

Design by Contract – beyond class modelling

Introduction

Design by Contract (DbC) or Programming by Contract is an approach to designing software. It says that designers should define precise and verifiable interface specifications for software components, with the use of preconditions, postconditions and invariants. These specifications are referred to as "contracts"; in the same way as a business contract entails certain obligations and conditions.

- ❑ developed by Bertrand Meyer, “Bubbles (aka UML boxes) don’t crash”, and forms a central feature of Eiffel
- ❑ uses **assertions** – assertion is a Boolean statement that should never be false. If so, it indicates a bug
- ❑ assertions usually only checked during debugging/testing
- ❑ three kinds of assertions:
 - post-conditions
 - pre-conditions
 - invariants
- ❑ pre-conditions and post- conditions are associated with operations

Pre-condition

- ❑ expresses something about the state of a program that should be true before an operation is executed
- ❑ e.g. a pre-condition for an *integer-square-root* operation might be

```
input >= 0
```

indicates it is an error to invoke *square-root* on a negative number
- ❑ e.g. pop() should not be called on an empty stack, precondition is

```
s.isEmpty() == false
```
- ❑ makes explicit that the calling routine is responsible for ensuring that something is true before operation is invoked
- ❑ lack of one may lead to too little or too much checking (duplicate checking code and thus more complicated program)

Post-condition

- ❑ a statement of what things should be like after the execution of an operation
- ❑ e.g. *integer-square-root* operation

```
result * result <= input < (result+1)*(result+1)
```
- ❑ expresses **what** an operation does rather than **how** it does it
- ❑ separates implementation from interface

- leads to a stronger definition of an **exception**. Exception occurs when an operation is invoked with its pre-condition satisfied and is unable to return with its post-condition true

Benefits - Obligations

	Benefit	Obligation
Client	<ul style="list-style-type: none">- no need to check output values- result guaranteed to comply to postcondition <p>④</p>	<p>satisfy pre-conditions</p> <p>①</p>
Provider	<p>②</p> <ul style="list-style-type: none">- no need to check input values- input guaranteed to comply to precondition	<p>③</p> <p>satisfy post-conditions</p>

Spec# Example

<http://www.rise4fun.com/SpecSharp>

The following shows a C# implementation of *integer-square-root()* where the precondition (requires) and postcondition (ensures) are described in Spec#. Spec# supports DbC for C#. They form a contract for the method. It can be proved that the method code satisfies the contract.

```
class Fig1 {
    int ISqrt(int x)
        requires 0 <= x;
        ensures result*result <= x && x < (result+1)*(result+1);
    {
        int r = 0;
        while ((r+1)*(r+1) <= x)
            invariant r*r <= x;
        {
            r++;
        }
        return r;
    }
}
```

Invariant

- ❑ an assertion about a class or a method
- ❑ invariant is always true for class instances – meaning whenever object is available for an operation to be invoked on it
- ❑ may be temporarily false during execution of method
- ❑ e.g. Account class

balance == sum of transaction amounts

- ❑ e.g. in the above Spec# example, $r*r \leq x$ is an invariant of the while loop
- ❑ invariant is added to pre-conditions and post-conditions of public methods

Subclassing/Polymorphism

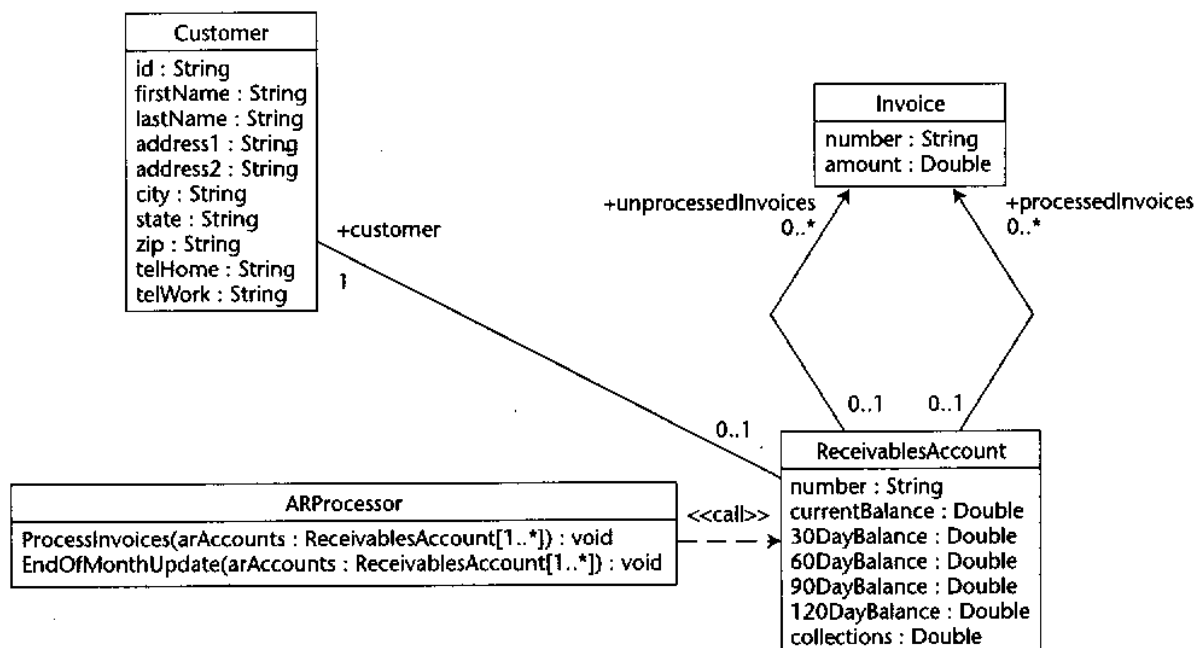
- ❑ assertions useful with polymorphism, assertions can help keep subclass operations consistent with those of superclass
- ❑ invariants and post-conditions must be true for all subclasses, subclass can strengthen these, i.e. make them more restrictive
- ❑ not allowed to strengthen pre-conditions – because of substitutability, can only weaken a pre-condition,
- ❑ if a subclass strengthened a pre-condition, then a superclass operation could fail when applied to instance of subclass
- ❑ pre-conditions: best pay-off for least overhead
- ❑ one language to support assertions is Eiffel

In class we have already looked at the role of design by contract (DbC) in software specification. We examined the notions of weakening the pre-condition and strengthening the post-condition in cases of polymorphism and inheritance.

DbC and OCL

The following material is largely drawn from the text: **Model Driven Architecture – Applying MDA to Enterprise Computing** by David S. Frankel.

The following class diagram is to be used with the DbC constraints described in OCL on the following page. Note that DbC does not have to use OCL (object constraint language). OCL is now officially part of UML 2.0.



```

-----
--ReceivablesAccount invariants
-----
--An invoice cannot be both unprocessed and processed.

context ReceivablesAccount inv:

    unprocessedInvoices->intersection(processedInvoices)->isEmpty ()

--An invoice number must be six characters in length.

context ReceivablesAccount inv:

    self.number->size () = 6

-----
--ARProcessor::ProcessInvoices pre-conditions
-----
--There must be some unprocessedInvoices.

context ARProcessor::ProcessInvoices (arAccounts : Set
(ReceivablesAccount)) pre:

    arAccounts->forAll (unprocessedInvoices->notEmpty () )

-----
--ARProcessor::ProcessInvoices post-conditions
-----

--unprocessedInvoices become processedInvoices.

context ARProcessor::ProcessInvoices (arAccounts : Set
(ReceivablesAccount)) post:

    arAccounts->forAll
    (
        ( unProcessedInvoices->isEmpty () and
          processedInvoices->includes (unprocessedInvoices@pre)
        )
    )

-----
--ARProcessor::EndOfMonthUpdate pre-conditions
-----
--There are no unprocessed invoices.

context ARProcessor::EndOfMonthUpdate (arAccounts : Set
(ReceivablesAccount)) pre:

    arAccounts->forAll (unprocessedInvoices->isEmpty () )

```

```

-----
--ARProcessor::EndOfMonthUpdate post-conditions
-----
--For all of the ARaccounts the following holds:
  --The Collections value is its previous value plus the previous
120DayBalance and
  --the 120DayBalance is the previous 90DayBalance and
  --the 90DayBalance is the previous 60DayBalance and
  --the 60DayBalance is the previous 30DayBalance and
  --the 30DayBalance is the previous currentBalance
  --the currentBalance is 0.

context ARProcessor::EndOfMonthUpdate (arAccounts : Set
(ReceivablesAccount)) post:

  arAccounts->forAll
  (
    -- @pre modifies an identifier to refer to the value it had
    -- before the operation executed.
    currentBalance = 0 and
    30DayBalance = currentBalance@pre and
    60DayBalance = 30DayBalance@pre and
    90DayBalance = 60DayBalance@pre and
    120DayBalance = 90DayBalance@pre and
    Collections = collections@pre + 120DayBalance@pre
  )

```

DbC constraints have two basic purposes:

- highlighting unacceptable corner cases
- expressing key semantics of class or operation to which they apply

E.g. ReceivableAccount invariant indicates that an invoice cannot be both processed and unprocessed.

Post-condition for EndOfMonthUpdate() says in platform-independent fashion that current balances need to be rolled over to 30 day balances. Reflects a business rule.

Similarly, EndOfMonthUpdate() pre-condition that there should be no unprocessed invoices reflects another business rule.

Constraints nail down properties of classes without predetermining platform specific techniques for implementing them.

Contracts are useful to clients and generators.

OCL constraints flesh out the model and connect the dots for generators and programmers and help discover design flaws.

The more precisely a classes properties are specified, the easier it is to **reuse** the class. OO and component based development promise reuse as major advantage. But developers find class or component reuse difficult without access to source code because the usage contract is not well specified.

Interoperability between components from different companies also an issue even when components support standardised interfaces, messages and protocols unless there is a good understanding of the contract that the interface must honour when implemented. Important to Java Community process, OMG, and B2Bi initiatives.

Interoperability is based on shared understanding among interoperating objects.

- Syntactic interoperability – e.g. CORBA's IDL & IIOP allow shared understanding of operation signature. Allows values to pass thru all transport layers transparently.
- Semantic interoperability based on objects' ability to coordinate their functioning based on a shared understanding of the semantics.
- Formal DbC contracts improve semantic interoperability. Hence have an important contribution to make to B2Bi.

Precision versus Detail

DbC constraint writing in OCL is a form of coding. It is more abstract than 3GL programming as it omits many implementation details that are necessary in a 3GL. But it is still no less precise than 3GL.

Example of an electric motor to illustrate precision versus detail. Spec can very precise while giving no detail as to internals of motor. Abstraction is suppression of irrelevant detail, not vagueness.

Can infer exceptions for operations from DbC constraints. Pre-condition maps directly to an exception to be thrown when the operation is invoked and the precondition is not satisfied.

A lot of tool support now available for checking constraints at runtime, e.g. with Spec#. Generate code for checking constraints. Can specify when in time they are to be checked.

DbC and QA

DbC can provide a framework for quality assurance (QA). Pre-conditions, post-conditions and invariants represent much of what QA engineer must validate. DbC constraints can be used by code generators to produce test harnesses or for program verification as in Boogie tool with Spec#.

Even when not used automatically, DbC constraints act as precise guides to QA engineers regarding what they must test. Also useful for code reviews or walk throughs.

There are other non-functional factors such as scalability and performance, that are not covered by DbC assertions. Can use other UML extension for these.

Tool Support

Many tools have slots for associating constraints with classes and operations.

MDA tools may include OCL editing facilities for writing OCL assertions, similar to IDEs for 3GLs.

Main barrier to effective use of DbC is software development pressure. Rigorous DbC can be time consuming. However, MDA DbC may be timesaving as generators elaborate each formal constraint into many lines of application and test harness code.

Without MDA, DbC only addresses quality. With MDA, it can help with production costs.

Comment

UML class diagrams act as constraints on instances of those classes. OCL describes other constraints that class diagram cannot show.

Behavioural Modelling

Class models with DbC do not describe control flow and state changes inside code. Do not detail how objects interact. They are static structure models rather than behavioural models such as use cases, state machines, activity diagrams and sequence diagrams.

However a DbC based class does describe behaviour when it specifies pre and post-conditions for the class operations. But it strictly confines itself to aspects of behaviour relevant to a client. See example below.

Behavioural modelling is at least as important to MDA as static modelling. Some of the most advanced industrial MDA is based on behavioural modelling. E.g. Shlaer-Mellor systems use state machines to generate embedded code for things like photocopiers and telecom switches.

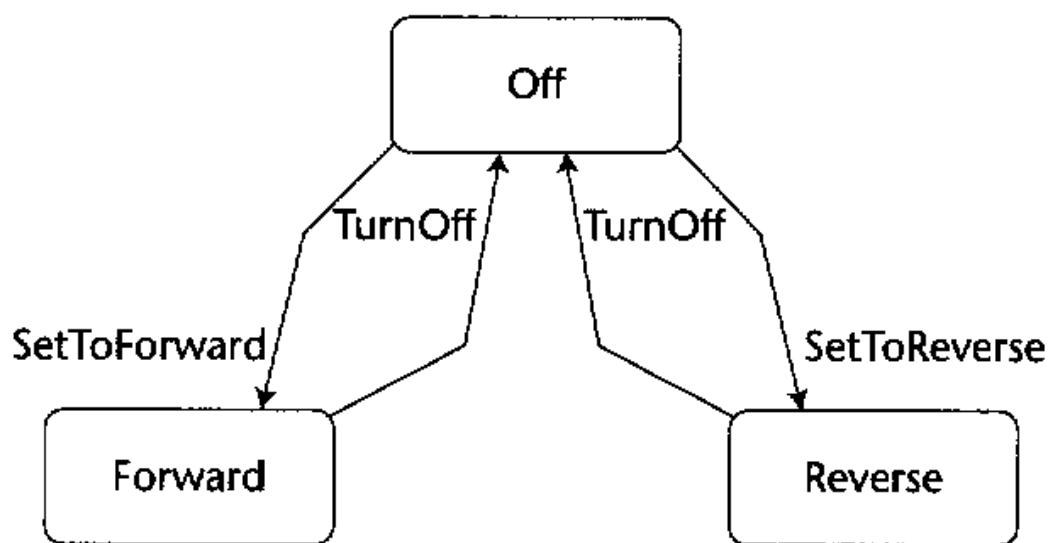
State machines fall into two categories. One deals with transitions of public class attributes, i.e. attributes visible to clients and thus part of the contract (protocol state machines).

For a DbC class, all the info in such a state machines would also be available in the class model (see following example). Protocol state machine is another representation of info available that a correct DbC-based class model provides.

Note, DbC constraints can convey more and be more complex than a state machine.

PowerDrill
mode : ModeKind
Turn Off() SetToForward() SetToReverse()

<<enumeration>> ModeKind
Off Forward Reverse



```

-----
--PowerDrill::TurnOff pre-conditions
-----
--The drill must be in Forward or Reverse mode.

context PowerDrill::TurnOff () pre:

    self.mode = ModeKind::#Forward or self.mode = ModeKind::#Reverse

-----
--PowerDrill::TurnOff post-conditions
-----
--The drill is off.

context PowerDrill::TurnOff () post:

    self.mode = ModeKind::#Off

-----
--PowerDrill::SetToForward pre-conditions
-----
--The drill must be off.

context PowerDrill::SetToForward () pre:

    self.mode = ModeKind::#Off

-----
--PowerDrill::SetToForward post-conditions
-----
--The drill is in Forward mode.

context PowerDrill::SetToForward () post:

    self.mode = ModeKind::#Forward

-----
--PowerDrill::SetToReverse pre-conditions
-----
--The drill must be off.

context PowerDrill::SetToReverse () pre:

    self.mode = ModeKind::#Off

-----
--PowerDrill::SetToReverse post-conditions
-----
--The drill is in Reverse mode.

context PowerDrill::SetToReverse () post:

    self.mode = ModeKind::#Reverse

```