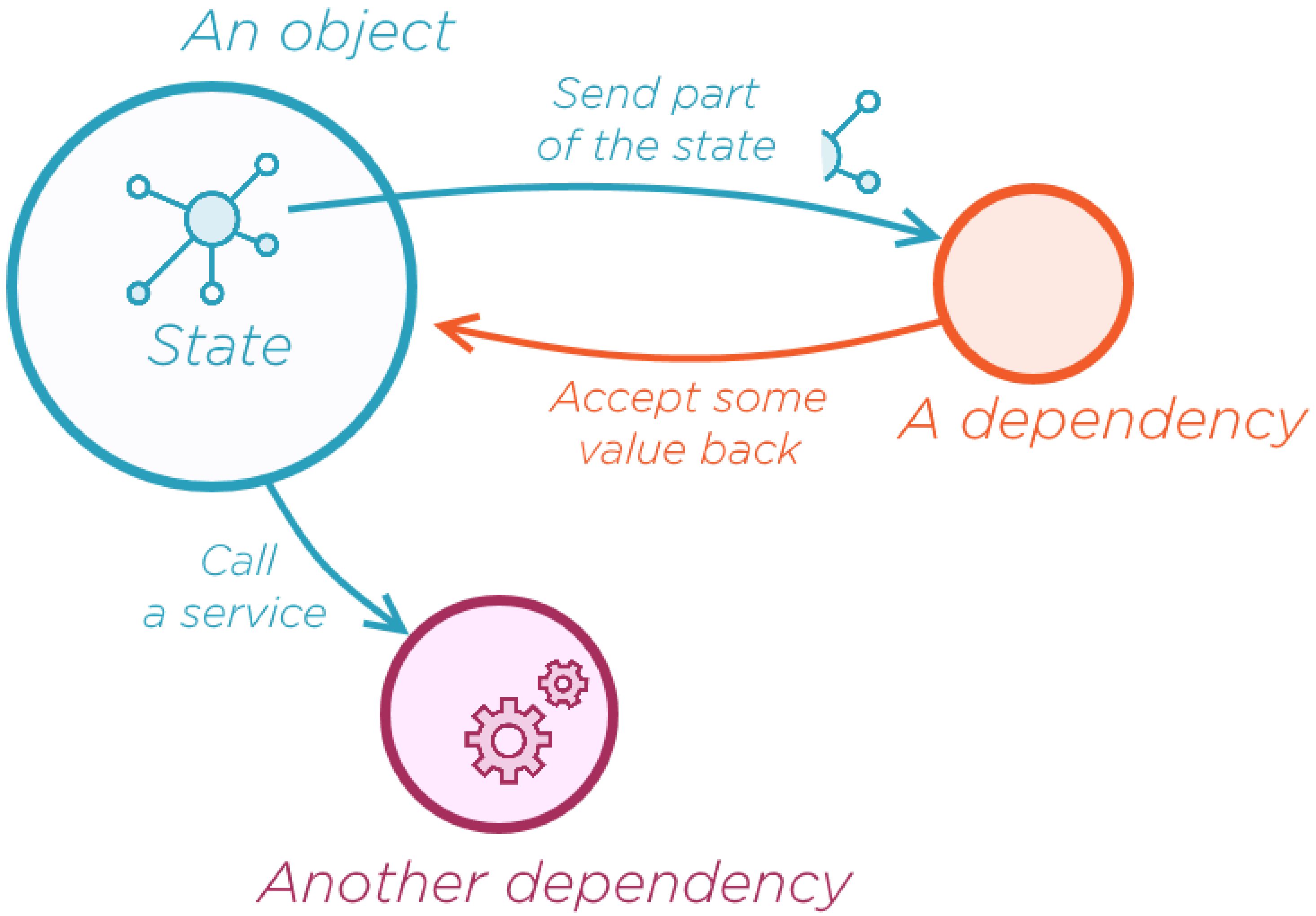
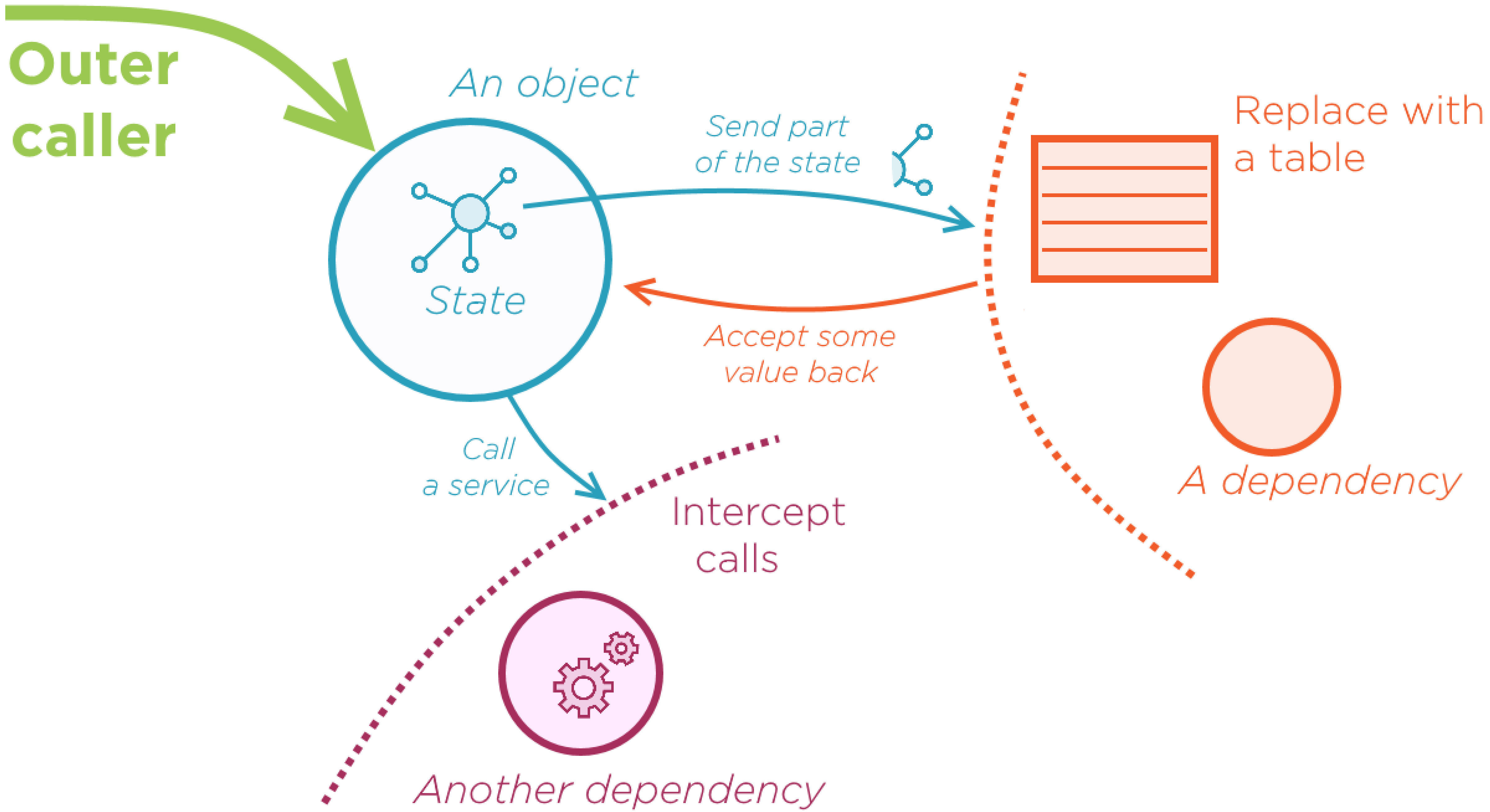
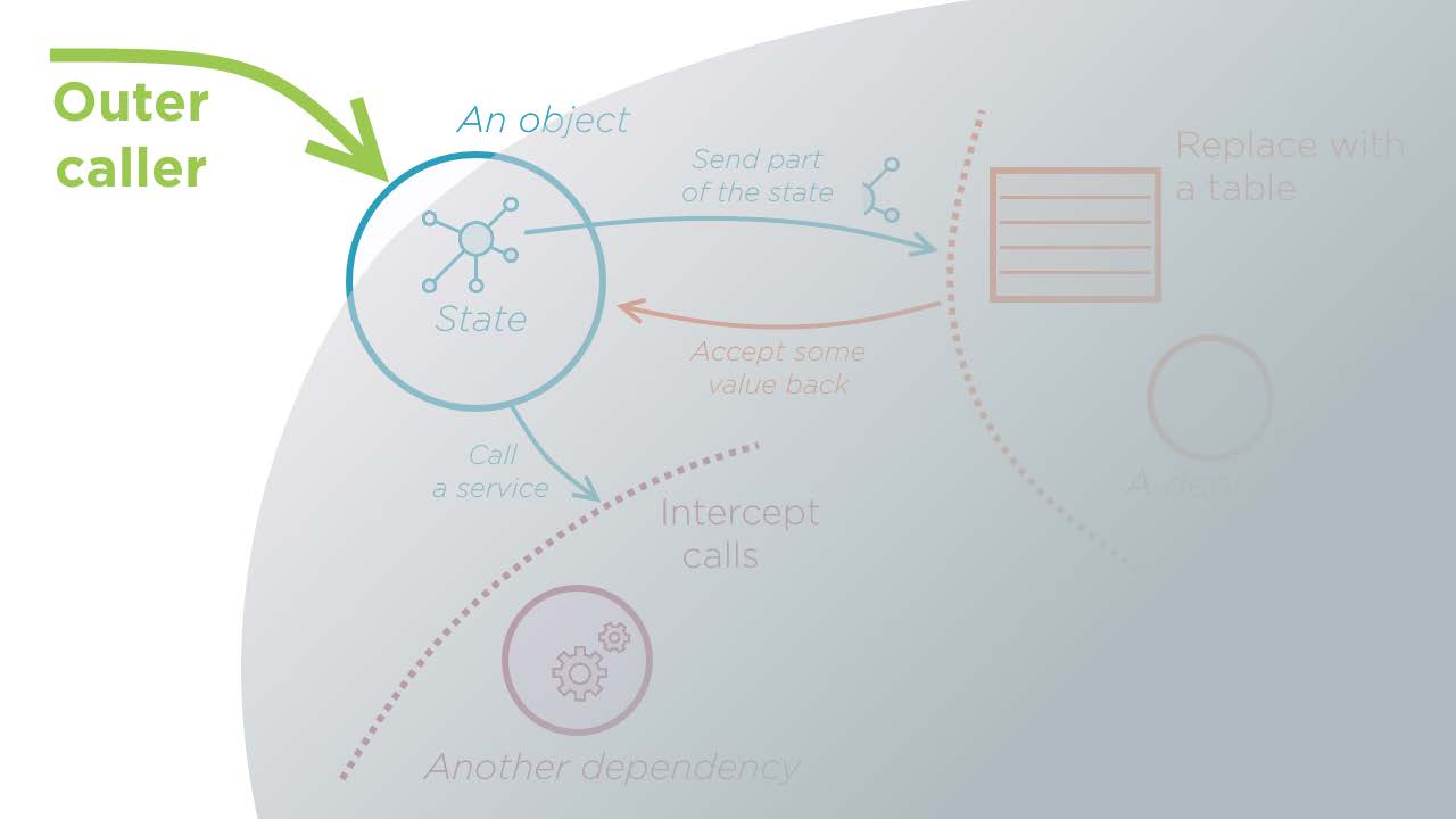
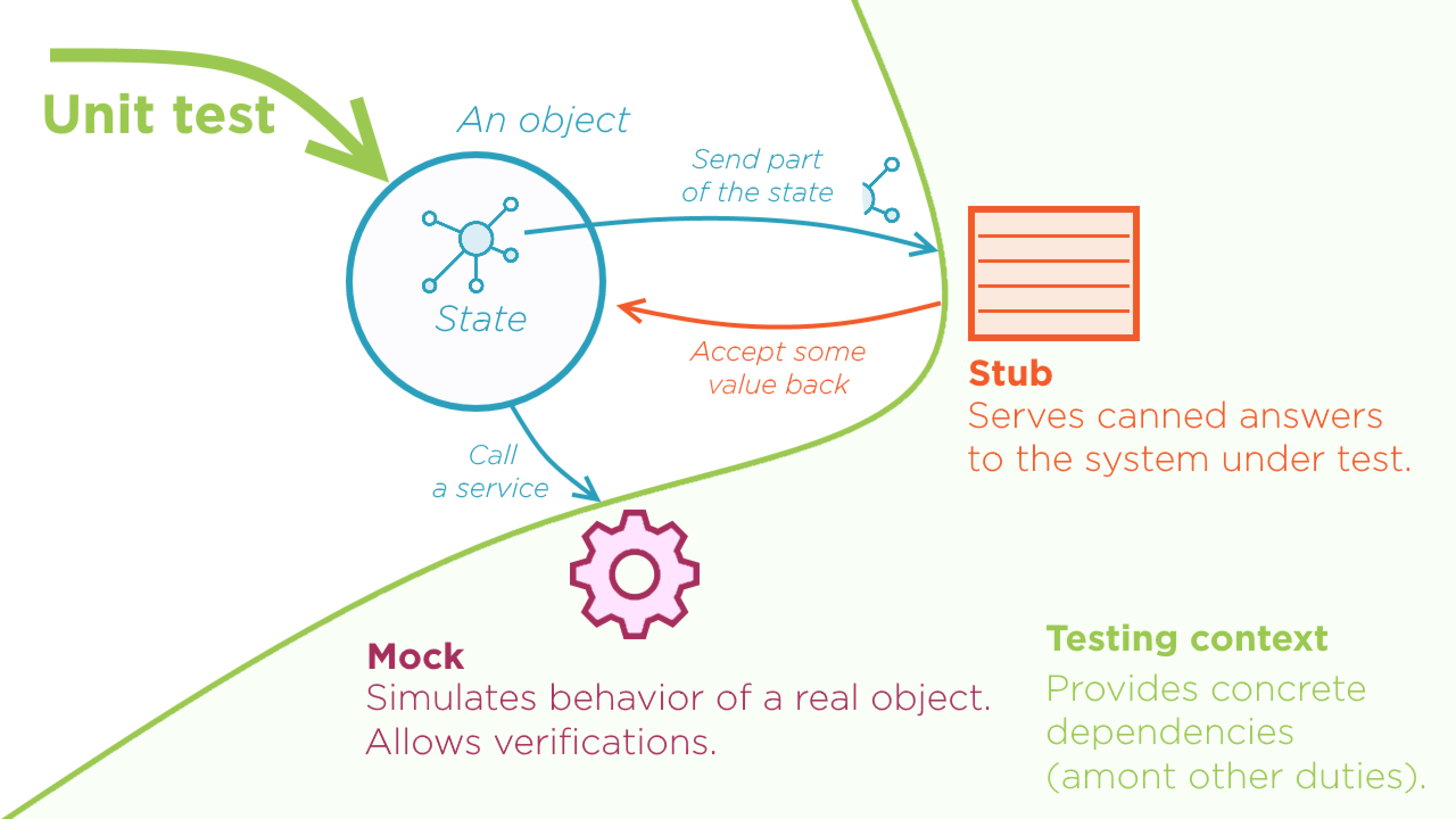
Modeling Class Dependencies in Tests

Zoran Horvat
PRINCIPAL CONSULTANT AT CODING HELMET
@zoranh75 csharpmentor.com


An object
State
Send part of the state
Accept some value back
A dependency
Call a service
Another dependency


Outer caller
An object
State
Send part of the state
Replace with a table
Accept some value back
A dependency
Call a service
Intercept calls
Another dependency


Outer caller
An object
State
Send part of the state
Replace with a table
Accept some value back
Call a service
Intercept calls
Another dependency



Stub
Serves canned answers to the system under test.

Mock
Simulates behavior of a real object. Allows verifications.

Testing context
Provides concrete dependencies (amont other duties).

Stubs vs. Mocks

Test double should either be a stub or a mock, but not both

Unit test should contain at most one mock

Make assertions against mocks

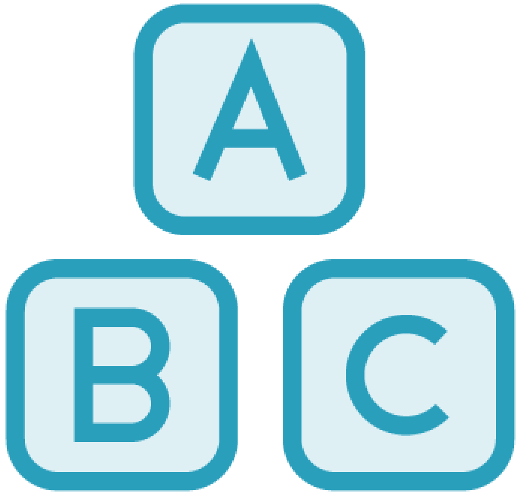
Therefore keep only one of them

Stubs vs. Mocks

Use mock to measure interactions with dependencies

Use stubs to make dependencies invisible to the class under test

Stubs vs. Mocks



Stubs are carrying little responsibilities

They are making tests easy to maintain

Mocks are carrying complex responsibilities

They are making tests rigid

Testing Effects of Dependencies

Dependencies affect publically observable behavior

Dependency misuse can be observed

Intercepting interaction with a dependency is too rigid

Class should be free to choose concrete implementation

Testing Effects of Dependencies

State-based tests usually rely on stubs

Interaction tests usually rely on mocks

Interaction tests are useful when interaction is required

Interaction tests are more brittle

Don't use them unless necessary

Where Are Dependencies Coming From?

Dependencies can grow out from the class

We love this case

We have control over dependency's interface

Sometimes we depend on a pre-existing class

Negative Tests

They obscure the fact that other (positive) tests are missing

Avoid tests that are proving that something did not happen

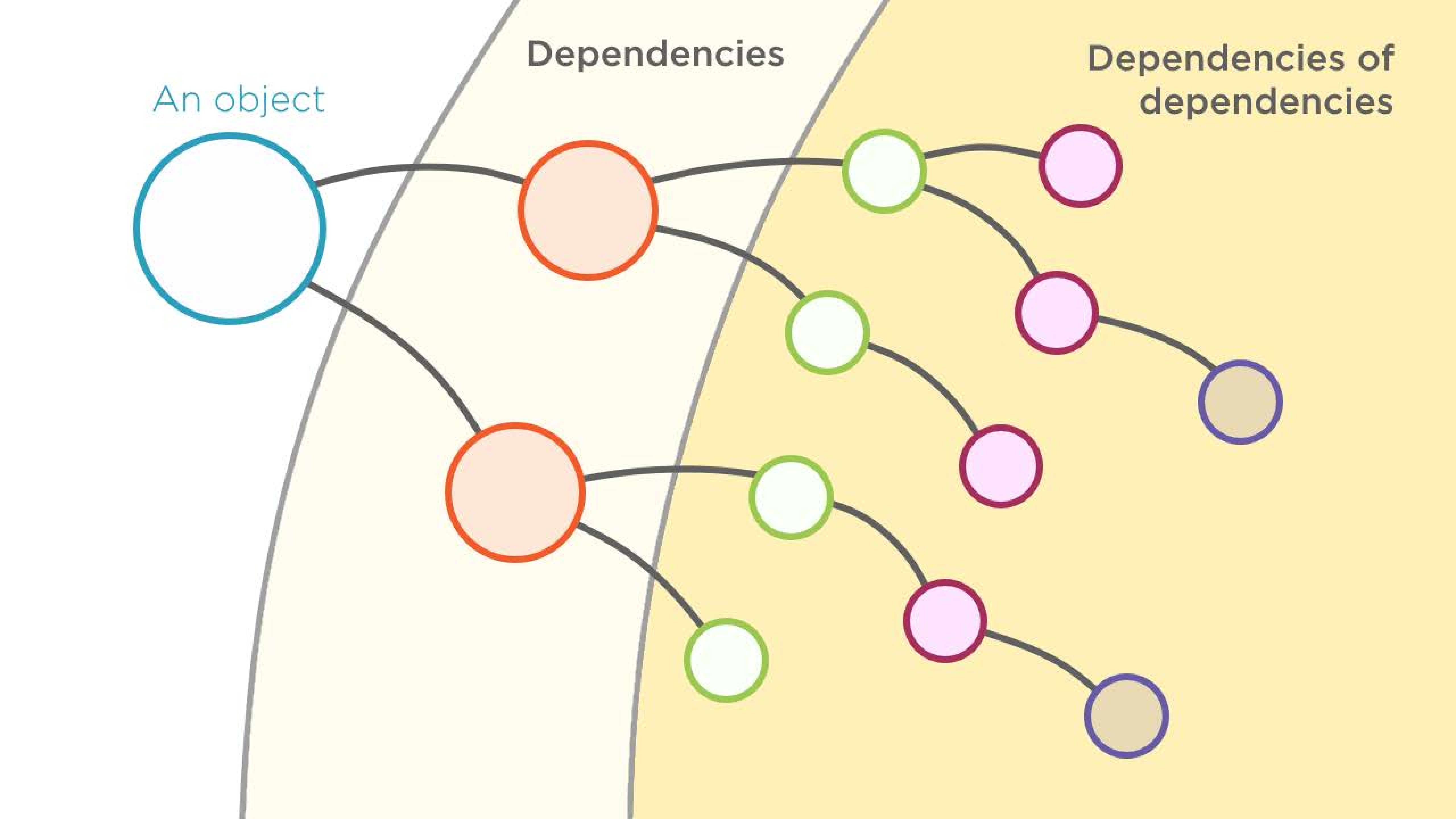
You will never cover all events that did ***not*** happen...

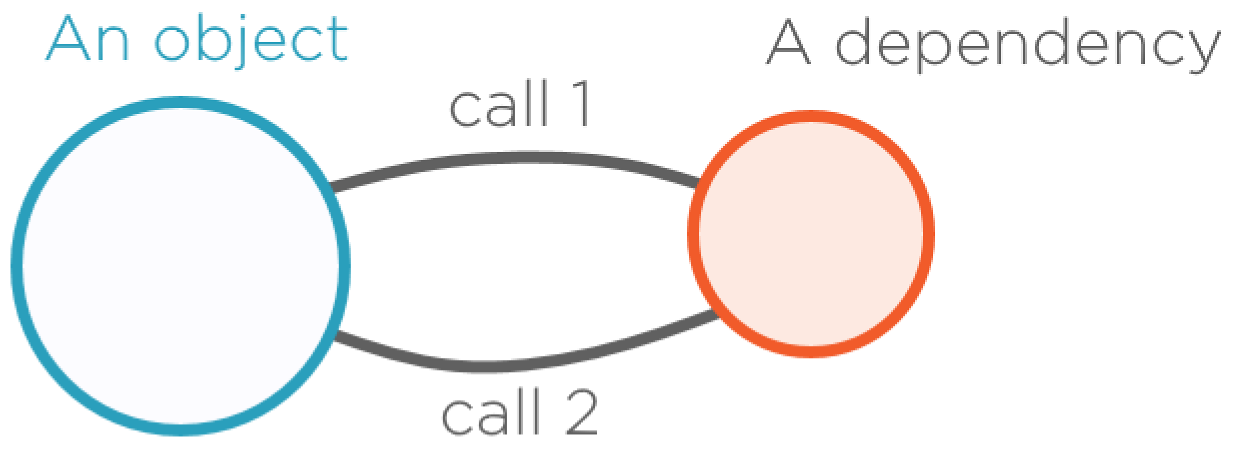
Positive Tests

Assert that expected behavior did happen

All required changes must truly happen under test

That naturally rules out defense against unexpected behavior

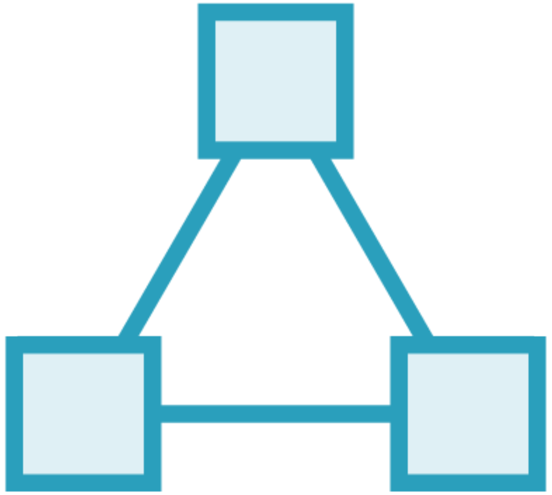

An object
Dependencies
Dependencies of dependencies



Temporal coupling

call 1 must precede call 2 or otherwise execution will not be correct.

Temporal Coupling



In theory
It is possible to remove temporal coupling from public interface

In practice
Insisting on removal of temporal coupling is very difficult

Disposable Pattern as Temporal Coupling

1. Perform an operation
2. Call Dispose()

1. Perform an operation
2. Operation disposes internally

We cannot combine operations if they dispose the object

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

interface IRepository<T> :
  IDisposable
{
  IEnumerable<T> GetAll();
  void AddAndSave(T obj);
}
```

- **Interface with temporal coupling**

  Dispose() must be called after other methods

  Save() must be called after Add()

- **Without temporal coupling on** Save()

  Now we can only add one object

  That limits use of the repository

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

- **Interface doesn’t communicate temporal coupling**

  Nothing says that Save() *must* be called after Add()

- **Temporal coupling is the restriction**

  We don’t have to call Save() at all

  Only Save() *must not* be called before Add()

- **Test case for temporal coupling**

  Save() not called before Add()

- **Compare to not-disposed test**

  That was an imaginary requirement

  Dispose() doesn’t say it must not be called

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

- **Test case for temporal coupling**

  $$t_{Add}$$

- **Any sequence of calls that satisfies this constraint is fine**

  We can vary implementation and still keep the code correct

Summary

Impact of dependencies on tests

- Dependencies may grow out from a class while designing and refactoring
- Dependencies may be the existing classes that are useful to current class

Growing through refactoring

- Depending class controls interface of its dependency
- Tests will probably be state-based because that makes them easy for us

Depending on existing classes

- Better to depend on abstract interface
- Dependency behavior must be defined
- Possibility of interaction tests

Summary

Don’t get too defensive with tests

- Cover positive use cases
- Tests must fail if implementation breaks rules

Temporal coupling

- Special case of negative test cases
- Generally considered code smell
- And still widely applied…
- Add test cases that verify that temporal coupling was not violated

Next module:
Testing Abstract Data Types