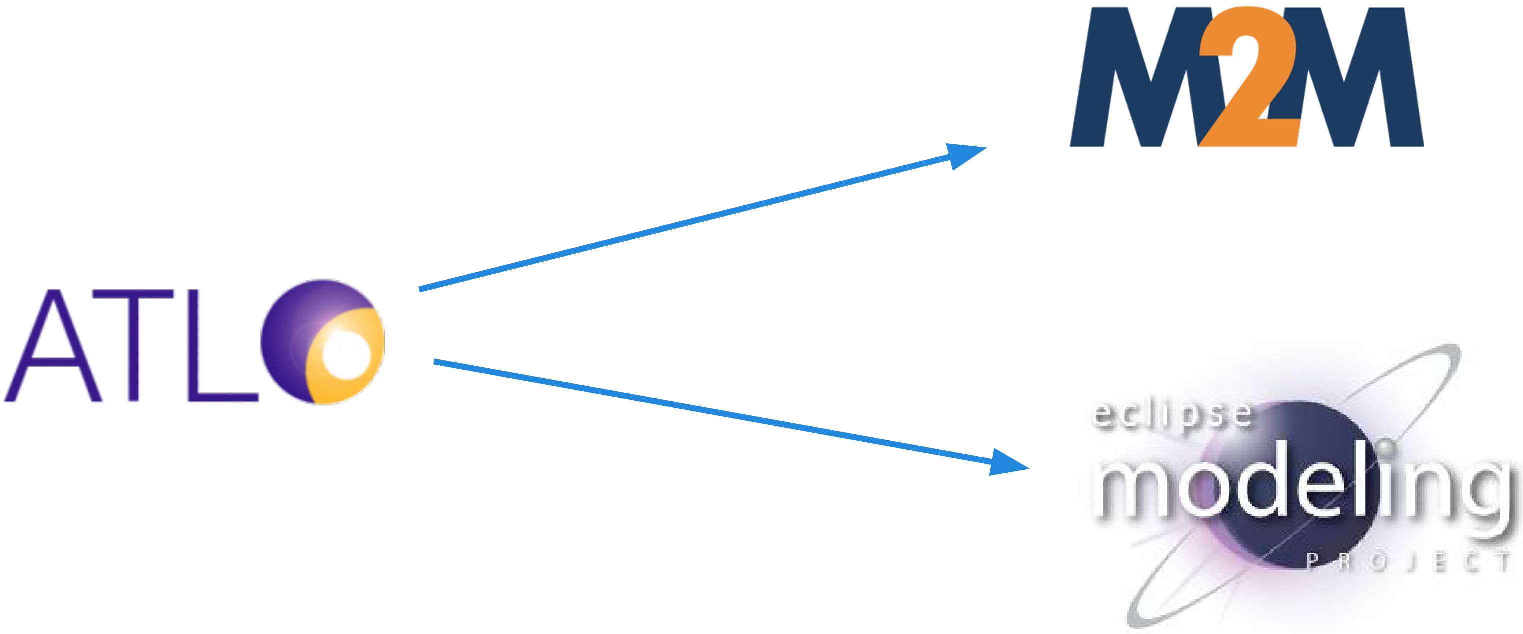
From Role-Playing Game to Petrinet, the ATL way.

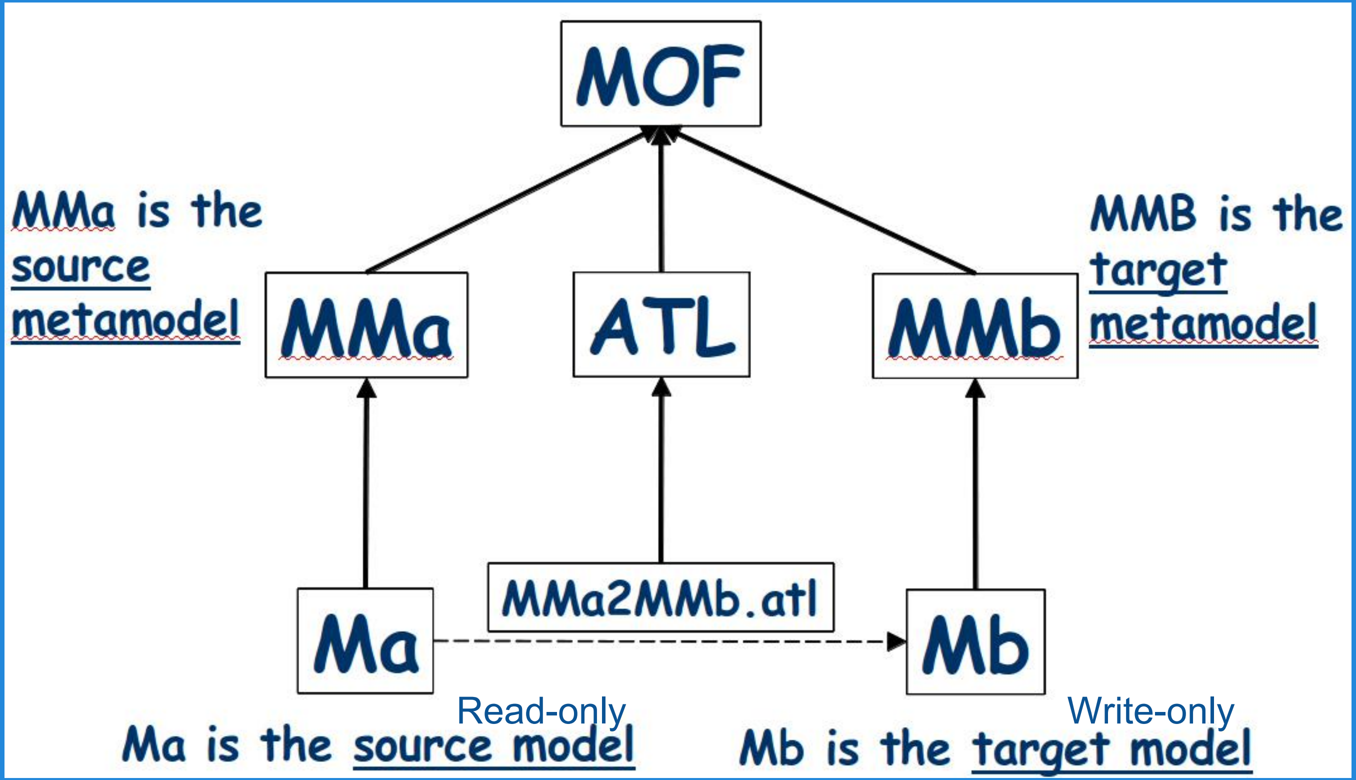
By Daan Janssens
Course: Model-driven Engineering 2013-2014
Supervisor: Prof. Vangheluwe

Universiteit Antwerpen

ATL

What was ATL again?





Transformation pattern

Rules

```
rule NameOfTheRule {
    from
        s: MMa!ClassA
    to
        t: MMb!ClassB (
            t.attribute <- s.attribute
        )
}
```

Rules

```
rule NameOfTheRule {
    from
        s: MMa!ClassA
    to
        t: MMb!ClassB (
            t.attribute <- s.attribute
        )
}
```

Name of the rule

Source element on which rule should be performed

Created target element

value of attribute of s assigned to attribute of t.

Experiment

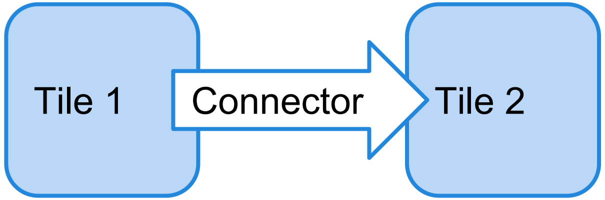
Introduction

- Computer games can be complex
- DSL helps coping with complexity.
- game models can help verificate various game properties.

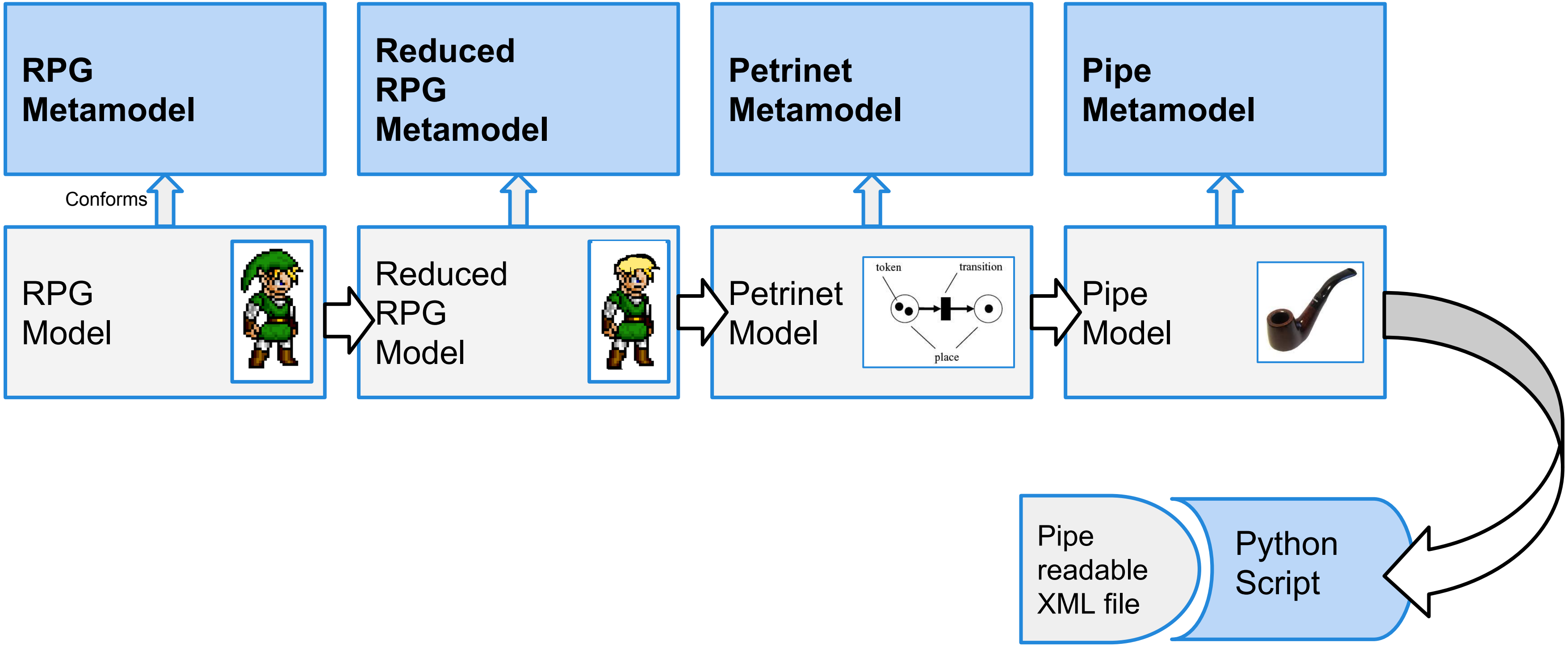
Our interest: *the completability of an RPG game*

Analysis : RPG Formalism

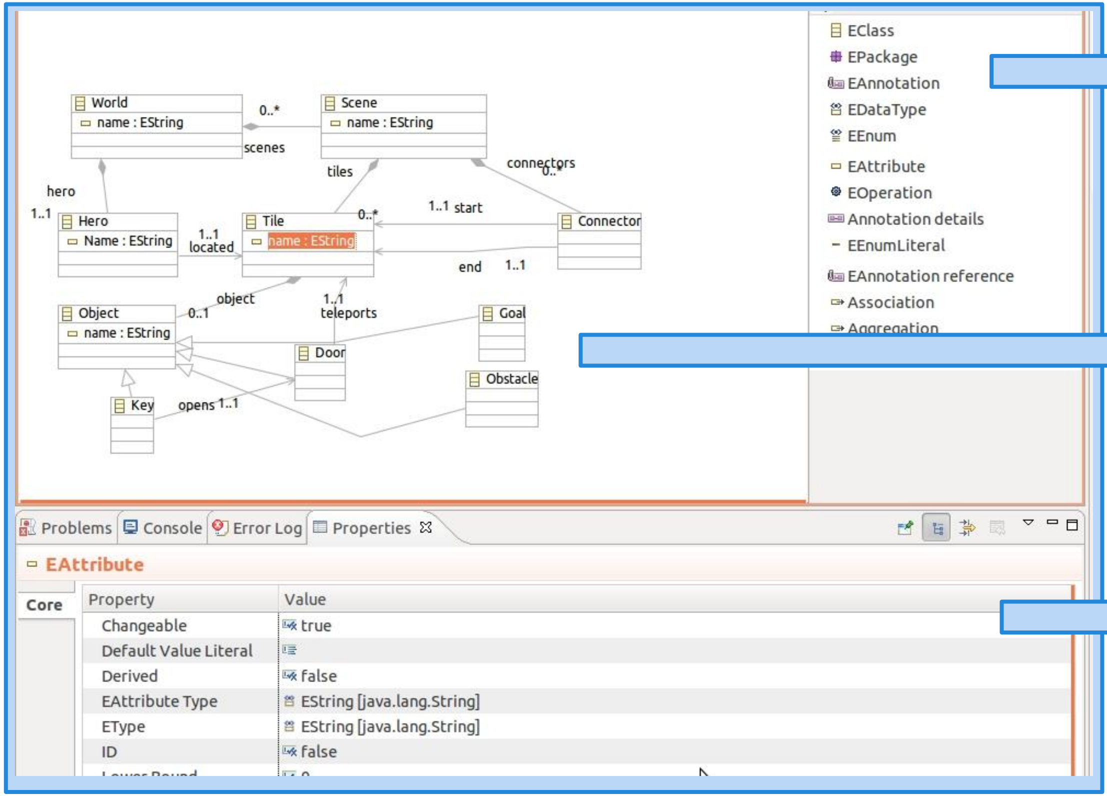
- Comparable to what we saw in practice
- Main difference:
  - Hero wins when <u>all</u> goals are reached.
  - No villains/traps, focus on path to goals.
  - Tiles are connected by #Connectors



Approach


RPG Metamodel
Reduced RPG Metamodel
Petrinet Metamodel
Pipe Metamodel
Conforms
RPG Model
Reduced RPG Model
Petrinet Model
token
transition
place
Pipe Model
Pipe readable XML file
Python Script

Design



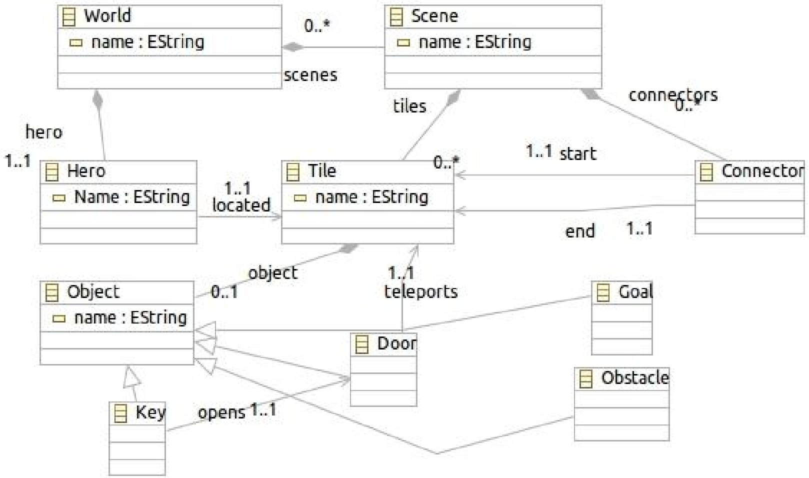
Drag and drop elements

Visual representation

Class attribute properties

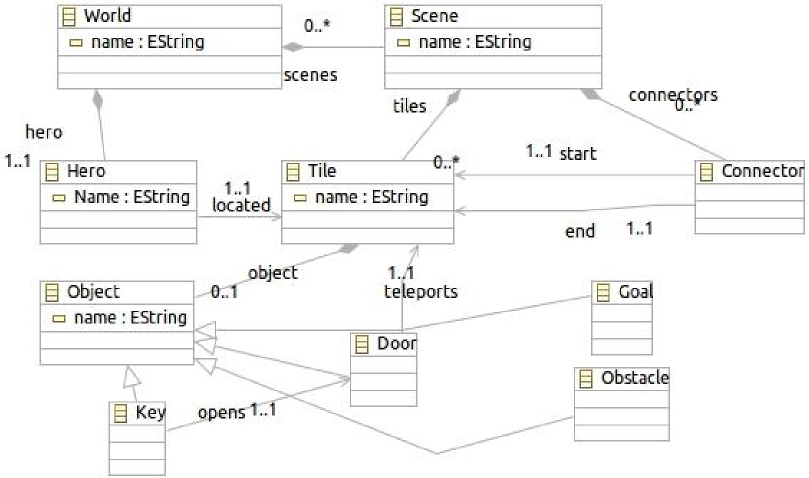
Design: RPG Metamodel

RPG Metamodel:

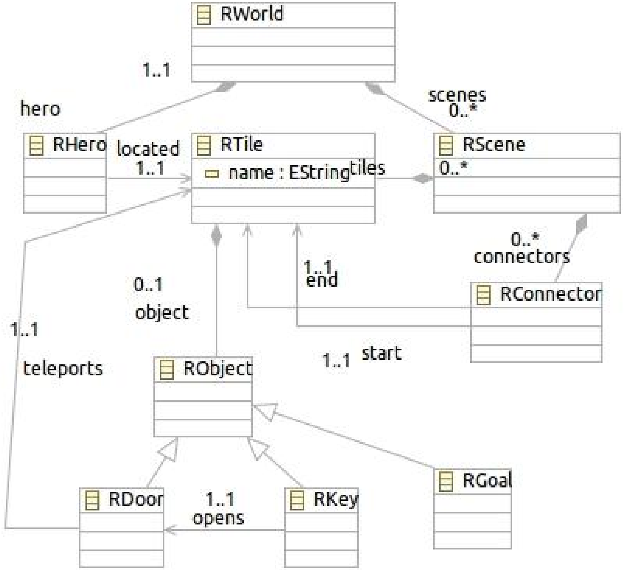


Design: RPG VS RRPG Metamodel

RPG Metamodel:

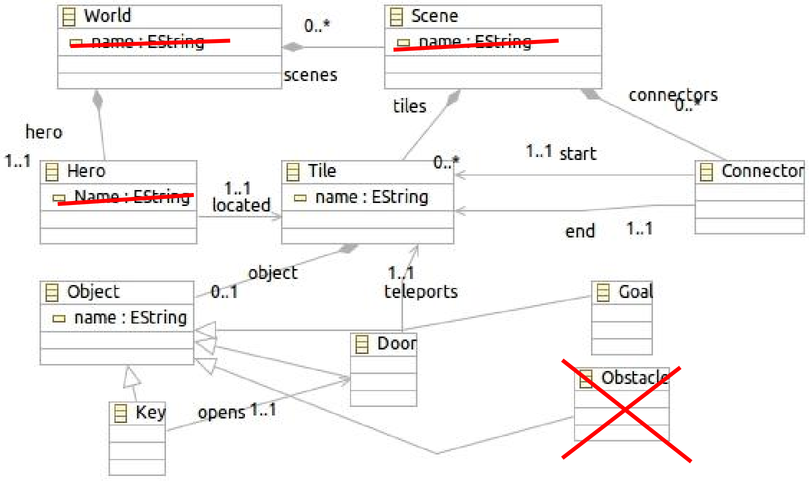


RRPG Metamodel:

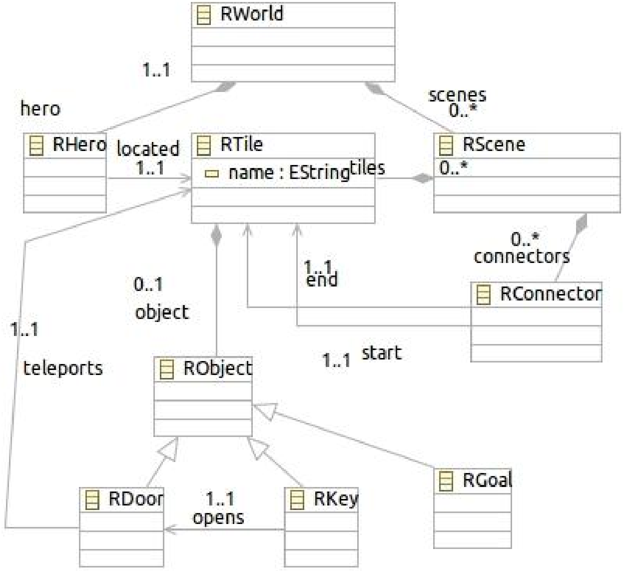


Design: RPG VS RRPG Metamodel

RPG Metamodel:

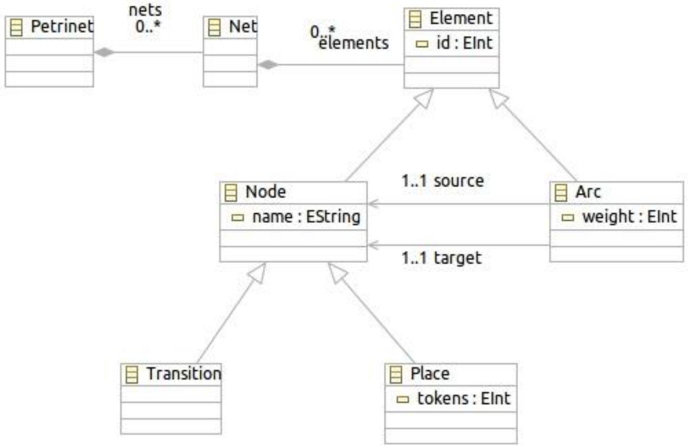


RRPG Metamodel:

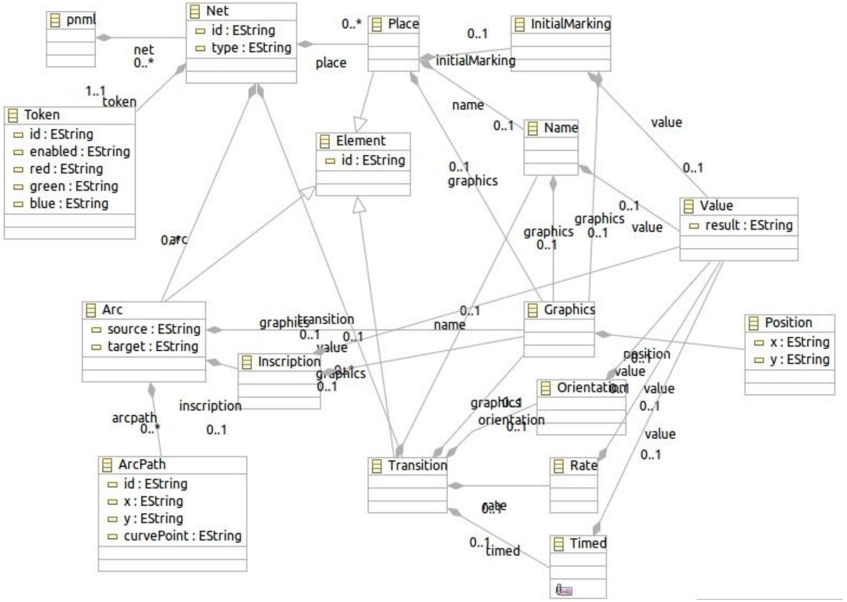


Design: Petrinet vs Pipe Metamodel

Petrinet Metamodel:

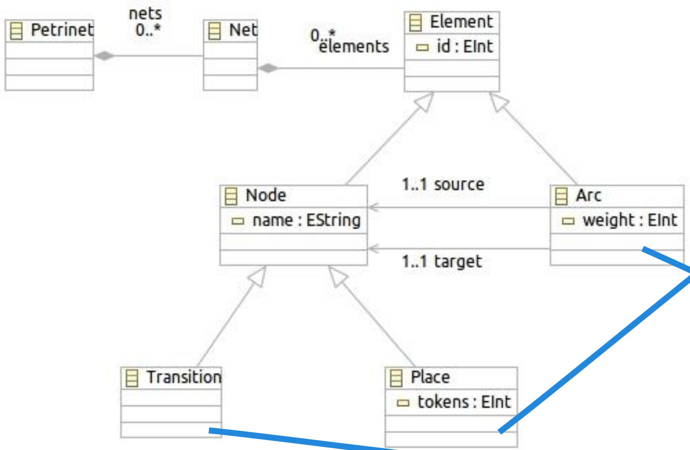


Pipe Metamodel:

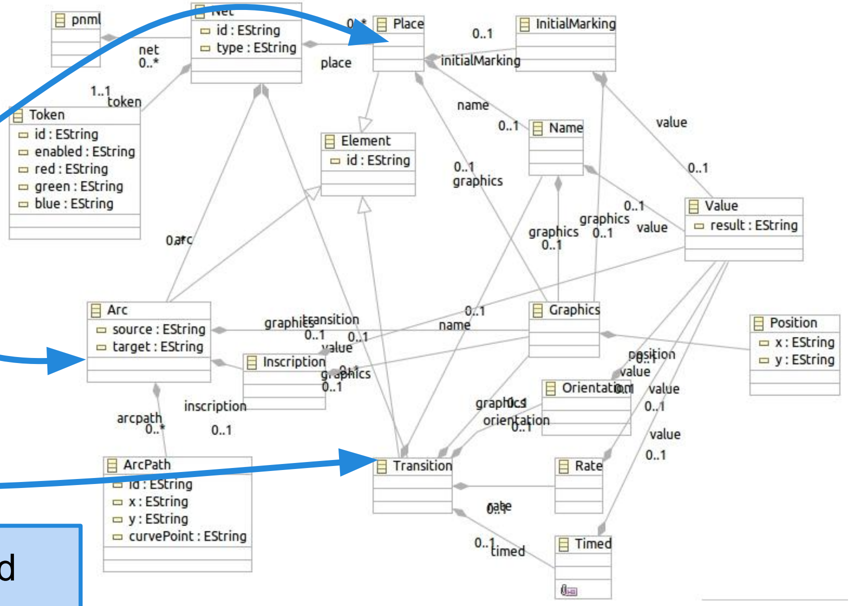


Design: Petrinet vs Pipe Metamodel

Petrinet Metamodel:



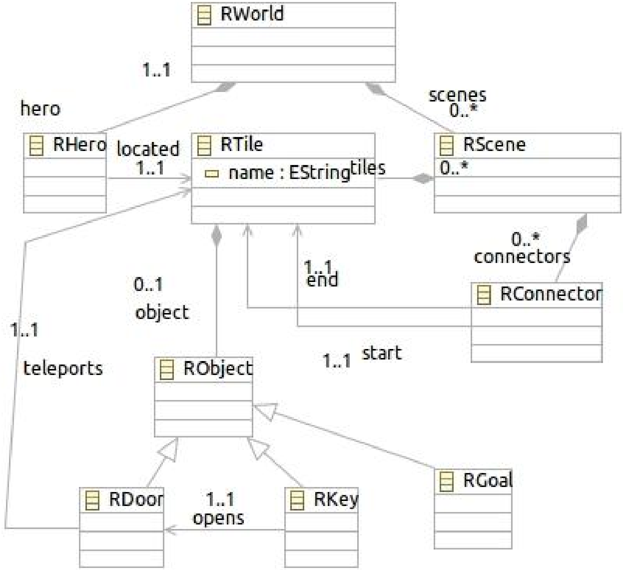
Pipe Metamodel:



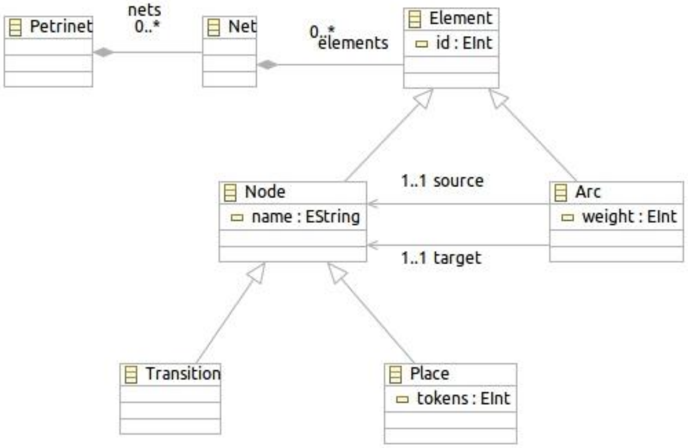
1-on-1 mapping where additional elements are added

Design: RRPG VS Petrinet

RRPG Metamodel:

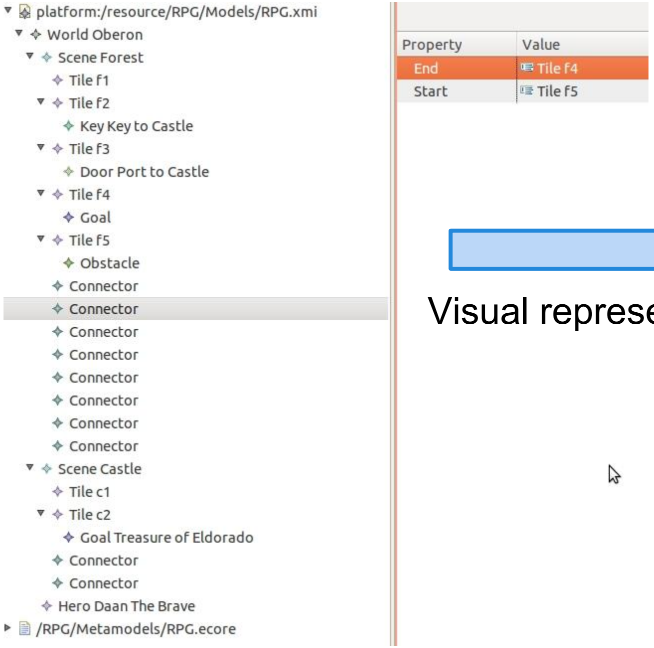


Petrinet Metamodel:

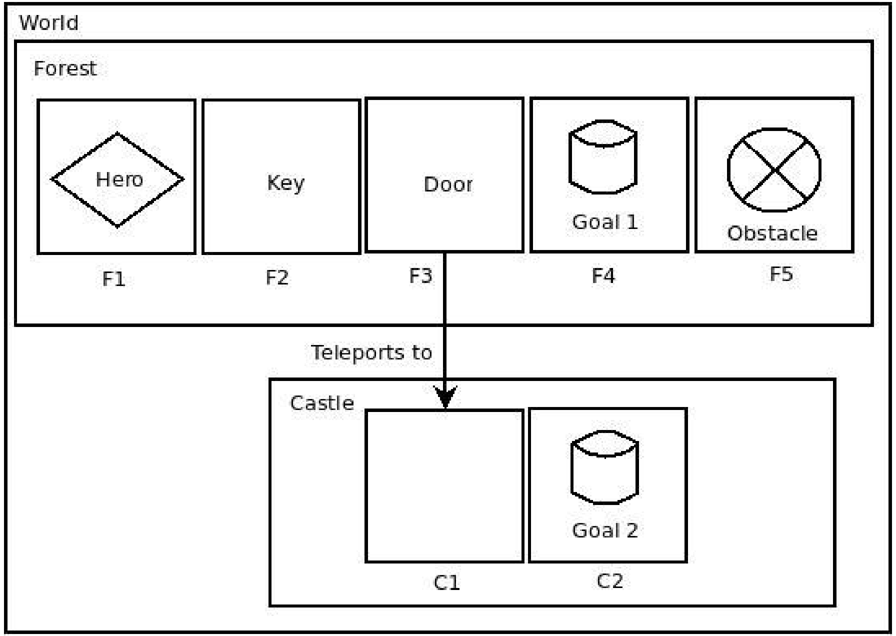


Implementation: Initial RPG model

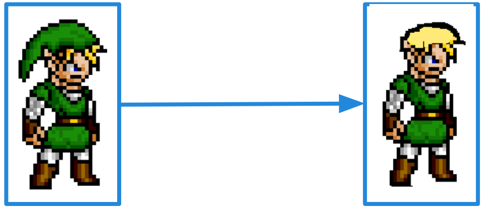
initial RPG model:



Visual representation



RPG2RRPG



Implementation: RPG2RRPG.atl

```
module RPG2RRPG;
create OUT : RRPG from IN : RPG;
```

Defines metamodels for input model & output model

```
helper context RPG!Scene def: getValidTiles : Set(RPG!Tile)
= self.tiles->select(c | not .object.oclIsKindOf(RPG!Obstacle));
```

Helper method getValidTiles on a Scene object of the RPG metamodel returns a subset of its tiles that do not contain Obstacles.

Expressions in ATL are written in OCL

Implementation: RPG2RRPG.atl

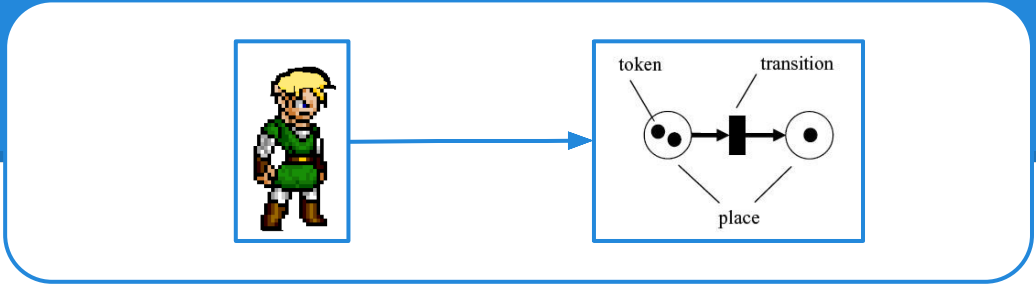
```
rule Scene2RScene {
 from
    s : RPG!Scene
 to
    rs : RRPG!RScene (
      tiles <- (s.getValidTiles),
      connectors <- (s.getValidConnectors)
    )
}
```

Is applied on every Scene object. and only assigns Tiles with no obstacles to them.

```
rule Tile2RTile {
 from
    t : RPG!Tile (not
    t.object.oclIsKindOf(RPG!Obstacle))
 to
    rt : RRPG!RTile (
      object <- t.object,
      name <- t.name
     )
}
```

Uses a guard to ignore tiles with Obstacles.

RRPG2Petrinet



Implementation: RRPG2Petrinet.atl

```
helper def : id: Integer = 1;
```

integer variable, to give unique id to each created petrinet element.

```
rule TileWithHero2Place {
 from
   r : MM!RTile (r =
   thisModule.getHero.located)
 to
    p1 : MM1!Place(
      id <- thisModule.id,
      tokens <- 1,
      name <- r.name
    )
 do {
  thisModule.id <- thisModule.id+1;
 }
}
```

2 variants of rules that work on RTile objects. The other one is for tiles without a hero and assigns 0 tokens to the created place.

Implementation: RRPG2Petrinet.atl

It quickly gets harder and larger!

```
rule Door2DoorPlace {
  from
    r : MM!RDoor
   using{
     Tile : MM!RTile = thisModule.getTileFromObject(r);
   }
   to
     a1 : MM1!Arc(
        source <- (Tile) ,
        target <- (t1),
        id <- thisModule.id
     ),
     t1 : MM1!Transition(
        id <- thisModule.id+1
     ),
     a2 : MM1!Arc(
       source <- (t1),
       target <- (r.teleports),
       id<- thisModule.id+2
     )
    do {
      thisModule.id <- thisModule.id+3;
      thisModule.elements <- thisModule.elements->including
(a1);
      thisModule.elements <- thisModule.elements->including(t1);
      thisModule.elements <- thisModule.elements->including
(a2);
    }
}
```

multiple created target elements for a single source element

```
rule Key2KeyPlace {
  from
    r : MM!RKey
  to
     p1 : MM1!Place(
        id <- thisModule.id,
        tokens <- 0,
        name <- 'KeyTaken'
     ),
     a1 : MM1!Arc(
        source <- p1,
        target <- thisModule.resolveTemp(r.opens,'t1'),
        id <- thisModule.id+1
     ),
     p2 : MM1!Place(
        id <- thisModule.id+2,
        tokens <- 1,
        name <- 'KeyNotYetTaken'
     )
    do {
        thisModule.id <- thisModule.id+3;
        thisModule.elements <- thisModule.elements->including
(p1);
        thisModule.elements <- thisModule.elements->including
(a1);
        thisModule.elements <- thisModule.elements->including
(p2);
    }
}
```

Refering to new created elements, from within other rules

Implementation: RRPG2Petrinet.atl

It quickly gets harder and larger!

```
rule Door2DoorPlace {
  from
    r : MM!RDoor
   using{
     Tile : MM!RTile = thisModule.getTileFromObject(r);
    }
    to
     a1 : MM1!Arc(
        source <- (Tile) ,
        target <- (t1),
        id <- thisModule.id
     ),
     t1 : MM1!Transition(
        id <- thisModule.id+1
     ),
     a2 : MM1!Arc(
       source <- (t1),
       target <- (r.teleports),
       id<- thisModule.id+2
     )
    do {
     thisModule.id <- thisModule.id+3;
     thisModule.elements <- thisModule.elements->including
(a1);
     thisModule.elements <- thisModule.elements->including(t1);
     thisModule.elements <- thisModule.elements->including
(a2);
    }
}
```

multiple created target elements for a single source element

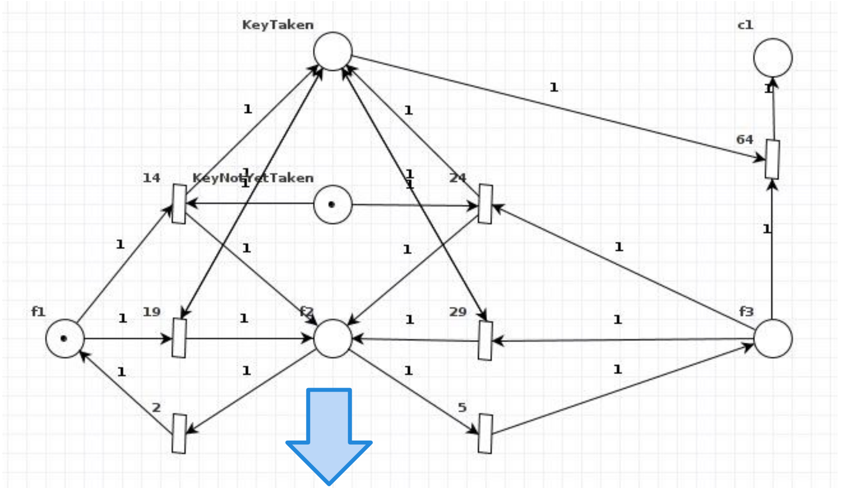
```
rule Key2KeyPlace {
  from
    r : MM!RKey
  to
     p1 : MM1!Place(
       id <- thisModule.id,
       tokens <- 0,
       name <- 'KeyTaken'
     ),
     a1 : MM1!Arc(
        source <- p1,
        target <- thisModule.resolveTemp(r.opens,'t1'),
        id <- thisModule.id+1
     ),
     p2 : MM1!Place(
        id <- thisModule.id+2,
        tokens <- 1,
        name <- 'KeyNotYetTaken'
     )
        thisModule.elements <- thisModule.elements->including
(p2);
     }
}
```

Refering to new created elements, from within other rules

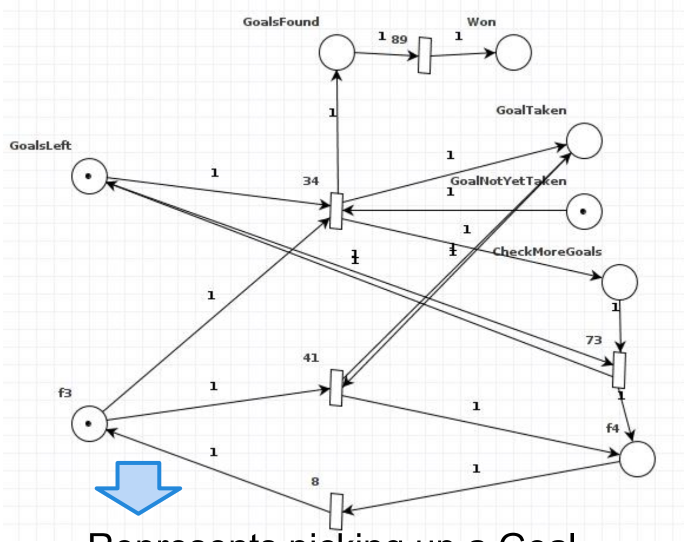
Too complex for a single presentation!
Read the paper.

Implementation: RRPG2Petrinet.atl

- Idea is to form the following petrinet constructions:

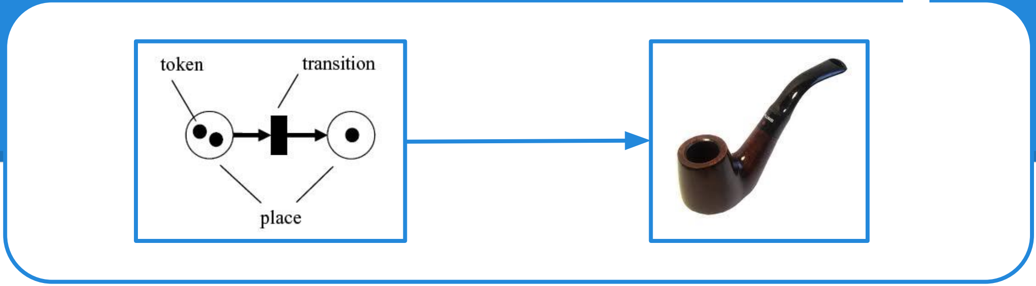


Represents picking up a key and teleporting.



Represents picking up a Goal and checking if it was the last one.

Petrinet2Pipe



Implementation: Petrinet2Pipe.atl

A 1-on-1 mapping of the Arcs, Transitions and Places, where additional elements are added to them.

An example:

```
rule Place2Place{
 from
   p : MM!Place
 to
   pn : MM1!Place(
     id <- p.id.toString(),
     graphics <- g1,
     name<-n,
     initialMarking<-im
 ),
```

```
g1 : MM1!Graphics(
   position <- pos
),
pos : MM1!Position(
   x <- '10',
   y <- '10'
),
v : MM1!Value(
   result<-p.name
),
```

```
n : MM1!Name(
   value <- v,
   graphics <- g2
),
g2 : MM1!Graphics(
),
im : MM1!InitialMarking(
   value<-v2,
   graphics<-g3
),
v2 : MM1!Value(
   result<-p.tokens,
g3 : MM1!Graphics(
)
```

Implementation: Python script

The python script changes XMI files into XML files that can be read by Pipe

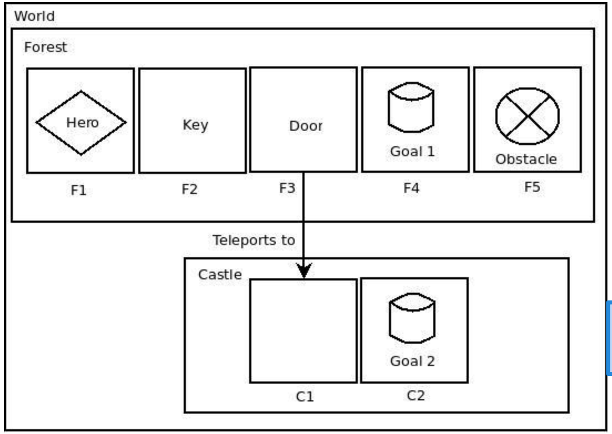
Pipe.xmi:

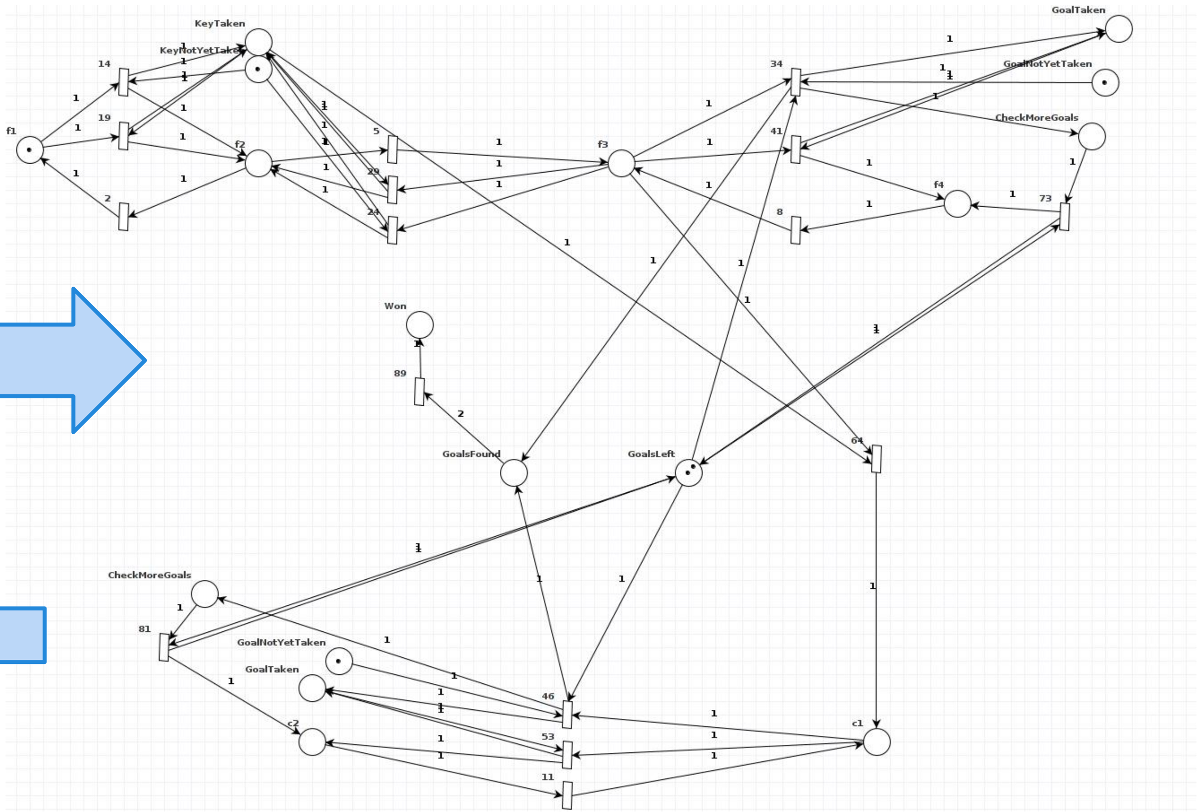
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Pipe:pnml xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:
Pipe="http://www.eclipse.org/uml2/4.0.0/UML">
  <net id="RPGame" type="P/T net">
        <transition id="24">
        <graphics>
        <position x="10" y="10"/>
        </graphics>
        <orientation>
        <value result="1"/>
        </orientation>
        <timed>
        <value result="true"/>
        </timed>
        <name>
        <graphics/>
        <value result="24"/>
        </name>
        </transition>
```

Pipe.xml:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<pnml>
  <net id="RPGame" type="P/T net">
        <transition id="24">
        <graphics>
        <position x="10" y="10"/>
        </graphics>
        <orientation>
        <value>1</value>
        </orientation>
        <timed>
        <value>true</value>
        </timed>
        <name>
        <graphics/>
        <value>24</value>
        </name>
        </transition>
```

Result:





Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	true

Shortest path to deadlock: 14 5 34 73 8 64 46 89

Conclusion

Conclusion:

- Amount of rules + size escalates quickly.
- Creation of multiple target objects from single source makes rules complex.
- Lack of visual nature

Main conclusion: *believe that ATL is the perfect language for performing small/medium sized 1-to-1 mapped transformations.*

Comparison with AtomPM:

- Visual representation of rules => more understandable.
- Both offer a visual editor for creating the metamodels.
- AtomPM allows ordering the application of the rules, while ATL executes all rules on their matched source elements at the same time.

Main conclusion: *Believe that a tool like AtomPM could clear the job in a significant less amount of time and still end up being more readable, reuseable and understandable*

Thank you