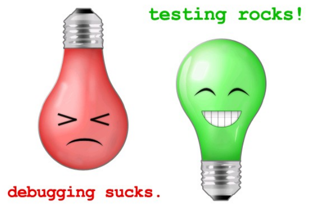

Testing on the Toilet

October 30, 2008

Contain Your Environment

Many modules must access elements of their environment that are too heavyweight for use in tests, for example, the file system or network. To keep tests lightweight, we mock out these elements. But what if no mockable interface is available, or the existing interfaces pull in extraneous dependencies? In such cases we can introduce a mediator interface that's directly associated with your module (usually as a public inner class). We call this mediator an "Env" (for environment); this name helps readers of your class recognize the purpose of this interface.

For example, consider a class that cleans the file system underlying a storage system:

```
// Deletes files that are no longer reachable via our storage system's metadata.
class FileCleaner {
 public:
  class Env {
   public:  // These methods return false or NULL on error.
    virtual bool MatchFiles(const char* pattern, vector<string>* filenames) = 0;
    virtual bool BulkDelete(const vector<string>& filenames) = 0;
    virtual MetadataReader* NewMetadataReader() = 0;  // factory method
    virtual ~Env();  // don't forget the virtual destructor!
  };
  // Constructs a FileCleaner.  Uses "env" to access files and metadata.
  FileCleaner(Env* env, QuotaManager* qm);  // non-Env args are fine, too!
  // Deletes files that are not reachable via metadata.  Returns true on success.
  bool CleanOnce();
};
```

FileCleaner::Env lets us test FileCleaner without accessing the real file system or metadata. It also makes it easy to **simulate various kinds of failures**, for example, of the file system:

```
class NoFileSystemEnv : public FileCleaner::Env {
  virtual bool MatchFiles(const char* pattern, vector<string>* filenames) {
    match_files_called_ = true;
    return false;  // failure
  }
  ...
};
TEST(FileCleanerTest, FileCleaningFailsWhenFileSystemFails) {
  NoFileSystemEnv* env = new NoFileSystemEnv();
  FileCleaner cleaner(env, new MockQuotaManager());  // takes ownership of "env"
  ASSERT_FALSE(cleaner.CleanOnce());
  ASSERT_TRUE(env->match_files_called_);
}
```

An Env object is particularly useful for **restricting access** to other modules or systems, for example, when those modules have overly-wide interfaces. This has the additional benefit of reducing your class's dependencies. However, be careful to **keep the "real" Env implementation simple**, lest you introduce hard-to-find bugs in the Env. The methods of your "real" Env implementation should just delegate to other, well-tested methods.

The most important benefits of an Env are that it **documents how your class accesses its environment** and it **encourages future modifications to your module to keep tests small** by extending and mocking out the Env.

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