

Emerging Patterns for Testing Model Management Tools

Louis M. Rose, Dimitrios S. Kolovos, Richard F.
Paige, Tara Gilliam and Fiona A.C. Polack

Context & Motivation

Model Management

- Managing MDE artefacts.
- Operations implemented in:
 - ATL
 - OpenArchitectureWare
 - Epsilon
 - ...

Testing ϵ psilon

- Operations consume / produce models.
- Tests need to:
 - Construct input models.
 - Compare results against output models.
- Readable, maintainable tests.

Epsilon tests

- Test suite requirements:
 - Provide documentation.
 - Malleable (not resistant to change).
 - Independent (but not repetitive).

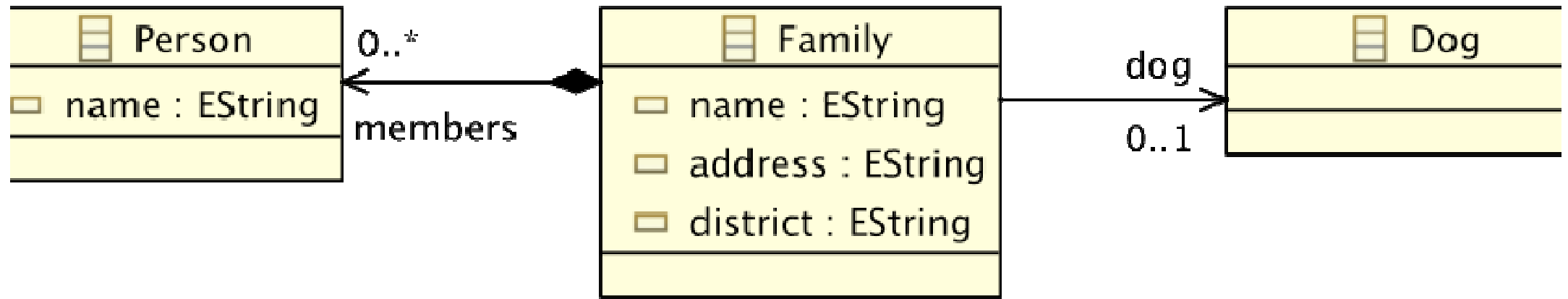
Emerging Patterns

Patterns

- Vocabulary
- Experience and expertise

Terminology

- Unit testing:
 - A test comprise many test cases.
 - Test case = JUnit4 test method.
 - A test case makes assertions on SUT.
 - Test fixture provides context per test.



Exemplar Metamodel

Constructing Models

Common solution - 1

- Use XML / XMI:

```
<?xml version="1.0" encoding="ASCII"?>  
<families:Family xmlns:families="families" name="The Smiths">  
  <members name="Jill"/>  
  <members name="Jack"/>  
</families:Family>
```

- Hard to read.

Common solution - 2

- Specify a domain-specific concrete syntax:

The Smiths [Jill, Jack]

- Requires maintenance of parser.
- Metamodel-specific.

HUIN

- Metamodel-independent concrete syntax.
- Human readable.

```
Family {  
  name: "The Smiths"  
  members: Person { name: "Jill" },  
           Person { name: "Jack" }  
}
```

- Can be needlessly verbose.

Factory classes

- ```
class FamiliesFixtureFactory {
 Family createFamily(Person... members) {
 Family f = FamiliesFactory.eINSTANCE.createFamily();
 f.getMembers().addAll(Arrays.asList(members));
 return f;
 }

 Person createPerson(String name) { ... }
}
```

```
// In the test
createFamily(createPerson("Jill"), createPerson("Jack"));
```

# Comparing Models

# Common solutions

- Use the modelling framework.
  - Unique identifiers?
  - Irrelevant data?

# Custom comparison

- Specify a metamodel-specific comparison:

```
rule Families
```

```
 match f : Expected!Family
```

```
 with g : Actual!Family {
```

```
 compare: f.name = g.name and
```

```
 f.members.matches(g.members)
```

```
}
```

```
rule Person
```

```
 match p : Expected!Person
```

```
 with q : Actual!Person {
```

```
 compare: p.name = q.name
```

```
}
```

# Model Assertions

- Test properties of the model

```
class FamilyTest {
 ModelWithEolAssertions model;
```

```
 @BeforeClass
```

```
 public static void setup() {
```

```
 // exercise unit under test
```

```
 Object outputModel = unitUnderTest.exercise();
```

```
 model = new ModelWithEolAssertions(outputModel);
```

```
 model.setVariable("f", "Family.all.first()");
```

```
 }
```

- @Test

```
 public void familyShouldContainMemberCalledJohn() {
```

- model.assertTrue("f.members.exists(p | p.name = 'John')");

- }

```
 }
```

# Summary

- Engineering malleable software.
- Patterns provide expertise and vocabulary.
- Identifying emerging patterns for tests that construct and compare EMF models.

Extra slides

# Software Evolution

- Up to 90% of development budget.
- Resistance to change (brittleness).

# Malleable software

- Better support for:
  - Evolutionary design
  - Refactoring
  - Regenerating bugs