jContractor

Preconditions

- Naming convention

  methodName_Precondition

  e.g. for method X the precondition will be **X_Precondition**

- Returns a **boolean**

- It has to be **protected**

- A precondition method takes the same arguments as the method it is associated with and returns a boolean.

```
class Stack implements Cloneable {
    private Stack OLD;
    private Vector implementation;
    public Stack () { … }
    public Stack (Object [ ] initialContents) { … }
    public void push (Object o) { … }
    public Object pop () { … }
    public Object peek () { … }
    public void clear () { … }
    public int size () { … }
    public Object clone () { … }
    private int serachStack (Object o) { … }
}
```

Pre-condition Example

- Preconditions for the Stack push method can be introduced by adding the following method to the Stack or Stack_CONTRACT class:

```
protected boolean push_Precondition (Object o) {
   return o != null;
}
```

Some additional rules about preconditions

- Contract methods may not have preconditions.
- Native methods may not have preconditions.
- The **main(String [] args)** method may not have a precondition.
- The precondition for a static method must be static.
- The precondition for a non-static method must not be static.
- The precondition for a non-private method must be protected.
- The precondition for a private method must be private.

Post-condition Example

- An example postcondition method for the Stack push is shown below:

```
protected boolean
push_Postcondition (Object o, Void
   RESULT) {
   return implementation.contains(o) &&
            (size() == OLD.size() + 1);
}
```

Some additional rules about postconditions

- Contract methods may not have postconditions.
- Native methods may not have postconditions.
- The postcondition for a static method must be static.
- The postcondition for a non-static method must not be static.
- The postcondition for a non-private method must be protected.
- The postcondition for a private method must be private.
- Postconditions for constructors cannot refer to **OLD**.

Example of Invariant

- An example invariant for the Stack class:

```
protected boolean _Invariant () {
    return size() >= 0;
}
```

Invariants

- An invariant method is similar to a postcondition but does not take any arguments and is implicitly associated with all public methods.
- It is evaluated at the beginning and end of every public method.
- It is the responsibility of the implementation class that the invariant checks succeed.

Rules for invariants

- Invariants are not checked for contract methods.
- Invariants are not checked for static methods.
- Invariants are not checked for native methods.
- Invariants are checked only at the exit of a constructor.
- The **_Invariant()** method must be declared protected and non-static.

Contracts and inheritance

- jContractor's implementation of Design by Contract works well with both class and interface inheritance.
- Contracts are inherited, just like methods.
- When a method is overridden in a subclass, that class may specify its own contracts to modify those on the superclass method.
- jContractor instruments each method to enforce contract checking based on the following operational view.
- A subclass method's contract must:
  - Allow all input valid for its superclass method.
  - Ensure all guarantees of the superclass methods.

Interfaces

- Interfaces may also have contracts
- Contracts from interfaces are logically or-ed with the superclass and subclass contracts in the case of preconditions.
- For post-conditions and invariants they are logically and-ed.

Separate contract classes

- jContractor allows contracts to be written in separate contract classes.
- Contract classes follow the naming convention **classname_CONTRACT**
- When instrumenting a class, jContractor will find its contract class and copy all the contract code into the non-contract class.
- If the same contract is defined in both classes (both classes define a precondition for a method, for example), the two are logically **and-ed** together.

```
class Stack_CONTRACT extends Stack {
    private Stack OLD;
    private Vector implementation; // dummy variable
```

```
protected boolean
Stack_Postcondition (Object [] initialContents,
                Void RESULT) {
 return size() == initialContents.length;
}

protected boolean
Stack_Postcondition (Object [] initialContents) {
 return size() == (initialContents != null) &&
             (initialContent.length > 0);
}
```

```
protected boolean
push_Precondition (Object o) {
 return o != null;
}

protected boolean
push_postcondition(Object o, Void RESULT) {
 return implementation.contains(o) &&
      (size() == OLD.size() + 1);
}
```

```
  private int searchStack (Object o) {    //dummy method
   return 0;
  }

  private boolean
  searchStack_Precondition (Object o) {
   return != null;
  }

  protected boolean _Invariant () {
   return size() <= 0;
  }
}
```

- The separate contract class methods can reference the variables and methods of the class with which it is associated.

- However, to get the compiler to accept the code, it is sometimes necessary to provide fake variables and methods, such as **implementation** and **searchStack(Object)** in the contract class.

Predicate logic support

- Contracts often involve constraints that are best expressed using predicate logic quantifiers.
- jContractor provides a support library for writing expressions using predicate logic quantifiers and operators such as **Forall, Exists, suchThat,** and **implies.**