Self-Study Module: Contracts

This module shows you how to tighten up your object design with the use of design-by-contract.

Learning objectives:

- To develop skill in understanding and writing pre/post conditions for operation behaviour within a contract programming approach

References:

- Chapter 14, section 14.4 and 14.5, Simon Bennett, Steve McRobb, Ray Farmer, "Object-Oriented Systems Analysis and Design using UML", McGraw-Hill, 2nd edition, 2002.
- Meyer, Design By Contract, IEEEComputer, 1992
- Excerpt from John McGregor and David Sykes, "A Practical Guide to Testing Object-Oriented Software", Addison-Wesley, 2001.

  p.171, Sidebar "Test Cases for Failed Preconditions"

Contents:

Worked example – Exploring pre/post conditions within a contract approach.........2
Exercise 1 - Visualising the change in state..8
Exercise 2 - Pre/Post Conditions: Design to Code...9

The following exercises explore the use of pre/post conditions for specifying class operations and object behaviour. You are provided with a set of use case scenarios and operation specifications. Before attempting the exercises, go through the *worked example* below. Then complete the following exercises to improve your understanding of pre/post conditions.

Worked example – Exploring pre/post conditions within a contract approach

This worked example explores using pre/postconditions in design of class operations and how they translate to code using a contract approach.

(Ref: Bennett p.257)

The use case *Change the grade for a member of creative staff*, involves the operation `CreativeStaff.changeGrade()` with a specification of

CreativeStaff.changeGrade(grade: Grade, gradeChangeDate: Date)

pre: `grade` object is valid

`gradeChangeDate` object is valid

post: a new `StaffGrade` object exists

the new `StaffGrade` object is linked to the `creativeStaff` object

the new `StaffGrade` object is linked to the previous one

the value of the previous object, `staffGrade.gradeFinishDate` is set equal to gradeChangeDate

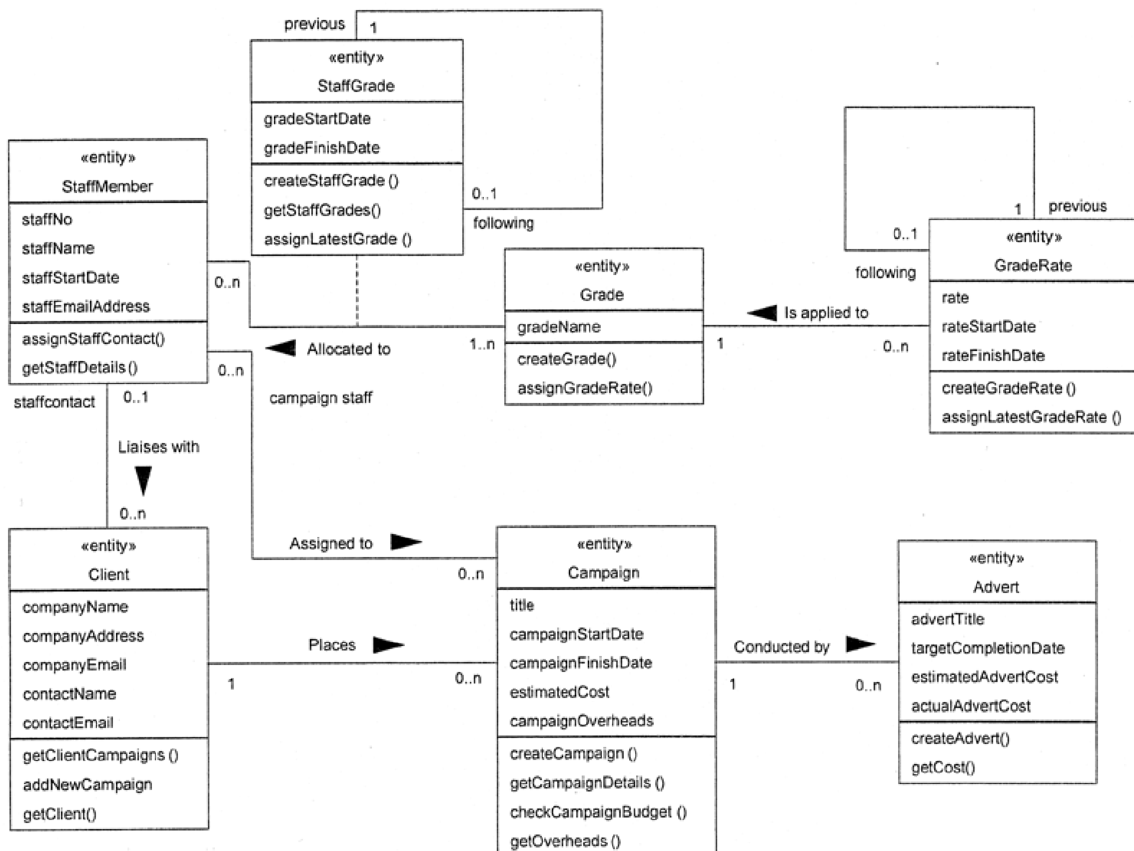


Figure 3. Class diagram - ref: Bennett Figure A3.14, p.207

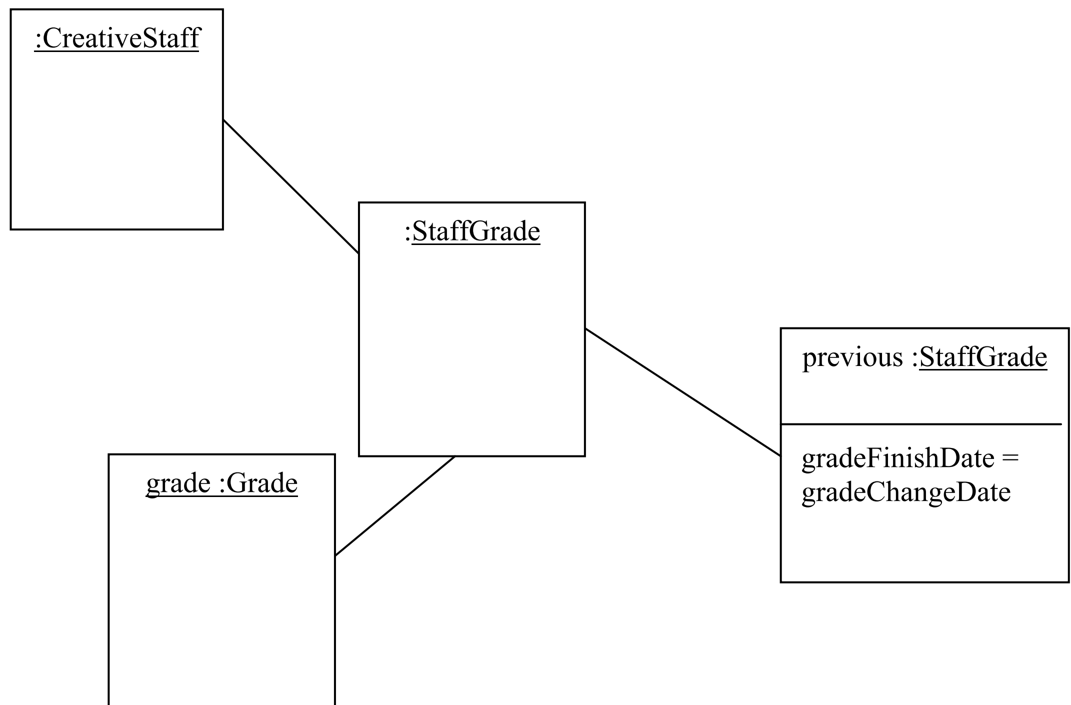
Some helpful interpretation of the class diagram:

The StaffGrade class is an association class. We can assume that each instance of StaffMember can have 1 to many Grade instances. However the StaffGrade class acts as an intermediary between the StaffMember and Grade classes. It can be modelled as a list or collection. Each time a staff member is allocated a new grade, a new StaffGrade object is created and added to the list. The new Grade object is actually linked to the new StaffGrade object rather than the StaffMember object.

Assume that the createStaffGrade() operation is the constructor.

> What does it mean for an object to be valid?
>
> To say an object is valid means that an instance exists and the values of its attributes correspond to its specification. So for a Date object to be valid, its attributes corresponding to the day, month and year must have integer values that fall into valid ranges for day (1-31), month (1-12) and year.

Object diagram (at completion of operation)



Discussion ...

What is the purpose of the input Grade object, grade?

The link between the new StaffGrade object and the Grade object, grade is not described in the postcondition - should it be?

What reference attributes are required in each class?

What to do if the precondition fails?

```
class CreativeStaff
{
    StaffGrade staffGrade;

    ...
    public changeGrade( Grade grade, Date gradeChangeDate )
    {
        // create new StaffGrade object
        StaffGrade newStaffGrade = new staffGrade(gradeChangeDate, grade);
        staffGrade.setFinishDate(gradeChangeDate);
```

Note that the method contains NO code for checking the

```
            // link the new StaffGrade object to the current
            staffGrade.assignLatestGrade(newStaffGrade);
        /* staffGrade now points to the latest StaffGrade object (done inside
    assignLatestGrade) */
        }
}
```

Calling code:

```
gradeChangeDate = new Date();

if (grade != null)
    CreativeStaff.changeGrade(grade, gradeChangeDate);
```

No need to explicitly check the precondition – can assume a valid Date object is created

Explicit check of precondition

Consider an alternative design where the precondition has been relaxed ...

pre: `gradeChangeDate` object is valid

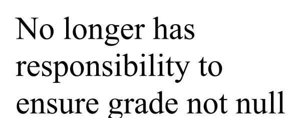
How does this affect the code implementation?

Now the responsibility for ensuring a valid Grade object lies with the method instead of the calling code. How should the method handle the case when `grade` is null?

Modified code (changes in *italics*):

```
class CreativeStaff
{
StaffGrade staffGrade;

...
public changeGrade( Grade grade, Date gradeChangeDate )
{
if (grade != null) {
// create new StaffGrade object
StaffGrade newStaffGrade = new staffGrade(gradeChangeDate, grade);
staffGrade.setFinishDate(gradeChangeDate);
// link the new StaffGrade object to the current
staffGrade.assignLatestGrade(newStaffGrade);
// staffGrade now points to the latest StaffGrade object
}
else {
/* decide how to handle case of grade == null */
}
}
}
```



Calling code:

```
gradeChangeDate = new Date();
CreativeStaff.changeGrade(grade, gradeChangeDate);
```

As you can see, this implementation is now more complicated for the method as it must handle the exceptions and error cases. Perhaps the original design is preferable because there is really no point for the method to do anything if one of its inputs is invalid. One should ask, does the calling code have ready access to the data specified in the precondition in order for it to confirm the precondition prior to using the method?

Exercise 1 - Visualising the change in state

For the operations in the box, check your understanding of the postconditions by drawing an object diagram showing the final state of the objects in the program. Make changes to your object diagram as you progress through the scenarios. Assume that the current state of the program has a single Campaign object containing 2 Advert objects, as depicted in the following object diagram.

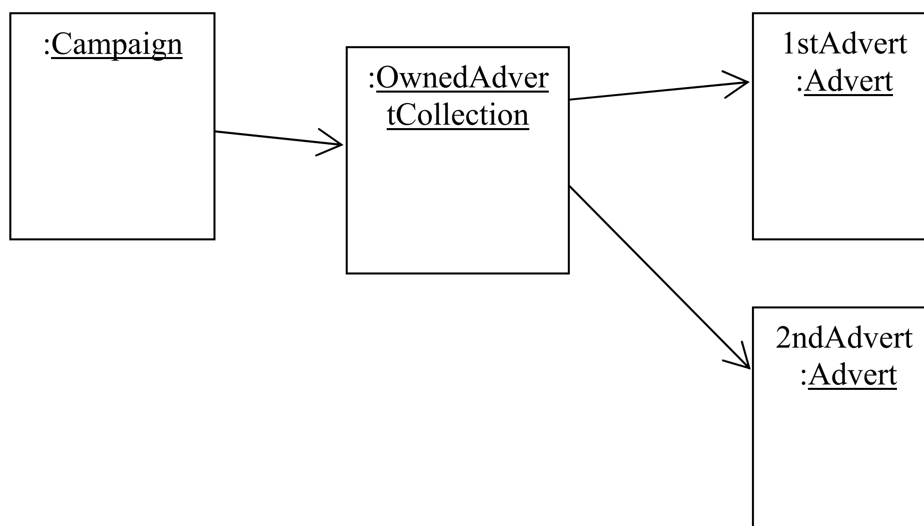


Figure 1. Object diagram of current state

Exercise 2 - Pre/Post Conditions: Design to Code

For the operations in the box, write the code implementation in Java that satisfies the pre/post conditions and the class invariant. Using an incremental approach, implement only what is required at this point to satisfy the scenarios. Remember that we are using a contract programming approach rather than a defensive one – see excerpt *McGregor and Sykes*.

Suggested steps and hints:

Firstly examine the postcondition – it describes what must be true after the operation has been performed. Typically, postconditions list changes to the state of the program, such as changes to attribute data, creation or deletion of instances, and creation or deletion of associations.

You need to decide on the algorithmic form of the operation and turn it into Java code.

Secondly examine the precondition. If there is no precondition, then the method itself may have to check for invalid or potentially incorrect cases that it will need to handle, otherwise runtime errors may be generated. Conversely if there is a precondition, then any part of the program that calls this method may need to check the validity of data specified in the precondition prior to calling the method. Do you need to implicitly or explicitly check the precondition in the calling code? Also make sure that the method itself does no validity checking for this type of precondition.

Lastly, write the code to satisfy the scenarios. When doing this you will need to consider the preconditions for each operation, as explained above. You will need to make some assumptions about what data is available from the user interface. The focus here is on using each operation in the context of the scenarios. This code will not necessarily be a proper working solution as some essential parts will be missing.

Use cases and Scenarios

The set of use cases, scenarios and operation specifications are taken from the Bennett case study A – Agate Ltd Advertising Agency. Use case descriptions are taken from Figure A2.2, p.152 Bennett.

Use case *Add a new advert to a campaign*.

A campaign can consist of many adverts. Details of each advert are entered into the system with a target completion date.

Scenario: The user selects a campaign. Details of the new advert are entered and a new advert is added to the selected campaign.

Use case *Record completion of an advert*.

When an advert is completed, the actual completion date is entered.

Scenario: The set of adverts for that campaign is displayed to the user for selection of the *2nd advert for completion*. The actual completion date is entered.

Scenario: Lastly the set of adverts for that campaign is displayed to the user for selection of the *1st advert for removal*.

Operation specifications

The operation signatures have been written in UML notation.

The pre/post conditions have been written informally, generally describing a query, attribute modification, object creation and association creation or deletion. Remember that a class invariant must hold true at the end of an operation, thus it can be treated as part of the postcondition.

Advert.createAdvert(title: String, type: String): Advert

pre: `title` is valid

`type` is valid

post: A new `Advert` object exists.

class invariant:

type = {"leaflet", "magazine", "newspaper", "poster", "radio", "tv"}

Advert.getTitle( ): String

pre: `Advert` object is valid

post: `Title` is returned.

Advert.setCompleted(endDate: Date)

pre: `Advert` object is valid

`endDate` is valid

post: completionDate is set to endDate

AdvertCollection.addAdvert(newAdvert: Advert): Boolean

pre: `newAdvert` object is valid

post: If no duplicate titles, then true is returned and the `newAdvert` object is linked to the `ownedAdvert` object. Else false is returned.

invariant:

`Advert.title` is unique (i.e. no duplicates allowed)

AdvertCollection.removeAdvert(oldAdvertTitle: String): Boolean

pre:

post: The `oldAdvert` object with matching title is no longer linked to the `ownedAdvert` object. True is returned if `oldAdvert` was successfully removed, else false.

AdvertCollection.find(title: String): Advert

pre:

post: The `Advert` object with matching `title` is returned, else null is returned.

AdvertCollection.findFirst(): Advert

pre:

post: The first `Advert` object in the collection is returned.

AdvertCollection.getNext(): Advert

pre:

post: The next `Advert` object in the collection is returned.

Campaign.listAdverts( ): String

pre:

post: Title of each `Advert` object in `ownedAdvertCollection` is retrieved and returned as a concatenated string, with each title separated by '+'.

invariant: valid `ownedAdvertCollection` (i.e. valid collection object which may be empty)

Campaign.addNewAdvert(newAdvert:Advert)

pre: `newAdvert` object is valid

post: the `newAdvert` object is linked to the `ownedAdvertCollection` object

Examine the class diagram below for more information about the design.

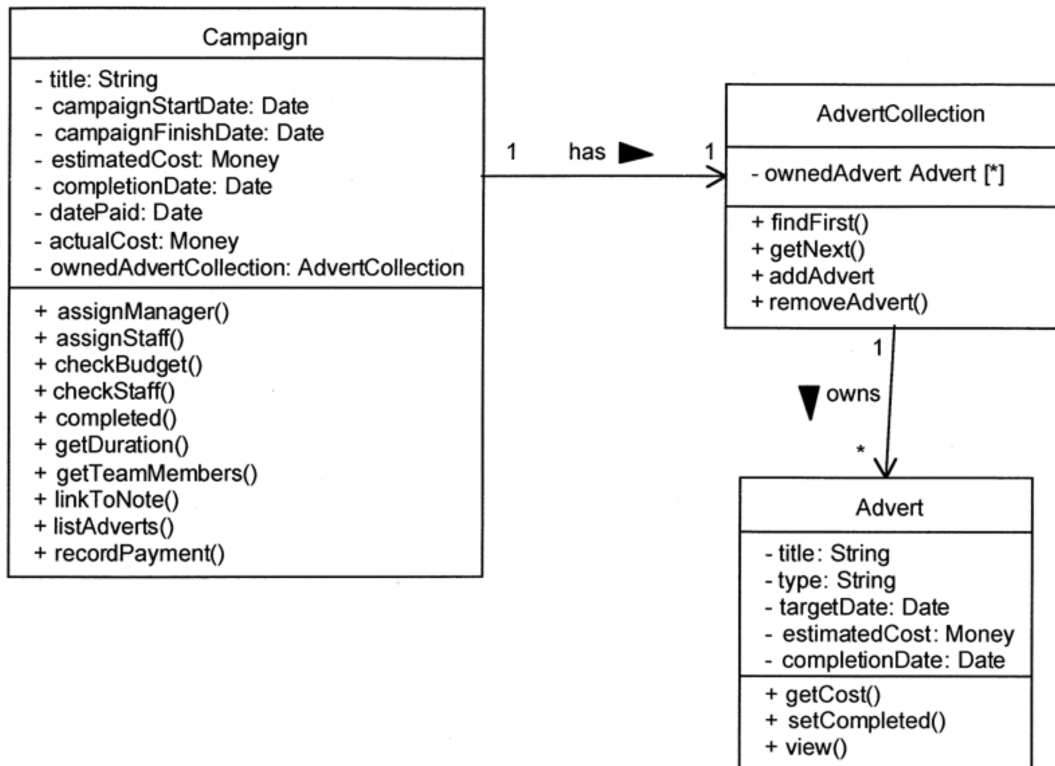


Figure 2. Class diagram - ref: Bennett Figure 14.13, p.359

Discussion ...

If the precondition for `Campaign.listAdverts()` was modified to `ownedAdvertCollection` object is not empty, how would this affect the code implementation?