Designing For Testability

Designing For Testability

- Incorporate design features that facilitate testing
- Include features to:
  - Support test automation at all levels (unit, integration, system)
  - Provide visibility into the program's internal state and activities
    - This allows more specific and accurate bug reports
    - It also helps during debugging

Logs

```
class Log {

public:
    const int DEBUG   = 1;
    const int INFO    = 2;
    const int WARNING = 3;
    const int ERROR   = 4;
    const int FATAL   = 5;

    // send log output to the console
    Log();
    // send log output to a file
    Log(const string & file);
    // Send log output to a remote server
    Log(const string & serverIpAddress, int serverPort);

    // enable/disable different message types
    void EnableMessageType(int type);
    void DisableMessageType(int type);

    // Log messages of various types
    void Debug(const string & message);
    void Info(const string & message);
    void Warning(const string & message);
    void Error(const string & message);
    void Fatal(const string & message);

};
```

- Log important events to a log file
- Example Log File
- Provide a configuration file that allows logging to be turned on and off in different application modules

Logging

- Events to consider
  - Significant processing in a subsystem.
  - Major system milestones. These include the initialization or shutdown of a
  - subsystem or the establishment of a persistent connection.
  - Internal interface crossings.
  - Internal errors.
  - Unusual events. These may be logged as warnings and a sign of trouble if they happen too often.
  - Completion of transactions.

Logging

- When an error occurs, save the ring buffer containing most recent logged events
- Include timestamps
- Variable logging levels
- Take advantage of logging already present
- May be able to instrument the executable

- Determine how logs correlate to actions in the software

MVC

- Using a Model-View-Controller design that separates Views and Controllers into separate classes allows automated testing of Controller logic

- All code with any GUI library dependencies goes in the Views
- All other GUI code goes in the Controllers

- This allows all GUI code that does not touch the GUI library to be tested in an automated fashion
- Test driver passes Controllers “mock” Views that implement the IView interface

The “Mock Object” design pattern

- Allows “mock objects” to be inserted anywhere in a program
- Allows a class to be isolated from its dependencies during unit testing, or for other reasons (e.g., a class I depend on doesn’t exist yet, or I want to avoid calling it for some reason)
- Supports “fault injection”
  - Cause the software to fail at points where it normally wouldn’t to test error handling code
- Example: URL Spelling Checker

The “Mock Object” design pattern

- Mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test.
- If an object has any of the following characteristics, it may be useful to use a mock object in its place:
  - supplies non-deterministic results (e.g., current time or current temperature);
  - has states that are difficult to create or reproduce (e.g., a network error);
  - is slow (e.g., a complete database, which would have to be initialized);
  - does not yet exist or may change behavior;
  - would have to include information and methods exclusively for testing purposes (and not for its actual task).
- Mock object method implementations can contain assertions of their own. This means that a true mock, in this sense, will examine the context of each call—perhaps checking the order in which its methods are called, perhaps performing tests on the data passed into the method calls as arguments.

Dependency Injection

- Technique that allows “mock objects” to be inserted anywhere in a program
  - Easier to implement than the Mock Object pattern
- Choose a “Dependency Injection Container”
  - Ninject, Google Guice, etc.
- Configure your dependency injection container with a mapping from abstract interfaces to concrete classes that implement them
- Create objects by asking the dependency injection container for an implementation of an abstract interface
- Allows program code to depend on abstract interfaces, and not on concrete classes
- Allows mock objects to be easily inserted anywhere in the code
- Dependency Injection Example

Provide automation interfaces

- When possible, test cases should be automated (i.e., test drivers that run test cases and automatically verify results)

- User interfaces often do not lend themselves to automation (especially GUIs)
  - Record and Playback tools
  - Tools that programmatically drive the application by generating simulated UI events

- Programs can provide “automation interfaces” that allow test drivers to run commands and query application state without going through the user interface

Provide automation interfaces

- Automated test cases can exercise the application through:
  - The Core Object Model
  - UI Controllers (using mock Views)
  - A command-line interface (CLI)
    - `inventory-tracker add-item <unit> <barcode> <count>`
    - `inventory-tracker print-product-stats`
    - `Etc.`
  - A web-service interface that allows test cases to drive the application remotely

Data generators

- Creating large data sets by hand is not feasible
  - Think about trying to manually enter large amounts of item and product data into Inventory Tracker

- Data Generator
  - A program that generates large amounts of test data
  - Configuration parameters let the user “shape” the generated data
  - Example: Generate an XML file that can be used with Inventory Tracker’s XML Importer