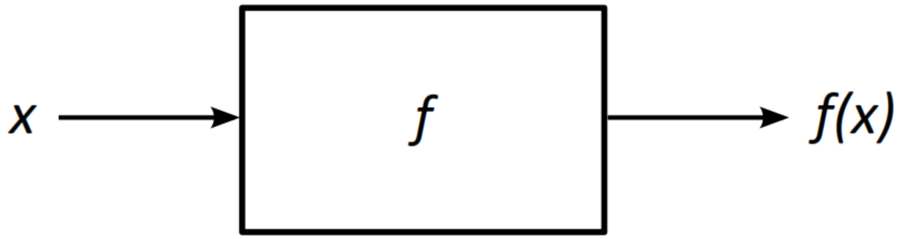
EDUMENT

Referentially Transparent Business Logic

What is referential transparency?

A scary way of saying that a piece of code **produces results solely based on its inputs**, not on any external state.



We already knew objects without dependencies are easy to test. **Outputs are purely based on inputs** that the test controls.

When things got more complicated, we turned to dependency inversion. By controlling the dependencies, our tests produced **outputs purely based on inputs and dependent objects**.

MVC was easy to test because **action methods produce outputs based on their inputs**.

Nobody expects the FP inquisition

Most developers today focus on **object-oriented programming**, which is about objects with state. This feels very different from **functional programming**, which is much more about an absence of state.

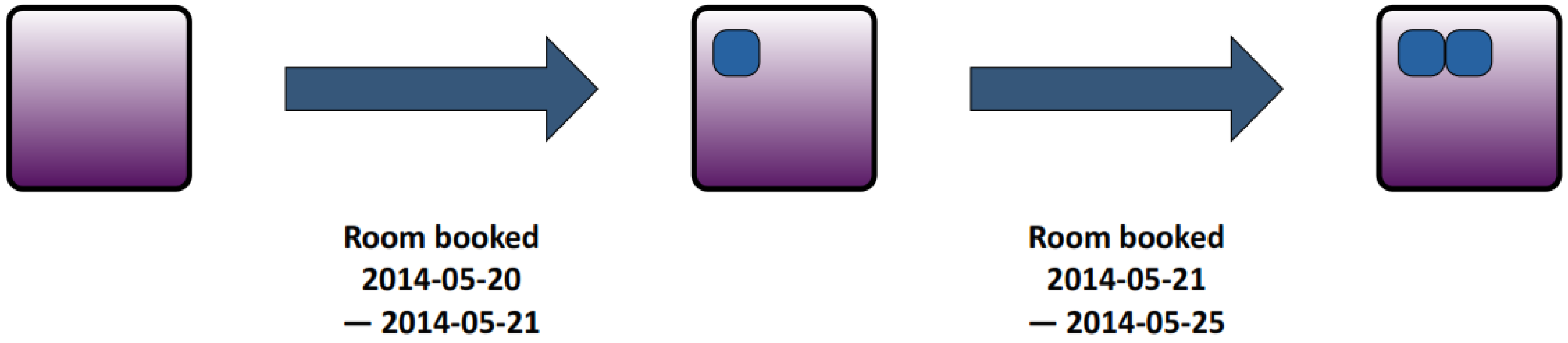
However, objects don't just have state. They **encapsulate state**. And, if an object builds state solely from arguments to its methods and **results of method calls on its dependencies**, then any **output** it can produce is **based completely on its previous inputs**.

Some of the best object-oriented design principles -- especially those that we turn to for help with testability -- boil down to taming state and making our objects more functional in nature!

So what if we took this further?

One may wonder if we can represent the state of such a "pure object" by simply recording all the calls made on it. However, that's not quite right; **an object may reject a method call by throwing an exception** if it would break some invariant.

We could, however, represent object state as **a series of events**, each one recording an accepted state change to the object.



EDUMENT

BDD

Enter BDD

Behavior **D**riven **D**evelopment builds on TDD. It encourages us to focus on **behavior** and **use cases** in our tests, rather than the state
inspection more typical in classic TDD.

BDD tests are typically structured into three phases:

- **Given** a certain state has been established
- **When** we do something
- **Then** we expect a certain outcome

Furthermore, BDD tests are meant to be **declarative** in nature, rather than imperative.

Enter BDD

Behavior **D**riven **D**evelopment builds on TDD. It encourages us to focus on **behavior** and **use cases** in our tests, rather than the state
inspection more typical in classic TDD.

BDD tests are typically structured into three phases:

- **Given** a certain state has been established
- **When** we do something
- **Then** we expect a certain outcome

Furthermore, BDD tests are meant to be **declarative** in nature, rather than imperative.

How can we turn our tests into data?

Turning a method call into data

A method call has a name and a set of arguments. Therefore, we can describe it using a DTO, whose name indicates the operation and whose fields carry the arguments.

```
public class MakeAppointment
{
    public byte Hour;
    public byte Minute;
    public int DurationInMinutes;
    public string Name;
}
```

Notice how the name contains a **verb in the imperative mood**.

We'll call this a **command**.

Describing state changes as data

Next, we want a way to describe the state changes that the object performs in response to a method call, or command. Here, a simple DTO will also do.

```
public class AppointmentMade
{
    public byte Hour;
    public byte Minute;
    public int DurationInMinutes;
    public string Name;
}
```

Notice that this time the name contains a **verb in the past tense**.

We'll call this an **event**.

BDD with commands, events and exceptions

Now that we have found a way to capture both method calls and state changes as data, we have a concrete way to write **declarative BDD style tests**.

- **Given** zero or more past events
- **When** command
- **Then**
  - Zero or more events if the command is accepted
  - An exception if the command is rejected

Notice that we never **ask** about state. Instead, we rely on the produced events to **tell** us what it decided to incorporate into the system's history.

An example happy path test

EDUMENT

An example happy path test using commands and events

```
[Test]
public void CanMakeAppointment()
{
    Test(
        Given(new DateOpenedForAppointments
            {
                Id = testId, Date = testDate
            }),
        When(new MakeAppointment
            {
                Id = testId, Hour = 9, Minute = 0,
                DurationInMinutes = 60, Name = "Mrs Ahnida Hairkut"
            }),
        Then(new AppointmentMade
            {
                Id = testId, Hour = 9, Minute = 0,
                DurationInMinutes = 60, Name = "Mrs Ahnida Hairkut"
            }));
}
```

An example sad path test

EDUMENT

An example sad path test using commands and events

```
[Test]
public void CannotMakeOverlappingAppointments()
{
    Test(
        Given(new DateOpenedForAppointments
        {
            Id = testId, Date = testDate
        },
        new AppointmentMade
        {
            Id = testId, Hour = 9, Minute = 0,
            DurationInMinutes = 60, Name = "Mrs Ahnida Hairkut"
        }),
        When(new MakeAppointment
        {
            Id = testId, Hour = 9, Minute = 30,
            DurationInMinutes = 60, Name = "Mr Asham Pu"
        }),
        ThenFailWith<OverlappingAppointmentsNotAllowed>());
}
```

That's nice, now how do we get there?

To build a system this way, we need...

- Something that can **handle commands**, deciding if they are valid and producing either zero or more events if so, or an exception if not.
- Something that can take a sequence of events and build a **model of the past**

The main goal of the model of the past is to help **validate** the commands. For example, it needs to be able to identify when a new booking would conflict with an existing one.

Aggregates: models of the past

An aggregate needs to have a sequences of events applied to it, each event contributing to its **model of the past**.

We can easily enough define an interface for event application:

```
public interface IApplyEvent<TEvent>
{
    void ApplyEvent(TEvent e);
}
```

The "building a model of the past" bit is encapsulated inside the aggregate. It just implements the generic interface once per event type it can apply.

Now, let's turn to handling commands...

Command handlers

The responsibility of the aggregate is to absorb events into a model of the past. Aggregates have nothing to do with commands. Thus, a *cohesive* solution should place command handling into a different object. We'll call it a **command handler**.

Command handlers will also implement a generic interface, this time once per command type. They need to **obtain an aggregate to work with**, so they will be passed an **a**ggregate **l**oader as well as the command.

```
public interface IHandleCommand<TCommand, TAggregate>
{
    IEnumerable HandleCommand(Func<Guid, TAggregate> al, TCommand c);
}
```

What a test does (1)

Recall the structure of a test:

```
Test(
        Given(/* events */),
        When(/* command */),
        Then(/* events */));
```

This calls a **Test** method, which we know should send the command to the command handler, observe the fallout, and pass or fail the test accordingly.

We also see that the three part pass through some **`Given`**, **`When`** and **`Then`** methods. These probably transform the inputs somehow. But how?

What a test does (2)

Under the hood, `Given`, `When` and `Then` will produce, respectively:

- `given`, which is simply the original events
- `when`, a lambda which expects a pre-populated aggregate and returns the result of handling the command
- `Then`, a lambda which expects an actual result and checks it against the expected result

So the core of the test framework, the `Test` method, ends up looking like this:

```
then(when(ApplyEvents(new TAggregate(), given)));
```

Pretty, huh?

Implementing a command handler

Passing the happy-path test is **easy**: the test will pass as soon as an event is returned when a command is sent in.

```
public IEnumerable HandleCommand(Func<Guid, AppointmentAggregate> al,
                                 MakeAppointment c)
{
    yield return new AppointmentMade
    {
        Id = c.Id, Hour = c.Hour, Minute = c.Minute,
        DurationInMinutes = c.DurationInMinutes, Name = c.Name
    };
}
```

In fact, this is so easy that we're not even using the aggregate yet. We generally don't need the aggregate until we start validating against the past.

Adding a check to the command handler

In order to pass the sad-path test, we need to have an awareness of previous appointments, so that we can reject double bookings. Let's extend our `HandleCommand` with a check:

```
public IEnumerable HandleCommand(Func<Guid, AppointmentAggregate> al,
                                 MakeAppointment c)
{
    var appointments = al(c.Id);

    if (!appointments.canBook(c.Hour, c.Minute, c.DurationInMinutes))
        throw new OverlappingAppointmentsNotAllowed();

    yield return new AppointmentMade
    {
        Id = c.Id, Hour = c.Hour, Minute = c.Minute,
        DurationInMinutes = c.DurationInMinutes, Name = c.Name
    };
}
```

Adding functionality to the aggregate

Now we need the **canBook** utility method in the aggregate:

```
private List<Booking> bookings = new List<Booking>();

public bool canBook(int hour, int minute, int durationInMinutes)
{
    var startMinute = hour*60 + minute;
    var endMinute = startMinute + durationInMinutes;

    return !bookings.Any(
        booking => booking.overlaps(startMinute, endMinute));
}
```

Adding functionality to the aggregate

The **Booking** class stores projections of bookings, and checks if a booking overlaps with a new proposed booking time.

```
private class Booking
{
    public int Start;
    public int End;

    public bool overlaps(int start, int end)
    {
        return start >= Start && start < End
        || end >= Start && end < End;
    }
}
```

Adding event handlers to the aggregate

Finally, the aggregate needs to apply `AppointmentMade` events to build up its model of the past.

```
public void ApplyEvent(AppointmentMade e)
{
    bookings.Add(new Booking
        {
            Start = e.Hour*60 + e.Minute,
            End = e.Hour*60 + e.Minute + e.DurationInMinutes,
        });
}
```

How do we test that producing a `AppointmentMade` event actually works?

Every event type should appear in a given

Easy: we check what happens when we try to make a duplicate appointment.

- `Given AppointmentMade`
- `When MakeAppointment`
- `Then (exception) OverlappingAppointmentsNotAllowed`

So in order to test whether an event worked, we write another test where that event figures in the `given` part of the test. It turns out that this is a general principle of behavioral testing.

So, a nice quick check for sufficient test coverage is: does each of your event types occur in at least one `given` in a test?

Exercise 7

In this exercise, we will revisit an earlier exercise and re-implement some of the tests and business logic.

However, this time, we will be focusing on making our logic **referentially transparent** and write **BDD style tests**.