EDOM - Engenharia de Domínio
Mestrado em Engenharia Informática
Lecture 06.2
ATL Demonstration

Alexandre Bragança atb@isep.ipp.pt

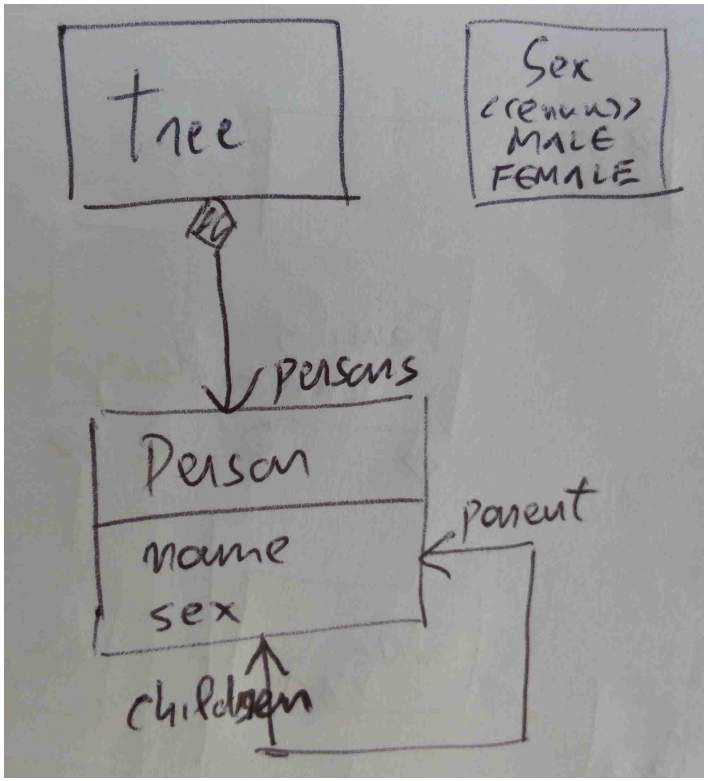
Dep. de Engenharia Informática – ISEP

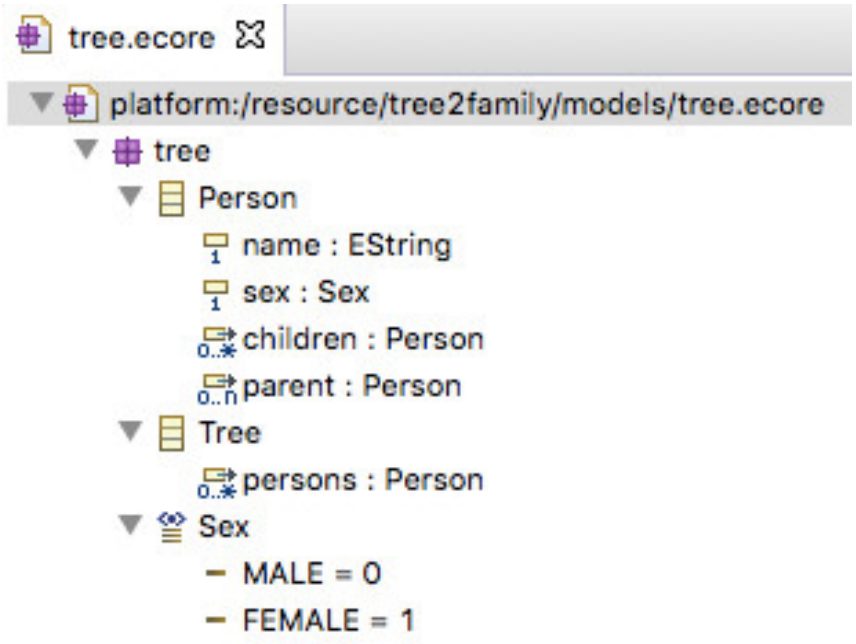
2017/2018

Goal

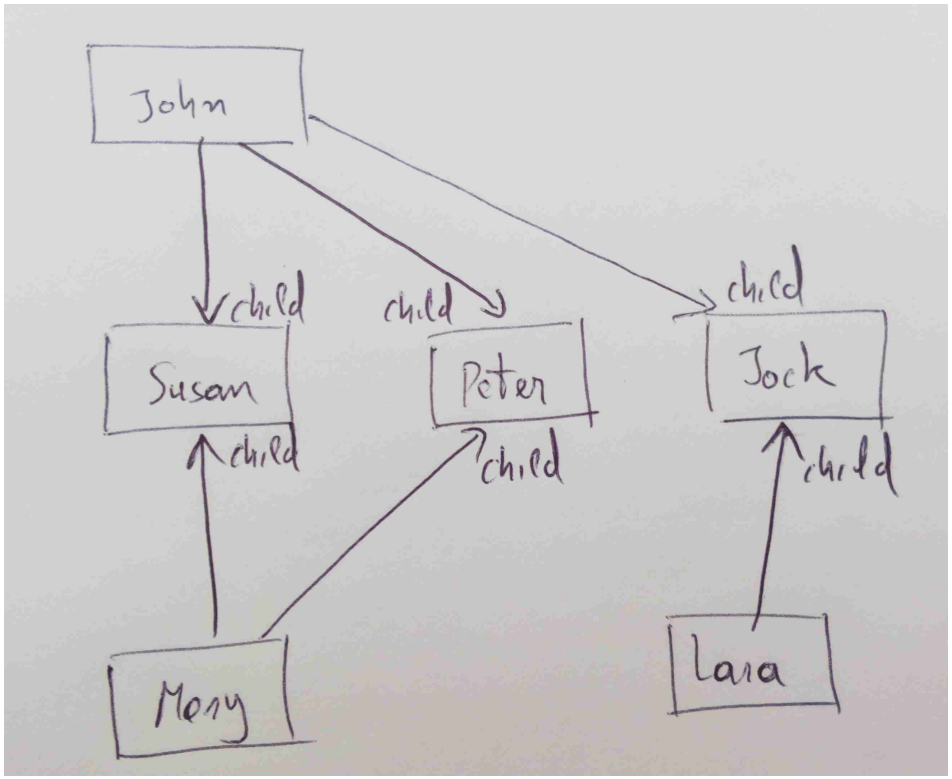
ATL Demonstration: Genealogy Tree to Family

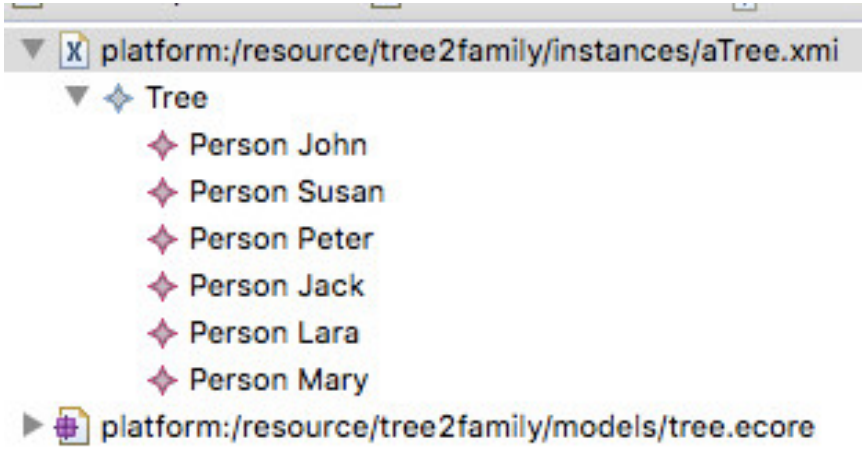
Genealogy Tree Metamodel





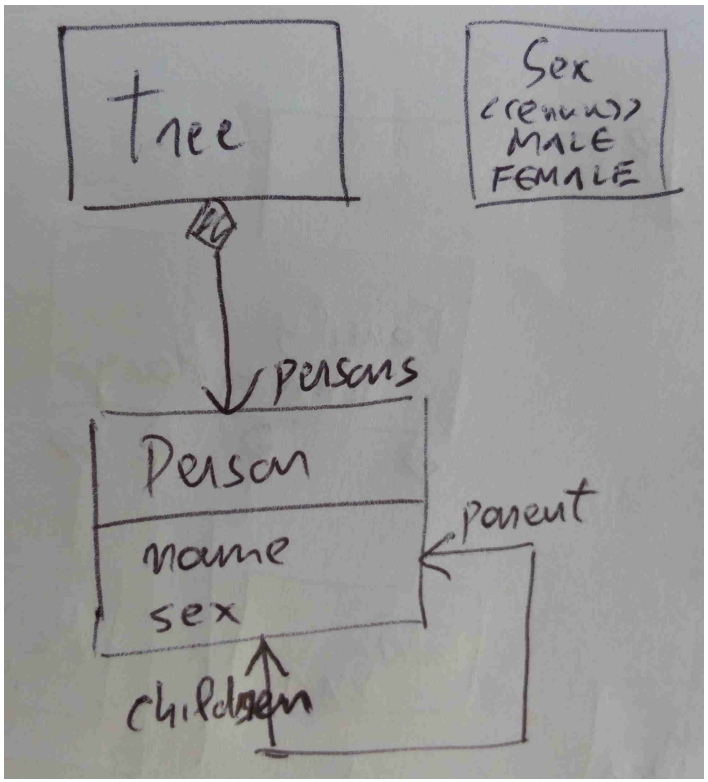
A Genealogy Tree Model (instance)





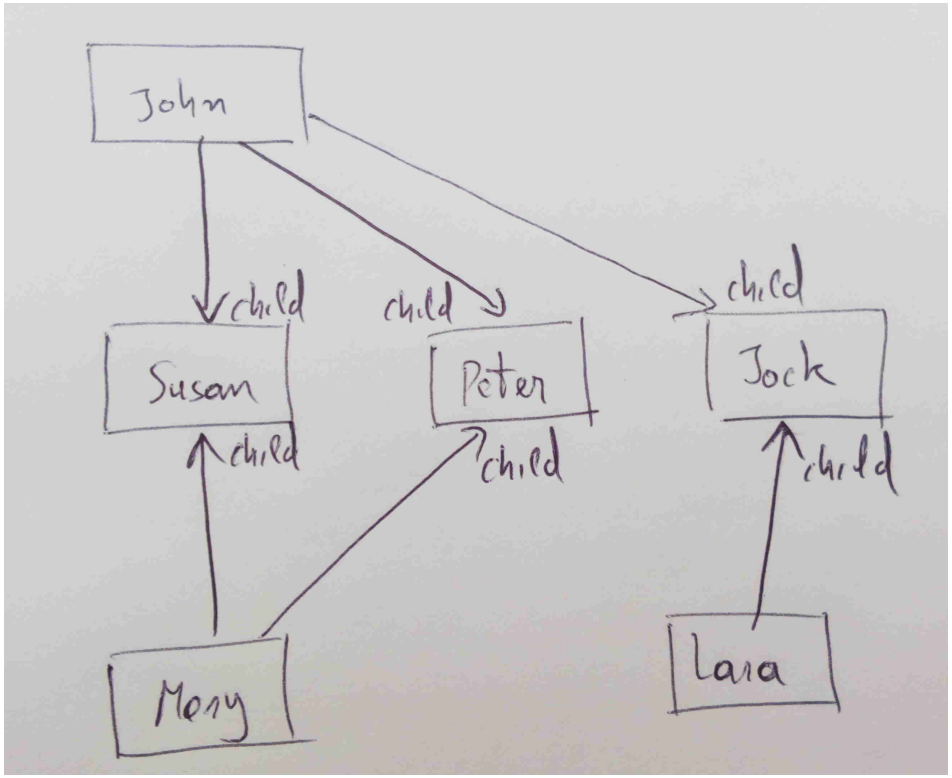
Genealogy Model and Metamodel

The Metamodel

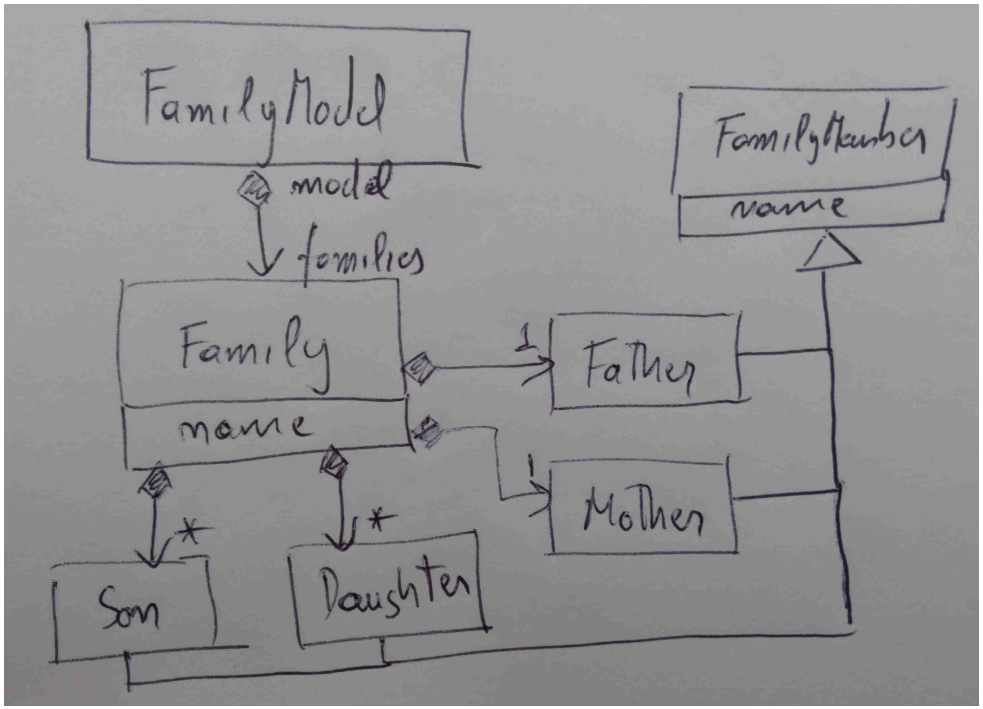


A Model...

Note: The instance of the `Tree` concept is not illustrated



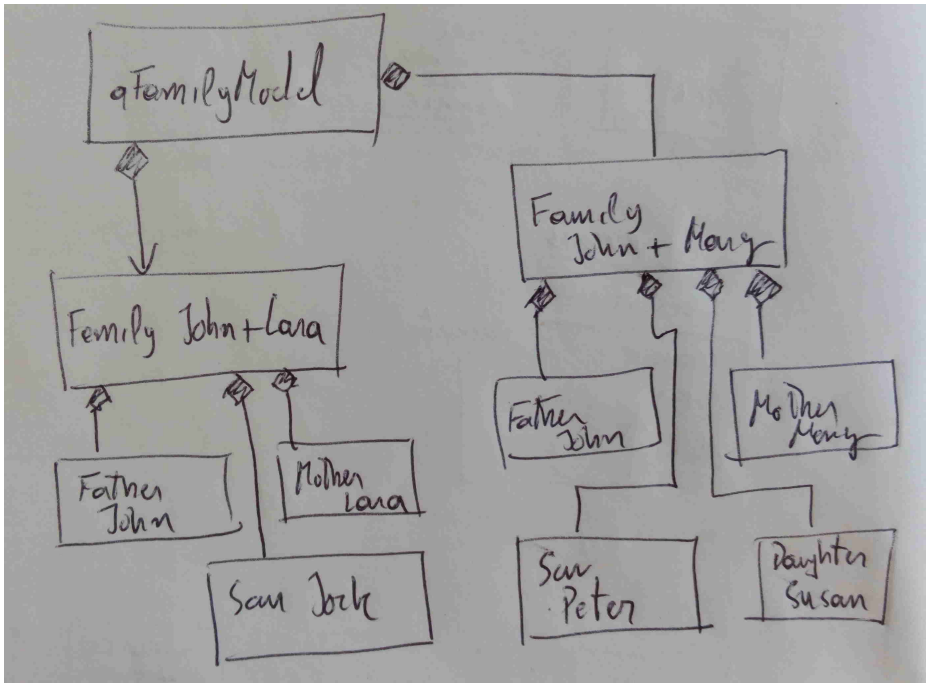
Family Metamodel

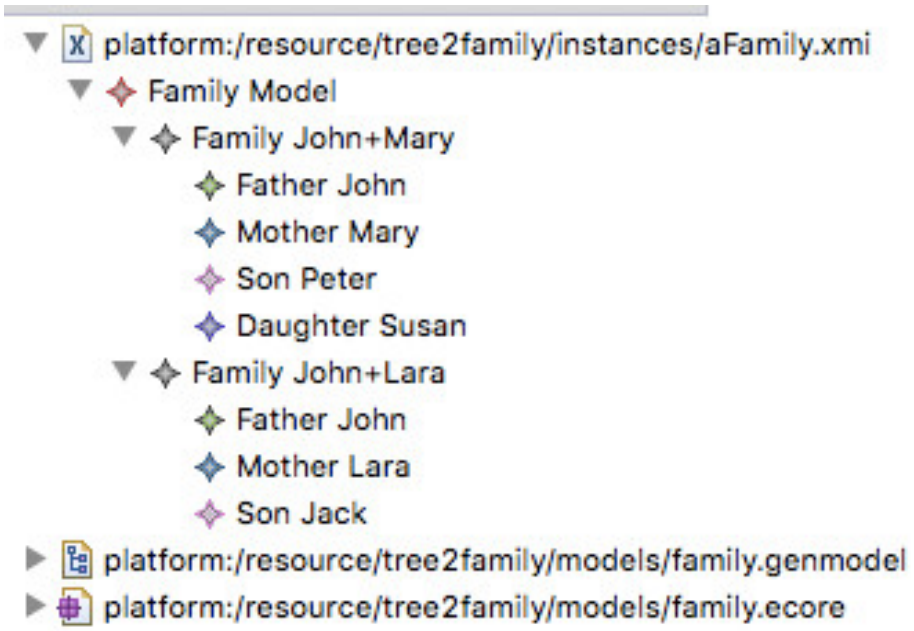


Outline family.ecore

- platform:/resource/tree2family/models/family.ecore
  - family
    - FamilyModel
      - families : Family
    - Family
      - father : Father
      - mother : Mother
      - sons : Son
      - daughters : Daughter
      - name : EString
      - model : FamilyModel
    - Father -> FamilyMember
    - Mother -> FamilyMember
    - Son -> FamilyMember
    - Daughter -> FamilyMember
    - FamilyMember
      - name : EString

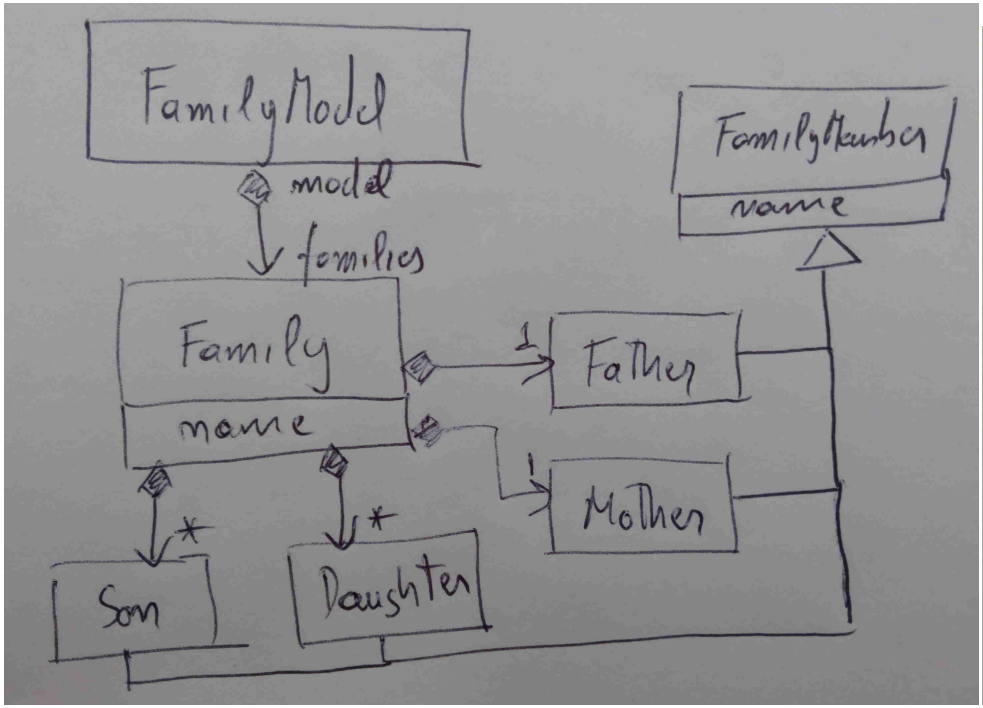
Family Model (instance)



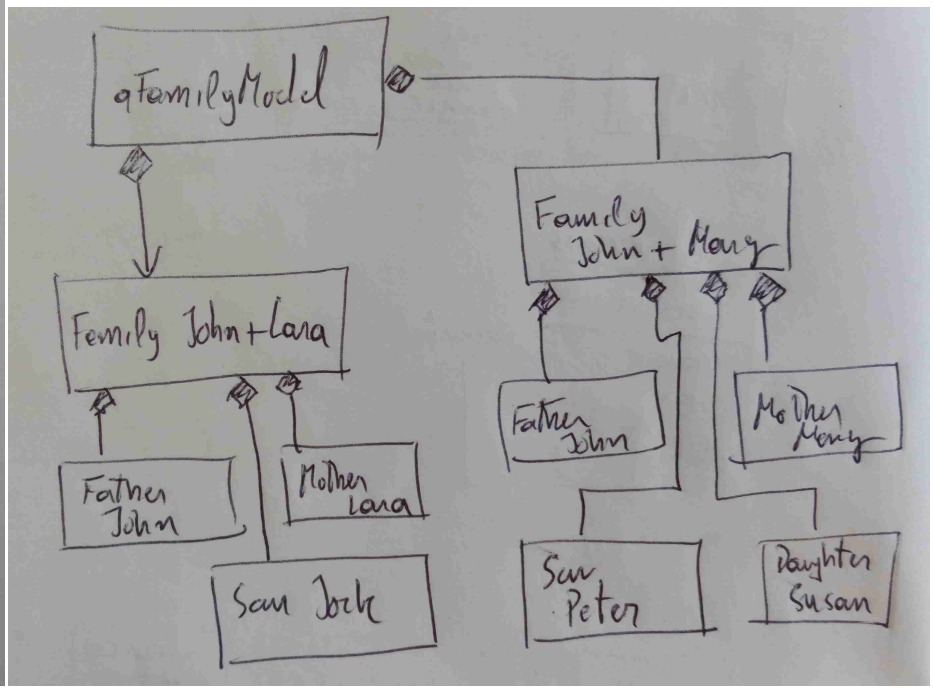


Family Model and Metamodel

The Metamodel

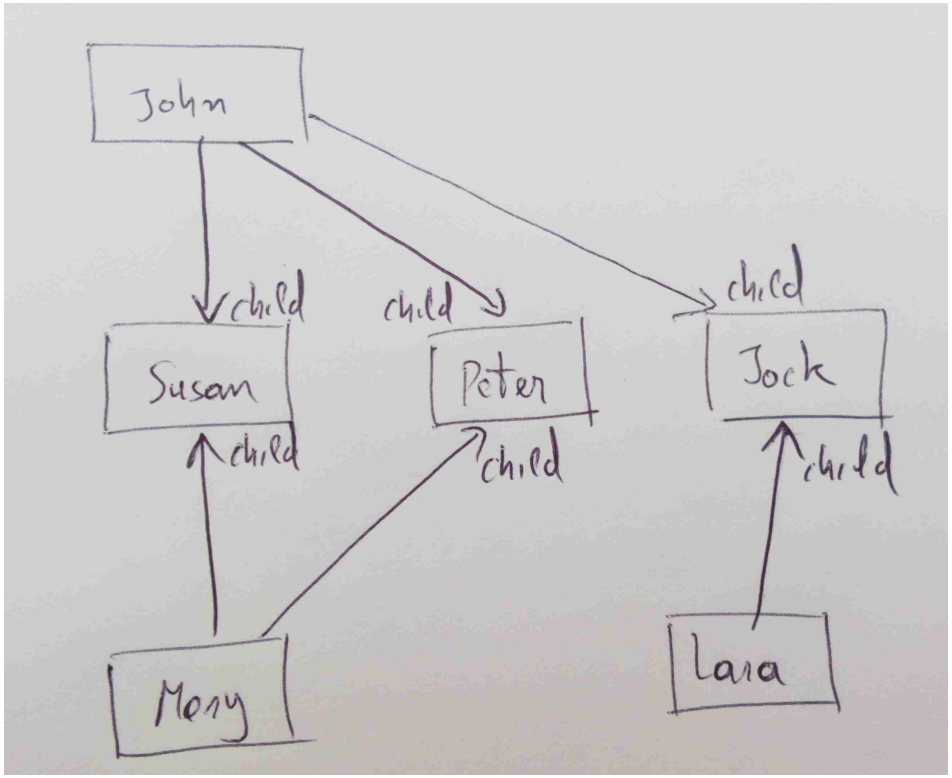


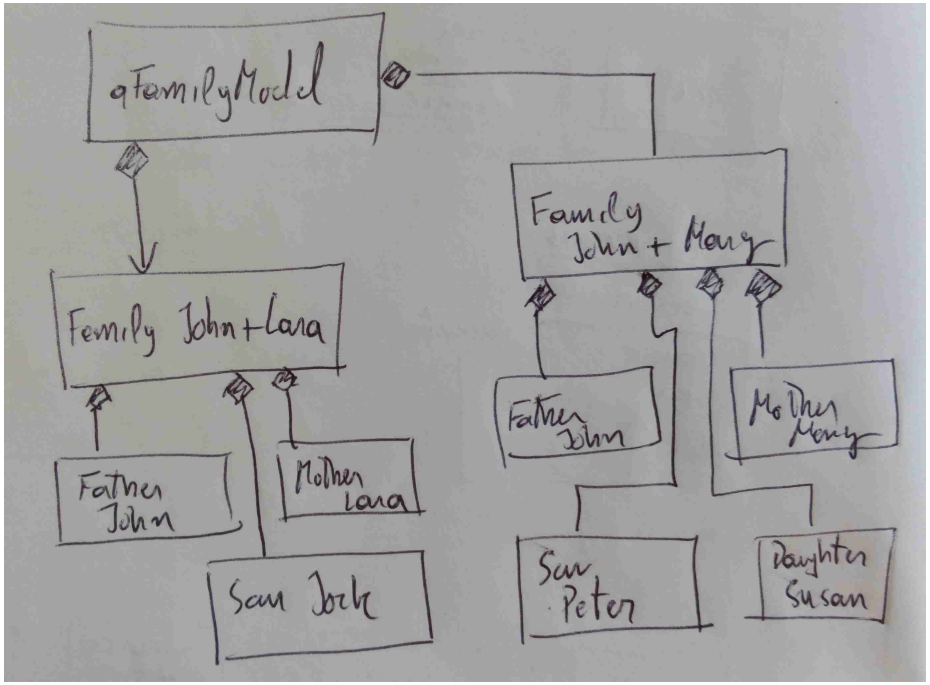
A Model...



What We Want to Achieve

From a model representing the relations between parents and sons/daughters we want to produce a model representing families





The main idea is: **each couple (male and female) sharing children is the base for a family.**

Steps

1. Create an Eclipse Project (ATL project or EMF Project)
2. Create the metamodels (ecore) in a folder named "models"
   - Optionally, create the genmodels and generate the model code.
3. Right click on each metamodel and execute "Register Metamodel"
4. Create the instance model (option Create Dynamic Instance) for the input model (in this case the tree model)
5. Create an ATL transformation file
   - Do not forget to specify the input metamodel (tree.ecore) and output metamodel (family.ecore)
6. Type de transformation rules in the ATL file (see next slides)
7. Create a new ATL Transformation Run Configuration
   - Do not forget to specify the input metamodel (tree.ecore) and output metamodel (family.ecore) as stated before
   - Specify as input model the tree instance model created previoulsy
   - Specify as output model a file name to receive the output model from the transformation
8. Run or Debug the transformation

Rule Tree2ModelFamily

Listing 1: Rule Tree2ModelFamily

```
-- One Tree Model produces One Family Model
rule Tree2ModelFamily {
    from
        m1: tree!Tree
    to
        m2: family!FamilyModel (
            -- At least we will have one family for each male person with children
            families <- m1.persons->select(x | x.sex=#MALE and not
                x.children->isEmpty() )
        )
    do {
        -- Initialize a helper collection with all people
        thisModule.persons <- m1.persons;
    }
}
```

Rule person2family

```
-- Each male parent originates a family
rule person2family {
    from p1: tree!Person ( p1.sex=#MALE and not p1.children->isEmpty() )
    using { -- Collect all females that share children with this male
        mothers : Sequence(tree!Person) = p1.children->collect(x | x.parent) ->
             flatten() ->select(x| x.sex=#FEMALE)->asSet()->asSequence(); }
    to
        f1: family!Family ( name <- p1.name )
    do { -- The first family
     f1.mother <- thisModule.newMother(mothers->first().name);
     f1.name <- p1.name + '+' + f1.mother.name;
     f1.father <- thisModule.newFather(p1.name);

     for (son in thisModule.brothers(thisModule.persons, p1, mothers->first())) {
           f1.sons <- thisModule.newSon(son.name); }

     for (daughter in thisModule.sisters(thisModule.persons, p1,
          mothers->first())) {
           f1.daughters <- thisModule.newDaughter(daughter.name); }

     -- We may have more families from the same father with other mothers...
     for (e in mothers->excluding(mothers->first())) {
           f1.model.families <- thisModule.newFamily(p1, e); }
    }
}
```

Rule newFamily

```
rule newFamily(f : tree!Person, m : tree!Person) {
    to
        fam : family!Family (
            name <- f.name + '+' + m.name
        )
    do {
        fam.father <- thisModule.newFather(f.name);
        fam.mother <- thisModule.newMother(m.name);

        for (son in thisModule.brothers(thisModule.persons, f, m)) {
            fam.sons <- thisModule.newSon(son.name);
        }

        for (daughter in thisModule.sisters(thisModule.persons, f, m)) {
            fam.daughters <- thisModule.newDaughter(daughter.name);
        }

        fam;
    }
}
```

Rule newFather

```
rule newFather(n : String) {
    to
        m : family!Father (
            name <- n
        )
    do {
        m;
    }
}
```

Rule newMother

```
rule newMother(n : String) {
    to
        m : family!Mother (
            name <- n
        )
    do {
        m;
    }
}
```

Rule newSon

```
rule newSon(n : String) {
    to
        m : family!Son (
            name <- n
        )
    do {
        m;
    }
}
```

Rule newDaughter

```
rule newDaughter(n : String) {
    to
        m : family!Daughter (
            name <- n
        )
    do {
        m;
    }
}
```

At the Beginning of the ATL File

```
-- @path tree=/tree2family/models/tree.ecore
-- @path family=/tree2family/models/family.ecore

module tree2families;
create OUT: family from IN: tree;

-- All persons
helper def: persons: Set(tree!Person) =
    Set{};

-- Input: all people, father, mother -> returns all brother (male children)
helper def: brothers(t: Set(tree!Person), p1: tree!Person, p2: tree!Person):
     Set(tree!Person) =
       t -> select(x | x.parent->includes(p1) and x.parent->includes(p2) and
            x.sex=#MALE ) -> asSet();

-- Input: all people, father, mother -> returns all sisters (female children)
helper def: sisters(t: Set(tree!Person), p1: tree!Person, p2: tree!Person):
     Set(tree!Person) =
       t -> select(x | x.parent->includes(p1) and x.parent->includes(p2) and
            x.sex=#FEMALE ) -> asSet();
```