Implementing DDD with C#

Pascal Laurin

May 2015

Microsoft C# MVP
MSDEVMTL user group organizer
Developer & Architect at GSoft

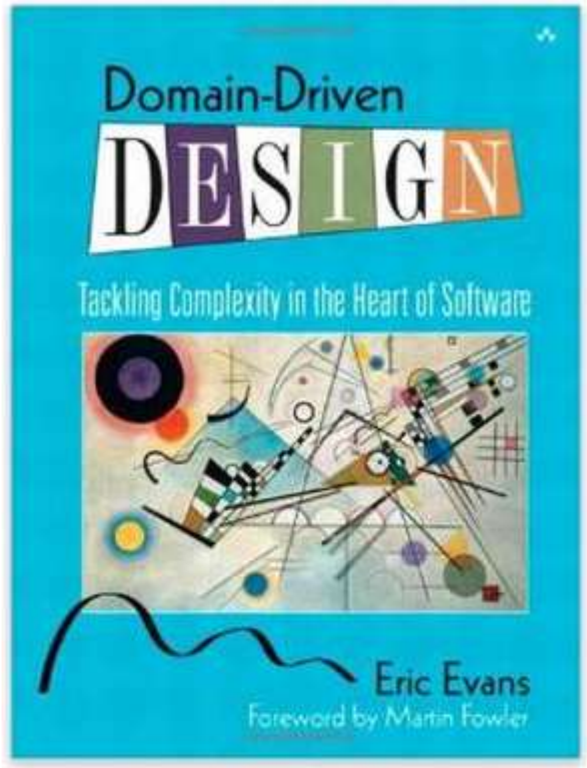
@plaurin78
pascal.laurin@outlook.com
www.pascallaurin.com

Agenda

- DDD Basics
- Overall Architecture
- Some Design Patterns to help
- Questions

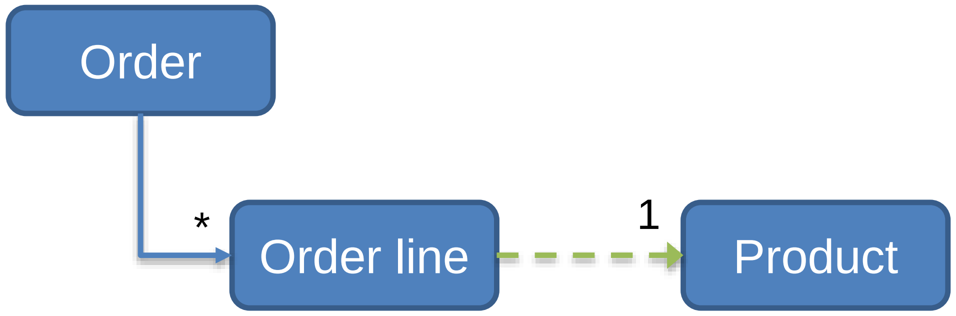
Domain Driven Design Basics

- Model-Driven Design and the Ubiquitous Language
- Entities, Value Objects and Services
- Aggregates
- Factories
- Repositories
- Bounded Contexts and Context Map
- Anticorruption Layers


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Model-Driven Design and the Ubiquitous Language

- Technology should have nothing to do with domain modeling of a system
- Put the domain at the center of the solution
- Should use the domain language even in code (at the lowest level: variable names)
- Only one word per concept!
- Use ULM if you like but simple diagrams should work too
- User stories, use cases, etc.



Entities, Value Objects and Services

- Entities have identities
- Value Objects don’t have identities (mostly immutable)
- Use services only with complex operations requiring multiples domain entities
- Some team use services with anemic entities (better fit for functional languages?)

```
public class Product // Entity
{
   public int Id { get; private set; }
   public string Description { get; private set; }
}

public class Money // Value Object
{
    public decimal Amount { get; private set; }
    public Currency Currency { get; private set; }
}
```

Aggregates

- Boundary around objects inside (can't access objects directly, must go through the Root)
- Enforce invariants of the Entities inside
- Root Entity has global identity
- Internal Entities have local identities
- Contains no direct references to other Aggregates, only IDs

```
public class Product // Aggregate root
{
    public int Id { get; private set; }
    public int CatalogId { get; private set; }
    public Money Price { get; private set; }

    public void ChangePrice(Money newPrice)
    {
    }
}
```

GSoft

Factories

- For creation and initialization of complex objects structures
- Respect invariants
- Creation as an atomic unit (complete or fail, no partial)
- Creates entire Aggregates
- Can use simple constructor on Aggregate Root Entity if construction is simple enough

```
public class ProductFactory
{
    public Product CreatePhysicalProduct(string name, Weight weight,
        Dimensions dimensions)
    {
        return new Product();
    }
    public Product CreateDigitalProduct(string name, StorageSize storageSize)
    {
        return new DigitalProduct();
    }
}
```

Repositories

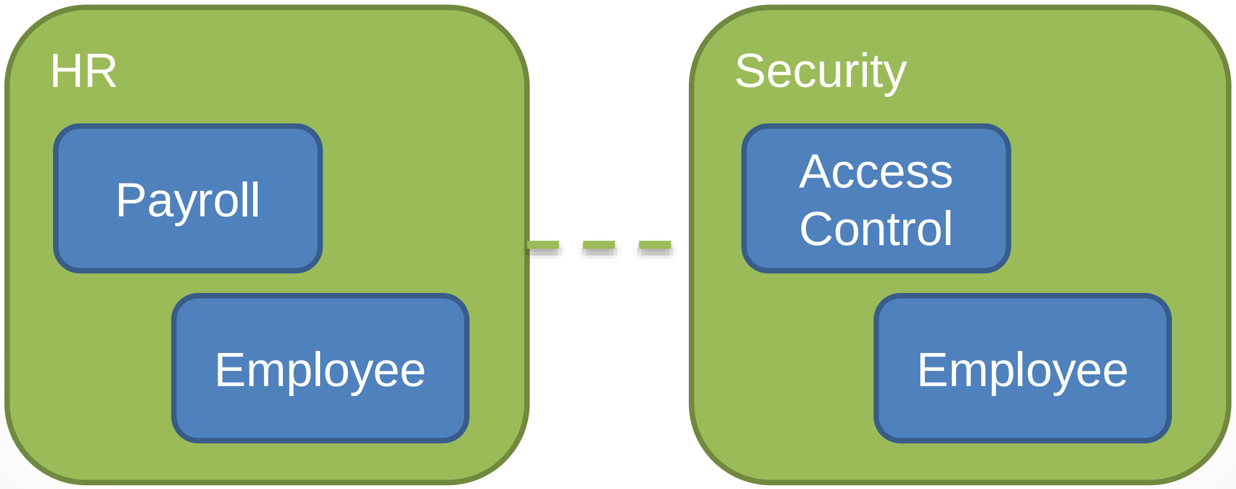
- Don’t use generic Repository<T>!
- Don’t leak DAL concerns in the domain
- Most DAL technology today already offer a generic abstraction
- Mostly for mocking with unit testing

```
public interface IProductRepository
{
    Product LoadProductAggregateById(int id);
    void AddNewProduct(Product product);

    IEnumerable<Product> QueryProductsByCatalog(string catalogName);
    IEnumerable<Product> QueryArchivedProducts(string catalogName);
    IEnumerable<Product> QueryDiscountedProducts(string nameFilter,
        DateTime? discountExpirationDateFilter, Money maxPriceFilter);

    ~~IQueryable<Product> GetById(int id);~~
    ~~IQueryable<Product> GetAll();~~
}
```

GSoft

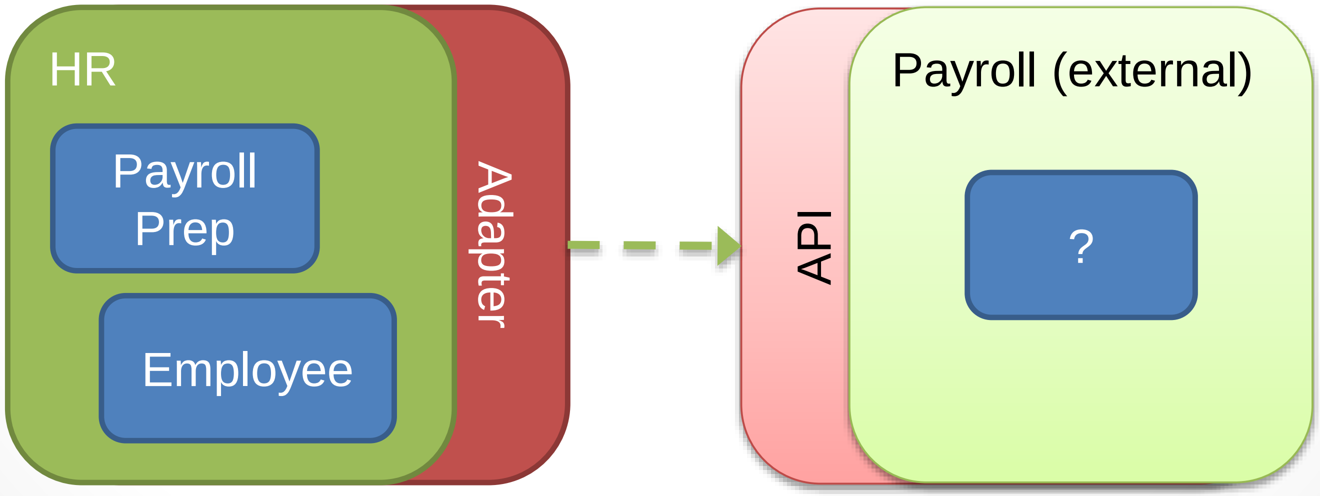
Bounded Contexts and Context Map

- Split large domains into smaller ones
- Especially if two vision of the same domain concept dependent of the view point
  - Usually along departments or divisions lines, business units, etc.
- Could be still be monolithic apps or separated apps



Anticorruption Layers

- Don’t pollute your domain with foreign concepts
- Abstract external providers, partners, etc.
- Translate between two different domains (Bounded Context) using Adapters
- Allow both to evolve independently

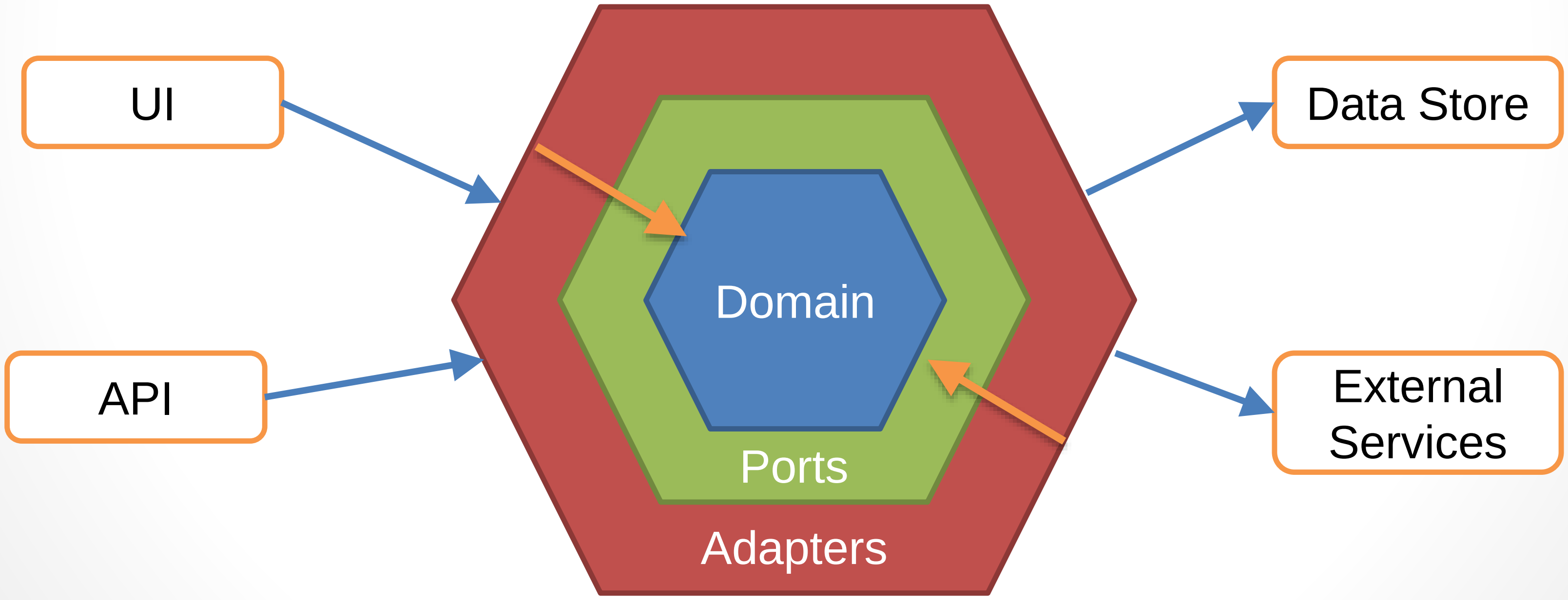


The overall architecture

- Hexagonal architecture or Port and Adapter
- Domain at the center
- Demo solution

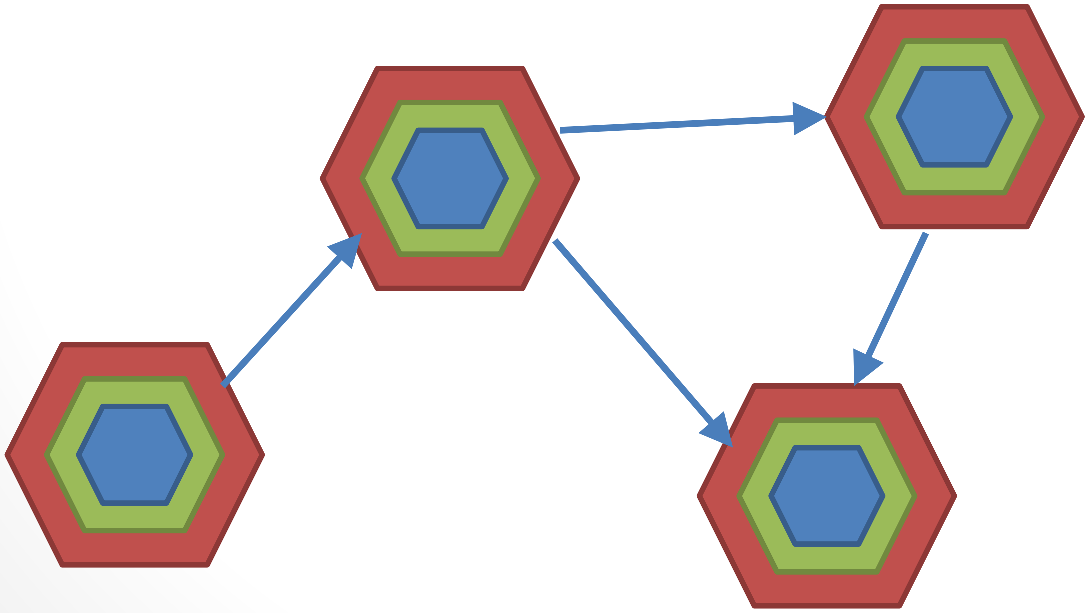
Hexagonal architecture or Port and Adapter

- Ports are API or contracts in and out of the domain
- Adapters translate between Ports and external dependencies
  - Swap out external dependencies implementation using different adapters or using mocks



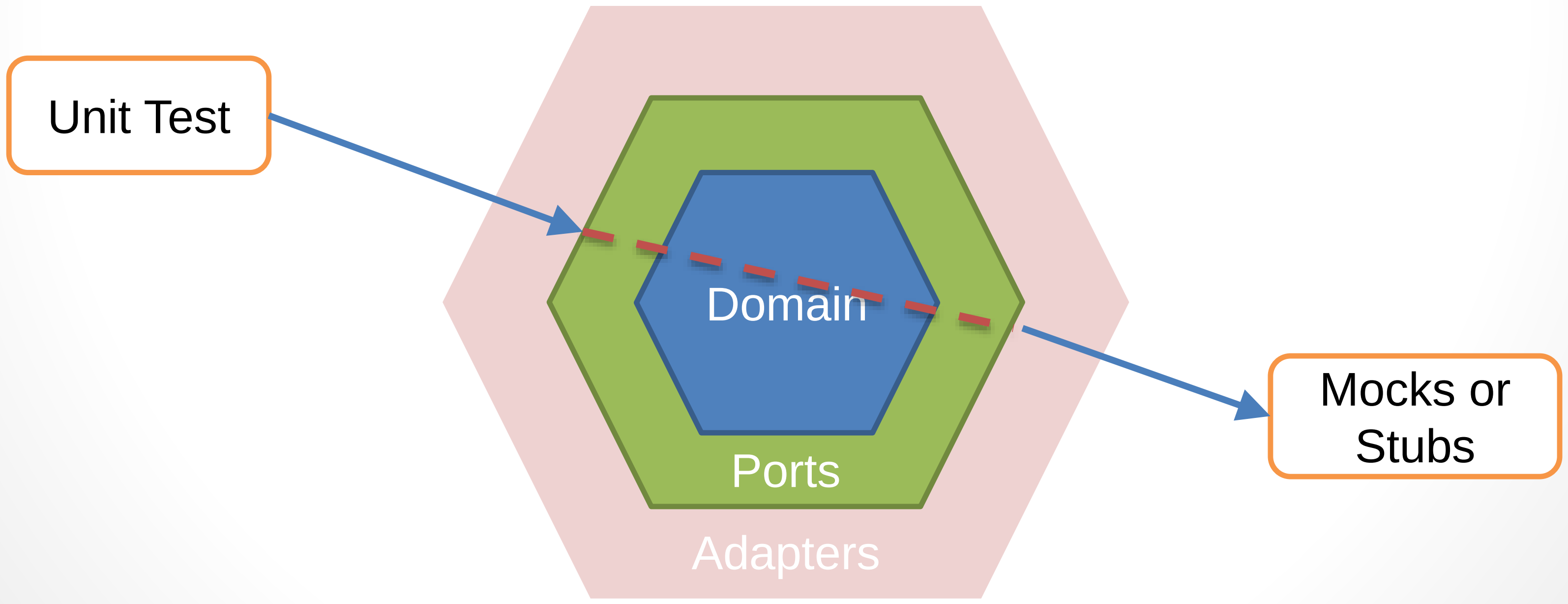
The Domain at the center of everything

- Push everything to the sides and concentrate on the middle
- For big app components => Hexagonal architecture to split into smaller chunk
  - Either monolithic app or micro-services



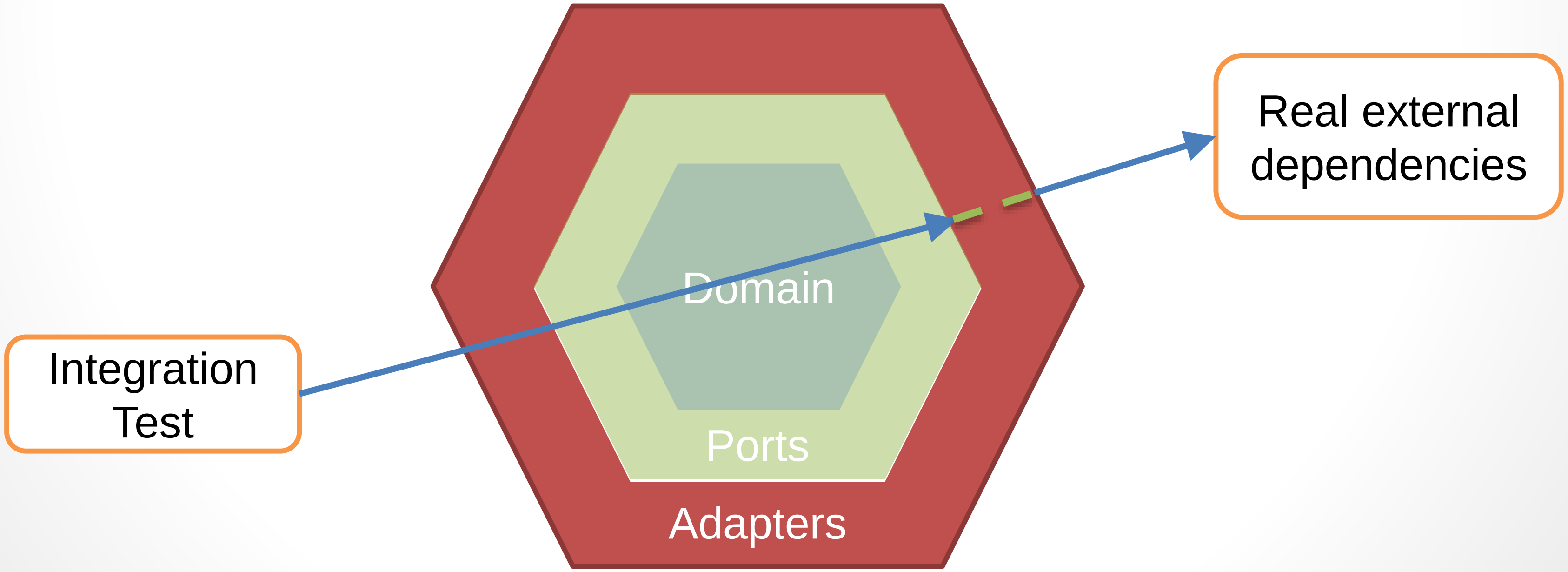
Unit Tests

- Only testing the Domain
- Testing at the Ports entering the Domain
- Use mocks or stubs to replace the Adapters



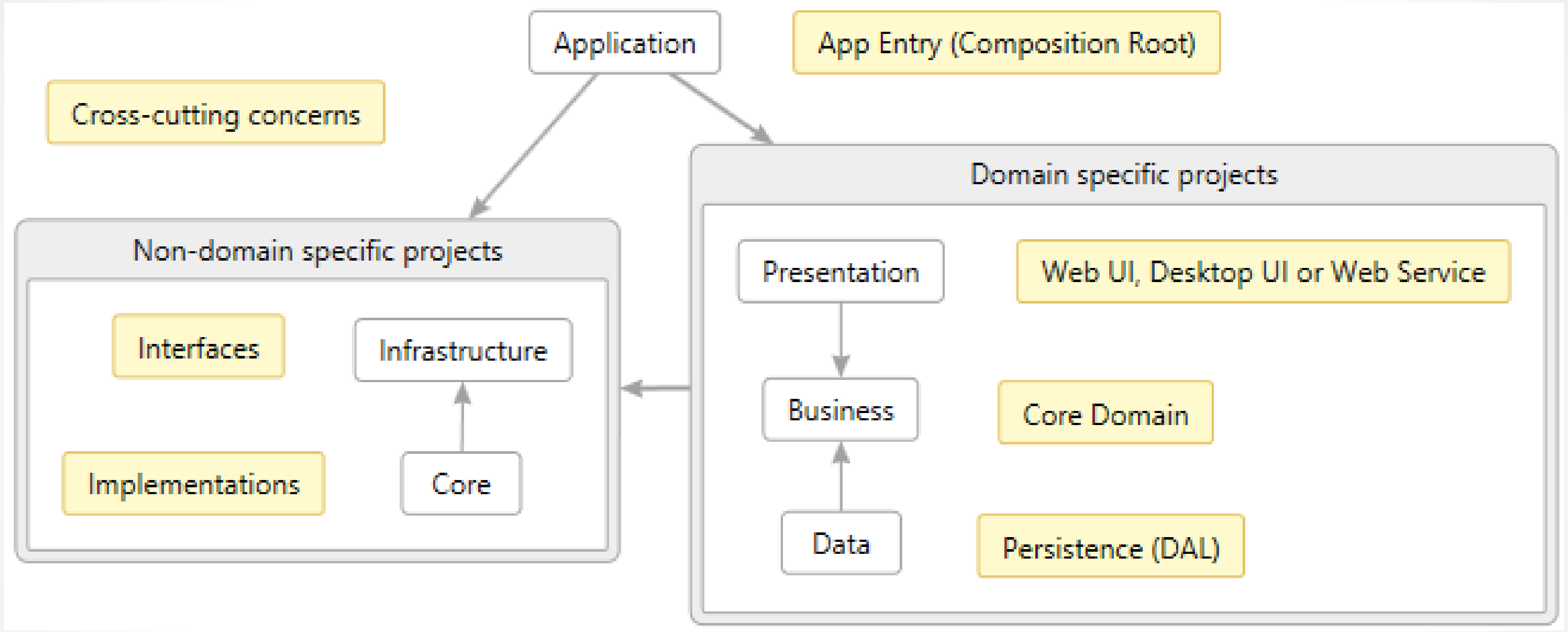
Integration Tests

- Only testing the Adapters
- Testing at the Ports exiting the Domain
- Adapters calling real external dependencies



GSoft

Demo Solution

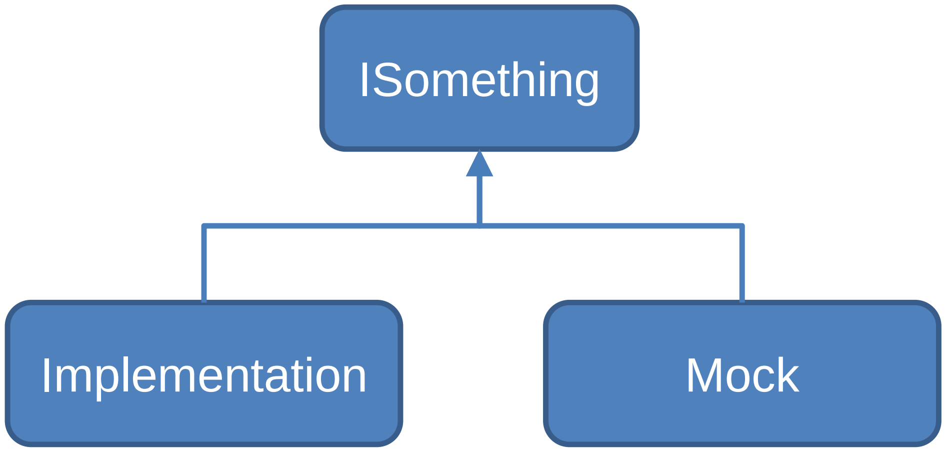


Design Patterns (and Architectural Patterns)

- Interfaces and abstractions
- Dependency injection
- Bootstrapper
- MVC, MVVM, MVP (UI)
- Command & Query responsibility segregation

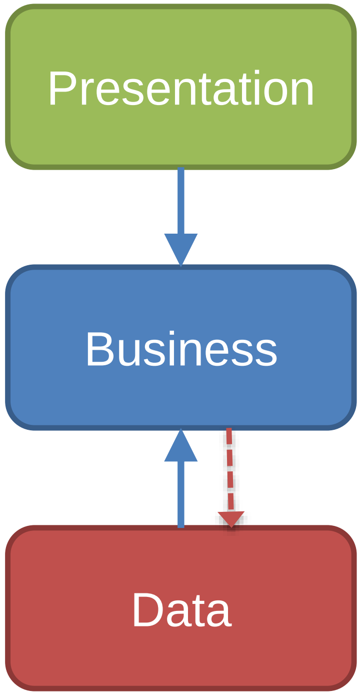
Interfaces and abstractions

- Most useful to mock dependencies
- Swap implementations by configuration
- Do not overuse!



Dependency injection

- Inversion of Control
- Domain should not depend on DAL, Web Services, IO, etc.
- Construction, composition and life cycle of non-domain concerns stay outside the domain (use Factories for domain objects)



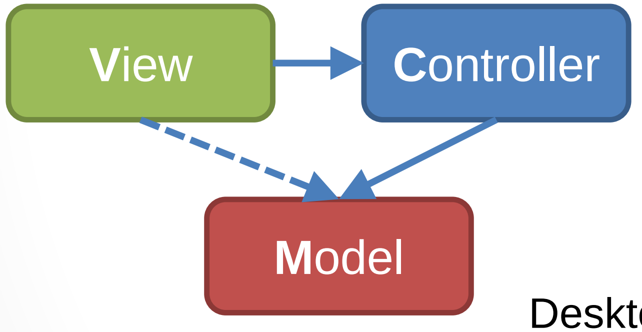
Bootstrapper

- Application startup code
- Composition of the application, services and infrastructure code
- Load configuration and setup
- Should help write integration tests by replacing some external dependencies in the tests

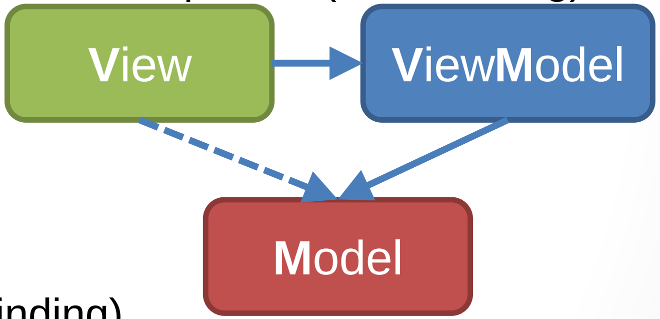
MVC, MVVM, MVP (UI)

- Presentation patterns
- Mostly useful when unit testing and reuse if multiple platforms targeting (mobile)
- Model and View always separated

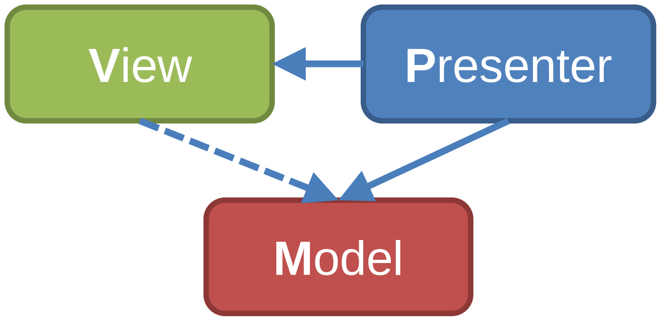
Web site, web service, API



Desktop, SPA (data binding)

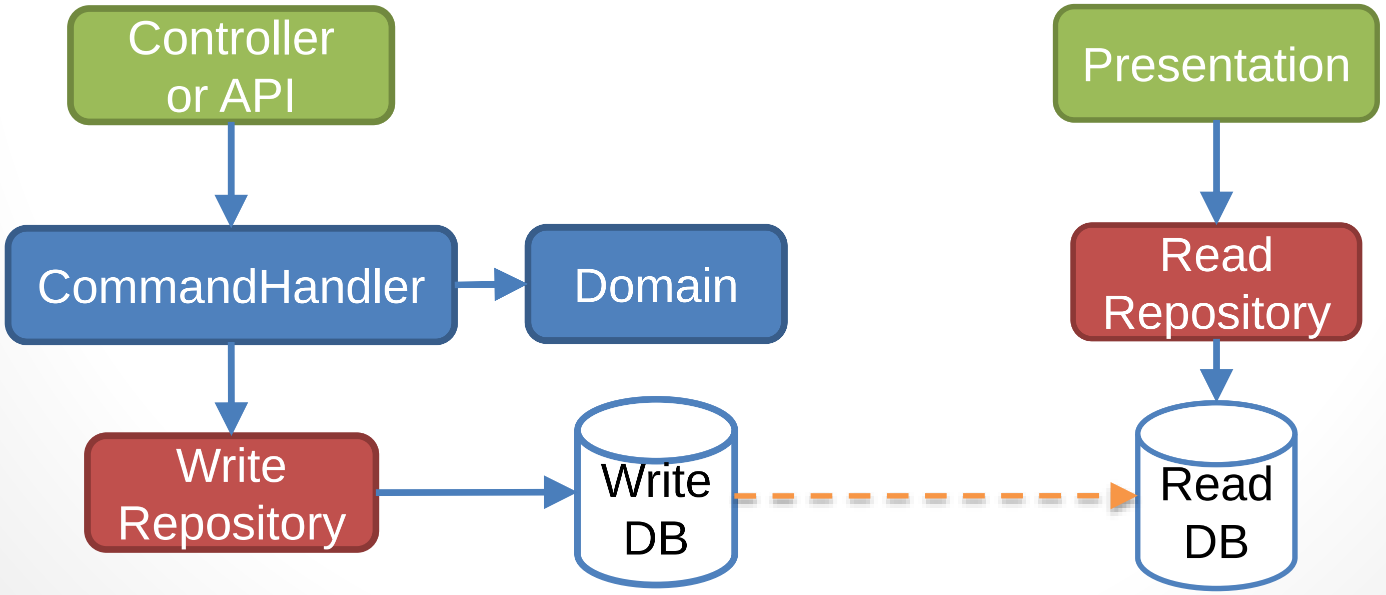


Desktop (no data binding)



Command & Query Responsibility Segregation

- Queries are simple and have no side effect
- All changes to entities go through commands
- With CommandHandler to execute Command
- Could still be using the same data store for both Command and Query
- Command Dispatcher



Conclusion

- Domain Driven Development improve the quality of the code by
  - Introducing useful design patterns to structure your application
  - Knowing where each piece of new code should go
  - Better communication by using the language of the domain in the team
  - Clear separation of business and non-business code

Questions?

- References
  - DDD book by Eric Evans
    - http://www.amazon.ca/dp/0321125215
  - DDD Quickly
    - http://www.infoq.com/minibooks/domain-driven-design-quickly
  - SlideShare of the presentation
    - http://www.slideshare.net/PascalLaurin
  - BitBucket for the code
    - https://bitbucket.org/pascallaurin/ddd-talk

@plaurin78
pascal.laurin@outlook.com
www.pascallaurin.com