Dependency Injection, Testing, and Mocks

SCOBIE SMITH

DECEMBER 1, 2021

Dependency injection means… never having to say "new"

- When a method needs an external dependency, such as a database, service, etc., ***don't*** do this:

```
Service myService = new Service(arg1);
```

… which hardcodes the creation of the dependency, making isolated unit tests of this code impossible.

- Instead, pass in an interface to the constructor—***inject*** it:

```
public UsefulClass(IService myService);
```

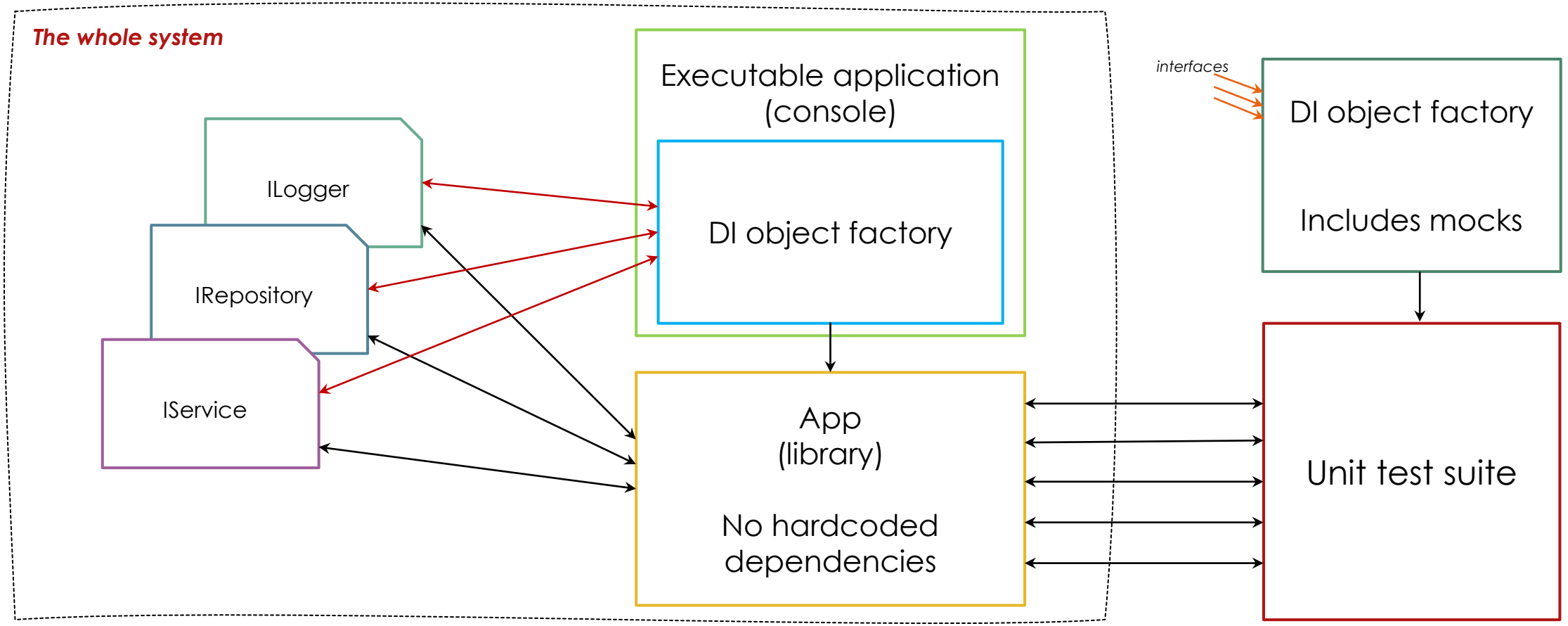
- And then *register a binding* in the DI container to specify how this IService is to be instantiated, when UsefulClass is created.

Injection is a chain reaction of object resolution

- When a binding is resolved, by getting the implementation specified, then the constructor of this class is called...
- This means those constructor-injected interfaces are resolved in turn, injecting these objects into the newly instantiated class.
- And each time a binding is resolved to create an object, this process is repeated, injected the objects required by the constructor.
- It is the DI container that maintains all these mappings (registrations, bindings).
- The chain reaction starts at the top level, e.g., when the App is resolved.

Dependency Injection

… factoring out how objects are instantiated



- **The app library has no hardcoded calls to new for external dependencies. So, it is fully unit-testable.**
- **The app library also has no direct reference to the DI container or how the dependencies are created.**
- The factory for object creation is separated into code outside the library, in the application executable.
- So, the application exe code holds DI container and any factory code, which might include use of new.
- But the app's real functionality is in the library, which now is testable.
- The dependencies are usually referenced via interfaces.
- **The unit test suite swaps in its own DI container with mocks for the external dependencies.**

Adding dependency injection

- Possible to introduce DI piecemeal
  - Start with new code
  - A chosen, existing component could be decoupled
  - Progressive introduction: leave new in place until ready
  - Add registrations to the object factory before applying these in constructor injection

DI frameworks explored in the sample code

- SimpleInjector
- LightInject
- Autofac
- Unity
- Ninject
- Grace

- There is a Word doc discusses more details of comparison, features, and evaluation of these.

My preferred choices…

- Yes: syntax, speed, simplicity, features
  - SimpleInjector
  - LightInject
- Maybe or ok: so so syntax, full featured, more complex
  - Autofac
- Probably not use, because showing its age
  - Unity
  - Ninject (
- No way: painful syntax, lack of docs/help, features?
  - Grace

The demo code...

- Intended to exemplify some uses relevant to us
- App.cs: Shows the simple application...
  - Get orders from service
  - Do logging
  - Execute transaction of the order
- Order grabs product from Product Repo
- Two kinds of logs:
  - Simple text to console
  - Write to database
- Transaction derived classes based on category of product
  - Computer, Peripheral, Storage
- Factories...
  - To make Order
  - To make Transaction

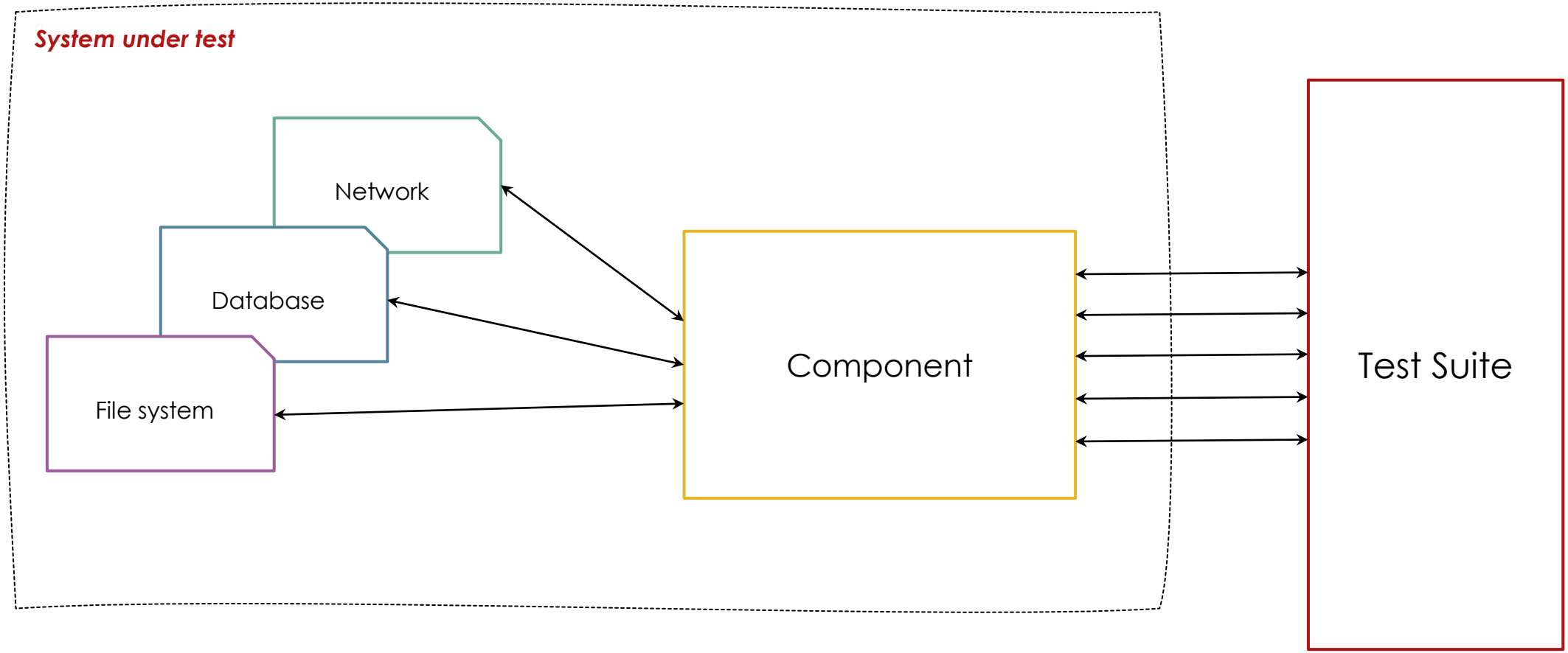
Unit testing… requirements for the “ideal”

- A *unit* test should test only the operation of the component in question, not all the external systems that is uses.
- True unit testing requires designing components to be testable.
- A component needs to be able to be isolated from external dependencies.
- For example, the problem arises when we directly instantiate external systems with C# new—hardcoding the dependency.
- Decoupling the component requires use of interfaces and dependency injection.

Decouple the system under test

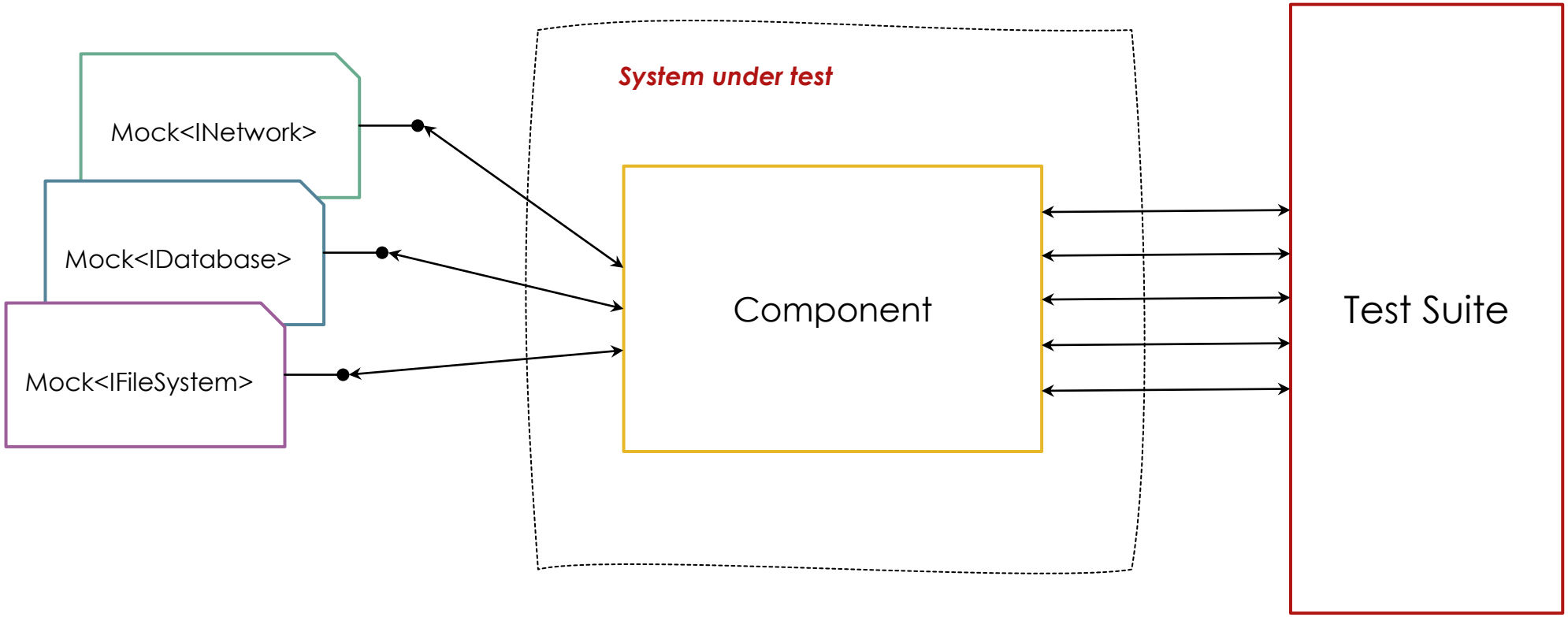
- Use dependency injection to achieve decoupling.
- External dependencies must be expressed by interfaces.
- Normal operation: Pass in instance implementing the interface.
- Test situation: Pass in a mock implementation of the interface.
- If an external system does not provide interfaces, it is necessary to create a wrapper/façade (with interfaces) to encapsulate it.

Not a true unit test…
… but an integration test



- System under test includes external systems, not just the component we are trying to test.
- The tests are also exercising the back-end dependencies, not just the component.
- Therefore, this is not true unit test set-up, but an integration test.
- The external systems here are not represented by interfaces, making it harder to isolate the component.

Real unit testing…
… using interfaces and mocking



- The external systems define their APIs with interfaces.
- These systems can be replaced with mock (fake) implementations, which define API responses for test purposes.
- The actual external systems are not involved in the test at all.
- The component receives an implementation of the interface, rather than instantiating the object (new).
- The component always uses the interface API, whether it is a mock or the real external system.

What really is a “mock”?

- An object that fakes an implementation of its target
- It implements the same interface as the original
- This assumes there is a public interface
  - Moq requires at least public virtual methods
- Only the needed methods or properties are given implementations
- The implementations are just test results at most
- After a test is run, it is possible to call a verify method on the mocks
  - To verify that the expected method was called
  - Count how many calls, etc.
- Examples in demo code

Different issues to distinguish: bad

- Bad: Overkill with complex (error-prone), superfluous unit tests.
- Bad: Hard-to-maintain unit tests…
  - Hard to add test data
  - Hard to read
  - Hard to know they test what we intend

Different issues to distinguish: good

- Good: Unit tests that are relatively “brittle”.
  - Not too smart to pass under changing circumstances
  - If in doubt, more likely to fail
- Good: Infrastructure to create unit tests easily.
  - Do not need to create tons of set-up code each time
  - Can add and use mocks easily
- Good: Testable code—designed for unit testing.
  - Mocking is necessary, hence dependency injection
  - Possible with good design patterns

Cost considerations

- Loss: Good unit testing adds overhead (developer time).
  - Create the infrastructure (mostly one-time, though adding mocks).
  - Design code for testability while being readable, maintainable.
  - Create unit tests (prudent choice of code coverage).
- Gain: Good unit testing can reduce other testing.
  - Integration testing could focus purely on integration of components.
  - Manual testing could do the same (if not already).
- Gain: Good unit testing reduces bugs.
  - Catch my bugs during development (if generally TDD).
  - Push-button regression catches new “unrelated” bugs.

Short-term vs. long-term: bugs

- Hours before vs. after development/deployment.
  - Spend more hours afterward fixing more bugs.
  - Spend more hours beforehand writing tests (fixing fewer bugs).
- Hidden cost of bug fixing…
  - Bugs are by nature unpredictable and thus introduce risk—time, impact.
  - Fixing bugs is unpredictably harder than writing tests—diverting devs.
  - Meanwhile, planned development is also delayed.
  - Debugging is a skill/art, but not a well-defined discipline (skillset)—less potential to grow reusable expertise.

Short-term vs. long-term: testing

- Hidden gain of good unit testing…
  - Unit testing can reduce risk.
  - Accumulation of unit tests adds increasing value to regression capability.
  - Reusable skillset grows in the process, speeding up test development.
- But gaps remain with unit testing.
  - Still get bugs (integration, hard to automate, etc.).
  - Environment-configuration bugs.
  - Consider other automated testing for these issues (functional).