DOMAIN DRIVEN DESIGN

Domain Driven Design basic concepts and best practises

DDD, some basic terminology

Domain: A Domain, in the broad sense, is what an organization does and the world it does it in. A business domain such as Project Management

Subdomain: A Domain can be broken into Subdomains e.g. Change Request Management

Domain Model: It's a software model of the very specific business domain you are working in

Ubiquitous Language: a shared language developed by the team - a team composed of both domain experts (e.g. project managers) and software developers.

Bounded Context: an explicit boundary within which a domain model exists. Inside the boundary all terms and phrases of the Ubiquitous Language have specific meaning, and the model reflects the Language with exactness.

DDD, some basic terminology

There is one Ubiquitous Language per Bounded Context!

What does this mean and why is it important?

Think of the term “Account” and the following domains: Banking, User Management, Customer Management

Banking: The term “Account” means your bank account and has to do with money

User Management: The term account is your user account and has to do with passwords, roles, permissions etc.

Customer Management: The term account has to do with your personal info, past purchases etc.

DDD, some basic terminology

This may sound a bit theoretical but in big, complex projects a clear definition of terms, their business meaning and their context is extremely important.

If you refer to an item with a name that does not make sense then…

- How will you communicate about it with Business Analysts / Users?
- How will you communicate with fellow developers who may be unaware of the naming conventions?
- How will you train new developers?

DDD, domains, subdomains & practical application

So let's say we have a Project Management Domain.
The core Domain we need to focus on is Project Management.
Some subdomains will also be identified, they may be business-driven or they can also be tech-driven

- Project Management Domain
    - Resource Management
    - Risk Management
    - User Management (technical – roles, permissions etc)
    - Change Request Management

So in simple terms what we will have is a "Project Management Application" with 5 Projects (Modules/DLLs)

A subdomain will typically correspond to a separate Module/Project/DLL in your code.

DDD – Key Points

- DDD isn’t first and foremost about technology. DDD is about discussion , listening, understanding, discovery, and business value.

- Domain experts (e.g. Business Analysts, Project Managers) & Developers working closely together

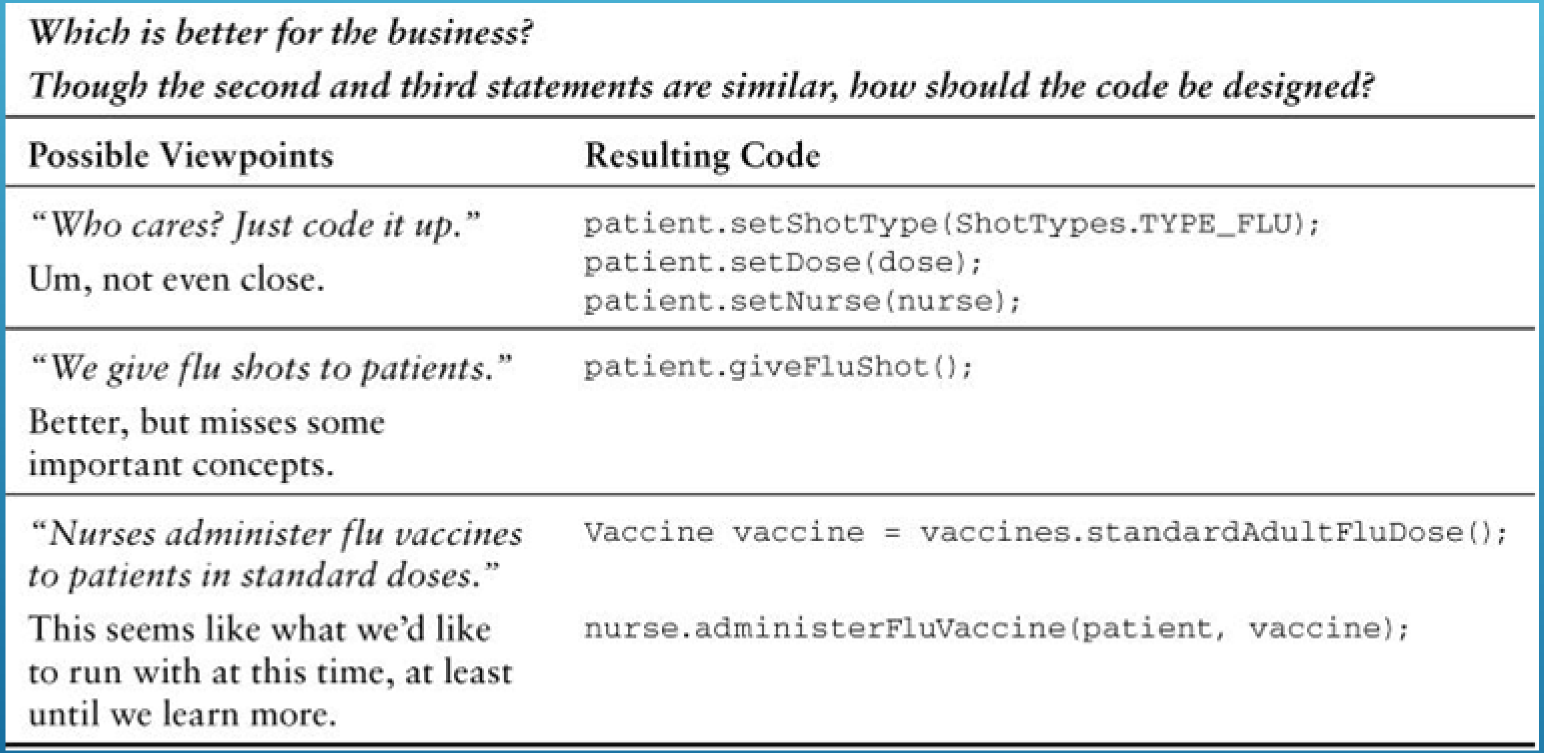
- Developers need to understand the core Domain for which they will write software and any supporting subdomains

- A common language needs to be agreed that both Domain experts and developers can understand and use throughout the Project (Ubiquitous Language)

- Use DDD to Simplify, Not to Complicate. Use DDD to model a complex domain in the simplest possible way.

DDD – What DDD code should look like

Code is at its best when it closely resembles natural language and what actually goes on in the real world.



Which is better for the business?

Though the second and third statements are similar, how should the code be designed?

Possible Viewpoints	Resulting Code

"Who cares? Just code it up."
Um, not even close.
	`patient.setShotType(ShotTypes.TYPE_FLU);`
	`patient.setDose(dose);`
	`patient.setNurse(nurse);`

"We give flu shots to patients." Better, but misses some important concepts.	`patient.giveFluShot();`

"Nurses administer flu vaccines to patients in standard doses."
This seems like what we'd like to run with at this time, at least until we learn more.
	`Vaccine vaccine = vaccines.standardAdultFluDose();`
	
	`nurse.administerFluVaccine(patient, vaccine);`

DDD – Architecture

"Architecture should speak of its time and place, but yearn for timelessness."

Avoiding architectural style and pattern overuse is just as important as using the right ones. Thus, **we must be able to justify every architectural influence in use, or we eliminate it from our system.**

Some basic architectural "styles"
- Traditional Layers
- Layers with Inversion Principle
- Ports & Adapters
- Event-Driven (not covered here)

DDD – Architecture – Traditional Layers

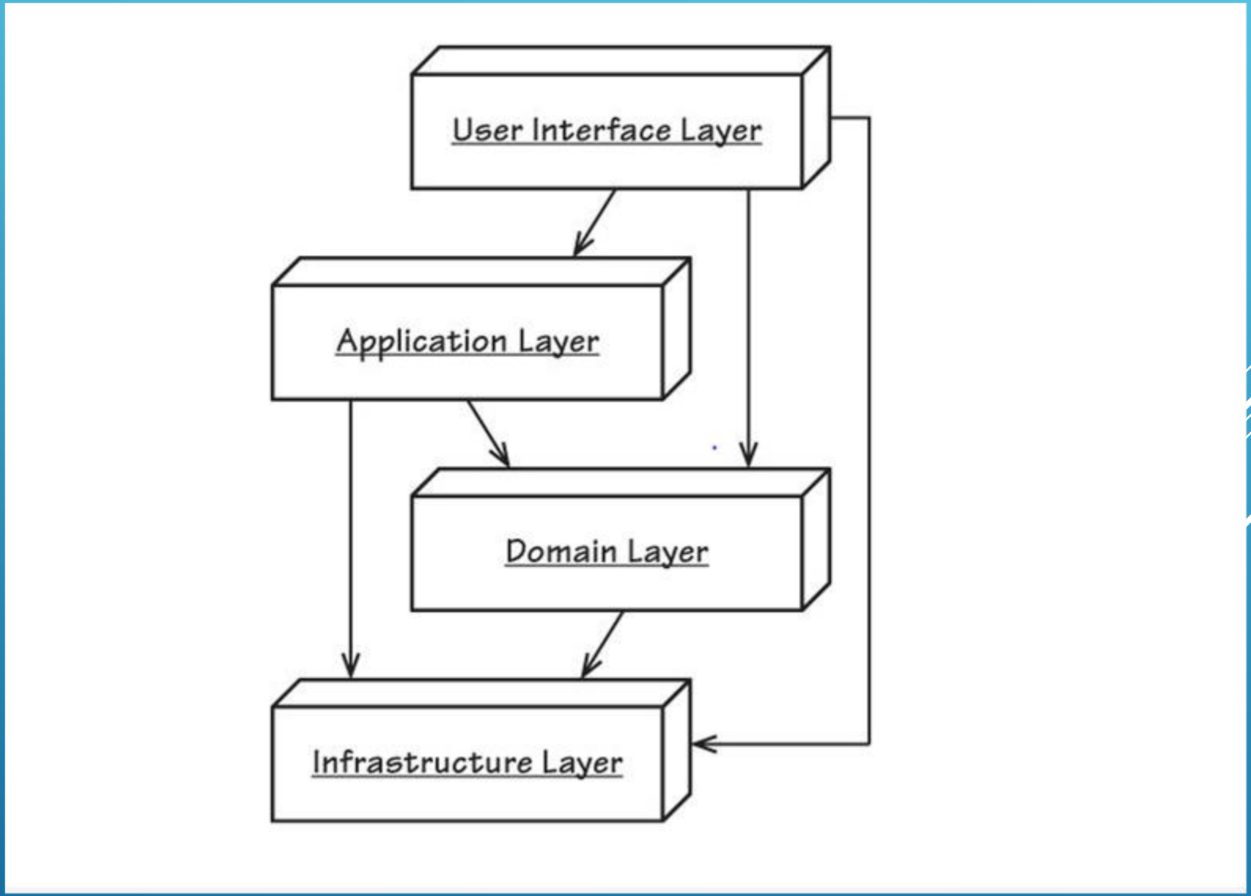
Develop a design within each layer that is cohesive and that depends only on the layers below.

So you have a ProjectUI dll

that references a ProjectApp dll

that references a ProjectDomain dll

that references a ProjectInfrastructure dll

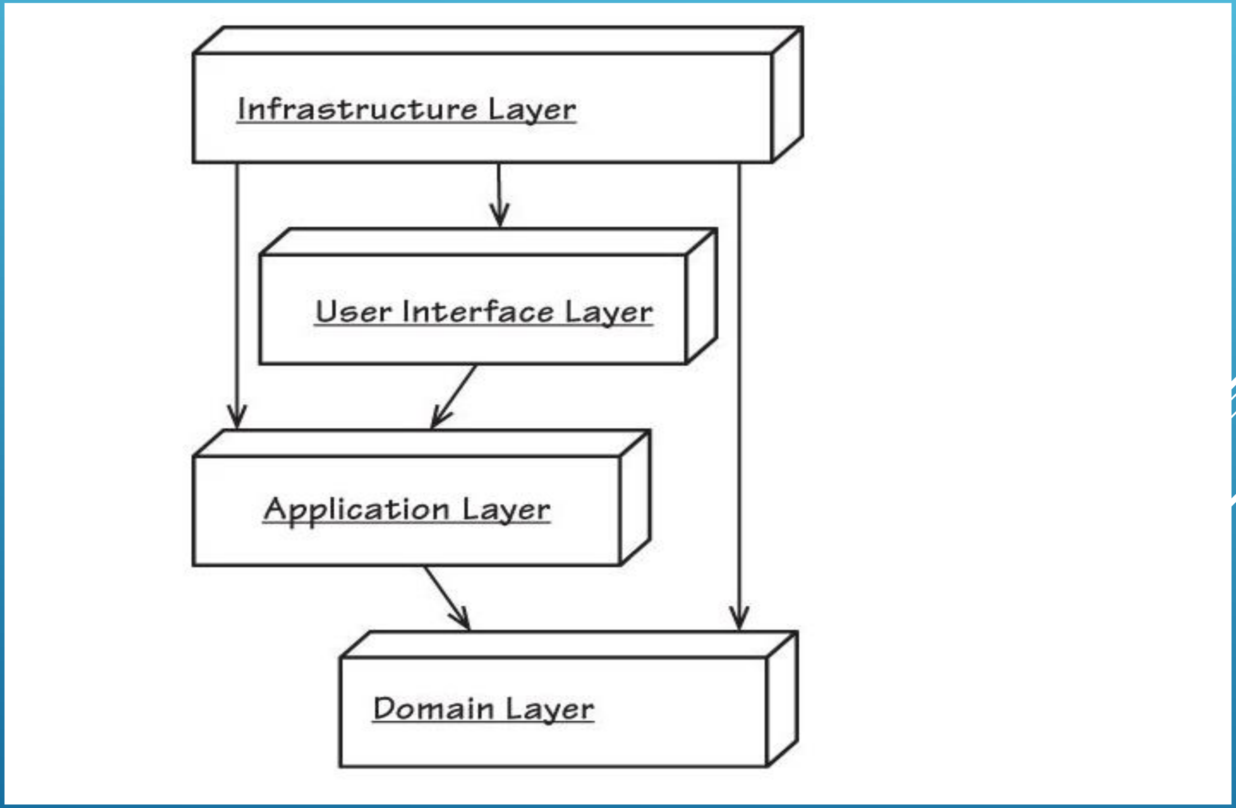


DDD – Architecture – Layers with Dependency Inversion

a component that provides low-level services (Infrastructure, for this discussion) should depend on interfaces defined by high-level components (for this discussion, User Interface, Application , and Domain).

There are different ways to achieve this, using a separate project that contains Interfaces can be one way of doing it.

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.

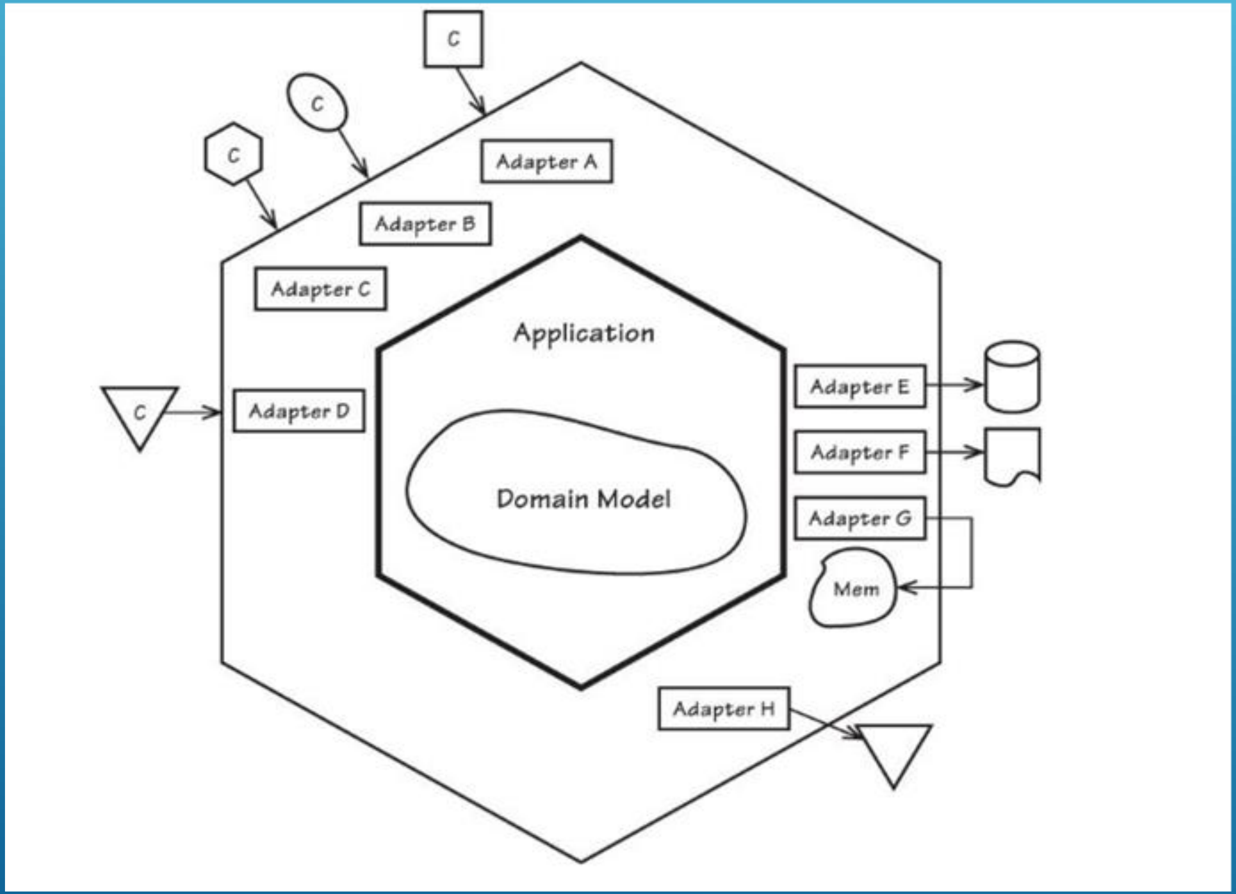


DDD – Architecture – What goes where

- **User Interface**
  - Views, aspx pages, controllers, code behinds, html helpers etc.
- **Application Layer**
  - Application Services providing transaction & security management, application level event handling and coordination
- **Domain Layer**
  - All core business logic, aggregates/aggregate roots
- **Infrastructure**
  - DAOs, Repositories, Import/Export, Various Helpers etc.

DDD – Architecture – Hexagonal Ports & Adapters

Need a new client? Not a problem. **Just add an Adapter to transform any given client's input into that understood by the internal application's API.**



DDD – Architecture – Hexagonal Ports & Adapters

Need a new client? Not a problem. **Just add an Adapter to transform any given client's input into that understood by the internal application's API.**

The application receives requests by way of its public API.

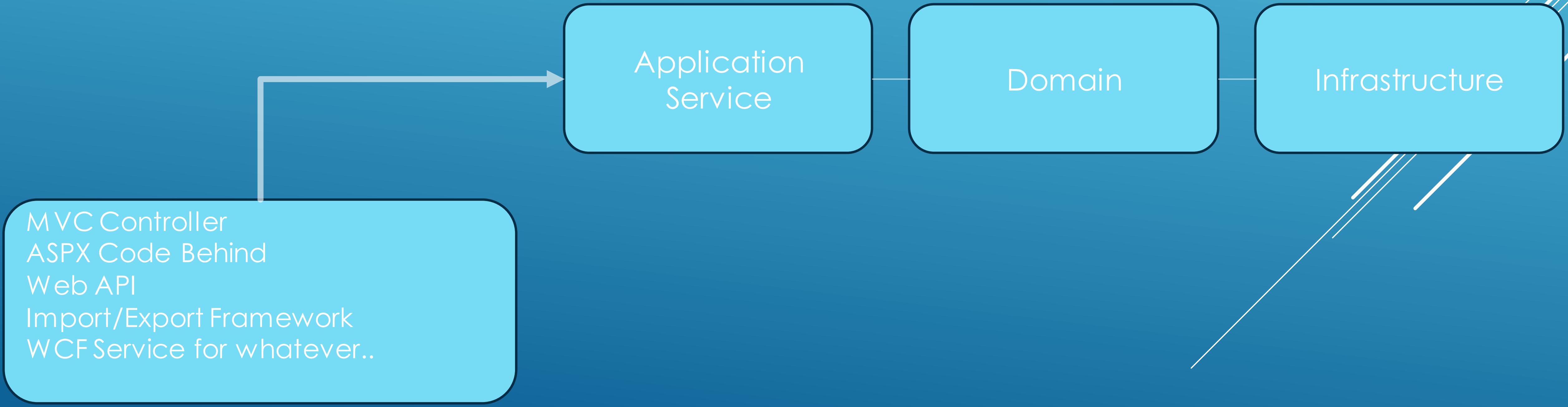
When the application receives a request via its API, it uses the domain model to fulfill all requests involving the execution of business logic. Thus, the application's API is published as a set of Application Services. Here again, Application Services are the direct client of the domain model, just as when using Layers.

So what does this mean for us?

DDD – Architecture

What are some of the potential clients a web application can have?

- Import/Export
- Mobile App
- Web Api
- The UI itself can be considered simply another client
- and many more!!



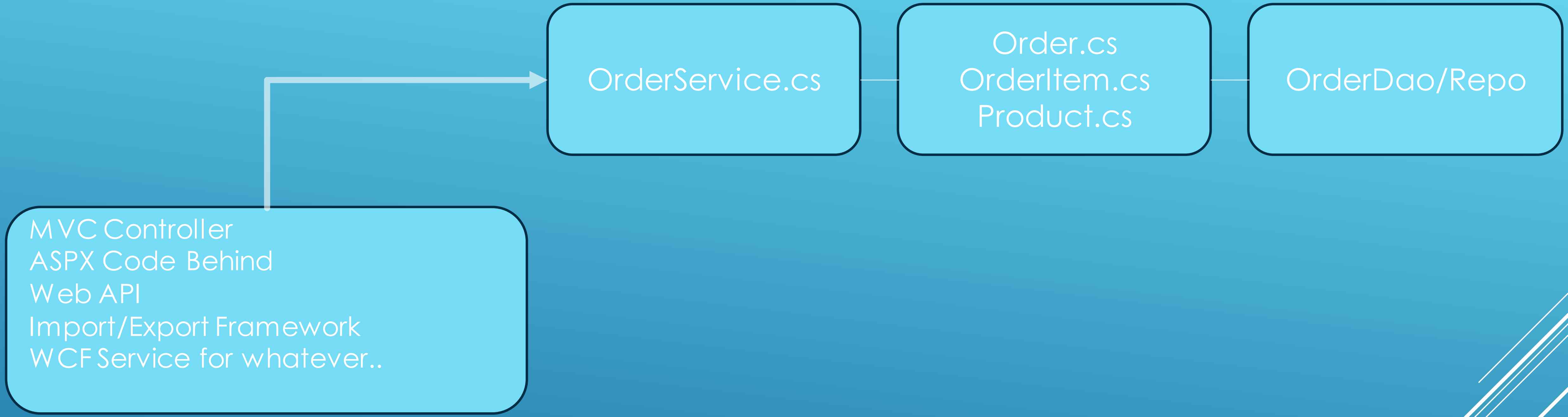
DDD – Architecture

No business logic in Code Behinds, Controllers, ASPX pages, Views.

Controllers and Code Behinds should use something like IOrderService and then use this class to perform any operations

```
protected void btnProcessOrderClick(object sender, EventArgs e)
{
    .....
    _orderService.Process(order);
}


[HttpPost]
public void Process(Order order)
{
    ......
     _orderService.Process(order);
}
```


OrderService.cs
Order.cs
OrderItem.cs
Product.cs
OrderDao/Repo
MVC Controller
ASPX Code Behind
Web API
Import/Export Framework
WCF Service for whatever..

DDD – Basic Concepts

Entities
The key characteristics of an Entity is that it has a unique identity and it is mutable.

2 entities with the same ID are equal. An entity can be modified during the execution of code. Examples of entities are users, products, orders etc. Typically they are the classes in your system corresponding to real world concepts.

Identity Stability
Trivial measures may be taken to prevent identity modification. We can hide identity setters from clients. We might also create guards in setters to prevent even the Entity itself from changing the state of the identity if it already exists.

Value Objects
They don't need a unique identity and they are immutable. They can be represented as Structs. You don't need to modify their state, you can simply destroy them and create new ones.

Example of a value object
A location with longitude, latitude properties. It does not need an identity, it can be a value type.

DDD – Entities vs Value Objects

"Value types that measure, quantify, or describe things are easier to create, test, use, optimize, and maintain. It may surprise you to learn that we should strive to model using Value Objects instead of Entities wherever possible."

You should try to use Value objects when possible, they are easy to create, lightweight and since their state does not change they can be less complex to work with.

Two value objects are considered equal when their type and properties are equal.
Two entities are considered equal when their type and IDs are equal.

There are a variety of ways to persist Value Object instances to a persistent store. In a general sense it involves serializing the object to some text or binary format and saving it to disk.

Sometimes you will just have to use an ID for your value objects in the database but this can be handled with some form of mapping. You should not design your Domain model based on persistence or I/O concerns.

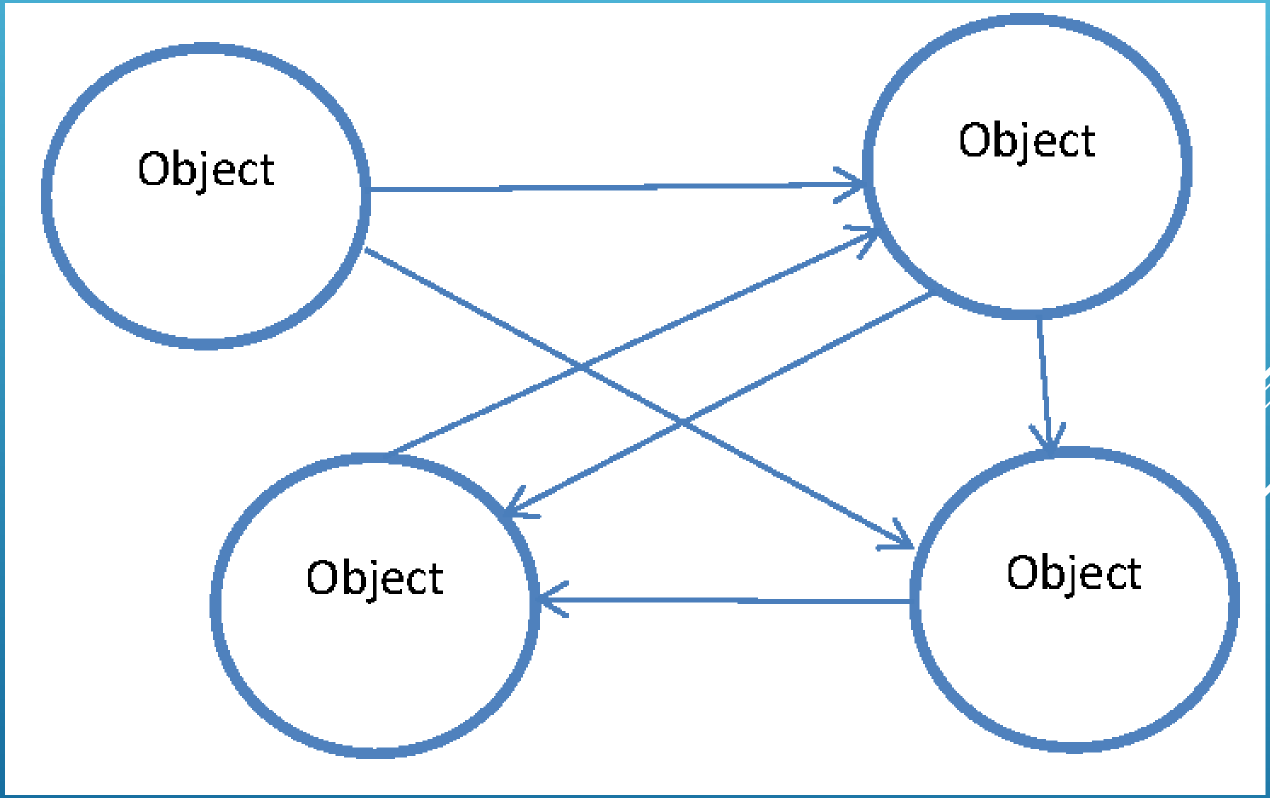
DDD – Aggregates

"Clustering Entities and Value Objects into an Aggregate with a carefully crafted consistency boundary may at first seem like quick work, but among all DDD tactical guidance, this pattern is one of the least well understood."

What is the problem?

Complexity a lot of the time has to do not with the code within a single class but with all the different ways that objects can interact with each other

Chaotic even with just 4 objects

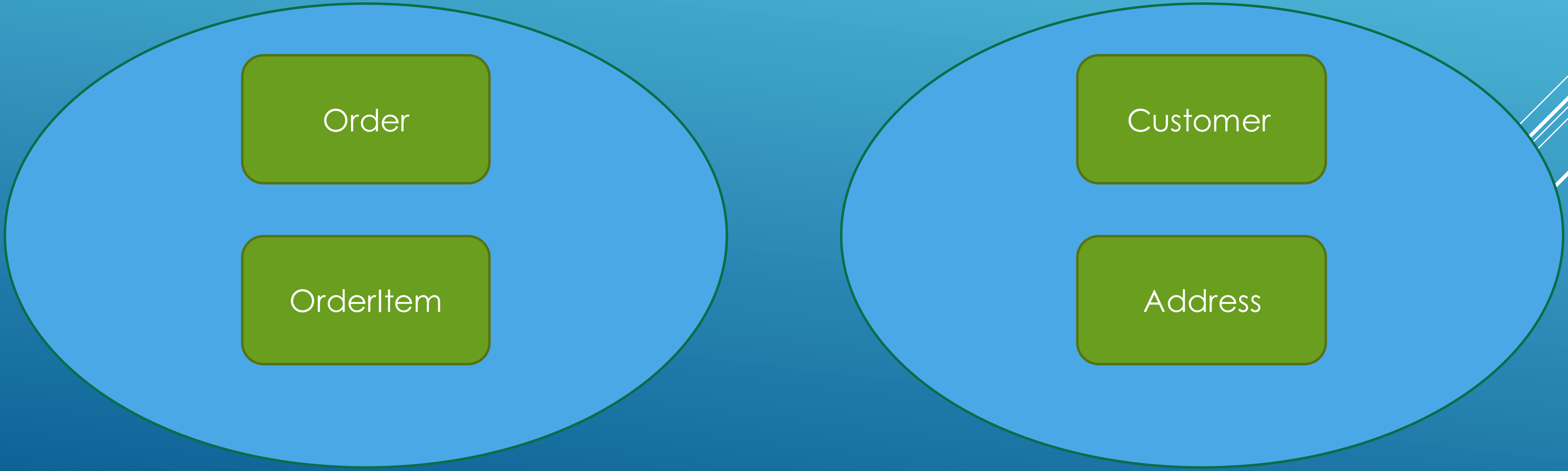


DDD – Aggregates

Aggregate is synonymous with transactional consistency boundary.

An invariant is a business rule that must always be consistent.

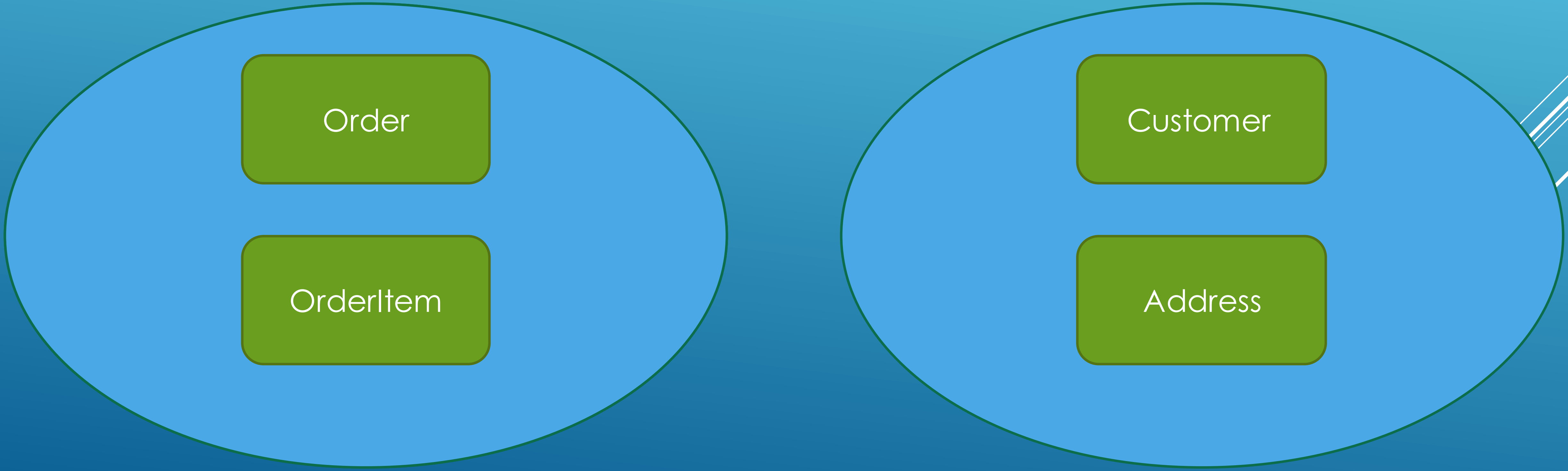
An Aggregate is a cluster of objects that should be modified as part of the same transaction.



DDD – Aggregates

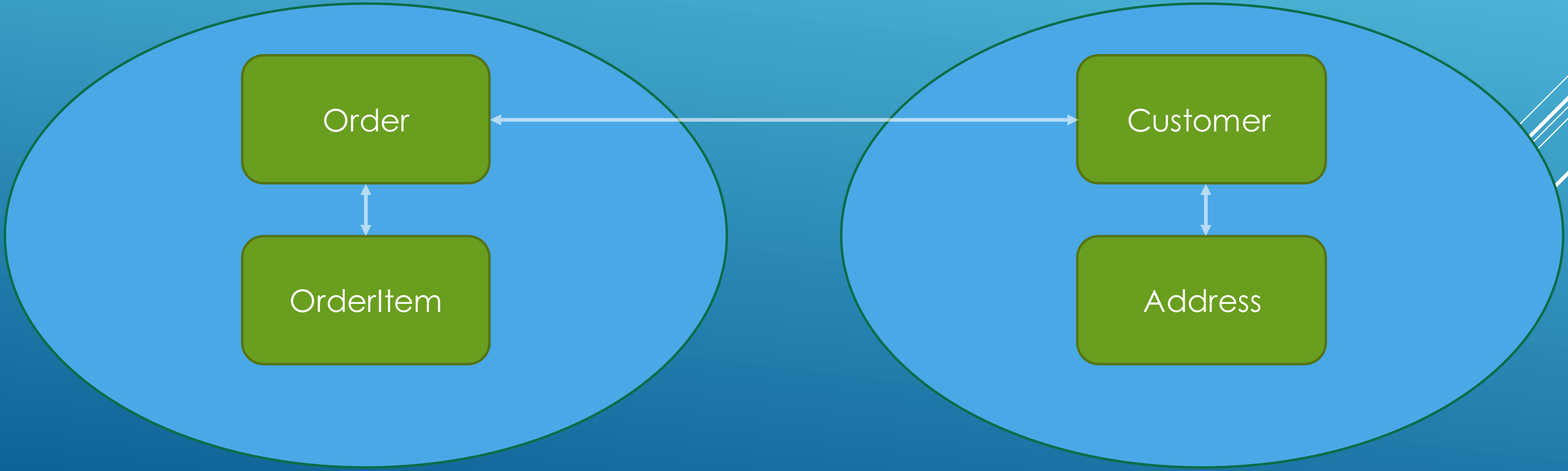
Can I have an order without items? Can the items be added later? If the answer is No then I need an aggregate because Order and OrderItems need to be updated together in the same transaction.

Do I need an aggregate for Customer/Address?



DDD – Aggregates

In general terms, if you think of the aggregate root as the root of a tree, the roots interact only with other roots or with their own children. OrderItems do not need to know about Customer, and the Address does not need to know anything about the Order. If they need to interact with other aggregates they should go through the Root Class.



DDD – Aggregates

- Choose a class as the Root Aggregate
- There should be one Repository per aggregate
- The root aggregate is responsible for handling persistence and validation
- The aggregate should not directly use Repositories or DAOs

The repositories should be the only way the rest of the code can interact with the persistence code and should be in their own project/dll

OrderRepository

OrderDao
OrderItemDao

CustomerRepository

CustomerDao
AddressDao

DDD – Aggregates

Avoid code like

```
public class Order
{
        public decimal GetPrice()
        {
                OrderDao.GetItems(orderId);
                …..
        }
}
```

Whenever you think that you need to access a Dao/Repository in your Domain use one of these alternatives

- Dto/Entity Mapping in the Repository (OrderItems can be a collection in Order mapped during data retrieval)
- Make the item(s) you need an argument in your function e.g. GetPrice(IEnumerable<OrderItems> items)
- If you need to use the dao/repo many times within a function consider passing it as an argument e.g. GetPrice(IOrderDao dao)

DDD – Repositories

One to One relationship between Aggregate and Repository

When you retrieve an object from the Repository then this should be the complete aggregate e.g.

```
public class Order
{
        public int OrderId {get; private set;}
        public List<IOrderItem> OrderItems {get; private set;}
        public decimal GetPrice() {…}
        …..
}
```

Once you retrieve the object and make a change to it you should be able to easily persist it using the Repository

```
// A simple interface for a Repo
public class OrderRepository : IOrderRepository
{
        public void Save(Order order){…}
        public Order Get(int orderId){…}
        public void Delete(int orderId){…}
}
```

DDD – Repositories

```
// A simple interface for a Repo
public class OrderRepository : IOrderRepository
{
        public void Save(Order order){…}
        public Order Get(int orderId){…}
        public void Delete(int orderId){…}
        public IEnumerable<IOrder> GetAll(){…}
}
```

When creating a Repo Interface place the interface definition in the same Module as the Aggregate type that it stores.
So IOrderRepository will be in the module/project where Order is defined.

What else can go in??
Some devs would like to keep this very basic interface and any other querying can be considered business logic that needs to go in the domain.

Others are happy to put some basic queries in the Repo e.g.

```
public IEnumerable<IOrder> GetUnapprovedOrders()
{
    // some filtering
}
```

DDD – Repositories

What should I return when it comes to Collections?

Depends on your requirements

- IEnumerable
  - Be aware of the danger of multiple enumerations from callers of your code
- IQueryable
  - Make sure you understand the difference with IEnumerable

  “The difference is that IQueryable<T> is the interface that allows LINQ-to-SQL (LINQ.-to- anything really) to work. So if you further refine your query on an IQueryable<T>, that query will be executed in the database, if possible.
  For the IEnumerable<T> case, it will be LINQ-to-object, meaning that all objects matching the original query will have to be loaded into memory from the database.”

- A more specific type like IList or even Array
  - May be ok if you don’t need the flexibility the other 2 options can provide

DDD – Services

A Service in the domain is a stateless operation that fulfils a domain-specific task.

“Often the best indication that you should create a Service in the domain model is when the operation you need to perform feels out of place as a method on an Aggregate or a Value Object. To alleviate that uncomfortable feeling, our natural tendency might be to create a static method on the class of an Aggregate Root. *However, when using DDD, that tactic is a code smell that likely indicates you need a Service instead.”*

2 types of Services

- Domain Service
  - Can have business logic
- Application Service
  - No business logic, delegates to domain, transaction and security management

DDD – Services

You can use a Domain Service to
- Perform a significant business process
- Transform a domain object from one composition to another
- Calculate a Value requiring input from more than one domain object

So typically Services will access Repositories and other service classes (Injected through the constructor) and delegate as much work as possible to Domain classes such as Order, Customer etc.

```
public class FinanceService
{
        public void CreditGoldMemberAccounts(decimal amountToCredit)
        {
                var goldMembers = customerRepository.GetByMemberType(MemberType.Gold);

                foreach (var goldMember in goldMembers)
                {
                        var amountOfRecentOrders = GetRecentOrdersAmount(goldMember.Id);
                        goldMember.Credit(amountToCredit, amountOfRecentOrders);
                }
                ……
        }
}
```

DDD – Services

Application Services used for Transaction Management & Security. They hide details of the Domain Model. They offer a public API for a number of different clients.

A “unit of work” pattern can be used. *("maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.“)*

```
public OrderService IOrderService
{
        public void DoSomeTask()
        {
                Transaction transaction = null;
                try
                {
                        transaction = this.session(). beginTransaction();
                        // use the domain model ...
                        transaction.commit();
                }
                catch (Exception e)
                {
                        if (transaction != null)
                        {
                                transaction.rollback();
                        }
                }
        }
}
```