

Attendance/Demo

To receive credit for this lab, you must make reasonable progress towards completing the exercises. When you have finished all the exercises, call your instructor or a TA, who will review your answers. For those who don't finish early, the TA will ask you to show whatever you have completed, starting at about 15 minutes before the end of the lab period. Finish any exercises that you don't complete on your own time.

An introduction to Design by Contract (DbC)

When specifying software, and more specifically software-to-software communications, we assign responsibilities to classes under the form of operations that communicate with (i.e., call) each other.

This is formalized by Design by Contract with the notions of preconditions and postconditions, which specify the rights and responsibilities of interacting operations.

A precondition expresses the constraints under which an operation will function properly, whereas a postcondition expresses properties of the state of the software resulting from the operation's execution. In other words, when specifying a method's precondition and a method's postcondition:

- the precondition binds the client, i.e., it defines the responsibility of any (other) operation calling this operation. At the same time, the precondition defines the rights of the operation being called, i.e., it has the right to expect that the condition(s) stated in its precondition hold;
- the postcondition binds the method being specified, i.e., it defines the conditions that must be ensured by this operation at the end of its execution. At the same time, the postcondition defines the rights of the calling operation, i.e., it has the right to expect that, assuming the condition(s) stated in the precondition of the called operation were satisfied, the conditions stated in the called operation's postcondition hold.

These rights and responsibilities can be summarized (pictured) in the following table.

	Obligations/Responsibilities	Benefits/Rights
Operation performing the call	To satisfy the precondition of the operation being called.	Expects/relies on what is stated in the postcondition of the operation being called
Operation being called	To ensure conditions stated in the postcondition hold.	Can assume it is called while its precondition holds.

Important consequence: because the precondition of an operation is the responsibility of the method performing the call and the right of the method being called, performing the call while the precondition does not hold may result in unexpected results and even failure (exception). Indeed, since it is the responsibility of the calling method to check that the precondition holds before performing the call, the method being called has the right to not check that it has been called under the expected conditions (e.g., an input parameter might be out of expected bounds), and as a consequence the method being called is not bound by the contract which is to ensure its postcondition holds. Re-read the definition of a postcondition: the method being called has to ensure its postcondition holds, assuming it has been called under expected conditions (its precondition).

To illustrate these notions, consider class `Stack` and its operation `put (x)` that add `x` at the top of the stack. The precondition and postcondition of operation `put (x)` can be stated as follows (each line specifies a condition; when there is more than one line for either the precondition or the postcondition, the semantics is that all conditions must hold together, i.e., this is a conjunction):

Precondition: the stack is not full
 Postcondition: the stack is not empty
 `x` has been added to the top of the stack
 the number of elements in the stack has increased by one.

Similarly, the precondition and postcondition of operation `remove ()` of class `stack`, which removes (and returns) the element at the top of the stack, are:

Precondition: the stack is not empty
 Postcondition: the returned value was the one at the top of the stack before the call to `remove ()`
 the number of elements in the stack has been decreased by one.

A more precise definition of those operations' precondition and postcondition (primarily the postconditions) could be the following.

<code>put (x)</code>	precondition:	the stack is not full
	postcondition:	the stack is not empty <code>x</code> has been added to the top of the stack assuming that before the call, the number of elements in the stack was one less than the maximum, now the stack is full assuming that before the call, the number of elements in the stack was strictly less than the maximum, now the stack is partially full.
<code>remove ()</code>	precondition:	the stack is not empty
	postcondition:	the returned value was the one at the top of the stack before the call to <code>remove ()</code> the number of elements in the stack has decreased by one assuming that before the call, the stack had only one element, now the stack is empty assuming that before the call, the stack was full, now the stack is partially full.

Since a stack is either empty, partially full, or full, the preconditions of these two methods could also be: the stack is empty or partially full (`put`); the stack full or partially full (`remove`).

Also, the postcondition could state that some things have not changed. Specifically, the postcondition of `put (x)` could state that the elements that were in the stack before the call, and their order in the stack, have not changed; the postcondition of `remove ()` could state that beside the last element of the stack (when considering the contents of the stack before the call), all the elements that were in the stack before the call, and their order in the stack, have not changed. Specifying such conditions, about things that do not change, might be considered superfluous. This might be important though in some cases. Such conditions are called *frame conditions*.

Specifying your own classes with DbC

Part I—continuing with class Stack

Class Stack has three other operations, quickly specified below. Specify their preconditions and postconditions.

- | | |
|------------------------|---|
| <code>peek()</code> | simply returns the element at the top of the stack without removing it. |
| <code>isEmpty()</code> | returns true if the stack has no element, false otherwise. |
| <code>isFull()</code> | returns true if the stack contains the maximum number of elements allowed, false otherwise. |

Note: Sometime an operation has no right, in which case the precondition is empty. In other words, it can accept any input, it can be called in any state. When this is the case, it is important to specify the precondition is empty anyway. Otherwise, this may be considered an omission.

Part II—a different class

Consider class Set, which implements the notion of set (from set theory), with the following operations:

`insert(x)`, `remove(x)`, and `isIn(x)`.

Specify the preconditions and postconditions of these operations.

(Recall that a set does not have duplicate elements.)

Part III—yet another class

Consider banking software with classes Account and AccountOperations.

Class account has the following operations:

- `getBalance()` returns the balance of the account (a real number).
- `setBalance(real b)` sets the balance of the account to the value of the parameter.
- `getCustomerName()` returns the name (string) of the owner of the account.

Class AccountOperations has the following operations:

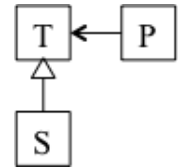
- `withdraw(Account acc, Real am)` withdraws amount `am` from account `acc`.
- `balance(Account acc)` returns the balance of account `acc`.
- `transfer(Account from, Account to, Real am)` transfers amount `am` from the first account to the second account, providing that the two accounts have the same owner.
- `deposit(Account acc, Real am)` deposits amount `am` to account `acc`.

What are the preconditions and postconditions of the operations of these two classes?

An introduction to subtyping

One of the advantages of the generalization (i.e., inheritance) relationship between classes is to facilitate reuse (the child class can reuse the data and behaviour specified in the parent class) as well as ease extensibility (one can extend the design/code without too much impact on what already exist). In this lab session, we will look at the latter use of generalization.

Consider this simple class diagram where class T is a parent class for S and at the same time provides services (server) to (client) class P. When generalization is used for extensibility purposes, we mean that it should be possible to extend T with class S (or any other class U) without changing P. In other word, P is oblivious to the actual subclass it might be using: either T, S or U. *Subtyping* is a design principle that will help us ensure P can use anything that “looks like” T, i.e., either T or any subclass of T, without “knowing” exactly which (sub)class it is actually using. This is called the *Liskov substitution principle*. We will say that when the Liskov substitution principle holds then S and U are subtypes of T. If the principle does not hold, we will simply say that S and U are subclasses.



Consider that class T has an operation `m(int i):int` that takes an integer parameter `i` as input and returns an integer value. Suppose that the precondition of this method states that parameter `i` should be between 1 and 10, and that the postcondition of this method states that the return value should not be equal to zero. These will be known by P (the client), that will have to ensure this precondition holds before calling `m`, and will rely on the return value to be different from zero.

Consider that class T is extended by class S, and that class S redefines method `m(int i):int`. Method `m` is redefined by S in such a way that it now expects (precondition) that the input parameter `i` is different from 5, and ensures (postcondition) it can return any integer value.

Consider that class T is extended by class U, and that class U redefines method `m(int i):int`. Method `m` is redefined by U in such a way that it now expects (precondition) that the input parameter `i` is in the range 0..10, and ensures (postcondition) it can return strictly positive integer value.

Considering these redefinitions, the question is then to decide whether S and U are subtypes of T, according to the definition of the Liskov substitution principle, or whether they are simply subclasses of T.

This leads to the following question: Given that P knows the contract it has with T, that is the precondition and the postcondition of `m()` as provided by (specified in) T, can T be substituted by an S or an U, without P knowing it (oblivious). In other words, if P is interacting with an S (or a U) will the system work?

Since P knows about the precondition of `m()` in T, it will ensure that this method precondition holds, that is it will call `m()` with a parameter value between 1 and 10, say 5. If P is interacting with an S instead of a T without knowing it, i.e., without having a behaviour specific to the interaction with S that would be different from its interaction with T, then P is breaking the precondition of `m()` in S (which requires an input different from 5). In order for this scenario to work, P would need to know whether it is interacting with an S rather than a T in order to perform the right call, with the right value. P should not be oblivious. So the Liskov principle does not hold: T cannot be substituted with an S without P knowing.

Suppose we fix `m()` in S, which now can accept (precondition) any value between 1 and 20. (The postcondition of `m()` in S discussed earlier remains the same.) P, that knows about T, will ensure a call to `m()` is performed with an input parameter between 1 and 10. Since any value between 1 and 10 is also between 1 and 20, the call satisfies the precondition of `m()` in S. This is reassuring but not the end of the analysis as we need to also consider the postcondition. Since the call is performed on an S, `m()` will return any integer value, i.e., a value that can be equal to zero. However, P will rely on the fact that the return value is different from zero: e.g., given the specification of `m()` in T, P may divide something by the value returned by `m()`, resulting in an exception. In other words, `m()` in S does not ensure the postcondition expected by P. P would need to behave differently

(regarding the value returned by $m()$) depending on whether it is interacting with an T or an S. P is not oblivious and the Liskov principle does not hold.

Let us turn our attention to U. Since P knows about T, it will call $m()$ on a U with a value between 1 and 10. Since any value between 1 and 10 is also between 0 and 10 (the precondition of $m()$ in U), the call performed by P satisfies the precondition of $m()$ in U. Executing $m()$ in U will ensure that the return value is strictly positive (postcondition of $m()$ in U). Since a strictly positive value is a non-null value (precondition of $m()$ in T) and this is what P expects, the call on $m()$ in U satisfies the postcondition P relies on. In other words, a T can be substituted with a U without P knowing it. The Liskov substitution principle holds.

These are only examples and we need to generalize situations whereby the Liskov substitution principle holds (or not). It is all about the logical relation that exist between the precondition of the method in the parent (T) and the precondition of the method in the child (S or U), and the relation that exist between the postcondition of the method in the parent (T) and the postcondition of the method in the child (S or U).

In order for the Liskov substitution principle to hold the following must be true:

- The precondition of the method in the parent must logically imply the precondition in the child. We say that the precondition in the child is weakened (or weaker) since it can accept more than what the parent method accepts, but not less.
- The postcondition of the method in the child must logically imply the postcondition in the parent. We say that the postcondition in the child is strengthened (or stronger) since it does not ensure more things than the one of the parent.

Let us go back to our example. The precondition of $m()$ in T was: $i \in [1, 10]$. The precondition of the first version of $m()$ in S was: $i < 5$. It is clear that $i \in [1, 10] \Rightarrow i < 5$ is not logically true. If we turn our attention to the second version of $m()$ in S, and the version of $m()$ in U, we have: $i \in [1, 10] \Rightarrow i \in [1, 20]$ (second version of $m()$ in S) and $i \in [1, 10] \Rightarrow i \in [0, 10]$ ($m()$ in U). With respect to postconditions, we have: $\text{return} \in [-\infty, +\infty] \Rightarrow \text{return} < 0$ is not logically true, but $\text{return} > 0 \Rightarrow \text{return} < 0$ is logically true.

Checking for the Liskov substitution principle.

Part IV—Classes IntSet and MaxIntSet

Consider the following two classes. Is MaxIntSet a subtype of IntSet? Justify your answer.

<pre>public class IntSet { private Vector els; // the elements // Post: Initializes this to be empty public IntSet() {...} // Post: Adds x to the elements of this public void insert (int x) {...} // Post: Remove x from the elements of // this public void remove (int x) {...} // Post: If x is in this returns true // else returns false public boolean isIn (int x) {...} // Post: Returns the cardinality of // this public int size () {...} // Post: Returns true if this is a // subset of s else returns false public boolean subset (IntSet s) {...} }</pre>	<pre>public class MaxIntSet extends IntSet { // biggest element if set not empty private int biggest; // call super() public MaxIntSet () {...} // new method public int max () throws EmptyException {...} // overrides IntSet::insert() // Additional Post: update biggest with // x if x > biggest public void insert (int x) {...} // overrides IntSet::remove() // Additional Post: update biggest with // next biggest element in this if // x = biggest public void remove (int x) {...} }</pre>
--	--

Part V—Classes LinkedList and Set

Consider the following two classes. Is Set a subtype of LinkedList? Justify your answer.

(old.getLength() refers to the value of getLength() at the beginning of the method execution.)

<pre>public class LinkedList { ... // Adds an element to the end of the // list /** PRE: element != null /** POST: this.getLength() == // old.getLength() + 1 /** && this.contains(element) == // true public void addElement(Object element) { ... } ... }</pre>	<pre>public class Set extends LinkedList { ... // Adds element, provided element is // not already in the set /** PRE: element != null // && this.contains(element) == // false /** POST: this.getLength() == // old.getLength() + 1 // && this.contains(element) == // true public void addElement(Object element) { ... } ... }</pre>
---	---

Part V—Classes Circle and Ellipse

Suppose you have two classes named Circle and Ellipse, which represent two forms that are close to one another. In essence, an ellipse is defined by a width and a height, and a circle is an ellipse with identical width and height. Suppose class Ellipse has a setSize(x, y) operation to set the ellipse's width (x) and height (y). Can Circle inherit from Ellipse (by redefining setSize) and be considered a subtype of Ellipse?

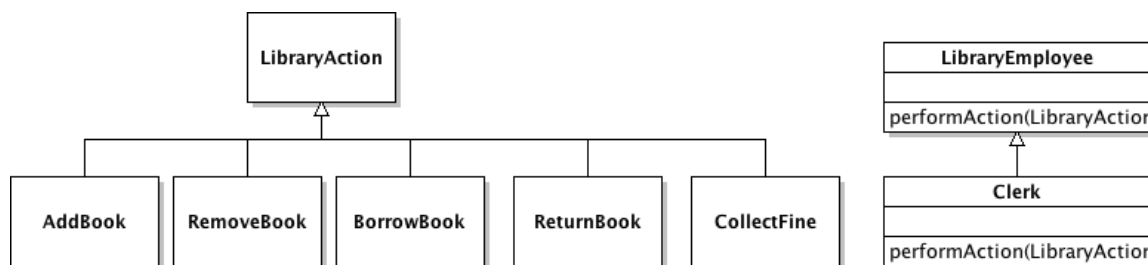
Part VI—Classes Stack and Vector

In the Java Development Kit (<http://docs.oracle.com/javase/6/docs/api/java/util/Stack.html>), class Stack extends (inherits from) class Vector. Without considering actual operations' preconditions and postconditions but simply relying on the substitutability principle discussed earlier, can you argue whether Stack is a subtype of Vector or not?

Part VII—Classes LibraryEmployee and Clerk

Consider the following excerpt of the class diagram of a library system that specifies a taxonomy of actions that can be performed at the library: adding/removing a book, borrowing/returning a book, collecting customer fines; and two kinds of library employees. The operation performAction in class LibraryEmployee specifies that any kind of LibraryAction can be used as a parameter. The redefined operation performAction in class Clerk specifies that a clerk can only perform actions BorrowBook, ReturnBook, and CollectFine.

Is Clerk a subtype of LibraryEmployee? Justify.



Part VIII—Classes Person and Employee

Suppose you have the following two classes (excerpts). Is Employee a subtype of Person? Justify.

<pre> public class Person { private String firstName; private String lastName; public String getName() { return firstName + lastName; } } </pre>	<pre> public class Employee extends Person{ public String getName() { return firstName + ":" + lastName; } } </pre>
---	---

Part IX—Different abstract situations

Consider parent class X and child class Y, both defining operation myOp (...).

Consider the following alternative precondition and postcondition specifications for operation myOp (...) in these two classes. Indicate whether the Liskov substitution principle (LSP) holds or not, i.e. whether the rule on the preconditions holds, the rule of the postcondition holds, or both rules holds. Each case is independent from the other ones.

				Precondition rule holds	Postcondition rule holds	LSP holds
1	X myOp(...)	Pre	Parameters satisfy condition C			
		Post	Return value satisfies condition D			
	Y.myOp(...)	Pre	Parameters satisfy condition C			
		Post	Return value satisfies condition D			
2	X myOp(...)	Pre	Parameters satisfy conditions A and B			
		Post	Return value satisfies condition E			
	Y.myOp(...)	Pre	Parameters satisfy conditions B			
		Post	Return value satisfies condition E			
3	X myOp(...)	Pre	Parameters satisfy condition F			
		Post	The resulting state satisfies I or J			
	Y.myOp(...)	Pre	Parameters satisfy conditions F and G			
		Post	The resulting state satisfies I			
4	X myOp(...)	Pre	Parameters and starting state satisfy condition M			
		Post	The resulting state satisfies condition N			
	Y.myOp(...)	Pre	Parameters and starting state satisfy condition M			
		Post	The resulting state satisfies conditions N and O			
5	X myOp(...)	Pre	Parameters satisfy condition Q			
		Post	Resulting state satisfies condition L			
	Y.myOp(...)	Pre	Parameters satisfy condition Q and starting state satisfy condition R			
		Post	Return value satisfies condition K and resulting state satisfies condition L			