The University of Sydney
School of Information Technologies

Software Quality Assurance: SOFT3302

Tutorial — Week 5

Objectives

This tutorial gives experience with the design of tests based on white-box testing methods. By the end of this tutorial you should have an understanding of the correspondence between an implementation and a graph representing the implementation, which may then be subject to path analysis; you should also have experience in developing test cases based on the testing paths. You should also have an understanding of coverage metrics and what they represent.

Prework

The SUT is the code attached in the appendices that inserts a value into a hash table.

Note: one would almost never write one's own code to do this since it's a prime way to introduce bugs into a programme. Python, C++ and Java all have hashing containers that have been written and debugged already.

Draw a flow graph from the code and label the nodes in preparation for the tutorial.

Labwork

1. Develop test cases to provide 100% statement coverage.
2. Develop test cases to provide 100% branch coverage.
3. Develop test cases to provide 0-path and 1-path independent path (basis path) tests of the loop.
4. Note that all of the tests so far will not necessarily catch a certain fault with this code. (Can you see what it is?) Develop test cases that use Beizer's loop test criteria to test iterations of the loop.

Test cases should include the following information at a minimum.

Module/Component:				
Tested by:				
Version:				
Testing date:				
Test Id	Path	Description	Inputs	Outcome

Appendix

The following code samples purport to (although may not) implement the same algorithm to insert a value into a hash table. Hash collisions are handled by a linear search through the table for the next empty bin.

To repeat the admonition above: one almost never needs to implement one's own version of a hash table in the languages below. Use the supplied hashed containers in the accompanying libraries. Even if one were to implement a hash table, choosing linear search on collision is most likely a bad choice.

In the three languages below, only one is not amenable to resizing the array within the insert if it is deemed that the hash table is too densely populated. Which? Why?

Python

```
def insert(hashTable, v):
	bin = hash(v) % len(hashTable)
	while hashTable[bin] is not None:
		bin += 1
		if bin == len(hashTable):
			bin = 0
	hashTable[bin] = v
```

C++

```
template <typename T>
void
insert(vector<T *>& hashTable, T *v)
{
	// int hash(const T& v) is a template function defined elsewhere
	int bin = hash(*v) % hashTable.size();
	while (hashTable[bin])
		if (++bin == hashTable.size())
			bin = 0;
	hashTable[bin] = v;
}
```

Java

```
void
insert(Object[] hashTable, Object v)
{
	int bin = v.hashCode() % hashTable.length;
	while (bashTable[bin] != null)
		bin += 1;
		if (bin == hashTable.length)
			bin = 0;
	hashTable[bin] = v;
}
```