

# Write tests you love, not hate

**Jens Happe**

Head of Software Engineering - Chrono24  
Co-Founder - Sparkteams



What does the following test check?

---

# What does this test check?

```
@Test
void testRegisterNewUser() {
    when(userRepository.save(any(User.class))).thenReturn(invocation -> {
        User user = invocation.getArgument(index: 0);
        user.setId(1);
        return user;
    });

    when(activationCodeGenerator.next()).thenReturn(value: "abc-123");

    userService.registerUser(email: "hans.wilsdorf@rolex.com", name: "Hans Wilsdorf");

    ArgumentCaptor<User> userArgumentCaptor = ArgumentCaptor.forClass(User.class);
    verify(userRepository, atLeastOnce()).save(userArgumentCaptor.capture());
    User savedUser = userArgumentCaptor.getValue();
    assertThat(savedUser).hasFieldOrPropertyWithValue(name: "id", value: 1L)
        .hasFieldOrPropertyWithValue(name: "email", value: "hans.wilsdorf@rolex.com")
        .hasFieldOrPropertyWithValue(name: "name", value: "Hans Wilsdorf")
        .hasFieldOrPropertyWithValue(name: "activationCode", value: "abc-123");

    ArgumentCaptor<Email> emailArgumentCaptor = ArgumentCaptor.forClass(Email.class);
    verify(emailService).send(emailArgumentCaptor.capture());
    Email sentEmail = emailArgumentCaptor.getValue();
    assertThat(sentEmail).hasFieldOrPropertyWithValue(name: "recipient", value: "hans.wilsdorf@rolex.com")
        .hasFieldOrPropertyWithValue(name: "subject", value: "Please confirm your email address")
        .hasFieldOrPropertyWithValue(name: "body", value: "To activate your account click the following link: https://chrono24.com/users/1/activations/abc-123");
}
```

# What does this test check?

```

@Test
void testRegisterNewUser() {
    when(userRepository.save(any(User.class))).thenReturn(new User(1L, "hans.wiltsdorf@rolex.com", "Hans Wiltsdorf", "abc-123"));
    userService.registerUser("hans.wiltsdorf@rolex.com", "Hans Wiltsdorf");

    ArgumentCaptor<User> userArgumentCaptor = ArgumentCaptor.forClass(User.class);
    verify(userRepository, atLeastOnce()).save(userArgumentCaptor.capture());
    User savedUser = userArgumentCaptor.getValue();
    assertThat(savedUser).hasFieldOrPropertyWithValue("id", 1L)
        .hasFieldOrPropertyWithValue("email", "hans.wiltsdorf@rolex.com")
        .hasFieldOrPropertyWithValue("name", "Hans Wiltsdorf")
        .hasFieldOrPropertyWithValue("activationCode", "abc-123");

    ArgumentCaptor<Email> emailArgumentCaptor = ArgumentCaptor.forClass(Email.class);
    verify(emailService).send(emailArgumentCaptor.capture());
    Email sentEmail = emailArgumentCaptor.getValue();
    assertThat(sentEmail).hasFieldOrPropertyWithValue("recipient", "hans.wiltsdorf@rolex.com")
        .hasFieldOrPropertyWithValue("subject", "Please confirm your email address")
        .hasFieldOrPropertyWithValue("body", "To activate your account click the following link: https://chrono24.com/users/1/activations/abc-123");
}

```

Set the next user id

Set the next activation code

Register new user

Verify user saved

Verify email sent

Configuration of mocked objects

Call to component under test (CUT)

Verification

# Common Problems with Unit Testing

## Too much boilerplate code

- Tests are hard to understand
- Writing tests is a lot of effort

## The *Fragile Test Problem*

- Adjusting tests after a minor change takes two days
- Tests generate too many false positives

## Tests are running too long

→ **No one likes writing tests**

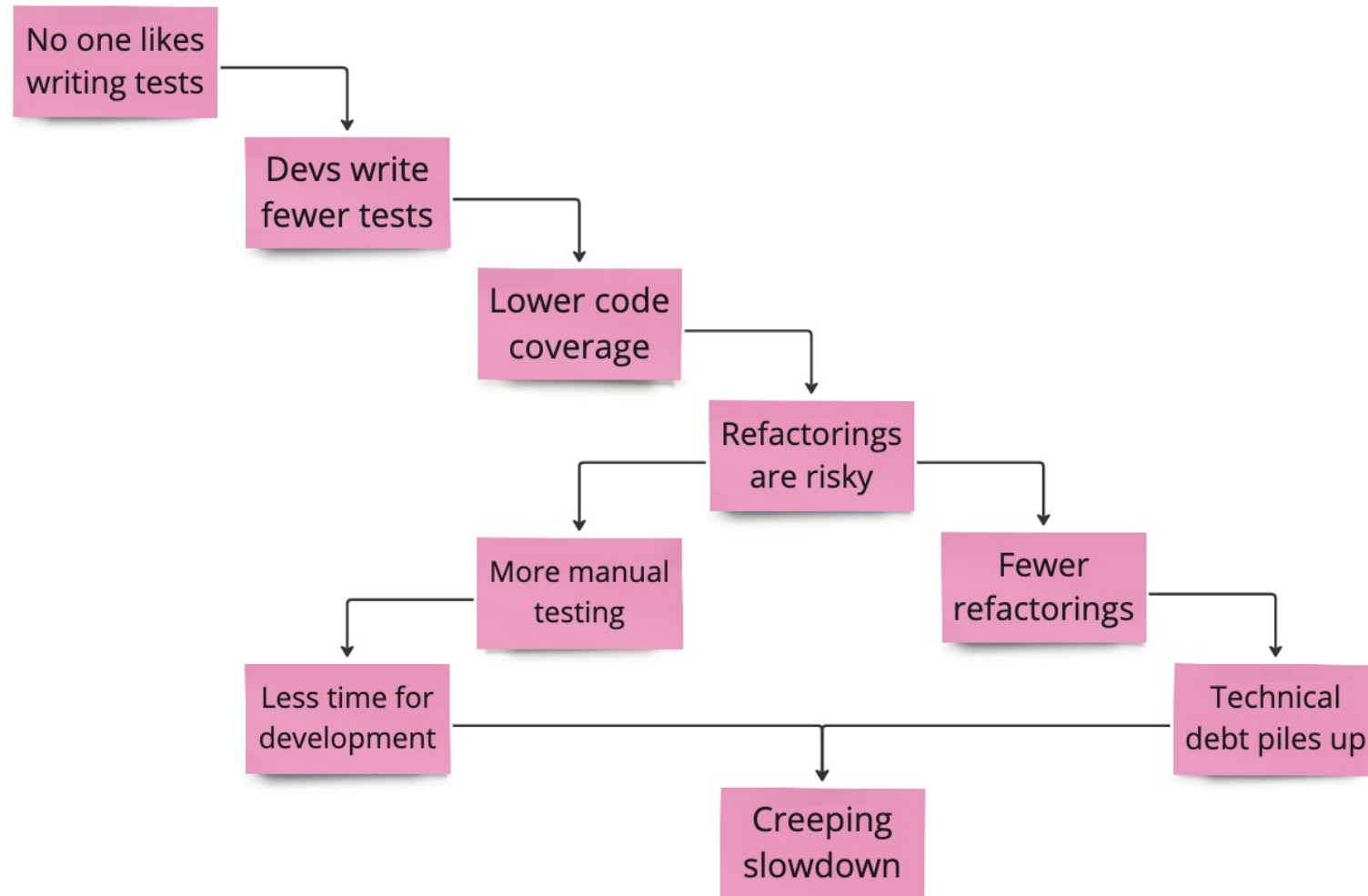


<https://imgs.xkcd.com/comics/compiling.png>

# Okay, no one likes writing tests. So what?!

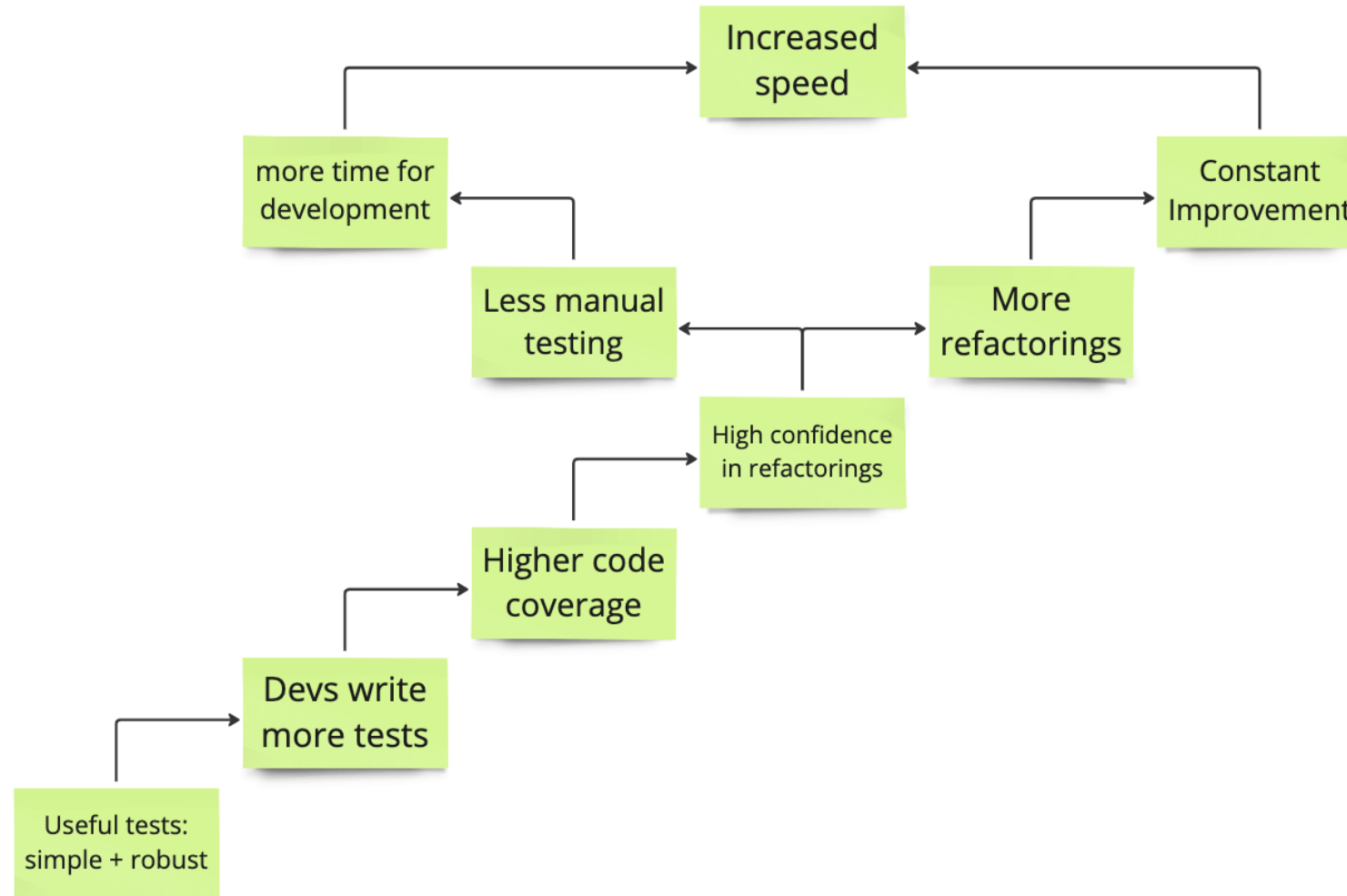


The consequences of a **bad** test structure (simplified)



# What if...

The consequences of a **good** test structure (simplified)



# Some facts and figures about us



**9.000.000**

website visitors monthly

**8.000.000**



app users

**129**

people in product and tech

**10**

scrum teams

26	 <b>whatnot</b> Whatnot	Shopping
27	 <b>Chrono24</b> Chrono24	Shopping
28	<b>GET YOUR GUIDE</b> Get Your Guide	Travel

<https://future.a16z.com/marketplace-100/>

← First non-US company on the list



# Common Problems with Unit Testing

## Too much boilerplate code

- Tests are hard to understand
- Writing tests is a lot of effort

## The *Fragile Test Problem*

- Adjusting tests after a minor change takes two days
- Tests generate too many false positives

## Tests are running too long

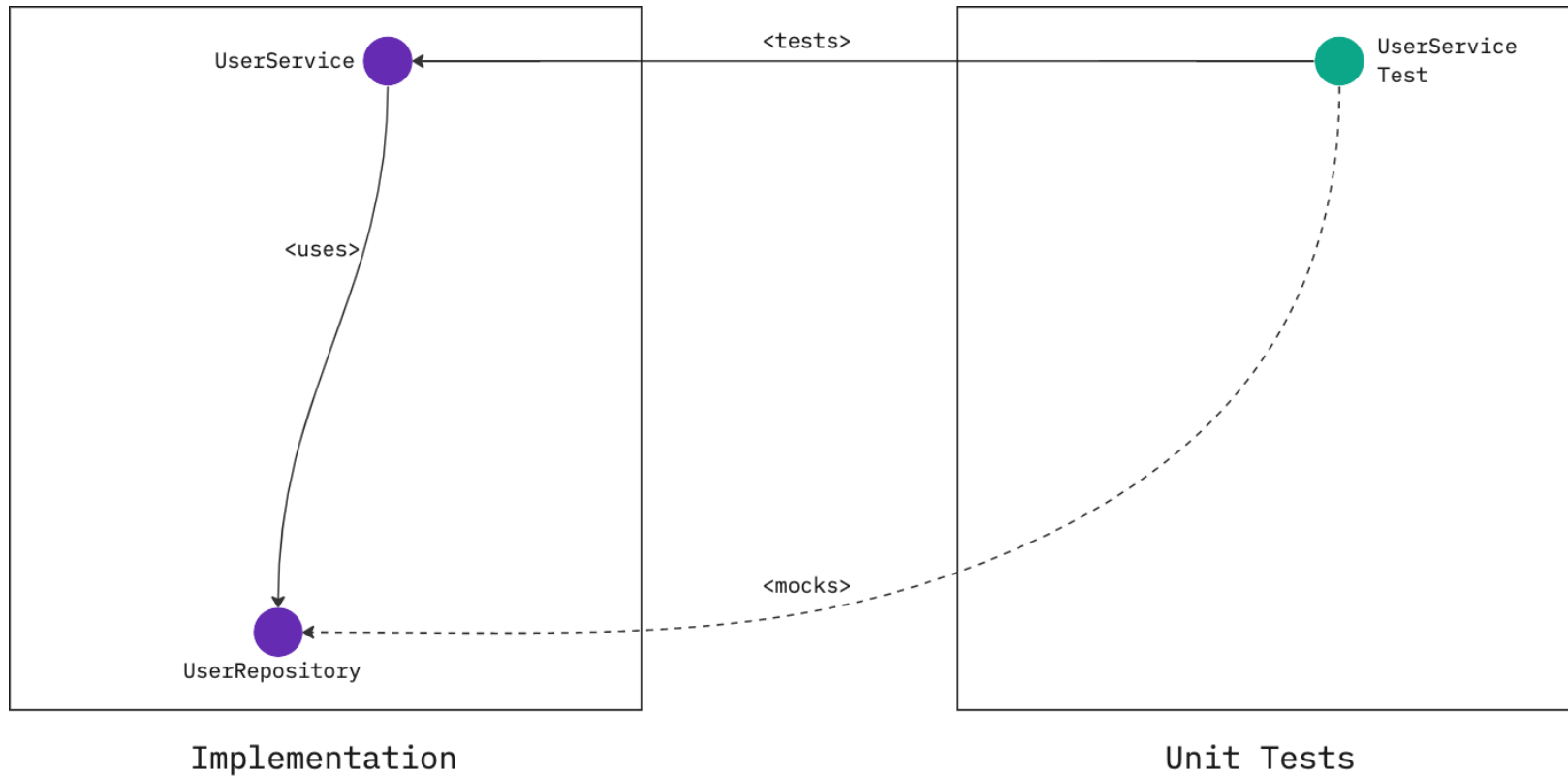
→ **No one likes writing tests**



<https://imgs.xkcd.com/comics/compiling.png>

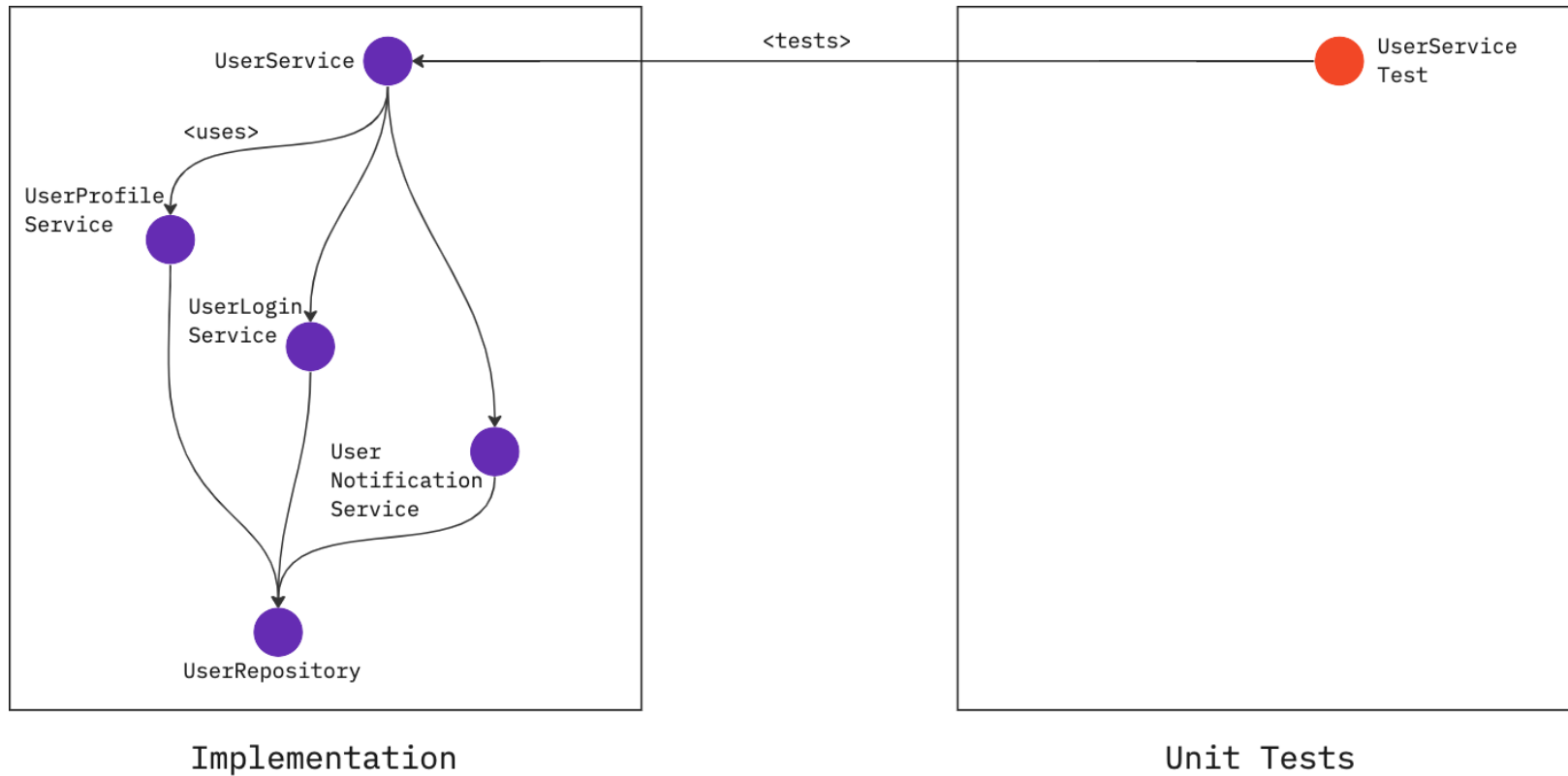
# Fragile Test Problem

## Definition



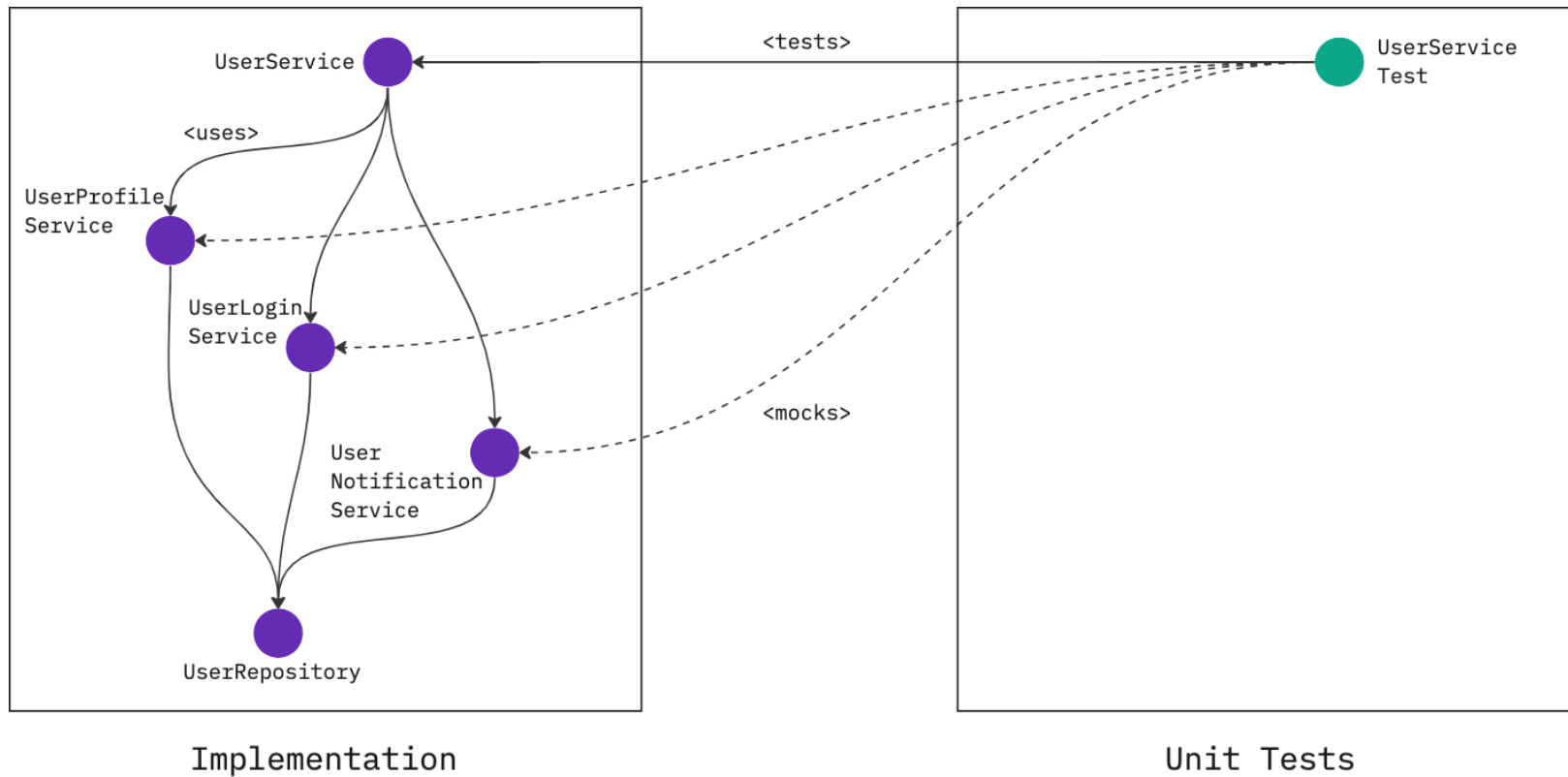
# Fragile Test Problem

## Definition



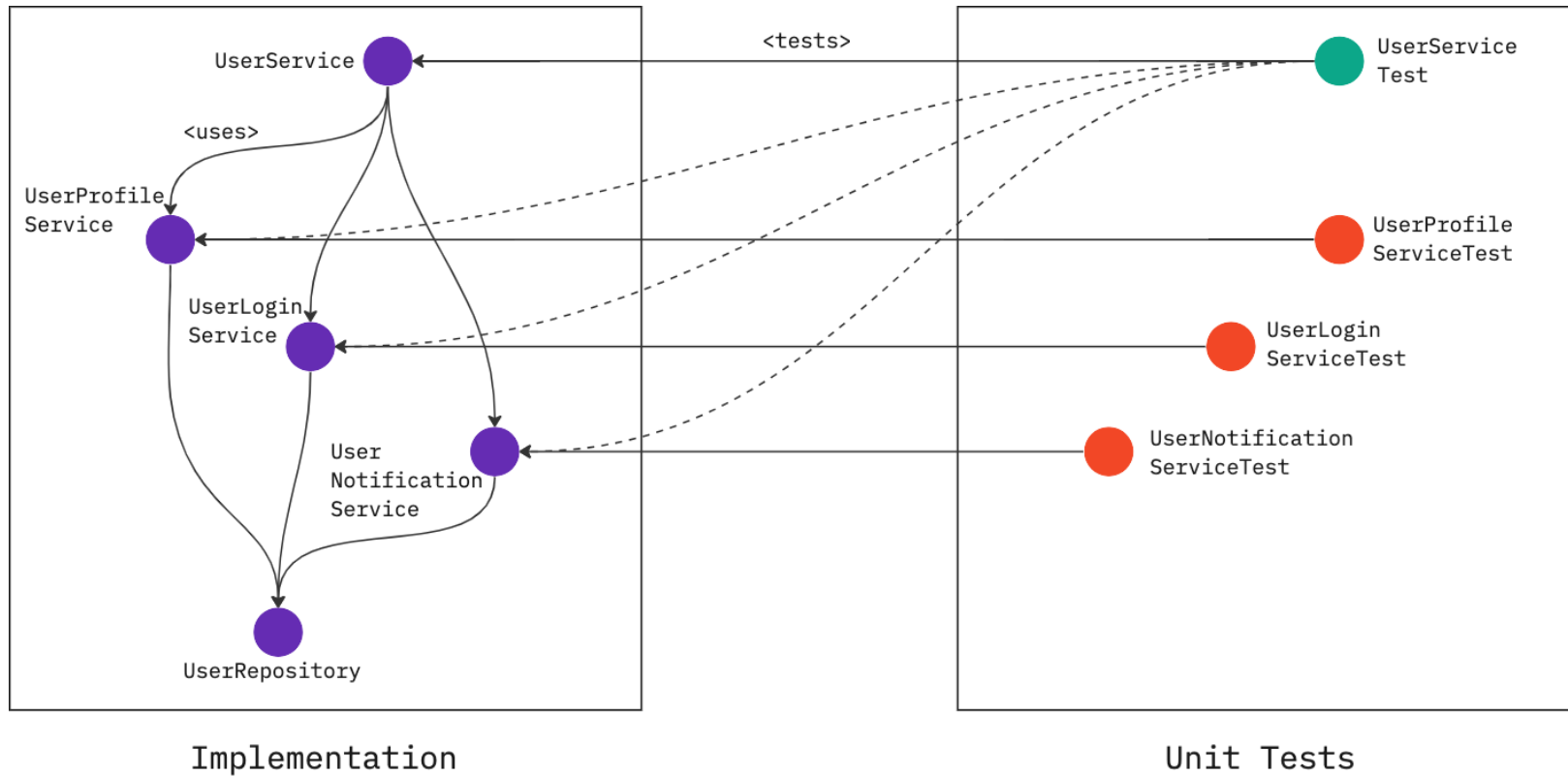
# Fragile Test Problem

## Definition



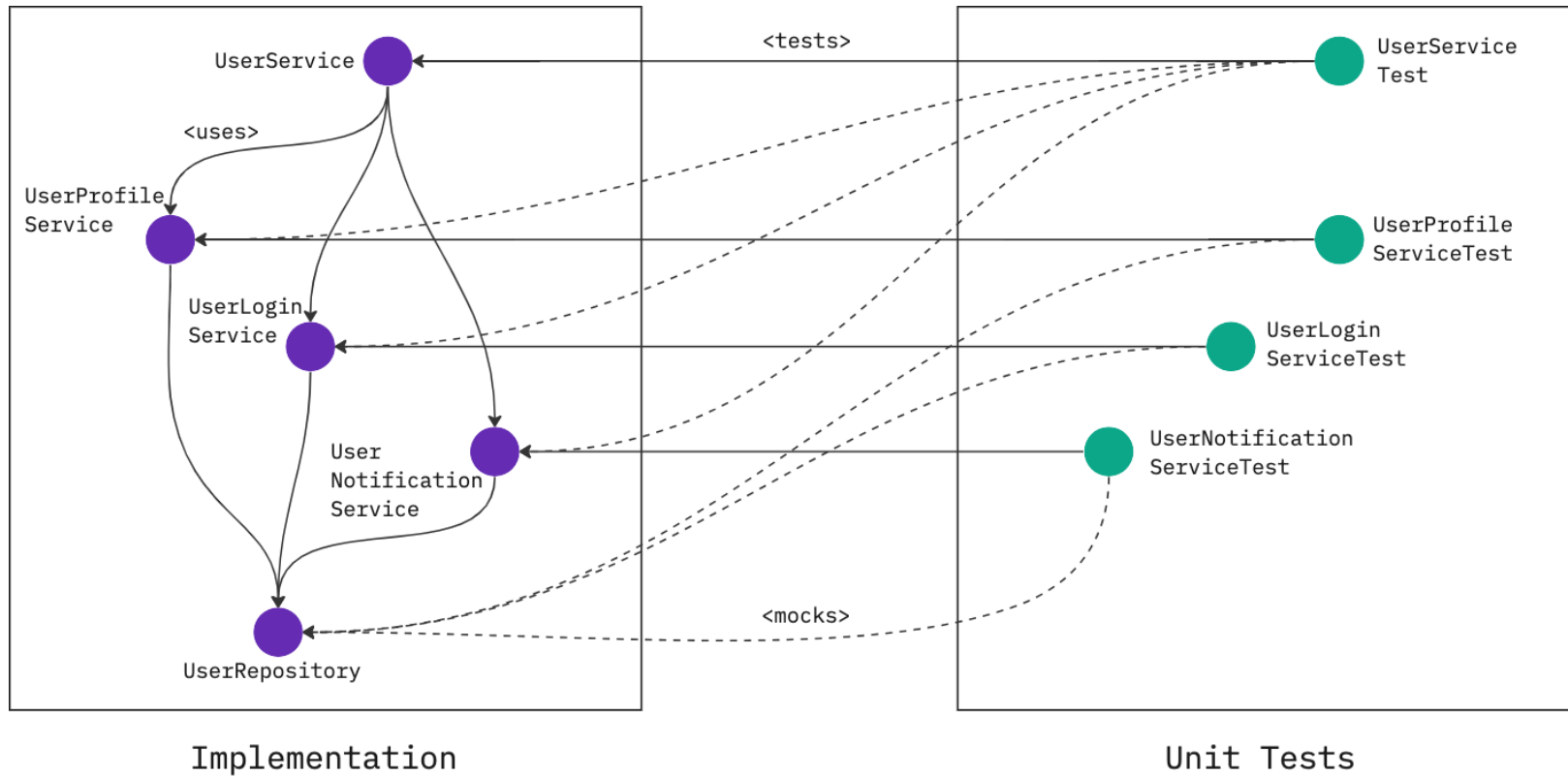
# Fragile Test Problem

## Definition



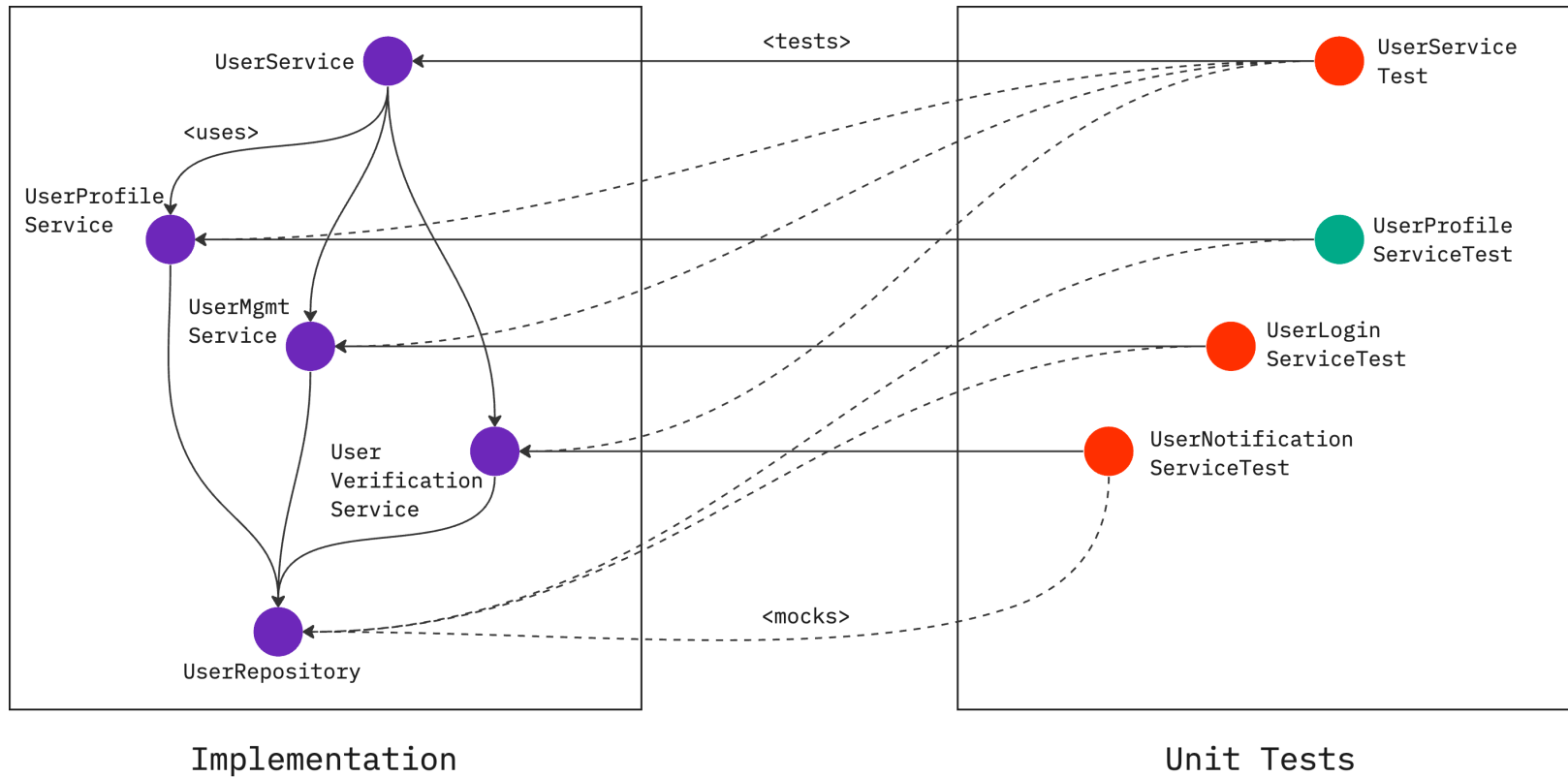
# Fragile Test Problem

## Definition



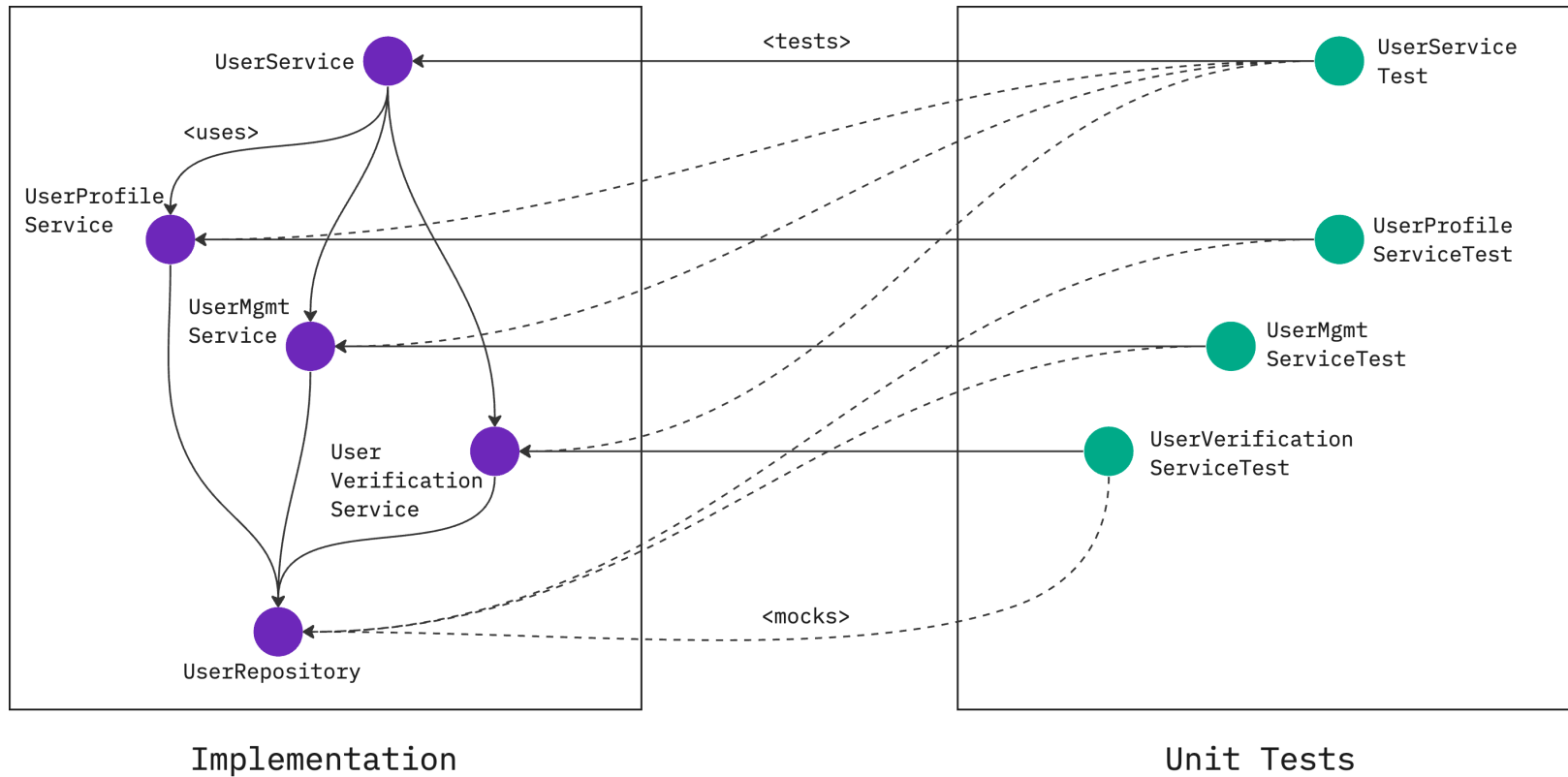
# Fragile Test Problem

## Definition



# Fragile Test Problem

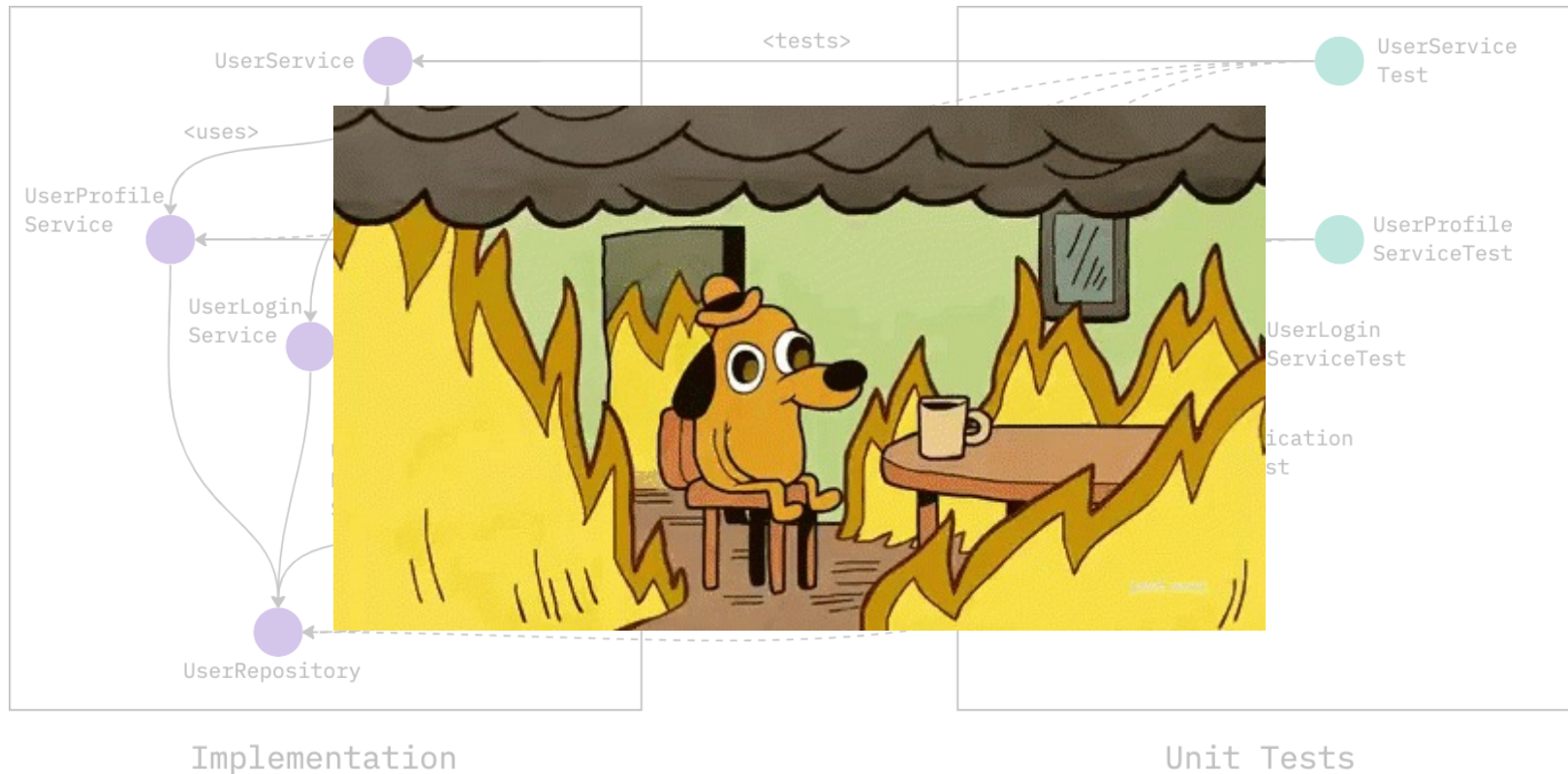
## Definition





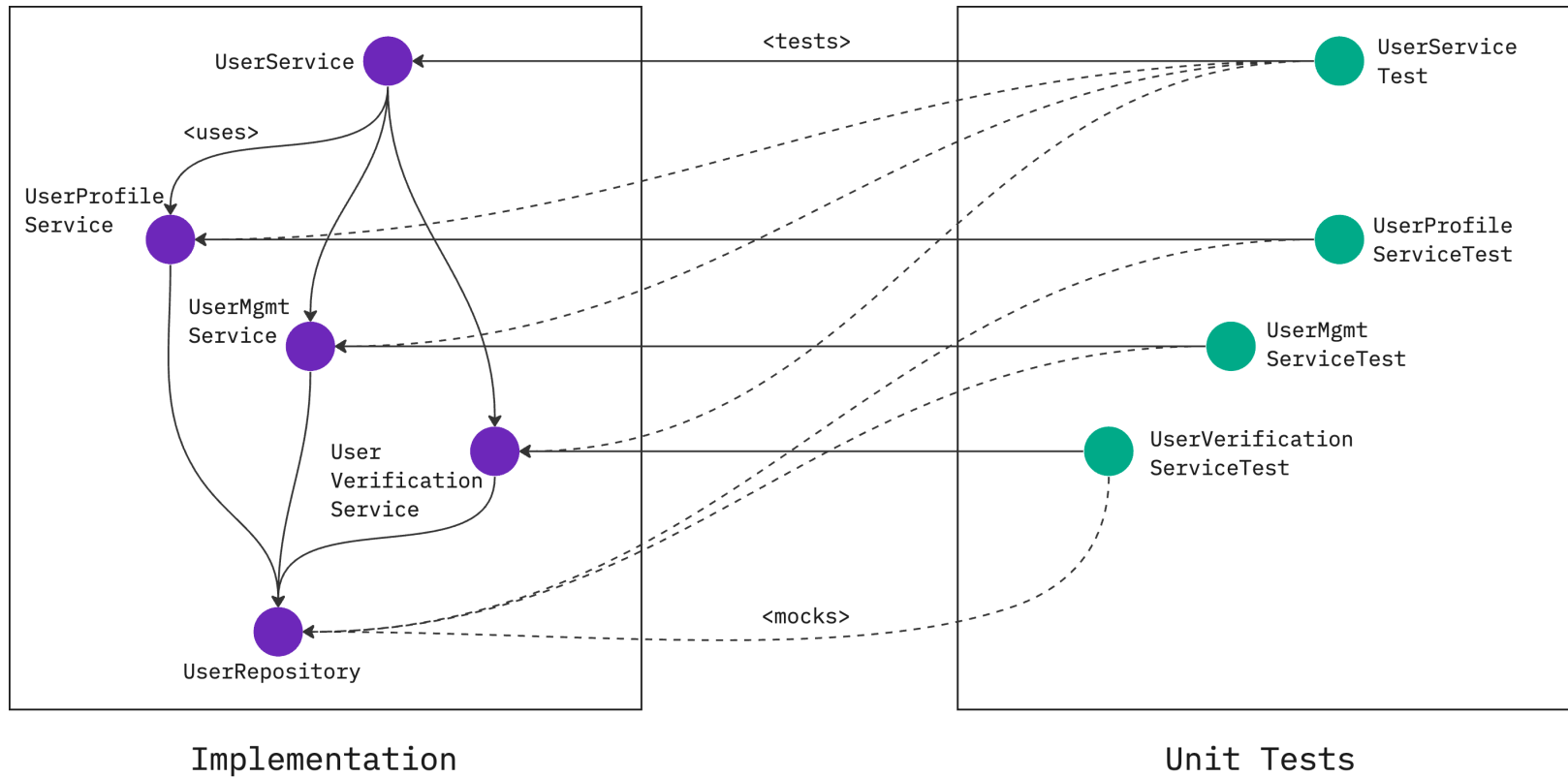
# Fragile Test Problem

This is fine.



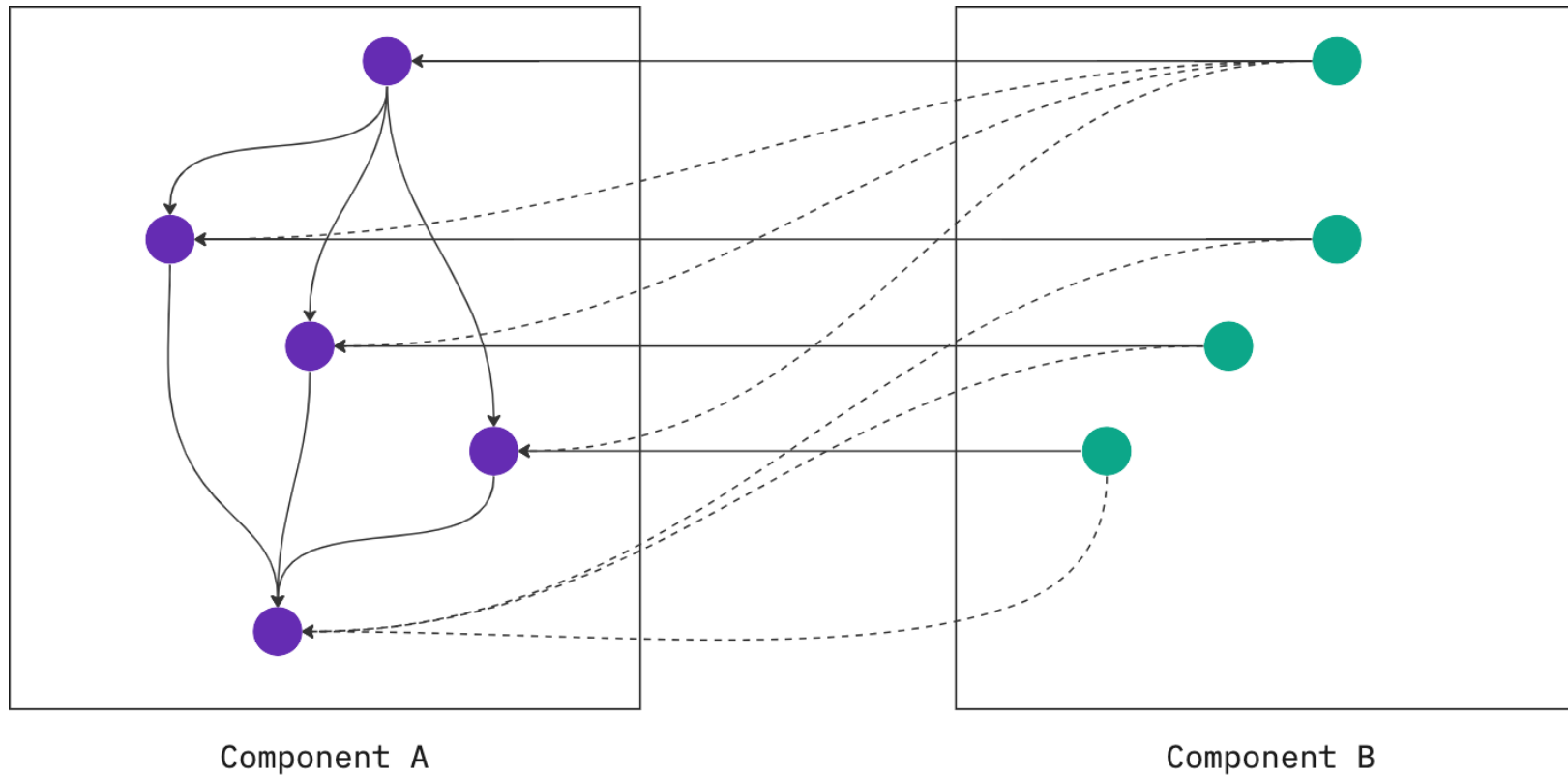
# Fragile Test Problem

This is no refactoring!



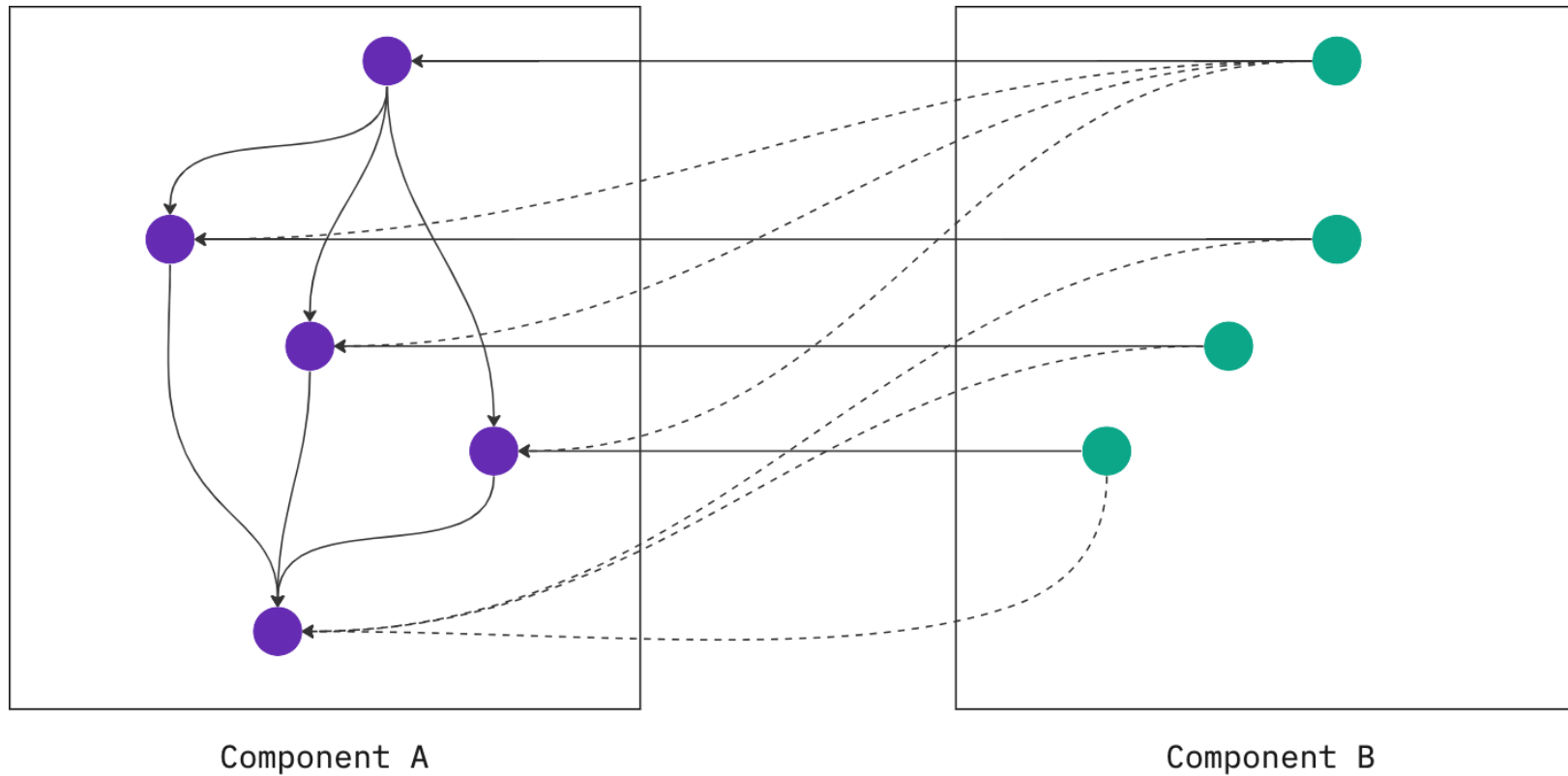
# Fragile Test Problem

## Tight Coupling



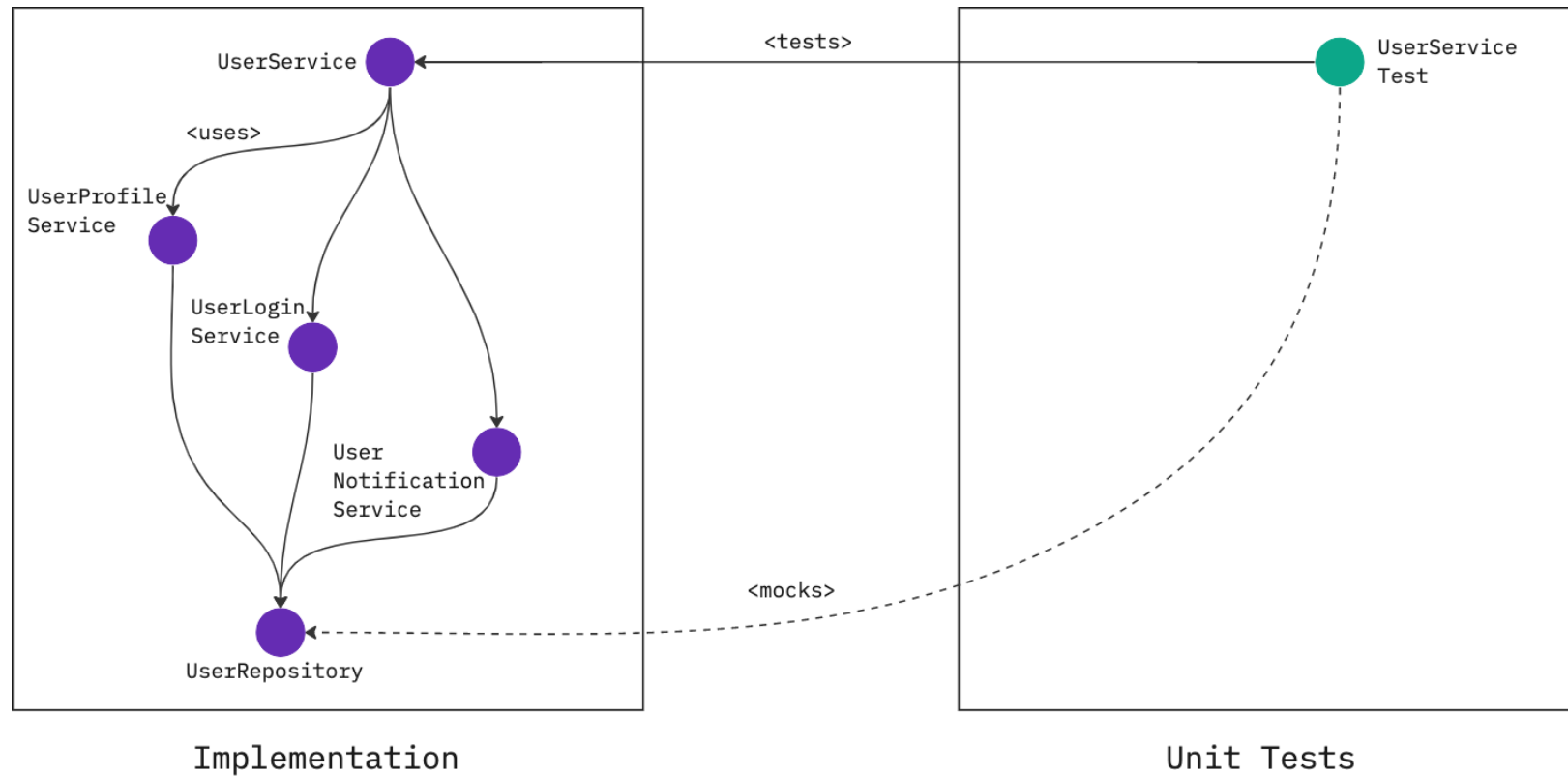
# Fragile Test Problem

Tight Coupling = Bad Design



# Fragile Test Problem

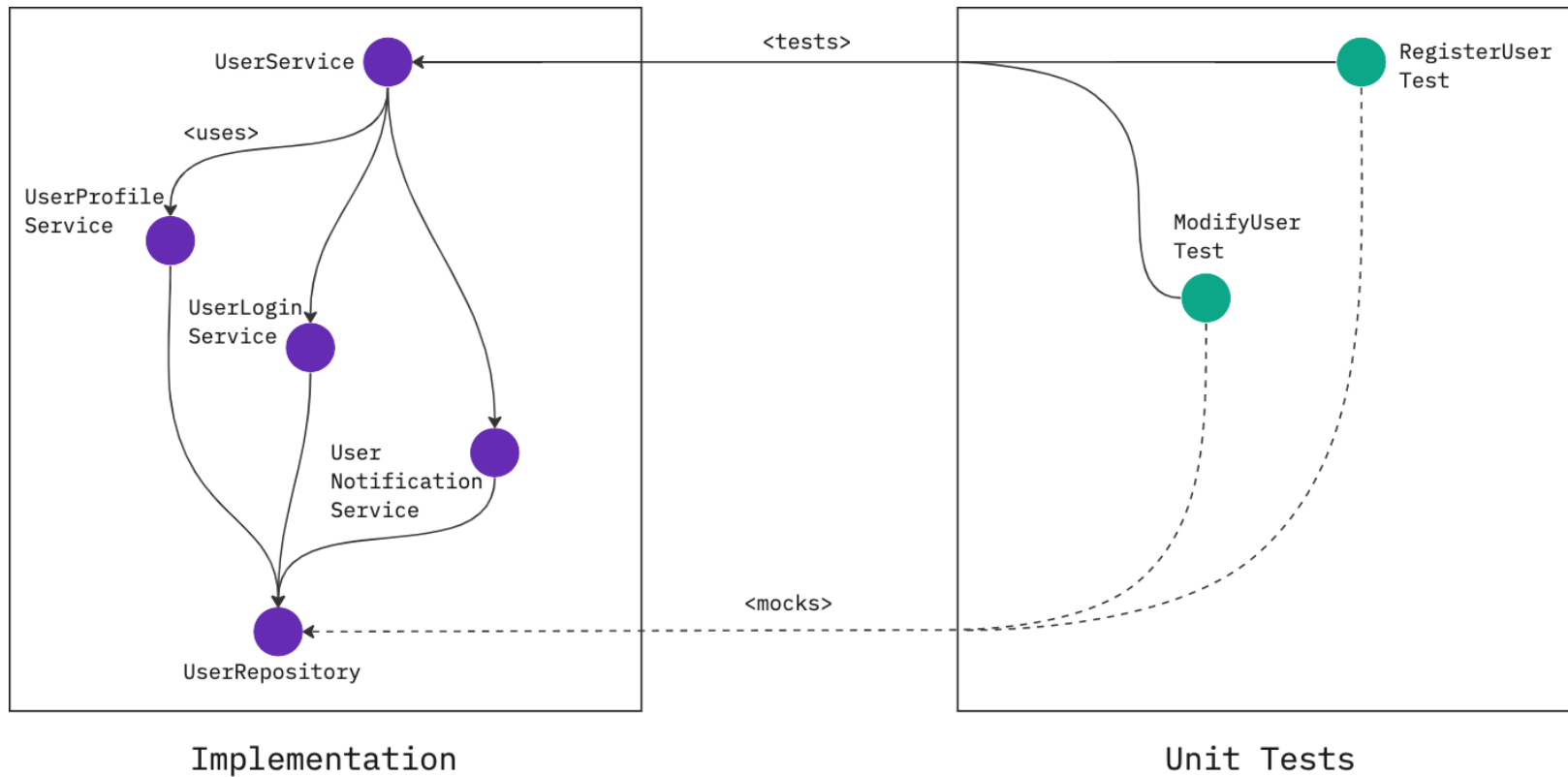
Solution: Loosely coupled tests



<https://blog.cleancoder.com/uncle-bob/2017/10/03/TestContravariance.html>

# Fragile Test Problem

Solution: Loosely coupled tests

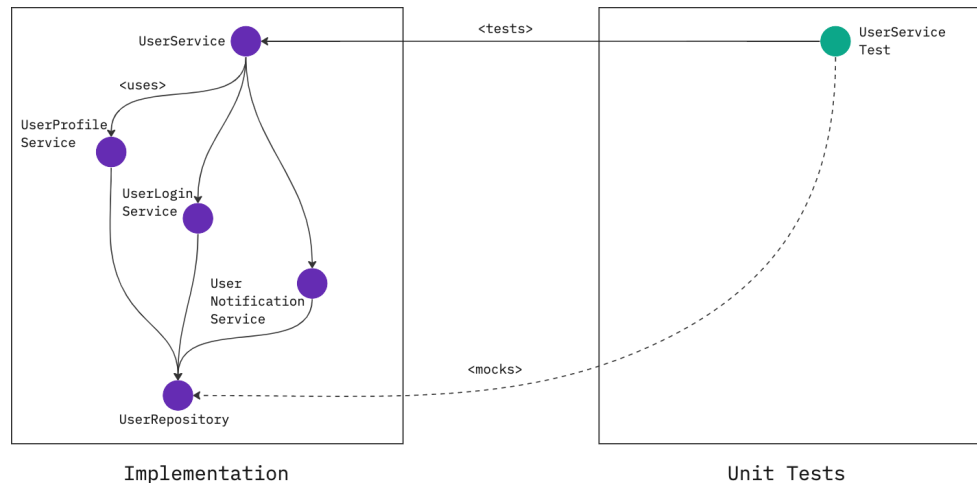


<https://blog.cleancoder.com/uncle-bob/2017/10/03/TestContravariance.html>

# Fragile Test Problem

But wait.

## That breaks the rules!



~~For every class  $X$  there should be a test class  $X$ Test.~~

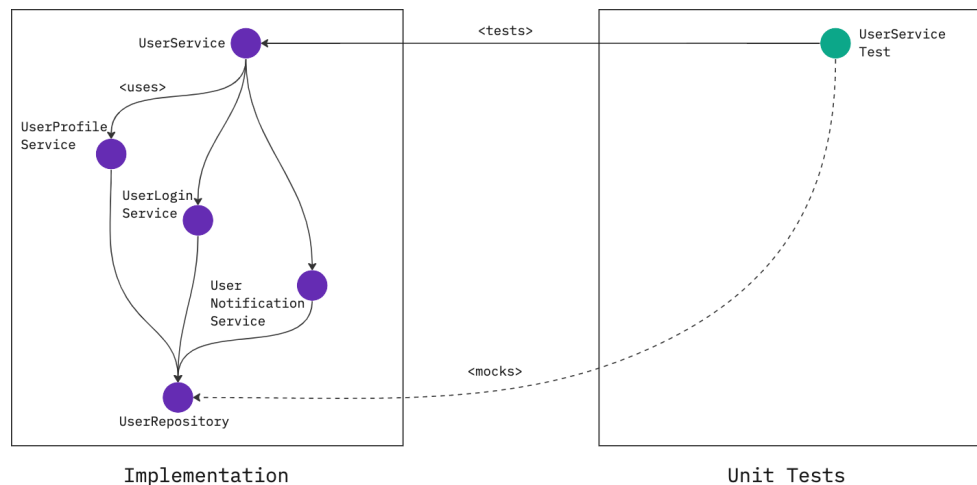
# Fragile Test Problem

But wait.

## Isn't that indirect testing?

Typical example for indirect testing: Testing through the presentation layer

No. Here we test the result of the interaction between the services, not individual services.





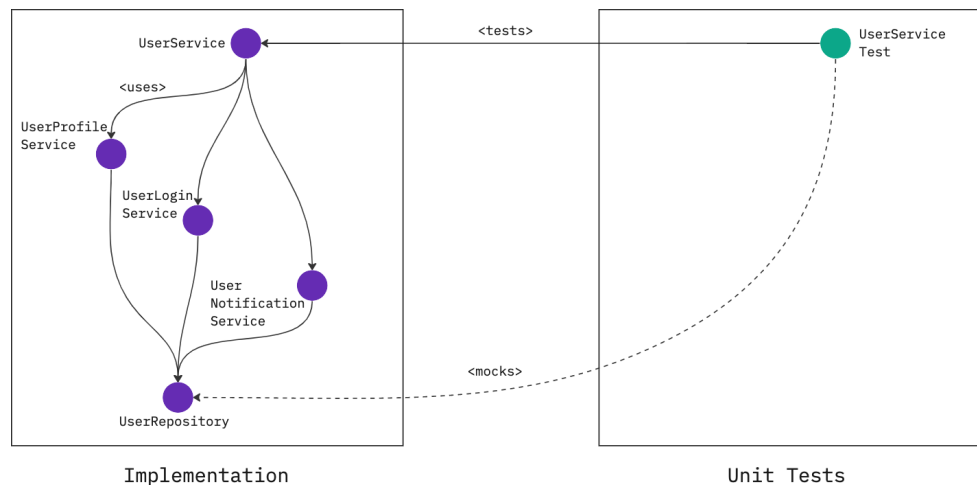
# Fragile Test Problem

But wait.

**It's much harder now to identify the cause of a failing test!**

No. You should work in small increments and run tests after every change.

That makes it easy to identify the cause of a failing test.



Tests should be useful.  
So, design them that way.



# Common Problems with Unit Testing

## Too much boilerplate code

- Tests are hard to understand
- Writing tests is a lot of effort

## The *Fragile Test Problem*

- Adjusting tests after a minor change takes two days
- Tests generate too many false positives

## Tests are running too long

→ **No one likes writing tests**



<https://imgs.xkcd.com/comics/compiling.png>

# Thank you!



Many people contributed to this work:

- Andreas Hager
- Niklas Keller
- Martin Hofmann-Sobik
- And many others...

**Maybe you? Join us!**

<https://about.chrono24.com/jobs/technology/>

**Email:**

[jens.happe@chrono24.com](mailto:jens.happe@chrono24.com)

**X / Twitter:**

@HappeJens

**LinkedIn:**

<https://www.linkedin.com/in/jens-happe-idea-to-impact/>

