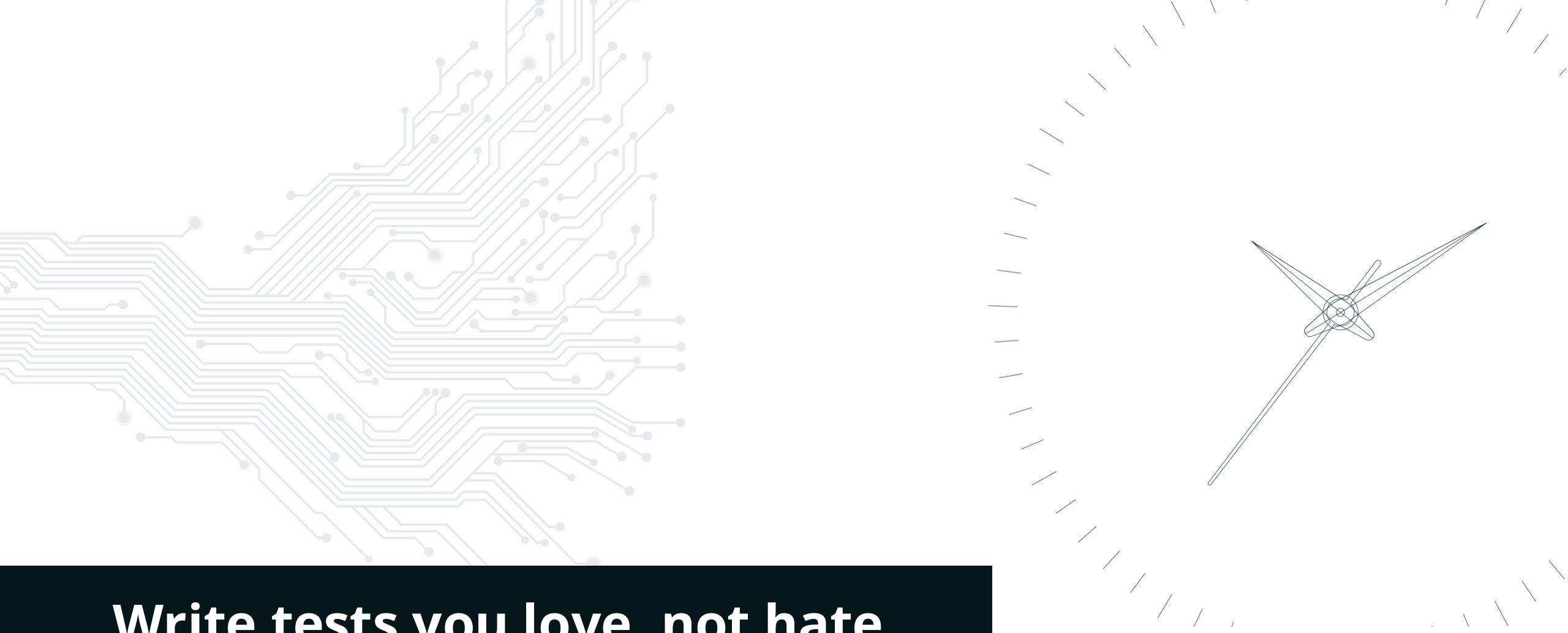


Write tests you love, not hate

Dr.-Ing. Jens Happe
Head of Software Engineering - Chrono24
Co-Founder - Sparkteams

chrono24
The World's Watch Market

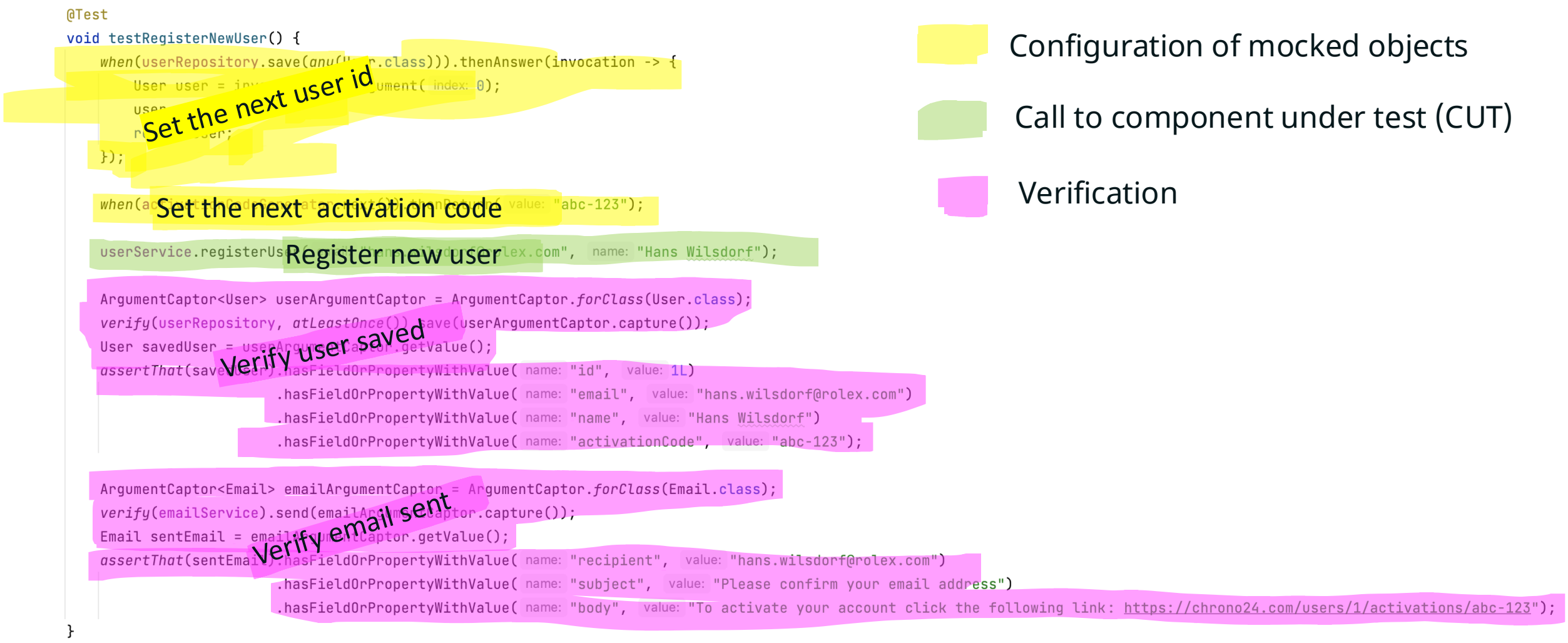
What does the following test check?

_____

What does this test check?

```
@Test
void testRegisterNewUser() {
    when(userRepository.save(any(User.class))).thenAnswer(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?



Configuration of mocked objects

Call to component under test (CUT)

Verification

Common Problems with Unit Testing

chrono24

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

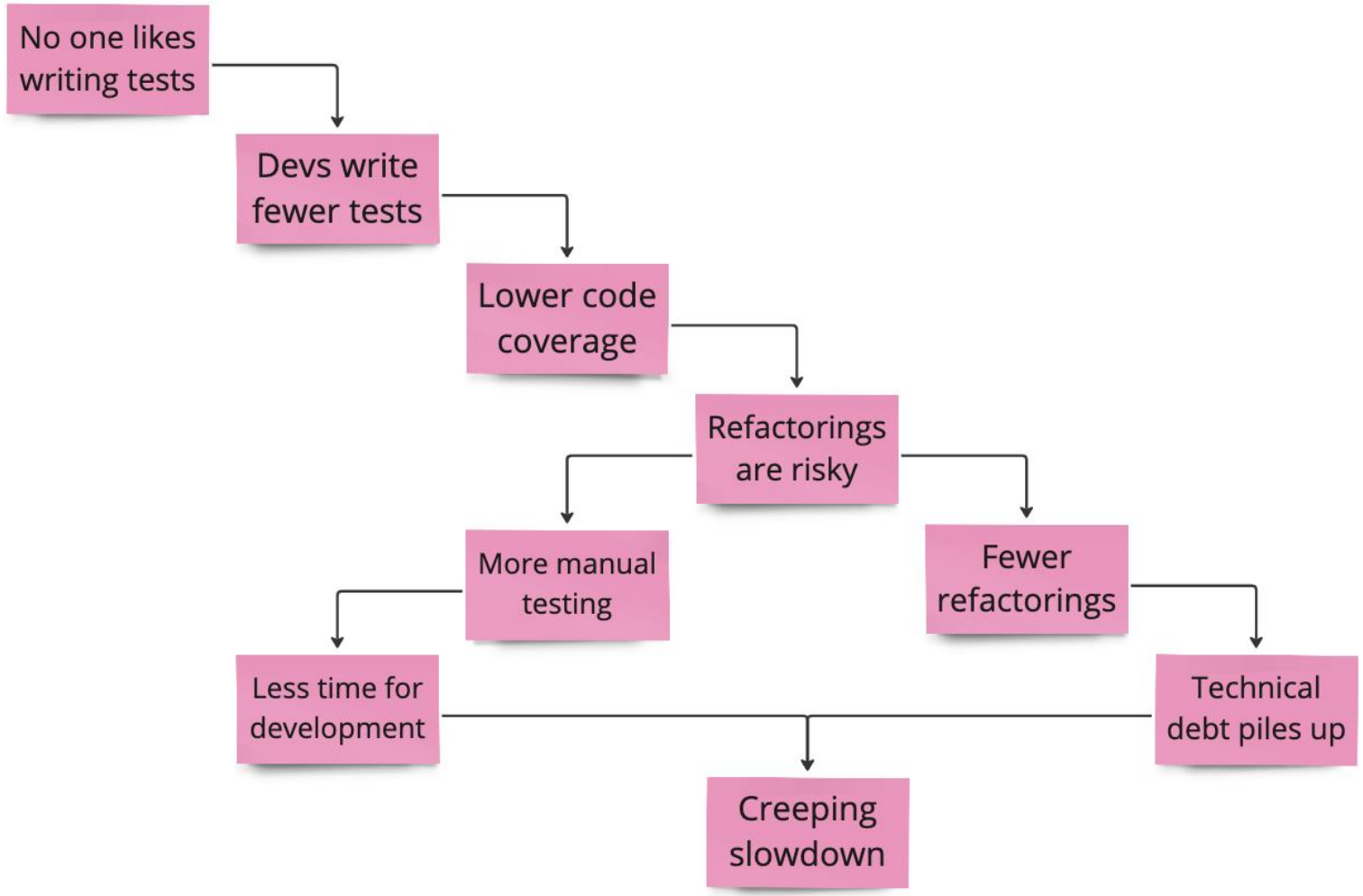
Tests are running too long

→ No one likes writing tests

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

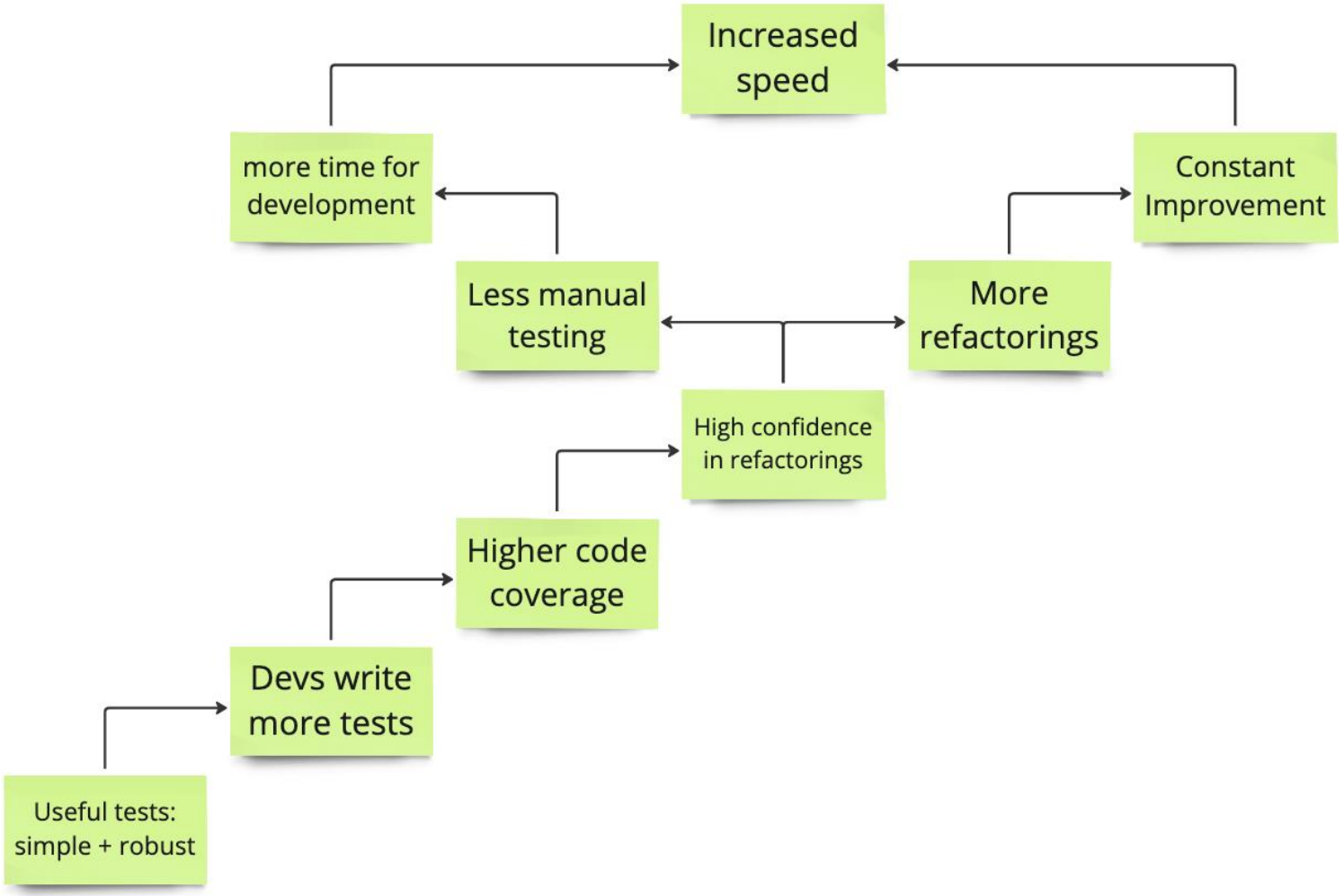
Okay, no one likes writing tests. So what?!

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

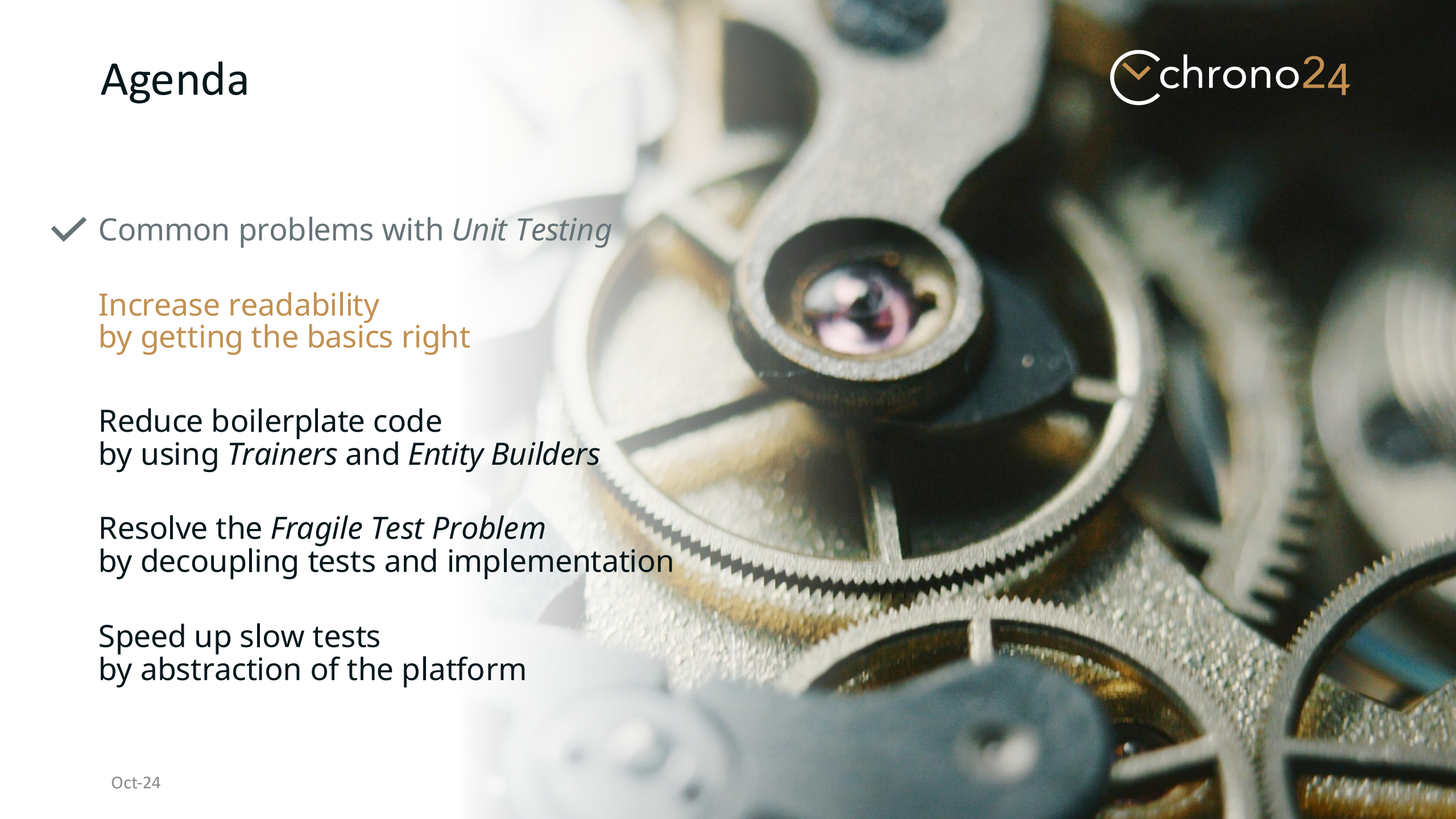


What if...

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



Agenda



- ✓ Common problems with *Unit Testing*

Increase readability
by getting the basics right

Reduce boilerplate code
by using *Trainers* and *Entity Builders*

Resolve the *Fragile Test Problem*
by decoupling tests and implementation

Speed up slow tests
by abstraction of the platform

Increase readability

Extract method with *Given | When | Then*

Configuration of mocked objects

Call to component under test (CUT)

Verification

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

Set the next user id

Set the next activation code

Register new user

Verify user saved

Verify email sent

Increase readability

Extract method with *Given | When | Then*

given

when

then

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

Set the next user id

Set the next activation code

Register new user

Verify user saved

Verify email sent

Increase readability

Extract method with *Given | When | Then*

given

when

then

```
@Test
void testRegisterNewUser_TestUtilityMethod() {
    givenNextUserId(27);
    givenNextActivationCode( value: "123-abc");

    whenRegisterNewUser( email: "louis.brandt@omega.com",  name: "Louis Brandt");

    thenNewUserIsCreated().hasEmail("louis.brandt@omega.com")
                          .hasName("Louis Brandt")
                          .hasActivationCode("123-abc");
    thenEmailIsSent().hasRecipient("louis.brandt@omega.com")
                     .hasSubject("Please confirm your email address")
                     .hasBody("To activate your account click the following link: https://chrono24.com/users/27/activations/123-abc");
}
```

https://martinfowler.com/bliki/GivenWhenThen.html

Increase readability

Explicit relationship between *Given* and *Then*

given

when

then

```
@Test
void testRegisterNewUser_DerivedValue() {
    givenNextUserId(ID);
    givenNextActivationCode(ACTIVATION_CODE);

    whenRegisterNewUser(EMAIL, NAME);

    thenNewUserIsCreated().hasEmail(EMAIL)
                          .hasName(NAME)
                          .hasActivationCode(ACTIVATION_CODE);
    thenEmailIsSent().hasRecipient(EMAIL)
                     .hasSubject(ACTIVATION_SUBJECT)
                     .hasBody(ACTIVATION_MESSAGE + BASE_URL + "/users/" + ID + "/activations/" + ACTIVATION_CODE);
}
```

Getting the basic right
for tests already gets you pretty far.

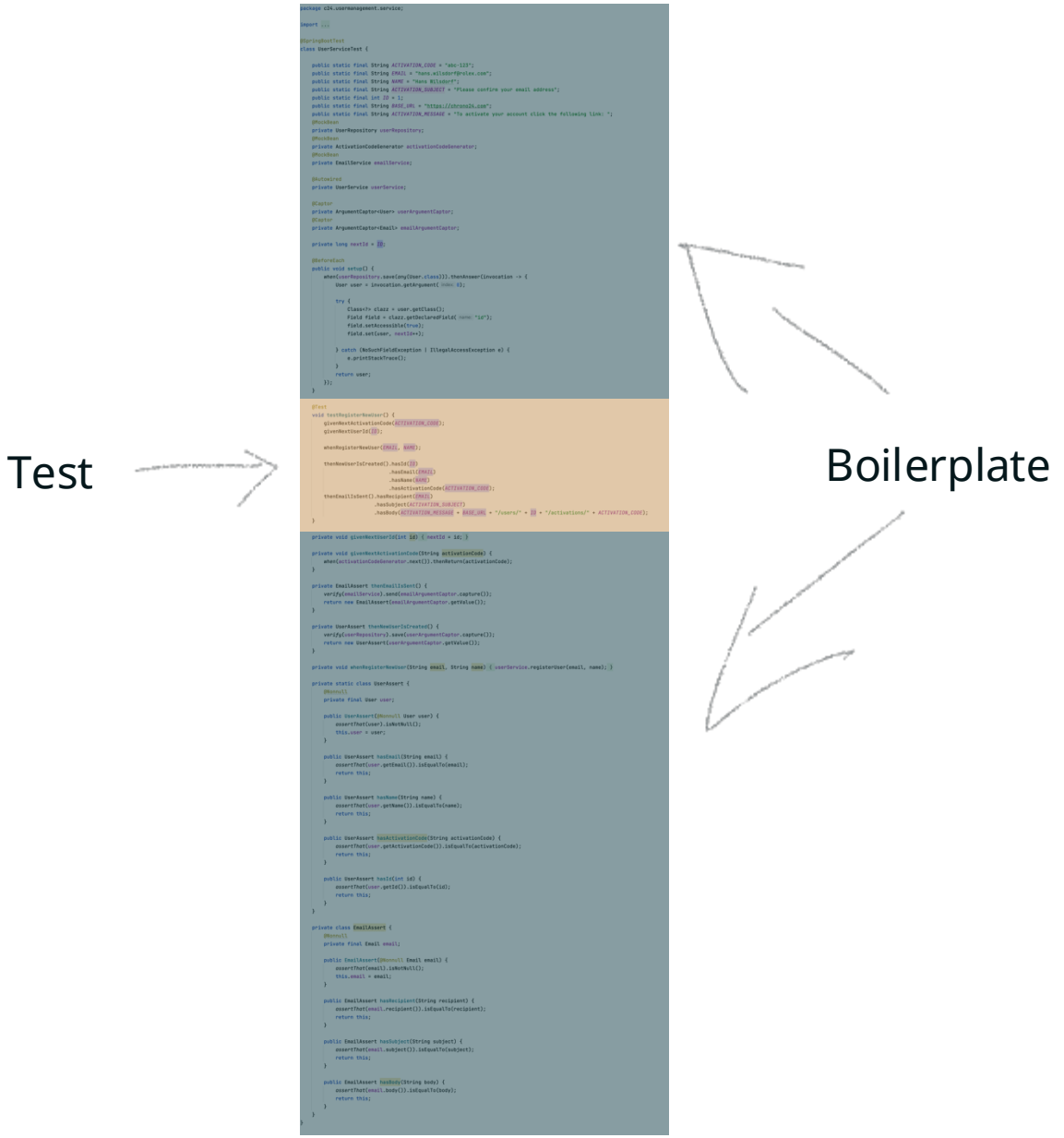
chrono24

So, we are done now, right?

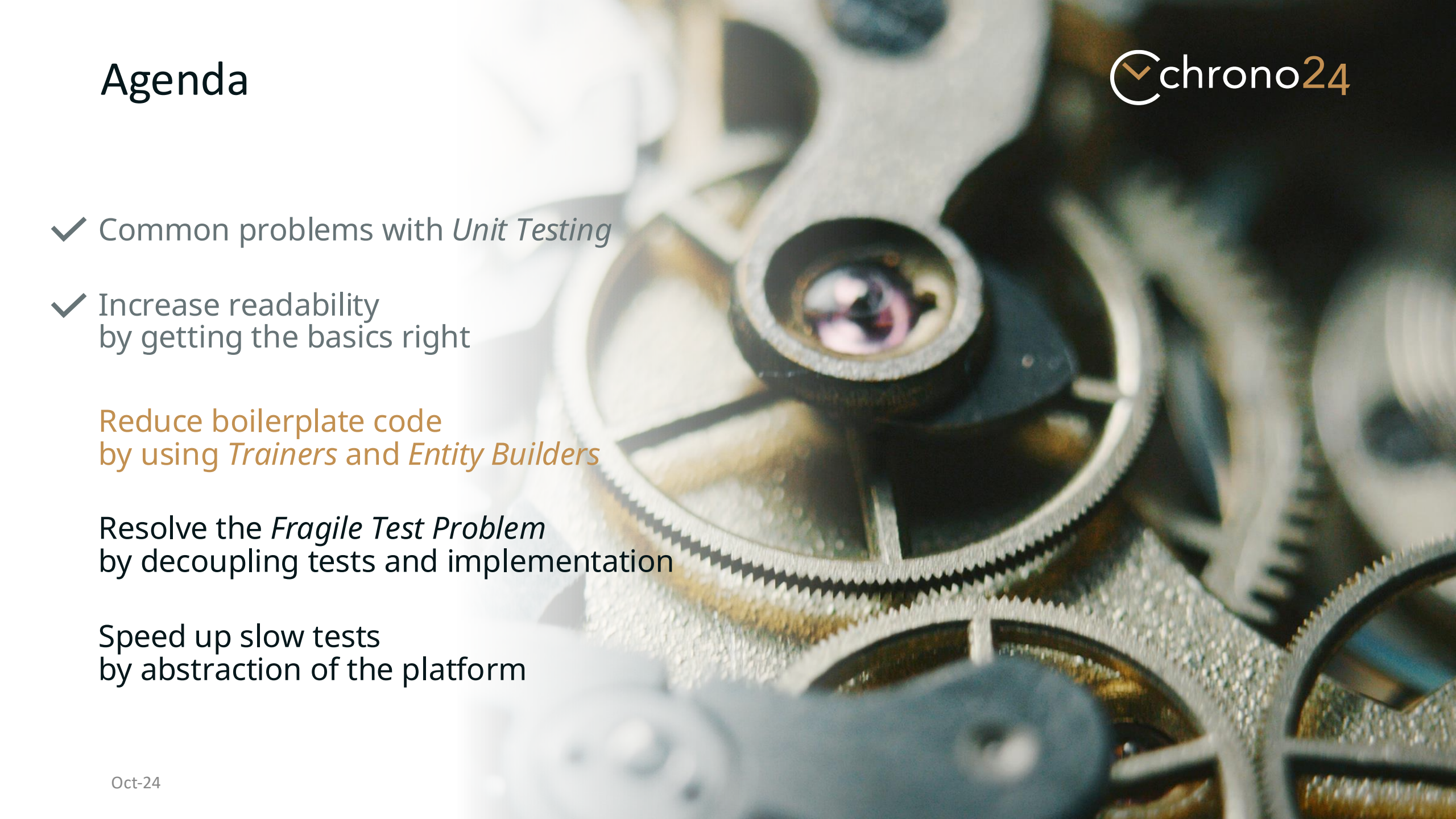
Thank you for your attention!
It was a pleasure ☺

Of course not.

- Our Test Class is still a mess containing almost only boiler plate code.
- We cannot reuse the `given…` and `then…` methods for other test cases.

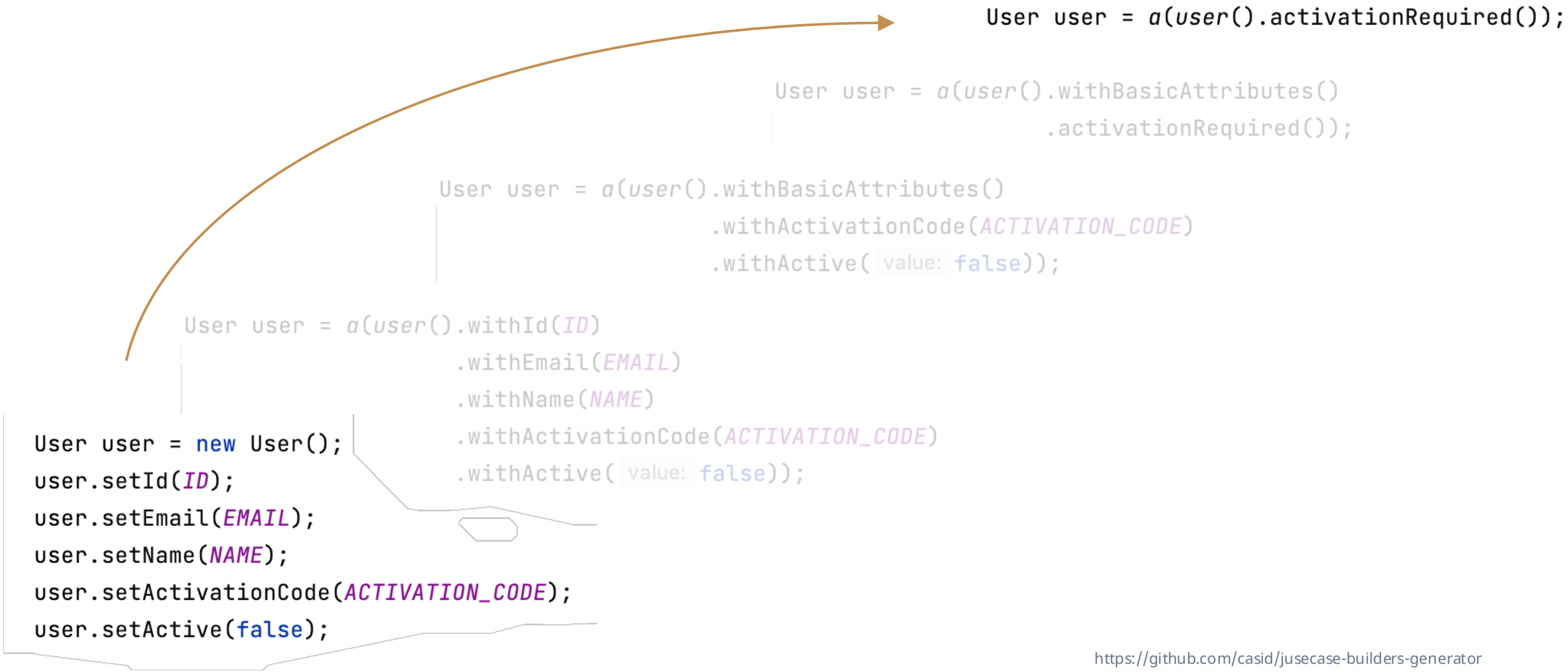


Agenda



- ✓ Common problems with *Unit Testing*
- ✓ Increase readability
  by getting the basics right
- Reduce boilerplate code
  by using *Trainers* and *Entity Builders*
- Resolve the *Fragile Test Problem*
  by decoupling tests and implementation
- Speed up slow tests
  by abstraction of the platform

Entity Builders simplify test setup



https://github.com/casid/jusecase-builders-generator

Trainers

Problem: Configuring mocked objects is very repetitive and verbose

```
@Test
void testRegisterNewUser() {
    when(userRepository.save(any(User.class))).thenAnswer(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");
}
```

Trainers

chrono24

Encapsulate the configuration in separate classes called *Trainers*

```
public class UserRepositoryTrainer {

    public UserRepositoryTrainer() {...}

    public UserRepository getMock() {...}

    public void givenNextId(int id) {...}

    public void givenEntityExists(User user) {...}

    public UserAssert thenNewUserIsCreated() {...}

    public UserAssert thenUserIsUpdated() {...}

    public void thenNoUserIsUpdated() {...}
}
```

```
public abstract class CrudRepositoryTrainer<...> {

    public abstract REPOSITORY getMock();

    public void givenNextId(ID id) {...}

    public void givenEntityExists(TYPE entity) {...}

    public ENTITY_ASSERT thenNewEntityIsCreated() {...}

    public ENTITY_ASSERT thenEntityIsUpdated() {...}

    public void thenNoEntityIsUpdated() {}

}
```

Trainers

chrono24

Encapsulate the configuration in separate classes called *Trainers*

```
public class UserRepositoryTrainer extends CrudRepositoryTrainer<UserRepository, UserAssert, User, Long> {

    private final UserRepository userRepository = mock(UserRepository.class);

    @Override
    public UserRepository getMock() {
        return userRepository;
    }
```

```
public abstract class CrudRepositoryTrainer<...> {

    public abstract REPOSITORY getMock();

    public void givenNextId(ID id) {...}

    public void givenEntityExists(TYPE entity) {...}

    public ENTITY_ASSERT thenNewEntityIsCreated() {...}

    public ENTITY_ASSERT thenEntityIsUpdated() {...}

    public void thenNoEntityIsUpdated() {}

}
```

Trainers

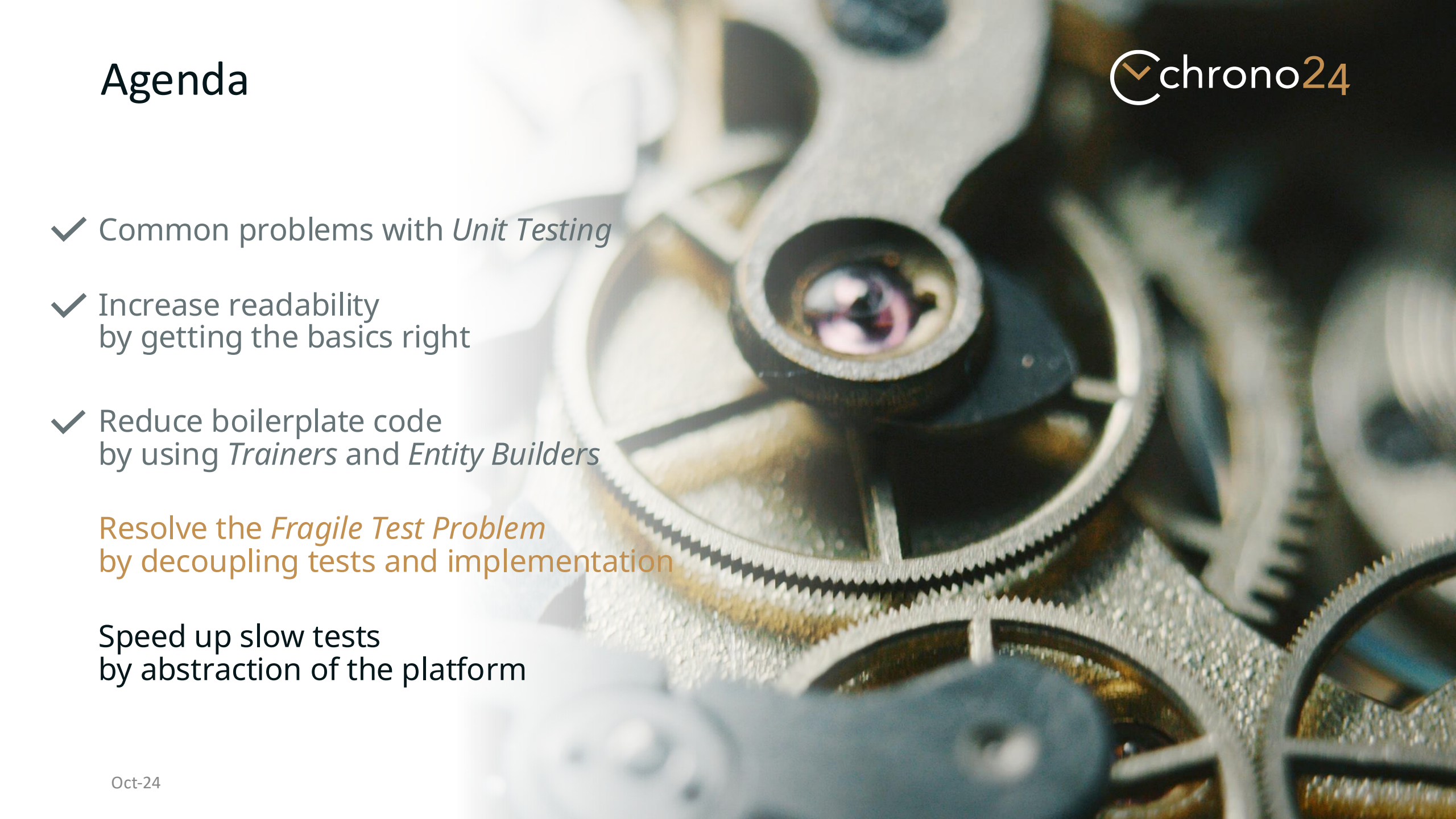
Benefits

- Configuration of mocked objects outside of the test classes
  -> increased readability
  -> increased reusability

- Generic Trainers can be developed, that further simplify the test setup

Entity Builders and *Trainers*
simplify the setup of the test fixture.

Agenda



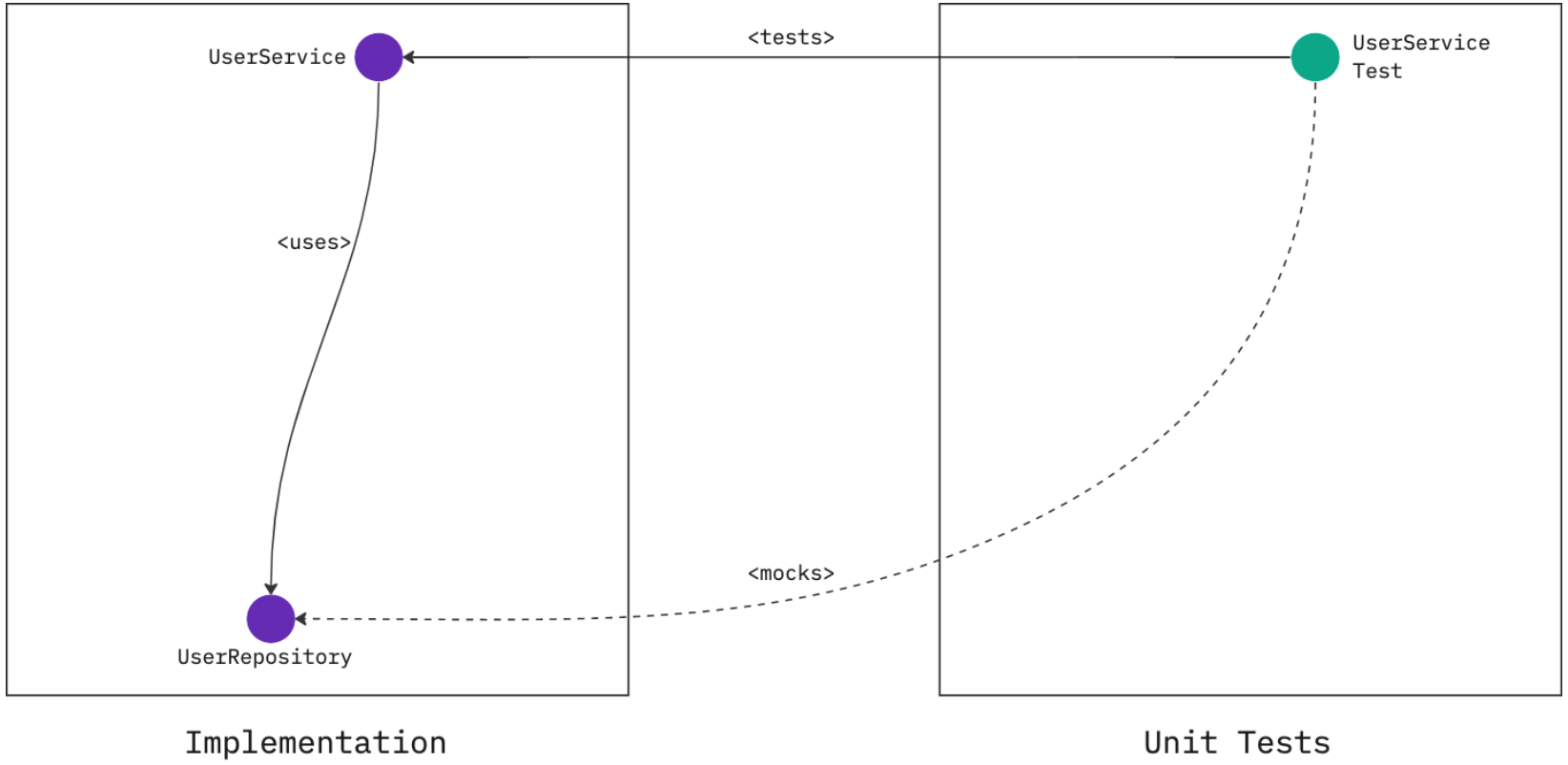
- ✓ Common problems with *Unit Testing*
- ✓ Increase readability
  by getting the basics right
- ✓ Reduce boilerplate code
  by using *Trainers* and *Entity Builders*

Resolve the *Fragile Test Problem*
by decoupling tests and implementation

Speed up slow tests
by abstraction of the platform

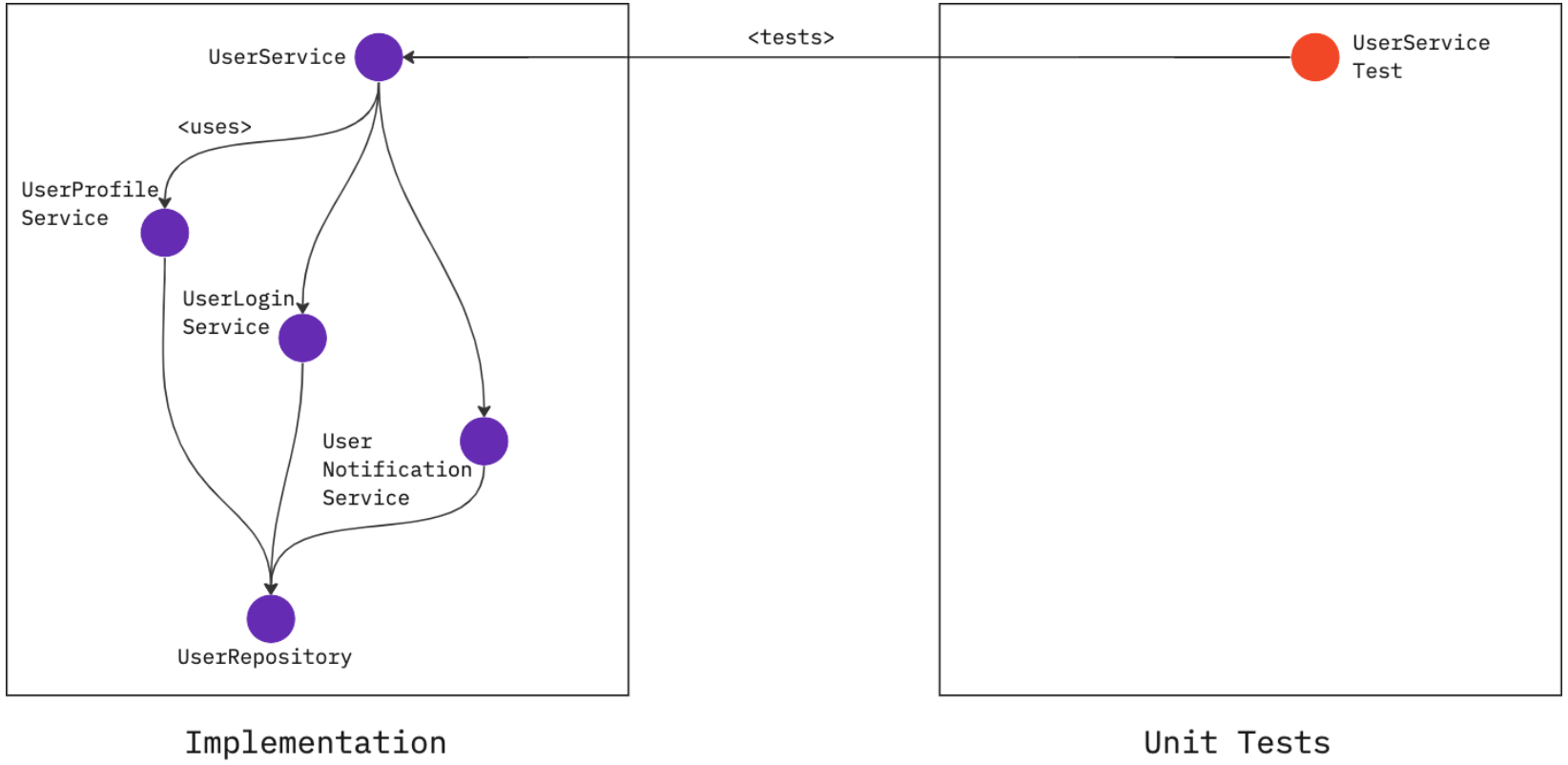
Fragile Test Problem

Definition



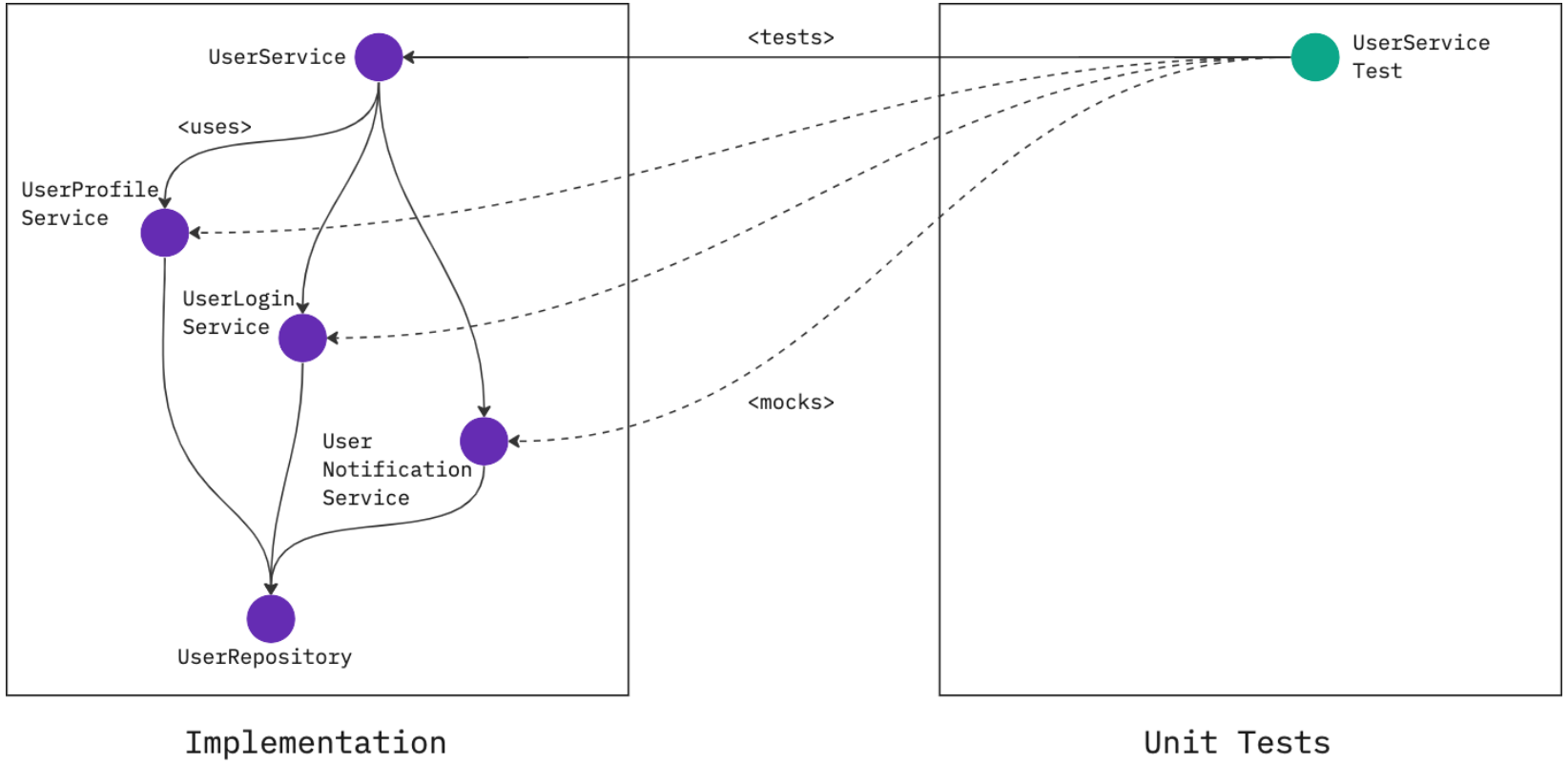
Fragile Test Problem

Definition



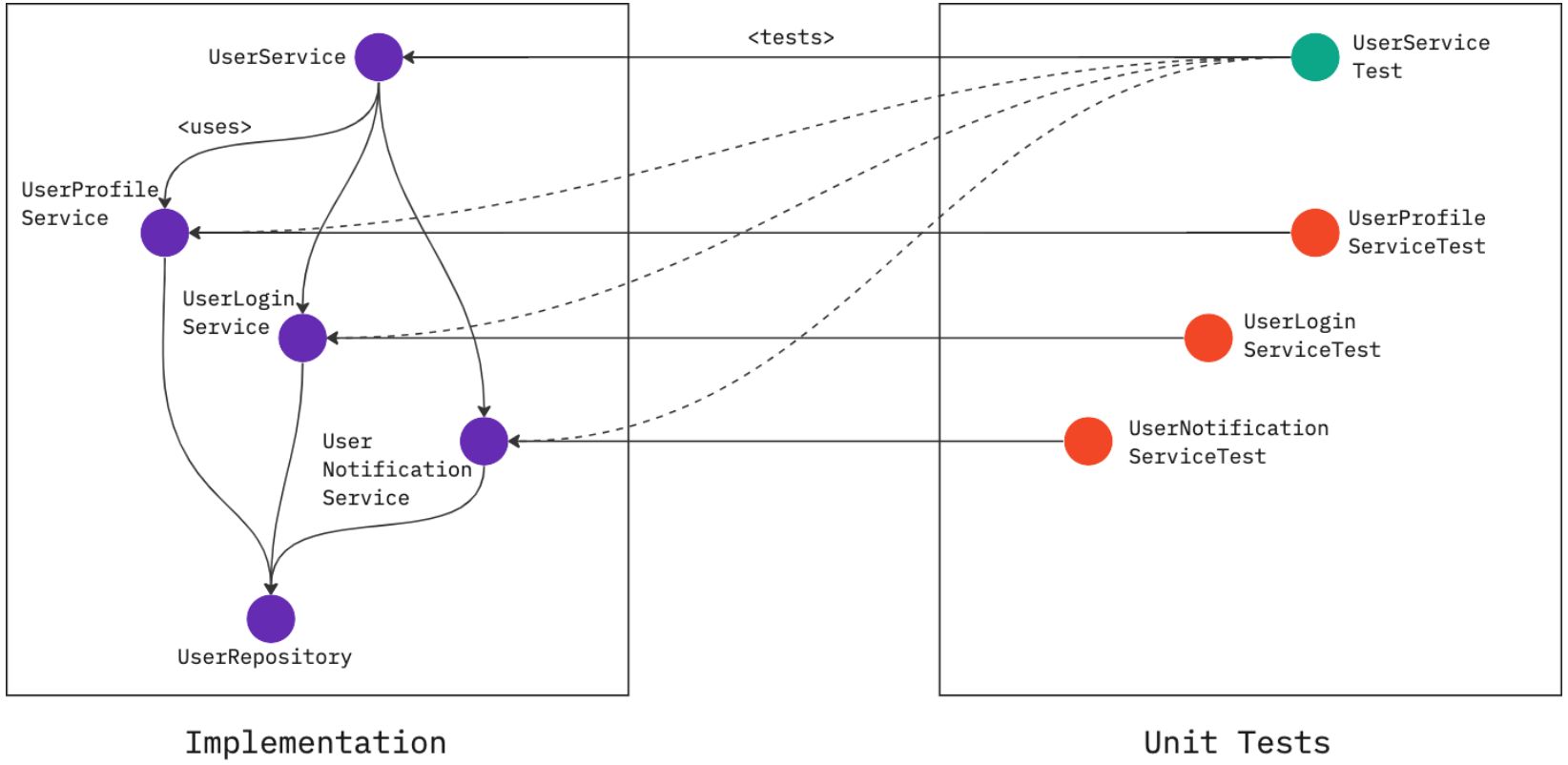
Fragile Test Problem

Definition



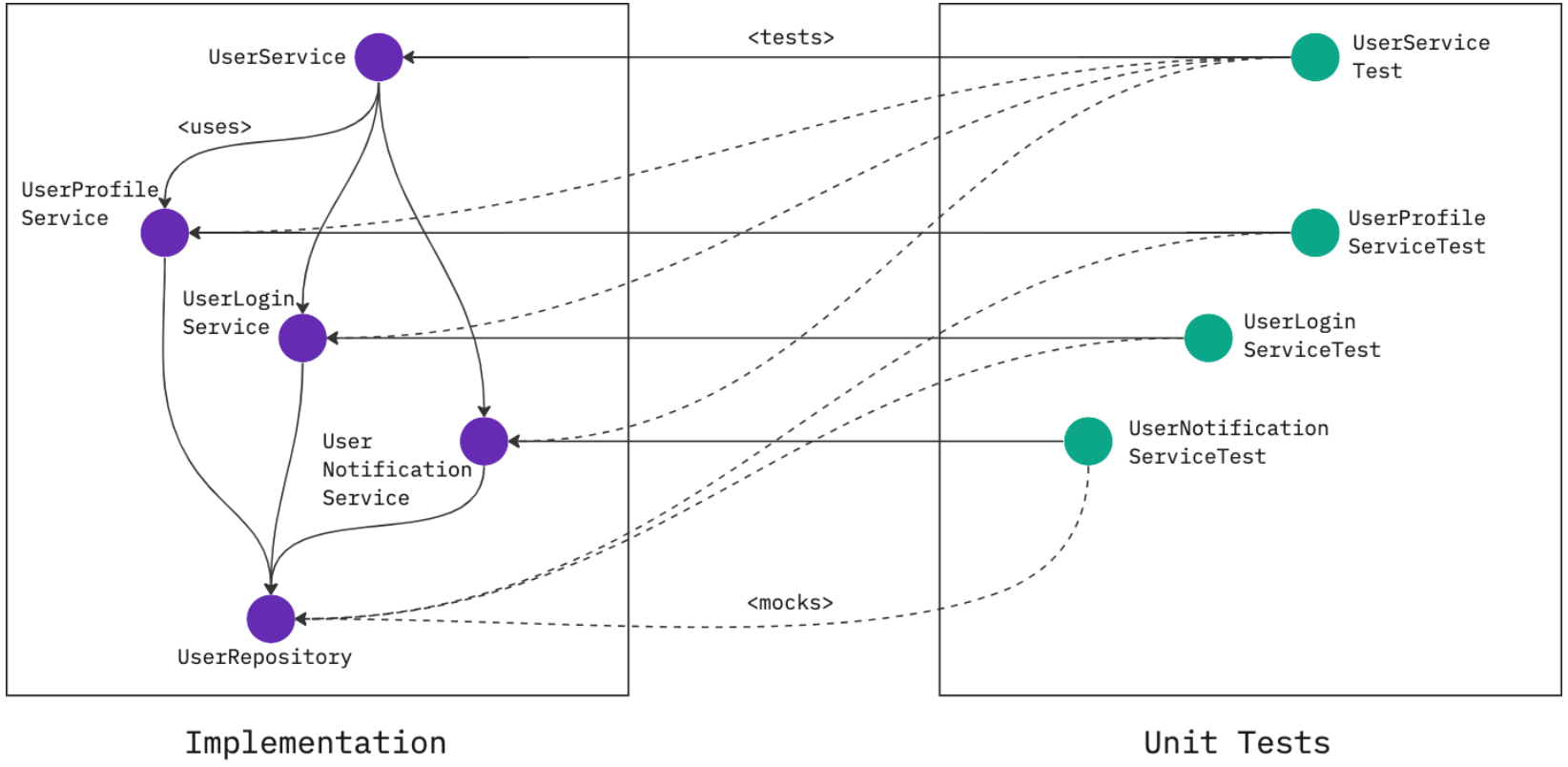
Fragile Test Problem

Definition



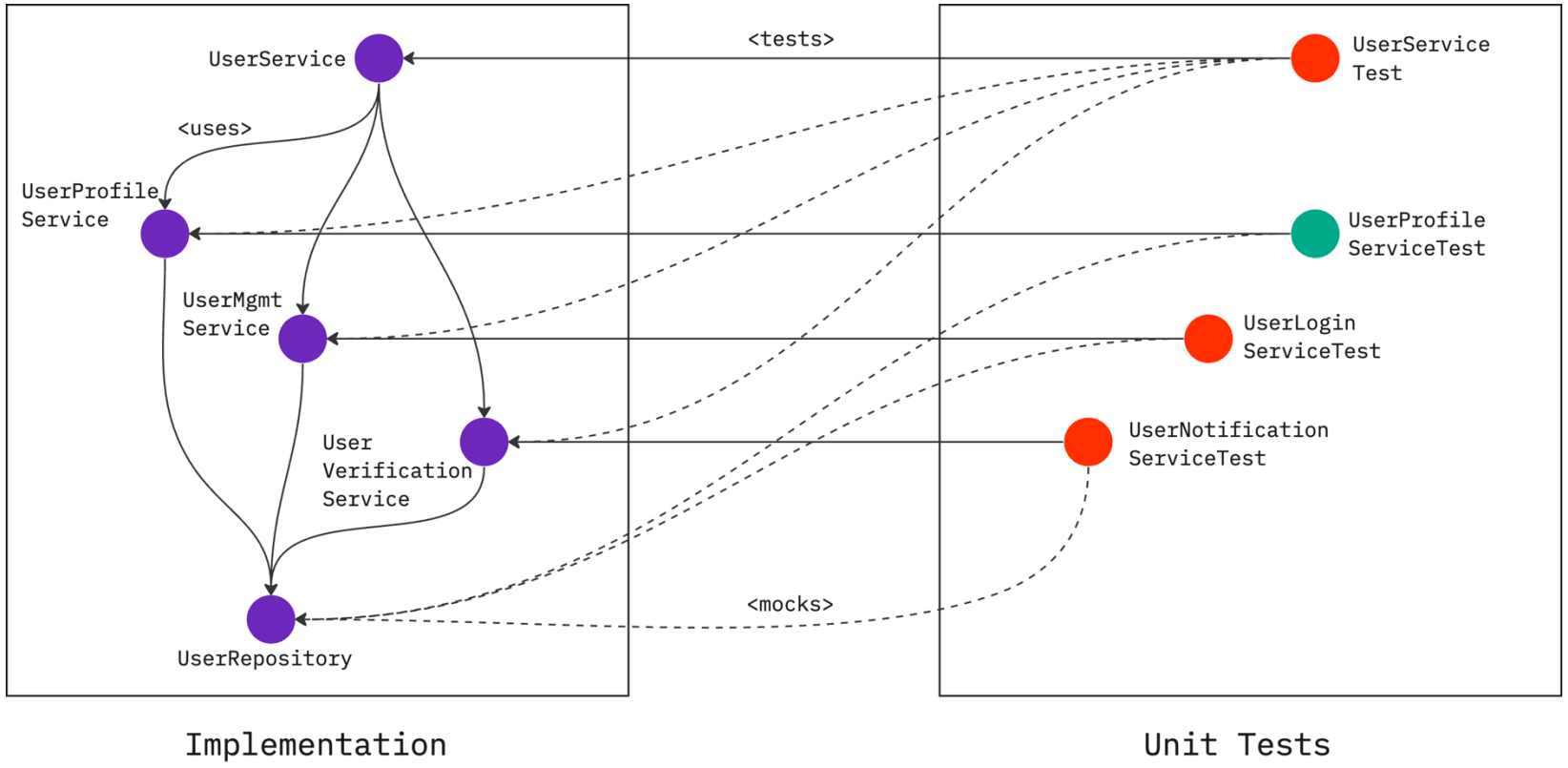
Fragile Test Problem

Definition



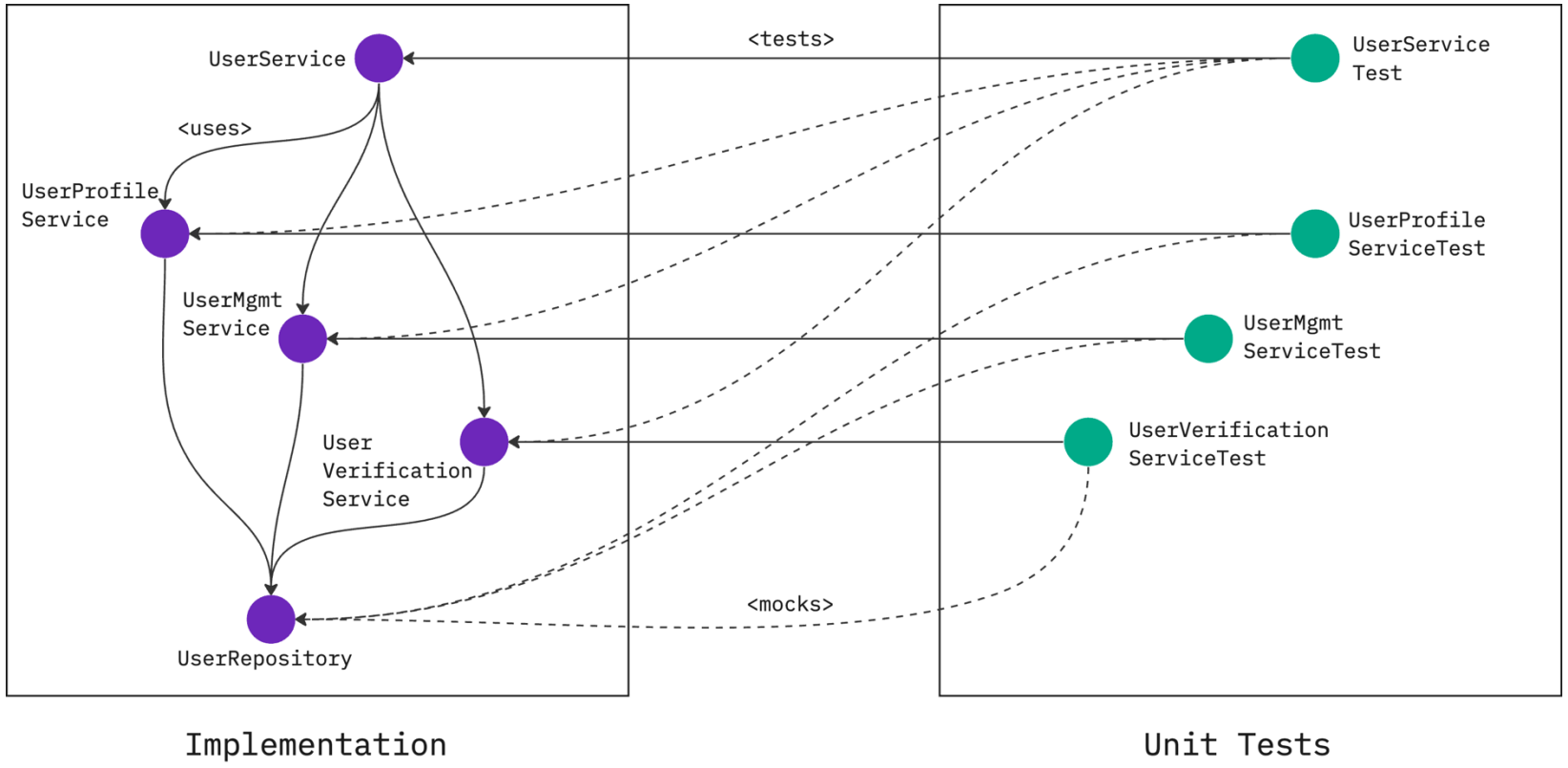
Fragile Test Problem

Definition



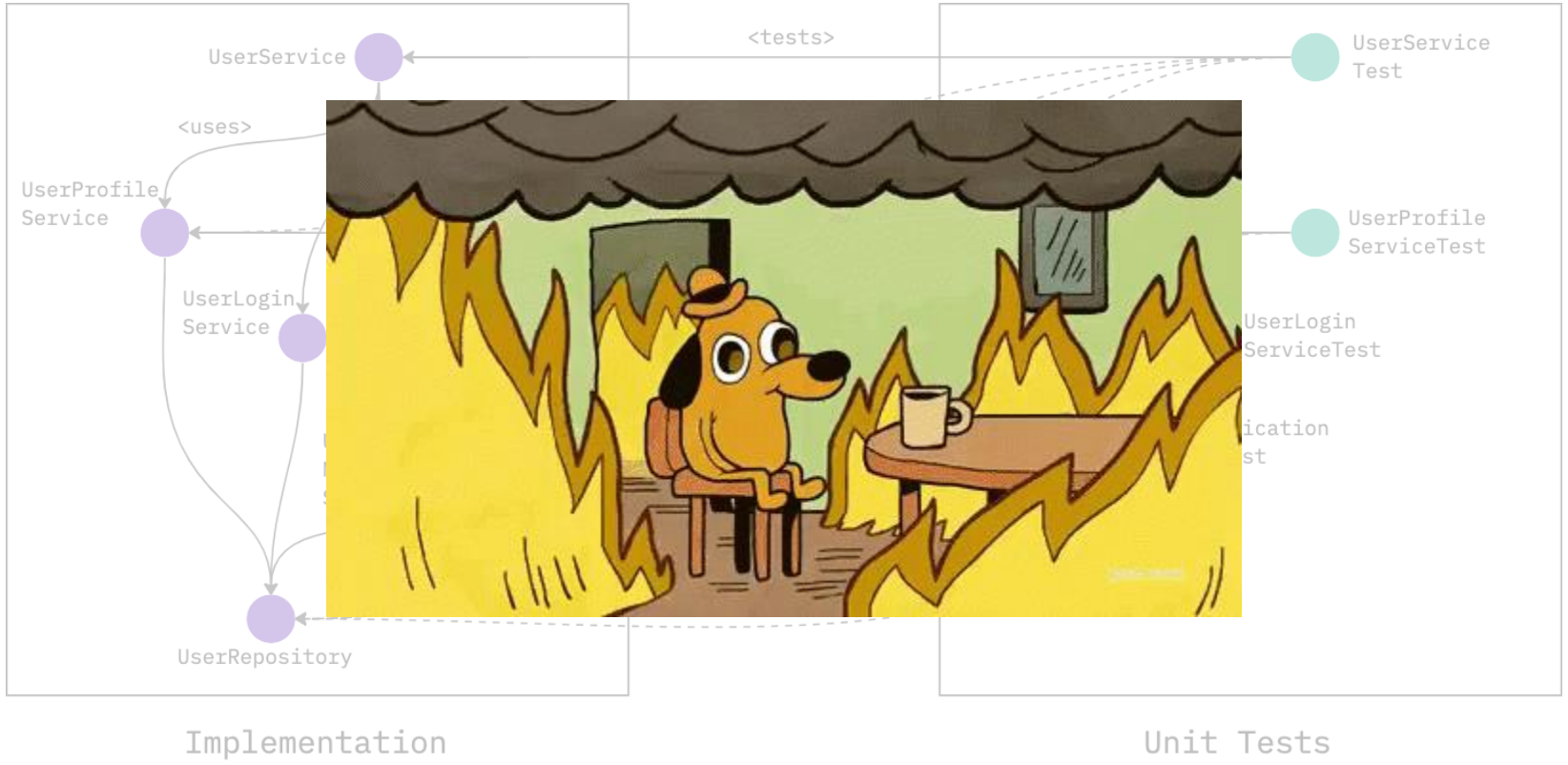
Fragile Test Problem

Definition



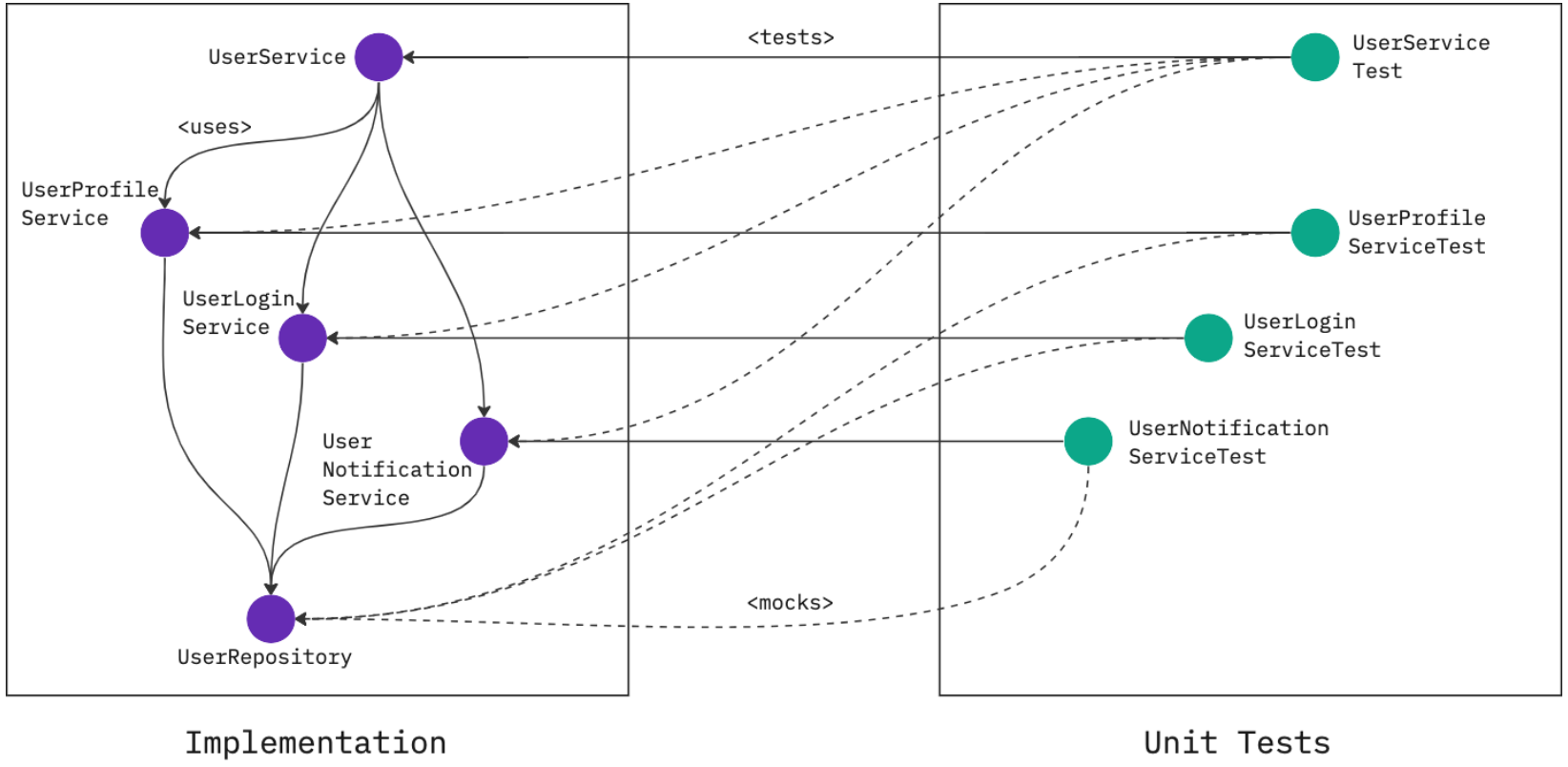
Fragile Test Problem

This is fine.



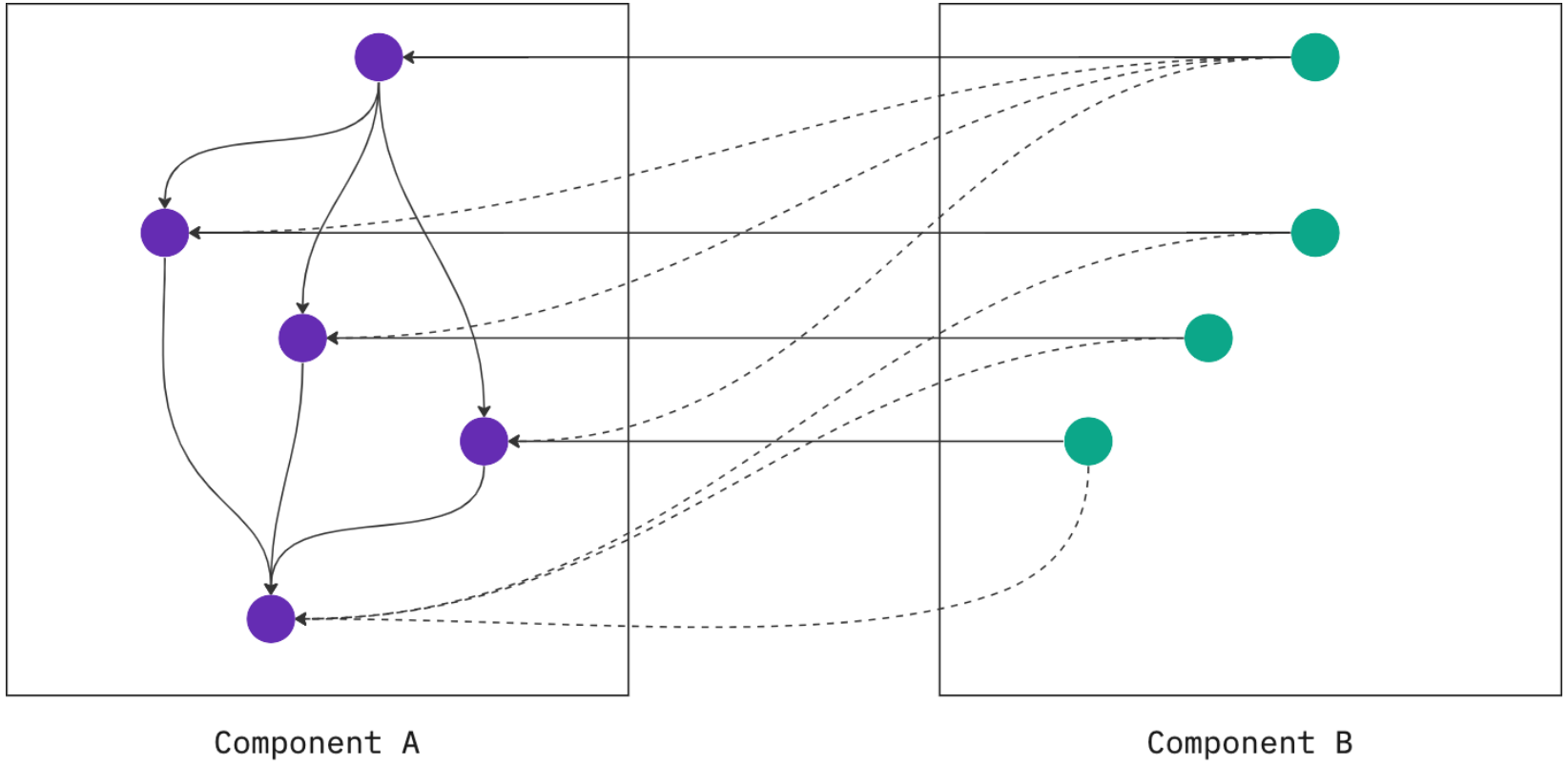
Fragile Test Problem

This is no refactoring!



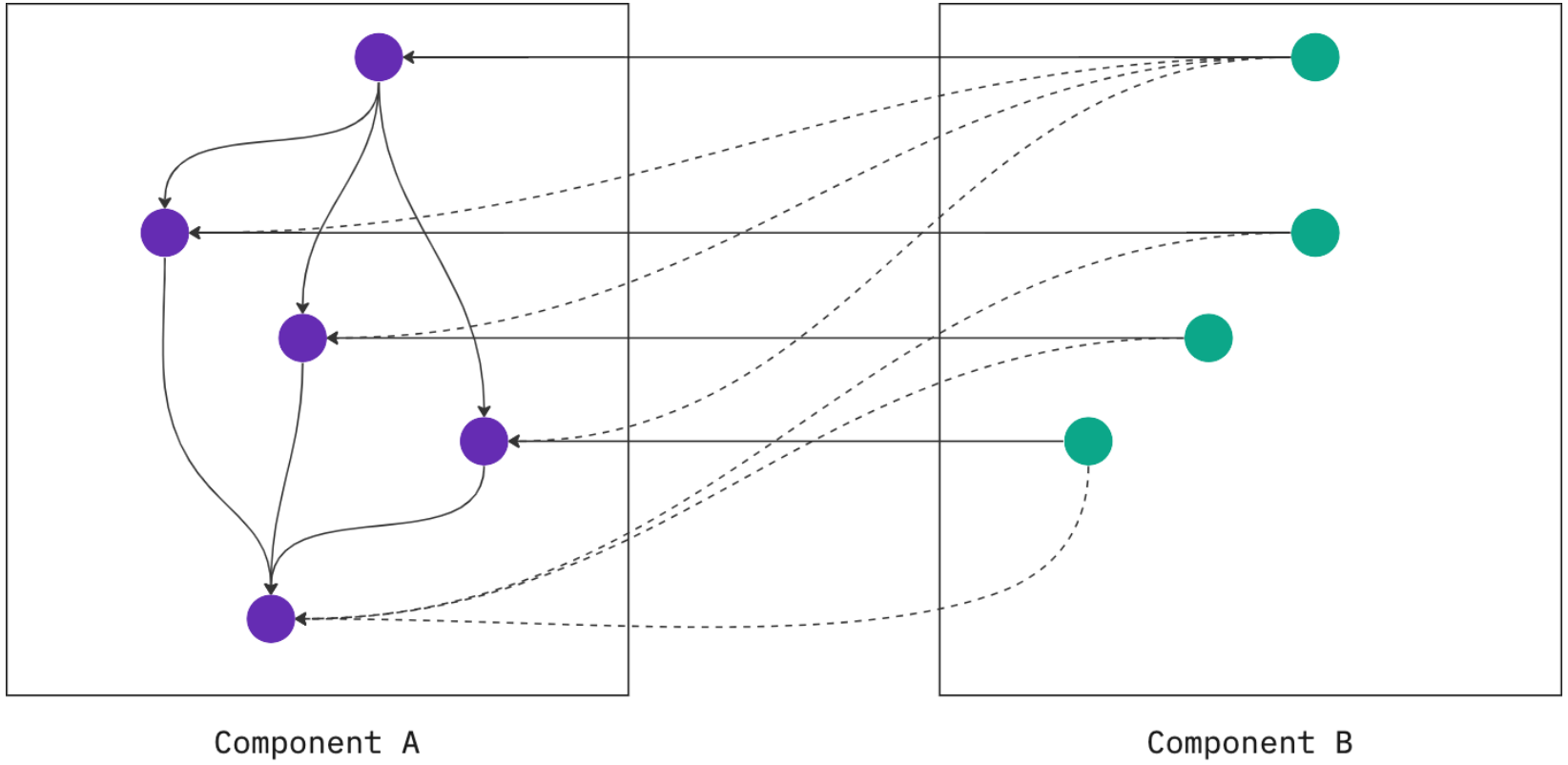
Fragile Test Problem

Tight Coupling



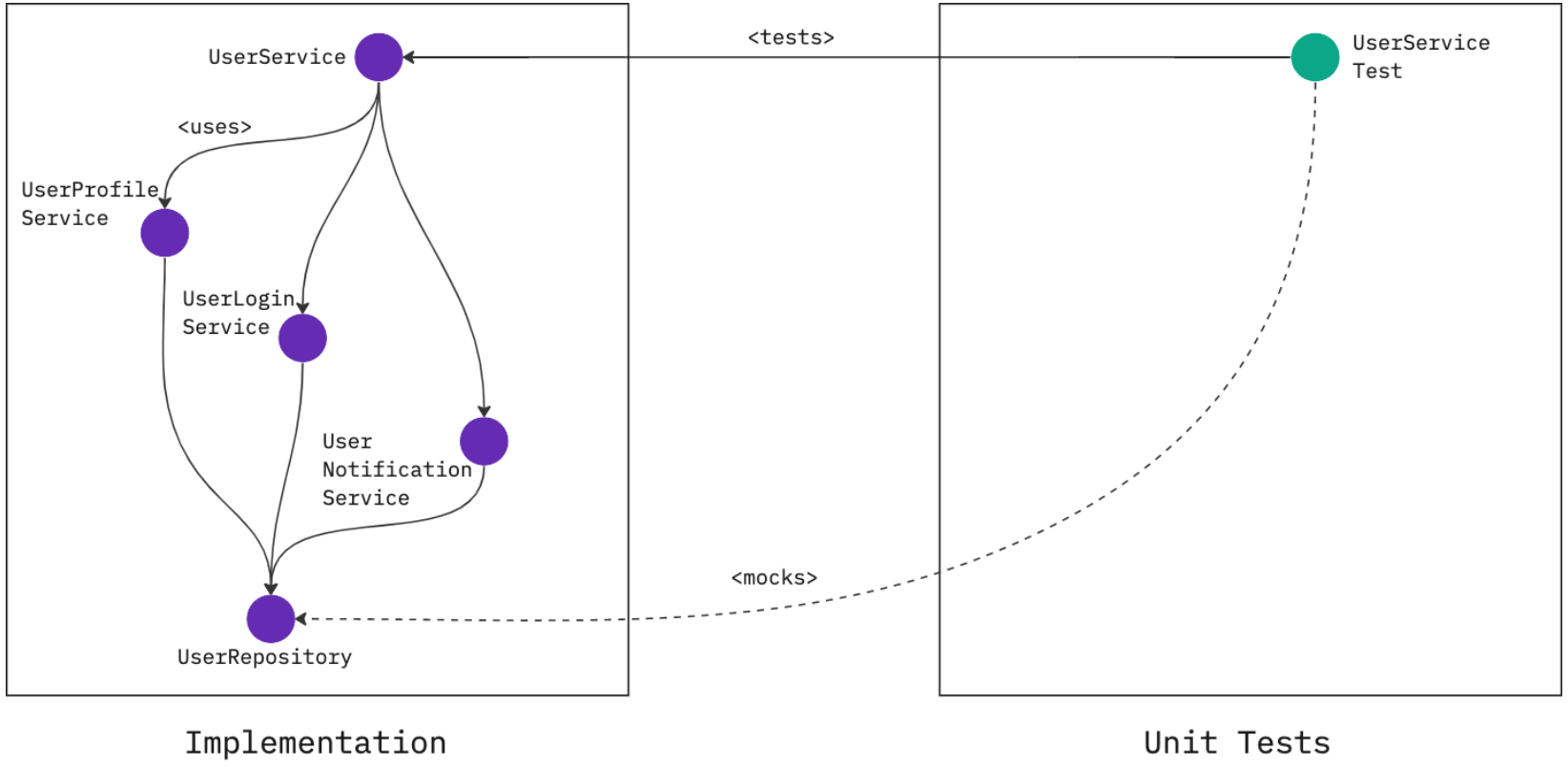
Fragile Test Problem

Tight Coupling = Bad Design



Fragile Test Problem

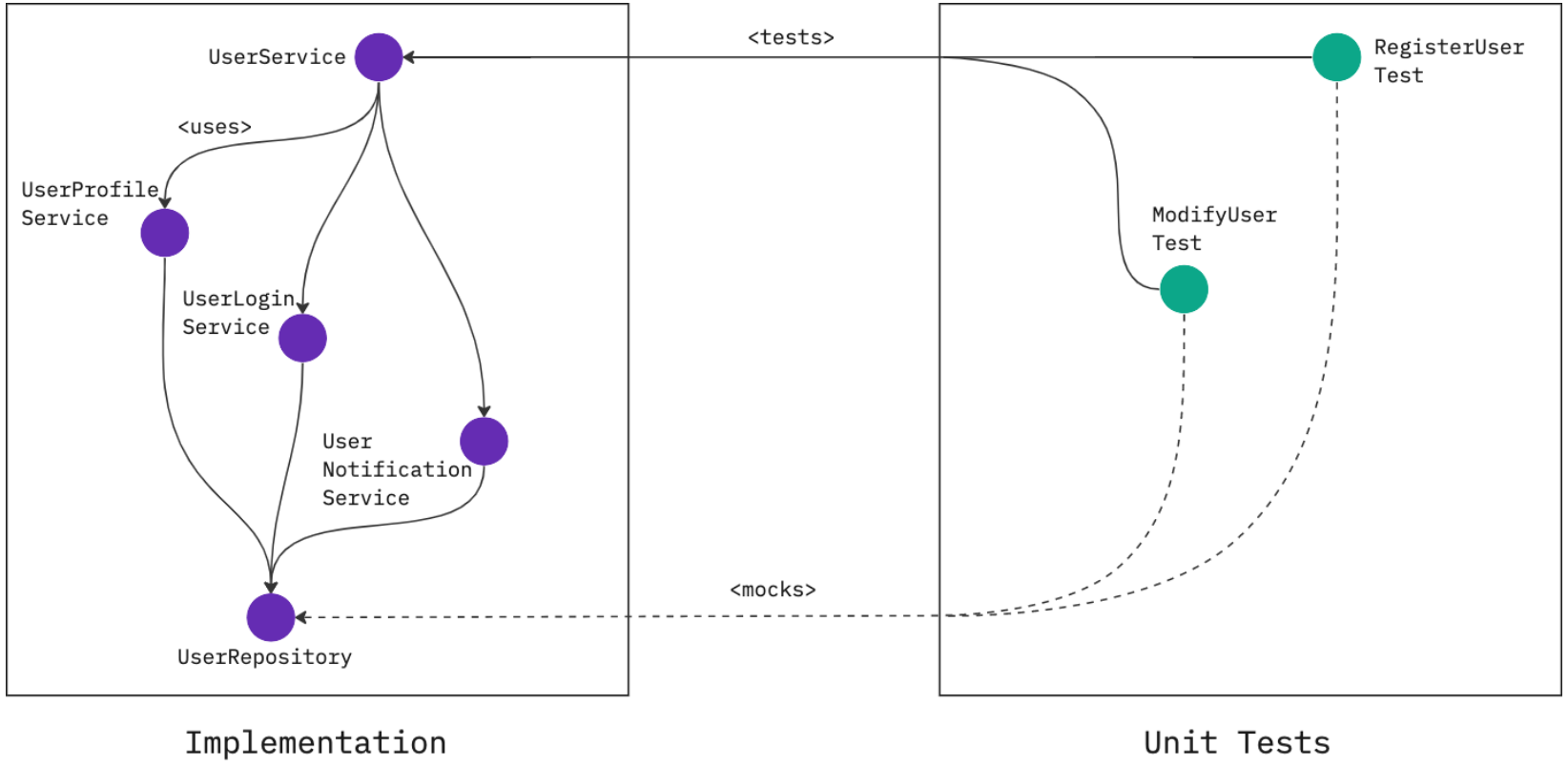
Solution: Loosely coupled tests



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

Fragile Test Problem

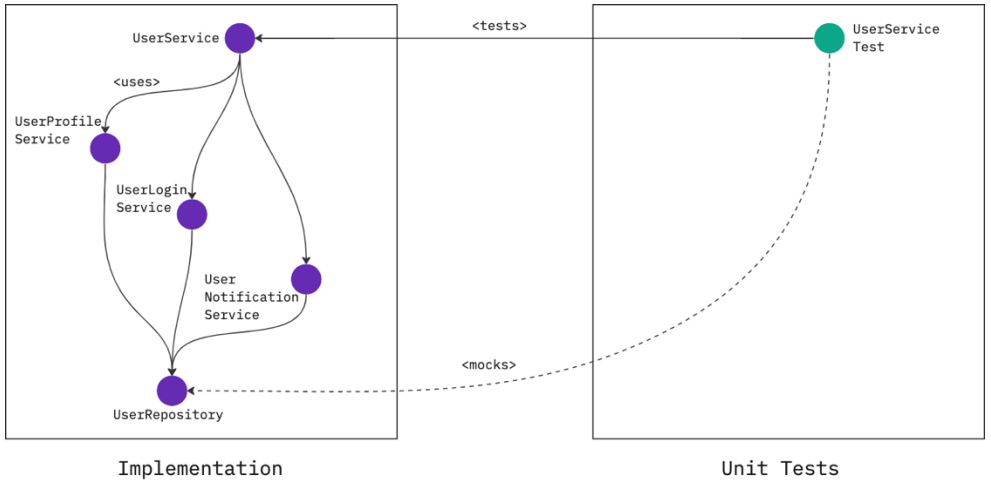
Solution: Loosely coupled tests



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

Fragile Test Problem

But wait.

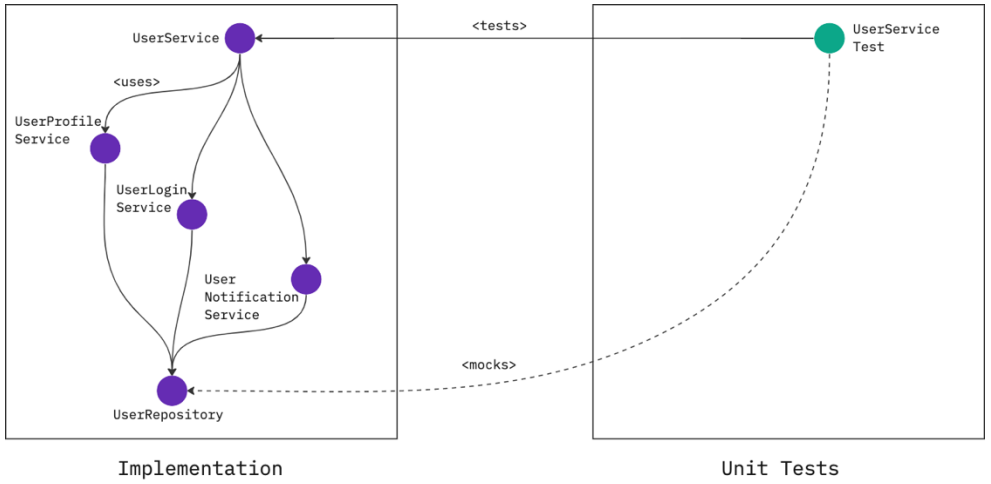


That breaks the rules!

~~For every class `X` there should be a test class `XTest`.~~

Fragile Test Problem

But wait.



Isn’t that indirect testing?

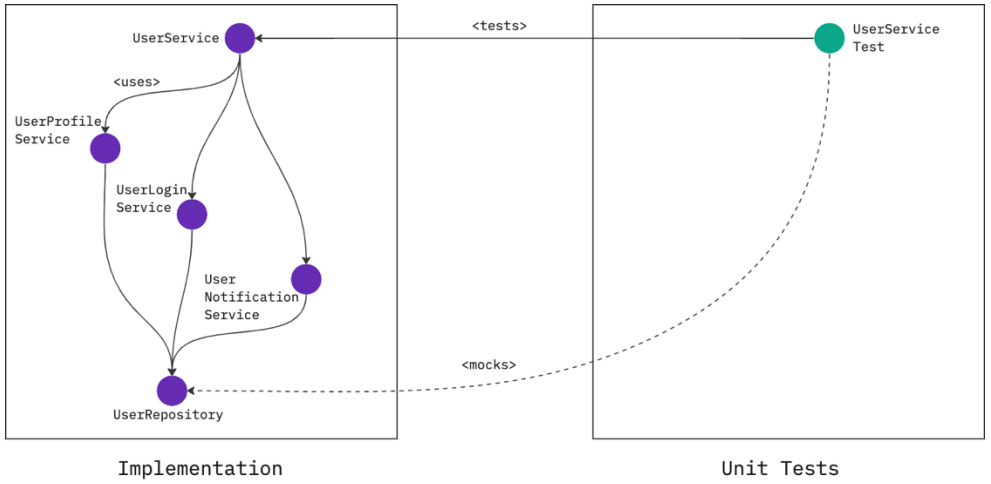
Typical example for indirect testing: Testing through the presentation layer

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

http://xunitpatterns.com/Obscure%20Test.html#Indirect%20Testing

Fragile Test Problem

But wait.



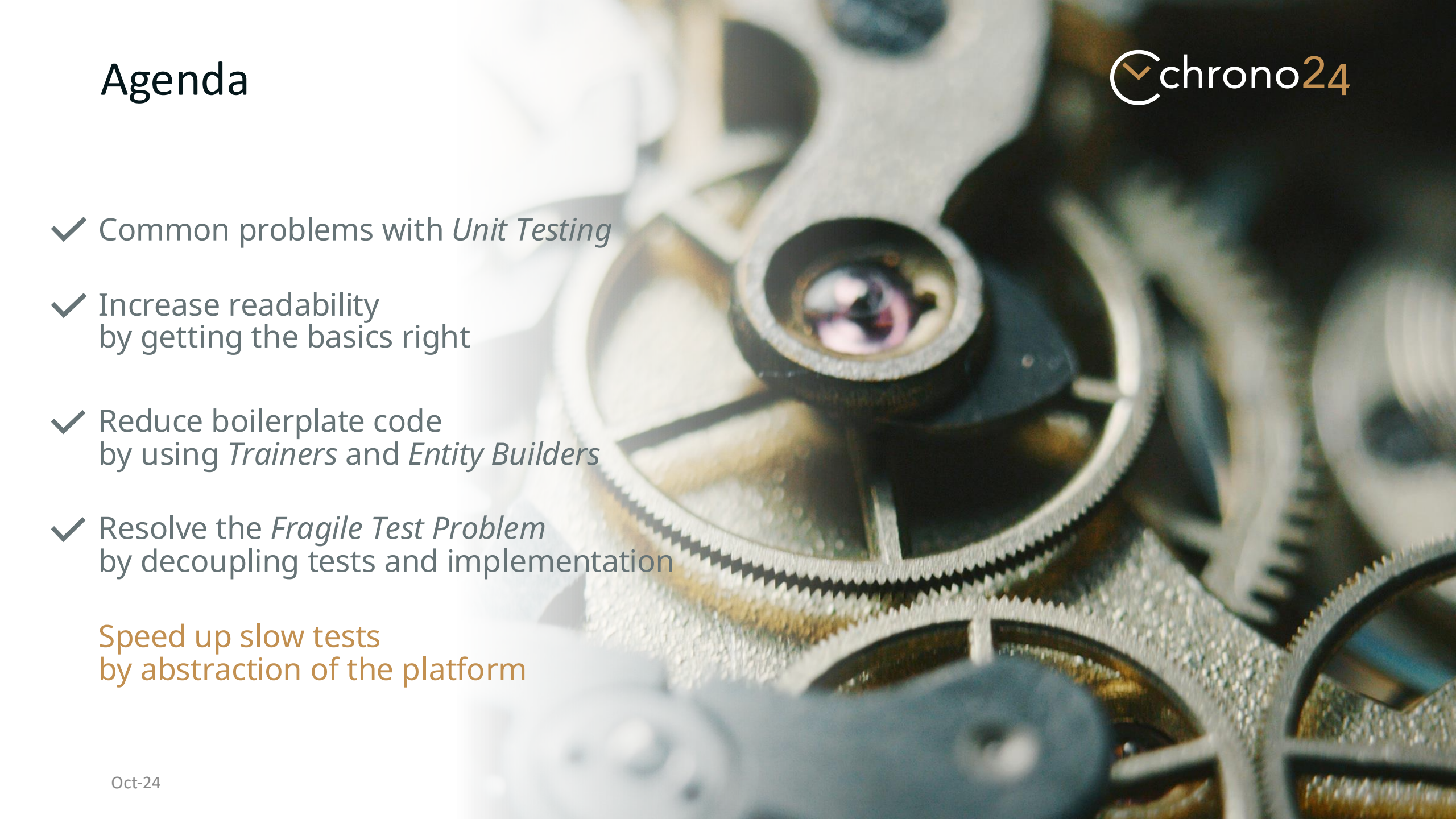
It's much harder now to identify the cause of a failing test!

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, let’s design them that way.

Agenda

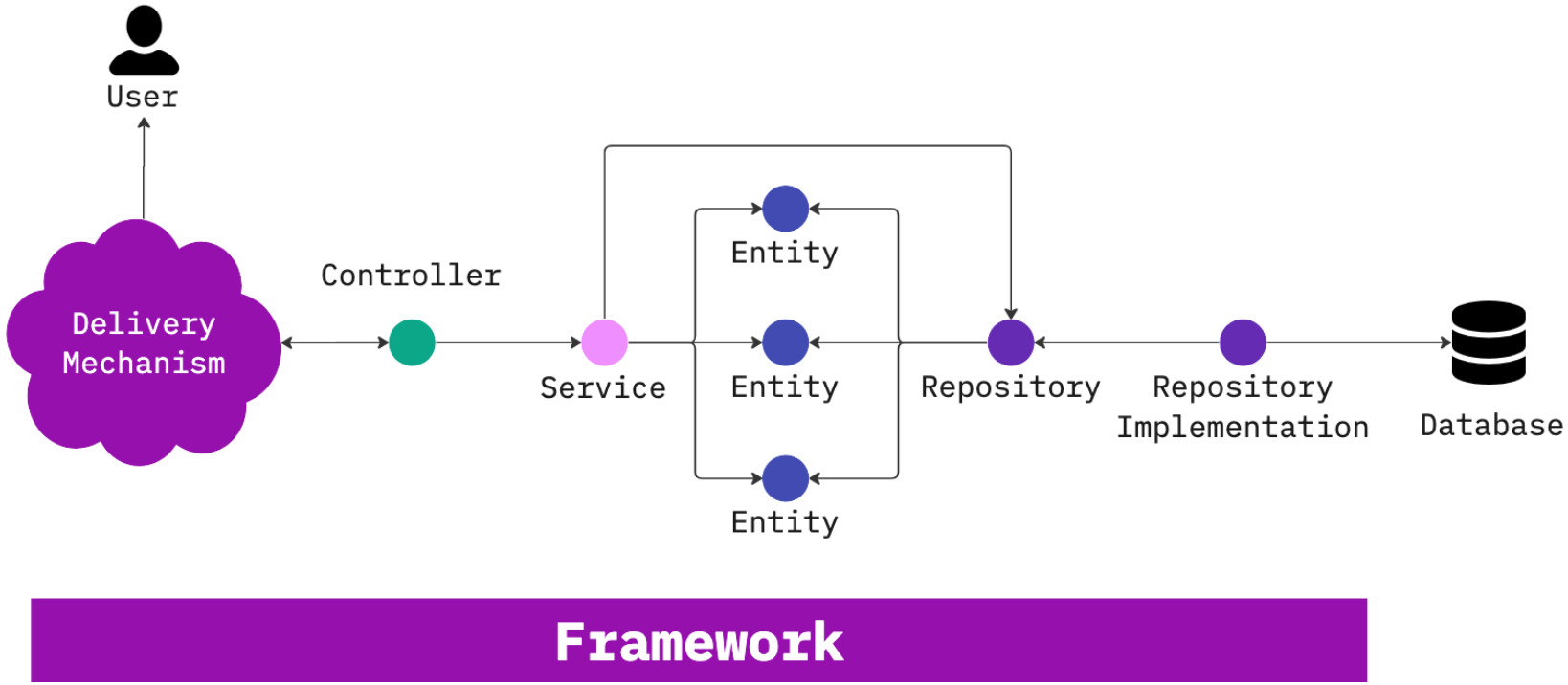


- ✓ Common problems with *Unit Testing*
- ✓ Increase readability
  by getting the basics right
- ✓ Reduce boilerplate code
  by using *Trainers* and *Entity Builders*
- ✓ Resolve the *Fragile Test Problem*
  by decoupling tests and implementation

Speed up slow tests
by abstraction of the platform

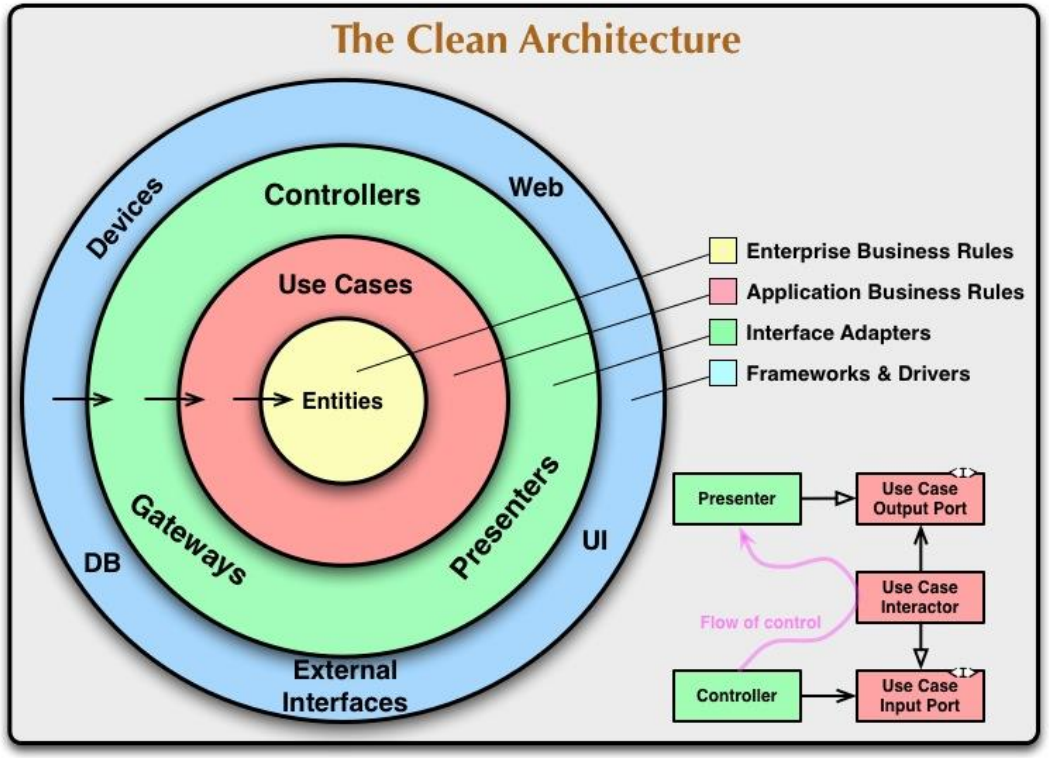
Speed up slow tests

External dependencies slow down test execution



Speed up slow tests

Push all external dependencies outside and only test the logic

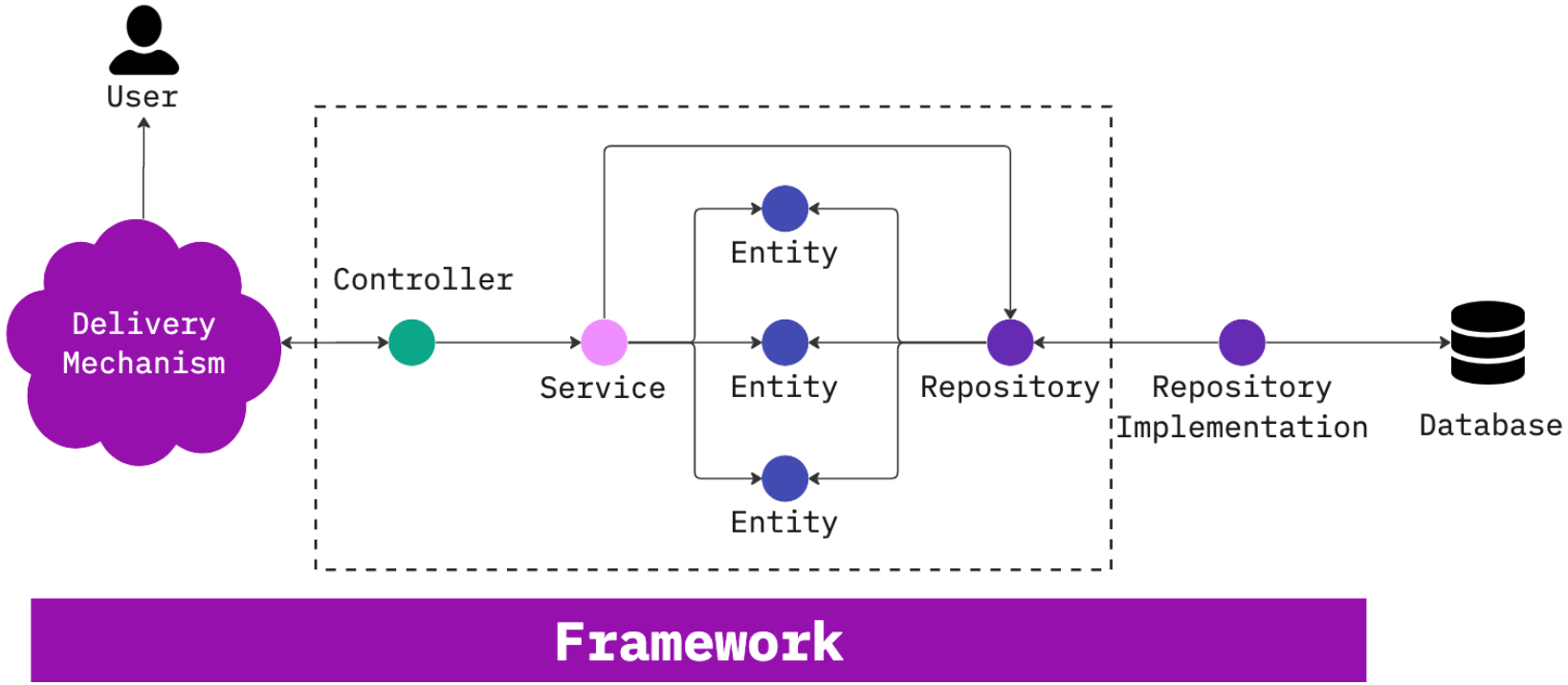


- Dependency Rule
- Outside concrete, inside abstract

-> Intrinsically testable

Speed up slow tests

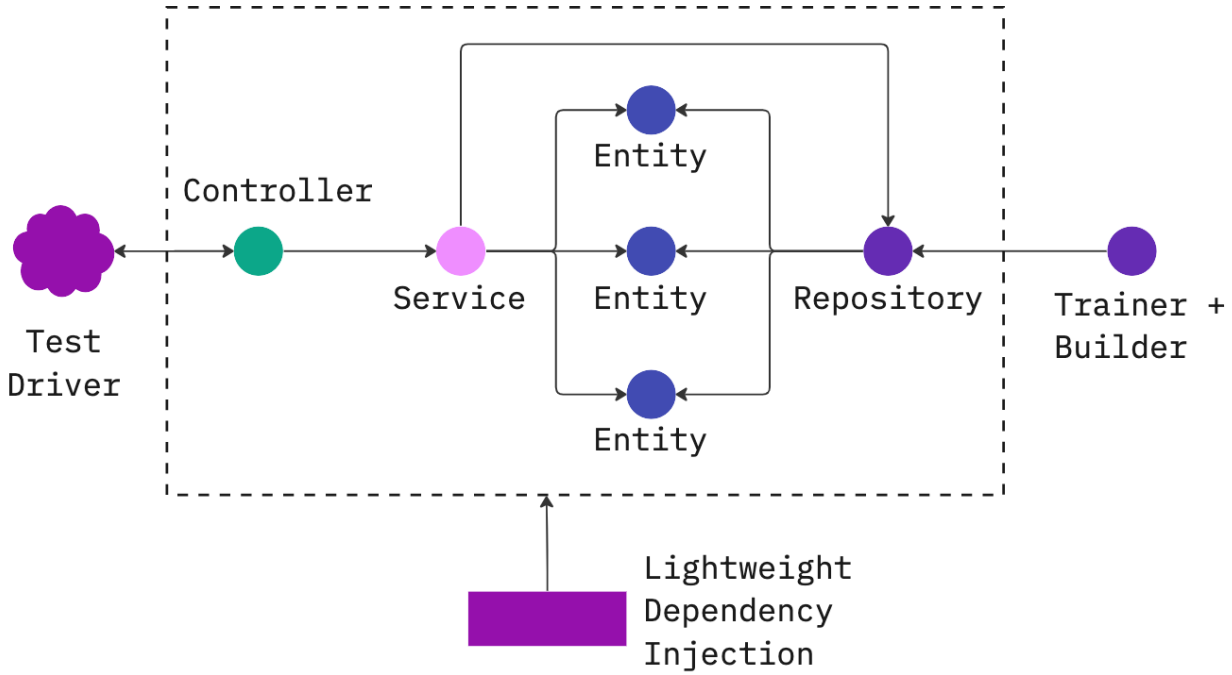
However, the reality is messy.



Speed up slow tests

chrono24

However, the reality is messy. So, let’s deal with it.



Don’t be afraid of tests.

Our journey

… until now

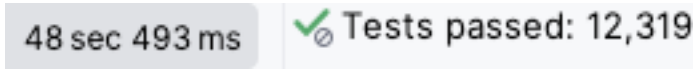
2003 – 2016
- Almost no automated tests

2016 – 2018
- *Test-Driven Development* ideas
- *Entity Builder* and *Trainer*
- Establishment of use case tests

2019
- Move from Jenkins to GitLab CI
- Run tests with every push

2022
- Monorepo with parallel build

2024

→ 

Key Takeaways

chrono24

Getting the basics right

- *Given* | *When* | *Then* gives structure
- Explicit relationship between pre- and post-condition

→ Increased readability

Defining scenarios simplified

- *Entity Builders* allow clear data definition
- *Trainers* simplify the configuration of dependencies

→ Reusable and simple setup

Decoupling test and implementation

- Testing at the borders of what matters at this point

→ Tests and implementation can be structured independently

Separating tests from external dependencies

- Removing and simplifying supporting frameworks to run tests

→ Fast test execution

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

Substack:
https://whattt.substack.com/

LinkedIn:
https://www.linkedin.com/in/jens-happe/

Feedback