Asset Compiler Framework

Asset Compiler Framework and Generator
Development Guide

Table of Contents

Architecture of the Asset Compiler

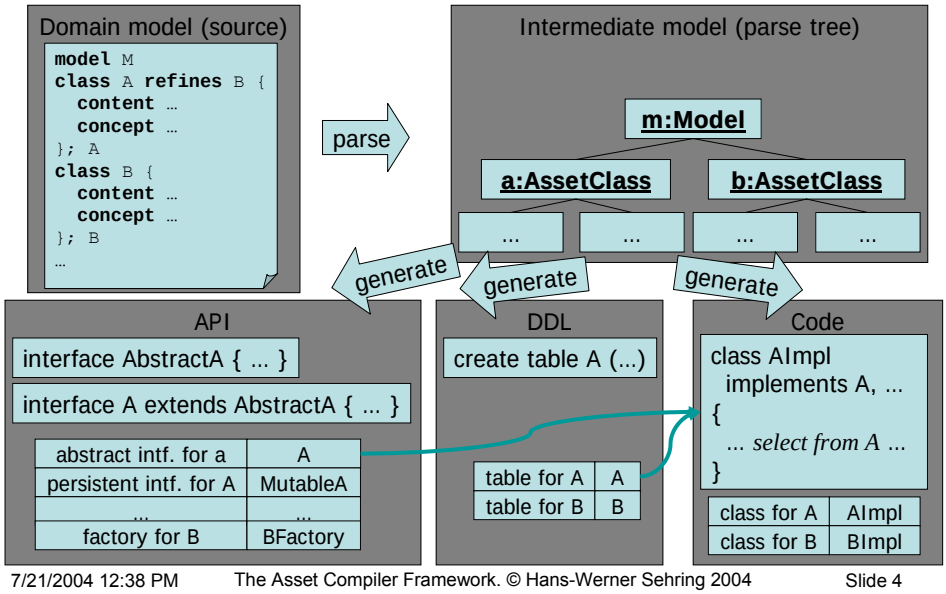
The basic structure follows the classical compiler architecture consisting of a frontend and a backend which communicate by exchanging some intermediate model.

- Frontend
  - Lexing and parsing the asset definitions
  - Creating and checking the intermediate model
- Backend
  - API generator
  - Module generators

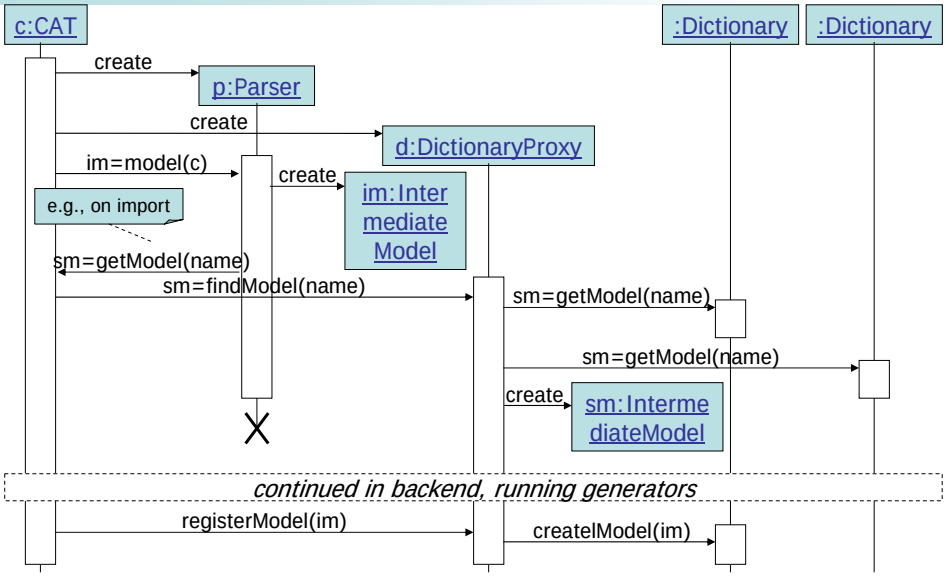
The compiler is designed as a framework. It can be extended by module generators.

The compiler itself controls the order in which generators are run and the data flow between them.

Overview over the Compilation Process



Compilation Process



Intermediate Model

The parser produces an internal representation of the asset model to be translated in the form of an `IntermediateModel`. It contains:

- `AssetClass`: an asset class (name, superclass, members)
- `Characteristic`: a characteristic attribute
- `Relationship`: a relationship attribute
- `Constraint`: a constraint, eventually with an associated rule

... continue description ...

Generator Interface (1)

The compiler backend incorporates an API generator and module generators. They create code for given asset definitions in a programming language (or a DDL or similar). While operating generators produce files as “side effects”.

Module generators create implementations based on the API and the generic components created by the API generator and the definitions done by DDL generators (and similar).

To add module generators to the compiler they need to implement the `Generator` interface from package `de.tuhh.sts.felida.cat.gen`.

Data is passed between generators using symbol tables. Each generator may read any number of symbol tables and produces exactly one new symbol table. The type of symbol table depends on the generator. Each symbol table is derived from class `de.tuhh.sts.felida.cat.SymbolTable`.

Generator Interface (2)

Callback methods (for template method):

- **`static`** `SymbolTableDescription [] getRequestedSymbolTables ()`
  Answers the types and names of symbol tables needed by this generator. From this dependency information a possible sequence of generator runs is computed. Return values are of type
  ```
  public interface SymbolTableDescription {
    String getName () ;
    Class getType () ;
  }// interface SymbolTableDescription
  ```
- **`static`** `SymbolTableDescription getProducedSymbolTable()`
  Answers the type and name of the symbol table which will be produced on `generate()`.

Generator Interface (3)

Callback methods (for template method) continued:

- `static ParameterDescription [] getRequestedParameters ()`
  with return values of type

```
public interface ParameterDescription {
  String getName () ;
  Class getType () ;
  String getDescriptionText () ;
}// interface ParameterDescription
```

  answers the parameters needed by the current generator. The have to be provided in the `generate()` call.

Generator Interface (4)

Callback methods (for template method) continued:

- `SymbolTable generate (IntermediateModel intermediateModel, SymbolTable [] symbolTables, Map parameters) throws GeneratorException`
  The actual performance method. Gets the intermediate model and the requested symbol tables as parameters. Returns the symbol table created by this generator. As a “side effect”, files are created etc.

Basically, these methods have to be implemented when developing a generator.

Generator Interface (5)

Additional methods have to do with progress indication. It is used to show the completion state of a compiler run in interactive development environments.

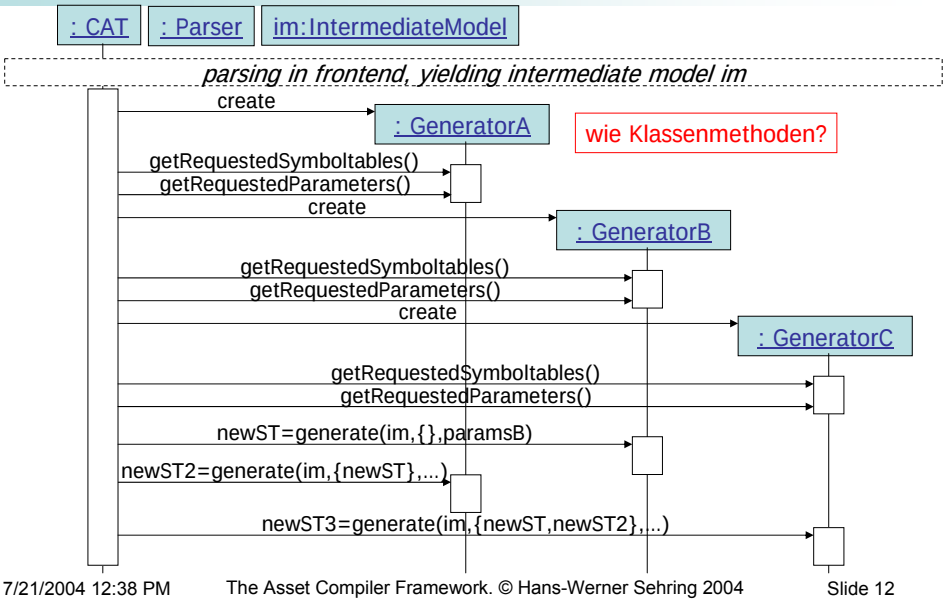
For this, there are to further methods defined in `Generator`:

- `Generator#addProgressListener(ProgressListener)`
- `Generator#removeProgressListener(ProgressListener)`

These methods are used to register observers (implementations of the interface `ProgressListener`) which are supplied with `ProgressEvent`s.

If progress has been made, `Generator`s inform their listeners by calling `ProgressListener#changedState(ProgressEvent)`.

Coordination of the Module Generators



API Generator

Usually, the first generator run is the API generator.

It produces Java interfaces for the unique interface all modules have to fulfill.

For the API see the respective documentation.

The Symbol Table Produced by the API Generator

The symbol table produced by the API generator, `APIGenerator$APISymbolTable`, has the following methods to access information:

- the role interfaces for asset objects (see API documentation)
  - `JavaInterface getAbstractInterface(AssetClass)`
    returns the base interface for all states
  - `JavaInterface getVolatileInterface(AssetClass)`
    returns the volatile interface
  - `JavaInterface getAbstractMutableInterface(AssetClass)`
    returns the base interface for all mutable states
  - `JavaInterface getLockedInterface(AssetClass)`
    returns the mutable interface
  - `JavaInterface getPersistentInterface(AssetClass)`
    returns the persistent interface
- `JavaInterface getFactoryInterface(AssetClass)`
  returns a factory object to produce new instances of the named class
- `JavaInterface getIteratorInterface(AssetClass)`
  returns the interface of iterators for sets of assets of the named class
- `JavaInterface getQueryInterface(AssetClass)`
  returns the interface of query classes for the named class and its subclasses
- `JavaInterface getTypeVisitor(AssetClass)`
  returns the visitor for the subtypes of the named class

The class `JavaInterface` is part of the code generation framework.

Module Kinds

A conceptual content management system is made up of module instances of the five module kinds:

- I-modules
- A-modules
- C-modules
- D-modules
- S-modules

For details see the architecture documentation.

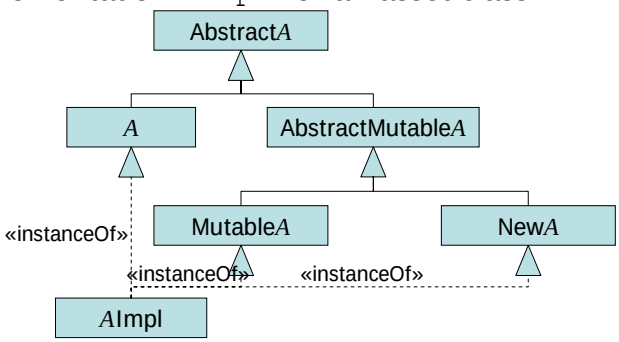
Module Implementations

Implementations for each module kind are classes implementing the interfaces from the API such that the required functionality is offered.

- I-modules: access to a base system; transactional behavior on the level of methods
- A-modules: delegation to mediate between two APIs
- C-modules: delegation to other modules according to defined behavior
- D-modules: stubs and skeletons
- S-modules: mapping standardized requests to API calls; transactional behavior on the level of asset manipulation language operations

Asset Objects

E.g., an implementation *A*`Impl` for an asset class *A*:



A`Impl` is able to behave according to each of the roles.

It implements all methods from the interfaces.

It has to check state on methods calls.

Module Generators

For each implementation of a module of one kind there is a (combination of) generator(s).

Examples:

- I-Module for relational databases:
  - generator for DDL statements in SQL
  - generator for access code in Java using JDBC
- S-Module for Web Services:
  - generator for WSDL interface descriptions
  - XML schema generator for data exchange format(?)
  - generator for Java stubs and skeletons
- D-Module for XML document exchange
  - generator for XML schema
  - generator for web server and client

Generator Obligations

As for every I-module generator, the following classes have to be created for each asset class definition:

- an implementation class (implements interfaces for concrete, locked, and volatile assets),
- a factory class which implements the factory interface so that initial values are set and cascading creates are realized,
- a query class with constrain methods for each content, characteristic, and relationship attribute, and
- an asset iterator.

Generation Obligations: Event Handling

work in progress

In I- and A-modules?

Alternatives:

- constraint checking in setter methods
- constraint objects as observers

Constraint checking in setter methods not always possible:

- ```
  class A { concept characteristic x : T }
  class B {
    concept relationship a : A
            constraint a.x = …
  }
  ```
- Constraint of `B` has to be checked on `AImpl#setX()`.

Constraint objects as observers have to implement a two-phase commit protocol to be able to check constraints before changes are made persistent (in current object and in others because of rules).
  ```

Generation Obligations: Event Handling

work in progress

B#setA:

- check own constraints
- notifyListenersPrepare
- a.removeAssetModificationListener (this)
- change binding for a
- a.addAssetModificationListener (this)
- notifyListenersDone

{a.r} = (lookfor A { ... }).r
has to be checked on setX() on a and all assets found by query
⇒ checking set of listeners depends on some predicate
⇒ have to determine objects to be notified dynamically
→ commands may not be carried out: a.r=(modify b {x:=y}).r

Generator Obligations – Generated constraint checking

work in progress

A.set$X$(...):

- change attribute value/binding
- c := { self }
- check constraints; fills set c of assets to be stored/committed
- on VetoException:
  - delete()/abort() all a ∈ c
  - change attribute back
  - re-raise VetoException
- store()/commit() all a ∈ c

Generator Obligations – Generated constraint checking

work in progress

Constraint#check(Set c):

- check condition;
  if it includes commands, to not commit affected assets (store()/commit()), but add them to c
- if condition is not meet:
  - execute handler without committing affected assets; instead, add these to c
  - re-check condition (what to do with commands this time?)
  - if condition is not met, throw VetoException