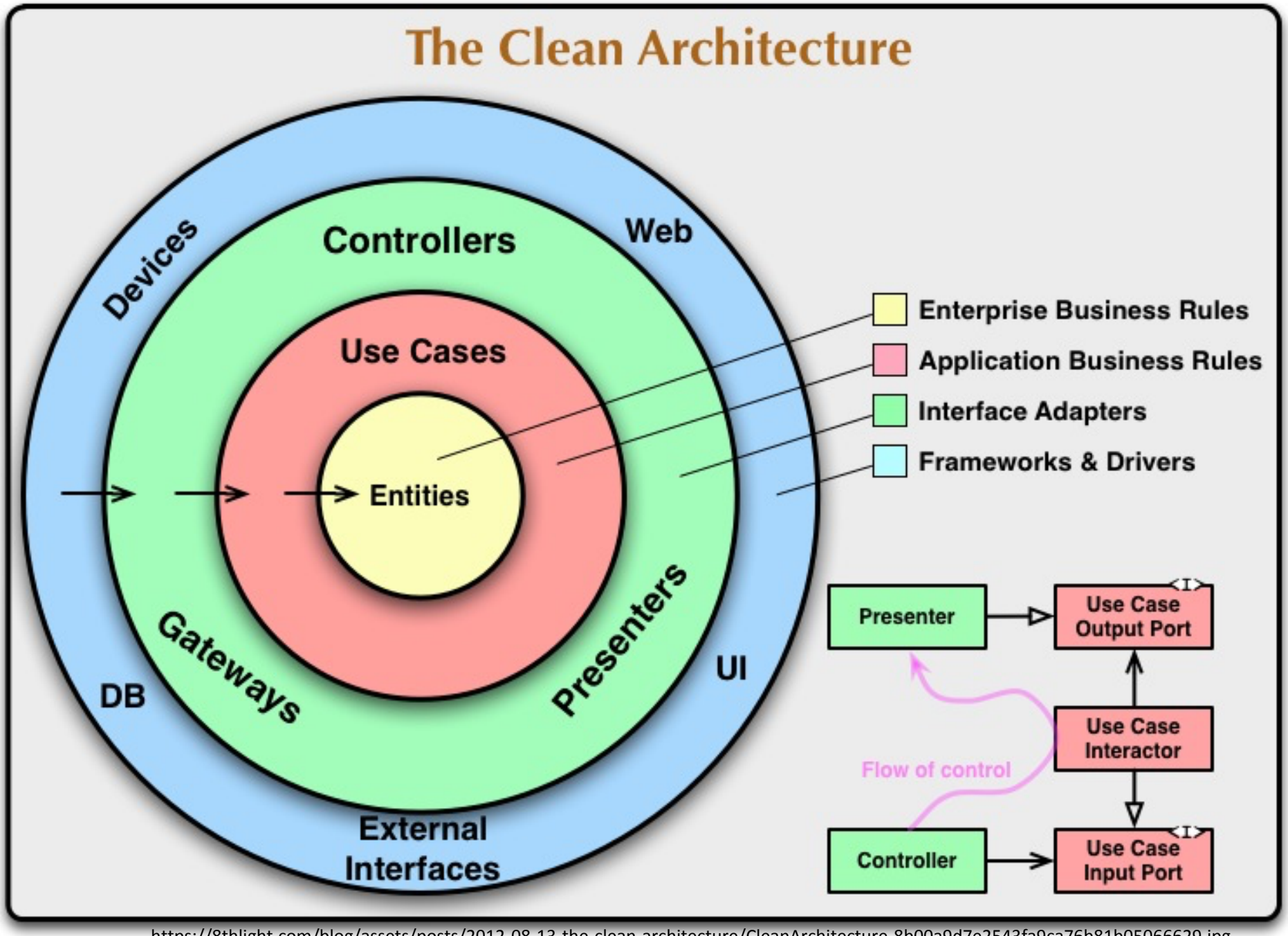
You can get a full explanation of Obvious at http://obvious.retromocha.com

Notice:

This project is no longer under active development is only made available for historical purposes.

Right now Brian is working on a little of this and Shawn is working on a little of that.

Most of our spare energy goes to the Unbranded Pocket Notebook and the Unbranded Pocket Journal.



https://8thlight.com/blog/assets/posts/2012-08-13-the-clean-architecture/CleanArchitecture-8b00a9d7e2543fa9ca76b81b05066629.jpg

Przypadki użycia?

Główny scenariusz powodzenia:

1. System **wyświetla** użytkownikowi stronę logowania z polami loginu i hasła
2. Użytkownik podaje login i hasło
3. System pozytywnie **weryfikuje** podane przez użytkownika dane
4. Użytkownik zostaje **zalogowany**

Przypadki użycia?

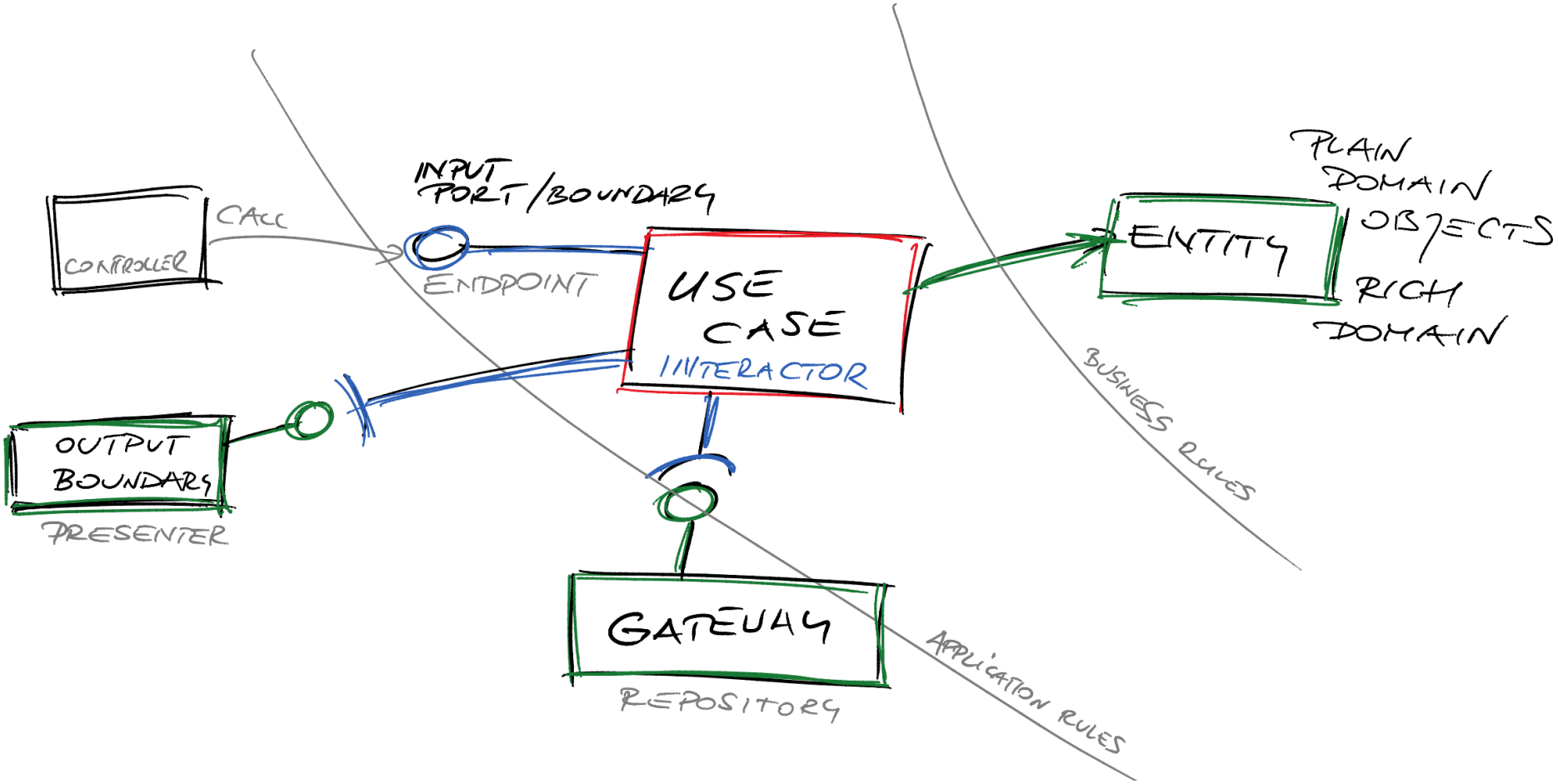
Scenariusze alternatywne:

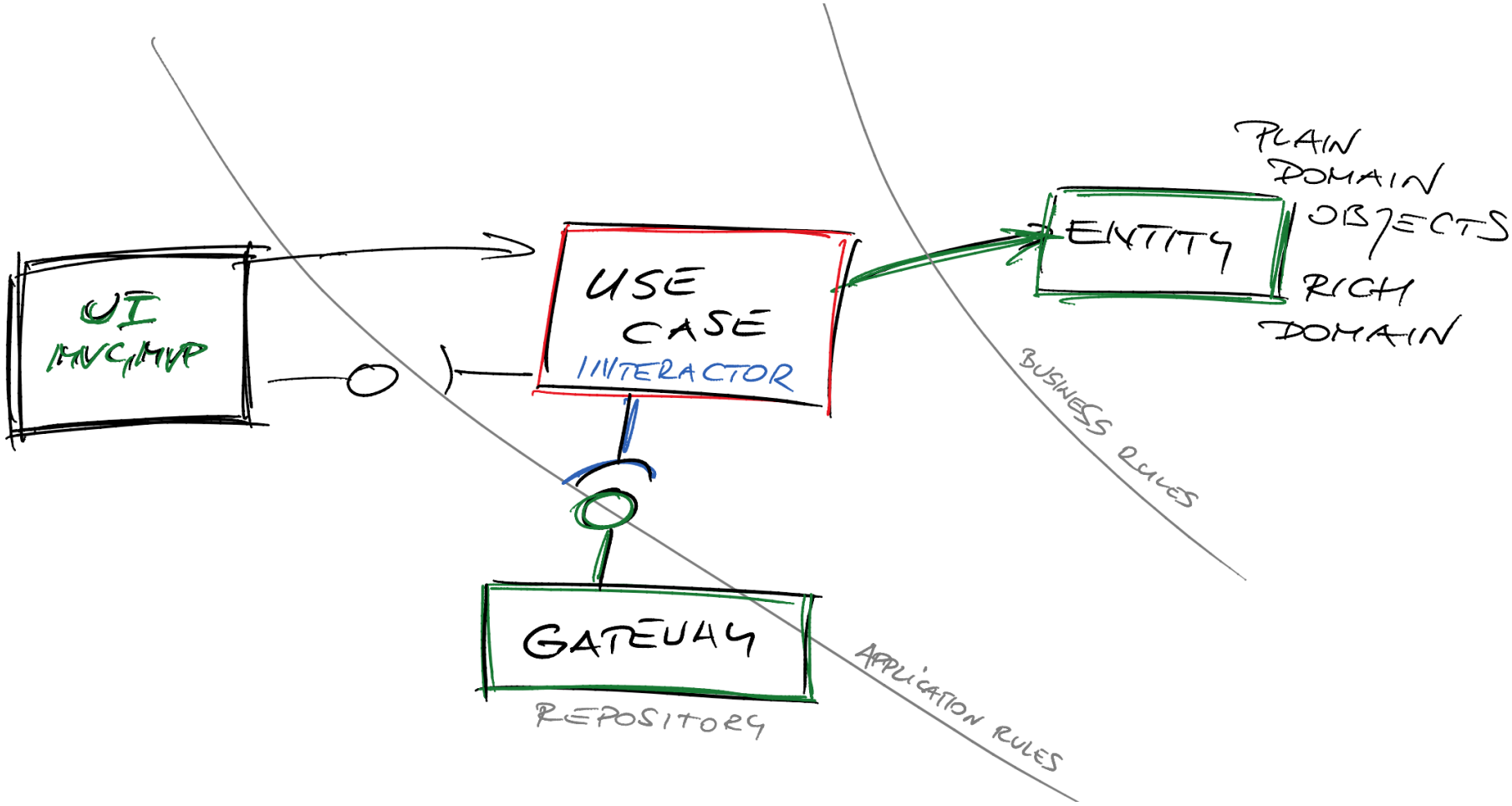
2.1 – Użytkownik nie ma jeszcze swojego konta, system proponuje ***UC- 05*** – rejestracja nowego użytkownika.

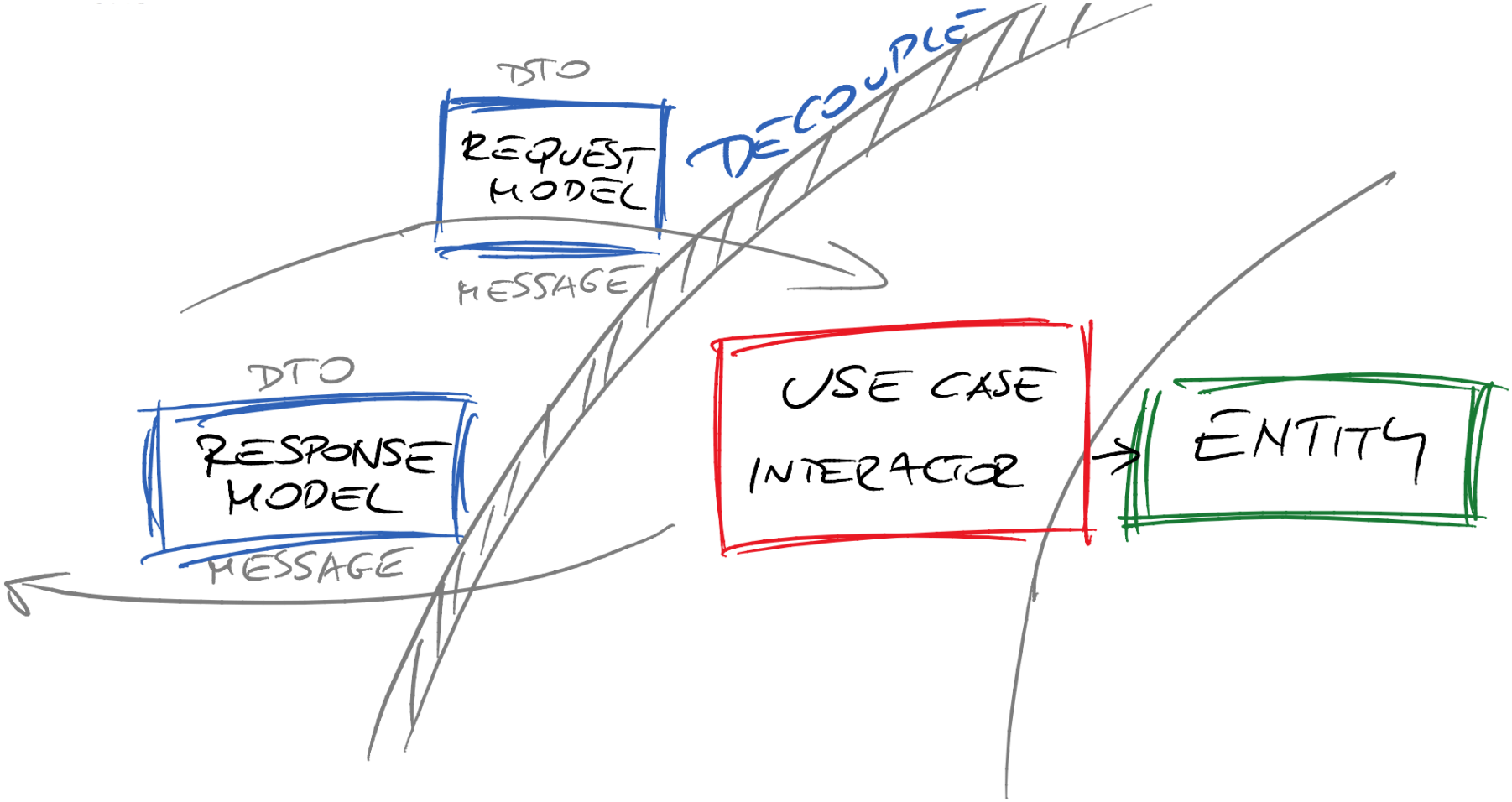
3.1 - System negatywnie weryfikuje podane przez użytkownika dane. System ponownie prosi użytkownika o podanie loginu i hasła.

3.2 – System negatywnie weryfikuje podane przez użytkownika dane, system proponuje ***UC-07*** – Procedura przypomnienia hasła.

3.3 - System negatywnie weryfikuje podane przez użytkownika dane (n>3). System blokuje konto klienta (***UC-09*** - blokada konta klienta)

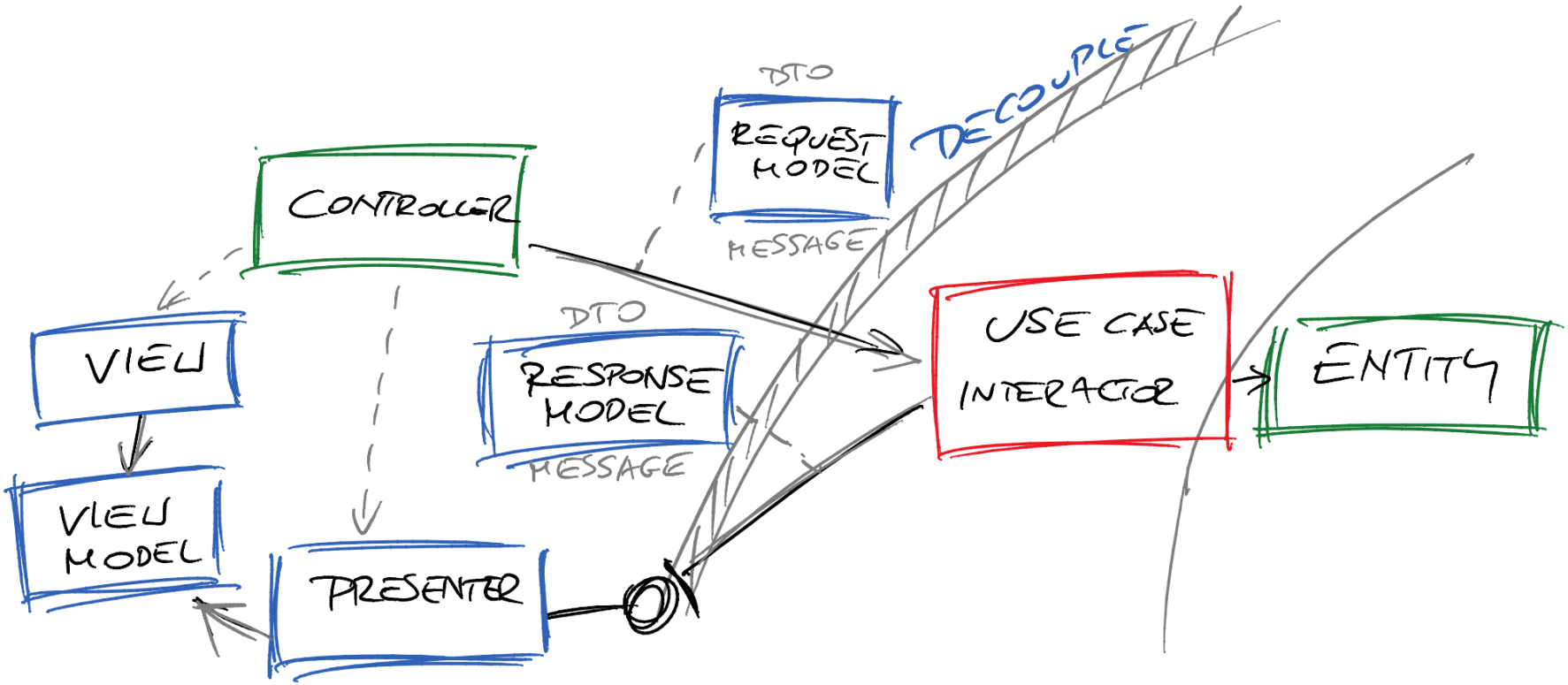

CONTROLLER
CALL
INPUT PORT / BOUNDARY
ENDPOINT
USE CASE
INTERACTOR
ENTITY
PLAIN DOMAIN OBJECTS
RICH DOMAIN
OUTPUT BOUNDARY
PRESENTER
GATEWAY
REPOSITORY
BUSINESS RULES
APPLICATION RULES

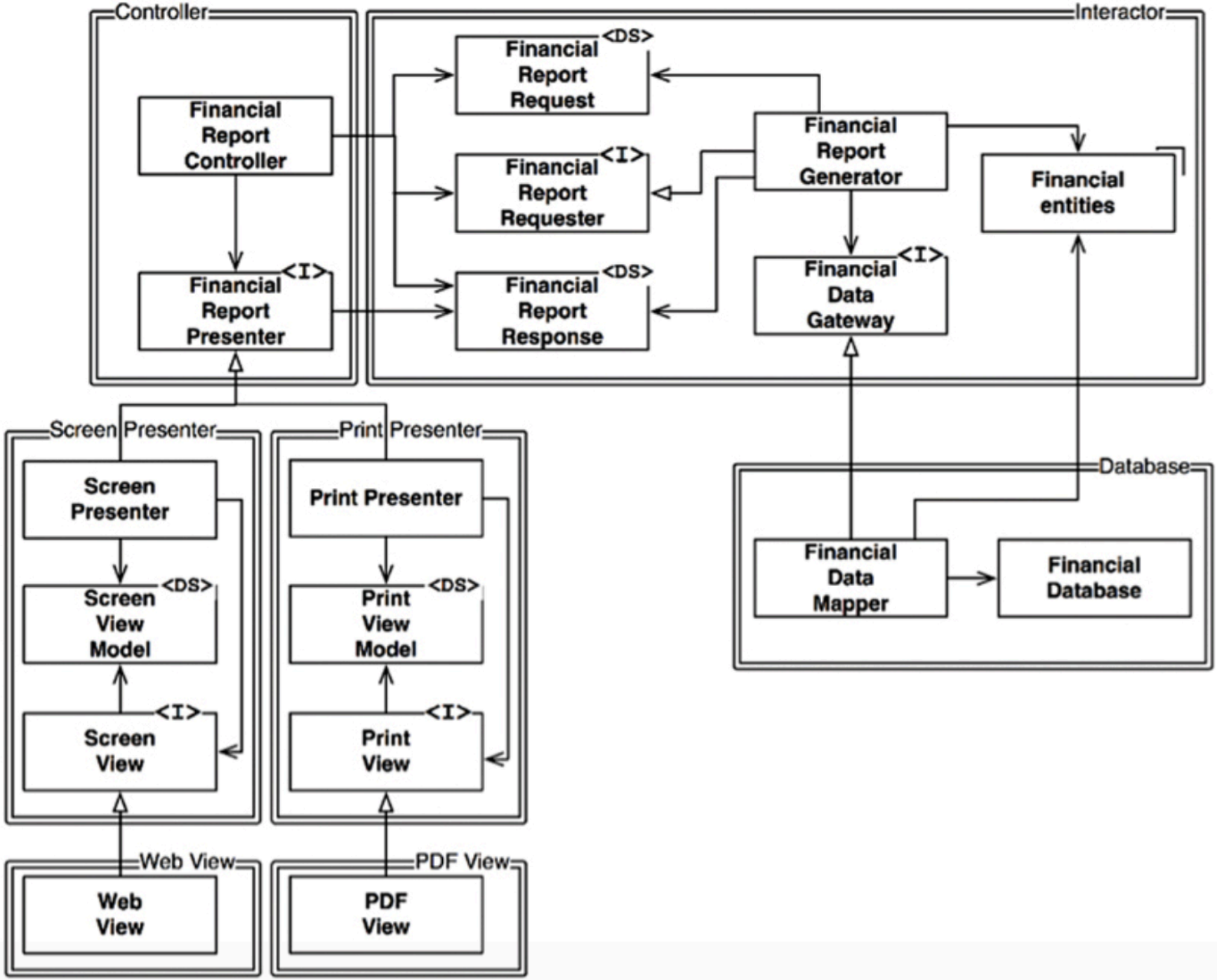

UI
MVC/MVP
USE CASE
INTERACTOR
ENTITY
PLAIN DOMAIN OBJECTS
RICH DOMAIN
BUSINESS RULES
GATEWAY
REPOSITORY
APPLICATION RULES


DTO
REQUEST MODEL
DECOUPLE
MESSAGE
DTO
RESPONSE MODEL
MESSAGE
USE CASE INTERACTOR
ENTITY

bns it

Prezentacja


DTO
REQUEST MODEL
MESSAGE
DECOUPLE
CONTROLLER
VIEW
VIEW MODEL
DTO
RESPONSE MODEL
MESSAGE
USE CASE INTERACTOR
ENTITY
PRESENTER


Controller
Financial Report Controller
Financial Report Presenter
<I>
Interactor
Financial Report Request
<DS>
Financial Report Requester
<I>
Financial Report Response
<DS>
Financial Report Generator
Financial Data Gateway
<I>
Financial entities
Screen Presenter
Screen Presenter
Screen View Model
<DS>
Screen View
<I>
Print Presenter
Print Presenter
Print View Model
<DS>
Print View
<I>
Database
Financial Data Mapper
Financial Database
Web View
Web View
PDF View
PDF View

bns it

Wady i zalety

Zalety

- Jasna separacja warstw - nie tylko oddzielamy domeny ale i „warstwy”
- Względnie łatwa wymiana warstwy technologicznej
- Architektura dobrze otwiera na różnych klientów
- Testowalność - przy zachowaniu interfejsów niemal w każdą stronę
- Można rozwijać logikę dziedzinową niezależnie od technikaliów

Wady

- Jak to się ma do UC?
- Duża złożoność, szczególnie przy purystycznej implementacji
  - Duża ilość interfejsów na granicach między warstwami
- Osobne modele dla dziedziny, repository, widoku, interactora Czy współdzielić modele Repository i Entity?
  - Purystycznie − konfiguracja zewnętrzna (XML, fluent api)
  - Akceptujemy niektóre adnotacje (np. w Java @Inject, ORM)
  - Dwie wersje − czysta i z adnotacjami „nadpisana”
- Dependency Inversion wprowadza niebezpośredniość komunikacji (zdegenerowany obserwator)

Kiedy używać?

- Złożone dziedziny – w połączeniu z DDD
- Krytyczne systemy/aplikacje
- Zespoły o wysokich kompetencjach
- Wielu klientów
- Dla długotrwałych projektów
  - Dla kilkutygodniowych lub CRUDowych to będzie killer
  - Chyba że chcesz potrenować
- Raczej ewolucja w stronę Clean Architecture
  - Clean Architecture nie jest KISS