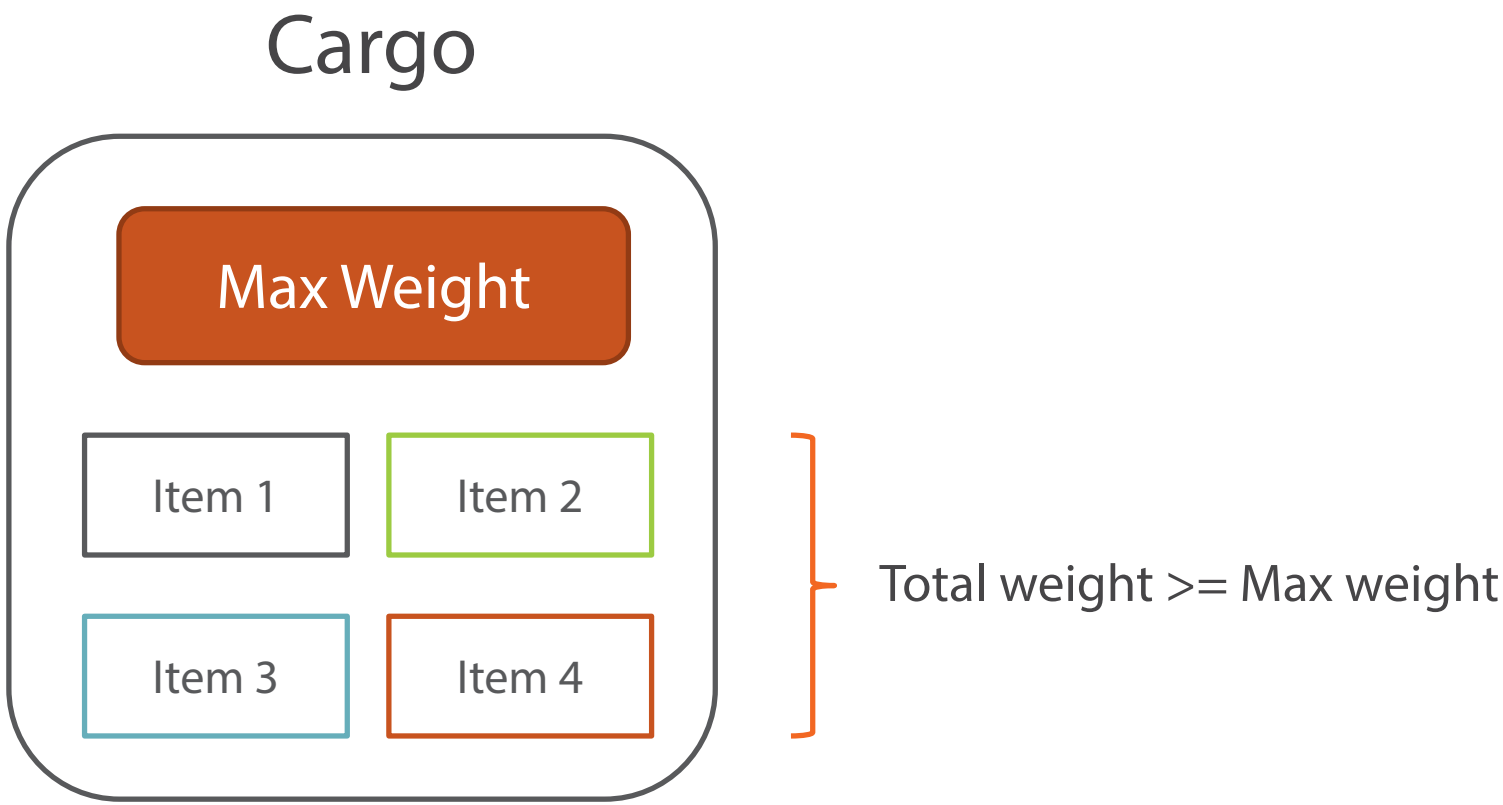
Looking Forward to Further Enhancements

Vladimir Khorikov

@vkhorikov | www.enterprisecraftsmanship.com

In This Module

Always Valid vs. Not Always Valid



“Always Valid” Approach

```
public class Cargo : AggregateRoot
{
    public int MaxWeight { get; protected set; }
    protected IList<Product> Items { get; }

    public Cargo(int maxWeight)
    {
        MaxWeight = maxWeight;
        Items = new List<Product>();
    }

    public void AddItem(Product product)
    {
        int currentWeight = Items.Sum(x => x.Weight);
        if (currentWeight + product.Weight > MaxWeight)
            throw new InvalidOperationException();

        Items.Add(product);
    }
}
```

“Not Always Valid” Approach

```
public class Cargo : AggregateRoot
{
    public int MaxWeight { get; protected set; }
    protected IList<Product> Items { get; }

    public Cargo(int maxWeight)
    {
        MaxWeight = maxWeight;
        Items = new List<Product>();
    }

    public void AddItem(Product product)
    {
        Items.Add(product);
    }

    public bool IsValid()
    {
        int currentWeight = Items.Sum(x => x.Weight);
        return currentWeight <= MaxWeight;
    }
}
```

Always Valid vs. Not Always Valid

Always valid

- Don't have to worry about validation

VS

Not always valid

- Gather most validations in one place

Always Valid vs. Not Always Valid

Prefer the "Always Valid" approach

- ☐ Removes temporal coupling
- ☐ Helps with DRY
- ☐ Classes maintain their invariants

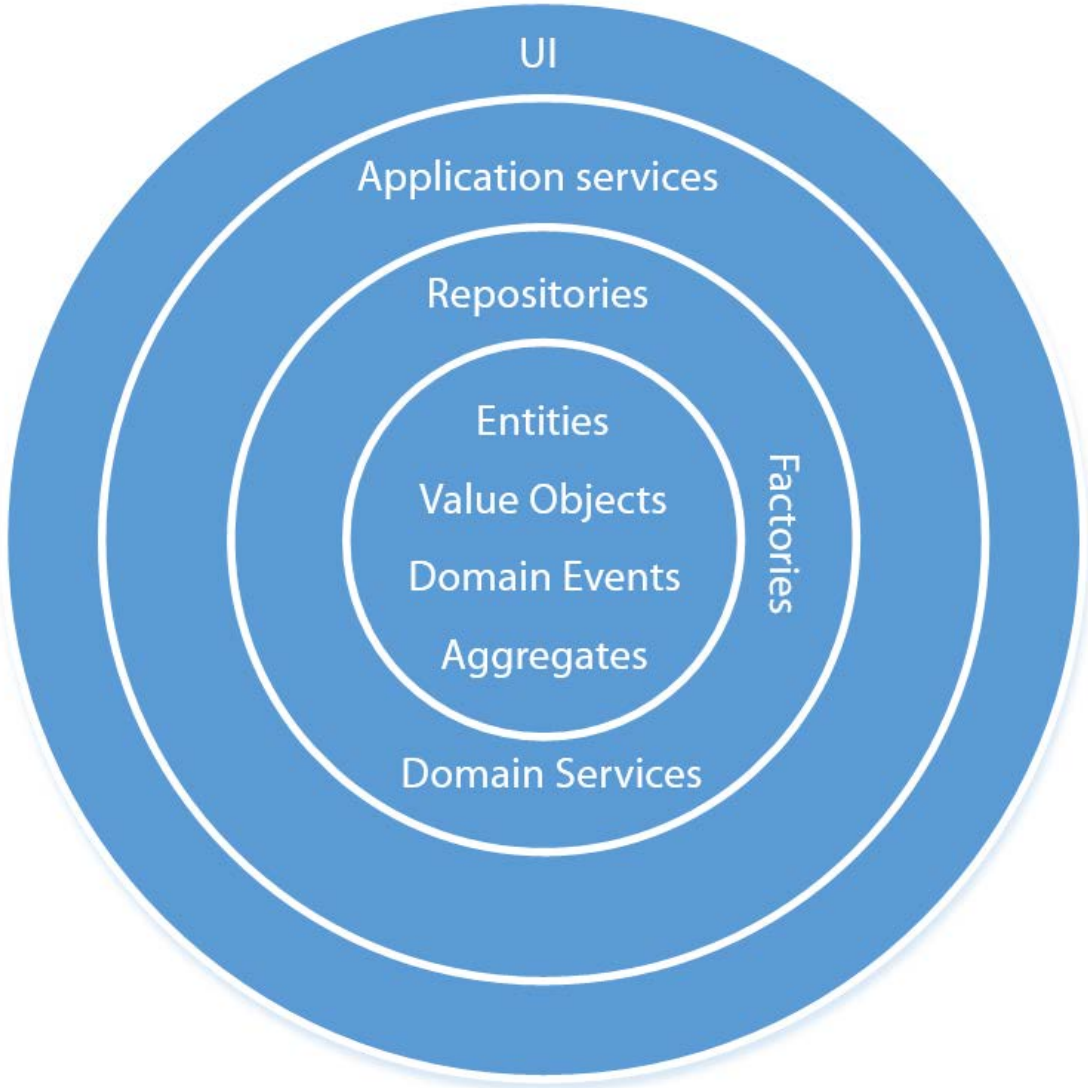
Fail fast principle: http://bit.ly/1RrHvj8

Always Valid vs. Not Always Valid

Where to perform validations?

At the domain layer boundaries

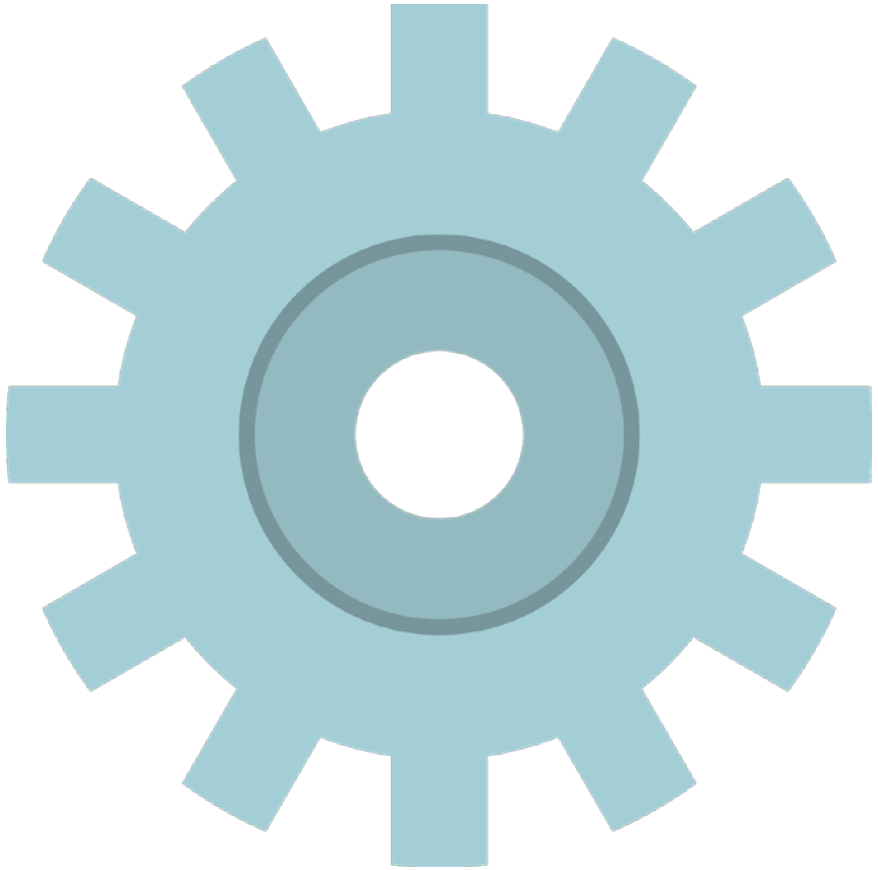
```
string err = _machine.CanBuySnack(position);
if (err != string.Empty)
{
    NotifyClient(error);
    return;
}
_machine.BuySnack(position);
```



Factories

- Create domain entities
  - Complex creation logic
  - Helps simplify entities
- Don’t use factories in case the creation logic is simple

Domain Services



- Don’t have state
- Contain domain logic
- Possess knowledge that doesn’t belong to entities and value objects

Domain Services vs. Application Services

Domain Service

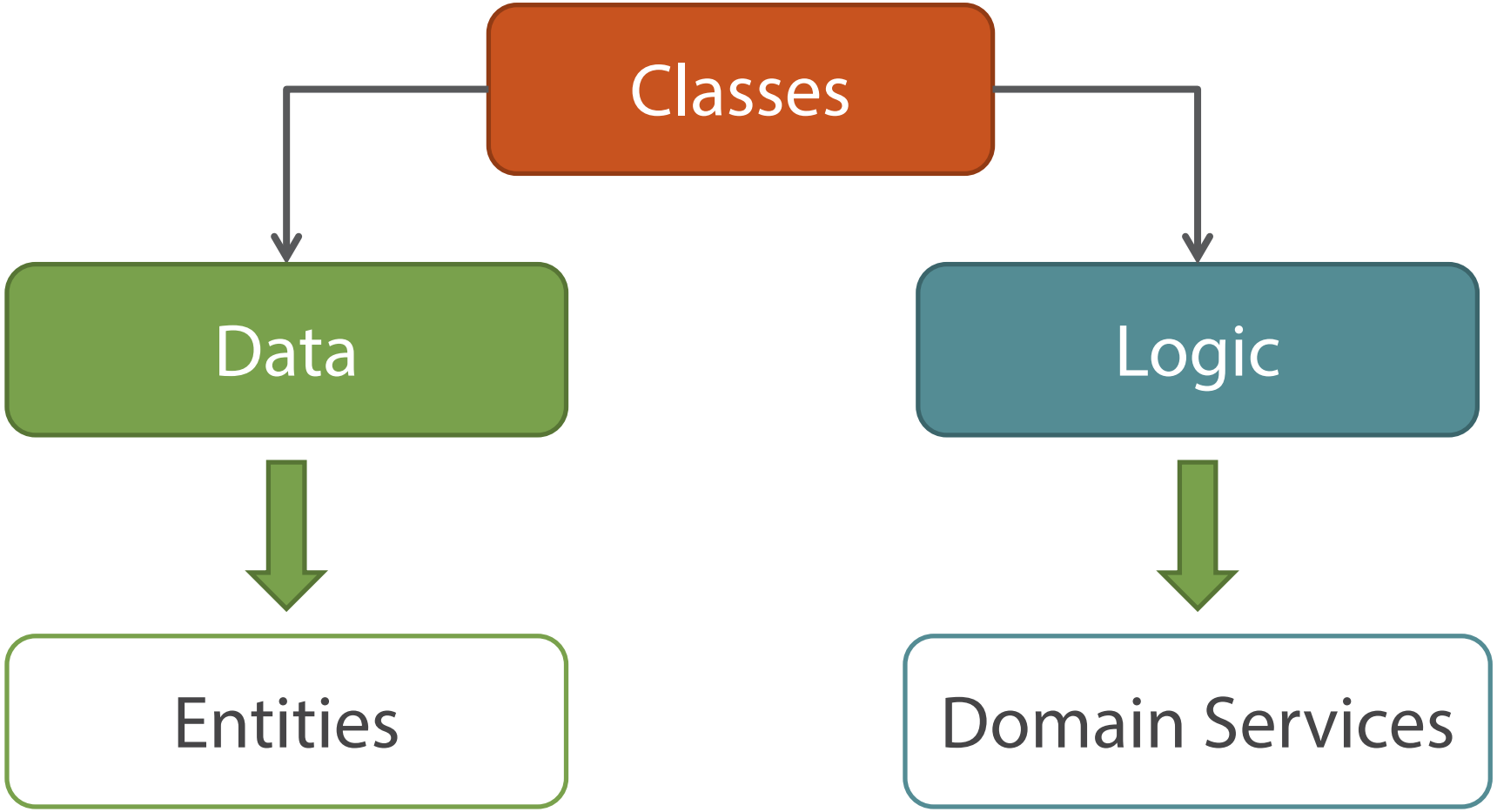
- Inside of the domain layer
- Contains domain logic
- Doesn't communicate with the outside world

VS

Application Service

- Outside of the domain layer
- Communicates with the outside world
- Doesn't contain domain logic

Anemic Domain Model Anti-pattern



Anemic Domain Model Anti-pattern

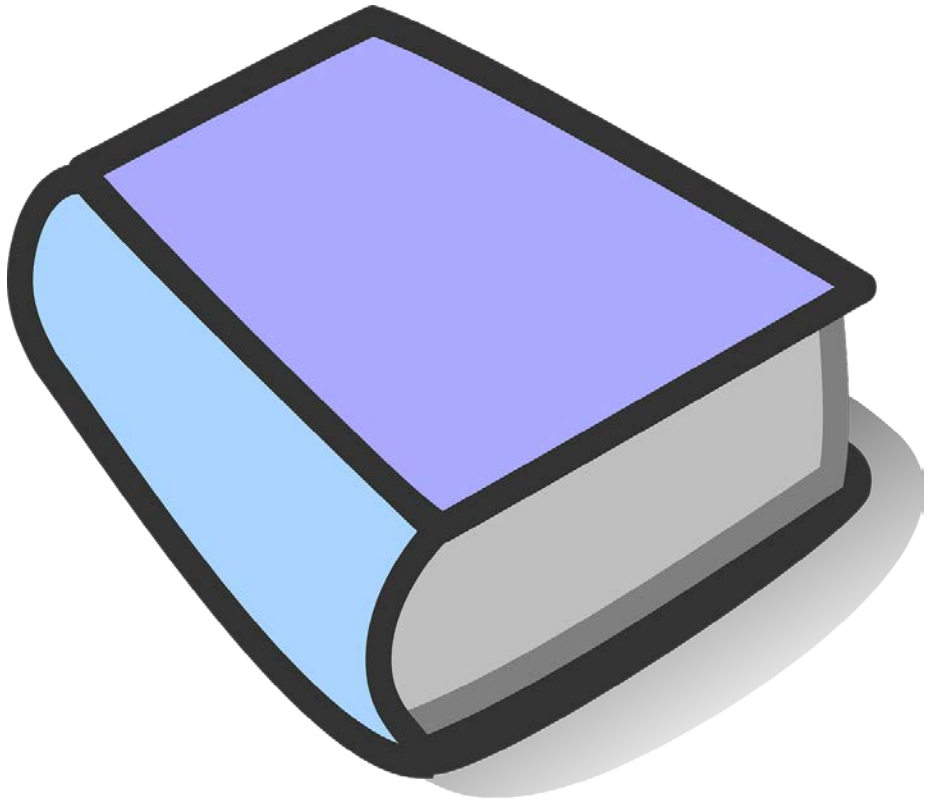
```
public class SnackMachine : AggregateRoot {
    public virtual Money MoneyInside { get; set; }
    public virtual decimal MoneyInTransaction { get; set; }
    public virtual IList<Slot> Slots { get; set; }
}

public class SnackMachineService {
    public void BuySnack(SnackMachine snackMachine, int position) {}
    public void LoadSnacks(SnackMachine snackMachine, int position, SnackPile snackPile) {}
    public void LoadMoney(SnackMachine snackMachine, Money money) {}
}
```

Poor encapsulation

http://bit.ly/LRPqYO

Fat Entities Anti-pattern



- Too much logic in entities
- Entities with unnatural responsibilities
- Look up date in external sources
- Communicate with external layers

Repository Anti-patterns

```
public class SnackMachineRepository : Repository<SnackMachine>
{
    public IReadOnlyList<SnackMachine> GetAll()
    {
        /* Return a list of fully initialized machines */
    }

    public IReadOnlyList<SnackMachine> GetAllWithoutSlots()
    {
        /* Return a list of machines without slots */
    }

    public IReadOnlyList<SnackMachine> GetOnlyIds()
    {
        /* Return a list of machines with only identifiers */
    }
}
```

Partially initialized entities

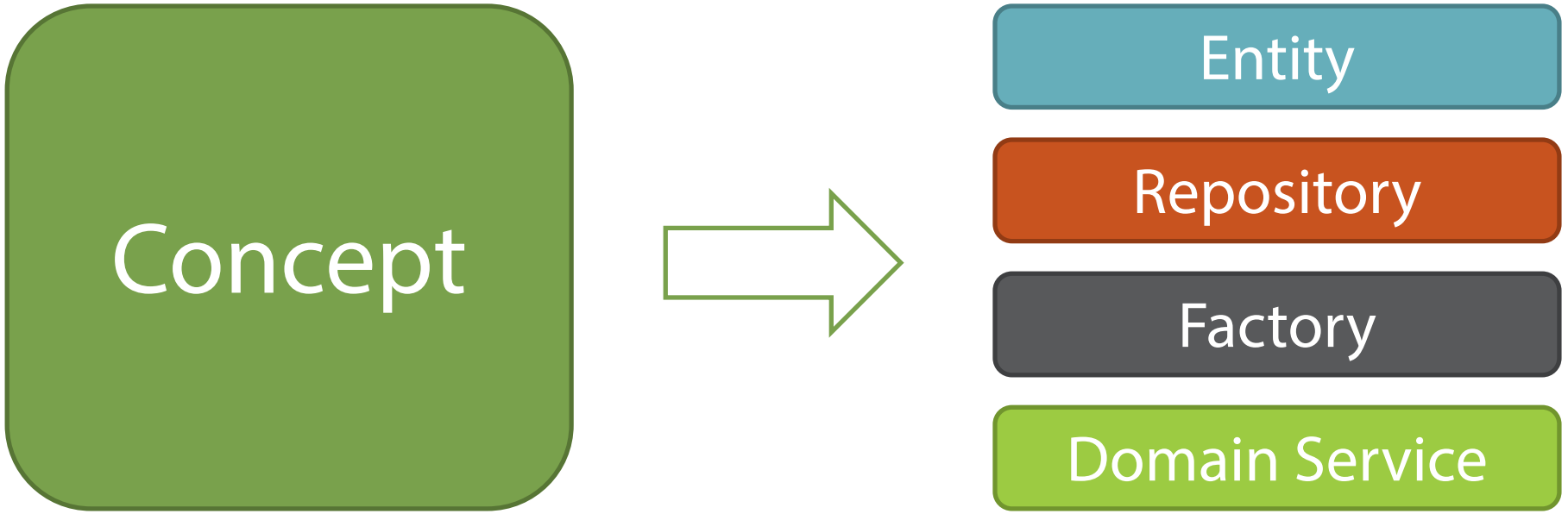
Repository Anti-patterns

```
public class SnackMachineRepository : Repository<SnackMachine>
{
    public IReadOnlyList<SnackMachine> GetAll()
    {
        /* Return a list of fully initialized machines */
    }

    public IReadOnlyList<SnackMachineDto> GetAllWithoutSlots()
    {
        /* Return a list of DTOs */
    }

    public IReadOnlyList<long> GetOnlyIds()
    {
        /* Return a list of identifiers */
    }
}
```

Mechanical Approach to DDD



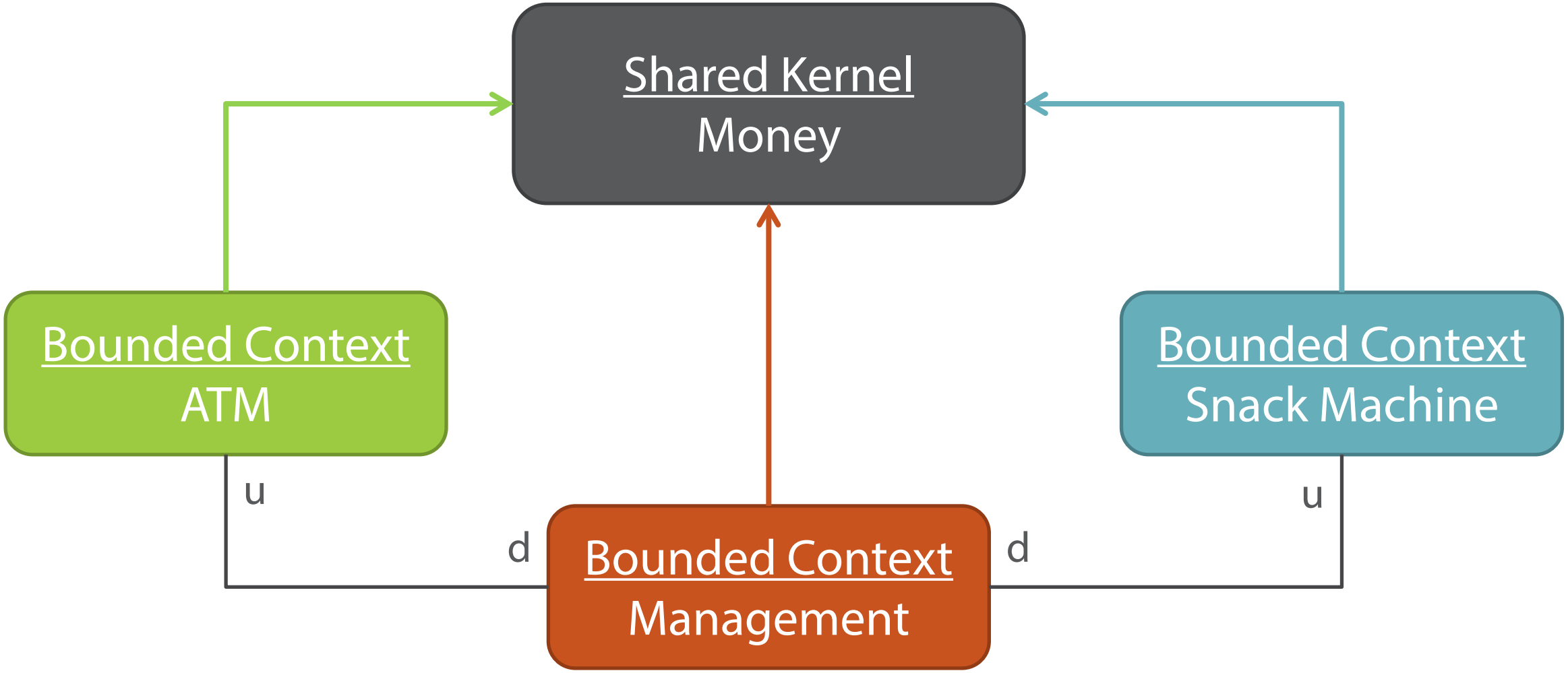
Mechanical Approach to DDD

Domain modelling is learning

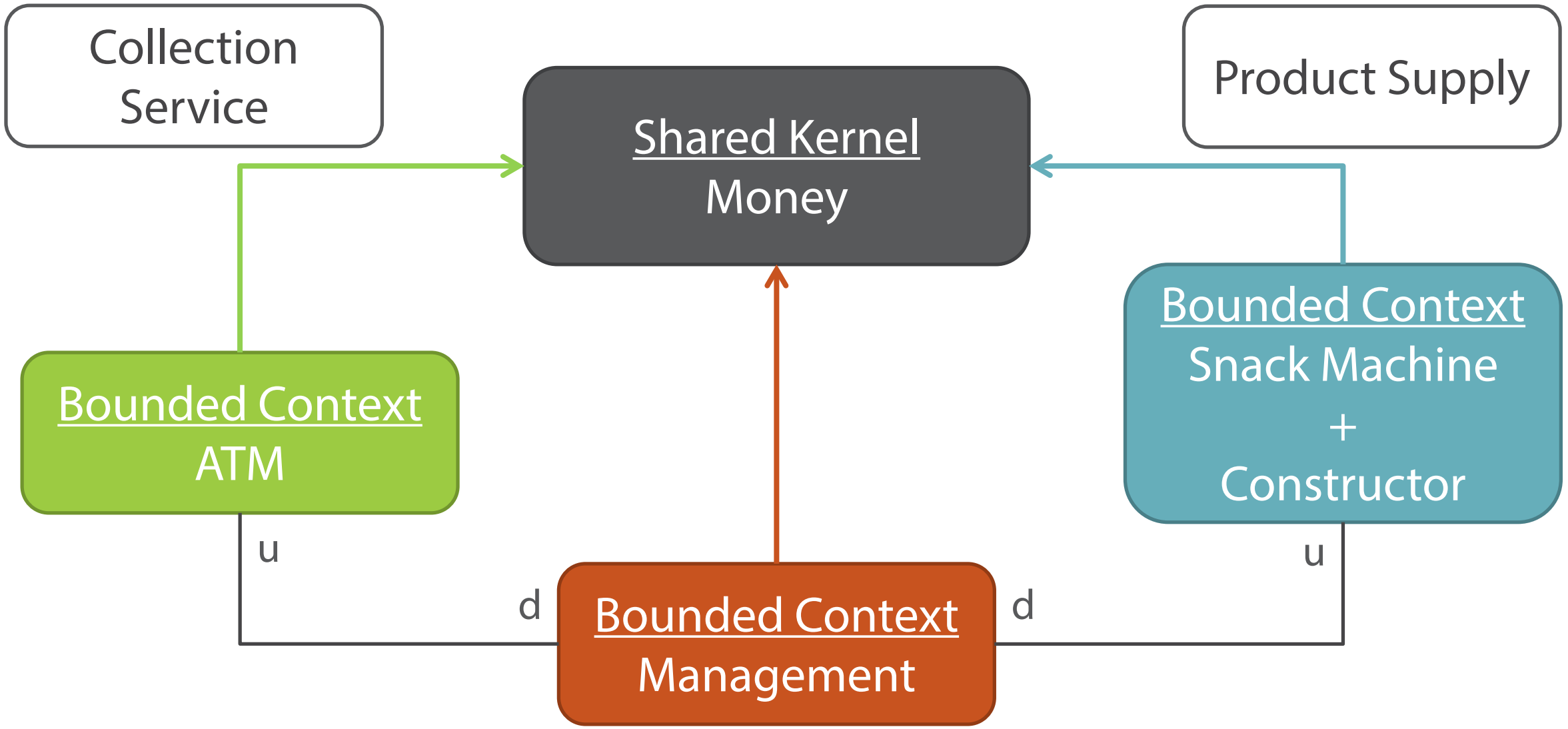
Code generation

Scaffolding

Further Enhancements



Further Enhancements



Module Summary

- Factories
- Domain Services vs Application Services
- Always Valid vs Not Always Valid approaches
- Anemic domain model
- Fat entities
- Repository anti-patterns
- Mechanical approach to DDD
- Further enhancements

Resource List

Source code	https://github.com/vkhorikov/DddInAction
	http://bit.ly/1OxbGEA
Integration testing	http://enterprisecraftsmanship.com/2015/07/13/integration-testing-or-how-to-sleep-well-at-nights/
	http://bit.ly/1hT842g
Overriding GetHashCode method	http://stackoverflow.com/questions/371328/why-is-it-important-to-override-gethashcode-when-equals-method-is-overridden
	http://bit.ly/1FSzTg1
Test-first vs code-first approaches to unit testing	http://enterprisecraftsmanship.com/2015/08/03/tdd-best-practices/
	http://bit.ly/1XF0J6H
Hi/Lo algorithm	http://stackoverflow.com/questions/282099/whats-the-hi-lo-algorithm/282113#282113
	http://bit.ly/1l4ablz
Example of context mapping	https://vimeo.com/125769142
Microservices	http://martinfowler.com/articles/microservices.html
	http://bit.ly/1dI7ZJQ
Cohesion and coupling	http://enterprisecraftsmanship.com/2015/09/02/cohesion-coupling-difference/
	http://bit.ly/1lisDBQ
Types of CQRS	http://enterprisecraftsmanship.com/2015/04/20/types-of-cqrs/
	http://bit.ly/1ZL8yc7
Fail fast principle	http://enterprisecraftsmanship.com/2015/09/15/fail-fast-principle/
	http://bit.ly/1RrHvj8
Anemic domain model	http://www.martinfowler.com/bliki/AnemicDomainModel.html
	http://bit.ly/LRPqYO

Course Summary

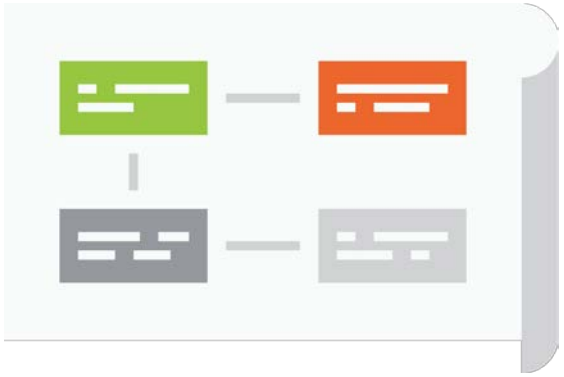
Full application from scratch

Domain modeling

DDD concepts in practice

Database and ORM

Unit testing



MVVM

Contacts

vladimir.khorikov@gmail.com

@vkhorikov

http://enterprisecraftsmanship.com/