FIVE relation engine

October 6, 2015

1 Purpose

The relation engine determines which objects of the world can be used and how they can be used. These questions can be asked to the engine considering its knowledge of the world.

> I have a threaded rod what can I do with it?

2 Data model

Here is a description of the data handled by the engine.

An **object** is an entity that can potentially interact with other objects.

> Let *Threaded rod* be an object with two threaded extremities.
> Let *Nut* be an object.

An object is defined from its **types**. Each type is linked to an identifier, the later is unique in the context of the object.

> *Threaded rod* has two types: *Male* and *Male*. The first type is identified with *Extremity A* and the second type is identified with *Extremity B*.
> *Nut* has three types: *Female*, *Female* and *Screwable*. The first type is identified with *Side A*, the second type is identified with *Side B* and the third type is identified by *Motion*.

An **object pattern** is an abstract object or an object interface. It is defined by a set of types mapped with their unique identifiers w.r.t. the object pattern. If an object has the same types as an object pattern, thus it matches this object pattern.

> The object pattern *Male thread* is defined by the type *Male* identified with *Thread*.
> The object pattern *Screwable female thread* is defined by two types: *Female* identified with *Thread* and *Screwable* identified by *Move*.

An **object match** matches an object and its types with an object pattern. It makes correspondences between the identifiers of the object and the identifiers of the object pattern.

> *Threaded rod* matches *Male thread* where *Extremity A* corresponds to *Thread*.
> *Threaded rod* matches *Male thread* where *Extremity B* corresponds to *Thread*.
> *Nut* matches *Screwable female thread* where *Side A* corresponds to *Thread* and *Motion* corresponds to *Move*.
> *Nut* matches *Screwable female thread* where *Side B* corresponds to *Thread* and *Motion* corresponds to *Move*.

A **relation** connects objects based on a set of object patterns. Moreover it gives a meaning to their union.

> *Screw* connects the objects that match *Male thread* and *Screwable female thread*.

A **realization** is the instantiation of a relation where concrete objects (v.s. abstract object patterns) are connected together.

> *Threaded rod* with type *Extremity A* and *Nut* with types *Side A* and *Motion* realize *Screw*.
> *Threaded rod* with type *Extremity A* and *Nut* with types *Side B* and *Motion* realize *Screw*.
> *Threaded rod* with type *Extremity B* and *Nut* with types *Side A* and *Motion* realize *Screw*.
> *Threaded rod* with type *Extremity B* and *Nut* with types *Side B* and *Motion* realize *Screw*.

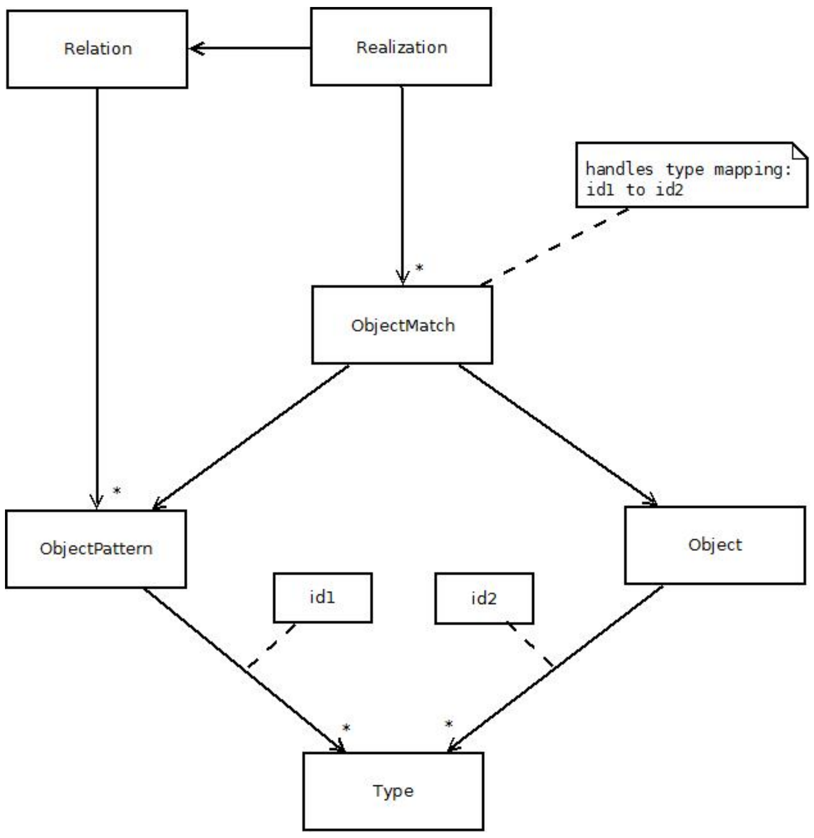

Figure 1: The data model

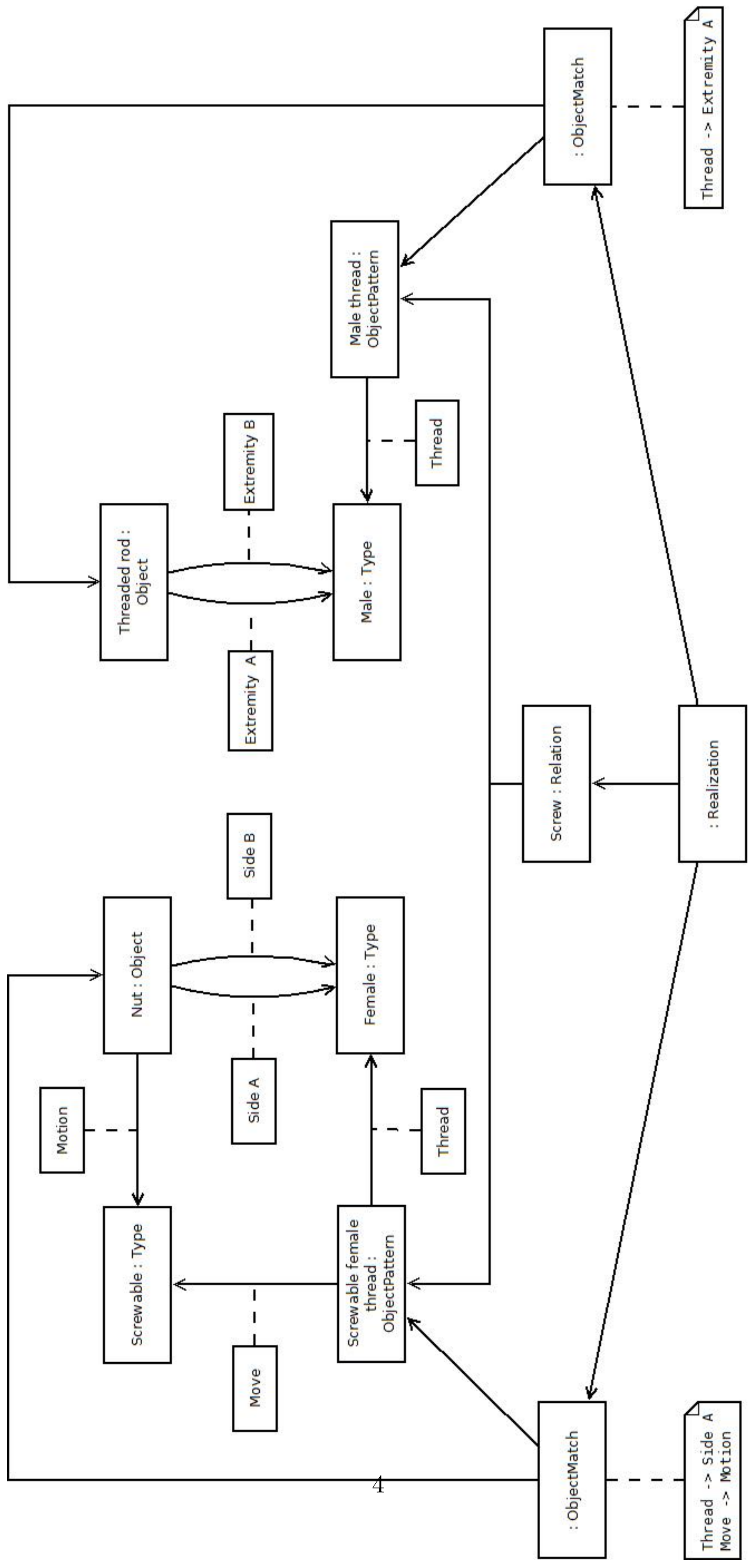

Figure 2: A possible instantiation of the data model

3 Questions

Note: Samples in this section are described using pseudo-code.

3.1 What are the realizations where objects are involved?

Listing 1: What are the realizations where *Threaded rod* and *Nut* are involved?

```
> GetRealizations(['Threaded rod', 'Nut'])
[
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity A'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side A',
     'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity A'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side B',
     'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity B'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side A',
     'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity B'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side B',
     'Move'='Motion'})])
]
```

3.2 What are the realizations of a relation?

Listing 2: What are the realizations of *Screw*?

```
> GetRealizations('Screw')
[
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity A'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side A',
     'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity A'}), ('Screwable female
     thread', 'Nut', {'Thread'='Side B',
     'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
     {'Thread'='Extremity B'}), ('Screwable female
```

```
      thread', 'Nut', {'Thread'='Side A',
      'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
      {'Thread'='Extremity B'}), ('Screwable female
      thread', 'Nut', {'Thread'='Side B',
      'Move'='Motion'})])
]
```

3.3 What are the realizations of a relation where objects are involved?

Listing 3: What are the realizations of *Screw* where *Nut* is involved?

```
> GetRealizations('Screw', ['Nut'])
[
  ('Screw', [('Male thread', 'Threaded rod',
      {'Thread'='Extremity A'}), ('Screwable female
      thread', 'Nut', {'Thread'='Side A',
      'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
      {'Thread'='Extremity A'}), ('Screwable female
      thread', 'Nut', {'Thread'='Side B',
      'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
      {'Thread'='Extremity B'}), ('Screwable female
      thread', 'Nut', {'Thread'='Side A',
      'Move'='Motion'})]),
  ('Screw', [('Male thread', 'Threaded rod',
      {'Thread'='Extremity B'}), ('Screwable female
      thread', 'Nut', {'Thread'='Side B',
      'Move'='Motion'})])
]
```

4 Programming interface

Implementations of the relation engine inherits from `FIVE.Relations.IEngine`. The most useful methods of this interface are listed bellow.

First of all, objects and relations need to be registered with the engine.

- `public void AddObject(Object object)`
  register an object with the engine.
- `public void AddRelation(Relation relation)`
  registers a relation with the engine.

Unregistering is also possible.

- `public void RemoveObject(Object object)`
  unregisters an object with the engine.

- `public void RemoveRelation(Relation relation)`
  unregisters a relation with the engine.

It is also possible to query the engine about the registered objects and relations

- `public IEnumerable<Object> GetObjects()`
  Returns all the objects registered in the engine.

- `public IEnumerable<Relation> GetRelations()`
  Returns all the relations registered in the engine.

- `public Object GetObject(string name)`
  Returns the registered object with the given name (null if not found).

- `public Relation GetRelation(string name)`
  Returns the registered relation with the given name (null if not found).

- `public bool HasObject(string name)`
  Checks if an object with the given name has been registered.

- `public bool HasRelation(string name)`
  Checks if a relation with the given name has been registered.

Here are the methods that correspond to the questions listed above.

1.
   ```
   public IEnumerable<Realization>
       GetRealizations(IEnumerable<Object> objects)
   ```
   returns the realizations that involve the specified objects.

2.
   ```
   public IEnumerable<Realization>
       GetRealizations(Relation relation)
   ```
   returns every possible realization of a relation involving any registered object.

3.
   ```
   public IEnumerable<Realization>
       GetRealizations(Relation relation,
       IEnumerable<Object> objects)
   ```
   returns the realizations of a relation that involve at least all the specified objects (can be completed with other registered objects).

5 Change log

- `1.0.0`: First version with five queries
- `2.0.0`: Addition of the factory
- `2.0.1`: Replacement of `log4net.dll` with `UnityLog4Net.dll`
- `2.1.0`: Addition of the unregistering of objects and relations
- `2.1.1`: Fix of the unregistering of relations
- `3.0.0`: Removal of the factory (avoiding instantiation using reflection)
- `4.0.0`: Reduction of the A.P.I. to three essential queries, the engine was rewritten using Yield Prolog
- `4.0.1`: Removal of dynamic code generation to ensure Mono AOT compliancy
- `4.1.0`: Adding new procedural implementation of Relation engine in C# with type inheritance handling
- `4.2.0`: Objects, relations and object patterns are identified by their name. Types are identified by their system type.
- `4.3.0`: Update documentation to #FIVE 4.3 that added queries about registered objects and relations