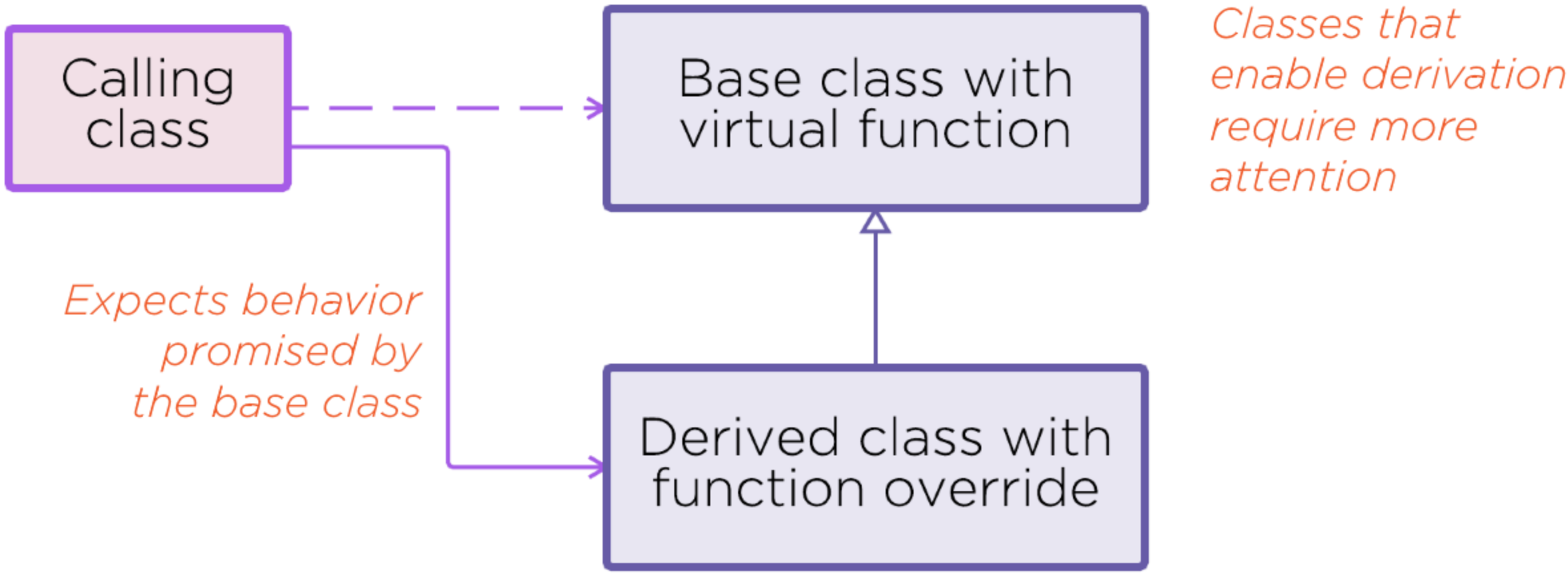
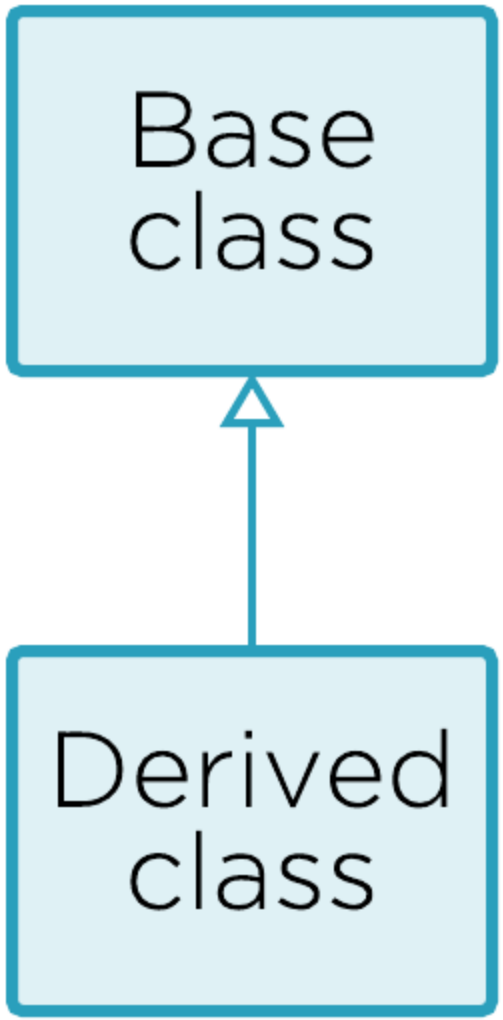
Testing Against Interfaces to Simplify Maintenance

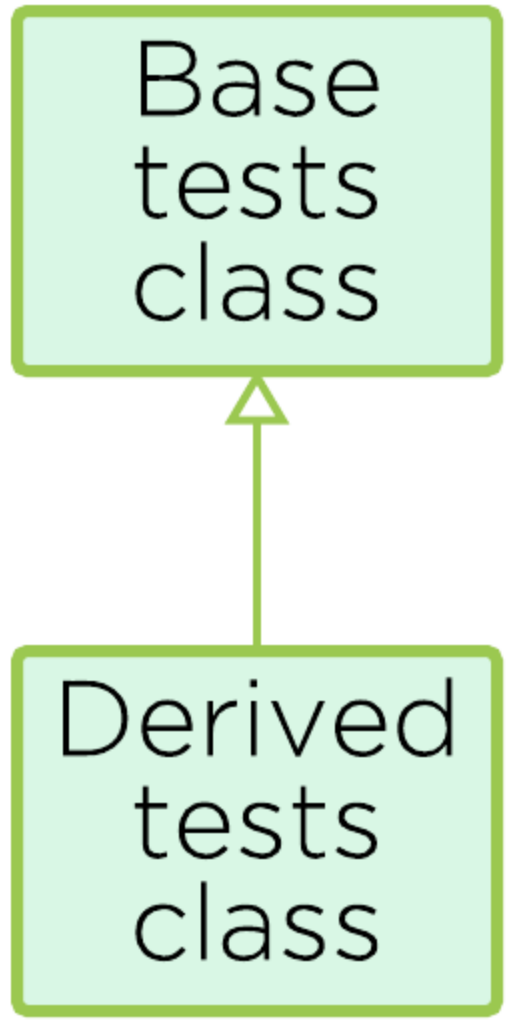
Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com


Calling class
Base class with virtual function
Derived class with function override
Expects behavior promised by the base class
Classes that enable derivation require more attention





Which responsibility to put into which class?

Code smell

Derived test calls member of the base tests class

Testing code must be bug-free

Advice

Keep all code regarding one test case in one place (base or derived class)

Rule of Conduct when Deriving Test Classes

Class derivation

```
abstract class Vehicle
{
  public void Drive { ... }

  protected abstract void
    BeforeStart();
  protected abstract void
    AfterStart();
}
```

Derived tests class specializes the SUT

Tests derivation

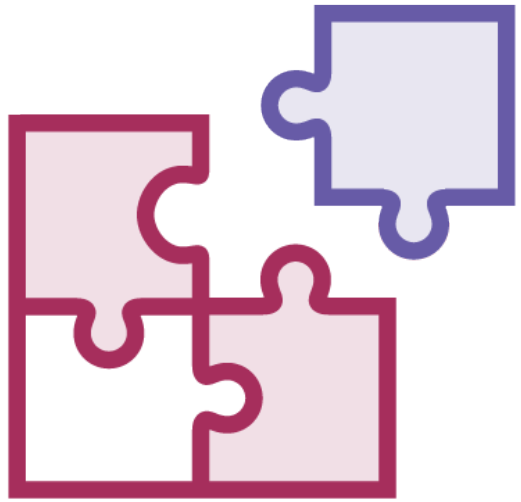
```
abstract class VehicleTests
{
  public void Test1() { ... }
  public void Test2() { ... }
  ...
  protected abstract Vehicle
    CreateSut();
}


class BusTests : VehicleTests
{
  protected override Vehicle
    CreateSut() => new Bus();
}
```

```
interface IRepository
{
  ...
}

abstract class RepositoryTests
{
  protected abstract IRepository
    CreateSut();
}

class ConcreteRepositoryTests
  : RepositoryTests
{
  protected override IRepository
    CreateSut() { ... }
}
```

- Demonstrate tests derivation on an example of abstract repository
- Base tests class will contain unit tests for the abstract repository
- Derived tests class will specialize tests to a concrete repository class

Practical Guide to Abstractions

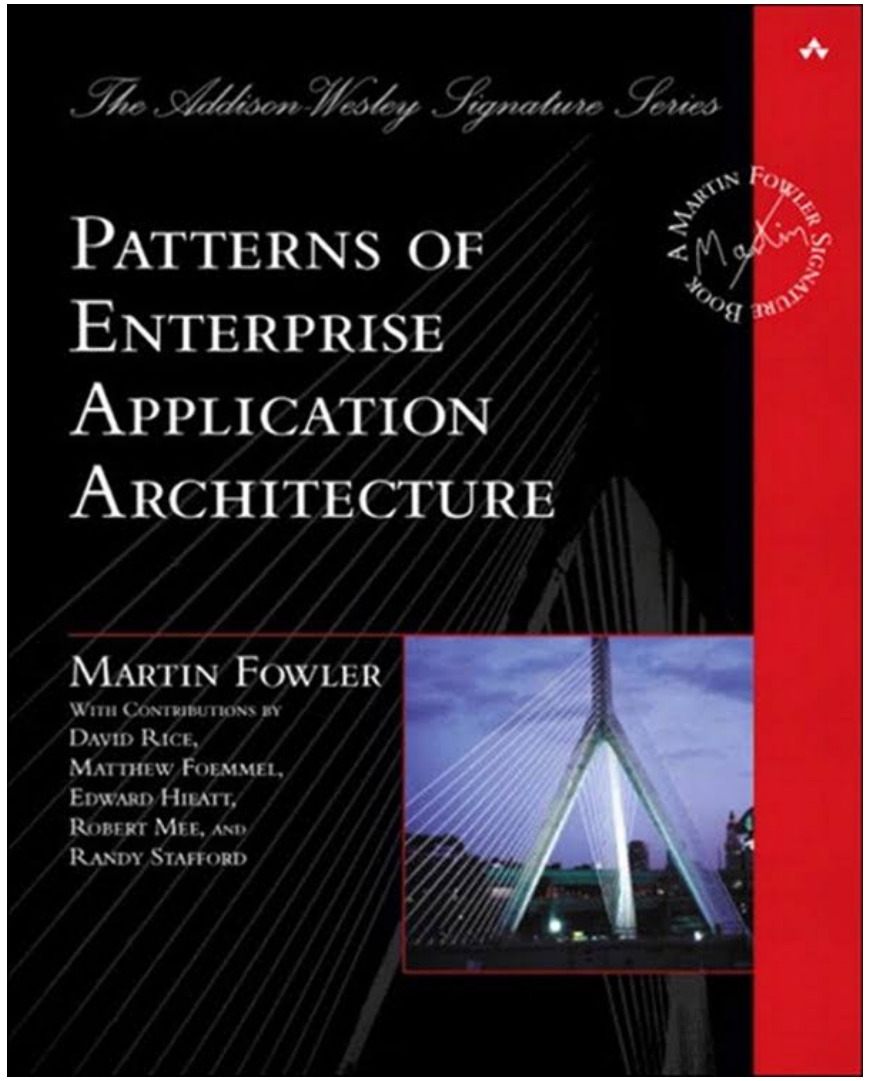
Many ideas can be defended on the grounds of logic



Still, many ideas that make sense lead to inconveniences

Advice

Step back and reconsider original ideas from time to time




```
interface IRepository<T>
{
  IEnumerable<T> GetAll();
  void Add(T obj);
}
```

Repository

In-memory representation of storage

Identity Map

Maps storage IDs to materialized objects

Unit of Work

Tracks changes to materialized objects

Data Mapper

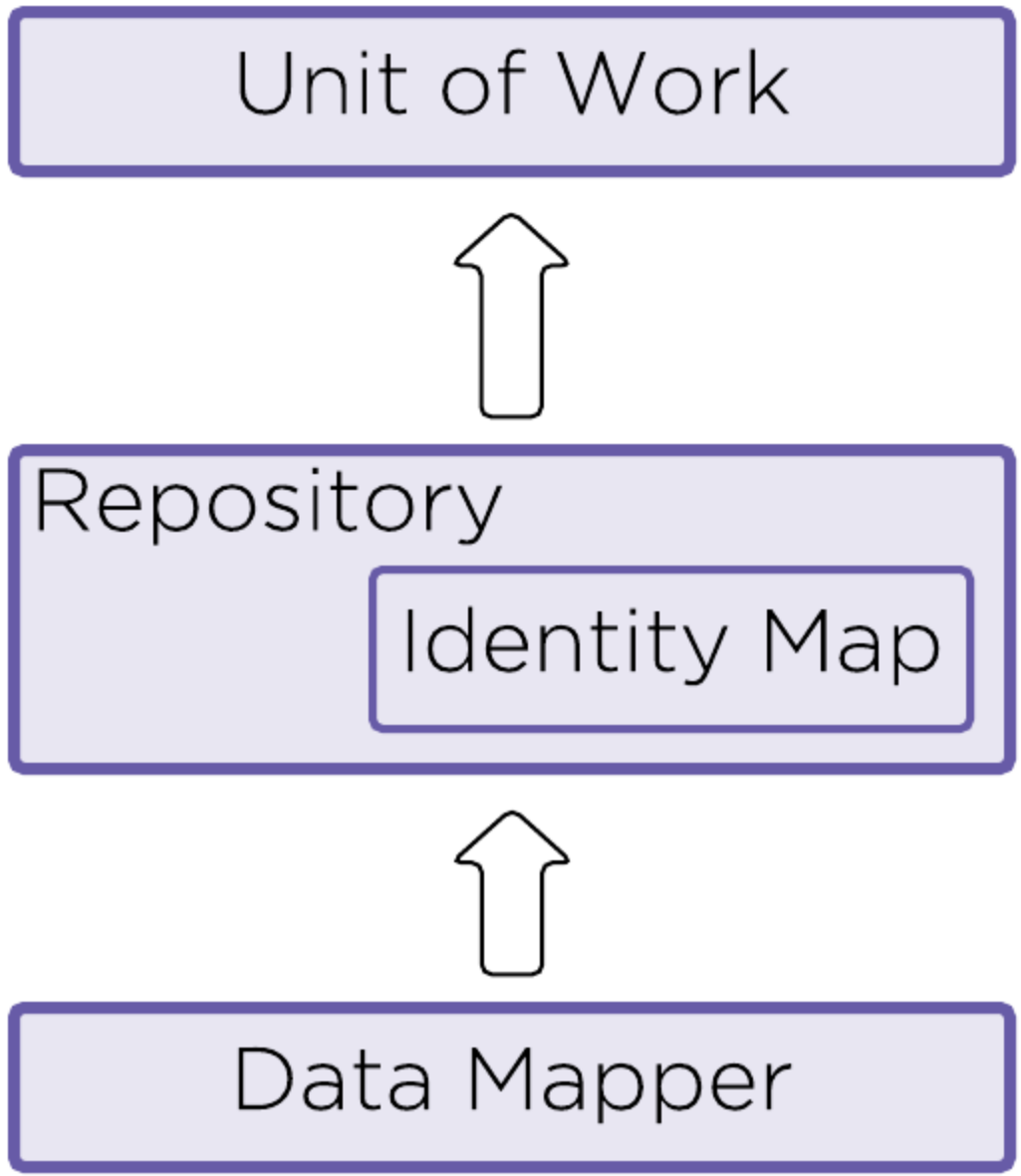
Maps changes to storage statements

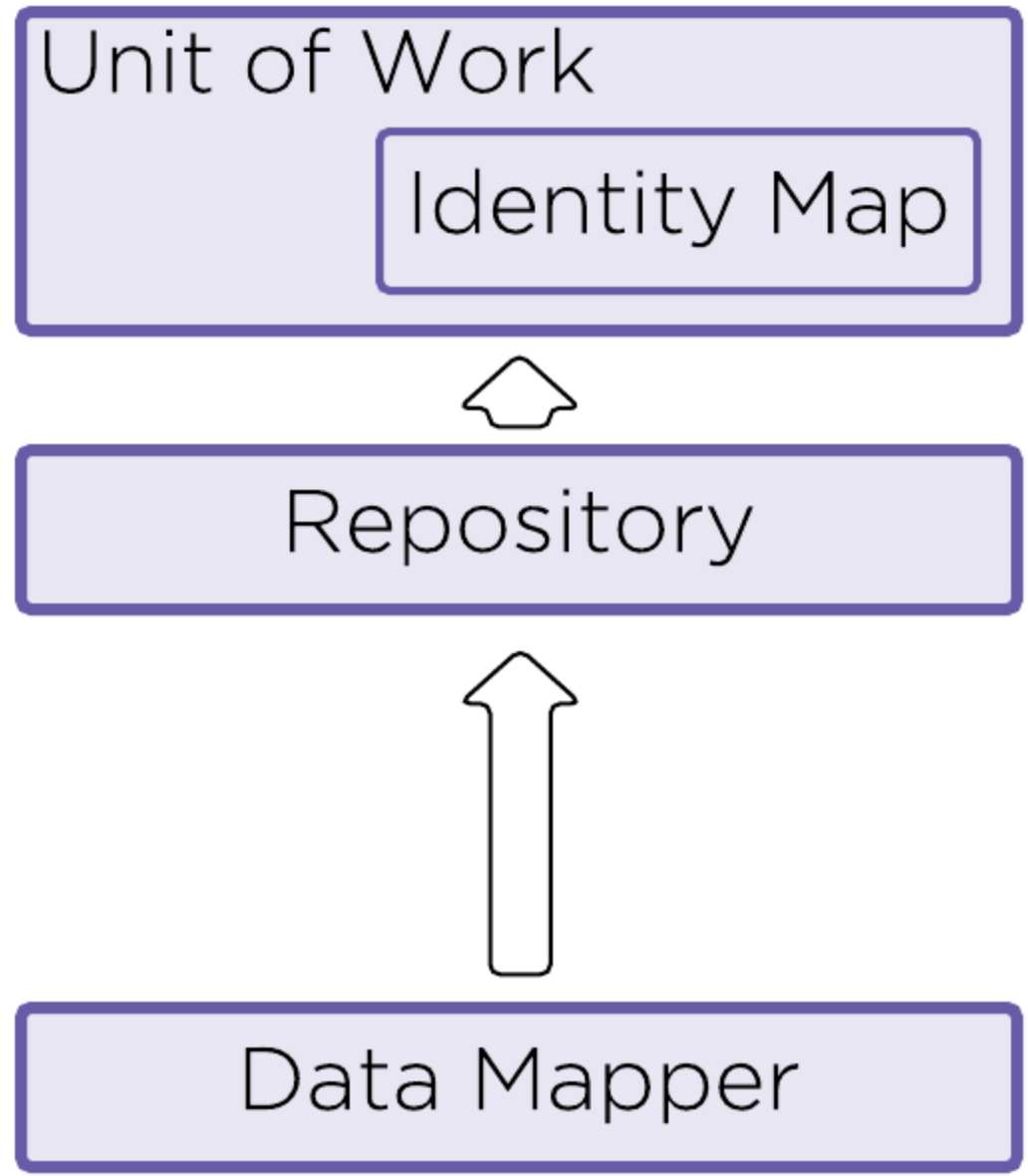
Unit of Work

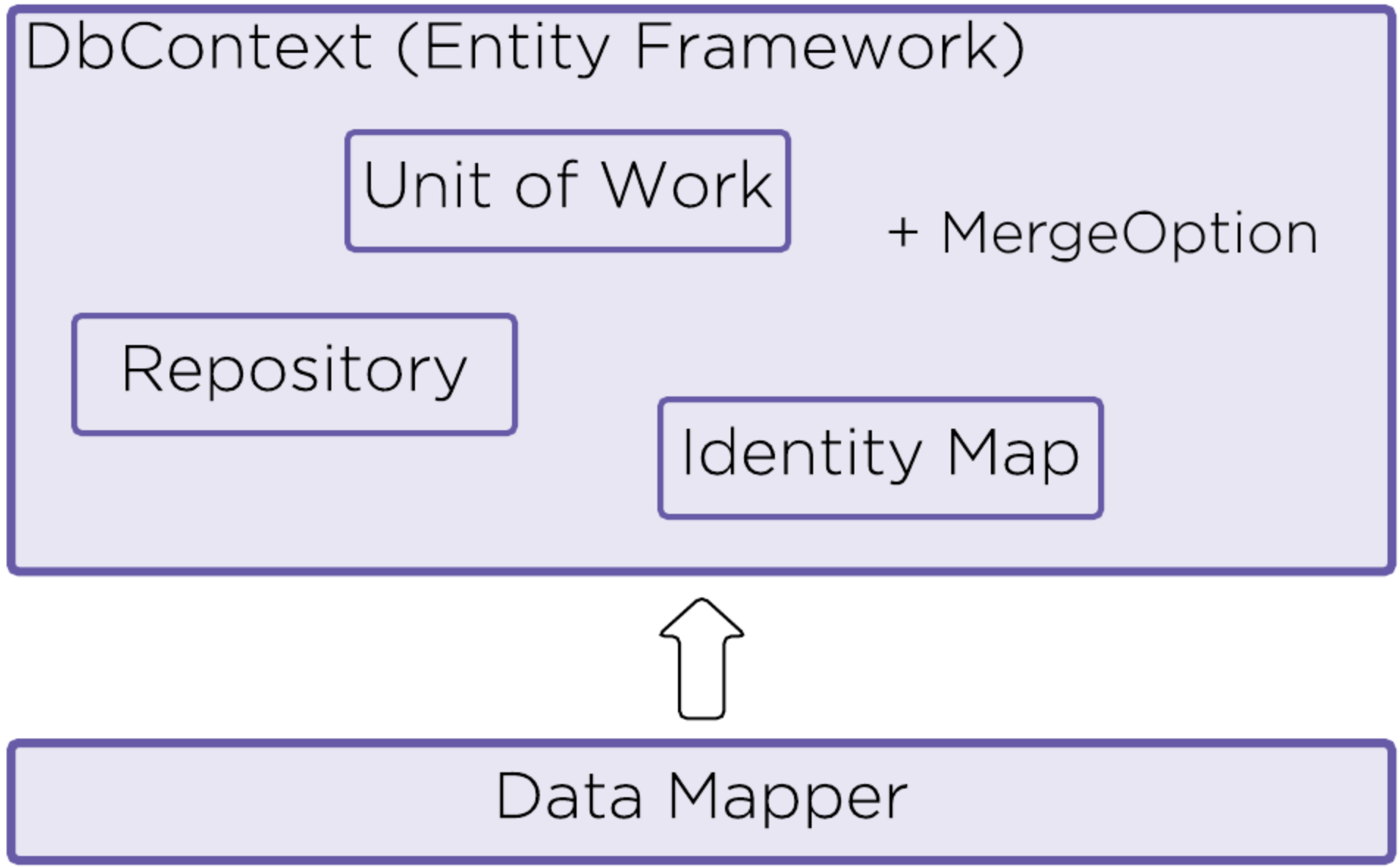
Repository

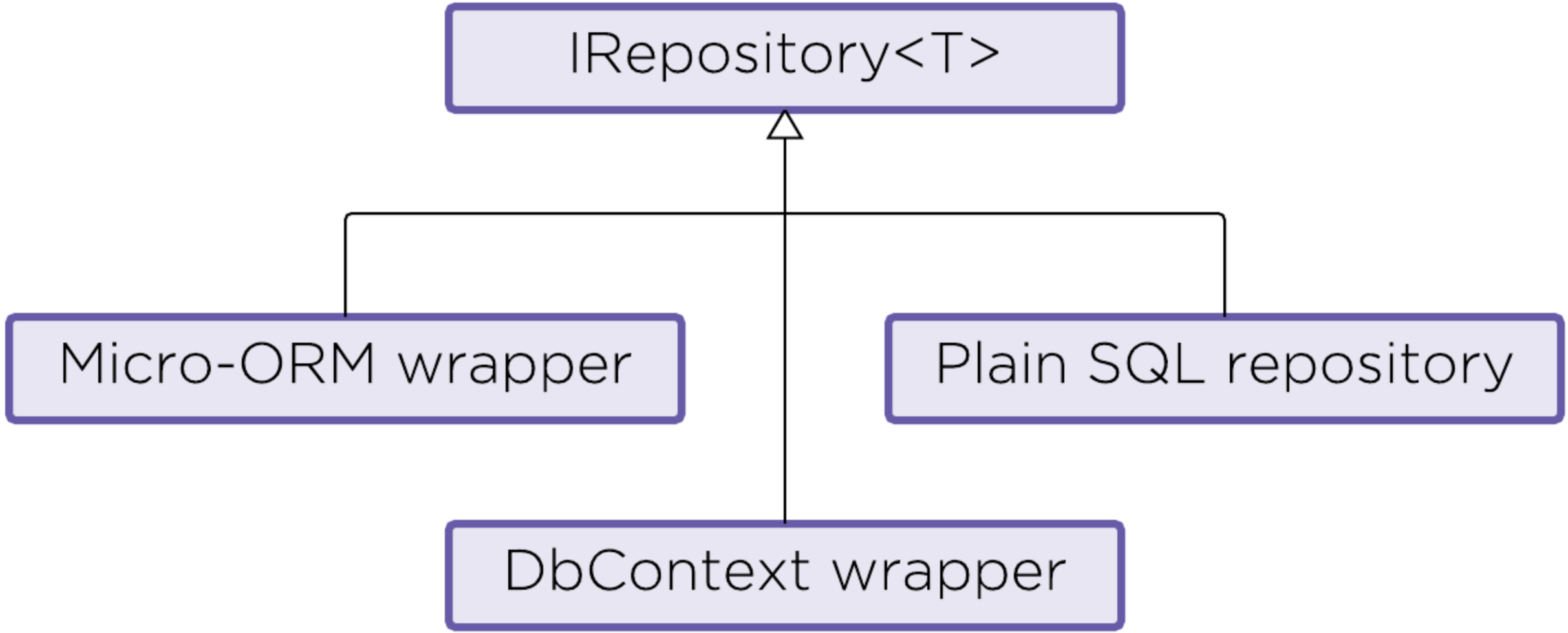
Identity Map

Data Mapper

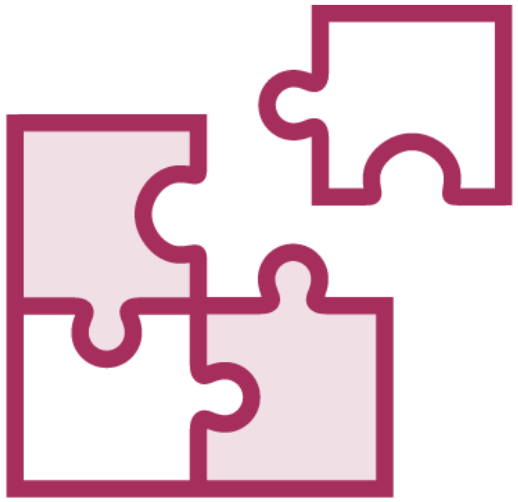

Unit of Work
Repository
Identity Map
Data Mapper


Unit of Work
Identity Map
Repository
Data Mapper

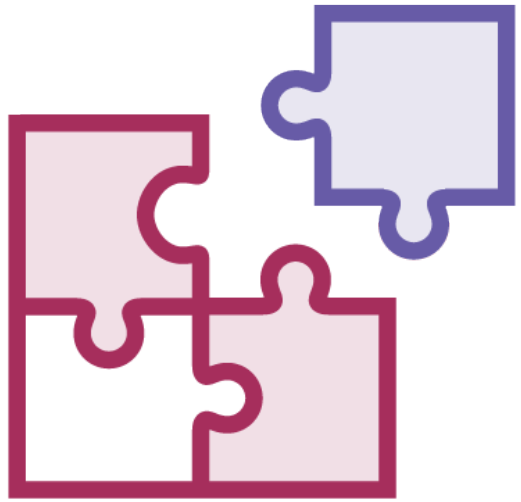

DbContext (Entity Framework)
Unit of Work
+ MergeOption
Repository
Identity Map
Data Mapper


IRepository<T>
Micro-ORM wrapper
Plain SQL repository
DbContext wrapper

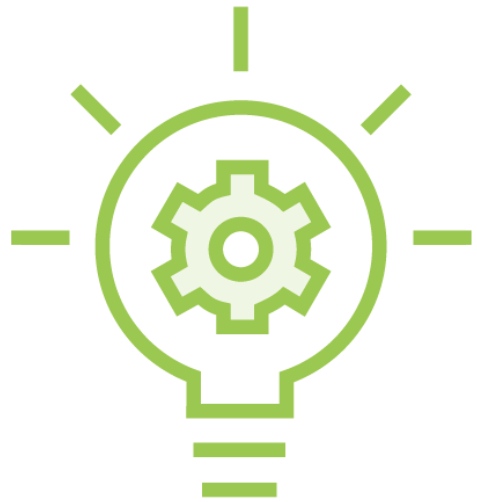
Does a Rule Belong to ADT?



Imagine concrete implementation which **satisfies** the rule (this should be easy)



Imagine concrete implementation which **breaks** the rule (this might be harder)



If both are valid implementations, then we're talking about an **implementation detail**

Comparing Implementations

Satisfying the rule

Add(obj)
Add **obj** to in-memory collection

GetAll()
Select data from storage
New objects are not included

Save()
Persist newly added objects

Breaking the rule

Add(obj)
Write **obj** to database (don't commit)

GetAll()
Select data from storage
New objects are included

Save()
Commit database transaction

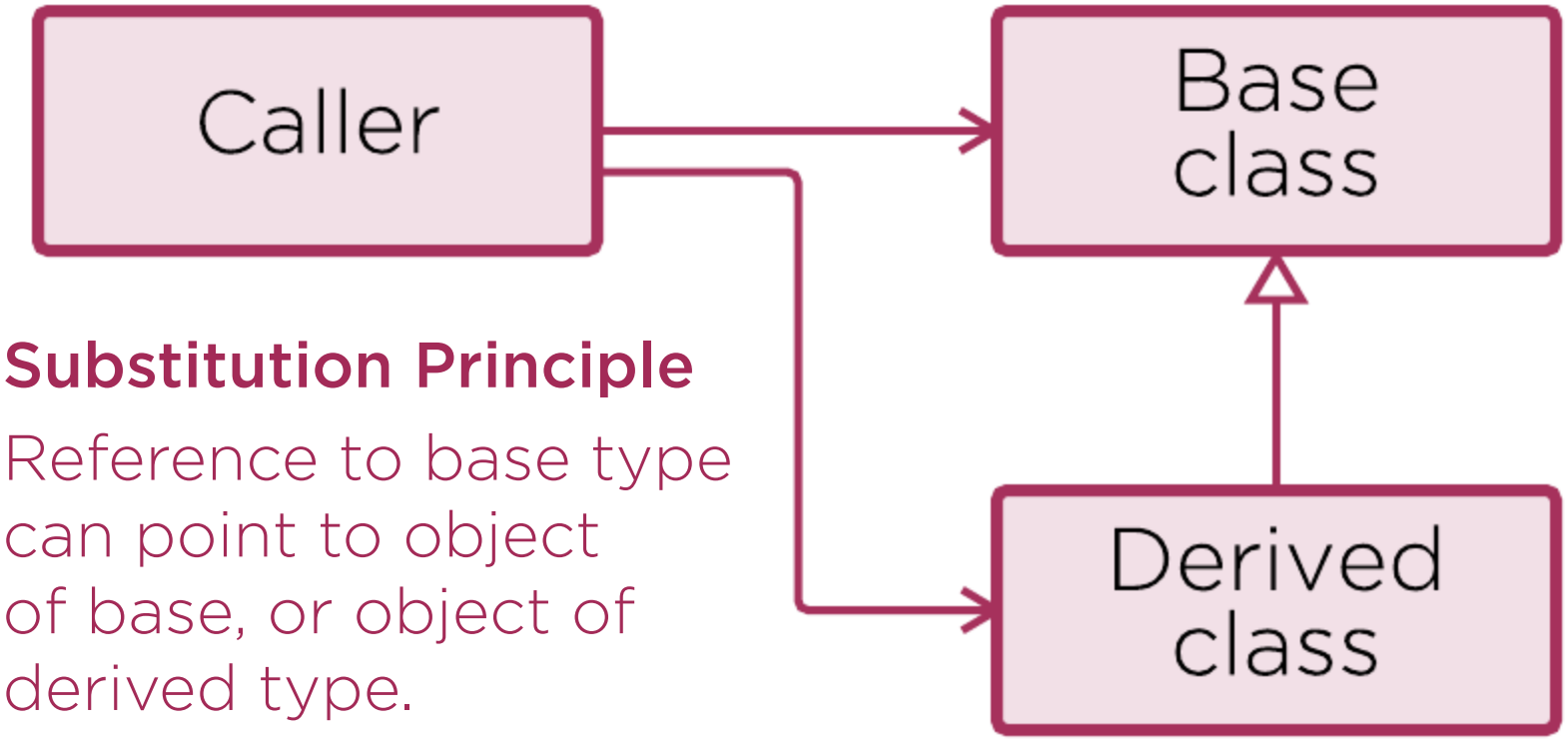
Conclusion

Rule in question is not part of the Abstract Data Type.

It is an implementation detail.

Object Substitution

Liskov Substitution Principle (LSP)



Substitution Principle

Reference to base type can point to object of base, or object of derived type.

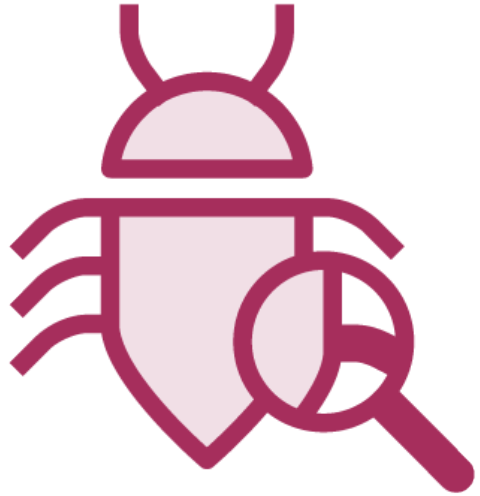
Attribute ***A*** holds on base objects

Attribute ***A*** must also hold on derived objects

If LSP is violated, call might fail after object substitution.

Liskov Substitution Principle

One of the most widely violated principles in object-oriented programming

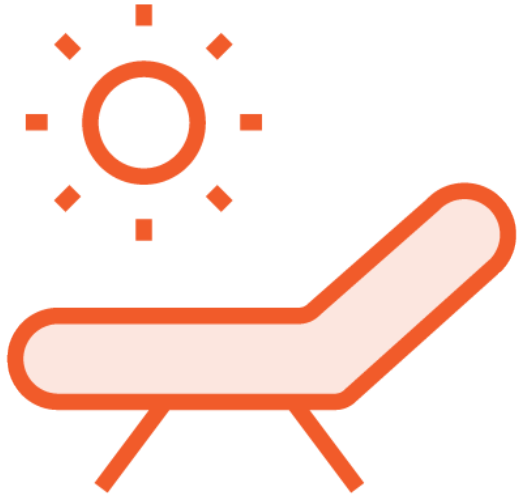


... primarily due to lack of care

ADT & LSP

Define rules
of the ADT

Satisfy rules
by all concrete
implementations

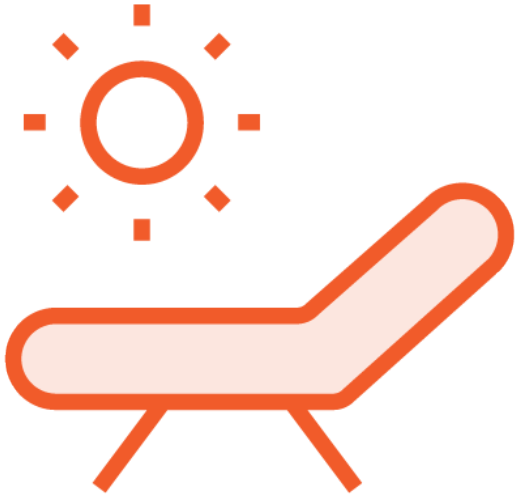


Now LSP will be
satisfied free of charge

ADT & LSP

Abstract tests for an interface are documenting the rules of ADT

Concrete implementations of the interface must pass the tests



LSP is satisfied

Summary

Developing abstractions

- Defining a type by only talking about its abstract behavior

Working with Abstract Data Types (ADTs)

- Clear view of rules defining behavior
- Turn abstract rules into unit tests

Design outcome

- ADT produces the interface
- Rules produce unit tests as executable documentation

Summary

Interface testing

- Possibility of writing unit tests against an interface
- Abstract tests are not discoverable
- Derived tests class must specialize the tests to a concrete class under test

Next module:

Testing derived classes