Assignment 1 - Meta-modelling and instance generation with Refinery

Stein Vandenbroeke - Ariana Fernández

October 7, 2025

1 Introduction

In this assignment, we create a meta-model to describe a character in a role-playing game. This meta-model then allows us to build character models, which can be easily checked to see whether the model is valid and stratifies the requirements.
The meta-model is created using Refinery, a tool that surprisingly often fails (even on localhost), but allows for easy meta-model creation.

Figure 1: The notification I had to say way too often

2 Initial Design

2.1 Classes and meta-model structure

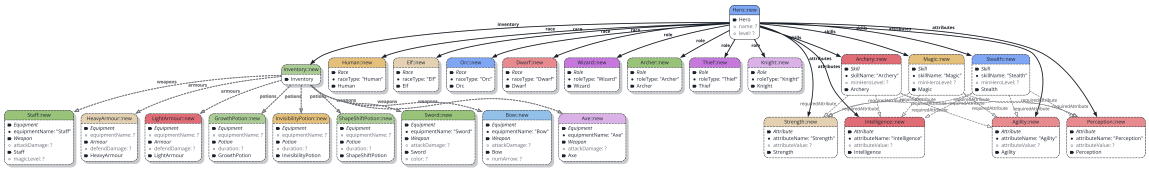

Figure 2: Our meta model

```
class Hero {
    string name;
    int level;
    contains Race[1] race;
    contains Role[1] role;
    contains Skill[1..2] skills; %Every hero has Skills (At least one - 1..*)
    contains Inventory [1] inventory; %A hero has an inventory. (At least one - 1..*)
    contains Attribute[1..2] attributes; %Can have Attributes
}
scope Hero = 1.
```

The code snippet above contains our implementation of the Hero class. To limit the number of items it can contain (for example, the number of skills), we use square brackets. Because we want to model only one hero at a time, we utilise ‘scope‘, which allows the model to have that item only once. The ‘contains‘ keyword is used to require an item to be present in the model and start from there. For example, the hero must have the attribute a skill requires. This is done using ‘contains‘ because every attribute present in the model must also be present in the hero.

2.2 Values

Every class reference is automatically created and filled in when specified in the meta-model. The values of primitive types are not automatically generated. Setting values can be done as follows:

```
decision rule setLevel100(Hero h) ==> level(h): 100.
decision rule setLevel50(Hero h) ==> level(h): 50.
concretization rule setLevel(Hero h) <-> !isConcrete(level(h))
    ==> level(h): 70.
```

This will set the hero level value to 100, 70 or 50. This is useful to later check whether certain values are met.

3 Meta-Model Refinement

The constraints in our meta-model are set using the 'error'; when generating a model, it will never create a model that would generate an error. For some constraints, we wrote some tests to easily check whether they are correct; these tests are annotated with 'should pass' and 'should fail'. In the following section, we will discuss some interesting constraints:

3.1 Interesting Constraints

3.1.1 At least one item needs a not zero value

```
% Rule: At least one of the attributes must be a value higher than one

% Set concrete values for attributes, to test only set values to zero
% -> should fail to create a model
decision rule setAttributeValue0(Attribute a) ==> attributeValue(a):0.
decision rule setAttributeValue10(Attribute a) ==> attributeValue(a):10.
decision rule setAttributeValue20(Attribute a) ==> attributeValue(a):20.
decision rule setAttributeValue30(Attribute a) ==> attributeValue(a):30.
decision rule setAttributeValue40(Attribute a) ==> attributeValue(a):40.
concretization rule setAttributeValue(Attribute a) <-> !isConcrete(attributeValue(a))
    ==> attributeValue(a): 50.

int sumOfAllAttributes() = sum {  Attribute(a) -> attributeValue(a)}.
error noZeroAttributes() <-> sumOfAllAttributes() == 0.
```

This rule checks whether there is at least one attribute with a value higher than one. This is done by calculation that sums all attributes and adds an error statement if they are all zero. This works because the Hero is the only one who can contain the attributes.

3.1.2 Inventory limit

The inventory class is split up into Weapon, Potion and Armour, but we still want to limit the total amount of items in the inventory:

```
error EquipmentLimitHero(Hero h) <->
    inventory(h, i),
    (count {  weapons(i, _) }
     + count {  potions(i, _) }
     + count {  armours(i, _) })  > 5. %equipment limit Hero (error if inventory > 5 items)
```

3.1.3 No duplicate items in skills or attributes

```
error noDubbleAtributesInHero(Hero h) <-> attributes(h, a1), attributes(h, a2),
                                          a1 != a2, attributeName(a1) == attributeName(a2).
```

```
error noDubbleSkillsInHero(Hero h) <-> skills(h, s1), skills(h, s2),
                                        s1 != s2, skillName(s1) == skillName(s2).
```

To prevent duplicate items in the attributes, we create two attributes when they are not the same; the attributeName should be different. For skills, we used the same approach.

3.1.4 Archer must have a bow

```
    pred hasBow(Hero h) <->
    inventory(h, i),
    weapons(i, b),
    Bow(b).

error archerMustHaveBow(Hero h) <->
    role(h, r), Archer(r), !hasBow(h).
```

Here we use a pred that states there should be a bow in inventory. And then in the error, we create an archer, and not our pred.

3.2 Rule tests

To verify that we created our rule correctly, we added some tests (which are commented out). For example, for the rule 'Archer must have a bow', we created a positive test (should pass) where there should be at least one archer and one bow, and a negative test (should fail) where we set the scope of archer to one and the scope of bow to zero.

```
% TEST 1 (should succeed): Generate archer with bow
% scope Archer = 1.
% scope Bow = 1.

% TEST 2 (should fail): generate archer without bow
% scope Archer = 1.
% scope Bow = 0.
```

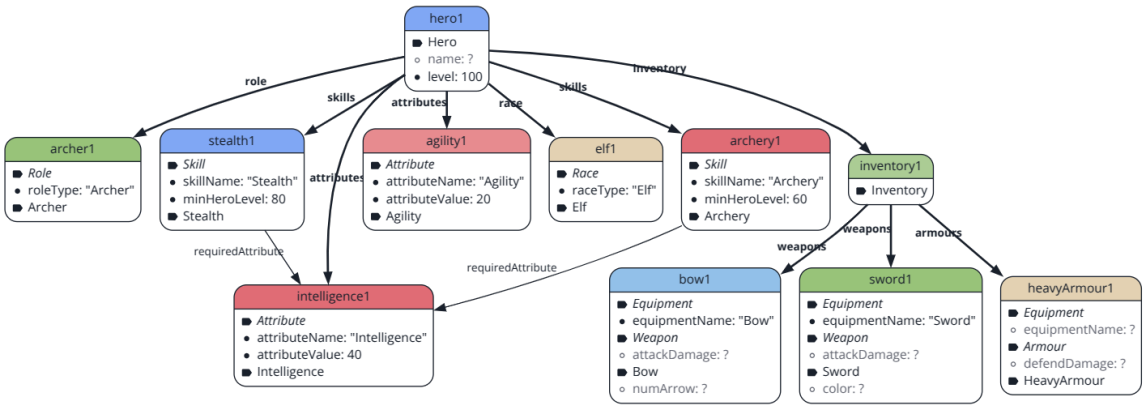

Figure 3: A generated model for TEST 1, bow and archer is present

Figure 4: Failed to create a model for TEST 2, because it is not possible to have an archer without a bow

3.3 Thoughts and decisions reflected in our model

When we first completed the section "Classes and Attributes" of our assignment, we designed some elements that, after analyzing the constraints and rereading more literally, did not fully make sense or lacked important details such as correct cardinalities or other structural aspects.

To mention one, we first used a Character superclass to share equipment, but letting Race extend it wrongly gave races an inventory. So we replaced Character with an Inventory: now a Hero has one Inventory with Weapons, Armour, or Potions. This approach matches the idea that "A Hero has an inventory, which can contain Equipment," keeps Race and Role as independent labels, and makes the constraints easier to understand and handle.

Also, we first tried to modeled each Skill with its own attributes, but this duplicated information. We realized that Attributes belong to hero, not to the Skill, so we linked Skills to Hero's attributes instead.

At the end, these decisions helped us better align our structure and constraints.

4 Example model

Now that we have created our meta-model with all constraints, we can generate models. Here is a randomly generated model:

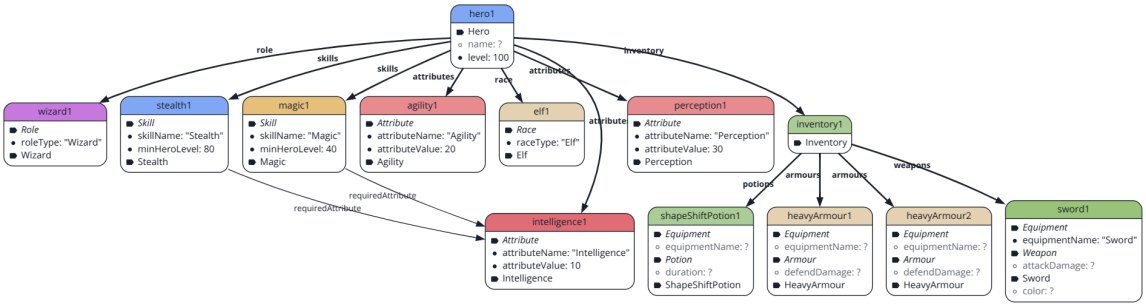

Figure 5: Model

Here is a model generated for an elf-wizard. Magic is present in the skills of the hero because this is a rule. The attribute intelligence is present in hero and magic because this is required for having magic, and if a skill has an attribute, the hero must have this.
These are just some randomly picked examples, and there are, of course, way more rules that the model is following.