Unit test naming and test data construction

What are maintainable tests?

- readability
- readability
- readability
- readability
- readability

Test names

- What about the following test names?

```
@Test public void testBookSaved();

@Test public void testFindBook();

@Test public void testListBook();
```

What do they test?

Scenario based

- Which scenario is tested in the following method?

```
@Test public void testBookSaved();
```

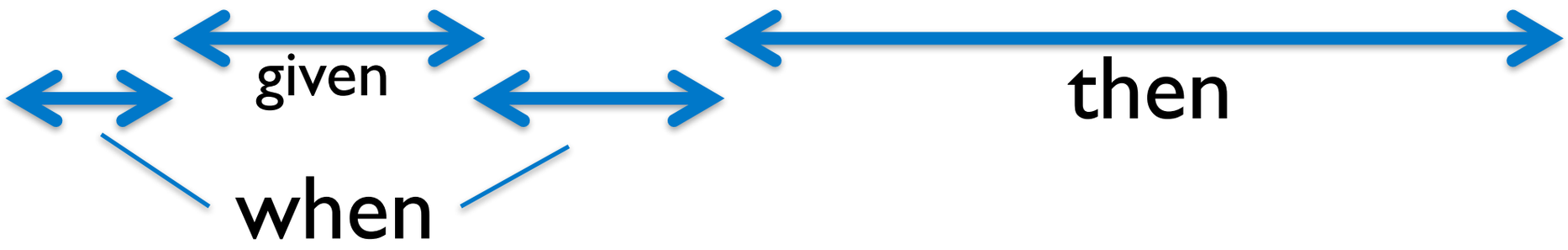
  - Happy scenario?
  - Unhappy scenario?

- If this test failed, which scenario is not working?

Scenario based

- What about:

```
@Test public void
whenValidBookIsSavedItsPersistedIntoDatabase();
```



- Describes a very clear scenario
- Easy to devise other test scenarios
  - whenInvalidBookIsSavedTheBookIsNotPersistedIntoDb
  - …

How to get good unit test names?

1. Don’t devise the initial name
2. Write the test
3. Get it to pass
4. Rename the test, based on the test scenario and Given-When-Then
5. Verify the test name with the test scenario
6. Reconsider the name with each revisit

Variable names

- What about understanding test:

```
public void
personWithInvalidAgeWillNotBeSavedIntoDb() {
   int invalidAge = 150;
   Person person = new Person();
   person.setAge(150);
   person.setAge(invalidAge);
   dao.savePerson(person);
   ...
}
```

Choose self-describing variables

Test data construction

- Well initialized test data is crucial to end up with maintainable unit tests!

```
public void negativePriceIsNotAccepted() {
  Product product = new Product();
  product.setPrice(-190);
  product.setDescription("test");
  product.setCategories("testcatagorie");

  dao.saveProduct(product);
```

Only the price is relevant. Why to set description / categories?

Valid business objects

- The reason to set the description and categories, is to create a valid business object
- But in this case, the knowledge about “what is a valid business object” is spread all over through the test-source.
- Results in a lot of overhead to create complex business objects.

Centralize the knowledge of valid

- What a Valid business object constitutes is defined with the Builder pattern!

```
public void negativePriceIsNotAccepted() {
  int invalidPrice = -190;
  Product product = ProductBuilder
                        .aProduct()
                        .withPrice(invalidPrice)
                        .build();
dao.saveProduct(product);
```

Builder pattern

- Construct always a valid business object

```
Product product = ProductBuilder
                      .aProduct()
                      .build();
```

- If specific values are necessary, specify it with a `with[property]` method

```
Product product = ProductBuilder
                      .aProduct()
                      .withPrice(...)
                      .build();
```

How to create a Builder?

```
public class ProductBuilder {
	private Product product;

	private ProductBuilder() {
		product = new Product();
		product.setPrice(100.0);
		product.setName("Testproduct");
		product.setCategory("TestProducts");
	}

	public ProductBuilder withPrice(double price) {
		product.setPrice(price);
		return this;
	}

	public ProductBuilder withName(String name) {
		product.setName(name);
		return this;
	}

	public Product build() {
		return product;
	}

	public static ProductBuilder newProduct() {
		return new ProductBuilder();
	}
}
```

Private constructor

static construct method

A more complex example

```
Order order = OrderBuilder.newOrder()
      .withOrderRegel(
         OrderRegelBuilder
                  .newOrderRegel()
                  .withProduct(ProductBuilder
                        .newProduct()
                        .withPrice(129.0)
                        .build())
                  .withAmount(5)
                  .build()
)
.withKlant(KlantBuilder.newKlant()
      .withNaam("Tester")
      .build()
)
.build();
```

Static methods

Use static imports

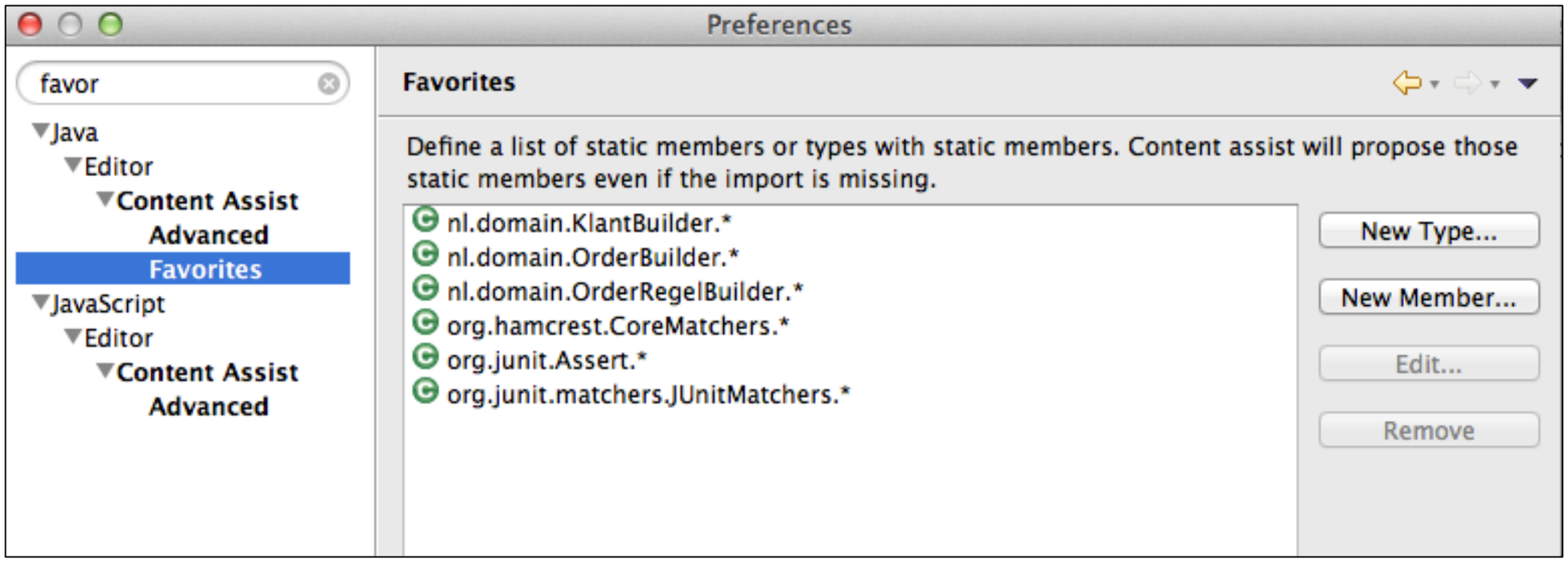
```
import static nl.packagename.domain.OrderRegelBuilder.*;
import static nl.packagename.domain.OrderBuilder.*;
import static nl.packagename.domain.KlantBuilder.*;

Order order = newOrder()
        .withOrderRegel(
                newOrderRegel()
                .withProduct(
                    newProduct()
                    .withPrice(129.0)
                    .build())
                .withAmount(5)
                .build()
    )
    .withKlant(newKlant()
        .withNaam("Tester")
        .build()
    )
    .build();
```

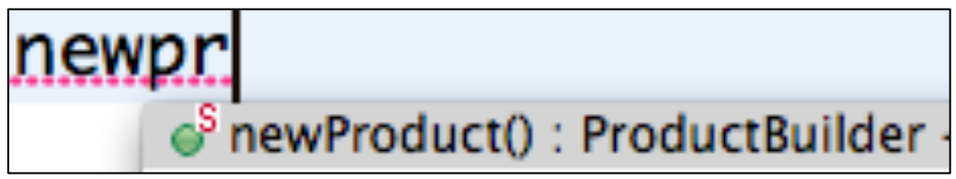
Static imports

Configure Eclipse

- Add e.a. builder objects to Favorites in Eclipse



- After that, you can use CTRL+SPACE in the editor



How to create reusable test content?

- Sometimes you want a fully preconfigured test content object.
- Important: the fully preconfigured content should be required for the unit test
- Builders are not meant for this usecase
- Use Factories instead

Test data Factory example

```
public final class OrderFactory {

  private OrderFactory() {}

  public static Order orderMetEenOrderRegelEnEenKlant (
       String klantNaam, double price, int productAmount){


       Order order = newOrder()
         .withOrderRegel(
           newOrderRegel()
             .withProduct(
               newProduct()
                .withPrice(price)
                .build())
             .withAmount(productAmount)
             .build()
         )
         .withKlant(newKlant()
           .withNaam(klantNaam)
           .build()
         )
         .build();
         return order;
    }
}
```

Don’t use this method if the Klant or OrderRegel is not required for the test.