THE DOMAIN IN FOCUS

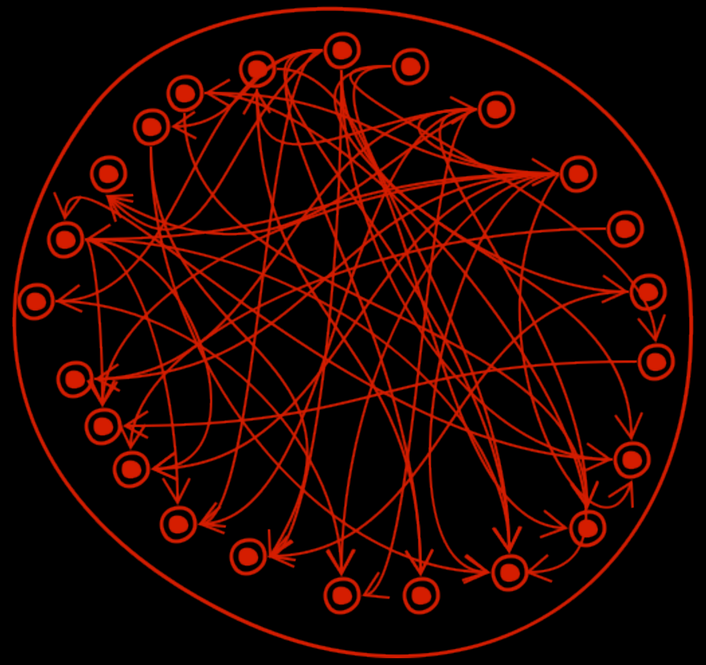
PORTS, ADAPTERS AND HEXAGONAL ARCHITECTURE

BJORN.BESKOW@CALLISTAENTERPRISE.SE

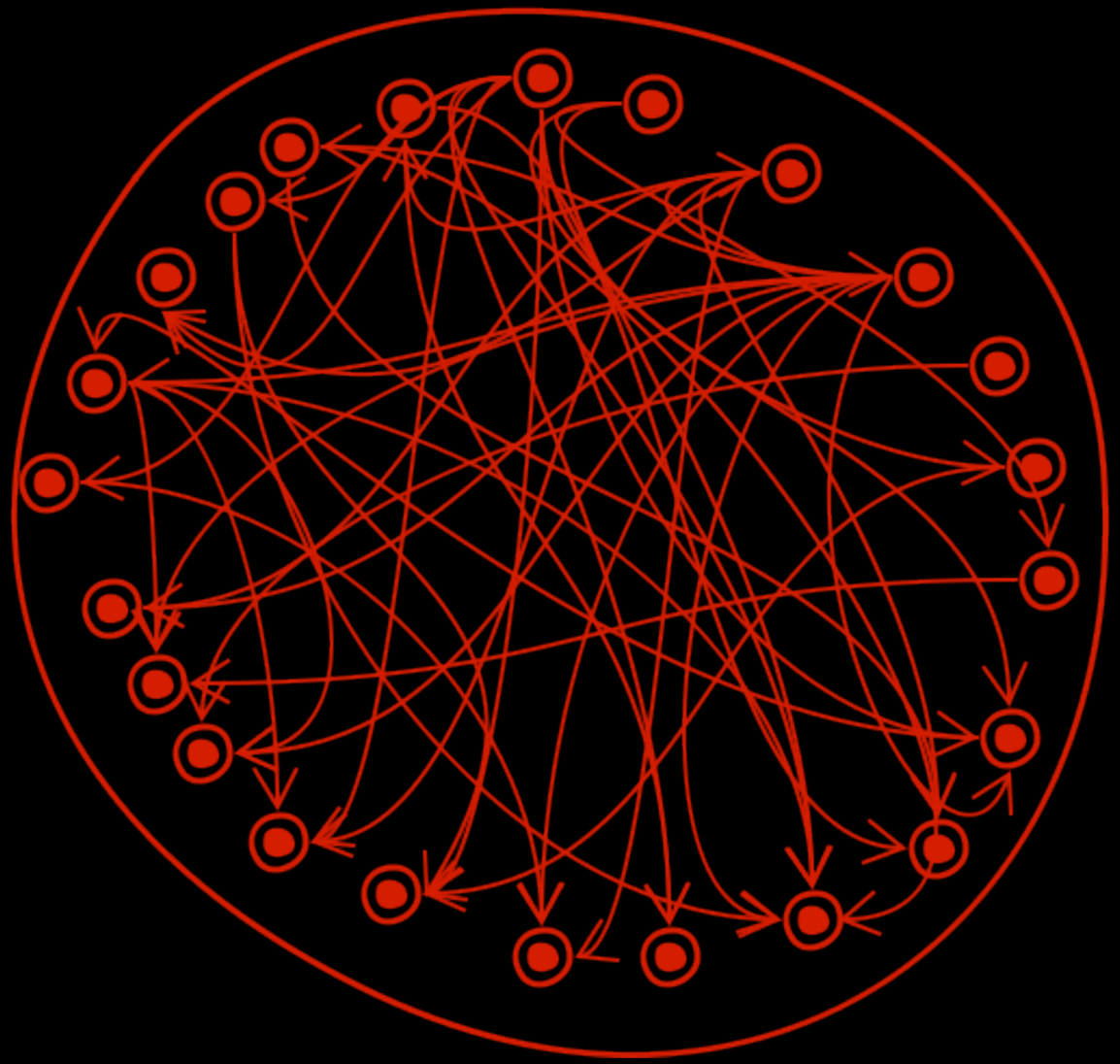
CADEC 2025.01.23 & 2025.01.29 | CALLISTAENTERPRISE.SE

CALLISTA

RECURRING THEME: SOFTWARE COMPLEXITY GROWS OVER TIME

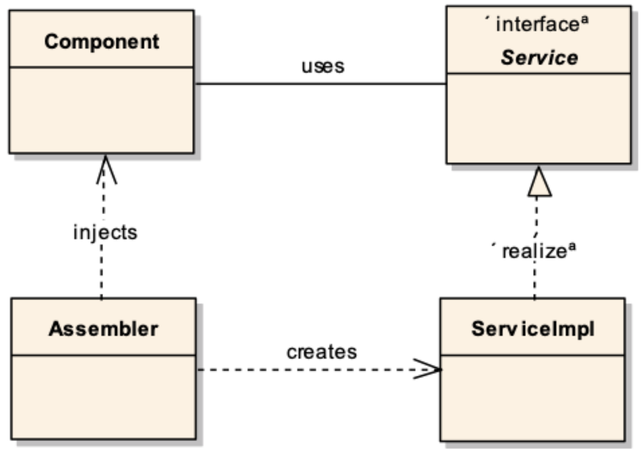


THE ROOT OF EVIL: UNMANAGED DEPENDENCIES



ARCHITECTURE: NEVER-ENDING BATTLE AGAINST CHAOS

Dependency Injection

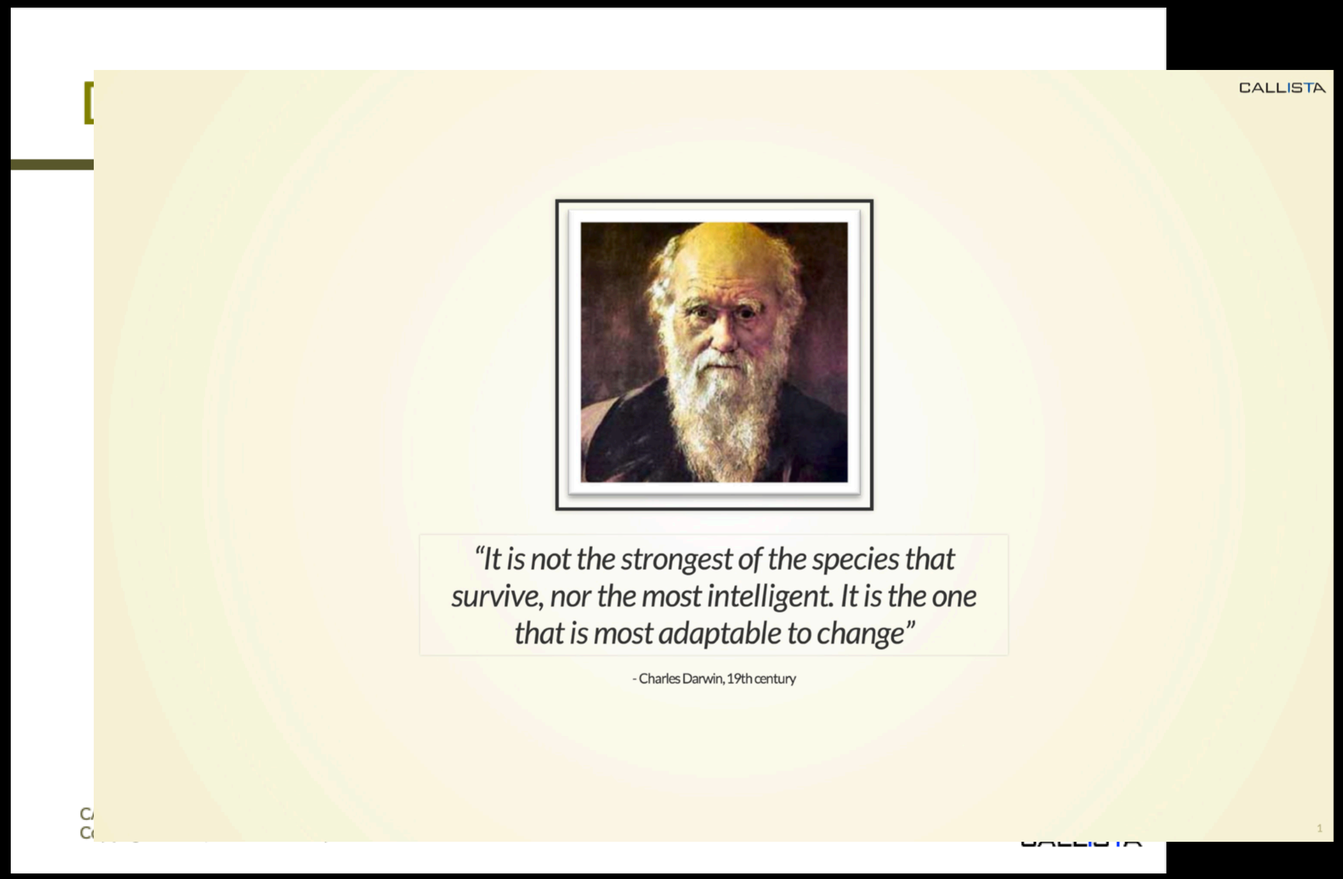



```
public void setDao(CustomerDao dao) {
    this.dao = dao;
}
```

CADEC2006, DI, Slide 6

CALLISTA

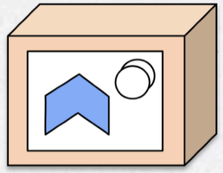
ARCHITECTURE: NEVER-ENDING BATTLE AGAINST CHAOS



ARCHITECTURE: NEVER-ENDING BATTLE AGAINST CHAOS

CALLISTA

WHAT'S A MICROSERVICE?



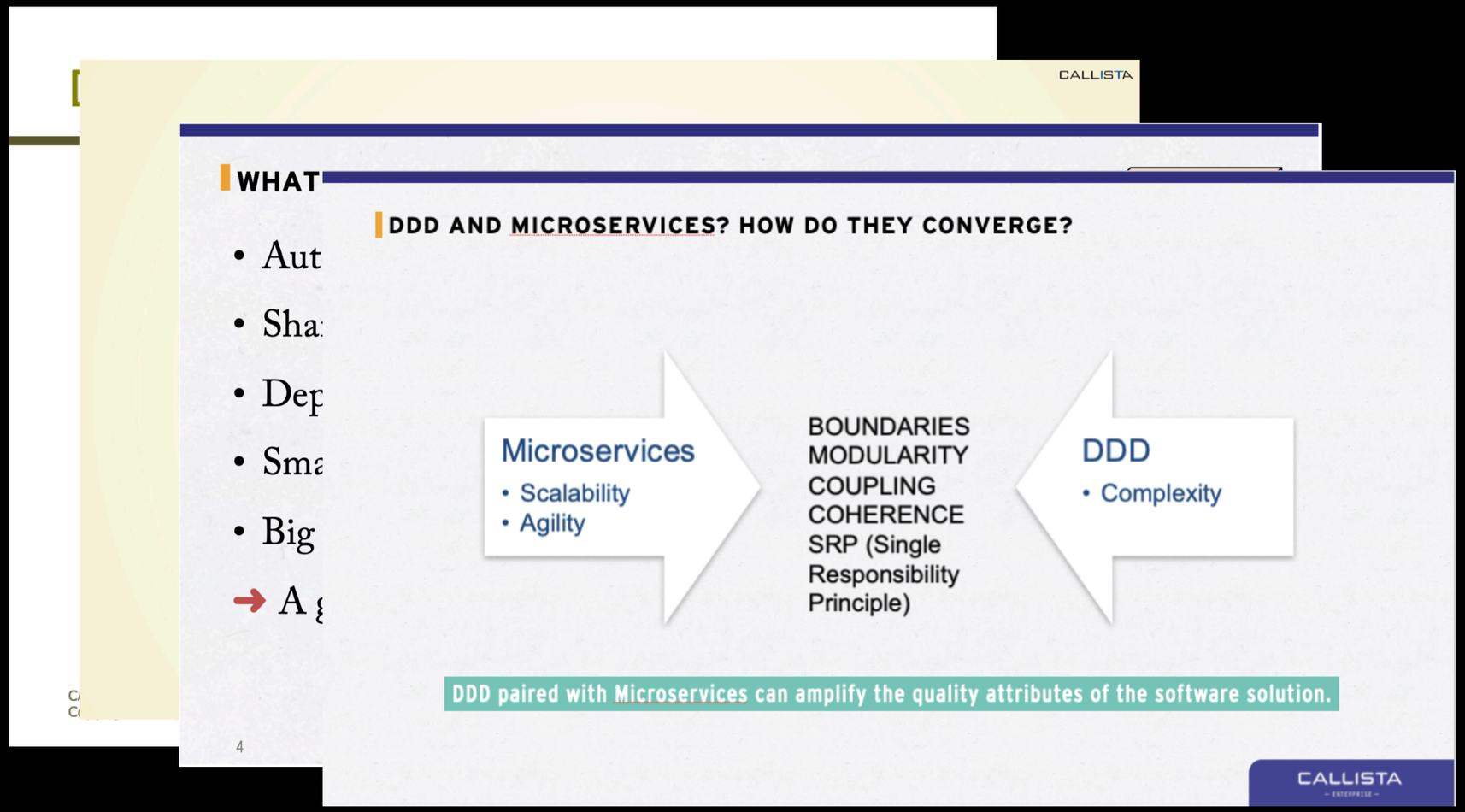
- Autonomous software component
- Share nothing architecture
- Deployed as a runtime processes
- Small enough to fit in the head of a developer
- Big enough to avoid unacceptable latency and data inconsistency…

➜ A group of microservices form a Distributed System

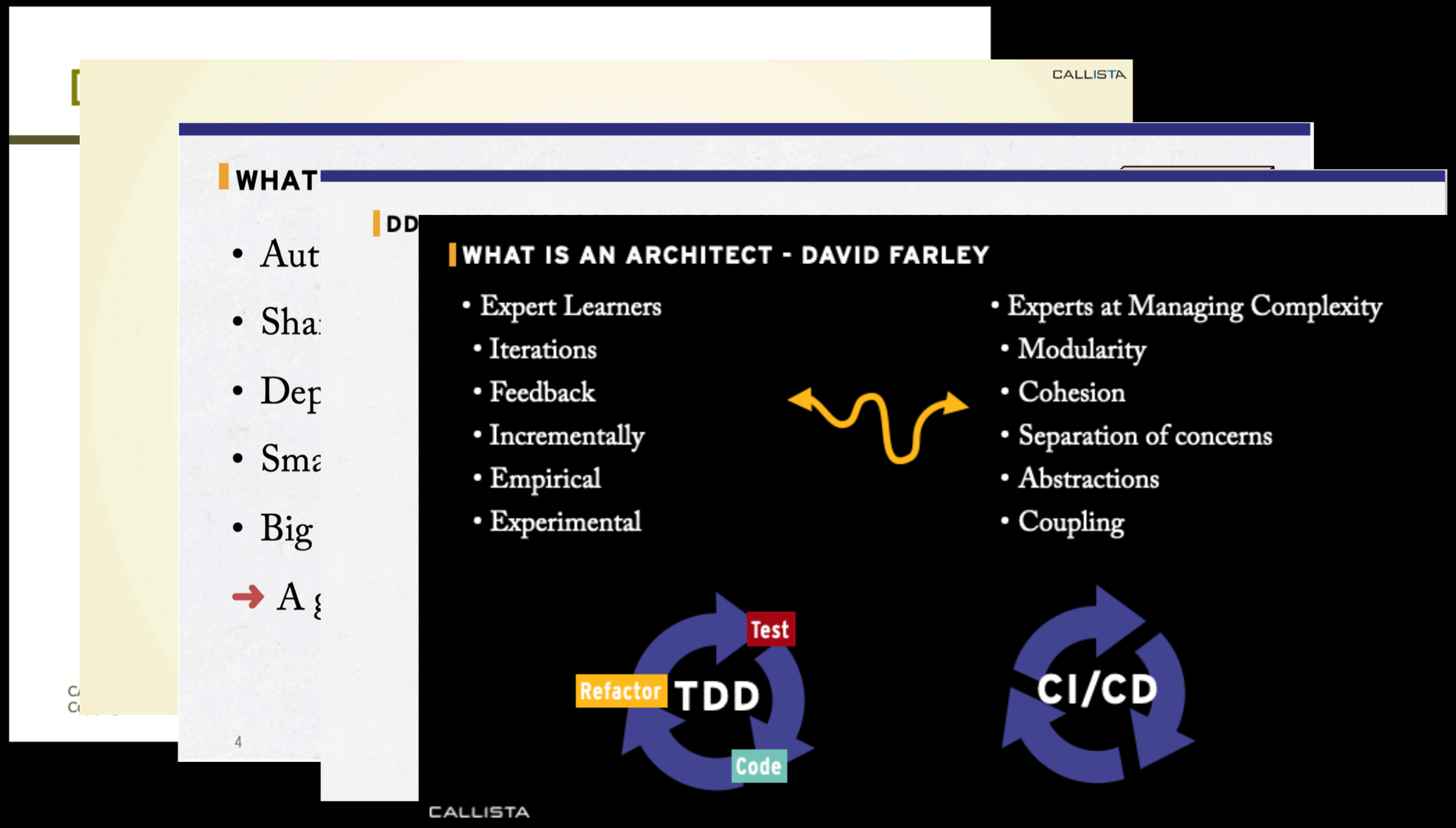
4

CALLISTA
— ENTERPRISE —

ARCHITECTURE: NEVER-ENDING BATTLE AGAINST CHAOS



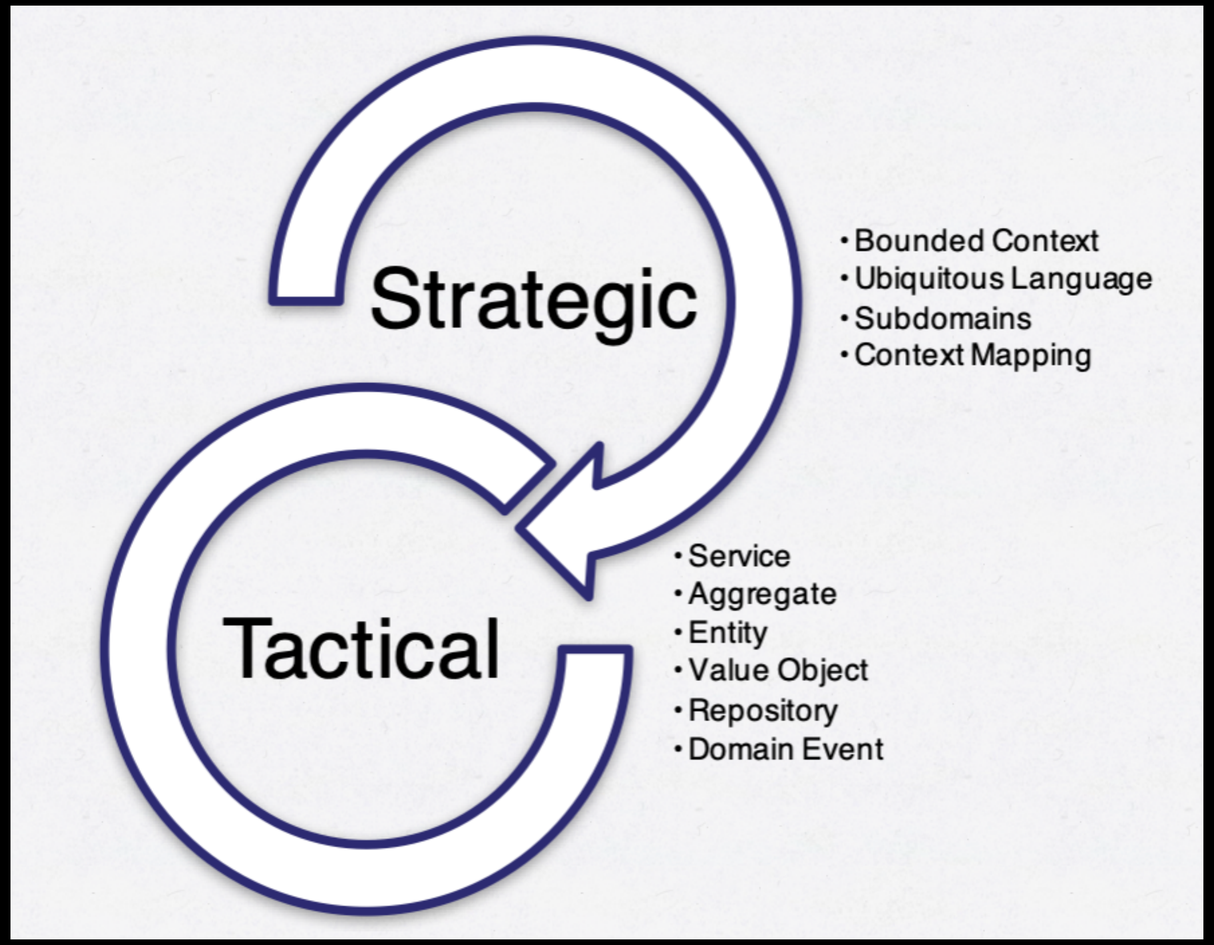
ARCHITECTURE: NEVER-ENDING BATTLE AGAINST CHAOS

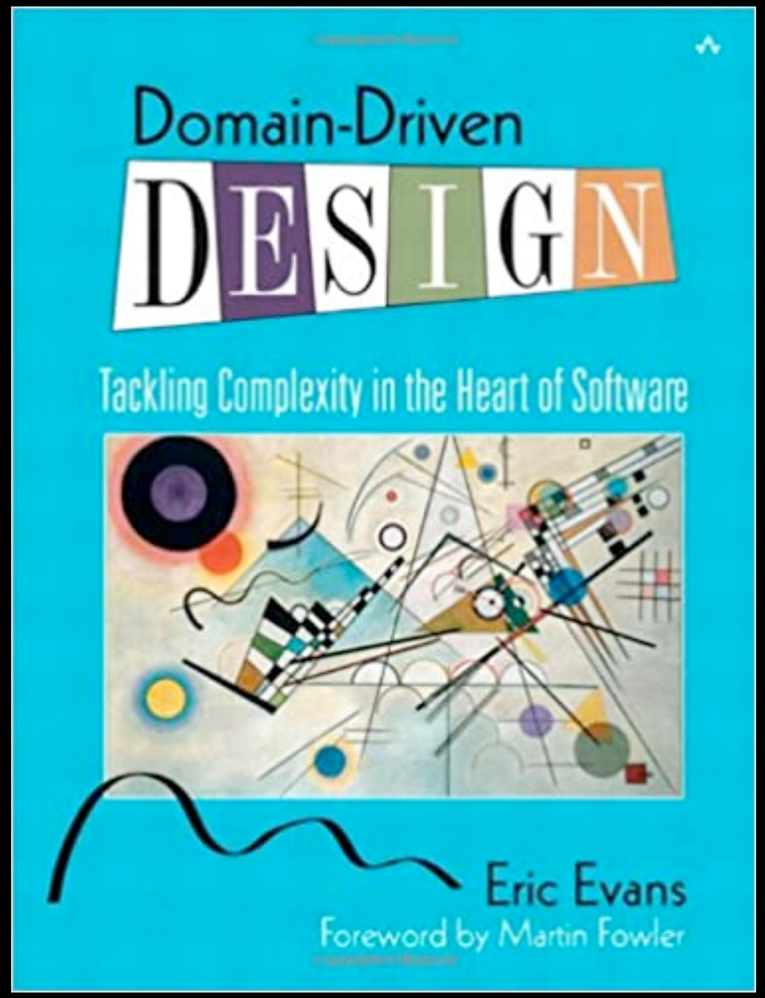


AGENDA

- The problem:
  - Taming complexity by managing dependencies
- Architectural Layering
  - The traditional way
  - Why does it hurt?
- Hexagons or Ports and Adapters
- Code examples
- Conclusions

DOMAIN-DRIVEN DESIGN

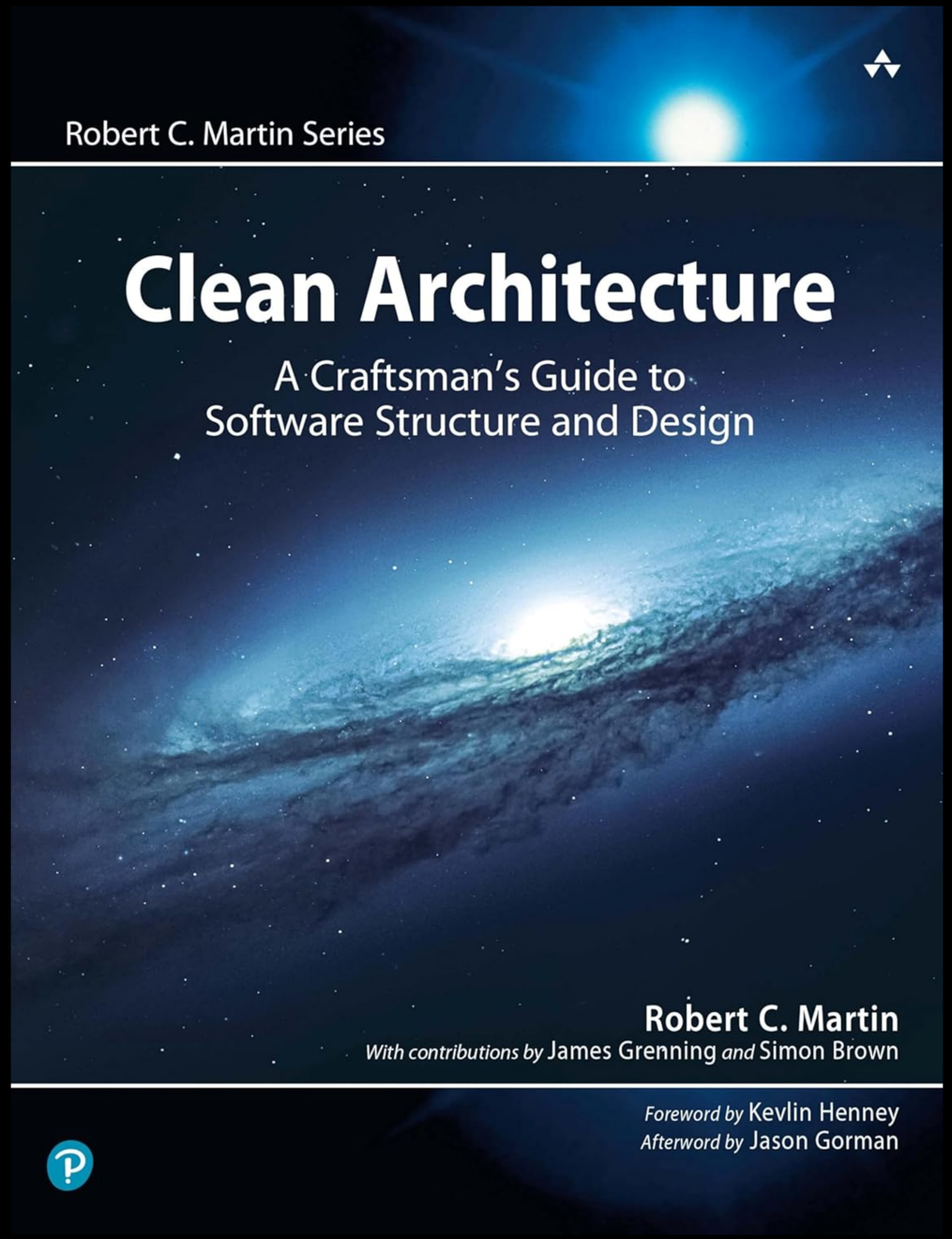




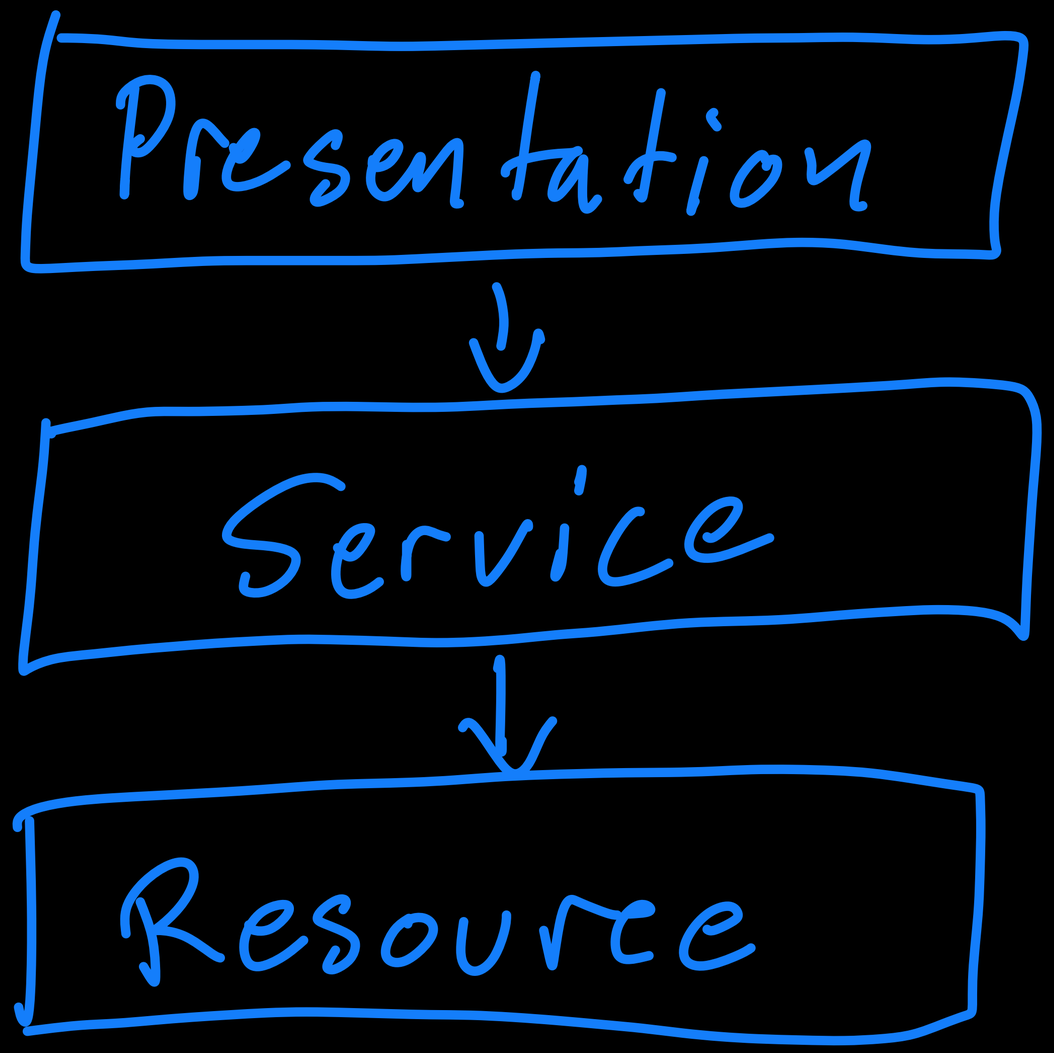
DESIGN PRINCIPLES

- SOLID
  - **S**ingle Responsibility Principle
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

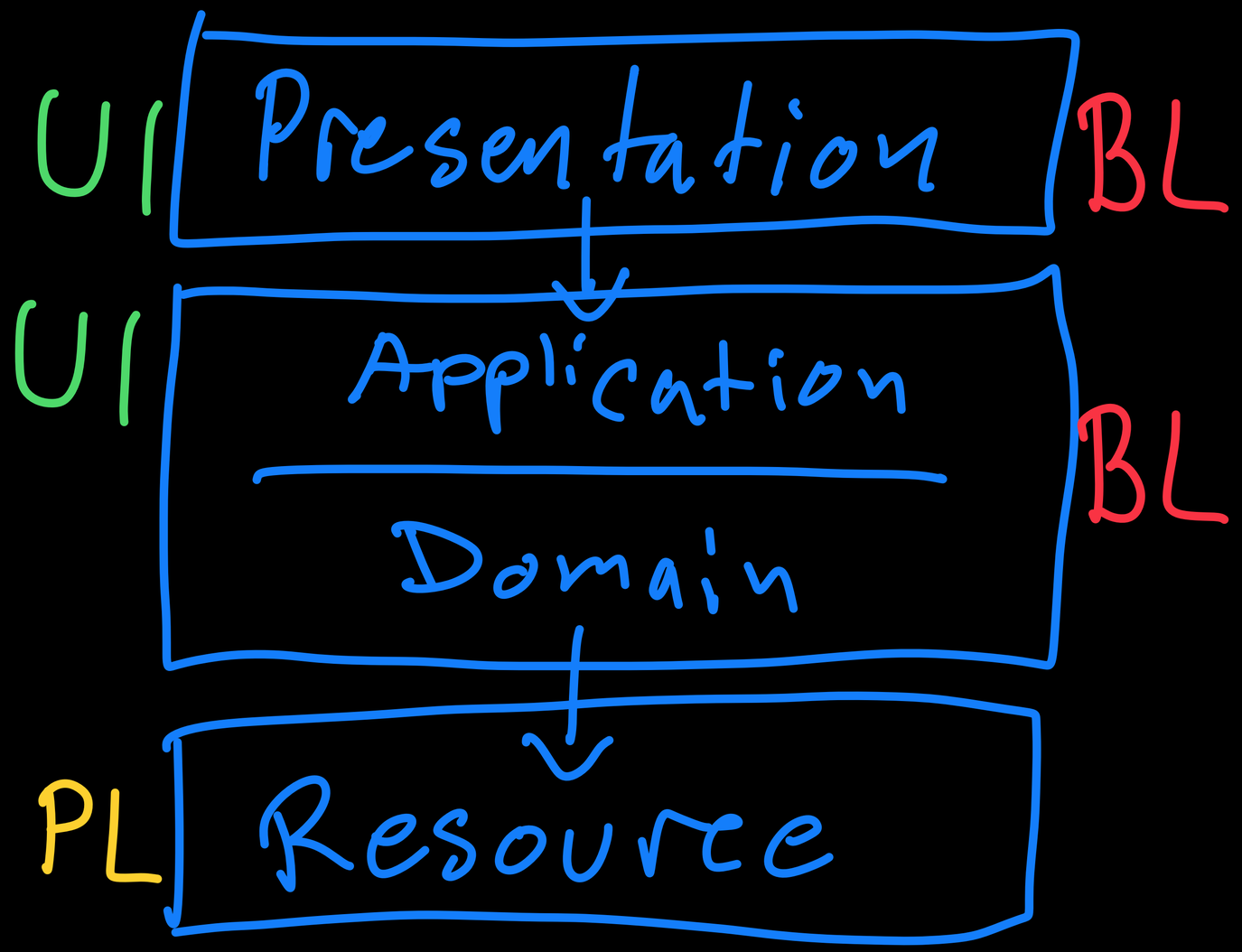
- Architectural Layering



TRADITIONAL 1-DIMENSIONAL LAYERING



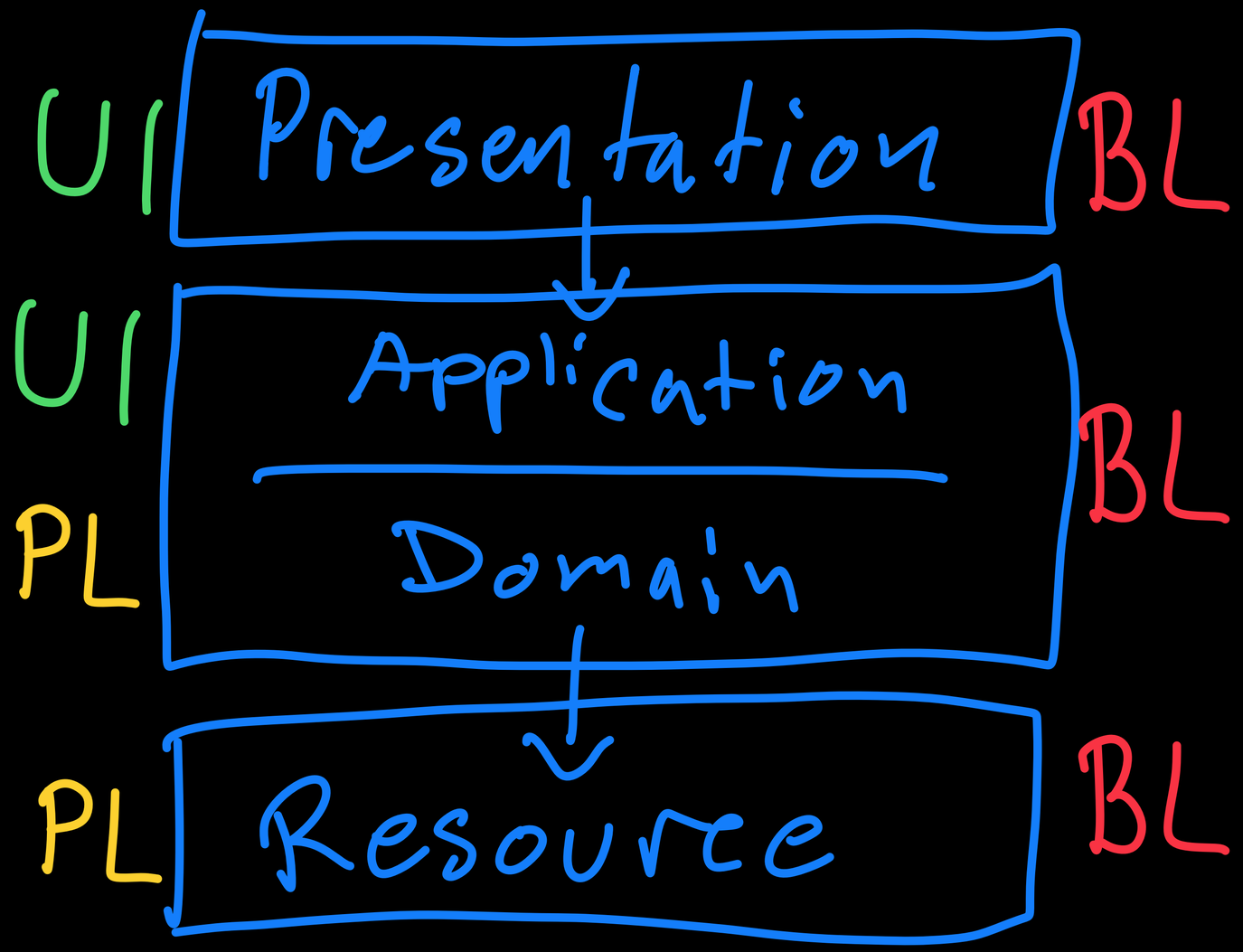
TRADITIONAL 1-DIMENSIONAL LAYERING



TRADITIONAL LAYERING - LEAKING PRESENTATION DETAILS

```
public class Product {

    @JsonProperty("productId")
    private Long productId;

    @NotNull
    @Size(max = 255)
    @JsonProperty("name")
    private String name;

    @NotNull
    @Size(max = 255)
    @JsonProperty("articleId")
    private String articleId;

    @NotNull
    @JsonProperty("inventory")
    private Long inventory;
```

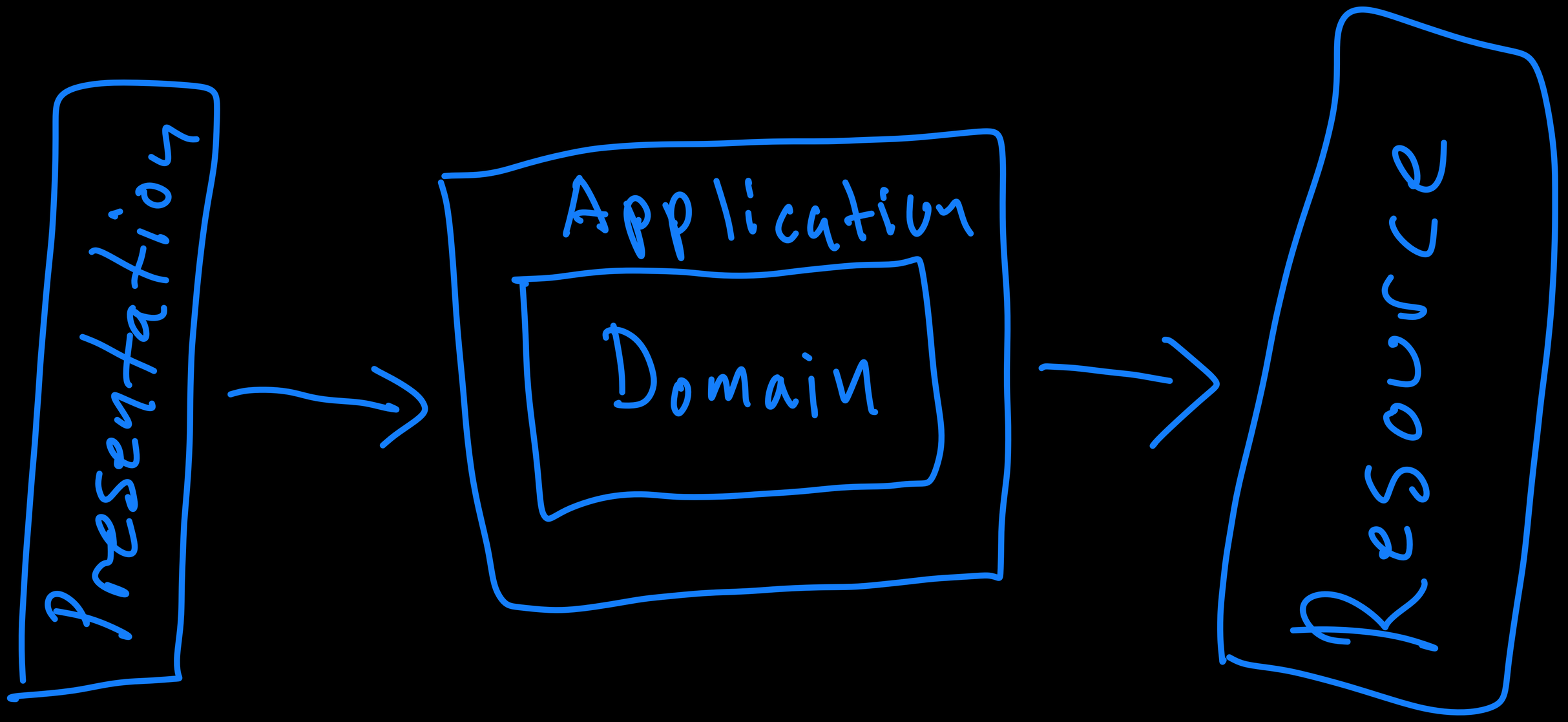
TRADITIONAL 1-DIMENSIONAL LAYERING



TRADITIONAL LAYERING - LEAKING PERSISTENCE DETAILS

```
@Entity
@Table(name = "product")
public class ProductVariant {

    @Id
    @Column(name = "id", unique = true, nullable = false, updatable = false)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "product_seq")
    @SequenceGenerator(name = "product_seq", sequenceName = "product_seq")
    protected Long productId;

    @Version
    @Column(name = "version", nullable = false,
            columnDefinition = "int default 0")
    protected Integer version;
```

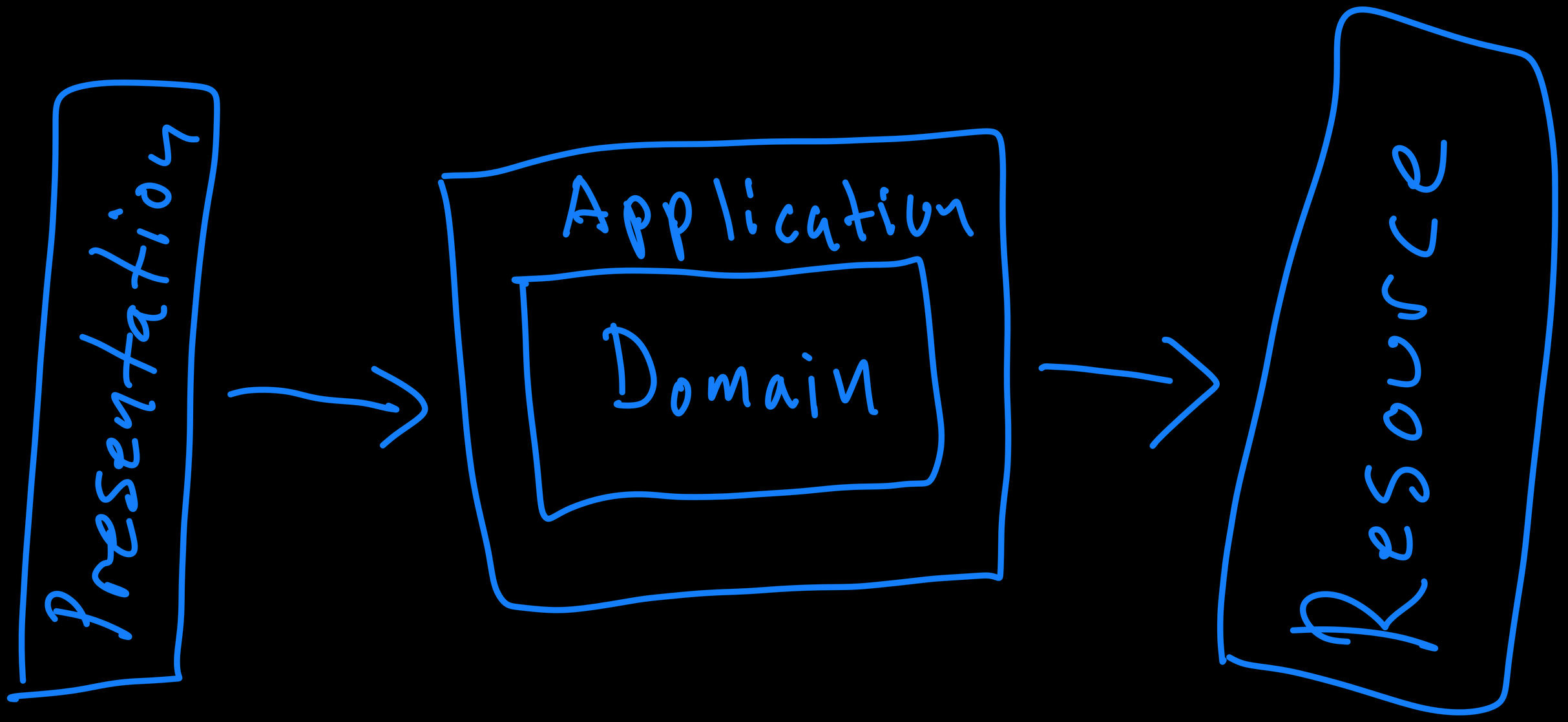
LAYERING



SOLID: SINGLE RESPONSIBILITY PRINCIPLE

"A class should have one,
and only one,
reason to change"

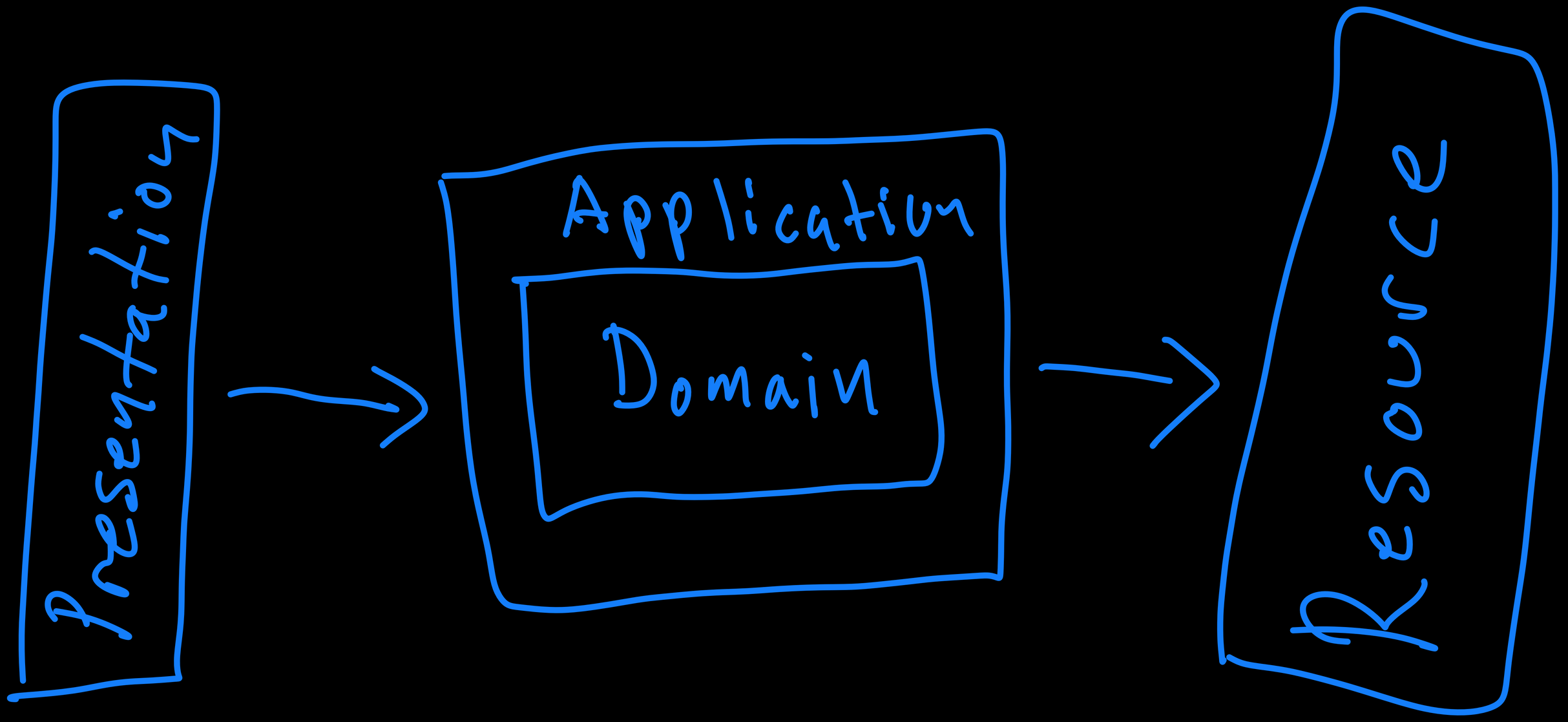
LAYERING



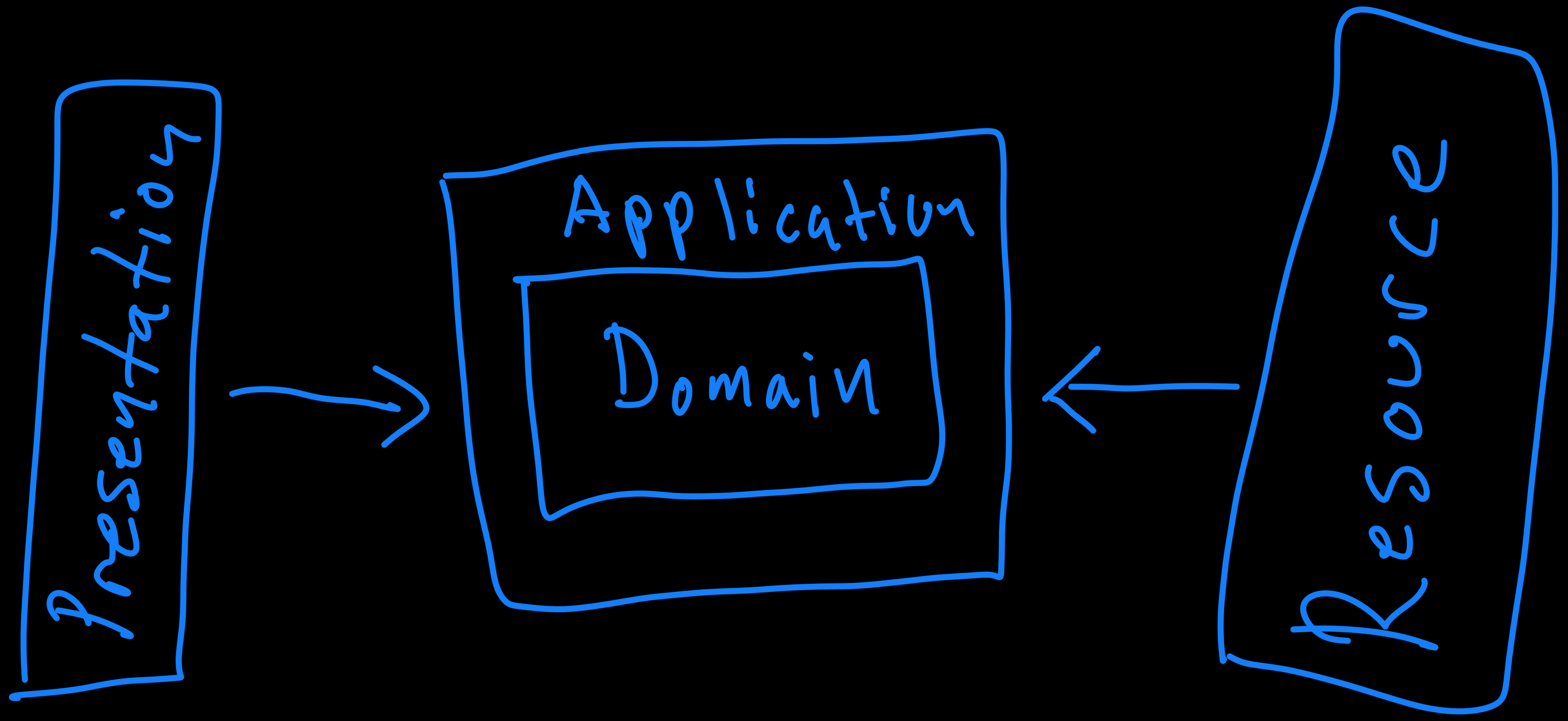
SOLID: DEPENDENCY INVERSION PRINCIPLE

"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."

LAYERING



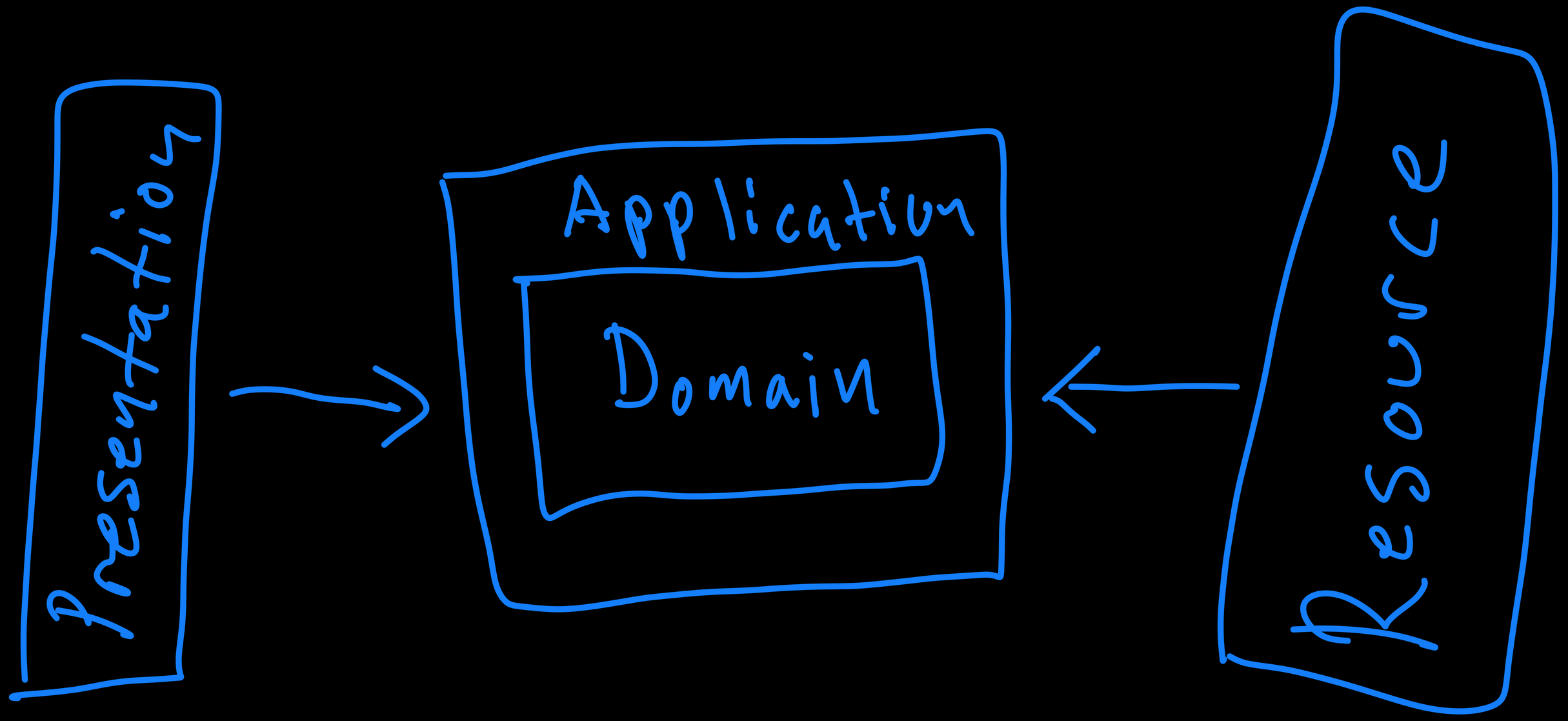
INVERTING THE DEPENDENCY



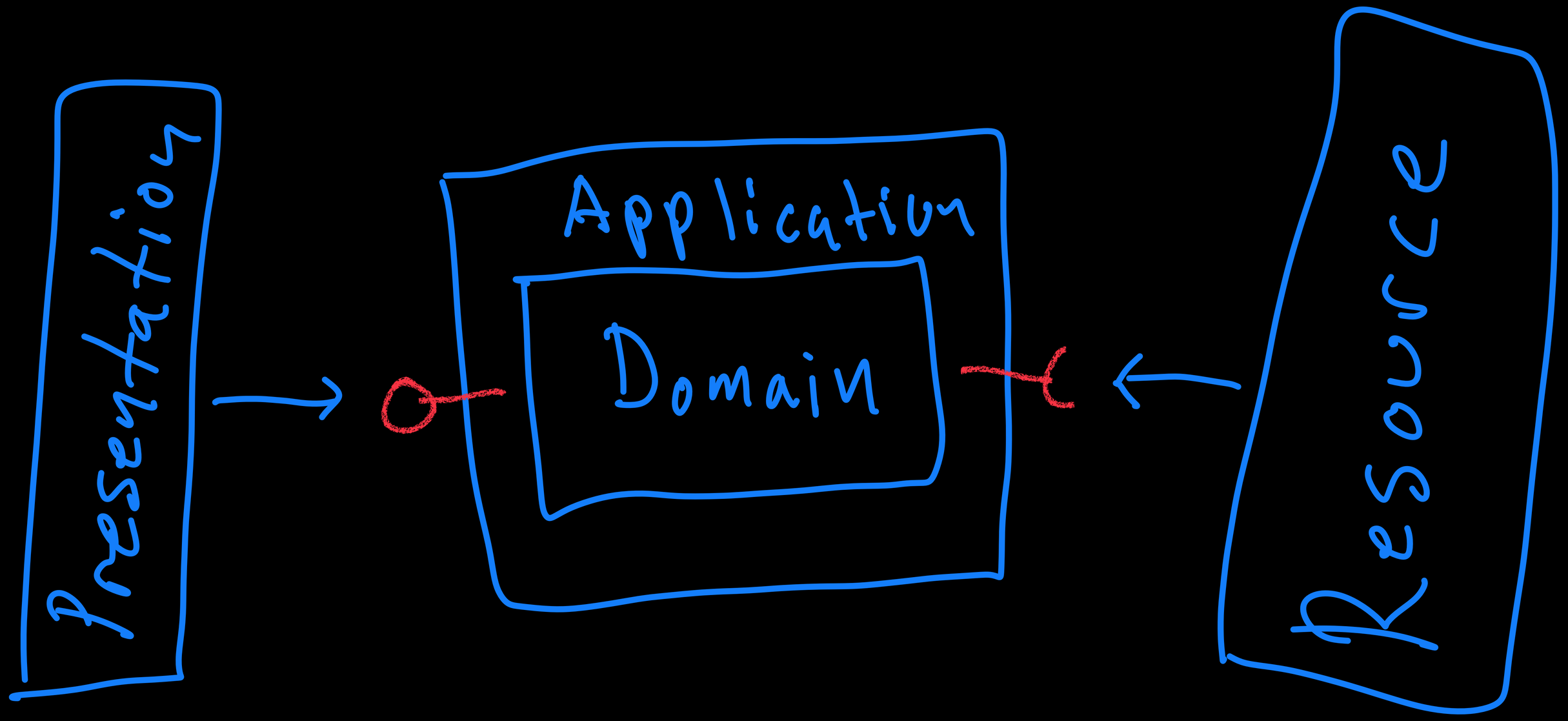
SOLID: DEPENDENCY INVERSION PRINCIPLE

"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."

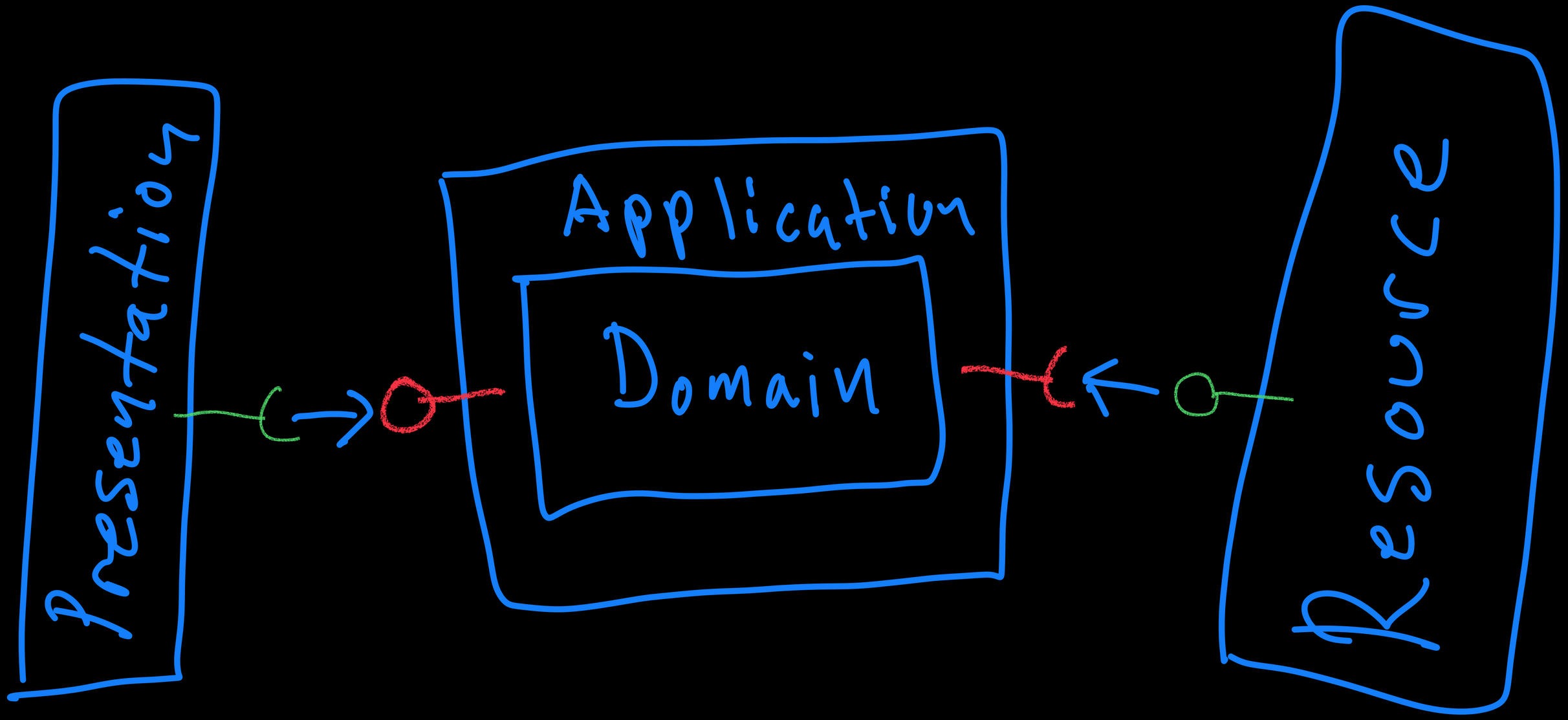
INVERTING THE DEPENDENCY



ADDING PORTS



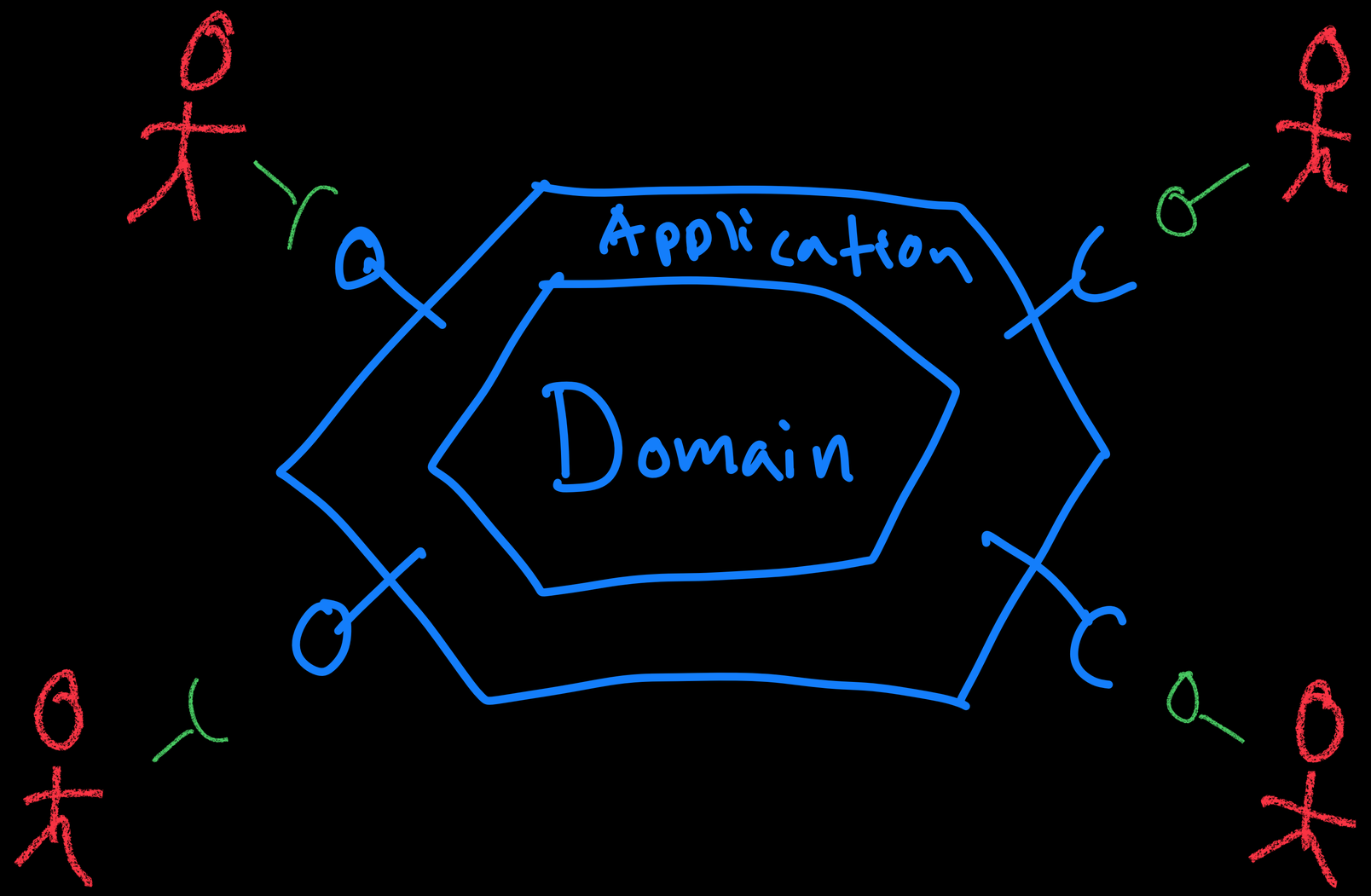
ADDING ADAPTERS



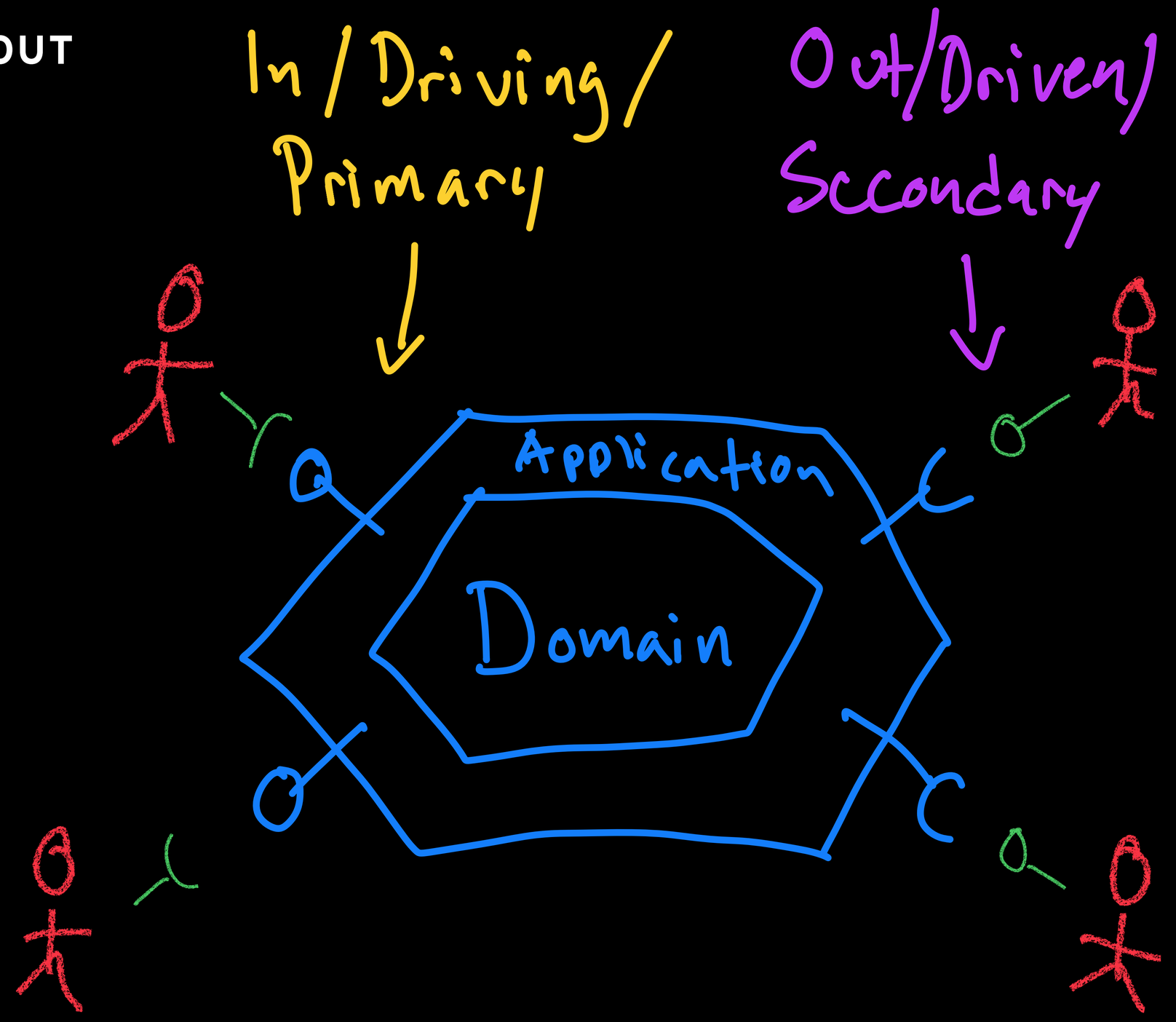
SOLID: INTERFACE SEGREGATION PRINCIPLE

"A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use."

MULTIPLE PORTS



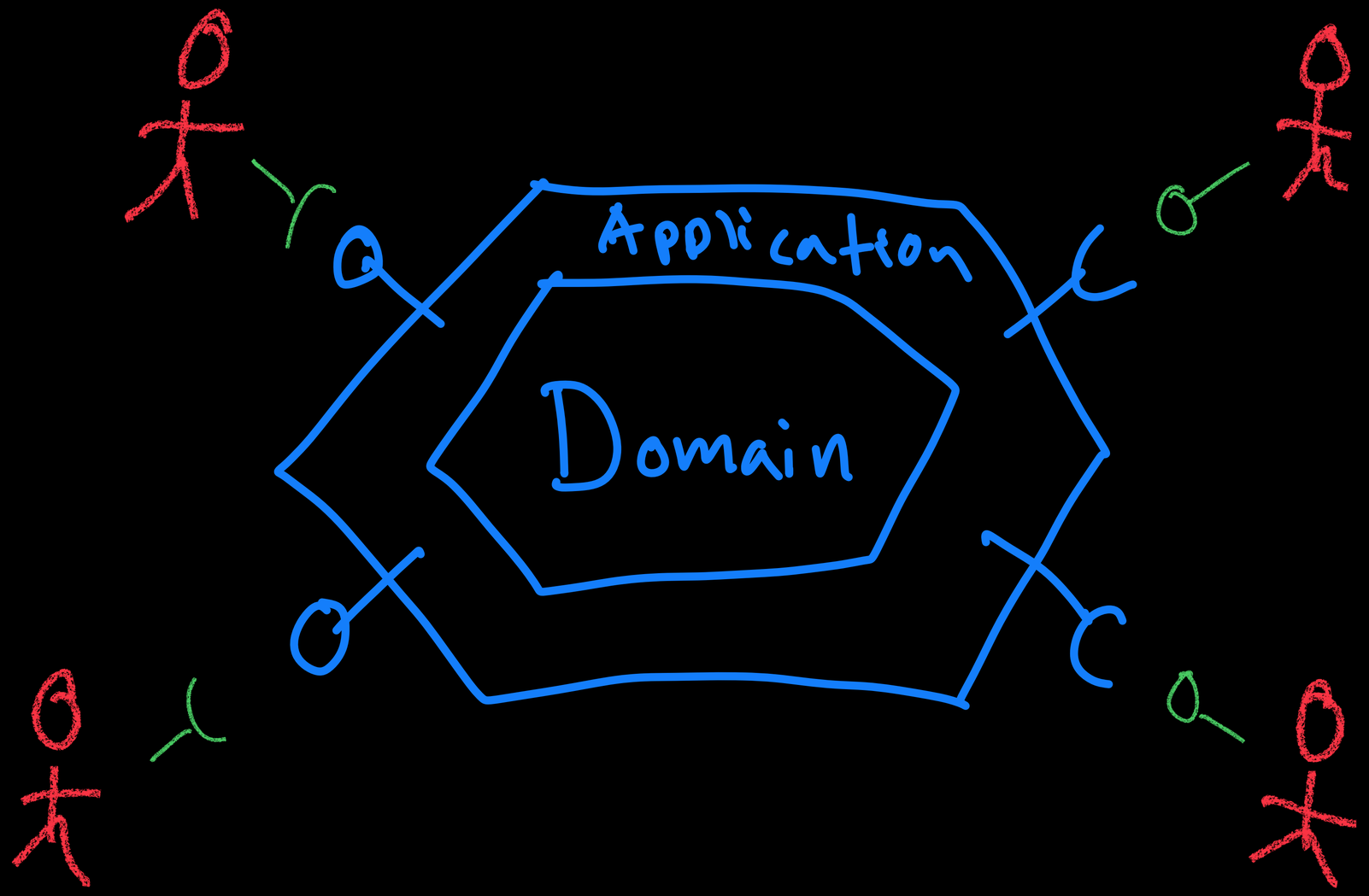
MULTIPLE PORTS - IN AND OUT



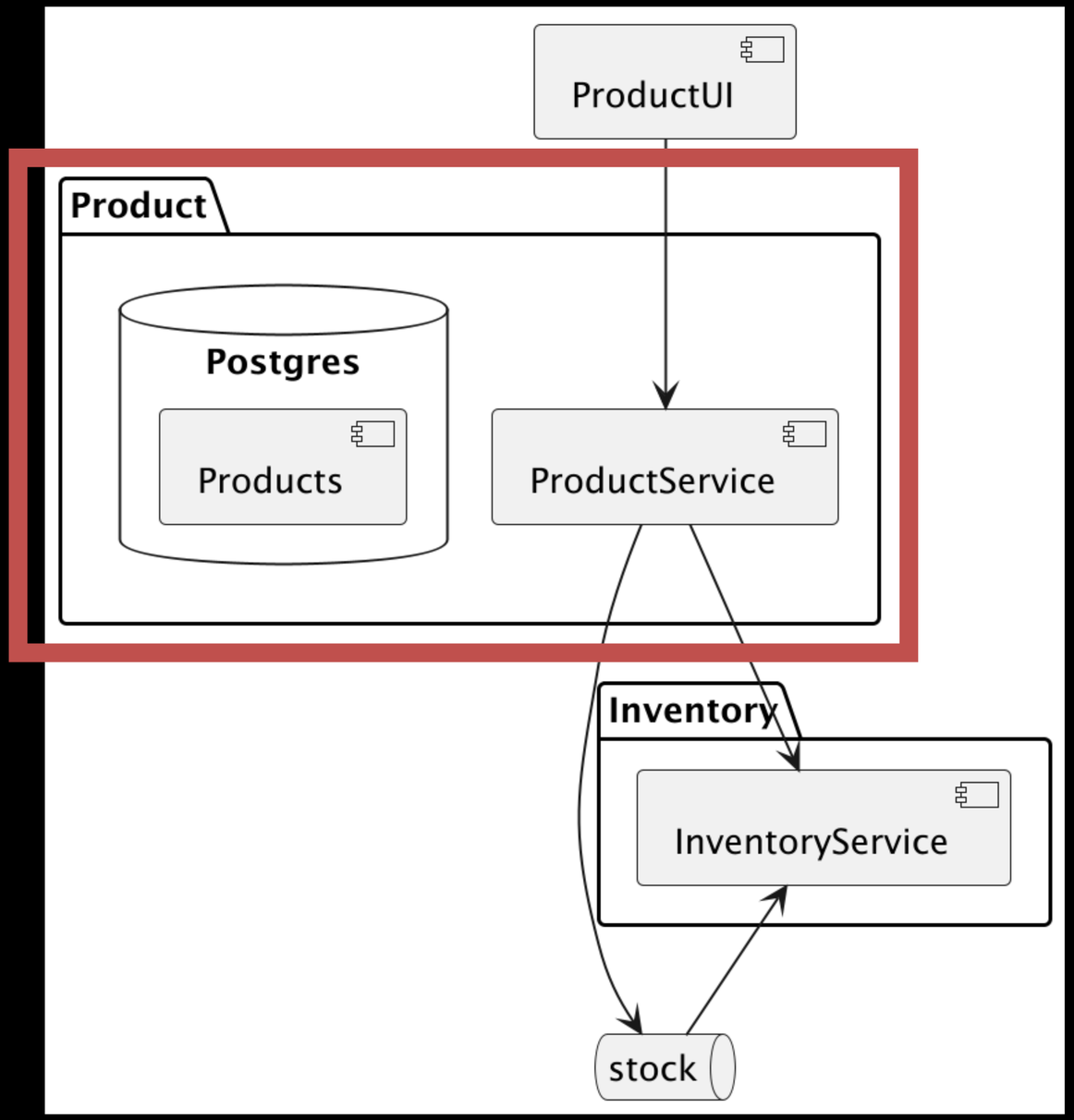
HEXAGONAL ARCHITECTURE

- a.k.a.
  - “Ports and Adapters” or
  - “Onion Architecture” or
  - “Clean Architecture”

Alistair Cockburn



CODE EXAMPLES ...



IMPLICATIONS

- Structural separation between “inside” and “outside”, using explicit Ports and Adapters
- ”Configurator” or Dependency Injection Framework required to wire the parts together
- Mapping between Layers/ Abstractions
- Reduced coupling leads to greatly simplified testing

“Implications icons created by Circlon Tech - Flaticon"

CONCLUSIONS

- Pros:
  - Reduces unhealthy coupling between business/domain logic and input/output details
  - Greatly simplifies testing by decoupling technical detail
  - Important enabler for keeping the domain model simple, concise and maintainable
- Cons:
  - Slightly more advanced with steeper learning curve
  - Increased number of classes/interfaces to maintain
  - Increased mapping effort

TIME FOR QUESTIONS?

- Pros:
  - Reduces unhealthy coupling between business/domain logic and input/output
  - Greatly simplifies testing by decoupling technical detail
  - Important enabler for keeping the domain model simple, concise and maintainable
- Cons:
  - Slightly more advanced with steeper learning curve
  - Increased number of classes/interfaces to maintain
  - Increased mapping effort

https://github.com/callistaenterprise/cadec2025-hexagonal