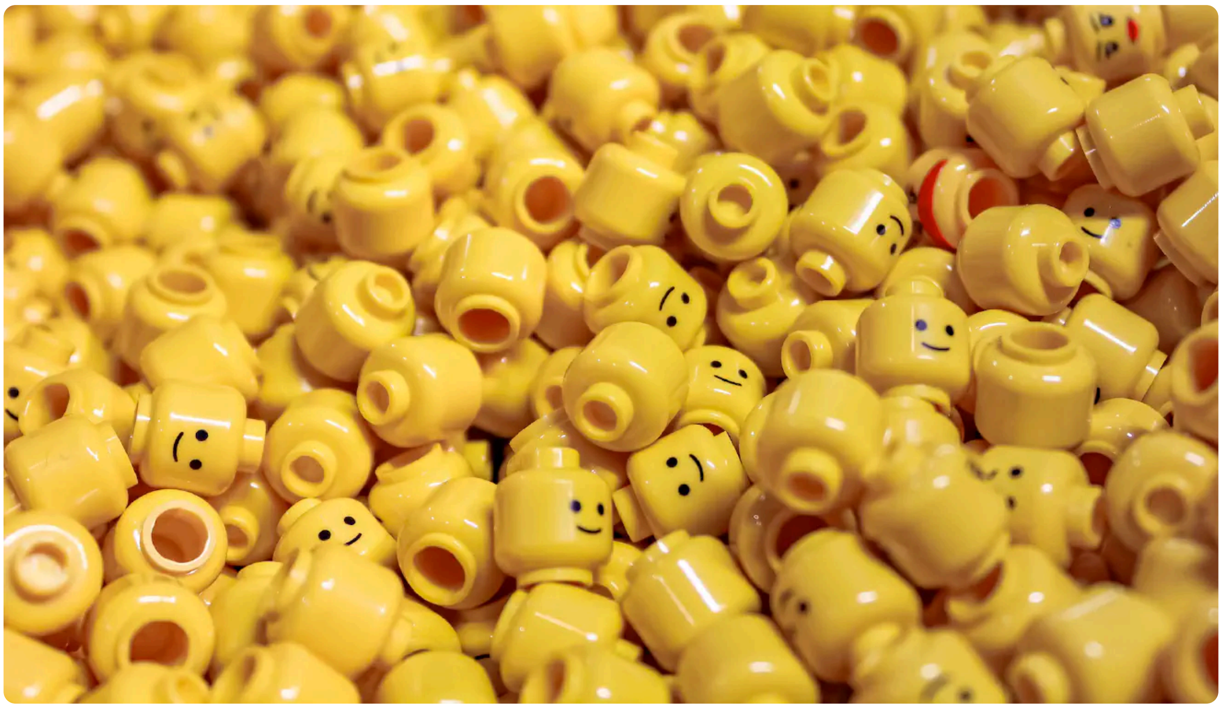
Mastering the Object Mother



Discover how the Object Mother concept empowers developers to effortlessly generate intricate test objects, enhancing code readability, maintainability, and overall testing efficiency.

The creational problem

A good structured test exists out of three parts commonly known as: Given When Then or Arrange Act Assert.

Within the *Given* part you declare a set of objects that will drive your test. Initially the creational logic is simple, as the objects may have fewer fields or lack coherence. However, inevitably over time, the set of objects grows as their intricacy, they become more complex and time-consuming to construct.

Various solutions exist to address those issues, such as *Test Data Factories*, *Data Fakers*, *Test Data Generators* or *Fixture Builders* etc .. In the end they all share a common goal: **simplifying** object creation while ensuring **reusability**. Martin Fowler has even coined a term for this concept, known as Object Mother.

Let's delve deeper into the problem by examining a concrete example. While this example may not be overly complex or entirely realistic, the purpose of this article is

to work with practical scenarios without getting bogged down in the intricacie
object creation.

```
Address billingAddress = new Address.Builder()
        .streetAddress("123 Main St")
        .city("Anytown")
        .state("CA")
        .zipCode("12345")
        .country(Country.US)
        .build();
Address shippingAddress = new Address.Builder()
        .streetAddress("456 Oak Ave")
        .city("Othertown")
        .state("CA")
        .zipCode("67890")
        .country(Country.US)
        .build();
List<InvoiceItem> items = new ArrayList<>();
items.add(new InvoiceItem("Product A", Amount.EUR(100.00), Tax.vatPerc
items.add(new InvoiceItem("Product B", Amount.EUR(100.00), Tax.vatPerc
Customer customer = new Customer.Builder()
        .name("John Doe")
        .email("john.doe@example.com")
        .phoneNumber("555-123-4567")
        .build();
Invoice invoice = new Invoice(
        new InvoiceNumber("001"),
        customer,
        billingAddressm,
        shippingAddress,
        LocalDate.now(),
        items);

Amount amount = invoice.getVatAmount();

assertThat(amount).isEqualTo(Amount.EUR(42))
```

In the case of the aforementioned test, an Invoice object is needed, which, in turn, depends on several other objects. However, it is noteworthy that, for this specific test, **only the invoice items hold significance**. They play a vital role in asserting and calculating the VAT amount, all the other objects and fields just **clutter the readability of the test**.

Test Data - Factory, Generator, Builder, Fixtures ...

There are numerous patterns available to create the required objects for your tests. A very common approach to simplify the creational logic while at the same time having some kind of of code reuse in place is to create static factory methods returning the required objects.

```
InvoiceTestDataFactory.invoice();
```

While this approach may initially appear to solve the creational issue, it tends to scale poorly As Martin Fowler highlights:

> *"Object Mothers do have their faults. In particular there's a heavy coupling in that many tests will depend on the exact data in the mothers."*

I've observed several solutions to address this issue, including;

- creating specific static factory methods
- adding parameters to differentiate the creational logic.

```
InvoiceTestDataFactory.invoiceWithoutShippingAddress();
InvoiceTestDataFactory.invoiceWithoutBillingAddressAndThreeItems();
InvoiceTestDataFactory.invoice(new InvoiceNumber("001"), "john@doe,com
InvoiceTestDataFactory.invoice(
        new InvoiceItem("Product A", Amount.EUR(100.00), Tax.vatPercen
        new InvoiceItem("Product B", Amount.EUR(100.00), Tax.vatPercen
```

While these solutions may initially appear to solve the coupling issue, they often result in tight coupling between specific factory methods and the requirements of individual tests, limiting their reusability.

One introduces *different* factory methods or several sets of parameters because there are tests that require *different permutations* of the same objects under test.

Imagine you have an object with 5 primitive fields and 5 custom-typed fields. The question arises: how many permutations of factory methods or parameter sets would be required to cover all your test cases?

Introducing static factory methods or parameters might make the code challenging to maintain and may not facilitate effective communication within the Given part of your tests. Ultimately, pursuing this path leads to test data factories that are difficult to maintain and confusing to use.

So, how can we **simplify the technical creational logic while simultaneously highlighting its essential aspects**?

Emphasize what matters and hide the irrelevant parts

By combining the Object Mother concept with **pre-filled builders** it becomes possible to hide away all unnecessary complexity while at the same time **emphasizing what matters** for your test-case.

Let's put this into practice using the previous example:

```
Invoice invoice = InvoiceMother.invoice()
        .withInvoiceItems(
            new InvoiceItem("Product A", Amount.EUR(100.00), Tax.vatPe
            new InvoiceItem("Product B", Amount.EUR(100.00), Tax.vatPe
        .build();

Amount amount = invoice.getVatAmount();

assertThat(amount).isEqualTo(Amount.EUR(42))
```

👉 It's important to **stress the prefilled nature of a mother.** Calling **`` `InvoiceMother.invoice().build()` ``** will return a fully fledged **`` `Invoice` ``** were all fields are filled in containing **sensible default values**.

By utilizing the approach mentioned above, the unnecessary clutter is eliminated, allowing the focus to be solely on what is essential for the test. In this particular case, it becomes evident that having two invoice items priced at EUR 100 each, with a 21% tax applied to each item, results in EUR 42 of taxes

We are still using some kind of *static factory method*, yet it does not immediately return our Invoice rather it returns a **builder** that allows you to override those defaults **that matter for your specific test**.

I like to make the builder part of the Mother class as to reduce the chance to confuse it with production code InvoiceBuilders.

```
public class InvoiceMother {

    private InvoiceMother() { }

    public static Builder invoice() {
        return new Builder();
    }

    public static class Builder {

        InvoiceNumber invoiceNumber = new InvoiceNumber("001");
        Customer customer = CustomerMother.customer().build();
        Address billingAddress = AddressMother.address().build();
        Address shippingAddress = AddressMother.address().build();
        LocalDate creationDate = LocalDate.now();
        List<InvoiceItem> items = List.of(InvoiceItemMother.item().bui

        public Builder withItems(List<InvoiceItem> items) {
            this.items = items;
            return this;
        }

        // other setters are left out for brevity

        public Invoice build() {
            return new Invoice(
                    invoiceNumber,
                    customer,
                    billingAddressm,
                    shippingAddress,
                    creationDate,
                    items);
        }
    }
}
```

It's important to note that the concept discussed here is not about the suffix 'Mother.' If you find the suffix unfavorable, feel free to use any other suffix that better suits your needs, such as *InvoiceTestDataFactory, InvoiceFixture, InvoiceTestData,* and so on.

What is important:

- Let your factory methods return *Builders* not the object under creation.
- Use a **pre-filled Builder**
- **Limit** your **static factory methods** to the absolute minimum.
- Rather **override a field using the Builder** than to introduce a new static factory method.

What can flex:

- Let the defaults for custom complex objects depend on other Mothers
- The Mother suffix
- Making the Builder part of your Mother class

But what if your test requires a specific field in one of the nested custom objects to be in a particular state? For instance, you may need the shipping address's country to be set as 'US,' while the other field values are irrelevant for your test. With the current setup, you would have to pass in another mother object and build it fully changing only that field that matter for your test.

```
InvoiceMother.invoice()
    .withShippingAddress(AddressMother.address()
        .withCountry(Country.US)
        .build())
    .build();
```

Although this approach works, we can further improve it by utilizing a **`java.util.function.Consumer`** with the **`AddressMother.Builder`**:

```
InvoiceMother.invoice()
        .withShippingAddress(b -> b.withCountry(Country.US))
        .build();
```

In this case, the builder method would look like this:

```
public Builder withShippingAddress(Consumer<AddressMother.Builder> add
    Address.Builder builder = AddressMother.address();
    addressConsumer.accept(builder)
    this.shippingAddress = builder.build();
    return this;
}
```

These enhancements aim to improve the readability and clarity of the code example while retaining the original meaning and intent.

Takeaways

In conclusion, the Object Mother concept offers developers a powerful approach to effortlessly generate intricate test objects. By combining the concept with pre-filled builders, developers can effectively simplify the technical creational logic while emphasizing the essential aspects of their test cases.

By embracing the Object Mother concept and its principles, developers can achieve enhanced code readability, maintainability, and overall testing efficiency in their software projects. This becomes particularly crucial in complex projects, as tests serve not only to verify code correctness and guide design but also to serve as documentation and prevention of regression. When regression does occur, it is vital for both your future self and colleagues to clearly understand what is being tested.