Domain-Driven Design

Baby Steps

Žilvinas Kuusas / Kaunas PHP v.28

How I found DDD?

- Project with complicated logic
- Complex business problems
- Implementation via experiments - domain modeling

When you need DDD

- Want to build long-lasting codebase
- Project contains lots of business logic
- You have a team
- Multiple teams working on a project

Domain-driven design

- Not a technology
- Not a methodology

Domain-driven design

- Structure of practices for making design decisions
- Focused on core domain and domain logic
- Technical and business people collaboration
- Ubiquitous language

Domain

Concept of specific business area - real world problem.

Domain model

Systematic code which solves problems described in domain in software level.

Ubiquitous language

The same language used in domain model both by tech and business people to describe activities in domain.

Design pattern knowledge

- Dependency Injection
- Factory
- Data Mapper
- Adapter
- Mediator
- Command
- …

Using a framework?

Forget it for a while.

Take advantage of OOP

- Modular structure
- Clear interface of object
- Messaging between objects

Model-driven design

Example

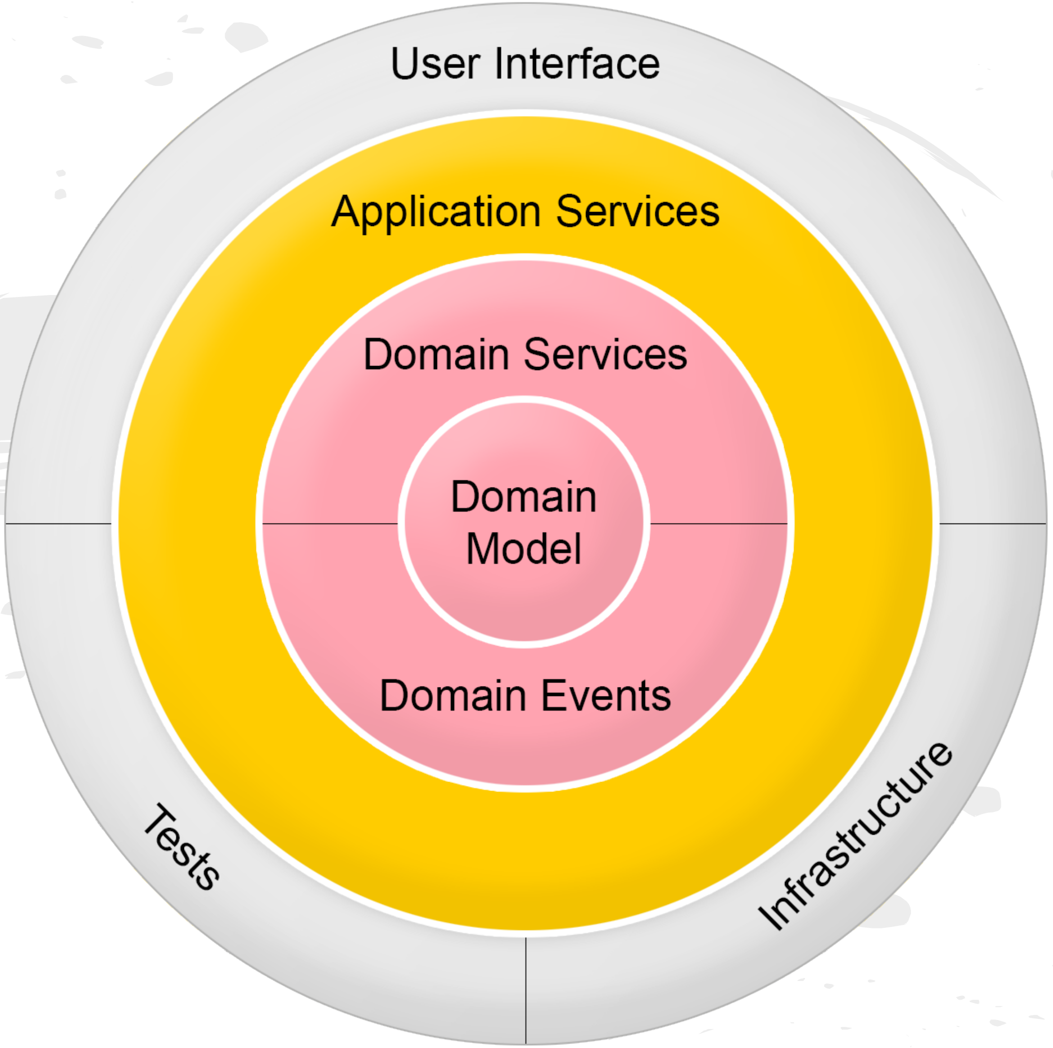
Where to start?

- Analyze domain problems
- Use same language in a team
- Distillate domain objects (mostly entities)
- Define domain events/actions
- Write code
- Repeat

Building blocks

- Entity
- Value object
- Repository
- Domain service
- Domain event
- Application service

Layered architecture



Domain

- Expenses tracker
  - Log incomes
  - Log outcomes
  - Tag transactions

Modeling

- Understand domain (notes, drawings, UML)
- Write behaviors (BDD)
- Write tests (TDD)
- Create classes

Entities

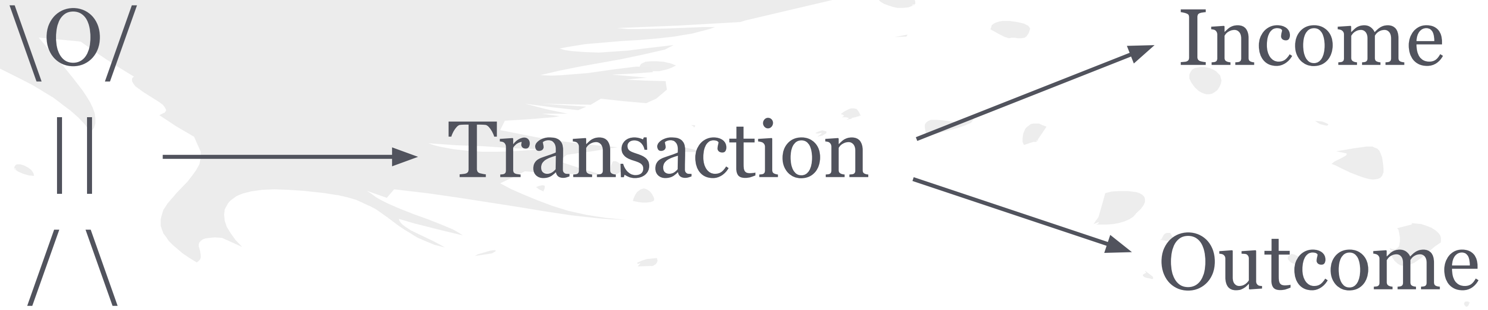
Transaction

- created : DateTime
- amount : float
- tag : Tag
- type : string
- description : text

Tag

- name : string

Use case



Domain events and actions

Events

- Transaction created

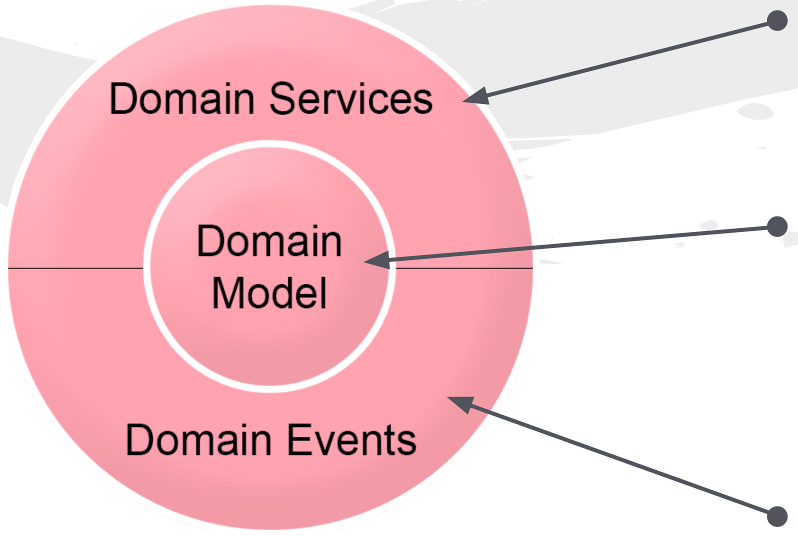
Actions / use cases

- Create outcome transaction
- Create income transaction

My blocks

- Entities
  - Transaction
  - Tag
- Repositories
  - TransactionRepository
  - TagRepository
- Events
  - TransactionCreated
  - TransactionFailed
- Domain Services
  - OutcomeTransaction
  - IncomeTransaction
- Application Service
  - CreateTransaction

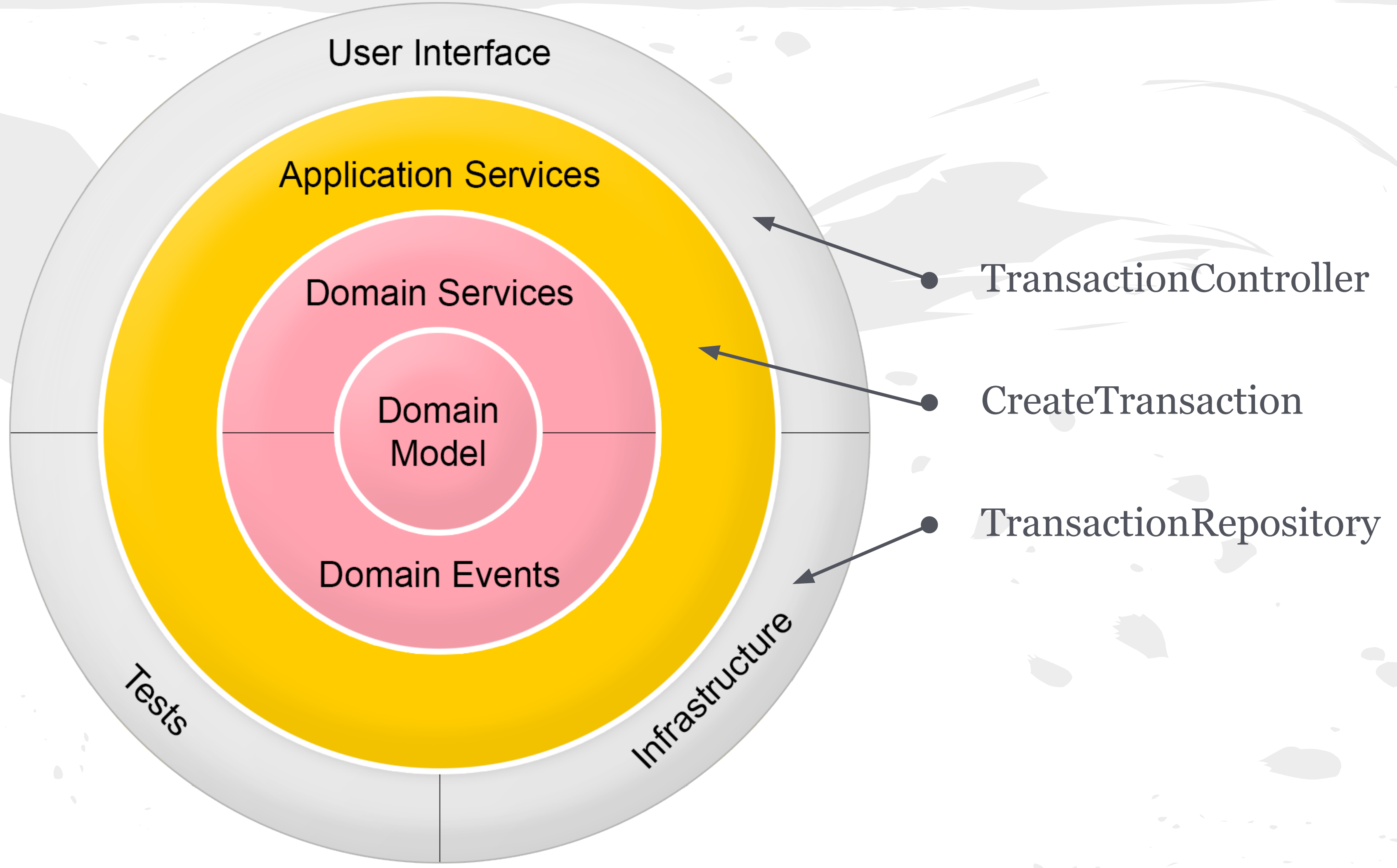
Core domain



CreateOutcome
CreateIncome

Transaction
TransactionRepository**Interface**
Tag
TagRepository**Interface**

TransactionCreated
TransactionFailed


User Interface
Application Services
Domain Services
Domain Model
Domain Events
Tests
Infrastructure
TransactionController
CreateTransaction
TransactionRepository

```
class TransactionController
{
    public function __construct(
        CreateTransaction $useCase,
        OutcomeTransaction $outcome
    ) {
        $this->useCase = $useCase;
        $this->outcome = $outcome;
    }

    public function createOutcomeAction()
    {
        $this->useCase->create($this->outcome->create(12.99));
    }
}
```

Controller

```
class CreateTransaction
{
   ...
   public function __construct(
        TransactionRepositoryInterface $repository,
        EventDispatcherInterface $dispatcher
   ) {
      $this->repository = $repository;
      $this->dispatcher = $dispatcher;
   }

   public function create(Transaction $transaction)
   {
      $this->repository->save($transaction);
      $event = new TransactionCreated($transaction);
      $this->dispatcher->dispatch($event::NAME, $event);
   }
}
```

Application Service

```
class OutcomeTransaction
{
    public function create($amount, Tag $tag, $description)
    {
        $transaction = new Transaction();
        $transaction->setCreated(new \DateTime());
        $transaction->setAmount($amount);
        $transaction->setTag($tag);
        $transaction->setType($transaction::OUTCOME);
        $transaction->setDescription($description);

        return $transaction;
    }
}
```

Domain Service

```
class TransactionRepository
    extends EntityRepository
    implements TransactionRepositoryInterface
{
    public function save(Transaction $transaction)
    {
        $this->_em->persist($transaction);
        $this->_em->flush();
    }
}
```

Repository

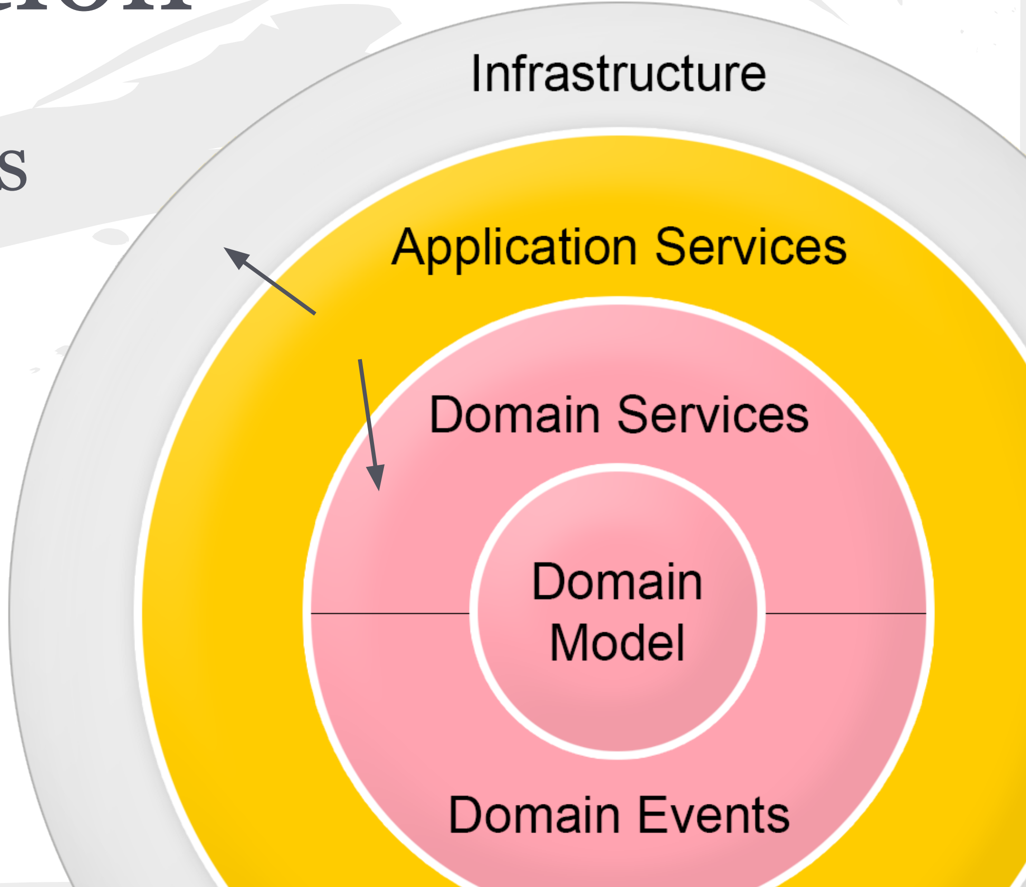
```
interface TransactionRepositoryInterface
{
    public function save(Transaction $transaction);
}
```

Repository Interface

Domain model is always in valid state.

Domain isolation

- Isolate via use-cases
- Infrastructure connected via clear interfaces



```php
<?php

interface TransactionRepository
{
    public function getTaggedBy(Tag $tag);

    public function getByTimeframe(
        DateTime $from,
        DateTime $to
    );
}
```

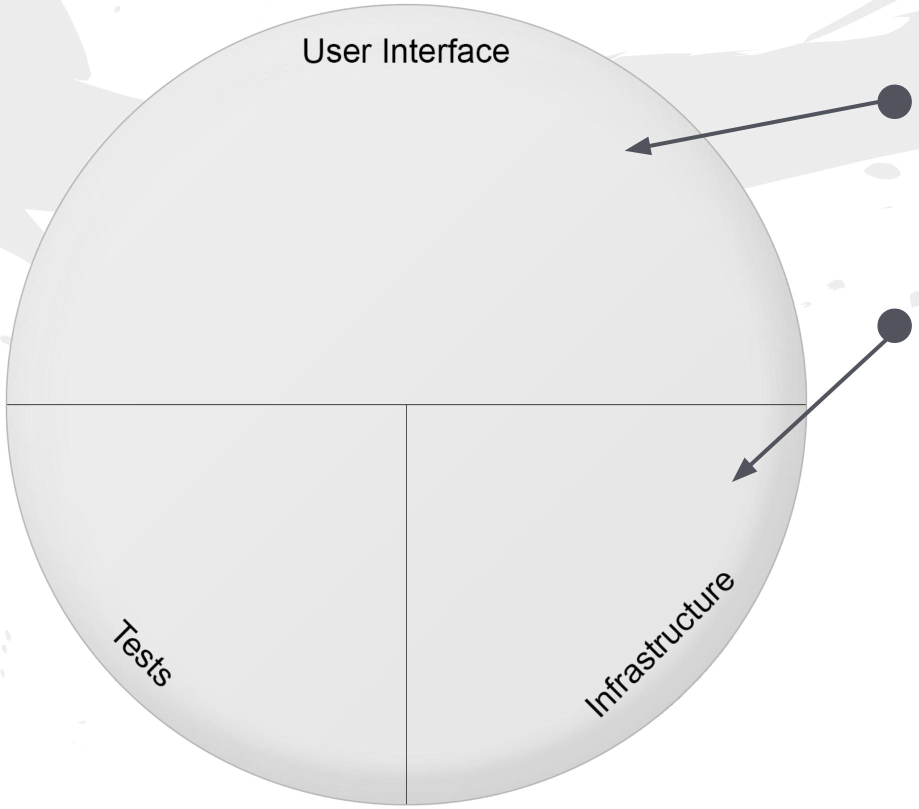
✓

```php
<?php

interface TransactionRepository
{
    public function getTagged($id);

    public function getByTimeframe(
        $from,
        $to
    );
}
```

X

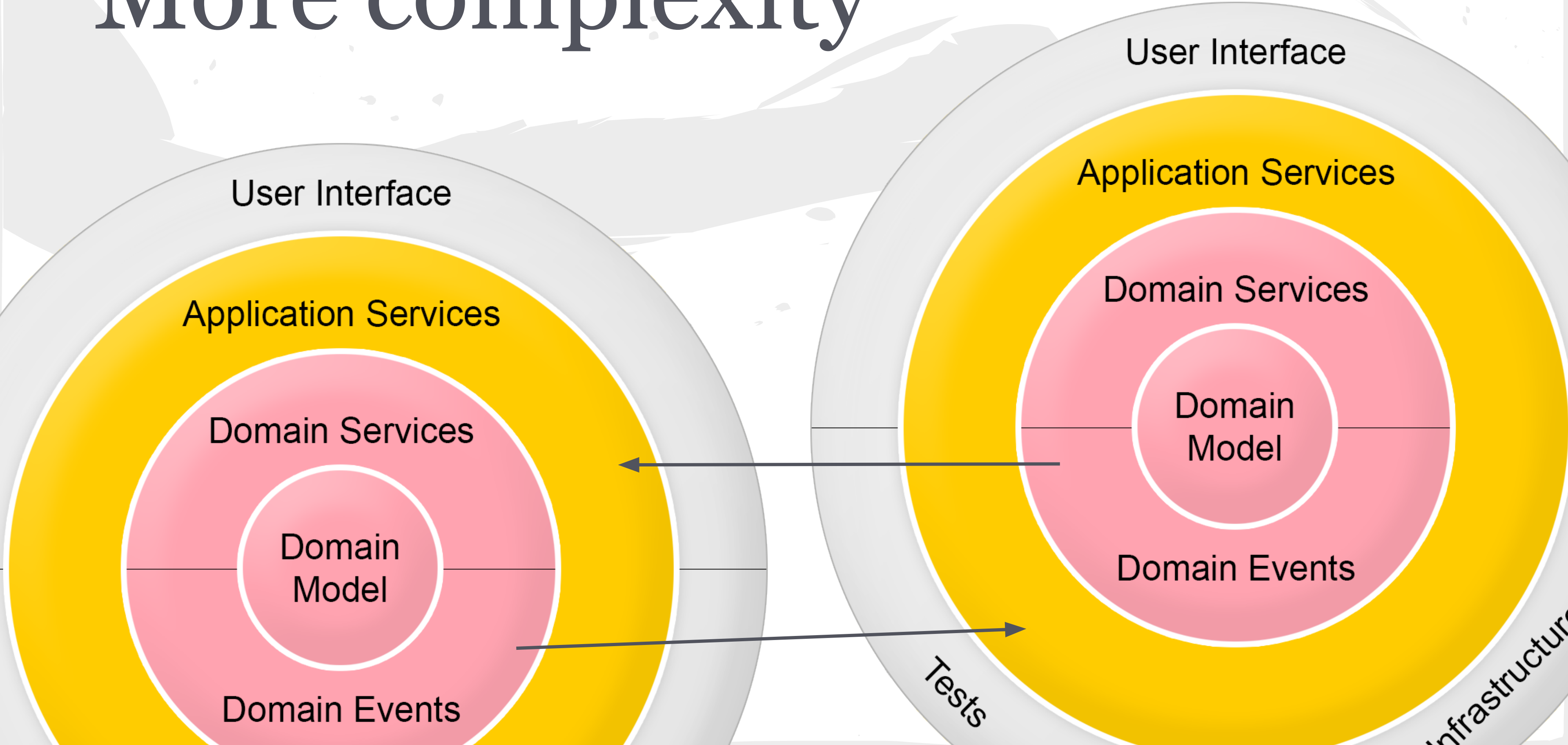
Intention-revealing interfaces

Implementation



- Web controller, CLI
- Persistence / DB
  File system
  Web services
  Email sending

More complexity



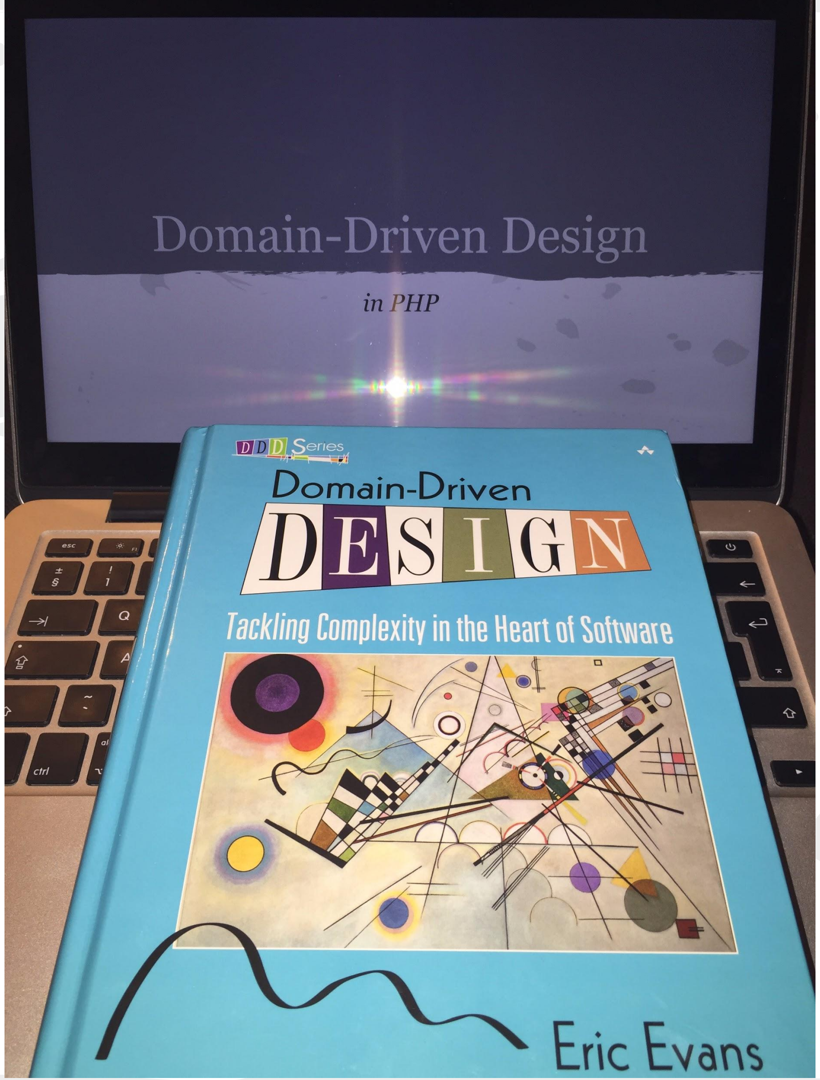
Summary

- No need to think about framework, external tools
- Focus on business problems
- Explore them
- Solve them
- Build long-lasting implementation

Reality check

- Every domain is unique
- Don't force design principles
- Explore domain and how real things get done
- Development is iterative

The big blue book about DDD by Eric Evans



Ačiū!

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.“

Martin Fowler