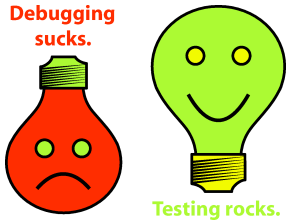

Testing on the Toilet

Testable Contracts Make Exceptional Neighbors

May 1, 2008

Consider the following function, which modifies a client-supplied object:

```
bool SomeCollection::GetObjects(vector<Object*>* objects) const {
  objects->clear();
  typedef vector<Object*>::const_iterator Iterator;
  for (Iterator i = collection_.begin(); i != collection_.end(); ++i) {
    if ((*i)->IsFubarred()) return false;
    objects->push_back(*i);
  }
  return true;
}
```

Consider when GetObjects() is called. What if the caller doesn't check the return value, and assumes the data is in a valid state when it actually is not? If the caller does check the return value, **what can it assume about the state of its objects in the failure case**? When GetObjects() fails, it would be much better if either all the objects were collected or none of them. This can help avoid introducing hard to find bugs.

By using good design contracts and a solid implementation, it is reasonably easy to make functions like GetObjects() behave like transactions. By following Sutter's rule of **modifying externally-visible state only after completing all operations which could possibly fail** [1], and mixing in Meyers's "swap trick" [2], we move from the realm of undefined behavior to what Abrahams defines as the strong guarantee [3]:

```
bool SomeCollection::GetObjects(vector<Object*>* objects) const {
  vector<Object*> known_good_objects;
  typedef vector<Object*>::const_iterator Iterator;
  for (Iterator i = collection_.begin(); i != collection_.end(); ++i) {
    if ((*i)->IsFubarred()) return false;
    known_good_objects->push_back(*i);
  }
  objects->swap(known_good_objects);
  return true;
}
```

At the cost of one temporary and a pointer swap, we've strengthened the contract of our interface such that, at best, the caller received a complete, new collection of valid objects; at worst, the state of the caller's objects remains unchanged. The caller might not verify the return value, but will not suffer from undefined results. This allows us to reason much more clearly about the program state, making it much easier to verify the intended outcome with automated tests as well as recreate, pinpoint, and banish bugs with regression tests.

1 http://www.gotw.ca/publications
2 Scott Meyers, *Effective C++*
3 http://www.boost.org/more/generic_exception_safety.html

More information, discussion, and archives:
http://googletesting.blogspot.com