Module API

The uniform API of the Modules of a COCoMaS

2004 Hans-Werner Sehring

Interfaces Making up a Module

For each asset class A the following interfaces are created:

- interfaces for the states/roles of an asset object,
- an iterator interface for collections of asset objects,
- a factory interface to create asset objects,
- an interface for query classes,
- a visitor interface to distinguish between subtypes, and
- a visitor interface for asset object states.

Additionally, the following classes are created:

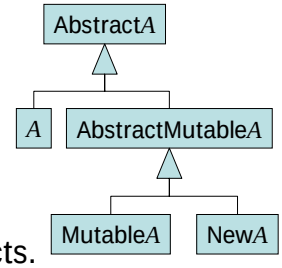
- constraint objects, and
- a metaclass A`Class` reflecting the asset definition.

Asset Roles

Roles are reflected in interfaces of assets objects.

There are five such interfaces for a class A, namely

- `Abstract`A,
- A,
- `AbstractMutable`A,
- `Mutable`A, and
- `New`A



for the possible states/roles of asset objects.

Note: Generators should not rely on this naming scheme. Instead, they have to use the symbol table created by the API generator (see compiler framework and generator development guide).

Asset Roles – General Base Interface

There is one abstract base interface factoring out all common methods available in all roles: `Abstract`A. If A is a top level class, `Abstract`A is derived from the generic interface `de.tuhh.sts.felida.….AbstractAsset`. If A is a subclass of a class B, Abstract$A$ is derived from Abstract$B$.

It contains:

- One `get…()` method for each content object, characteristics, and one-to-one relationship attribute, and
- one `has…()` method for each one-to-many relationship attribute.
- A method `getType()` to access an asset's metadata (see below, class `AssetClass`).
- `accept()` methods for subtype and state visitors as introduced below.
- A method `getId()` return an `ID` object as explained below.

Asset Roles – Persistent State

An asset object for an asset class *A* in persistent state fulfills an interface *A*. This interface defines just one method `lock()` to transfer the asset object to locked state, granting exclusive access.

The interface *A* is derived from the generic interface `Asset` (or an interface *B*) and the generated interface `Abstract`*A*.

In Java:

```
public interface A
  extends AbstractA, Asset
{
  MutableA lock () ;
}// interface A
```


Asset Roles – Base Interface for Mutable States

For all mutable states there is one abstract base interface `AbstractMutable`*A* which is derived from `Abstract`*A*. It defines:

- a `set…()` method for each content object, characteristics, and one-to-one relationship attribute, and
- both an `add…()` and a `remove…()` method for each one-to-many relationship attribute.

Methods for relationships have locked assets as parameters:

- For an asset class
  ```
  class A refines B {
    concept relationship r : C*
  }
  ```
  a Java interface
  ```
  public interface AbstractMutableA
    extends AbstractA, AbstractMutableB
  {
    void addR (MutableC r) ;
    void removeR (MutableC r) ;
  }// interface AbstractMutableA
  ```
  is created.

Asset Roles – Locked State

If locked an asset object fulfills the interface `Mutable`*A* which extends `AbstractMutable`*A* and the generic interface `MutableAsset`. It defines methods to change state:

- a method `commit()` to make changes persistent; afterwards the asset object is in persistent state,
- a method `abort()` to discard changes; afterwards the asset object is in persistent state, and
- a method `delete()` to transfer the asset object to volatile state.

In Java:

```
public interface MutableA
  extends AbstractMutableA, MutableB
{
  A abort () ;
  A commit () ;
  NewA delete () ;
}// interface MutableA
```


Asset Roles – Volatile State

A newly created asset object is in volatile state, indicated by the interfaces `New`*A*, subtype of `AbstractMutable`*A* and `NewAsset` (or `New`*B*). This interface defines one method `store()` to transfer an asset object to persistent state.

In Java:

```
public interface NewA
  extends AbstractMutableA, NewB
{
  A store () ;
}// interface NewA
```

As long as it is in volatile state the asset object is not persistent. Instead, it resides in main memory only.

Not all operations can be used in conjunction with volatile objects. E.g., they can not be related to other objects (see `add…()` methods).

Asset Roles – Method Exceptions

Each of the methods in the state interfaces can be declared to throw several exceptions:

- A `ServerException` (package `de.tuhh.sts.felida…`) is raised if the execution of a method fails for technical reasons.
- A `StateException` (package `de.tuhh.sts.felida.…`) is thrown when methods are called from an interface that does not match an object's state.
- A `VetoException` (from package `java.…`) may be thrown by methods generated for attributes with a defined constraint.

Asset Iterators

Asset iterators iterate over sets of asset objects like

- directly created sets,
- assets bound to a one-to-many relationship, and
- set-valued results of operations.

For an asset class A which is a refinement of a class B an iterator interface

```
public interface AIterator
  extends BIterator
{
  A nextA () ;
}// interface AIterator
```

is created. If no superclass is given explicitly the interface `AssetIterator` is extended instead.

Asset Object Factories

For asset class definitions

```
class A { … }
class B refines A { … }
class C refines A { … }
```

a factory interface

```
public interface AFactory {
  B createB () ;
  C createC () ;
}// interface AFactory
```

is created.

For classes without an explicitly stated base class factory methods are added to the interface `AssetFactory`.

Factory method implementations are responsible for the initialization according to the asset definition.

Asset Query Objects

The retrieval of asset objects is done through query objects.
For an asset class *A* an interface *A*Query is created with the following methods:

- Constrain methods:
  - For each characteristic *c* there are methods ???
    - constrain*C*Equal()
    - constrain*C*Less()
    - constrain*C*LessOrEqual()
    - constrain*C*Greater()
    - constrain*C*GreaterOrEqual()
    - constrain*C*NotEqual()
    - constrain*C*Similar()
  - For each relationship *r* there are methods
    - constrain*R*Referenced()
    - constrain*R*References()
- Execute methods: for asset classes without subclasses a method `execute()` which returns an *A*`Iterator` is added. If there are subclasses, the method `executeFor`*A*`()` return an *A*`Iterator` is added instead.

Furthermore, the interface *A*`QueryFactory` describes a factory to create such query objects through a method `start`*A*`Query()`.

Visitor Interfaces to Distinguish between Subclasses

For asset class definitions

```
class A { … }
class B refines A { … }
class C refines A { … }
```

a visitor interface

```
public interface AVisitor {
  Object visit (B b) ;
  Object visit (C c) ;
}// interface AVisitor
```

is created.

The interface Abstract*A* generated for *A* defines a method `accept(AVisitor)` which has to be implemented in classes for *B* and *C*. For classes with no explicit base class a `visit()` method is added to the generic interface `de.tuhh.sts.felida.….AssetVisitor`, and the `accept()` method is defined accordingly.

Generic Interfaces and Classes

The set of generated interfaces is amended by a set of generic interfaces and classes:

- the role interfaces `Asset`, `MutableAsset`, and `NewAsset`,
- the abstract base class `Asset`,
- the abstract class `AssetClass`,
- a base iterator `AssetIterator`,
- a basic object factory `AssetFactory`,
- a basic query interface `AssetQuery`, and
- an interface for visitors which distinguish between objects' states.

Identifiers

Artificial identifiers for asset objects

method getId() return an ID defined as

```
public interface ID {
  String getKey () ; // implementation-specific
  String getComponentName () ; // for routing
  static ID fromString () {
    final String key = ... ;
    final String component = ... ;
    return new ID () {
      public String getName () { return name ; }
      public String getComponent () { return component ; }
    } ;
  }// fromString
}// interface ID
```


Interface Asset

...

Abstract class AssetClass (1)

The base class `AssetClass` for all asset meta classes is defined as an asset class itself. A generic base implementation is given. Some methods need to be implemented by concrete sub classes:

```
public abstract class AssetClass extends Asset {
  static private Map instances ;
  private String name ;
  private AssetClass superClass ;
  protected AssetClass (String clsName, String superClsName) {
    this.name = clsName ;
    instances.put (clsName, this) ;
    if (superClsName != null)
      superClass = (AssetClass)instances.get (superClsName) ;
  }// constructor
  abstract public NewAsset createInstance () ;
  abstract public AbstractAsset getInstance (ID Id) ;
  public String getName () { return name ; }
  public boolean hasInstance (Asset a) { … }
  public boolean hasSubClass (AssetClass c) { … }
  public boolean hasSuperClass (AssetClass c) { … }
  abstract public AssetQuery startQuery () ;
}// abstract class AssetClass
```


Abstract class AssetClass (2)

The method `getInstance(ID)` is implemented depending on the target module. E.g., for a relational database it might be:

```
public AbstractAsset getInstance (ID id) {
  ResultSet rs =
    … ("SELECT * FROM x WHERE id_ = " + id.getKey () ;
  if (rs.next ())
    construct asset a object from data
    return a ;
  else {
    for each subclass s {
      AbstractAsset a = s.getInstance (id) ;
      if (a != null)
        return a ;
    }
    return null ;
  }
}// getInstance
```

Visitor Interfaces to Analyze an Asset Object's State

To distinguish between object states the visitor interface

```
public interface StateVisitor {
  Object visit (Asset a) ;
  Object visit (MutableAsset a) ;
  Object visit (NewAsset a) ;
}// interface StateVisitor
```

from package `de.tuhh.sts.felida.…` is used.

Generic Classes for Event Handling

work in progress

The following interfaces, classes, and methods make up the event handling:

- Modification of assets:
  - AssetModificationEvent(AbstractAsset, AttributeAssignment [])
  - AssetModificationListener#modificationPreparation/modification
  - Asset#addAssetModificationListener()
  - Asset#removeAssetModificationListener()
  - Asset#notifyListenersPrepare(AssetEvent) throws VetoException
  - Asset#notifyListenersDone(AssetEvent)
- Creation of assets:
  - AssetCreationEvent(NewAsset)
  - AssetCreationListener#assetCreated(AssetCreationEvent)
  - AssetFactory#addAssetCreationListener(AssetCreationListener)
  - AssetFactory#removeAssetCreationListener(AssetCreationListener)
- State changes:
  - AssetStateChangeEvent(AbstractAsset)
  - AssetStateChangeListener#stateChanged(AssetStateChangeEvent)
  - Asset#addStateChangeListener(AssetStateChangeListener)
  - Asset#removeStateChangeListener(AssetStateChangeListener)

Creation of Asset Implementation Classes: Constraints

work in progress

Methods implement constraints (setter may throw `VetoExceptions`) and rules (on `VetoExceptions` code may be executed).

```
class A1 refines A2
[
  ...
|
  characteristic a1 : T1 = a3.b
  ...
  constraint b
    onviolation s
]
```

let A_3 define characteristics b

A_1Impl

…
getA$_1$ () : BT$_1$
setA$_1$ (a_1 : BT$_1$) : void
…

```
setA1 (a1 : BT1) : void {
  try {
    a1Constraint.check () ;
  }
  catch (VetoException exc) {
    a1Constraint.fixup () ;
    a1Constraint.check () ;
  }
  this.a1 = a1 ; // or similar
}
```

```
setA1 (a1 : BT1) : void {
  a1Constraint.evaluate () ;
  this.a1 = a1 ;
}
```


Creation of Asset Implementation Classes: Constraint Checking

work in progress

Checking constraints in setters has to be done using a 2-phase commit protocol:

```
setA1 (a1 : BT1) : void {
  // check constraints on this class
  … ;
  // recursively applies rules
  // may raise a VetoException
  a1Constraint.prepareSetA1 (a1) ;
  // actually carry out changes
  … ;
  // notify listeners about changes
  AssetModificationEvent evt =
    new AssetModificationEvent (…) ;
  notifyListeners (evt) ;
}// setA1
```

Creation of Asset Implementation Classes: Constraint Checking

work in progress

Checking constraints in setters has to be done using a 2-phase commit protocol:

```
setA1 (a1 : BT1) : void {
  AssetModificationEvent evt =
    new AssetModificationEvent () ;
  // check constraints on this class
  … ;
  // recursively applies rules
  // may raise a VetoException
  notifyListenersPrepare (evt) ;
  // actually carry out changes
  … ;
  // notify listeners about changes carried out
  notifyListenersDone (evt) ;
}// setA1
```


Creation of Asset Implementation Classes: Rules

work in progress

Methods implement constraints (setter may throw `VetoExceptions`) and rules (on `VetoExceptions` code may be executed).

```
class A1 refines A2
[
  ...
|
  characteristic a1 : T1 = a3.a4
  ...
  constraint b
    onviolation s
]
```

let A_3 define charakteristics a_4

A_1Impl

…
getA$_1$ () : BT$_1$
setA$_1$ (t : BT$_1$) : void
…

```
setA1 (a : BT1) : void {
  AssetModificationEvent evt =
    new AssetModificationEvent (…) ;
  a1Constraint.check (evt) ;
  … set a1 to a in local buffer ;
  notifyAssetModificationListeners (evt) ;
}
```

Abstract Base Class of Constraint Objects

work in progress

Constraints classes are implementations of the abstract base class:

```
public interface Constraint {
  public boolean evaluate (Asset assetToCheck) {
    if (holds (assetToCheck))
      return true ;
    else {
      Asset [] changedAssets = violationFixup (assetToCheck) ;
      while (one contributingAssets () in changedAssets)  {
        if (holds (assetToCheck))
          return true ;
        else
          changedAssets = violationFixup () ; // oder false??
      }// while
      return false ;
    }
  }// evaluate
  abstract public Asset [] contributingAssets () ;
  abstract public Asset [] violationFixup () ;
  abstract public boolean holds (Asset assetToCheck) ;
}// interface Constraint
```


Asset Metaclasses

For a class definition

```
  class A refines B {
    …
  }; class A
```

a Java class is created whose `getType()` method returns an implementation of the generic abstract class `de.tuhh.sts.felida.….AssetClass` (generated from a corresponding definition in model “MetaModel”) is created:

```
  new AssetClass () {
    { super ("A", "B") ; } ???
    public NewA newA () {
      return BFactory.getInstance ().createA();
    }
    public NewAsset createInstance () {
      return newA () ;
    }
    public AQuery startAQuery () {
      return AQueryFactory.getInstance ().startAQuery () ;
    }
    public AssetQuery startQuery () {
      return startAQuery () ;
    }
    …
  }// AssetClass
```