

*Object-Oriented Design : Module 7*

# Contracts

---

Lian Loke  
University of Technology, Sydney

## **Contributors**

John Leaney, Lizveth Robles, John Reekie,  
University of Technology, Sydney

Terms of Use: Creative Commons Attribution-ShareAlike 2.5  
<http://creativecommons.org/licenses/by-sa/2.5/>

# Software Quality = Reliability

---

- Correctness
    - Perform according to specification
  - Robustness
    - Handle abnormal situations
-

## Design by Contract

---

- Mutual obligations
  - Inspired by business or service contracts
  - Bertrand Meyer, Eiffel language
- 

## Contract between customer and airline

---

### □ Obligations

- **Client:** Be at the airport at least 30 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.
- **Supplier:** Bring customer to destination.

### □ Benefits

- **Client:** Reach destination.
  - **Supplier:** No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.
-

## Services Among Objects

---

- When objects collaborate, one object typically provides a service to another
  - Examples:
    - A `client` object might ask a `Campaign` object for its details
    - The same `client` object might then ask a boundary object to display its related `Campaign` details to the user
- 

## Software contracts

---

- Specifications (or contracts) govern the interaction of the object with the rest of the world
  - States what the object should do
-

# Assertions

---

- Precondition
    - What conditions must be satisfied *before* an operation can take place?
  - Postcondition
    - Expresses the valid results of the operation after completion
    - What are the conditions that apply *after* an operation is completed?
    - What states will the system be in?
  - Class invariant
    - Constraints on state of object that always holds true
    - Implied in every postcondition
- 

# The schema for a contract

---

## Contract reference and name

**Operation:** Name of operation, and parameters (the signature)

### **Cross**

**References:** Use Cases this operation can occur within

**Preconditions:** Noteworthy (non-trivial) assumptions about the state of the system or objects (in the Domain Model) before execution of the operation.

**Postconditions:** The state of objects (in the Domain Model) after completion of the operation, typically:

- \* instance creation
  - \* association formed
  - \* attribute modification
  - \* association broken
-

## Example 1

---

The operation `Advert.getCost( )` has a signature  
`getCost( ) :Money`

and contract specified by

**pre-condition:** none

**postcondition:** a valid money value is returned

- The postcondition could be tightened by specifying a valid range of values for money. e.g. `money >= 0`

---

Ref: Bennett, Chapter 10

## Example 2

---

The operation `Campaign.assignStaff( )` is responsible for assigning a member of staff to a campaign, and has a signature

`Campaign.assignStaff( CreativeStaff creativeStaff )`

and contract specified by

**pre-condition:** `creativeStaff` object is valid

**postcondition:** a link is created between `campaign` object and `creativeStaff` object

---

## Example 3

---

The use case

*Change the grade for a member of creative staff*

When a member of staff is promoted, the new grade and the date on which they start on that grade are entered. The old staff grade is retrieved and the finish date set to the day before the start of the new one.

involves the operation

`CreativeStaff.changeGrade( )` with a signature

`CreativeStaff.changeGrade( String revisedGrade,  
Date gradeChangeDate )`

---

## Example 3 ctd

---

with contract specified by

**pre-conditions:**

revisedGrade is a valid value (eg. "step2")  
gradeChangeDate is a valid date

**post-conditions:**

a new `Grade` object exists, with the grade variable set to revisedGrade  
new `Grade` object linked to `creativeStaff` object  
new `Grade` object linked to previous (called `staffGrade`)  
value of previous `staffGrade.gradeFinishDate` set equal to day before `gradeChangeDate`

---

## Example 4

---

The use case *Process Sale* involves the operation `makePayment( )` with a signature

```
Sale.makePayment( amount: Money )
```

and contract specified by

**preconditions:**

```
There is a sale underway.  
amount >= Sale.total
```


**postconditions:**

```
a Payment instance p was created.  
p.amountTendered became amount.  
p was associated with the current Sale.  
The current Sale was associated with the Store  
(to add it to the historical log of completed sales)
```

---

## Class Exercise

---

 Draw an object diagram showing the effect of the postcondition for the `makePayment( )` operation

---

## What if the precondition is not satisfied?

---

- Example:

function square\_root ( x: REAL ) : REAL

precondition:  $x \geq 0$

- Consider the consequences if this function was part of a satellite navigation system ...
- 

## Implementing Pre/Post Conditions with a CP Approach

---

- A *contract* approach trusts the service guaranteed by an operation as long as its precondition is respected.
  - Clear allocation of responsibilities
  - Results in lean code
-

## The opposing camp

---

- Defensive programming
- "Anything can go wrong, any time, so we better be prepared for it."

```
function do_something(type1 arg1, type2 arg2) {  
    if (makes_no_sense(arg1))  
        toss_it_back_and_exit(arg1);  
    else if ((makes_no_sense(arg2))  
        toss_it_back_and_exit(arg2);  
    type3 var1 = do_something_to(arg1);  
    if (makes_no_sense(var1))  
        exit_in_panic();  
    etc...  
}
```

---

## Java code example

---

```
class sale {  
    private Payment payment;  
    private Store store;  
  
    public void makePayment(Money amount)  
    {  
        if (amount >= this.total) {  
            Payment p = new Payment(amount);  
            this.payment = p;  
            store.addSale(this);  
        }  
    }  
}
```

This conditional statement is only coded with a *defensive* approach. With a *contract* approach, the check may need to appear in the calling code.

---

# Unvarnished comparison

---

- ❑ DbC makes you **think** more about the **interface** of operations
  - ❑ DP is a lot easier to understand for most programmers: "Check everything."
  - ❑ DbC requires technology support to do it properly. In Eiffel, DbC is supported fully by the language. In Java and C++, we can use *assert* as an rough approximation...
  - ❑ DP naturally leads to a proliferation of error-checking code. More lines of code = more bugs..
  - ❑ DbC provides you with a useful tool for *specifying* operations on classes, which can then be used to generate test cases. (This is important in this subject.)
-