EDDSlideshow

Example-Driven Development

Example-Driven Development is superficially like Test-Driven Development, where you drive development by constructing tests methods that return example objects.

It sounds simple, but it actually changes the development process in several fundamental ways.

The Trouble with TDD

With TDD, you develop code by incrementally adding a test for a new feature, which fails.

Then you write the “simplest code” that passes the new test.

You add new tests, refactoring as needed, until you have fully covered everything that the new feature should fulfil, as specified by the tests. **But:** *Where do tests come from?*

When you write a test, you actually have to “guess first” to imagine what objects to create, exercise and test.

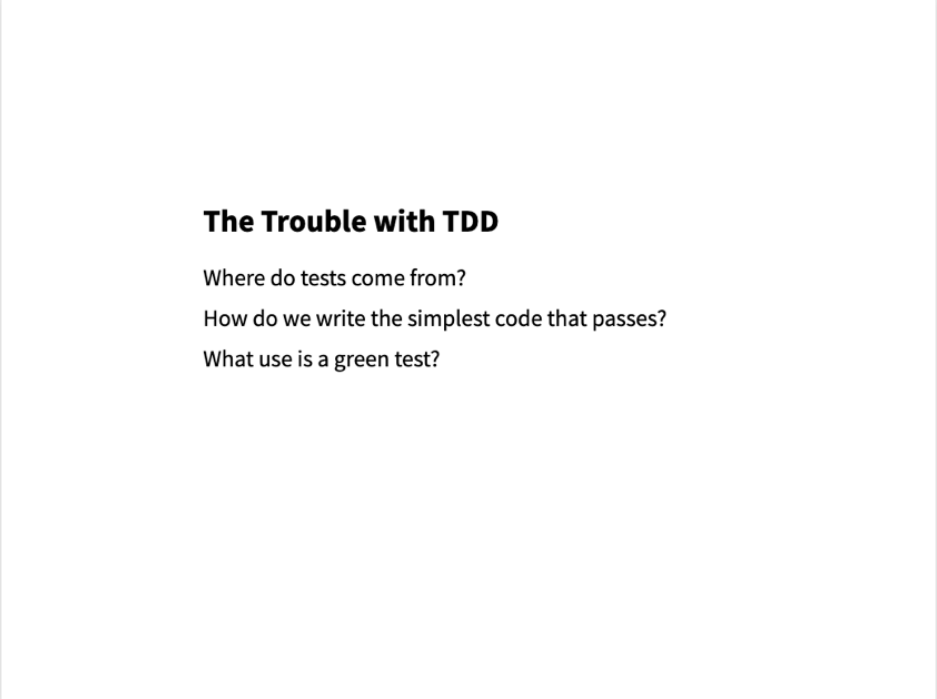
How do we write the simplest code that passes?

A test that fails gives you a debugger context, but then you have to go somewhere else to add some new classes and methods.

What use is a green test?

Green tests can be used to detect regressions, but otherwise they don't help you much to create new tests or explore the running system.

With Example-Driven Development we try to answer these questions.

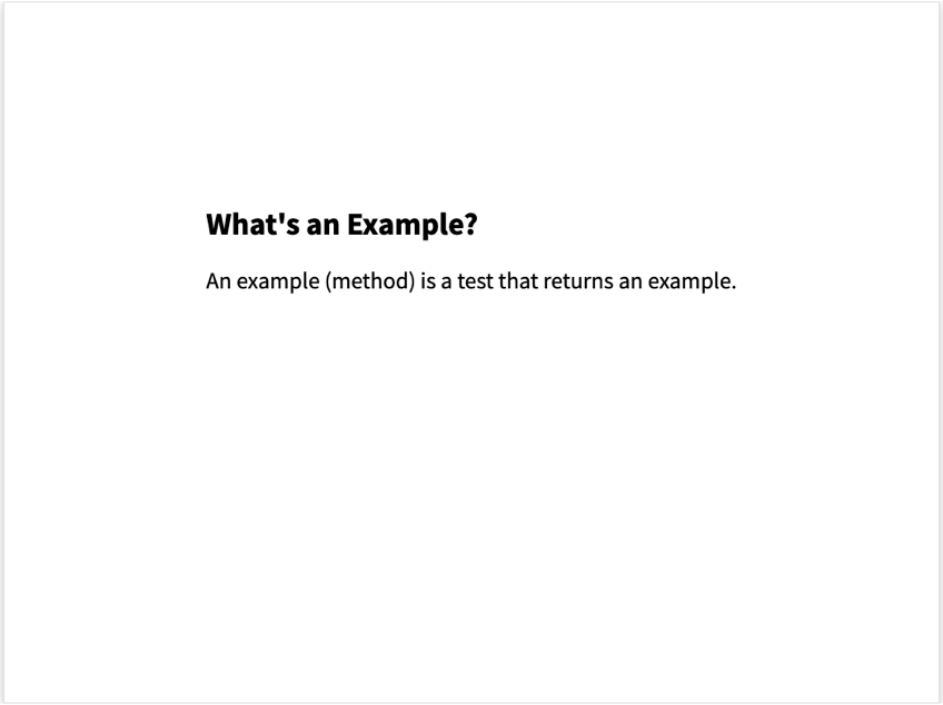


What's an Example?

An example method is just a test method that happens to return the object being

tested.

Through this simple change, instead of a passing test simply being green, we get back an object that we can inspect, explore, and reuse for various purposes.

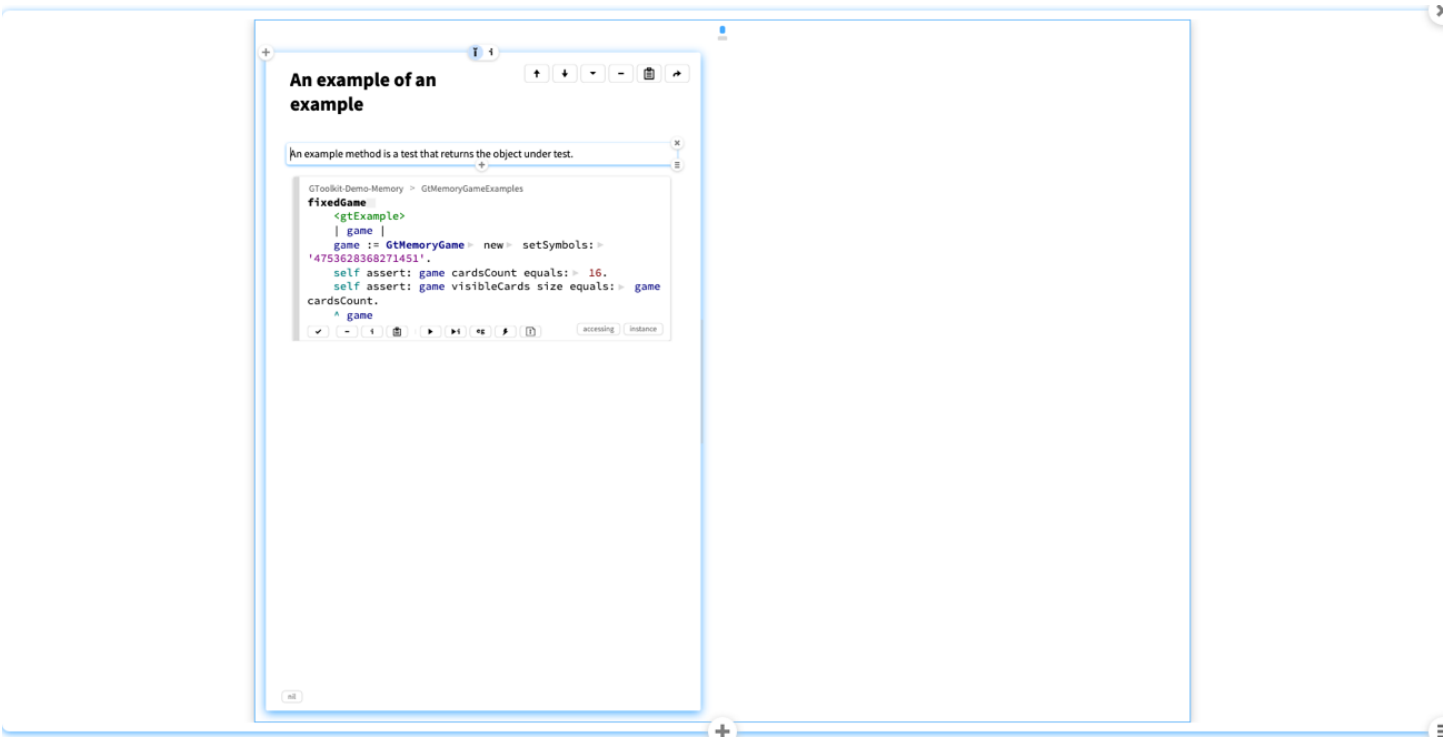


An example of an example ...

Here we see a simple example of an example method. It is annotated with a `<gtExample>` pragma to flag it as an example.

Like any test, it has a setup, which in this case creates a `game` object. We check some assertions, in this case perform no further operations, and then we return the object under test.

This allows not only to carry out the tests, but also to inspect the result.



Composing examples

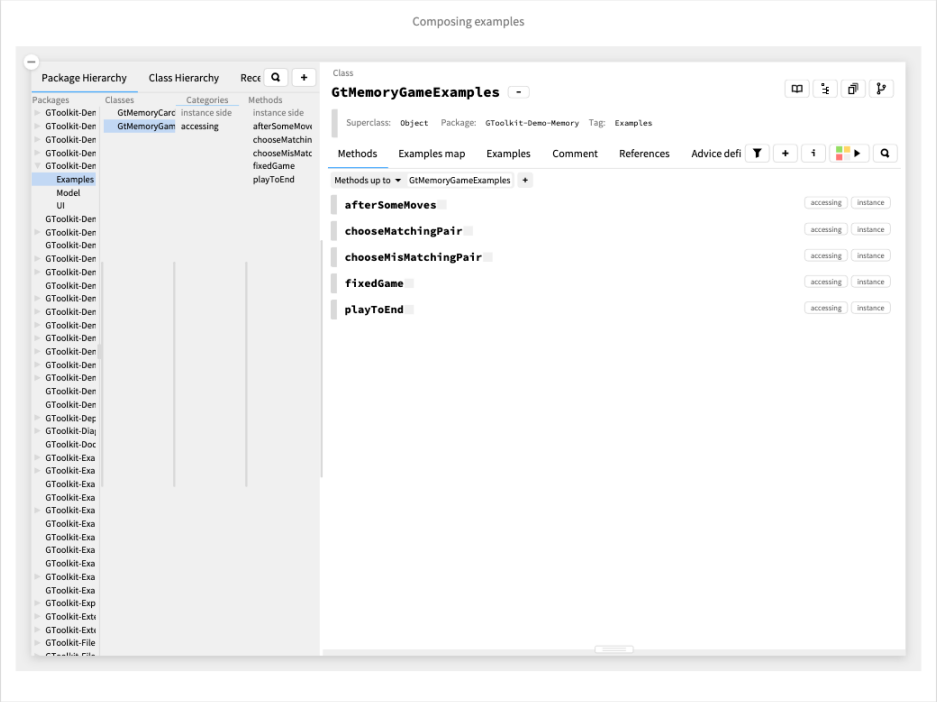
Once we have an example, we can also use it as a setup for another example.

`chooseMatchingPair` is another example method that starts with `fixedGame` as its setup. As in a conventional test, we can check some preconditions, perform one or more operations, and then check some postconditions.

The difference, again, is that we return the object under test, so we can explore it.

We can also reuse it as a setup for yet another example, in this case, `playToEnd`.

If we switch to the *Examples map* view, we can see all the dependencies between the examples.



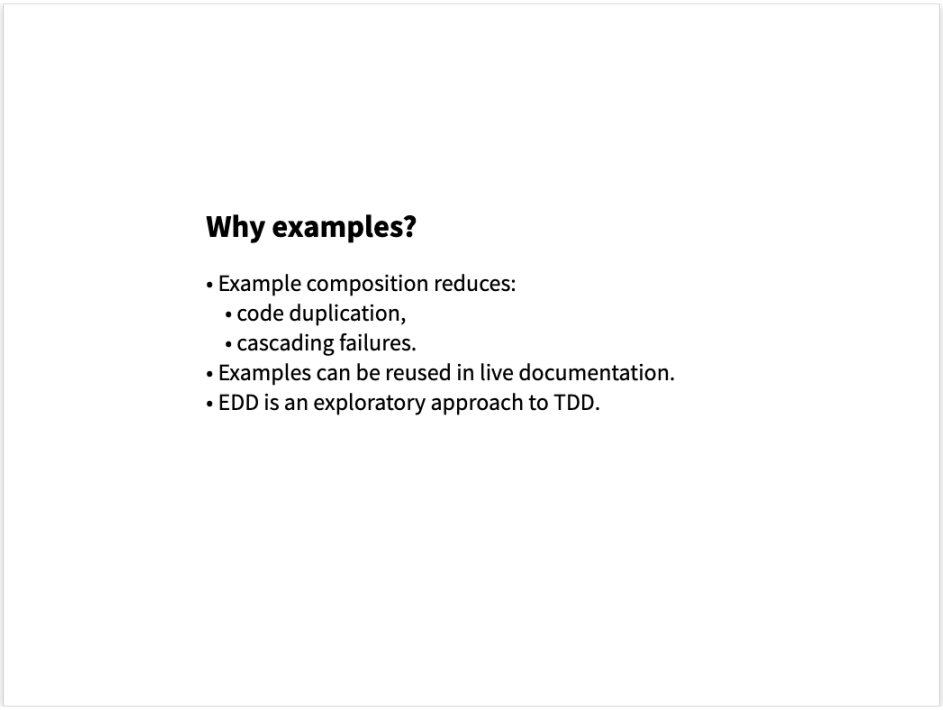
Why examples?

What are example methods good for?

As we have seen, examples make dependencies between tests explicit by reusing examples as setups for other examples, thus formaing a hierarchy of examples.

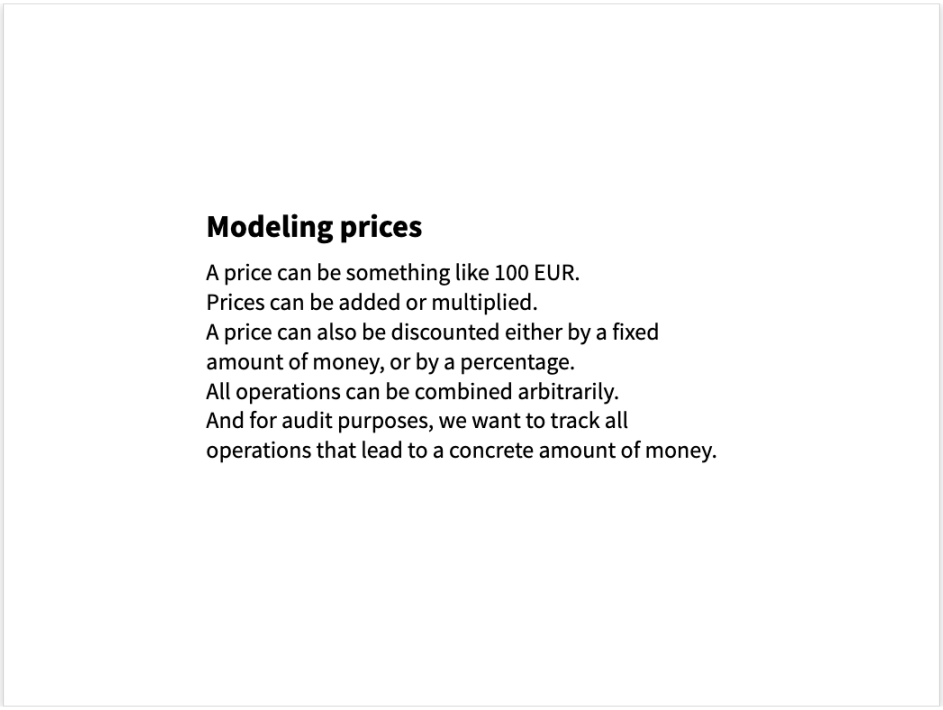
Best practice in test design supposedly should avoid dependencies between tests, but studies have shown that this is a lie. In reality tests without dependencies lead to a great deal of duplicated code, and cascading failures due to the same setups being repeated in numerous tests. By factoring out the commonalities as examples, the duplication is removed, and cascading failures are avoided.

A further benefit is that examples can be used in live documentation, and, as we shall see, examples support an exploratory approach to test-driven development, that we call example-driven development, or EDD.



Modeling prices

Let's work through an example where we want to model *prices* for goods, that may be *discounted* by fixed amounts, or percentages, or even combinations of different types of discounts.



Money classes

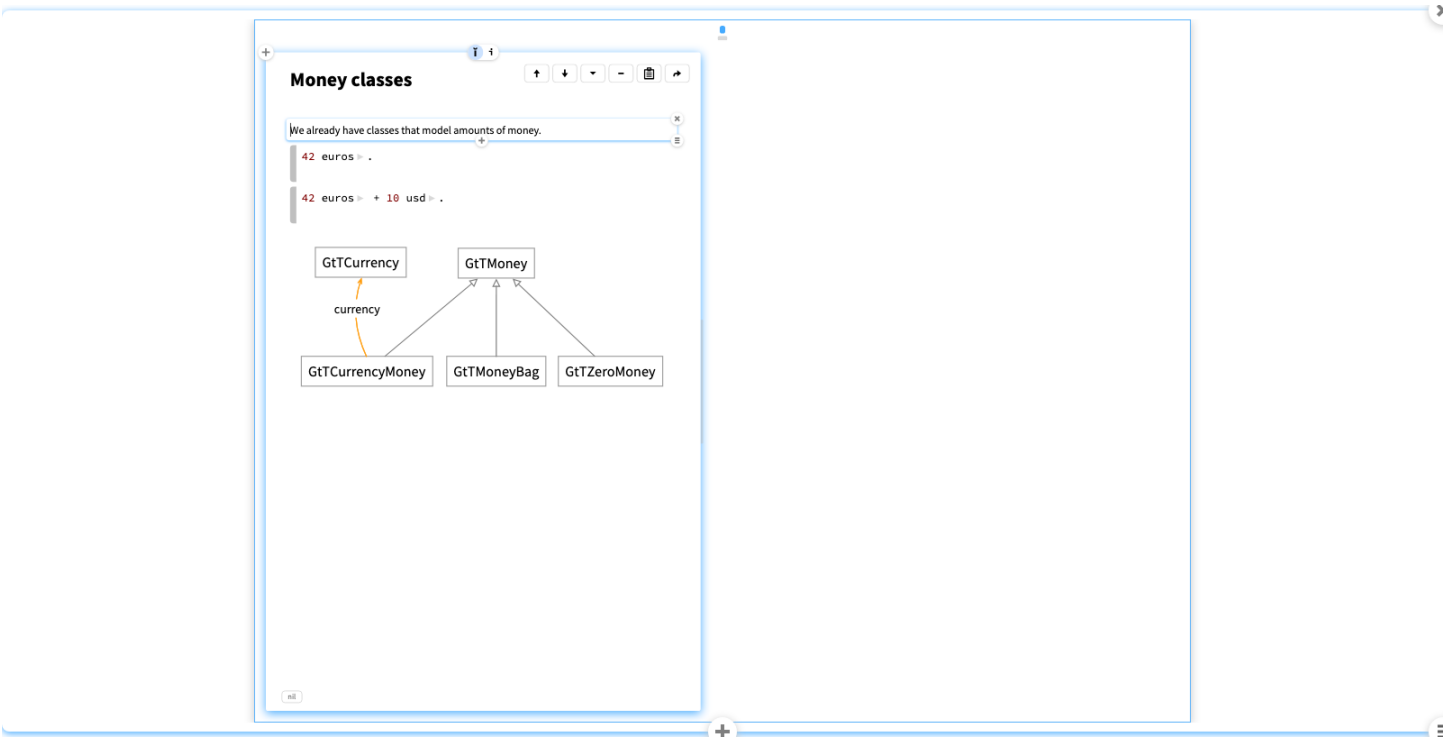
To simplify our task, we assume that we already have classes that model different amounts of money, such as 42 € or 10 USD.

An amount of money is always in a *currency* such as euros or US dollars.

A *bag* of money consists of amounts of mixed currencies.

A *zero* amount of money doesn't have a currency.

All these classes have a common abstract *Money* superclass for shared behavior.



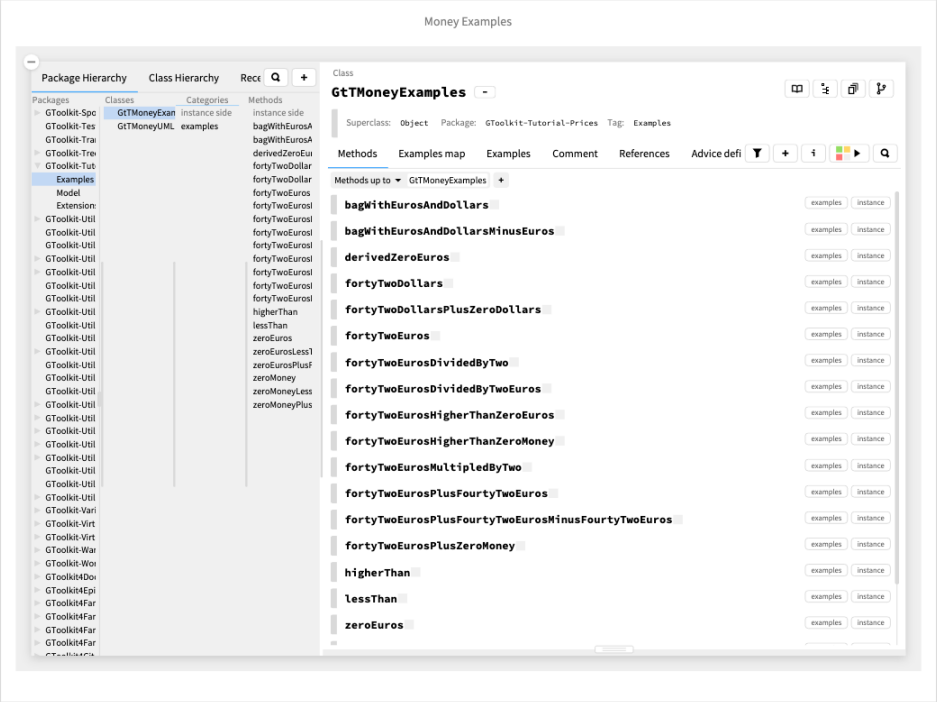
Inspect the `42 euros` snippet. Inspect the `42 euros + 10 usd` snippet. Click on the GtTZeroMoney class. Click on the GtTMoney class.

Money examples

The money classes are heavily covered by examples, which are essentially unit tests that also return example objects.

This means that a passing test is not just green, but also returns an object that can be explored, reused as a setup for another example, or embedded into live documentation.

Unlike tests, however, examples don't come “first” but they are extracted during the example-driven development process.



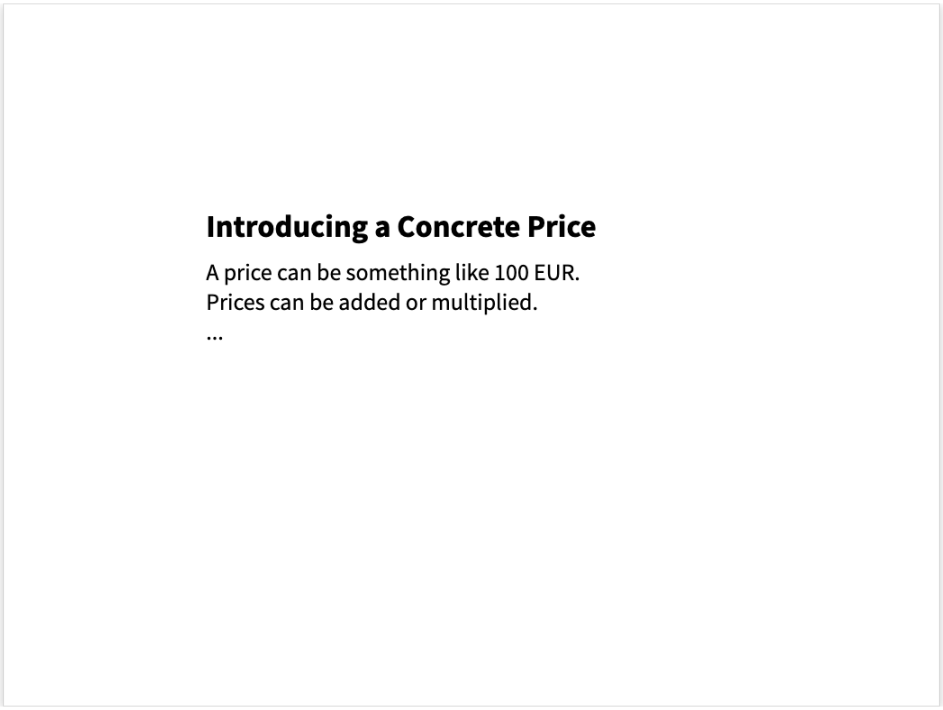
Run all the examples. Inspect the first example. Open the code bubbles to see how they are composed. Go to the Examples map to show all the dependencies.

Introducing a Concrete Price

Just like we have a hierarchy of Money classes, we expect to end up with a hierarchy of Price objects, including an abstract root class, a concrete, fixed price, and several kinds of discounted prices.

Instead of designing this hierarchy up-front, we'll develop it incrementally, driven by examples.

We'll start with an example of a concrete (as opposed to an abstract) Price object.



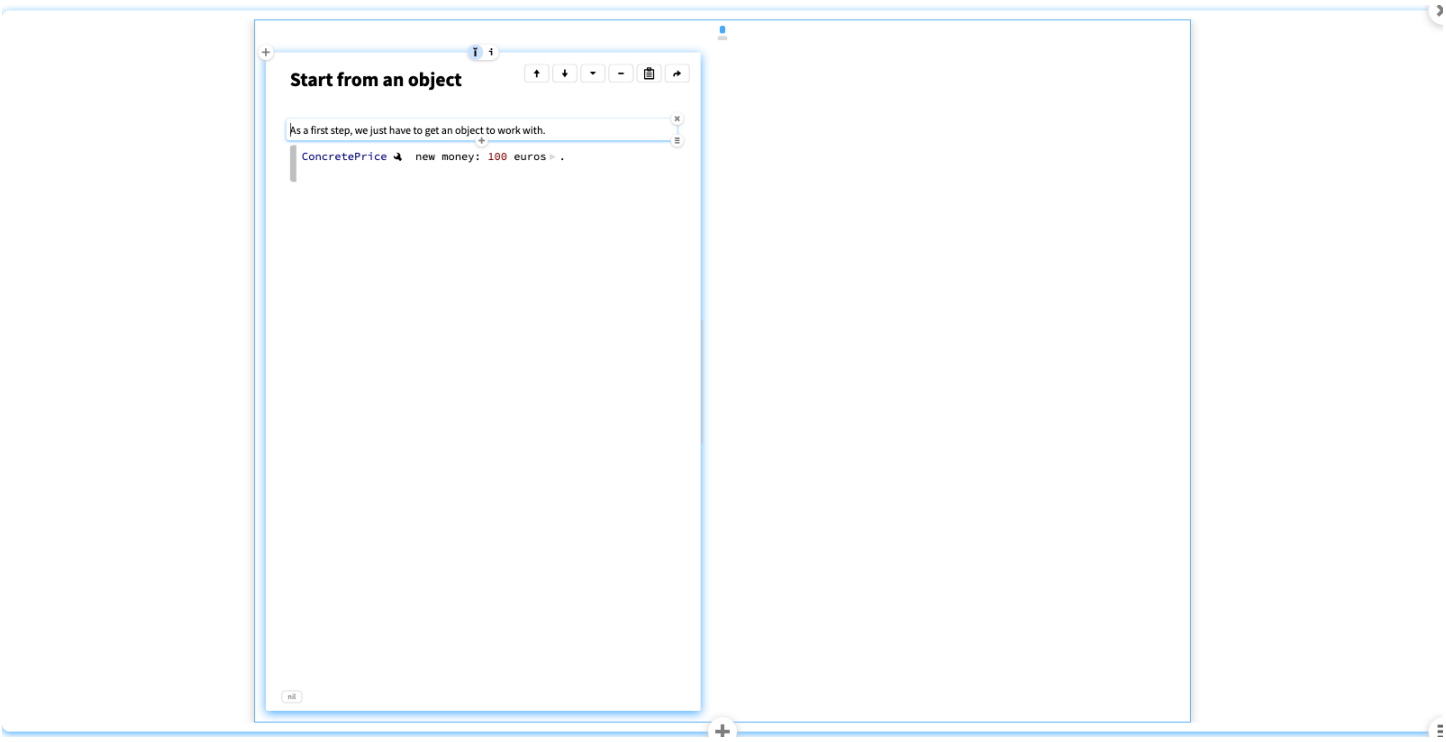
Start from an object

Instead of starting by imagining and writing a test case as an example method, we start by creating an instance of the class we need.

We first simply ask how we want to create our concrete instance of a price, and we write that code in a snippet.

Neither the class nor the constructor exist, so we create them as fixit operations.

Now we have a first concrete Price object!



Create the ConcretePrice class as a fixit. Give it the `EDDPrices` package, the `Model` tag, and a `money` slot. Create the accessors. Change the argument of `money:` to `aMoney`. Inspect the result. Inspect its `money` slot.

Create a factory method

We would like to be able to create a price object by sending `asPrice` to a Money instance.

We start by inspecting the Money instance.

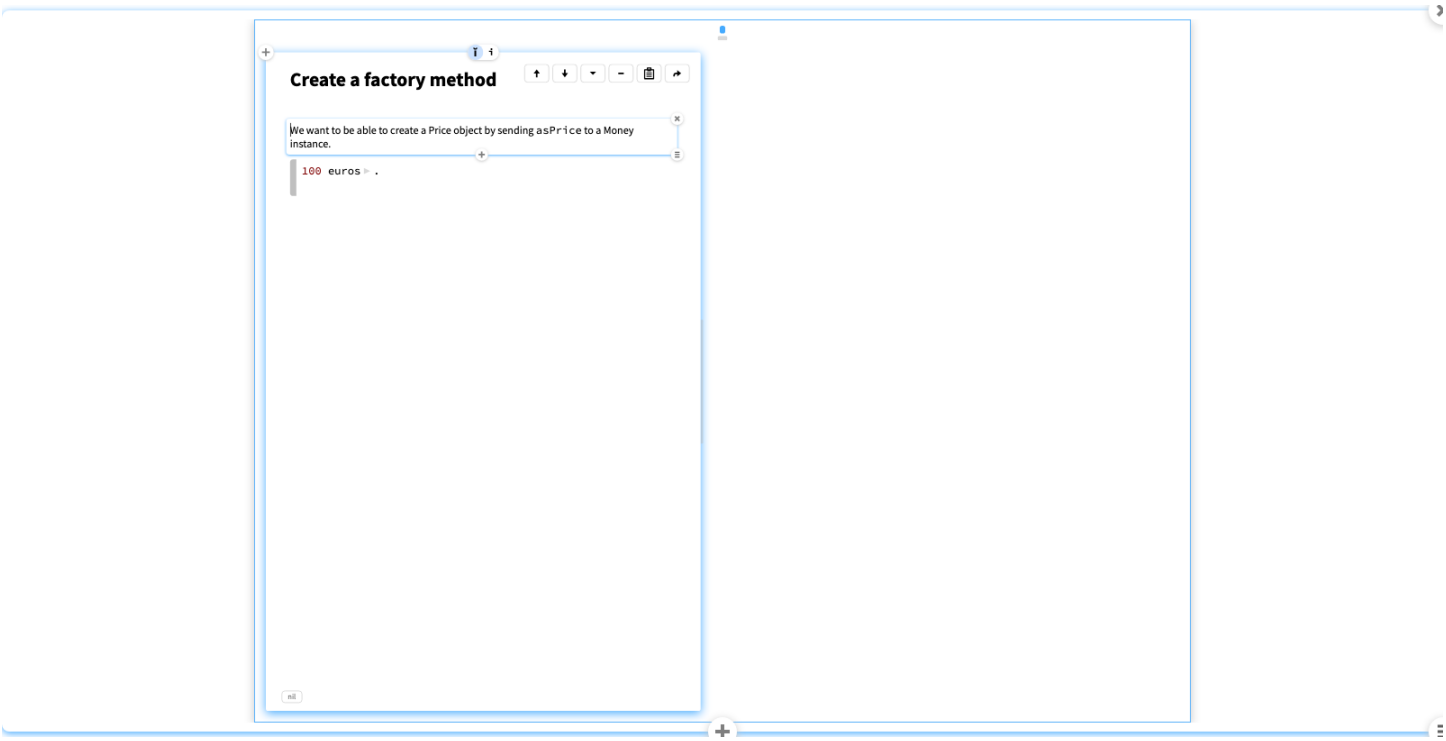
We prototype the code to create the Price instance.

We try it out.

And we extract the factory method.

We change the method to be an extension from the `EDDPrices` package.

Now we can simply write `100 euros asPrice`.



Inspect the money. Open the playground. Code up `ConcretePrice new money: self; yourself` Inspect the result. Extract the `asPrice` method. Browse the new method. Change the category to `*EDDPrices`. Go back to the page and change the code to `100 euros asPrice` and inspect it.

Adding a view

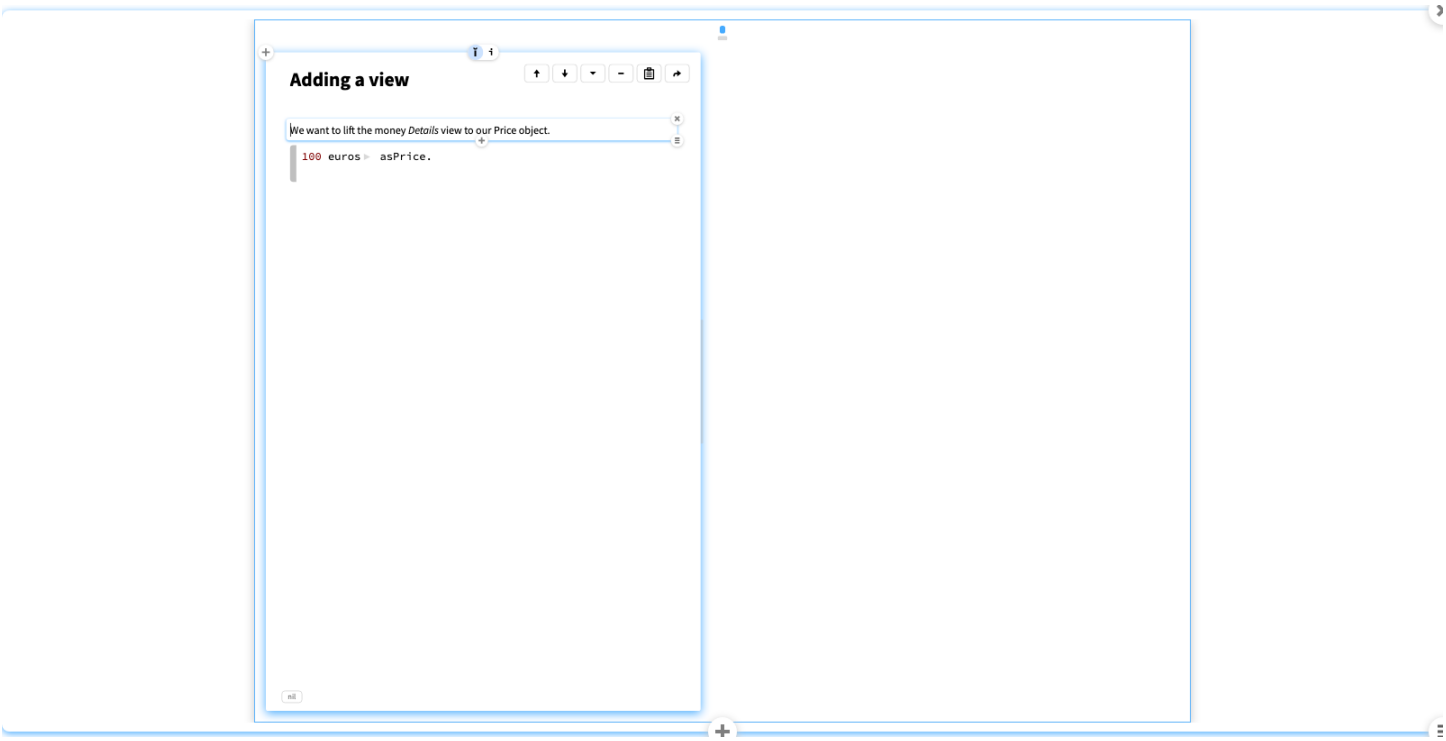
Our new Price object has only an ugly generic view, but its money slot has a nice view we could reuse.

We go to the `Meta` view of the Price object and add a new view method that forwards itself to the Details view of its `money` slot.

A view is just a method that takes a view object as an argument, has a `<gtView>` pragma, and uses the view API to create the view we want, in this case a `forward` view.

We set the title of the view to `Money`, the priority to 10 so it appears early in the list of views, the object we want to forward to is the money slot, and the view is its `gtDisplayFor:` view.

The moment we commit the view code, the view becomes available.



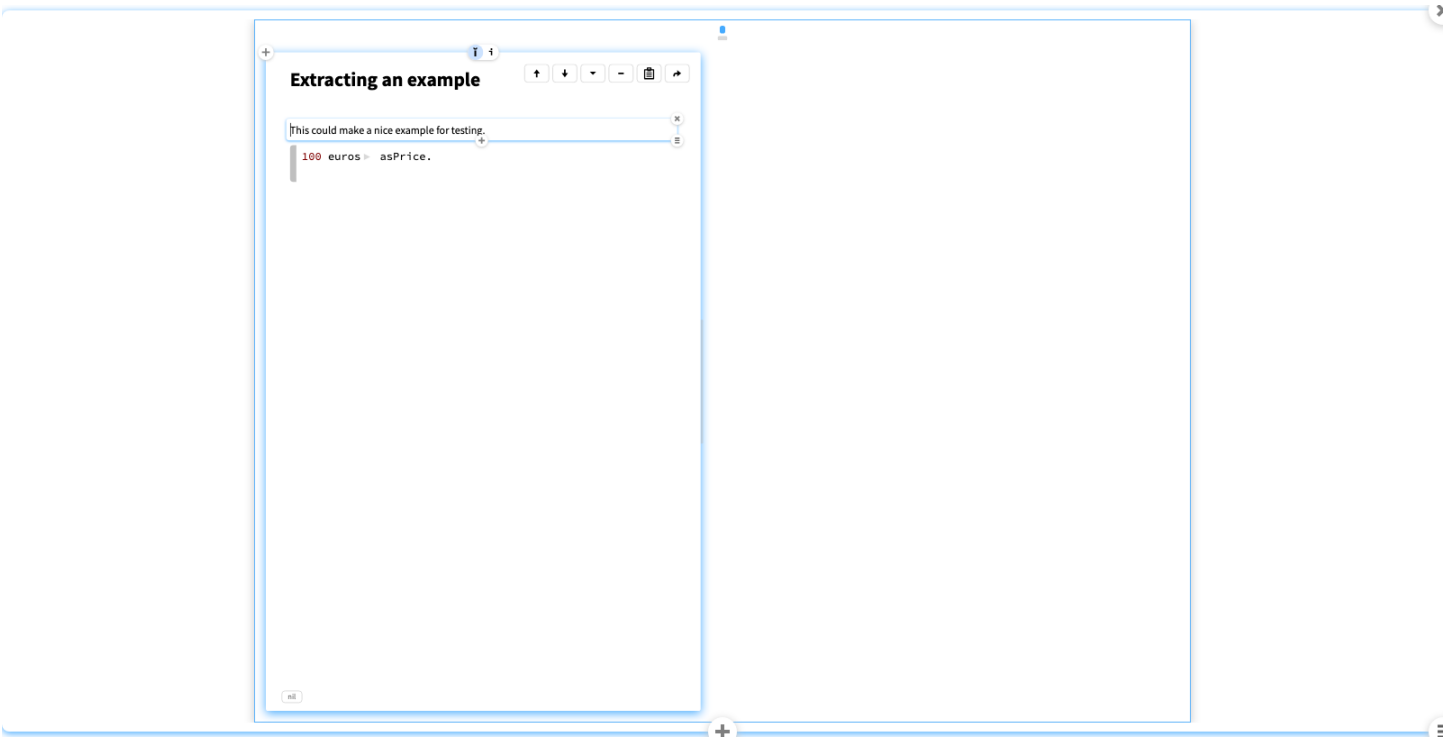
Inspect the price object. Inspect the money slot and view the `Details` view code. Copy the method name. Go back to the Price inspector and switch to the `Meta` view. Add a forwarding `gtMoneyFor:` method. Show the new view.

Extracting an example

At this point it looks like we have a nice example for testing, so let's extract it as an example.

We introduce a new class to hold our examples, and give the example a suitable name.

Note that the extracted example method has a `<gtExample>` pragma, and unlike a usual test method, it returns an instance.



Select all the code, right-click and *Extract example* Set the receiver to `PriceExamples` and the selector to `hundredEuros`. Choose `EDDPrices` as the package and `Examples` as the tag. Accept the refactoring. Inspect the result. Browse the code bubble.

Adding assertions

We now have an example, but we aren't testing anything yet.

Rather than directly adding tests to the example method, let's explore first.

We expect that a price object should equal another price object with the same money value.

We see that this fails.

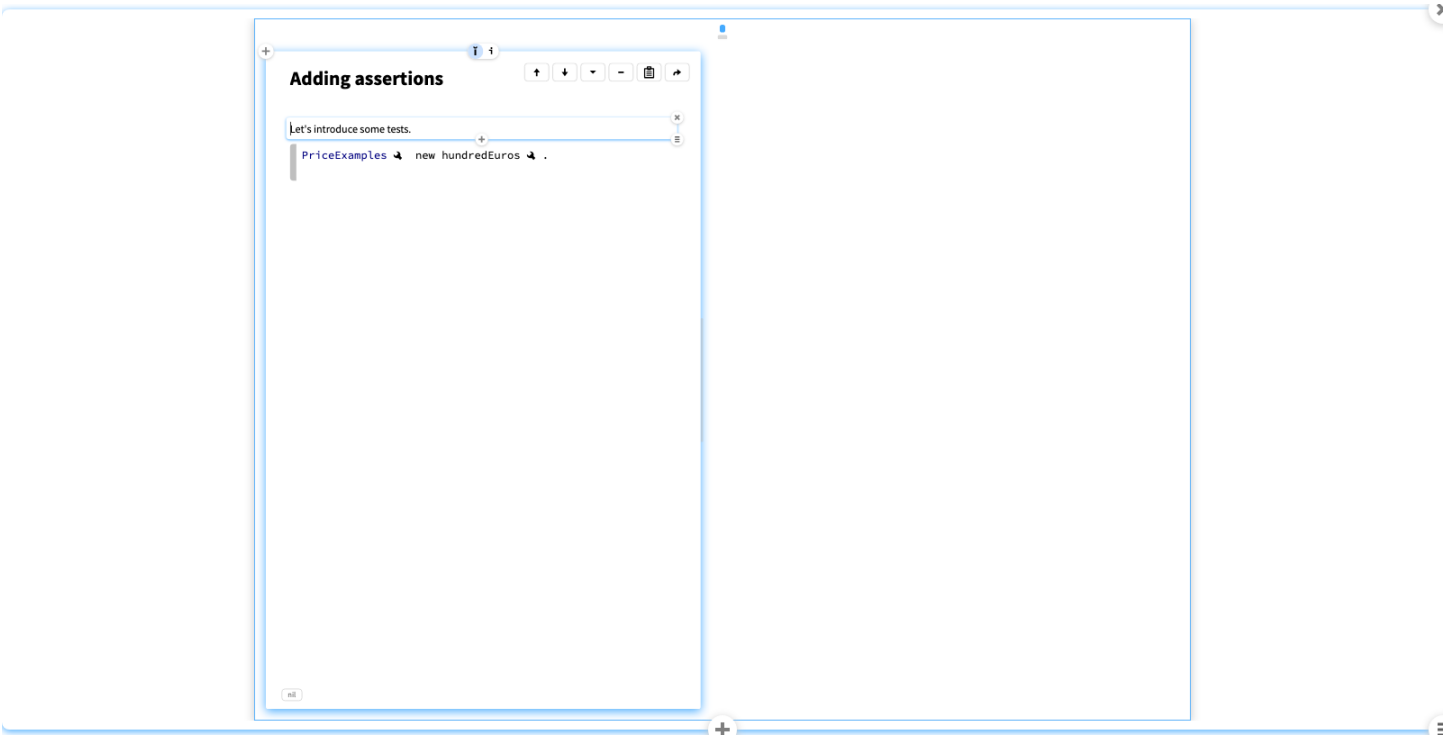
Let's have a look at the `=` method. It's testing for object identity.

Now let's see what happens if we directly compare the `money` slots.

This passes. We see that Money has implemented `=`, so we should do the same.

We have the code we want right here, so let's extract it.

Now we can go back to the example and add a test.



Inspect the example and open the playground. Evaluate `other := 100 euros asPrice. self = other` and see it fails. Open the code bubble for = (alternatively search for = in the Meta view). Evaluate a new snippet `self money = other money` and see it passes. Browse the = method of money and see it has been overridden. Rewrite the `hundredEuros` example, adding the assertion.

EDD in a Nutshell

Summing up, instead of starting by writing a test, we first create a live object to explore.

We prototype any behavior in the playground of the live object, and then extract methods that work.

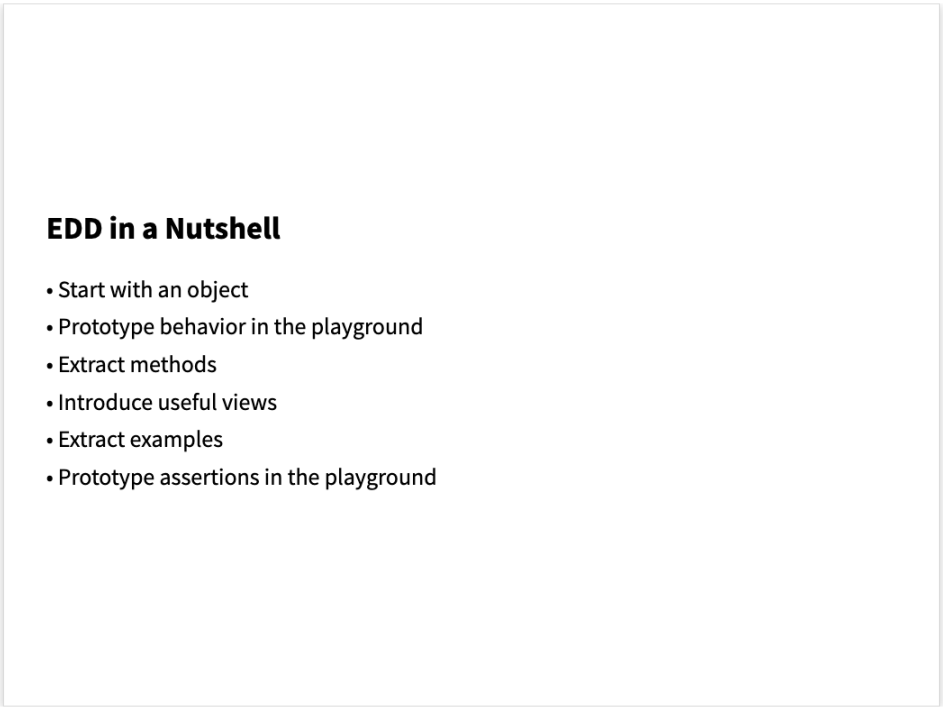
We create views that explain what is interesting about the object.

We extract interesting instances as example methods of a dedicated examples class.

We prototype tests in the playground of the live example, before adding them as

assertions to an example.

We iterate until we're done!



We leave room at the right of the slide for overviews of the points.