Implementing Abstract Interfaces TDD Style

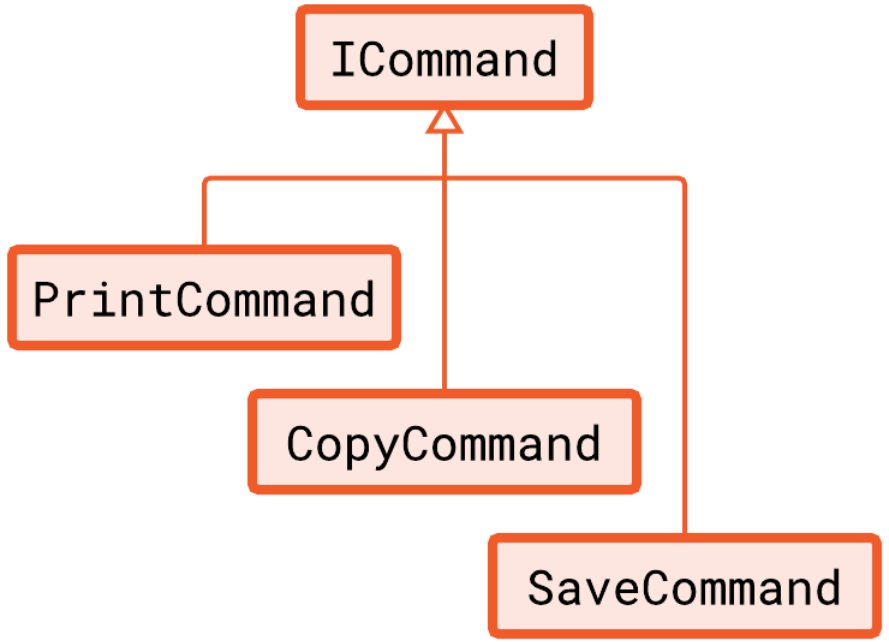
Zoran Horvat

PRINCIPAL CONSULTANT AT CODING HELMET

@zoranh75 csharpmentor.com

Inheritance Options

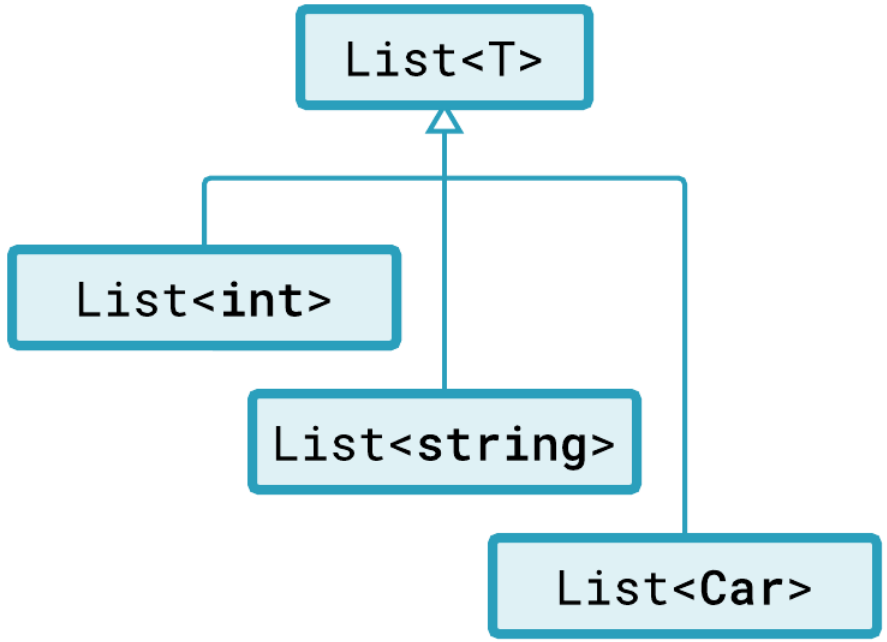
— **Interface implementation**




```
PrintCommand is ICommand
 CopyCommand is ICommand
 SaveCommand is ICommand
```

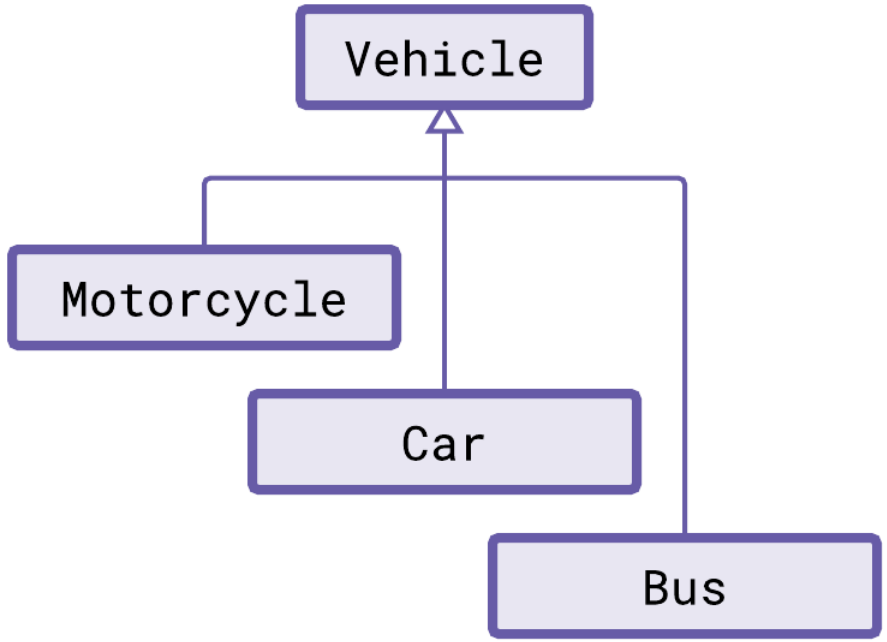
Inheritance Options

— **Interface implementation**
— **Generic derivation**

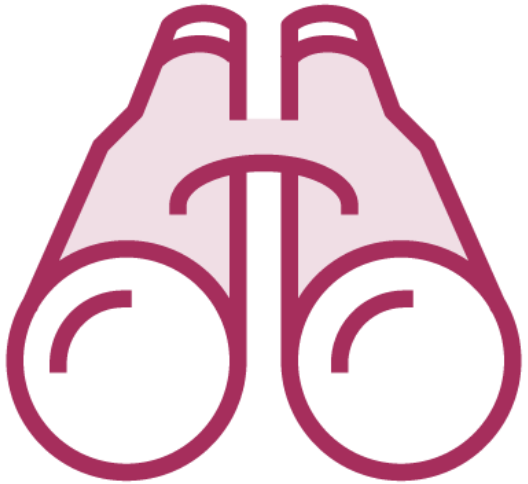


Inheritance Options

— Interface implementation
— Generic derivation
— Class derivation



Lesson Learned



We found a test case missing in prior work

Step 1

Don’t panic

This happens all the time

Step 2

Cover the missing scenario with new tests

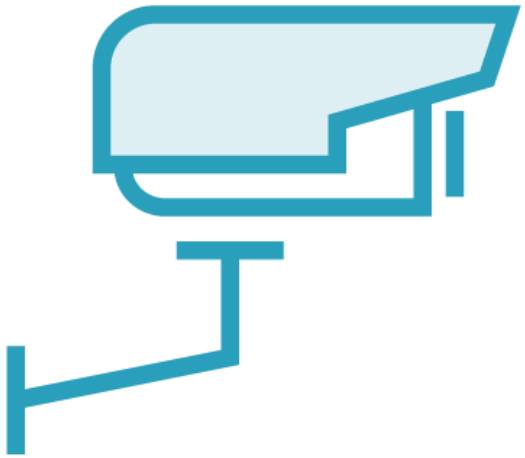
Lesson Learned

Abstract and concrete types communicate both ways

Concrete implementation may give you an idea about abstract type!

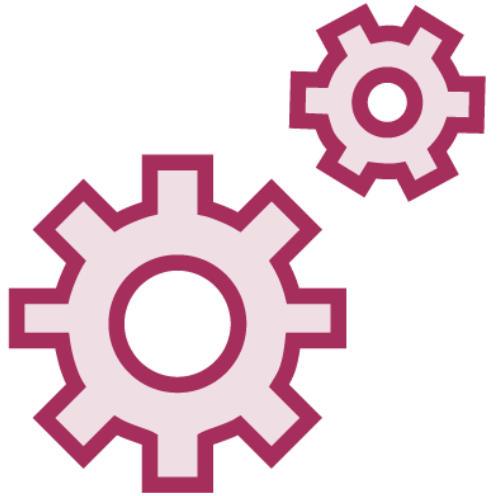
Learn to listen to signals

Safety Net for Refactoring

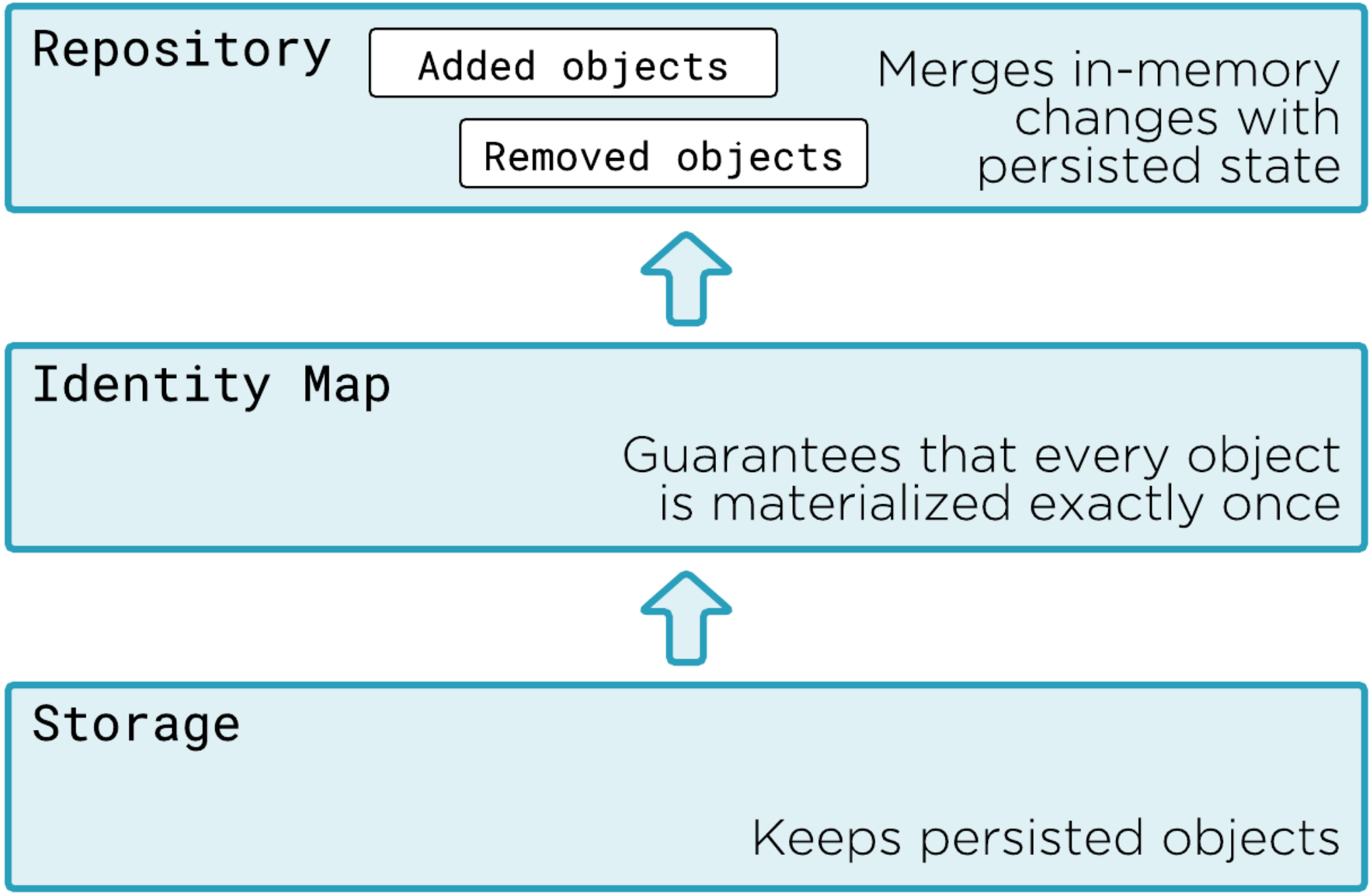


We should be safe to refactor a class which is covered by tests

... but only if we are testing publicly observable behavior



Implementation tests might fail after we made a change


Repository
Added objects
Removed objects
Merges in-memory changes with persisted state
Identity Map
Guarantees that every object is materialized exactly once
Storage
Keeps persisted objects

```
interface IMyList<T>
{
  int Count { get; }

  void Add(T value);



  bool Remove(T value);

}
```

- `Add()` ⇒ `Count` returns value by one larger
- new list
  N×`Add()` ⇒ `Count` returns N
- `Remove()` ⇒ `Count` returns value by one smaller

  `Equals()` override is used

  `IEquatable<T>.Equals()` takes precedence
- `Add(null)` ⇒ `Count` returns value by one larger
- List contains `null`
  `Remove(null)` ⇒ Returns `True`
- List contains no `null`
  `Remove(null)` ⇒ Returns `False`

```
interface IMyList<T>
{
  ...
}

class MyLinkedList<T>
  : IMyList<T>
{
  ...
}

new MyLinkedList<int>();
new MyLinkedList<string>();
new MyLinkedList<Car>();
```

- **Concrete class implementing generic interface**
  This class is still generic
  It cannot produce objects

- **Specify concrete generic argument to define a concrete class**
  Each of these is a new class

```
interface IMyList<T>
{
  ...
}

class MyLinkedList<T>
  : IMyList<T>
{
  ...
}

new MyLinkedList<int>();
new MyLinkedList<string>();
new MyLinkedList<Car>();
```

- **Are we testing a list of T?**
  No – There is no **list of T** class!
  There is a list of ints, strings, etc.

```
interface IMyList<T>
{
  ...
}

class MyLinkedList<T>
  : IMyList<T>
{
  ...
}

new MyLinkedList<int>();
new MyLinkedList<string>();
new MyLinkedList<Car>();
```

- **How do we pick a concrete generic derivation to test?**
  Read the requirements
- **Concrete lists to test**
  Value types – int, some struct
  Reference types – object or string
  Type overriding `Equals()`
  Type implementing `IEquatable<T>`

```
struct AnyStruct
{
  public int Content { get; set; }
}

class AnyClass { }

class AnyWithEquals
{
  public int Content { get; set; }
  public override bool Equals(object obj);
  public override int GetHashCode();
}

class AnyEquatable
  : IEquatable<AnyEquatable>
{
  public int Content { get; set; }
  public bool Equals(AnyEquatable other);
}
```

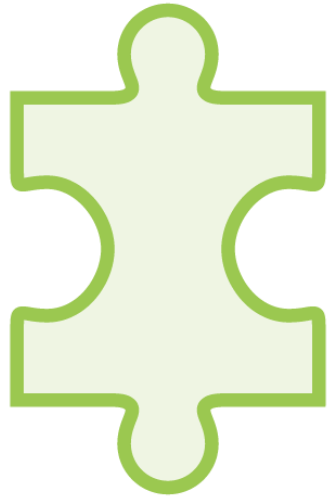
Prepare guinea pigs when testing generic classes

- **Any value type**
  Compiler will fill in the equality comparison logic
- **Any reference type**
- **Class which overrides** `Equals()`
  Contained integer field is used for equality comparison
- **Class implementing** `IEquatable<T>`
  Contained integer field is used for strongly-typed equality comparison

Testing Real-world Interfaces

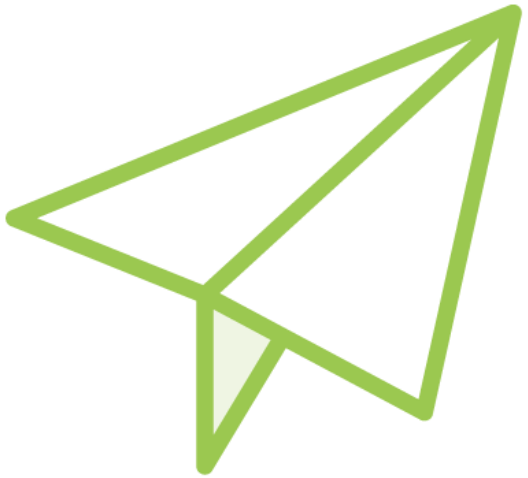
Generic list example is admittedly too general

A lot of test cases to tell that generic list is free of bugs

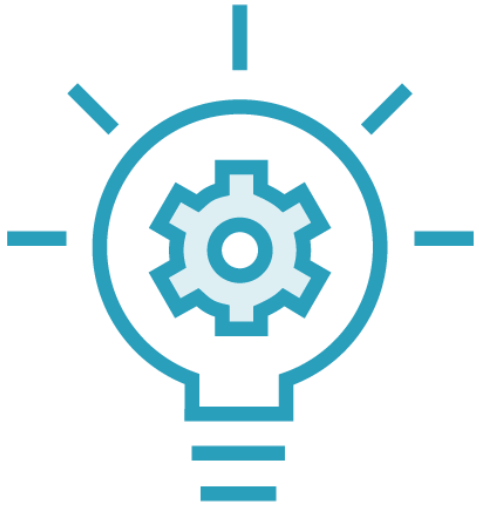


Common business domain interfaces only require a few tests

Testing Real-world Interfaces



Writing tests against an interface improves maintainability



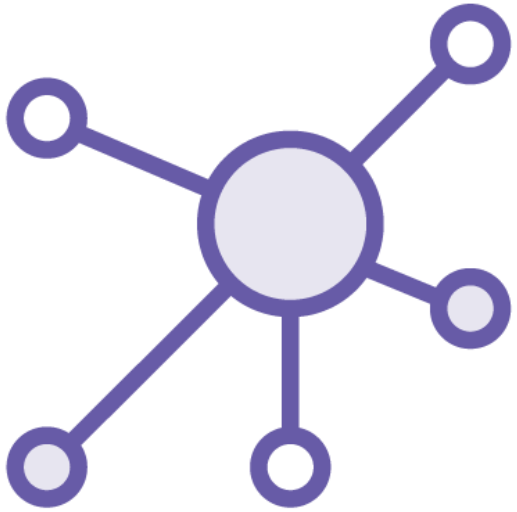
Forces you to test observable behavior

Refactoring and reimplementing doesn't affect tests

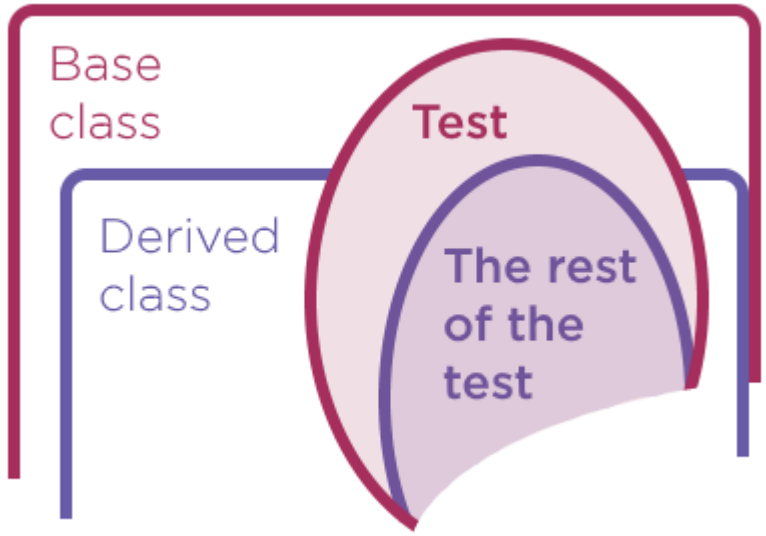
Tests do not deal
with implementation.

Deriving Test Classes: The Antipattern

It's commonplace to say that deriving tests is an antipattern



... but that requires to know how responsibilities are divided



Here's the bad idea:

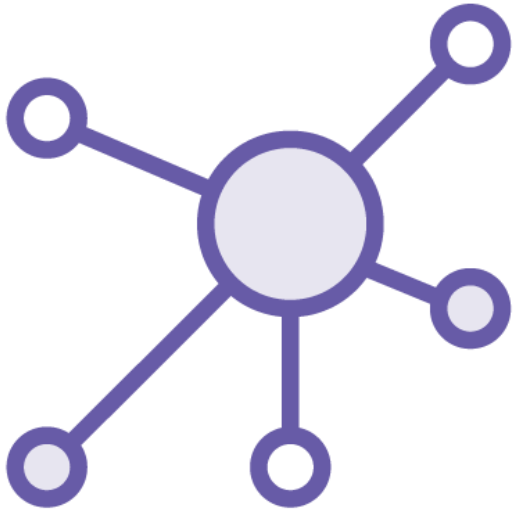
Put part of the test into the base class

Put the rest into the derived class

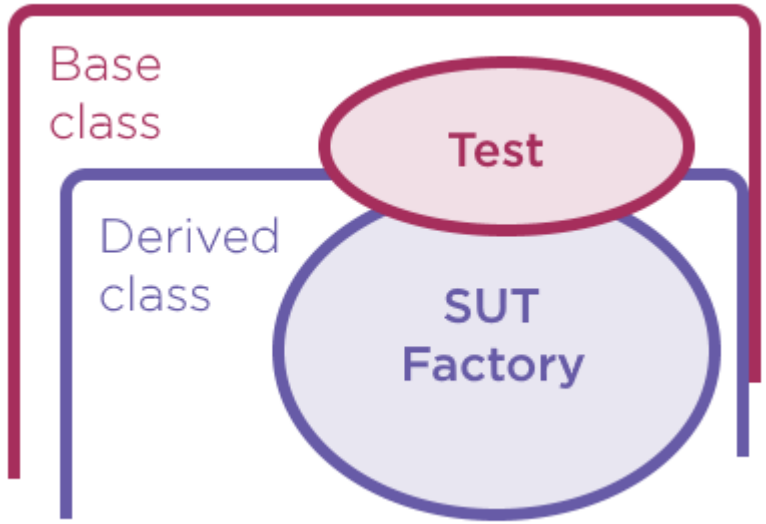
Reading one class doesn't give the whole picture!

Deriving Test Classes: The Antipattern

It's commonplace to say that deriving tests is an antipattern



... but that requires to know how responsibilities are divided



A better idea:

Put entire test in the base class

Specialize SUT factory in the derived class

Each class gives the whole picture on one responsibility

Do not split a single test between base and derived test classes.

Summary

Diving deeper on Abstract Data Types

- ADTs fit together with TDD
- Write tests against ADT first
- Implement concrete class which satisfies tests

Example

- Implementing and then refactoring a concrete repository class
- Unit tests used to define abstract repository

Summary

Writing unit tests for generic classes

- Generic derivation only happens at run time
- Generic class can produce many different classes
- It's pointless to write tests for many different generic derivations
- Write tests for indicative generic derivations as per requirements

Next module:
Testing entire design principles