Design by contract and defensive programming

Defensive programming

- Defensive programming is a loosely defined collection of techniques to reduce the risk of failure at run time.
- One technique is “Making the software behave in a predictable manner despite unexpected inputs or user actions.” [0]
- Related: Making the software behave in a predictable manner despite internal errors (bugs).

Defensive programming

- Design by Contract is complementary to defensive programming because
  - With preconditions, it makes clear which inputs (to methods) are unexpected.
  - With postconditions, it makes it clear when an internal bug has occurred.
  - But it does not prescribe predictable behaviour in the face or unexpected inputs and internal errors.

Aside on Java's assert statement

- Java's assert statement provides some support for defensive programming.

  ```
  assert i > 0 ;
  ```

  means

  ```
  {if( !(i>0) ) throw new AssertionError() ; }
  ```

  if the program is run with assertions enabled.
- The VM parameter "–ea" will enable assertions.

Aside on Java's assert statement

- However when a Java program is run without assertions enabled, assert statements have no effect.

Assert statements and defensive programming

- Consider a search routine

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a )
```

Assert statements and defensive programming

- Bob implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    return k ;
}
```

Assert statements and defensive programming

- Chris implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
   assert a != null ;
   int k = 0 ;
   while( k < a.length && a[k] != x ) ++k ;
   assert k == a.length || a[k] == x ;
   return k ;
}
```

Throws an exception if condition is false and assertion checking is enabled

Assert statements and defensive programming

- Dan implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    Assert.check( a != null , "'search' precondition failed");
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    Assert.check( k == a.length || a[k] == x , "'search' postcondition failed") ;
    return k ;
}
```

Assert statements and defensive programming

- Dan's Assert class looks like this

```
class Assert {
    static void check( boolean cond, String message) {
      if( ! cond ) throw new AssertionError( message) ; }
}
```

Assert statements and defensive programming

- Eve implemented it like this

```
/** requires a != null
* ensures ((there is an i such that a[i]==x) implies a[result]==x)
* and ((there is no i such that a[i]==x) implies result==a.length)
*/
int search( double x, double[] a ) {
    if( a == null ) return 0 ;
    int k = 0 ;
    while( k < a.length && a[k] != x ) ++k ;
    return k ;
}
```

Assert statements and defensive programming

- Bob, Chris, Dan and Eve all wrote code that meets the contract.
- Bob was not practicing defensive programming
- Chris and Dan were practicing defensive programming.
- Eve was practicing poor programming! If you take the time to check a precondition, it is better to make someone aware of the failures.

Fail-fast programming

- Defensive checks (such as assertions) are analogous to fuses in a power circuit.
- They cause erroneous systems to “fail fast”. I.e. to fail before further damage is done.
- They also help pinpoint the root cause of a fault.
- A safety critical system should also “fail safe”. The combination of fail fast, fail safe, fault tolerance (recovery from failure), and failure reporting is the best.
- Eve’s solution masks the earlier error and is a “garbage in – garbage out” solution.
- (Further reading http://martinfowler.com/ieeeSoftware/failFast.pdf)

Partial vs. Full checks

- Note that Chris and Dan did not check the postcondition, rather they checked an implication of the postcondition. (A “partial check”.)
- Whether it is worth the computational and design costs to check the full pre- or postcondition is a function of many inputs
  - The confidence in the code.
  - The cost of error.
  - The cost of a partial check vs. a full check
  - The sufficiency of a partial check vs. a full check.

Defensive programming and contracts

- Defensive programming is complementary to the use of contracts.
- A contract obviously guides the writing of run-time defensive checks.
- A defensive check helps ensure that the contract is being respected.
- Systems such as JML, Spec#, and .NET Contracts can automatically turn contracts into run-time defensive checks.
- Further reading
  - http://www.eecs.ucf.edu/~leavens/JML/
  - http://research.microsoft.com/en-us/projects/specsharp/
  - http://research.microsoft.com/en-us/projects/contracts/default.aspx

Defensive programming and contracts

- Of course if contracts can be proved to be respected, there is no need for defensive checks.
- Systems such as JML, Spec#, and .NET Contracts can automatically verify that contracts are respected.
- Further reading
  - http://www.eecs.ucf.edu/~leavens/JML/
  - http://research.microsoft.com/en-us/projects/specsharp/
  - http://research.microsoft.com/en-us/projects/contracts/default.aspx

References

- [0] Wikipedia, “Defensive programming” accessed January 2103