REPORT 2
JINGMIAN ZHANG
2016 NOVEMBER

Table of Contents

1 Introduction
Multiple inheritance and design by contract are one of the fundamental cornerstones in designing reliable software system in computer science. It is imperative that all programmers can understand these crucial concepts in object-oriented programming. This is report 2 of the course EECS 3311 which is generated corresponding to the specification in the report 2 specification as well as the program text given. In report 2, I add adaptation clauses to the multiple inheritance adaptation system, and I also design, implement and document a contract for the class VM. Through the process of this report, I have learned about the features and properties of Eiffel language and object-oriented concept considerably.

2 Multiple inheritance adaptation

The adaptation system has a BON diagram shown as the following, the features written on the BON diagrams are only features that are written in the program text, NOT necessarily all the features of an individual class (Figure 1).

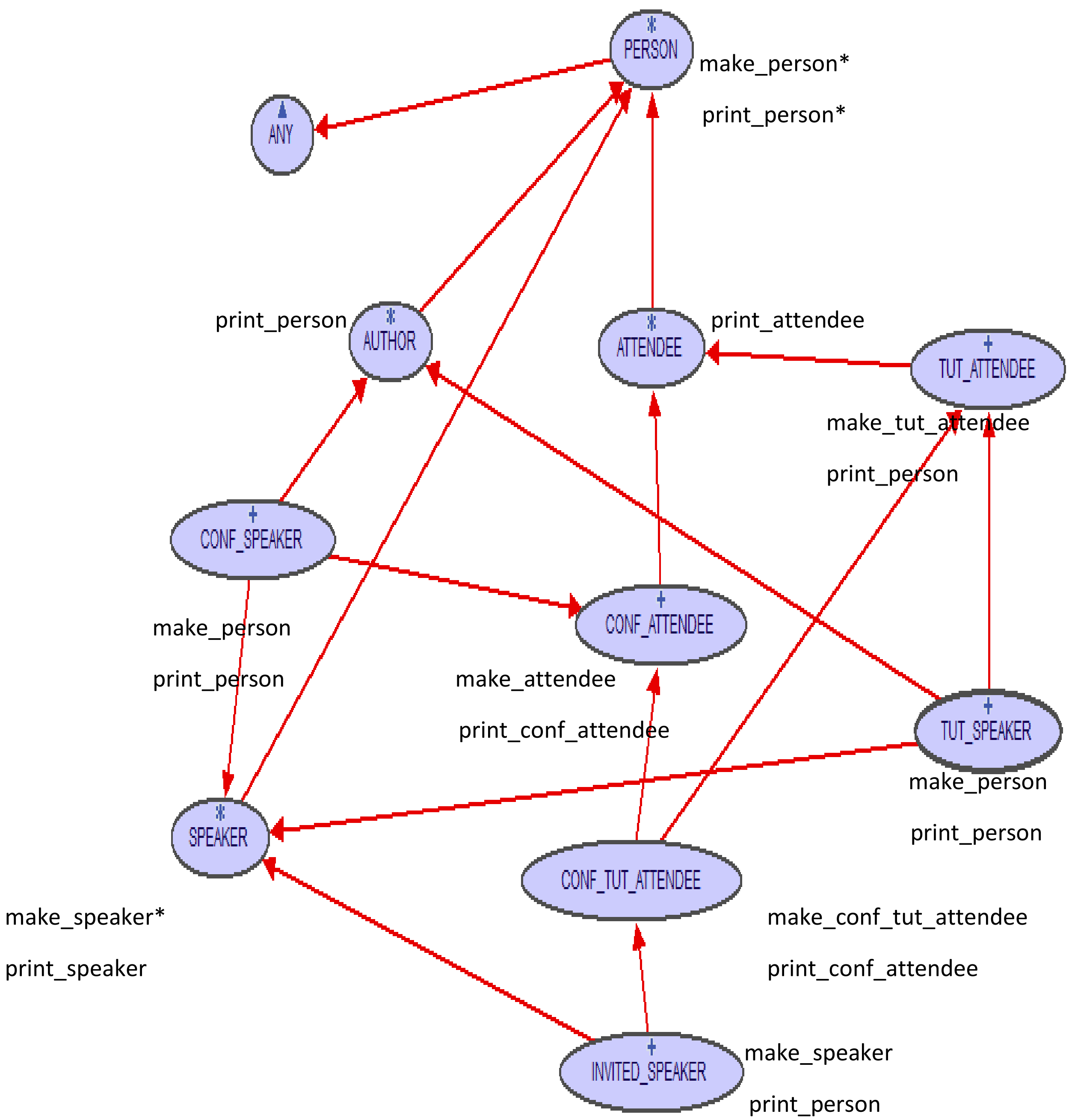

Figure 1 :BON diagram of adaptation system

Given the BON diagram of the adaptation system, we have a more general overview of what is going on between classes with their multiple inheritance and therefore an approach to complete the missing adaptation clauses.

2.1 The ATTENDEE Class

```
deferred class ATTENDEE
  inherit PERSON
            rename make_person as make_attendee end
            -- ?? replace with one clause
```

Figure 2 :ATTENDEE adaptation clause

Figure 2 demonstrates the adaption clause for deferred class ATTENDEE, which inherits from another deferred class PERSON. The deferred feature in class PERSON make_person is renamed as deferred feature make_attendee, which is later to be implemented(effected) the class CONF_ATTENDEE.

2.2 The CONF_ATTENDEE Class

```
class CONF_ATTENDEE
  inherit ATTENDEE
            -- ?? replace with one clause
            rename print_person as  print_conf_attendee
             end
```

Figure 3:CONF_ATTENDEE adaptation clause

Figure 3 demonstrates the adaptation clauses for class CONF_ATTENDEE, which inherits from the deferred class ATTENDEE. By deducing the solution output, decision is made such that print_person is renamed as print_conf_attendee in the class CONF_ATTENDEE. Print_person is deferred in PERSON as well as ATTENDEE, it is now renamed and then implemented(effected) in class CONF_ATTENDEE.

2.3 The TUT_ATTENDEE Class

```
 class TUT_ATTENDEE
  inherit ATTENDEE
            -- ?? replace with one clause
            rename make_attendee as make_tut_attendee end
```

Figure 4:TUT_ATTENDEE adaptation clause

Figure 4 demonstrates the adaptation clauses for class TUT_ATTENDEE, which inherits from deferred class ATTENDEE. The deferred feature in ATTENDEE print_person is effected in this class. The other deferred feature make_attendee(which is renamed from make_person in class ATTENDEE) is now renamed as make_tut_attendee and implemented(effected) in this class to serve the purpose of dynamic binding.

2.4 The CONF_TUT_ATTENDEE Class

```
class CONF_TUT_ATTENDEE
 inherit CONF_ATTENDEE
              rename make_attendee as make_conf_tut_attendee
              redefine print_conf_attendee, make_conf_tut_attendee
              select print_conf_attendee
              end
          -- ?? replace with three clauses
         TUT_ATTENDEE
          rename make_tut_attendee as make_conf_tut_attendee
          redefine make_conf_tut_attendee

          end
           -- ?? replace with two clauses
```

Figure 5: CONF_TUT_ATTENDEE adaptation clause

Figure 5 demonstrates the adaptation clauses for class CONF_TUT_ATTENDEE, which inherits from both class CONF_ATTENDEE and TUT_ATTENDEE. However, an issue of repeated inheritance arises, make_attend in class ATTENDEE is inherited twice from class CONF_ATTENDEE and TUT_ATTENDEE. Thus, the two different versions (make_attendee in class CONF_ATTENDEE and make_tut_attendee in class TUT_ATTENDEE) are both renamed as make_conf_tut_attendee and then redefined in this class, producing only one version. Another issue of repeated inheritance, the deferred feature print_person in class ATTENDEE is also inherited twice from class CONF_ATTENDEE and TUT_ATTENDEE. Thus, a select clause is used to select the version print_conf_attendee(which is also redefined in this class) in class CONF_ATTENDEE, to resolve the problem.

2.5 The SPEAKER Class

```
deferred class SPEAKER
  inherit PERSON
          rename make_person as make_speaker end
          -- ?? replace with one clause
```

Figure 6: SPEAKER adaptation clause

Figure 6 demonstrates the adaptation clauses for deferred class SPEAKER, which inherits from class PERSON. The deferred feature make_person in class PERSON is renamed as a deferred feature make_speaker to serve the purposes for its descendants, which will be explained later.

2.6 The CONF_SPEAKER Class

```
class CONF_SPEAKER
 inherit AUTHOR
 redefine print_person end
          -- ?? replace with one clause
        CONF_ATTENDEE
          rename make_attendee as make_person
              redefine make_person
              select print_conf_attendee
              end
          -- ?? replace with three clauses
        SPEAKER
        rename make_speaker as make_person end
          -- ?? replace with one clause
```

Figure 7: CONF_SPEAKER adaptation clause

Figure 7 demonstrates the adaptation clauses for class CONF_SPEAKER, which inherits from both class AUTHOR, CONF_ATTENDEE and SPEAKER. Since there is a print_person implementation in this class and print_person is already effective in class AUTHOR, it is evident that print_person should be redefined in this class. However, an issue of repeated inheritance arises, make_person in class PERSON is inherited twice from class CONF_ATTENDEE and SPEAKER. Thus, the two different versions (make_attendee in class CONF_ATTENDEE and make_speaker(deferred) in class SPEAKER) are merged into one feature make_person by renaming and redefining when inheriting from CONF_ATTENDEE and renaming when inheriting from SPEAKER. Moreover, the issue of repeatedly inheriting the deferred feature print_person in class PERSON is resolved by selecting the version print_conf_attendee in parent class CONF_ATTENDEE.

2.7 The TUT_SPEAKER Class

Figure 8: TUT_SPEAKER adaptation clause

```
class TUT_SPEAKER
 inherit AUTHOR
          redefine print_person end
          -- ?? replace with one clause
        SPEAKER
          rename make_speaker as make_person end
          -- ?? replace with one clause
        TUT_ATTENDEE
        rename make_tut_attendee as make_person
        redefine print_person, make_person end
          -- ?? replace with two clauses
```

Figure 8 demonstrates the adaptation clauses for class TUT_SPEAKER, which inherits from both class AUTHOR, TUT_ATTENDEE and SPEAKER. The repeatedly inheritance issue of the feature print_person in class PERSON is resolved by redefining it from class AUTHOR, effecting it from class SPEAKER, and redefining it from class TUT_ATTENDEE, which produces only one version. Same approach applies to the repeatedly inherited method make_person in class PERSON, which is renamed to make_speaker and effected when inheriting from SPEAKER, and renamed and redefined when inheriting from TUT_ATTENDEE.

2.8 The INVITED_SPEAKER Class

Figure 9: INVITED_SPEAKER adaptation clause

```
class INVITED_SPEAKER
 inherit CONF_TUT_ATTENDEE
           redefine print_person end
           -- ?? replace with one clause
         SPEAKER
         select make_speaker, print_person end
           -- ?? replace with one clause
```

Figure 9 demonstrates the adaptation clauses for class INVITED _SPEAKER, which inherits from both class CONF_TUT_ATTENDEE and SPEAKER. The repeatedly inheritance issue of the feature print_person in class PERSON is resolved by redefining it from class CONF_TUT_ATTENDEE, and selecting it from class SPEAKER. Same approach applies to the repeatedly inherited method make_person in class PERSON, the deferred version make_speaker is selected and effected in this class.

3 The Vending Machine Contract

3.1 The Vending Machine BON diagram

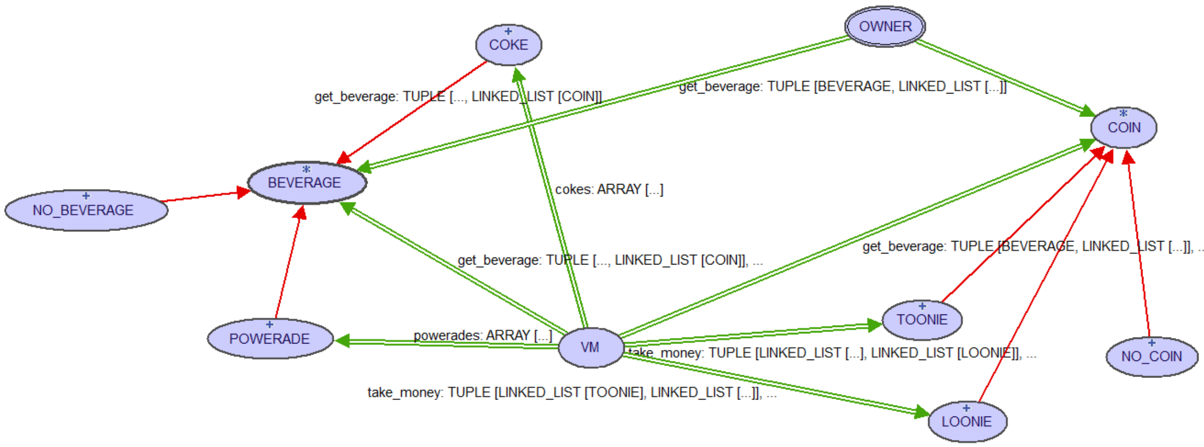

Figure 10: VM BON diagram

Figure 10 illustrates the relationship between classes in the vending machine system.

3.2 Contract statements

3.2.1 Create new machine

new_machine(coke_capacity, powerade_capacity : INTEGER)

require

minimum_case: coke_capacity >= 24; powerade_capacity >= 24

the machine must be able to hold at least one case of each beverage,

thus, the minimum case of each is 24.

maximum_case: coke_capacity + powerade_capacity <=240

the machine can hold up to ten cases in total, so the

capacity of each beverage should add up to number of ten cases

which is 240 in total.

ensure

max_coke_capacity: cokes.count = coke_capacity

check cokes instantiated to the correct capacity.

max_powerade_capacity: powerades.count = powerade_capacity

check powerades instantiated to the correct capacity.

last_full_check:

cokes_last_full = cokes.upper; powerades_last_full = powerades.upper

check last_full property of both cokes and powerades.

no_beverage: not beverage_selected.beverage_exists

check no beverage selected yet.

no_loonie_inserted: not loonie_1_inserted.coin_exists;

not loonie_2_inserted.coin_exists

check no loonie is inserted yet

no_toonie_inserted: not toonie_inserted.coin_exists

check no toonie is inserted yet

no_change: change.is_empty

check there is no change yet

3.2.2 Status feature

no_coke : BOOLEAN

require

True

ensure

count_coke: Result = (cokes.count = 0)

check if the number of cokes is zero

no_powerade : BOOLEAN

require

True

ensure

count_powerade: Result = (powerades.count = 0)

check if the number of powerades is zero

3.2.3 Customer input

insert_coin (the_coin : COIN) : TUPLE [STRING, COIN]

require True

ensure

accept_insertion: Result[1] = "accepted" implies (

is_toonie(the_coin)implies toonie_inserted.coin_exists

and (is_loonie(the_coin) implies (loonie_1_inserted.coin_exists or

(loonie_1_inserted.coin_exists and loonie_2_inserted.coin_exists))) and

Result[2] = void)

In an accepted state, if the coin is toonie, the toonie exists in temp storage;

if the coin is loonie,either one or two loonies are inserted and stored in temp storage.

And there is no return coin.

reject_insertion: Result[1] = "returned" implies (Result[2] = the_coin)

In a returned state, the coin is returned.

select_coke : STRING

require

have_coke: not no_coke

there is still coke left in the machine.

money_inserted: loonie_1_inserted.coin_exists or toonie_inserted.coin_exists

money inserted is sufficient for a purchase of coke.

ensure

result_check: Result = "a coke was selected"

check message.

coke_selected:is_coke(beverage_selected) and beverage_selected.beverage_exists

verify that beverage selected is coke and it exists.

settle_change:toonie_inserted.coin_exists or loonie_2_inserted.coin_exists implies

(change.count = 1 and is_loonie(change.last));

If a toonie or two loonie is inserted, one loonie should be the change.

(loonie_1_inserted.coin_exists and

not loonie_2_inserted.coin_exists) implies

change.is_empty

If only one loonie is inserted, change is none.

select_powerade : STRING

require

have_powerade: not no_powerade

there is still powerade left in the machine.

money_inserted: toonie_inserted.coin_exists

or loonie_2_inserted.coin_exists

money inserted is sufficient for a purchase of powerade.

ensure

result_check: Result = "a powerade was selected"

check message.

powerade_selected:is_powerade(beverage_selected);

beverage_selected.beverage_exists

verify that beverage selected is powerade and it exists.

settle_change:change.is_empty

change for a powerade is always empty.

3.2.4 Customer output

get_beverage : TUPLE[BEVERAGE, LINKED_LIST[COIN]]

require

selected_beverage: is_coke(beverage_selected) or

is_powerade(beverage_selected)

either coke or powerade is selected.

beverage_exists: beverage_selected.beverage_exists

the selected beverage exists.

ensure

result_check: Result[1] = old beverage_selected and Result[2]= change

check returned result.

store_money_for_coke:is_coke(old beverage_selected)

implies (cokes_last_full = old cokes_last_full - 1) ;

toonie_inserted.coin_exists implies (

toonies.count = old toonies.count+1 and

loonies.count = old loonies.count-1);

loonie_1_inserted.coin_exists implies (

toonies.count = old toonies.count and

loonies.count = old loonies.count+1)

If coke is selected, adjust coins storage and beverage storage for cokes accordingly.

store_money_for_powerade:is_powerade(old beverage_selected)

implies (powerades_last_full = old powerades_last_full - 1) ;

toonie_inserted.coin_exists implies (

toonies.count = old toonies.count+1 and

loonies.count = old loonies.count) ;

loonie_2_inserted.coin_exists implies (

toonies.count = old toonies.count and

loonies.count = old loonies.count+2)

If powerade is selected, adjust coins storage and beverage storage for powerades accordingly.

clear_temp_storage:

not toonie_inserted.coin_exists;

not   loonie_1_inserted.coin_exists;

not loonie_2_inserted.coin_exists ;

not beverage_selected.beverage_exists

clear up all temporary storage.

clear_change:change.is_empty

clear change.

cancel : TUPLE[TOONIE, LOONIE, LOONIE]

require

True

ensure

coins_returned:

Result = [old toonie_inserted, old loonie_1_inserted, old loonie_2_inserted]

all coins in temporary storage are returned.

clear_temp_storage:

not toonie_inserted.coin_exists;

not loonie_1_inserted.coin_exists;

not loonie_2_inserted.coin_exists ;

not beverage_selected.beverage_exists;

change.is_empty

clear up all temporary storage.

no_change_in_storage:

cokes_last_full = old cokes_last_full

powerades_last_full = old powerades_last_full

toonies.is_equal (old toonies)

loonies.is_equal (old loonies)

verify that no change is made in the storage of the machine.

3.2.5 Owner operations

restock (new_cokes : ARRAY[COKE] ; new_powerades : ARRAY[POWERADE])

require

True

ensure

cokes_full:new_cokes.count + old cokes_last_full = cokes_last_full;

powerade_full:new_powerades.count + old powerades_last_full = powerades_last_full;

verify that the numbers of each drinks are adjusted accordingly.

take_money : TUPLE[LINKED_LIST[TOONIE], LINKED_LIST[LOONIE]]

require

money_exists:

there_exists_in(toonies,agent is_toonie(?)) or

there_exists_in(loonies,agent is_loonie(?))

verify that coins exist in the storage area for coins.

ensure

empty_toonies_list: toonies.is_empty

empty_loonies_list:loonies.is_empty

ensure that all coins are withdrawn and storage area for coins is now empty.

status : TUPLE[INTEGER, INTEGER]

require

True

ensure

result_check: Result = [cokes_last_full, powerades_last_full]

verify that the result have correct number of each drink available in the machine.

paid : INTEGER

require

True

ensure

result_check: Result = (toonies.count) * 2 + (loonies.count)

verify that total dollar in the machine is calculated correctly.

reorganize (new_max_cokes, new_max_powerades : INTEGER)

require

minimum_case: new_max_cokes >= 24; new_max_powerades >= 24

maximum_case: new_max_cokes + new_max_powerades <=240

the machine must be able to hold at least one case of each beverage,

thus the minimum case of each is 24.the machine can hold up to ten cases in total

, so the capacity of each beverage should add up to number of ten cases which is 240 in total

```
valid_ratio:new_max_cokes >=cokes_last_full;
        new_max_powerades >= powerades_last_full;
```

The new capacity of each drink must not be less than the current number of each drink available.

```
ensure
    cokes_no_change:
    forall_coke(cokes,cokes.lower,cokes_last_full,agent is_coke(?))
    powerades_no_change:
    forall_powerade(powerades,powerades.lower,powerades_last_full,agent is_powerade(?))
```

Both beverage storage area should remain intact.

3.2.6 Class Invariant

```
cokes_check:
    forall_coke(cokes,cokes.lower,cokes_last_full,agent is_coke(?));
    cokes.upper = cokes_last_full;
    cokes.lower = 1;
    cokes_last_full = cokes.count
  powerades_check:
  forall_powerade(powerades,powerades.lower,powerades_last_full,agent is_powerade(?));
  powerades.upper = powerades_last_full;
  powerades.lower = 1;
  powerades_last_full = powerades.count
```

boundaries and content check for both cokes and powerades.

```
    change_property: change.count = 1 or change.is_empty
```

in this problem can be at most one loonie, so change has either one loonie or

change is empty.