LECTURE 3: DESIGN BY CONTRACTS

Abstract Data Types (ADT) technology is inherently beneficial to the Object Oriented Programming (OOP) paradigm. An integral aspect for the success of OOP is for the existence of guidelines or methodologies that ensure software reliability. Reliability has therefore become an important nonfunctional requirement in software construction. Design by contracts stipulates a methodology and approach that promises reliability in software construction. For this lecture, we explore the approaches and viewpoints that describe contract based programming.

Reliability can be defined as the combination of correctness and robustness or more prosaically, as the absence of bugs. At least three reasons justify devoting particular attention to reliability in the context of object-oriented development:

- The cornerstone of object-oriented technology is reuse. For reusable components, which may be used in thousands of different applications, the potential consequences of incorrect behavior are even more serious than for application specific developments.

- Proponents of object-oriented methods make strong claims about their beneficial effect on software quality. Reliability is certainly a central component of any reasonable definition of quality as applied to software.

- The object-oriented approach, based on the theory of abstract data types, provides a particularly appropriate framework for discussing and enforcing reliability.

NOTE: Building software components on the basis of carefully designed contracts would help reduce bugs.

To enhance reliability, a software construction approach should be able to address, in the least, the following three issues:

- Adhere to a coherent set of methodological principles helping to produce correct and robust software.

- Follow a systematic approach to the delicate problem of how to deal with abnormal cases. Leading to a simple and powerful *exception-handling* mechanisms

- Exude a better understanding of inheritance and of the associated techniques (*redeclaration*, *polymorphism*, and *dynamic binding*) through the notion of subcontract, allowing a systematic approach to using these powerful but sometimes dangerous mechanisms.

3.1 Defensive Programming

Software engineering and programming methodology textbooks that discuss reliability often emphasize the technique known as defensive programming, which directs developers to protect every software module against the slings and arrows of outrageous fortune. In particular, this

encourages programmers to include as many checks as possible, even if they are redundant with checks made by callers. Include them anyway, the advice goes; if they do not help, at least they will not harm (what would be referred to as a **wide contract**). This approach suggests that routines should be as general as possible. A partial routine (one that works only if the caller ensures certain restrictive conditions at the time of the call) is considered dangerous because it might produce unwanted consequences if a caller does not abide by the rules.

Disadvantages of Defensive programming

- **Introduction of Complexity**: Adding possibly redundant code "just in case" only contributes to the software's complexity - the single worst obstacle to software quality in general, and to reliability in particular.

- **More sources of bugs:** The result of such blind checking is simply to introduce more software; hence more sources of things that could go wrong at execution time, hence the need for more checks, and so on ad infinitum.

Such blind and often redundant checking causes much of the complexity and unwieldiness that often characterizes software.

NOTE: Obtaining and guaranteeing reliability requires a more systematic approach. In particular, software elements should be considered as *implementations* meant to satisfy well-understood *specifications* (**Recall Lecture 1**), not as arbitrary executable texts. This is where the contract theory comes in.

3.2 Contracts: a Notion

```
my-task is
do
    subtask1 ;
    subtask2 ;
     ..............
    subtaskn ;
end
```

Assume `my-task`, program unit implementing a task to be performed at runtime. For each of these subtasks, you may either write the corresponding solution in line as part of the body of my-task, or rely on a call to another unit. The decision is a typical design trade-off: Too much calling causes fragmentation of the software text: too little results in overcomplex individual units.

Assume you decide to use a routine call for one of the subtasks, you may adopt an approach for the program in the client – supplier model as shown in the table below

Party	Obligation	Benefit
Client	Provides the requirements for the task to be performed by the supplier	Obtains the products generated by the supplier
Supplier	Has the duty of carrying out the task correctly and delivering the result.	Expects exact items it requires from the supplier. Does not concern itself with what the supplier does with the product.

Each party expects some benefits from the contract and is prepared to incur some obligations to obtain them. These benefits and obligations are documented in a contract document. The contract document protects both sides:

- It protects the client by specifying how much should be done: The client is entitled to receive a certain result.
- It protects the contractor by specifying how little is acceptable: The contractor must not be liable for failing to carry out tasks outside of the specified scope.

3.3 Assertions

If the execution of a certain task relies on a routine call to handle one of its subtasks, it is necessary to specify the relationship between the client (the caller) and the supplier (the called routine) as precisely as possible. The mechanisms for expressing such conditions are called assertions. Some assertions, called *preconditions* and *postconditions* apply to individual routines. Others, the *class invariants*, constrain all the routines of a given class and will be discussed later.

Recall Lecture 1, discussion on invariants, preconditions and postconditions. It is important to include the preconditions and postconditions as part of routine declarations.

```
routine-name (argument declarations) is
                    -- Header comment
     require
          Precondition
     do
          Routine body, ie. instructions
     ensure
          Postcondition
     end
```

An assertion is a list of Boolean expressions, separated by semicolons: here a semicolon is equivalent to a Boolean "and" but allows individual identification of the assertion clauses.

The precondition expresses requirements that any call must satisfy if it is to be correct; the postcondition expresses properties that are ensured in return by the execution of the call.

A missing precondition clause is equivalent to the clause **Require True**, and a missing postcondition to the clause **Ensure True**. The assertion **True** is the least committing of all possible assertions. Any possible state of the computation will satisfy it.

```
put-child (new: NODE) is
-- Add new to the children of current node
     require
          new != Void
     do
          . . . Insertion algorithm . . .
     ensure
          new.parent = Current;
          child-count = old child-count + 1
     end -- put-child
```

Consider a routine *put-child()* for adding a new child to a tree node *Current*. The child is accessible through a reference, which must be attached to an existing node object. The table below informally expresses the contract forced by *put-child()* on any potential caller.

Party	Obligation	Benefit
Client	Use as argument a reference, say *new* to an existing node object.	Get updated tree where the Current node has one more child than before; *new* now has *Current* as its parent.
Supplier	Insert new node as required.	No need to do anything if the argument is not attached to an object.

- A reference such as *new* is either void (not attached to any object) or attached to an object: In the first case, it equals the value Void. Here the precondition expresses that the reference new must not be void. The routine will appear in the text of a class describing trees, or tree nodes. This is why it does not need an argument representing the node to which the routine will add the reference new as a child; all routines of the class are relative to a typical tree node. the "current instance" of the class. In a specific call such as *somenode.put-child (x)*, the value before the period, here *some-node*, serves as the current instance.

In the text of the class, the predefined name *Current* serves, if necessary, to refer to the current instance. Here it is used in the postcondition. The notation Old child-count, appearing in the postcondition of *put-child()*, denotes the value of *child-count* as captured on entry to a particular call. In other words, the second clause of the postcondition expresses that the routine must increase child-count by one. The construct Old may appear only in a routine postcondition.

3.4 Role of Assertions

The question of what happens if the conditions in assertions are not met during runtime is not the concern of the assertions themselves. As such, assertions serve only to ensure software is reliable. What happens if they fail to work can only be answered by whether assertions are monitored during runtime by a programmer or not.

Assertions do not describe special but expected cases that call for special treatment. In other words, the above assertions are not a way to describe (for example) the handling of void arguments to `put-child()`. If we wanted to treat void arguments as an acceptable (although) special case, we would handle it not through assertions but through standard conditional control structures as shown below:

```
if new = void then
     …..Take care of special case…..
else
     …..Take care of standard case….
end
```

Assertions are ways to describe the conditions on which software elements will work, and the conditions they will achieve in return. By putting the condition `new! = Void` in the precondition, we make it part of the routine's specification; the last form shown (with the If) would mean that we have changed that specification, broadening it to include the special case `new = Void` as acceptable. Any runtime violation of an assertion is not a special case but always the manifestation of a ***software bug*** i.e.

- Precondition violation indicates a bug in the client; the caller did not observe the conditions imposed on correct calls.
- Postcondition violation is a bug in the supplier, the routine failed to deliver on its promise

NOTE: if precondition is not satisfied, the routine is not bound to do anything, this means that using the defensive programming version (with the if statements) defeats the purpose of having preconditions. Therefore you should either have the conditions in the **Require** or you have it in an **if instruction** in the body of the routine but never in both.

This principle is the exact opposite of the idea of ***defensive programming***, since it directs programmers to avoid redundant tests. Such an approach is possible and fruitful because the use of assertions encourages writing software to spell out the consistency conditions that could go wrong at runtime. If the contract is precise and explicit, there is no need for redundant checks.

The matter of who should deal with abnormal values is essentially a pragmatic decision about division of labor: The best solution is the one that achieves the simplest architecture. If every routine and caller checked for every possible call error, routines would never perform any useful work.

> *In many existing programs, one can hardly find the islands of useful processing in oceans of error-checking code. In the absence of assertions, defensive programming may be the only reasonable approach. But with techniques for defining precisely each party's responsibility, as provided by assertions, such redundancy (so harmful to the consistency and simplicity of the structure) is not needed.*

Who Should Check Assertions (Suppliers or Clients)?

The rejection of defensive programming means that the client and supplier are not both held responsible for a consistency condition. Either the condition is part of the precondition and must be guaranteed by the client, or it is not stated in the precondition and must be handled by the supplier. Which of these two solutions should be chosen? There is no absolute rule; several styles of writing routines are possible, ranging from "**demanding**" ones where the precondition is strong (putting the responsibility on clients) to "**tolerant**" ones where it is weak (increasing the routine's burden). Choosing between them is to a certain extent a matter of personal preference; again, the key criterion is to maximize the overall simplicity of the architecture.

Client programmers do not expect miracles. As long as the conditions on the use of a routine make sense, and the routine's documentation states these conditions (*the contract*) explicitly, the programmers will be able to use the routine properly by observing their part of the deal. One objection to this style is that it seems to force every client to make the same checks, corresponding to the precondition, and thus results in unnecessary and damaging repetitions. But this argument is not justified:

- The presence of a preconditions in a routine *r* does not necessarily mean that every call must test for *p*, as in

```
if x.p then
        x.r
else
        ...Special Treatment...
end
```

  What the precondition means is that the client must guarantee property *p*; this is not the same as testing for this condition before each call. If the context of the call implies p, then there is no need for such a test. A typical scheme is

```
        x.s; x.r
```

  Where the postcondition of *s* implies *p*.

- Assume that many clients will indeed need to check for the precondition. Then what matters is the "Special Treatment." It is either the same for all calls or specific to each call. If it is the same, causing undue repetition in various clients, this is simply the sign of a poor class interface design, using an overly demanding contract for *r*. The contract should be renegotiated and made broader (more tolerant) to include the standard Special Treatment as part of the routine's specification.

- If, however, the Special Treatment is different for various clients, then the need for each client to perform its own individual test for *p* is intrinsic and not a consequence of the design method suggested here. These tests would have to be included anyway.

The last case corresponds to the frequent situation in which a supplier simply lacks the proper context to handle abnormal cases. For example, it is impossible for a general-purpose *STACK* module to know what to do when requested to *pop()* an element from an empty stack. Only the client - a module from a compiler or other system that uses stacks - has the needed information.

3.5 Class Invariants

Routine preconditions and postconditions may be used in non-object-oriented approaches, although they fit particularly well with the object-oriented method. Invariants, the next major use of assertions, are inconceivable outside of the object-oriented approach. A class invariant is a property that applies to all instances of the class, transcending particular routines. For example, the invariant of a class describing nodes of a binary tree could be of the form shown below, stating that the parent of both the left and right children of a node, if these children exist, is the node itself. In a class declaration, the invariant close is placed at after the attributes and routine declarations.

```
invariant
        left /= void implies(left.parent=Current);
        right /= void implies(right.parent = Current);
```

Two properties characterize a class invariant:

- The invariant must be satisfied after the creation of every instance of the class (every binary tree in this example). This means that every creation procedure of the class must yield an object satisfying the invariant. (A class may have one or more creation procedures, which serve to initialize objects. The creation procedure to be called in any given case is specified in the creation instruction.)
- The invariant must be preserved by every exported routine of the class (that is to say, every routine available to clients). Any such routine must guarantee that the invariant is satisfied on exit if it was satisfied on entry.

3.6 Contracting and Inheritance

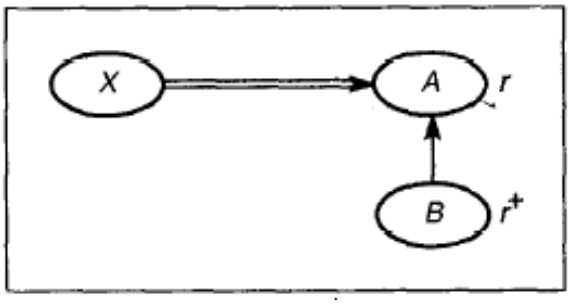
Contract theory is a better understanding and control of the fundamental object oriented notion of inheritance and of the key associated techniques: **redeclaration**, **polymorphism**, and **dynamic binding**. Through inheritance, you can define new classes by combining previous ones. A class that inherits from another has all the features (routines and attributes) defined in that class, plus its own. But it is not required to retain the exact form of inherited features: It may redeclare them to change their specification, their implementation, or both. This flexibility of the inheritance mechanism is central to the power of the object-oriented method.

For example, a binary tree class could provide a default representation and the corresponding implementations for search and insertion operations. A descendant of that class may provide a representation that is specifically adapted to certain cases (such as almost full binary trees) and redeclare the routines accordingly. Such a form of redeclaration is called a *redefinition or overriding*. It assumes that the inherited routine already had an implementation (**Consider inherited and redefined methods of an abstract class**).

The other form of redeclaration, called *effecting*, applies to features for which the inherited version, known as a *deferred* (or abstract) feature, had no implementation, but only a specification. The effecting then provides an implementation (making the feature effective, the reverse of deferred). The subsequent discussion applies to both forms of redeclaration, although for simplicity it concentrates on redefinition. Redeclaration takes its full power thanks to *polymorphism and dynamic binding*. Polymorphism is type adaptation controlled by inheritance. More concretely, this means that if you have b of type `BINARY-TREE` and `sb` of type

SPECIAL-BINARY-TREE, the latter class a descendant of the former, then the assignment b:=sb is permitted, allowing b to become attached at runtime to instances of SPECIAL-BINARY-TREE, of a more specialized form than the declaration of b specifies. Of course, this is only possible if the inheritance relation holds between the two classes as indicated.

What happens then for a call of the form t.insert (v) which applies procedure insert, with argument v, to the object attached to t? Dynamic binding means that such a call always uses the appropriate version of the procedure - the original one if the object to which t is attached is an instance of BINARY-TREE, the redefined version if it is an instance of SPECIAL-BINARY-TREE. The reverse policy, static binding (using the declaration of b to make the choice), would be an absurdity: deliberately choosing the wrong version of an operation. The combination of inheritance, redeclaration, polymorphism, and dynamic binding yields much of the power an flexibility that result from the use of the object-oriented approach.: Yet these techniques may also raise concerns of possible misuse: What is to prevent a redeclaration from producing an effect that is incompatible with the semantics of the original version -fooling clients in a particularly bad way, especially in the context of dynamic binding? Nothing of course.



No design technique is immune to misuse. But at least it is possible to help serious designers use the technique properly; here the contract theory provides the proper perspective. What redeclaration and dynamic binding mean is the ability to subcontract a task; preventing misuse then means guaranteeing that subcontractors honor the prime contractor's promises in the original contract. Consider the situation described in the figure above. A exports a routine r to its clients. (For simplicity, we ignore any arguments to r.) A client X executes a call u.r where u is declared of type A. Now B, a descendant of A, redeclares r. Through polymorphism, u may well become attached to an instance of B rather than A. Note that often there is no way to know this from the text of X alone; for example, the call just shown could be in a routine of X beginning with

```
some-routine (u: A) is...
```

where the polymorphism only results from a call of the form

```
z.some-routine (v)
```

for which the actual argument v is of type B. If this last call is in a class other than X, the author of X does not even know that u may become attached to an instance of B. In fact, he may not even know about the existence of a class B. But then the danger is clear. To ascertain the properties of the call u.r, the author of X can only look at the contract for r in A. Yet, because

of dynamic binding, A may subcontract the execution of r to B, and it is B's contract that will be applied.

How do you avoid "fooling" X in the process? There are two ways B could violate its prime contractor's promises:

- B could make the precondition stronger, raising the risk that some calls that are correct from x's viewpoint (they satisfy the original client obligations) will not be handled properly.

- B could make the postcondition weaker, returning a result less favorable than what has been promised to X.

None of this, then, is permitted. But the reverse changes are of course legitimate. A redeclaration may weaken the original's precondition or it may strengthen the postcondition. Changes of either kind mean that the subcontractor does a better job than the original contractor-which there is no reason to prohibit.

These rules illuminate some of the fundamental properties of inheritance, redeclaration, polymorphism, and dynamic binding. Redeclaration, for all the power it brings to software development is not a way to turn a routine into something completely different. The new version must remain compatible with the original specification although it may improve on it. The noted rules express this precisely. These rules must be enforced by the language. Eiffel uses a simple convention. In a redeclaration, it is not permitted to use the forms require... and ensure....
The absence of a precondition or postcondition clause means that the redeclared version retains the original version's assertion. Since this is the most frequent situation, the class author is not required to write anything special in this case.

Other areas Affected by Contract Programming

Contract programming has immediate application in *Dynamic binding* and *Exception handling*. It is not hard to devise an exception mechanism that directly supports the preceding method for handling abnormal cases. A "rescue" clause, which expresses the alternate behavior of the routine should be provided with every subroutine definitions. When a routine includes a rescue clause, any exception occurring during the routine's execution interrupts the execution of the body (the Do clause) and starts execution of the rescue clause.

The clause contains zero or more instructions, one of which may be a Retry. The execution terminates in either of two ways:

- If the rescue clause terminates without executing a Retry, the routine fails. It reports failure to its caller by triggering a new exception. This is the organized panic case.
- If the rescue clause executes a Retry, the body of the routine (Do clause) is executed again.

NB: Exception handling and Dynamic bind are two well defined Object Oriented Programming techniques Assumed in this course to have been covered in ICS 2104 and ICS 2201