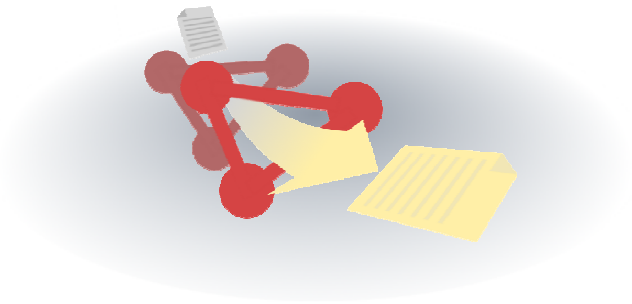
CODE GENERATION 2009

June 16-18 Cambridge UK

The RGen Ruby Code Generator and its use in Automotive SW Development

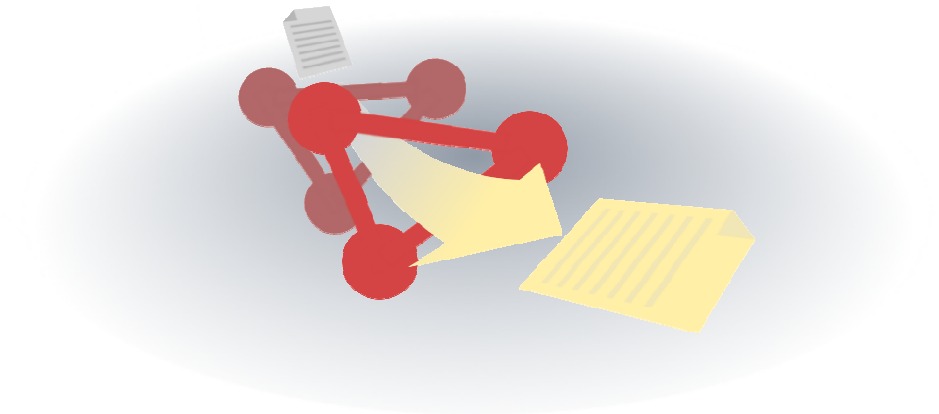
Part I: RGen - Ruby Modelling and Code Generation Framework

Martin Thiede



What‘s RGen?

- Ruby Modelling and Code Generation Framework
- http://ruby-gen.org
- 0,3 MB download
- ~5000 LOC
- No dependencies apart from Ruby Interpreter
- Platform independent
- Permissive MIT License



The RGen Idea
Design Goals

Internal Ruby DSLs for the domain of Modelling and Code Generation!

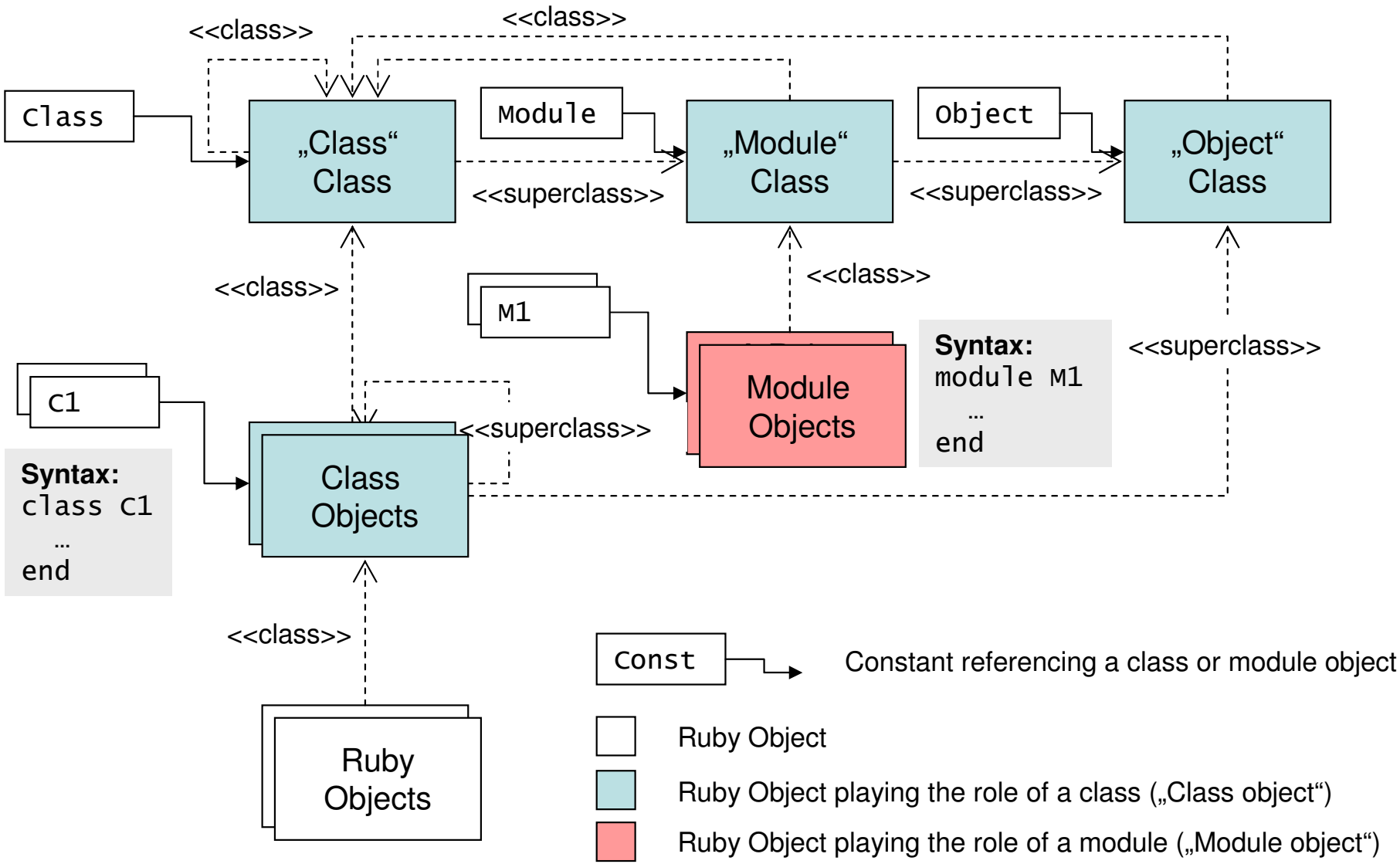
- Simple and Lightweight
  - Simple design, Small core
  - No dependencies

- Flexible and Powerful
  - It‘s all just Ruby code
  - Use Ruby to adapt and extend

What‘s Ruby?

- Dynamic, reflective, OO general purpose programming language
- http://ruby-lang.org
- Concise, expressive syntax
- Well suited for building internal DSLs
- Several Open Source implementations
- Popular "Ruby on Rails" project

Ruby Primer
(Almost) Everything is an Object

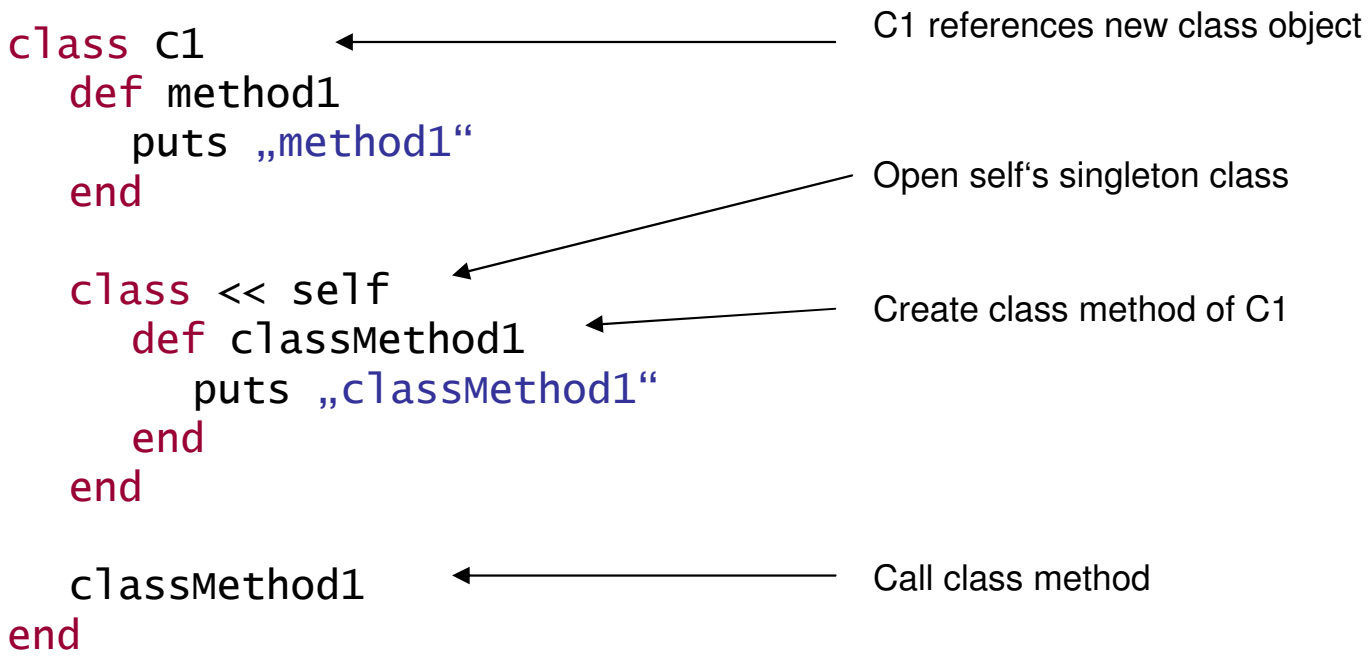


Ruby Primer
Classes, Modules and Constants

- Classes are Modules are Objects
- Constants are used to refer to class- and module-objects
- Modules are namespaces for constants
- Modules group methods (for „Mix-Ins“)
- Per-Object methods (Singleton Classes)

Ruby Primer
Class and Module Definitions

- Class/Module definitions are executed while being loaded
- Contained code is evaluated in context of the Class/Module object
- Classes/Models can be reopened at any time („Open Classes“)



Ruby Primer
Mix-Ins and Meta-Programming

- Methods added at runtime

```
module M1
   module M2
      def method2
         puts „method2“
      end
      module_eval <<-END
         def method3
            puts „method3“
         end
      END
   end
end

class C1
   include M1::M2
end

c1 = C1.new
c1.method1
c1.method2
```

M1 references new module object
M1 is the namespace of M2

Add method „method2“ to module

Add method „method3“ using Meta-Programming

String

Reopen class C1

Mix in module M1::M2

Create an instance of C1
Call method from original definition
Call mixed in method

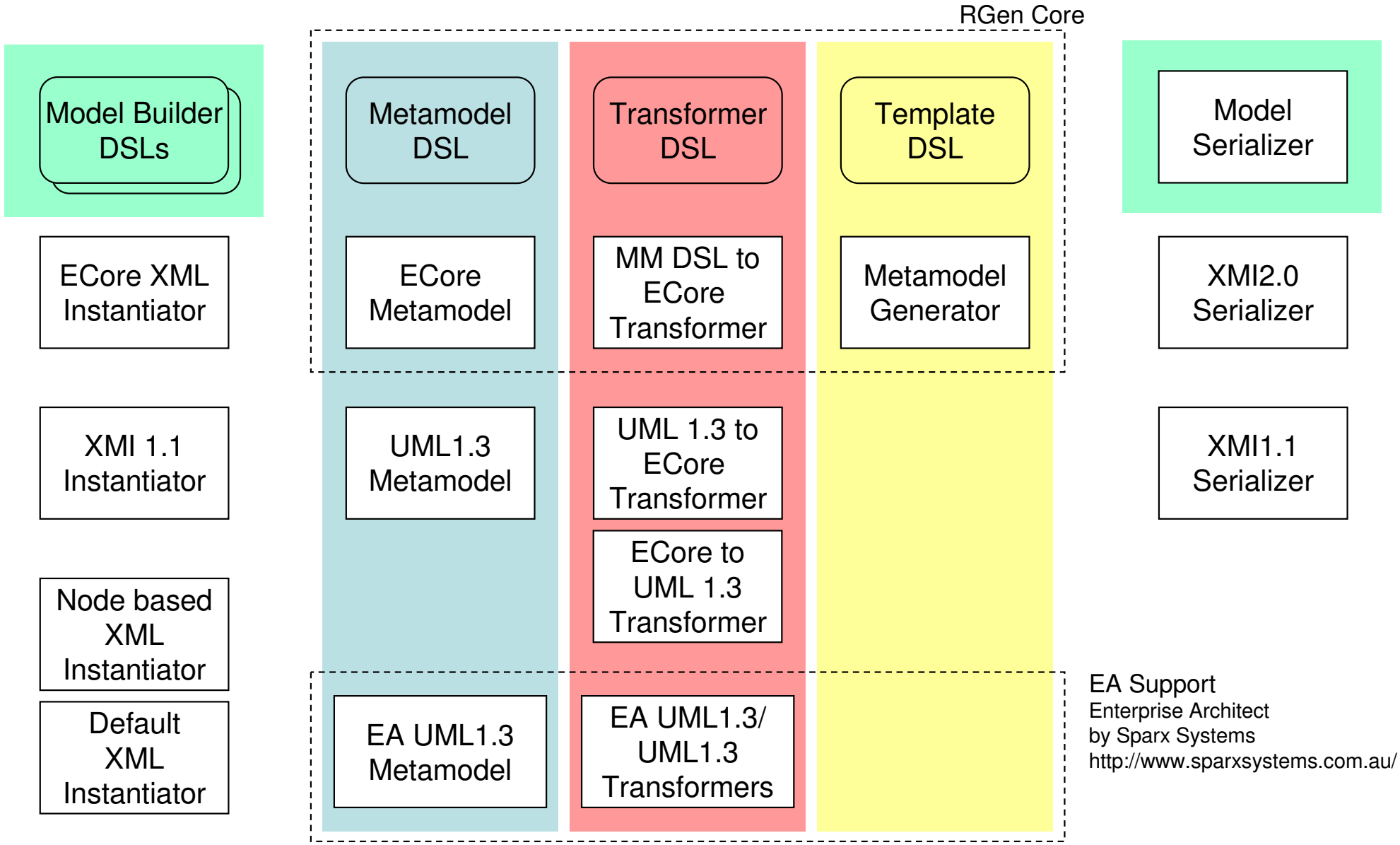
Ruby Primer
Blocks/Closures

- Blocks: code between `do..end` or `{..}`
- Blocks can be turned into objects, stored and executed later on (aka. Ruby Proc objects, closures)

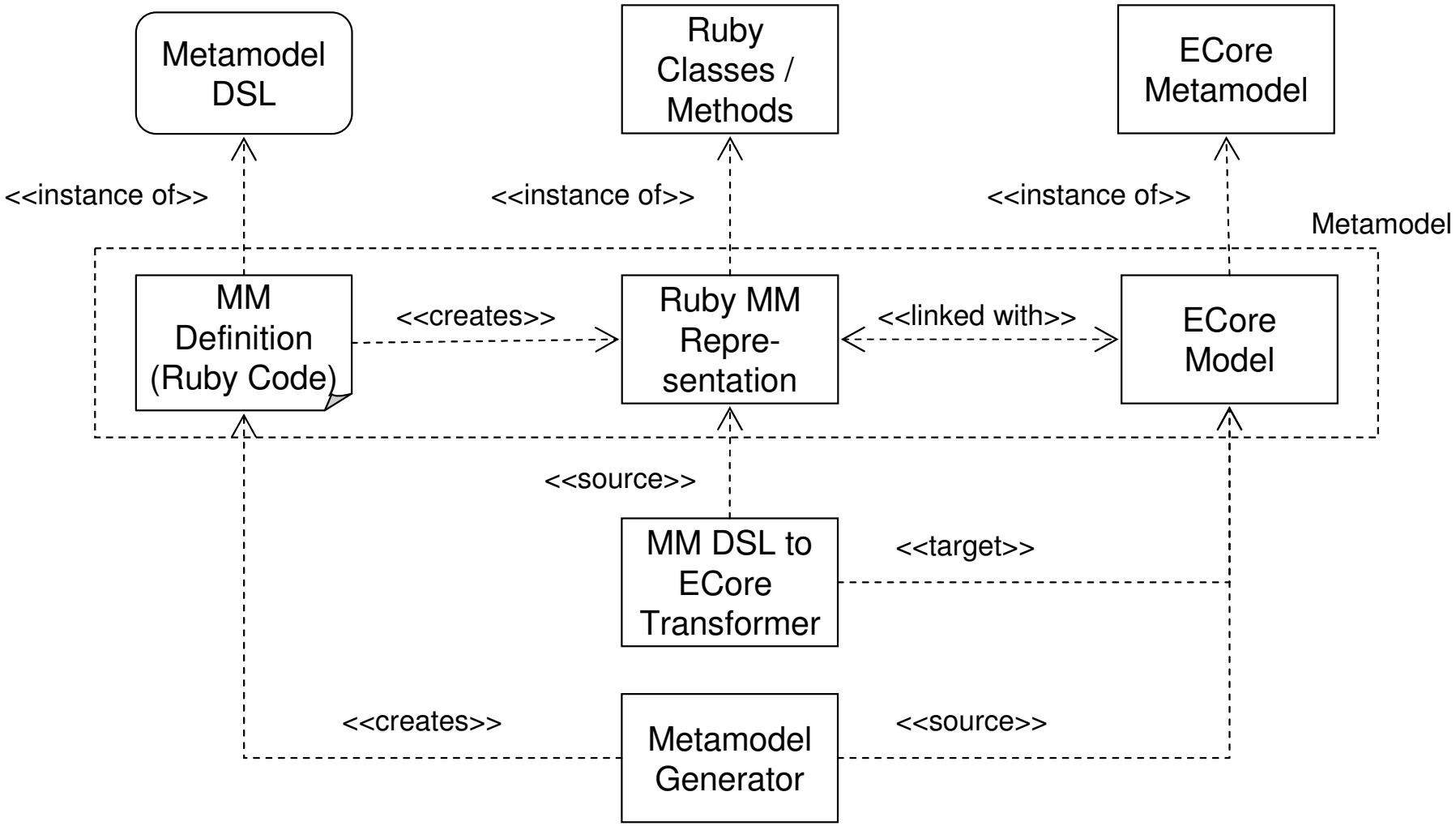
```ruby
numbers = [1, 2, 3, 4]

numbers.select { |n| n > 2 }
  # => [3, 4]

faktor = 2
numbers.collect do |n|
  n * faktor
end
  # => [2, 4, 6, 8]
```


The „select“ and „collect“ methods of Array use a block as parameter

RGen Release Package Overview



RGen Metamodels Overview



RGen Metamodel DSL Commands

```
module Module1
  extend RGen::MetamodelBuilder::ModuleExtension
  annotation

  class Class1 < RGen::MetamodelBuilder::MMBase
  end

  class Class2 < Class1
    abstract
    annotation
    has_attr
    has_one
    has_many
    contains_one_uni
    contains_many_uni
    one_to_one
    one_to_many
    many_to_one
    many_to_many
    contains_one
    contains_many
  end

  class Class3 < RGen::MetamodelBuilder::MMMultiple(Class1, Class2); end
end
```

Enable RGen Metamodel DSL

Inheritance

Mark abstract

Add annotation

Define attribute

Define unidirectional reference

Define bidirectional reference

Multiple Inheritance

RGen Metamodel DSL
Statemachine Example

```
module StatemachineMetamodel
  extend RGen::MetamodelBuilder::ModuleExtension

  class Statemachine < RGen::MetamodelBuilder::MMBase
    has_attr ,name‘, String
  end

  class State < RGen::MetamodelBuilder::MMBase
    has_attr ,name‘, String
  end
  Statemachine.contains_many ,state‘, State, ,statemachine‘

  class CompositeState < State
  end
  CompositeState.contains_many ,subState‘, State, ,compositeState‘

  class Transition < RGen::MetamodelBuilder::MMBase
  end
  Statemachine.contains_many ,transition‘, Transition, ,statemachine‘

  Transition.one_to_many ,source‘, State, ,outTrans‘
  Transition.one_to_many ,target‘, State, ,inTrans‘
end
```

RGen Metamodel DSL
Reflection using the ECore Model

- Ruby Metamodel classes and module are connected with an ECore instance

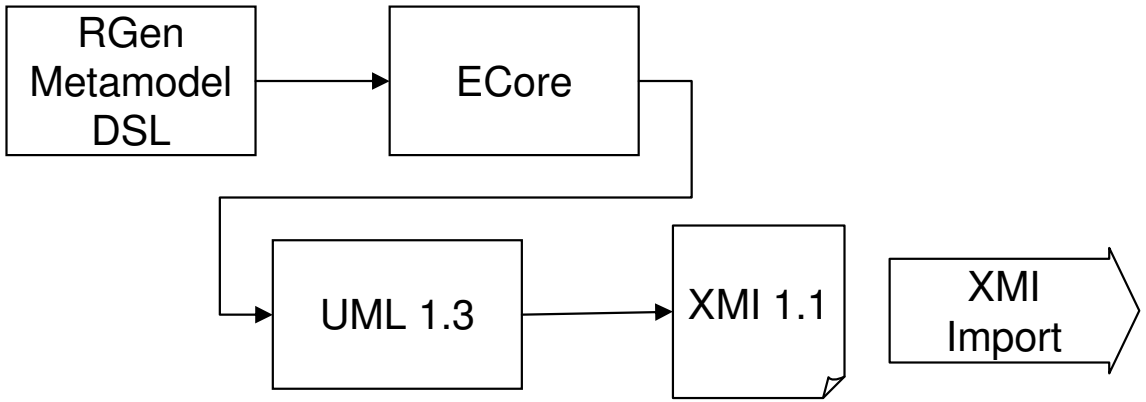
```
pack = StatemachineMetamodel.ecore
assert pack.is_a?(RGen::ECore::EPackage)

stateClass = pack.eClassifiers.find {|c| c.name == "State"}

assert_equal ["outTrans", "inTrans", "compositeState", "statemachine"],
  stateClass.eAllReferences.name
```

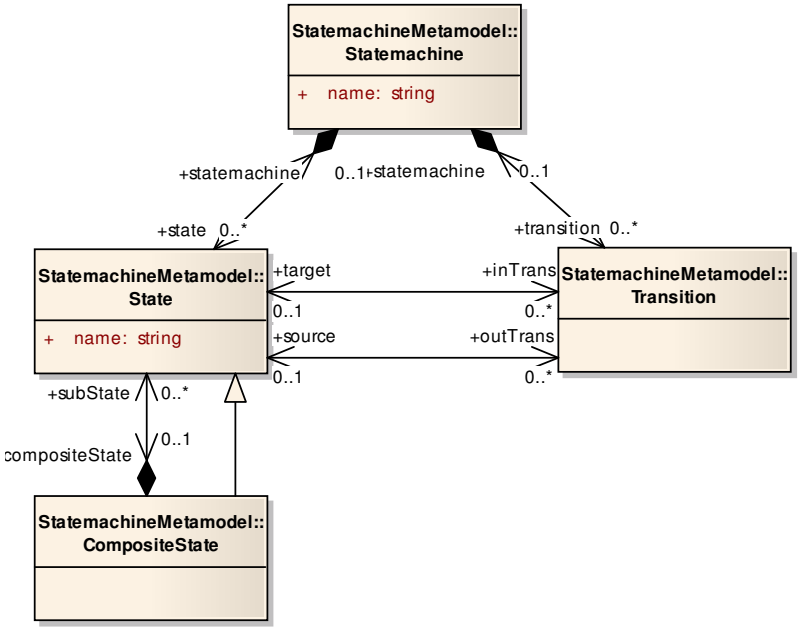
- ECore Export

```
File.open("statemachine.ecore","w") do |f|
   ser = RGen::Serializer::XMI20Serializer.new(f)
   ser.serialize(StatemachineMetamodel.ecore)
end
```

RGen Metamodel DSL
Example: Metamodel to UML Class Diagram

```
envUML = RGen::Environment.new

p = StatemachineMetamodel.ecore          <-- RGen MM DSL to ECore

ECoreToUML13.new(nil, envUML).trans(p)   <-- ECore to UML

EASupport.serializeUML13ToXMI11(
   envUML, "statemachine_metamodel_import.xml", :keep_ids => true)
```





RGen Metamodel DSL
Example: UML Class Diagram to Metamodel

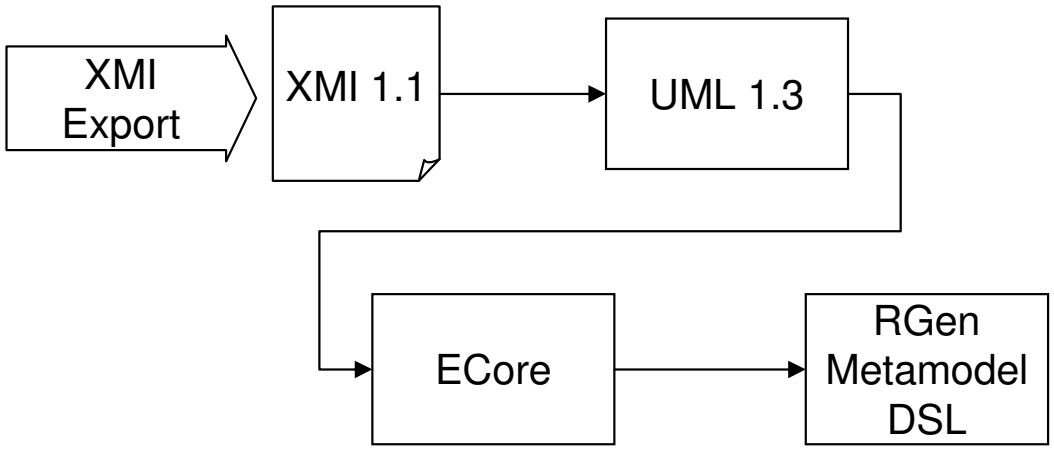
```
envUML = RGen::Environment.new
EASupport.instantiateUML13FromXMI11(envUML, "statemachine_metamodel.xml")

envECore = RGen::Environment.new
UML13ToECore.new(envUML, envECore).transform
```

UML to ECore

```
p = envECore.find(:class => RGen::ECore::EPackage,
   :name => "StatemachineMetamodel").first
```

ECore to RGen MM DSL

```
generateMetamodel(p, "statemachine_metamodel_regenerated.rb")
```




```
module StatemachineMetamodel
     extend RGen::MetamodelBuilder::ModuleExtension

     class Statemachine < RGen::MetamodelBuilder::MMBase
          has_attr ,name‘, String
     end

     class State < RGen::MetamodelBuilder::MMBase
          has_attr ,name‘, String
     end
     Statemachine.contains_many ,state‘, State, ,statemachine‘

     class CompositeState < State
     end
     CompositeState.contains_many ,subState‘, State, ,compositeState‘

     class Transition < RGen::MetamodelBuilder::MMBase
     end
     Statemachine.contains_many ,transition‘, Transition, ,statemachine‘

     Transition.one_to_many ,source‘, State, ,outTrans‘
     Transition.one_to_many ,target‘, State, ,inTrans‘
end
```

RGen Metamodel DSL
Metamodel Extensions

- Non-generated methods added to Metamodel classes
- E.g. calculating derived data
- Using Ruby‘s „Open Classes“ mechanism
- Focused on a specific aspect of the system (e.g. a code generator)

```
module StatemachineMetamodel
  module State::ClassModule
    def allSubstates
      []
    end
  end
  module CompositeState::ClassModule
    def allSubstates
      subState + subState.allSubstates
    end
  end
end
```

Special „ClassModule“ supporting multiple inheritance

Array extension delegates calls to contained elements and collects results

RGen Metamodel DSL
A look into implementation

- Attribute/reference methods created by Meta-Programming
- ECore model created by Ruby-to-ECore Transformation

```
class MMBase
  class << self

    def has_attr(name)
      _metamodel_description << AttributeDescription.new(name)
      module_eval <<-END
        def #{name}
          @#{name}
        end
      END
    end

    def ecore
      @@transformer ||= ECoreTransformer.new
      @@transformer.trans(self)
    end
  end
end
```

DSL commands are class methods of MMBase

Metainformation required for ECore transformation

Accessor methods created by Meta-Programming

Access to ECore model

Transform class object into an instance of EClass

How to create Models?

- XML/UML instantiators
- Custom instantiators, External DSLs
- Transformers
- Just program:

```
include StatemachineMetamodel

sm = Statemachine.new(:name => “Airconditon”)
off = State.new(:name => „Off“, :statemachine => sm)
on = CompositeState.new(:name => „On“, :subState => [
   heat = State.new(:name => „Heating“),
   cool = State.new(:name => „Cooling“)
])
t1 = Transition.new(:source => [off], :target => [heat])
```

- RGen Modelbuider DSLs

RGen Modelbuilder DSLs Concept

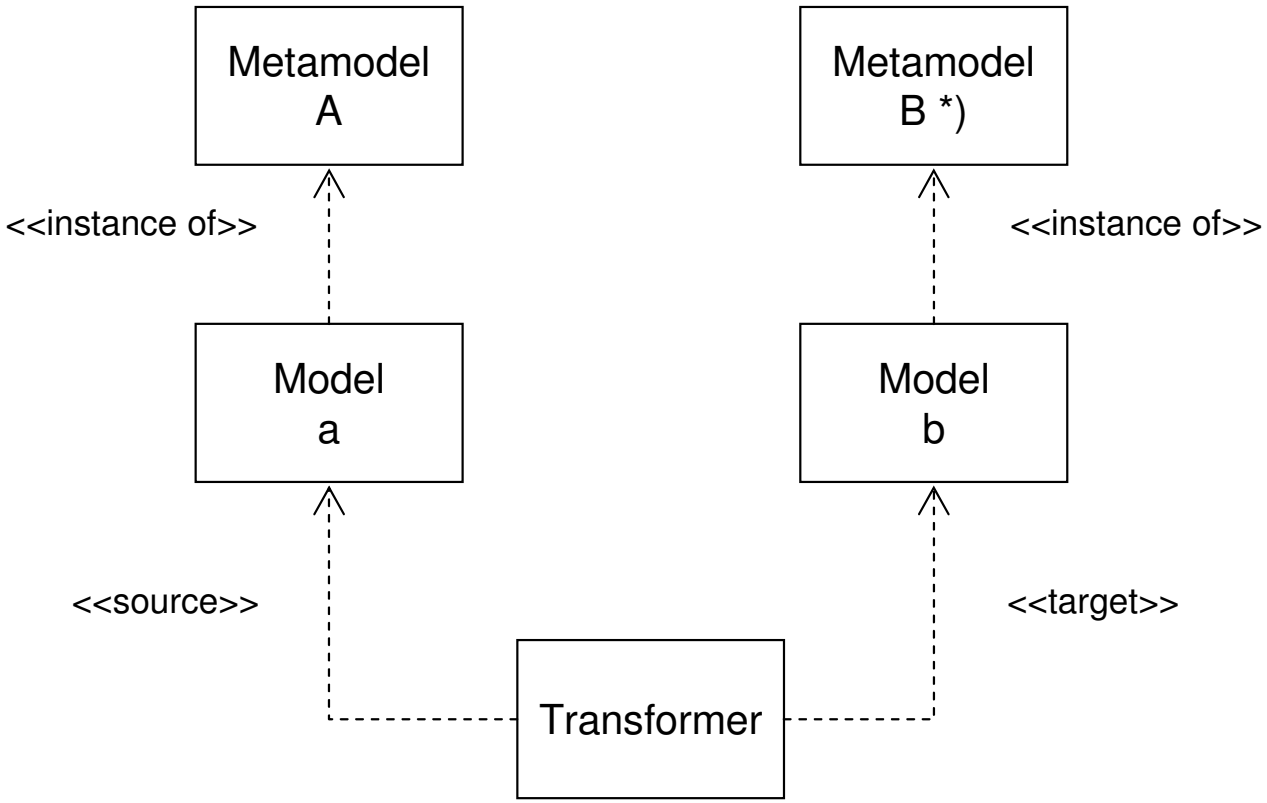
Definition of a Metamodel fully defines the Metamodel‘s (internal) DSL!

- ModelBuilder creates models from DSL representation
- ModelSerializer turns model into its DSL representation

RGen Modelbuilder DSLs
Example: Air Conditioner Statemachine

```
RGen::ModelBuilder.build(StatemachineMetamodel) do
   stateMachine „Aircondition“ do
      state „Off“
      compositeState „On“ do
          state „Heating“
          state „Cooling“
      end
      trigger „PowerButton“
      trigger „ModeButton“
      transition :source => „Off“, :target => „On.Heating“,
         :trigger => „PowerButton“
      transition :source => „On.Heating“, :target => „On.Cooling“,
         :trigger => „ModeButton“
      transition :source => „On.Cooling“, :target => „On.Heating“,
         :trigger => „ModeButton“
      transition :source => „On“, :target => „Off“,
         :trigger => „PowerButton“
   end
end
```

RGen Model Transformation Overview



*) Optionally Metamodel A and B may be the same

RGen Model Transformation DSL Commands

```
class Transformer1 < RGen::Transformer

  def transform
    trans(:class => MM1::Class1)
  end

  transform MM1::Class1, :to => MM2::Class2 do
    # additional code here
    { :targetFeature1 => trans(sourceFeature1),
      :targetFeature2 => trans(sourceFeature2) }
  end

  transform MM1::Class5, :to => MM2::Class6 do
    copy_features :except => [:feature3] do
      { :feature4 => trans(feature3) }
    end
  end

  copy MM1::Class3, :to => MM2::Class4

  copy_all MM1, :to => MM2, :except => [
    „Class1“, „Class2“, „Class3“ ]
end
```

Transformation Start Point „trans“ triggers other rules

Transformation Rule

Target feature assignment hash (result of the given block)

Automatic creation of assignment hash with exceptions and extensions

Complete copy

Copy of all metaclasses with exceptions

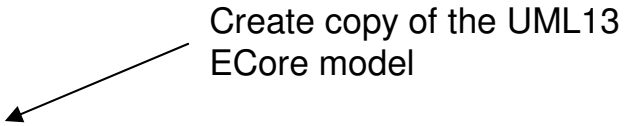
RGen Model Transformation DSL
Example: ECore to UML

```
transform EPackage, :to => UML13::Package do
   {  :name => name,
      :namespace => trans(eSuperPackage) || model,
      :ownedElement =>
        trans(eClassifiers.select{|c| c.is_a?(EClass)} + eSubpackages)
   }
end

transform EClass, :to => UML13::Class do
   {  :name => name,
      :namespace => trans(ePackage),
      :feature => trans(eStructuralFeatures.select { |f|
        f.is_a?(EAttribute) } + eOperations),
      :associationEnd => trans(eStructuralFeatures.select { |f|
        f.is_a?(EReference) } ),
      :generalization => eSuperTypes.collect { |st|
        @env_out.new(UML13::Generalization,
           :parent => trans(st), :namespace => trans(ePackage) || model)
      }
   }
end
```

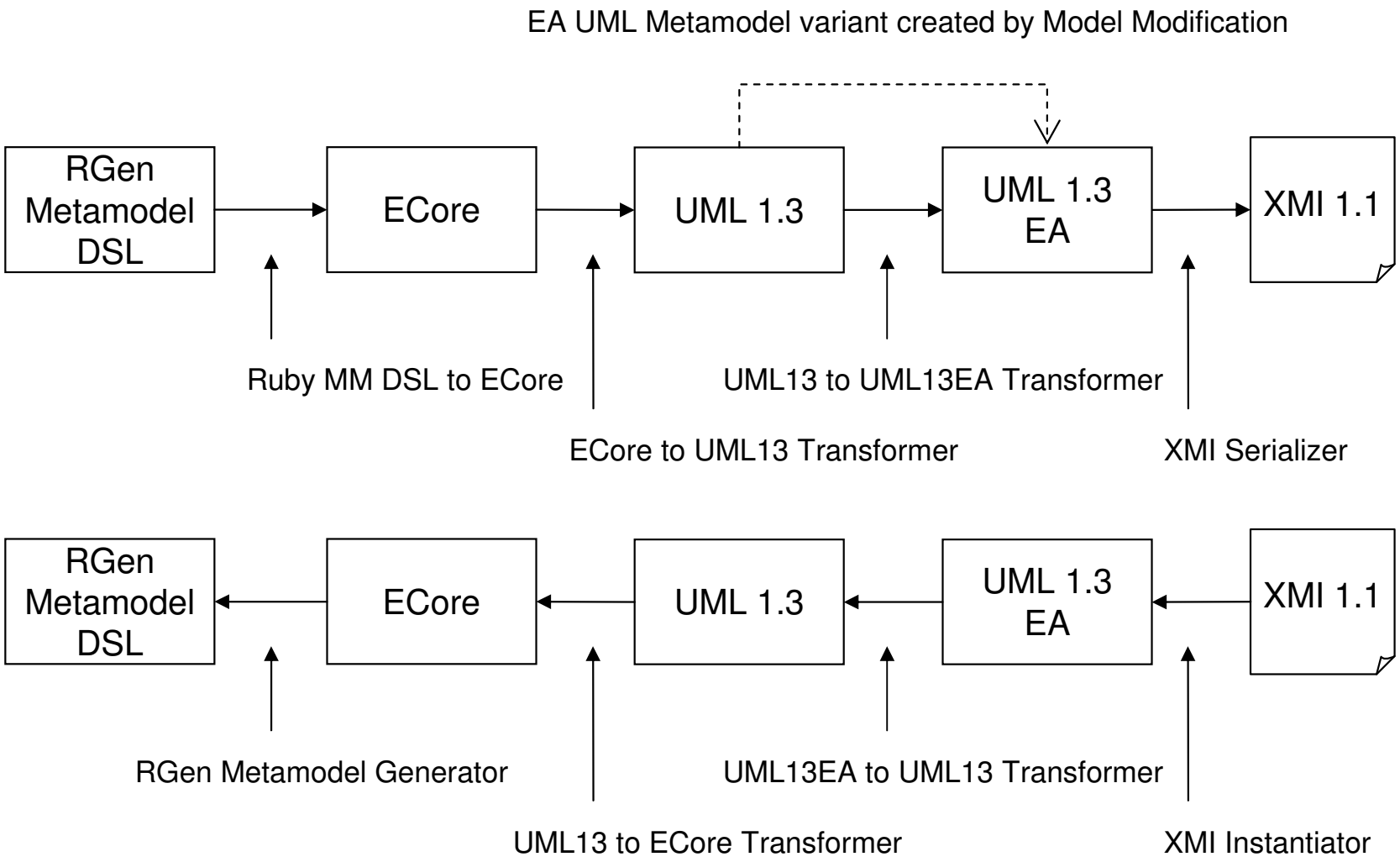
RGen Transformation DSL Example: Copy Transformer

- Automatically created copy rules for all Metaclasses
- Source and Target Metamodel are the same
- Used to copy a model

```
class ECoreCopyTransformer < RGen::Transformer
  copy_all RGen::ECore
end

rootPackage = ECoreCopyTransformer.new.trans(UML13.ecore)

# modify copy to reflect EA specifics
rootPackage.eClassifiers.find { |c|
   c.name == "ActivityGraph" }.name = "ActivityModel"

generateMetamodel(rootPackage, „uml13_ea_metamodel.rb")
```



RGen Transformation DSL
Example: Metamodel Import/Export to EA



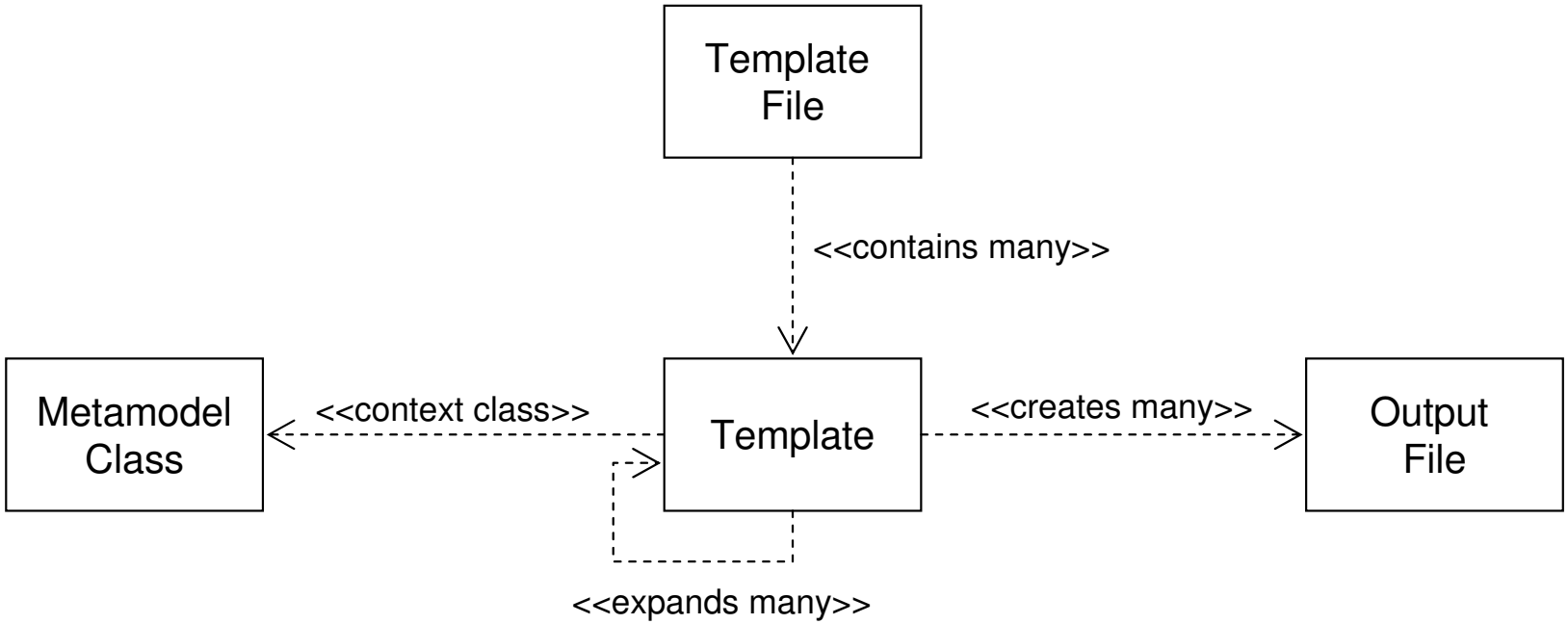
RGen Transformation DSL
A look into implementation

- Internal DSL commands are just Ruby methods
- „Scripting“ DSL commands allows to build new commands

```ruby
def self.copy_all(from, hash={})
   to = hash[:to] || from

   from.ecore.eAllClasses.each do |c|
      copy c.instanceClass, :to => to.const_get(c.name)}
   end

end
```

Example simplified, won‘t work
for nested metamodel packages

RGen Template based Generator Overview



RGen Template DSL Commands

```
<% define ‚Template1‘, :for => Class1 do |a1, a2| %>
  <% file ‚OutputFileA‘ do %>
    Some output text
    <% expand ‚dir1/File2::Template4‘, %>
    <% expand ‚Template2‘, :for => feature1 %>
    <% expand ‚Template2‘, :foreach => feature2 %>
    <% for i in feature3 %>
      Text for every feature3
    <% end %>
    <% if feature2.size > 1 %>
      Only in case we have several feature2
    <% end %>
  <% end %>
<% end %>

<% define ‚Template2‘, :for => Class2A do %>
  Func<%= name %> {<%iinc%>
    <% expand ‚Template3‘ %><%idec%>
  }
<% end %>

<% define ‚Template2‘, :for => Class2B do %>
<% end %>
```

Template definition for Class1 with arguments a1 and a2

Generating into file „OutputFileA“

Actual template text content

Expansion of other templates

Using regular Ruby control flow statements

Explicit indentation commands

Defining „Template2“ for both Class2A and Class2B

RGen Template DSL
Example: Statemachine Code Generator

```
<% define 'Root', :for => Statemachine do %>
   <% file name+".c" do %>
   void trigger(TriggerId trigger)
   {<%iinc%>
      switch(currentState)
      {<%iinc%>
         <% expand 'Case', :foreach => (state + state.allSubStates).
            reject{|s| s.is_a?(CompositeState)} %><%idec%>
      }<%idec%>
   }
   <% end %>
<% end %>

<% define 'Case', :for => State do %>
   case S_<%= qualifiedName %>:<%iinc%>
      <% for t in allOutTrans %>
         if (trigger == T_<%= t.trigger.name %>) {<%iinc%>
            currentState = S_<%= t.target.name %>;<%idec%>
         }
         else <%nows%>
      <% end %> {}
      break;<%idec%>
<% end %>
```

RGen Template DSL
Example: Statemachine Code Generator

- Output generated from Air Conditioner example model:

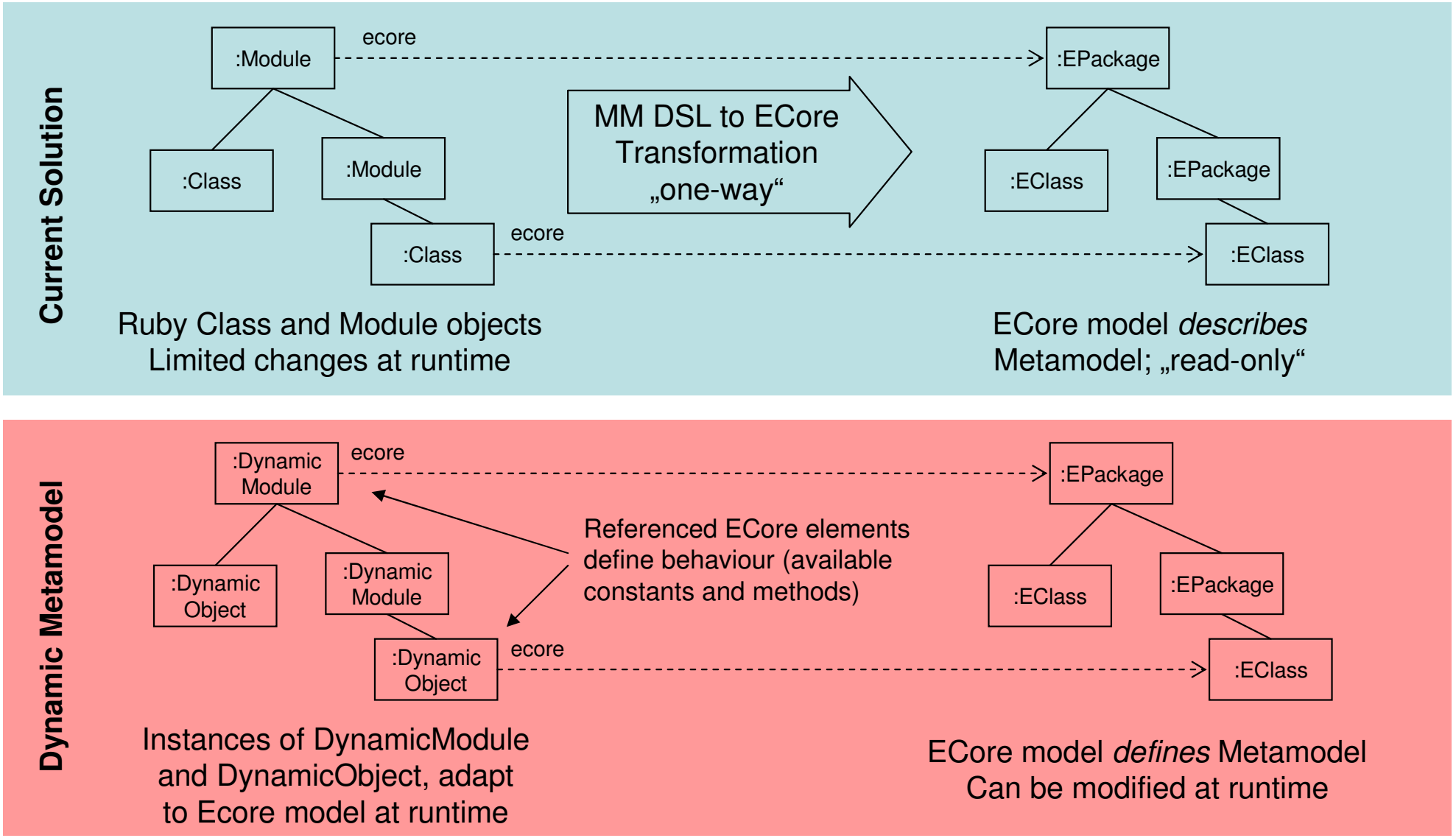
```
void trigger(TriggerId trigger)
{
   switch(currentState)
   {
      case S_Off:
         if (trigger == T_PowerButton) { currentState = S_Heating; }
         else {}
         break;
      case S_On_Heating:
         if (trigger == T_PowerButton) { currentState = S_Off; }
         else if (trigger == T_ModeButton) { currentState = S_Cooling; }
         else {}
         break;
      case S_On_Cooling:
         if (trigger == T_PowerButton) { currentState = S_Off; }
         else if (trigger == T_ModeButton) { currentState = S_Heating; }
         else {}
         break;
   }
}
```

RGen Template DSL
A look into implementation

- Ruby ERB mechanism parses ERB files and creates Ruby code
- Ruby code between <%..%> is copied verbatim
- Text outside is transformed into Ruby code `@output = „text“`
- `define` stores templates a Ruby Proc objects
- `expand` executes the Proc object and produces its output

```
<% define “TemplateA”, :for => SomeClass do %>
   Text
   <% expand “TemplateB” %>          ⎫ Block
   Text                              ⎭
<% end %>

<% define “TemplateB”, :for => SomeClass do %>
   Text                              ⎫ Block
<% end %>                            ⎭
```

Outlook
Dynamic Metamodel



RGen
Summary

- Covers most Modelling and Code Generation aspects
- BUT: No IDE support

- Powerful and flexible
- BUT: Use with care!

- USE IT:
  - In non-Eclipse projects
  - In everyday scripts
  - For experiments
  - To have fun