Test Fixtures

Unit testing with data fixtures is good habit to get into, and having a simple means of creating and managing reusable fixture data makes it much more likely. The `FixtureBuilder` and `Fixture` class can simplify the creation of reusable test fixtures using a small DSL.

The text fixtures created using the `FixtureBuilder` are the properties required to build new instances of objects needed for testing.

The reasoning behind using Maps is that Groovy allows them to be used as constructor arguments for creating objects; therefore, the maps give you a reusable and detached data set for use in creating your test fixture instances. Two objects instances created from the same fixture data will be equivalent at the level of the properties defined by the fixture; however, each can be manipulated without effecting the other.

One thing to note about the fixtures is that the fixture container and the maps that are passed in as individual fixture data are all made immutable via the asImmutable() method; however, if the data inside the fixture is mutable, it still may have the potential for being changed. Be aware of this and take proper precautions when you create an interact with such data types.

Fixture DSL

The `FixtureBuilder` has three operations available to its DSL:

`define`
: The DSL entry point method, which accpets the DSL closure and creates the `Fixture` container.

`fix(Object,Map)`
: Uses the specified Map data as the content for the fixture with the provided key.

`fix(Object,PropertyRandomizer)`
: Uses the specified `PropertyRandomizer` to generate random content for the fixture with the provided key.

The `build()` method is then used to create the populated `Fixture` container object.

Usage

A fixture for a `Person` class, such as:

```
class Person {
    Name name
    LocalDate birthDate
    int score
}
```

could have fixtures created using the following code:

```
class PersonFixtures {

    static final String BOB = 'Bob'
    static final String LARRY = 'Larry'

    static final Fixture FIXTURES = define {
        fix BOB, [ name:new Name('Bob','Q','Public'), birthDate:LocalDate.of(1952,5,14),
score:120 ]
        fix LARRY, [ name:new Name('Larry','G','Larson'), birthDate:LocalDate.of(1970,2,
8), score:100 ]
    }
}
```

I tend to create a main class to contain my fixtures and to also provide the set of supported fixture keys. Notice that the `define` method is where you create the data contained by the fixtures, each mapped with an object key. The key can be any object which may be used as a `Map` key (proper equals and hashCode implementation).

Once your fixtures are defined, you can use them in various ways. You can request the immutable data map for a fixture:

```
Map data = PersonFixtures.FIXTURES.map(PersonFixtures.BOB)
```

You can create an instance of the target object using the data mapped to a specified fixture:

```
Person person = PersonFixtures.FIXTURES.object(Person, PersonFixtures.LARRY)
```

Or, you can request the data or an instance for a fixture while applying additional (or overridden) properties to the fixture data:

```
Map data = PersonFixtures.FIXTURES.map(PersonFixtures.BOB, score:53)
Person person = PersonFixtures.FIXTURES.object(Person, PersonFixtures.LARRY, score:200)
```

You can easily retrieve field property values for each fixture for use in your tests:

```
assert 100 == PersonFixtures.FIXTURES.field('score', PersonFixtures.LARRY)
```

This allows field-by-field comparisons for testing and the ability to use the field values as parameters as needed.

Lastly, you can verify that an object instance contains the expected data that is associated with a fixture:

```
assert PersonFixtures.FIXTURES.verify(person, PersonFixtures.LARRY)
```

which will compare the given object to the specified fixture and return true of all of the properties defined in the fixture match the same properties of the given object. There is also a second version of the method which allows property customizations before comparison.

One step farther… you can combine fixtures with property randomization to make fixture creation even simpler for those cases where you don't care about what the properties are, just that you can get at them reliably.

```
static final Fixture FIXTURES = define {
    fix FIX_A, [ name:randomize(Name).one(), birthDate:LocalDate.of(1952,5,14), score:120
]
    fix FIX_B, randomize(Person){
        typeRandomizers(
            (Name): randomize(Name),
            (LocalDate): { LocalDate.now() }
        )
    }
}
```

The fixture mapper accepts `PropertyRandomizer` instances and will use them to generate the random content once, when the fixture is created and then it will be available unchanged during the testing.