Smart0.3の実装

佐藤真紗美

平成15年11月9日

概 要

この文章は、Smart0.3の実装について、議論の帰結として得たものをまとめたものである。

なお、この文章の内容は議論によってたびたび改変される。

1 RuntimeInstance

Fig1に、Smart0.3のRuntimeInstance周辺のクラス図を示した。

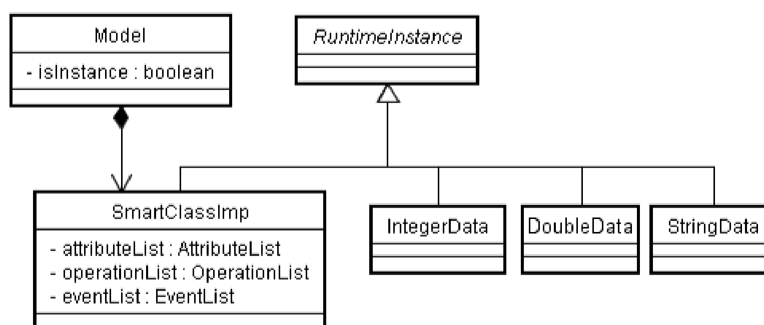

Fig 1: RuntimeInstance

次に、Fig2に概念図を示す。

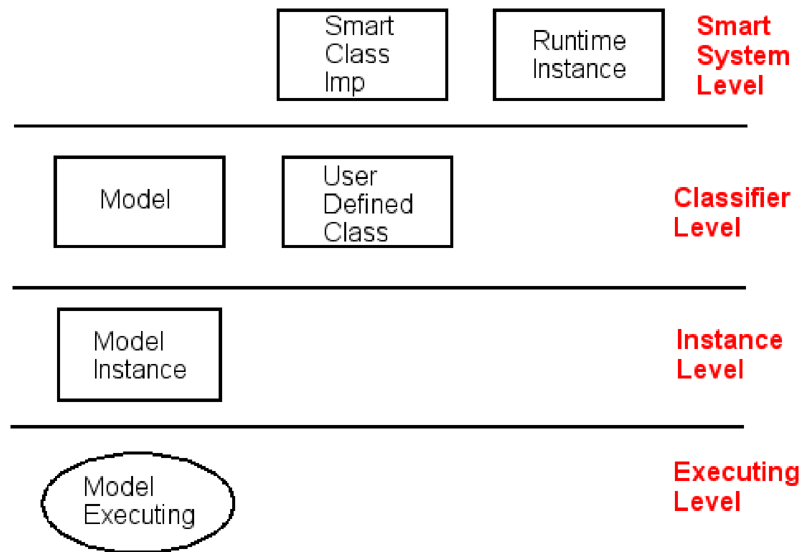

Fig 2: Concept

1.1 RuntimeInstance

RuntimeInstance は、runtime 時に必要な data についての情報を持つ abstract class in Java である。したがって、RuntimeInstance 自身はインスタンスにならない。

RuntimeInstance のサブクラスのインスタンスは、Action Semantics によって action を実行するときの各インスタンス in smart の data を表現する。つまり、RuntimeInstance は、UML メタモデルとは関係なく、Action Semantics 実装のために用意された実装用クラスである。

Changes from Smart0.2 RuntimeInstance 本体について、その実装が大きく変わったところはないが、Smart0.2 であやふやだった意味論が、はっきりと「Action Semantics 実装用のクラス」として定義された。

1.2 基本型

Smart0.2 同様、基本型 (PrimitiveDataType) として、次の 3 つを定義する。

IntegerData

DoubleData

StringData

この基本型は、それぞれ RuntimeInstance の直接のサブクラスとして定義する。ただし、**この基本型は、UML2.0 のメタモデルの PrimitiveDataType とは関係がない**。この基本型の目的は、前項で説明した、スーパークラスである RuntimeInstance と同様に、Action Semantics の実装用である。

1.3 SmartClassImp

SmartClassImp は、RuntimeInstance の直接のサブクラスであり、Smart でユーザがクラスを定義するための Smart メタクラスである。すべてのユーザ定義クラスは、SmartClassImp のインスタンスとして実装される。(これはちょうど、ArgoUML などでクラス図に新しいクラスを生成する際、そのクラスの論理モデルとして UML メタクラスの Class のインスタンスを持たせるのと同様である。)

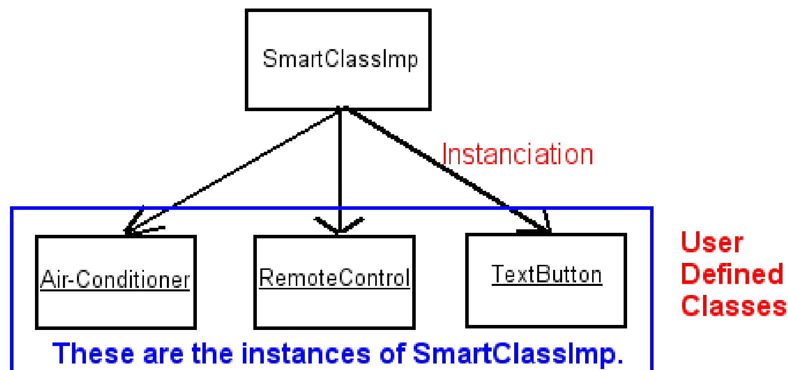

Fig 3: SmartClassImp and the Instances

SmartClassImp は、Fig1 に示したように、AttributeList、OperationList、EventList を持つ。AttributeList は、ユーザ定義クラスの持つ属性のリストであり、OperationList は、ユーザ定義クラスの持つ操作のリスト、EventList はユーザ定義クラス上で起こりうる Event の種類を列挙するリストである。

なお、ユーザ定義クラスのインスタンスの生成の説明は 2.2 に行う。

SmartClassImp は、最初 SmartObject と呼ばれていたが、実際は Object in Smart ではなく Class in Smart の実装 (Implementation) であるので、SmartClassImp に改称した。

Changes from Smart0.2　SmartClassImp は、Smart0.2 の RootOfModel のうち、AttributeList の部分のみを切り出したクラスである。逆に言えば、RootOfModel が State や Transition のリストを持っていた点は Model に渡し、また、UI の Model として使われることのないようにした。RuntimeInstance がスーパークラスである点に変わりはない。

疑問　UML メタモデルの Classifier との関係は?

疑問　EventList ではなく、TriggerList とするべきか?

2　Model

2.1　Model

Model とは、Classifier の「表現」方法の一種である。つまり、ある Classifier の中身を、わかりやすくユーザに伝えるための手段である。

Model は、自身を表す Classifier として SmartClassImp のインスタンスを持つ。たとえば、Model Air-Conditioner は、SmartClassImp Air-Conditioner を、Model が表現している target classifier として持っている。

```
class Model{
    SmartClassImp classifier;
    ...
}
```

また、Model は、この targer classifier を表す次の 4 点を持っている。

Composite Structure

User Interface (UI)

State Machine

Interaction (as Sequence Diagram)

Use Case

この点から、この target classifier は、“Structured and behaviored classifier with User Interface” と位置づけることができる。

Model は抽象的なものであるので、target classifier(SmartClassImp のインスタンス) の AttributeList の attribute に仮に具体的な値が設定されたとしても、それを無視する。たとえば、「温度」という attribute に対して「23」というような具体的な値が入力されても無視する。

また、Model は、Instance 化されるためのコンストラクタを持つころができる。このコンストラクタは SAL テキストで書かれ、具体的には Model はこの SAL テキストの書かれたファイルへの参照を持っている。コンストラク

タを持たない場合、Model が Insntance 化されるときは最も簡便なコンストラクタを自動的に呼び出すが、それについては次項で述べる。

疑問　「コンストラクタ」でよいか?

2.2 Model Instance

Model を Instantiate したものが Model Instance である。Model Instance は、Model の target classifier の AttributeList(以下「Model の AttributeList」と省略する) に、属性として用いられることがある。

たとえば、"Phone" という Model について、Phone の AttributeList には、"Button" という Model の Instance、b1,b2,...b10 が存在する。

Type	Name	Initialization
Button	b1	num:=1;
Button	b2	num:=2;
Button	b10	num:=3;

Table 1: AttributeList of Model "Button"

2.2.1 Instanciation

Model が Instanciate されるとき、Model は SAL テキストで書かれたコンストラクタを呼び出す。そのとき、コンストラクタでこの Model Instance にセットされる値を Table1 の最右列のように書くことができる。

たとえば、Table1 では、Model"Button" は次のような属性を持つと仮定している。

```
Integer num;
```

このとき、Table1 の最右列のように

```
num:=1;
```

というような Initialization Parameter が書かれている場合、この表を次のような SAL テキストに変換する。

```
b1:=CreateObject Button;
b1.num:=1;
```

これを SAL が実行する。ただし、SAL の CreateObject 自体の命令の中身は、Modeler に実装する CreateObject を呼び出すことにある。

仮に、Table1 に入力された Initialization Parameter が

```
num>0;
```

であった場合は、SAL の実行時にエラーが発生する。

疑問　Model"Phone" に b1 という attribute が存在しない場合には、そのエラーは Modeler の createObject() で起きる?

2.2.2　CreateObject

CreateObject は実際には Modeler に実装する。その中身は、Model Instance の基になる Model を完全にコピーして、

```
isInstance=true;
```

とすることである。ModelInsntance というクラスは存在せず、Model の isInstance という boolean を変更するだけで、それが Model であるか Model Instance であるかの変更を行う。したがって、実装を考えれば、Model は Model Instance の prototype である。

注意　ここで、isInstance の true/false だけで Model ←→ Model Instance の変換が行われることで、実は Instance から Classifier への変換を簡単に行うことができる。つまり、Instance で擬似的に行った何らかの変更を Classifier にも反映させたいと考えるとき、isInstance を false とするだけでそれが可能になる。

2.2.3　Instance Pool

Model はそれぞれ Instance Pool を持つ。Model が Instanciate されるとき、Model はその Model Instance を Instance Pool にプールしていく。

2.3　Model Execution

Action Executor によって Model Instance が実行されているとき、この「実行されている Model Instance」を Model Execution と呼ぶ。

Model Instance が実行中であるかどうかは、Model Instance が持つ State Machine の State のどれかが currentState になっているかどうかで判別を行う。Model は、この判別を行う boolean isExecuting() メソッドを持つ。

2.4　Attribute

Smart0.3 では、Attribute には型がつく。ただし、この型は、Smart0.3 の SAL ではチェックされない。

「型」にあたるものは、Smart0.3 では Model にあたる。

重要な疑問１　Attribute は、UML2.0 に合わせて UML2.0 メタモデルの Property として実装しなおした。ただし、Smart0.3 の実行中に Property は値を持つことがあり、UML メタモデルの Property には「値」に当たる association や attribute がない。したがって、Property を少し拡張して、「value」を持つようにしている。

ここで、Fig4 のように、UML メタモデルの Property は、型として UML メタモデルの Type を参照する。

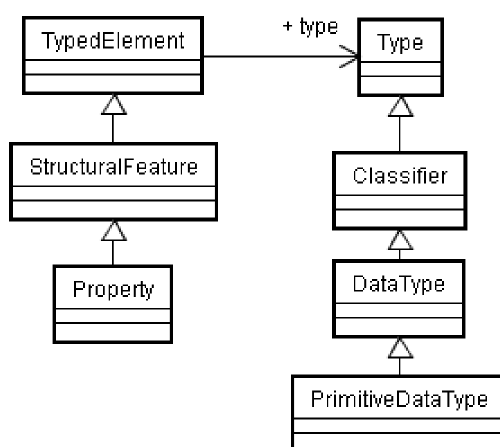

Fig 4: Type Hierarchy

Smart0.3 では、Property は UML2.0 に合わせて実装しているので、以下のようなプログラムとなる。

```
class Property extends StructuralFeature{
    Property(){
       ...
    }
    Type type; /*これがこの Property の型となる*/
    void setType(Type type){
       this.type=type;
    }
    ...
}
```

このとき、Model が Property の型となるのであれば、Model が Type のサブクラスでなければコンパイルエラーが起きてしまう。

たとえば、Model”Phone” の属性として Model “Button” の Model Insntance b1 を属性として持つとすると、以下のようなプログラムになる。

```
/*Model''Phone'' と Model''Button'' が存在*/
```

```
Model phone=new Model();
Model button=new Model();

/*Button の Model Instance として b1 を生成*/
/*
 * しかし、createObject() で return されるものは
 * Model (isInstance=true)
 * であるはずなので、ここでコンパイルエラーが起きる。
 */
Property b1=button.createObject();

/*''Phone'' に新しい Property として Button b1 を追加する*/
/*getClassifier() は、Model の持つ SmartClassImp を返すメソッド*/
phone.getClassifier().getAttributeList().add(b1);
```

b1 が Property であることが誤りか？しかし、AttributeList におさまるものが Property でなければ、Attribute として用いることができなくなる。

重要な疑問２　同様のことは、attribute の型が Integer だった場合にも起こるが、attribute の型として設定される Integer は、RuntimeInsntance のサブクラスの IntegerData か?

3 Smart System

3.1 ActivityList

Smart System は、System 全体で Activity のリストを持つ。これは、Activity を必ずしも Classifier が持っていなくてもよいという UML2.0 Superstructure のメタモデルに従ったものである。

Activity は、様々な Model から同時に使われることがある。ActivityList では、Smart System で、いま少なくとも名前が存在している Activity(中身がなくてもよい) をすべて列挙する。この Activity の中身は、Smart0.3 では SAL テキストである。

参照　この点の意味論については、BehavioredClassifier と Behavior のメタクラス図を見るとよい。これを見ると、Behavior は必ずしも所有者としての BehavioredClassifier がいなくてもよいことがわかる。なお、Activity は Behavior のサブクラスである。

注意　Activity 同様に Behavior のサブクラスである StateMachine と Interaction については、「所有者としての Classifier がいないという状況が生まれにくい」と判断して、Smart0.3 では「必ず何かの Classifier が StateMachine や Interaction を持っている (StateMachine や Interaction に所有者がいない状況はない)」と仮定して実装を行っている。

この点については、今後再考が必要かもしれない。UML2.0 の StateMachine の Description には、明確に”A state machine without a context classifier” という言葉が存在している。

3.2 SignalTrigger, CallTrigger

リモコンからエアコンに向けて飛ぶ「エアコンの温度を設定する」といったような Trigger は、SignalTrigger ではなく CallTrigger である。つまり、「温度を変えて」という signal を送る Trigger ではなく、「エアコンが持っている『温度設定』という Operation を、引数『温度=23 度』で起動して」という CallTrigger がリモコンからエアコンに飛ぶ。

3.3 UI の取り扱い

ここはあとで書き足す。

3.4 ハードウェアとソフトウェアの切り分け

ここもあとで書き足す。

4 GUI

この section では、主に Smart0.3 の UML Modeler の GUI について説明する。

4.1 2 画面構成

Smart0.2 の UI Modeler と Statechart Modeler は、まとめて UML Modeler として統合する。したがって、Model Modeling 中は、STT と UML Modeler の 2 つの画面が立ち上がっている。

Model Executing 中、Model Instance の状態を確かめたいことがある。この場合、選択された Instance の数だけ、UML Modeler とほぼ同じ画面構成の Instance Editor が立ち上がる。

疑問 UML Modeler -¿ Model Modeler とするべき?

4.2 Model and Model Instance Tree

Model を親 Node とし、この Model の Instance Pool の持つ Model Instance を子 Node とする Tree を UML Modeler に表示する。