The Knight Travails

# Overview

This document is a very simple and brief review of the design process, rationale and a discussion of task priorities for the *knight\_travail* project.

# Motivation

Simply completing and presenting the solution overlooks the considerable of decisions that need to be made during the development process. These choices and the reasoning behind them, can grant a useful insight into the developer’s thought processes, time management and priorities into whole-of-life software development.

Not to mention, it’s kinda fun..in a geeky way ☺

# The Plan

A singular question drove most of my thoughts, “how much is this a test of programming skill versus a test of developing against the full SDLC?”. I strongly believe the expectation of a senior engineer is on good engineering practice. Their work should address the necessary planning, scheduling, requirements analysis, process, change and risk management and how these correlate to a smooth, successful delivery.

Given the small scope of the assignment but limited time available (a day due to commitments) I knew I would firstly need to leverage existing tools and algorithms as much as possible. Secondly, I needed a solid build system with a TDD process to minimize integration pain and ease necessary refactoring. Thirdly, I would strictly adhere to the, “*you ain’t gonna need it*” manifesto. Whilst it was very tempting to probe some more fanciful assignment solutions, such as applying A\* against an implicit graph or the ultimate solution, precomputing all-pairs shortest paths and generating a source file with precomputed paths, such fancies were either impractical, beyond the scope of the assignment or premature optimization, eg, optimizing a single search when the bottleneck is user-input. What’s important, is maintaining a framework that affords fluid change. Lastly, I would aim for simplicity, in the code and solution as a whole. Complexity is the single biggest cost of software development.

# The Platform

To create an effective but light TDD framework I chose a combination of Git, CMake, Google Test and a simple shell script. The framework is independent of any build servers or other heavyweight tools, allowing a simple focus on compiling and running tests. The use of Git ensures a full history of commits, branches, etc are maintained locally and its performance allows rapid experimentation. CMake allows the repository layout to change erratically yet still swiftly rebuild. Finally, GoogleTest is a strong unit-testing and mocking framework.

To implement the assignment I decided to stick to C++ 2003 and a combination of STL and Boost. Although C++ 11 is a much welcome improvement, the risk of build failures across platforms was too high. I was of two minds with Boost – I personally believe the libraries add a wealth of functionality to C++ that can’t be ignored and that they align well with modern C++ abstractions. However, I’ve also encountered many individuals and companies that are hesitant towards using it. In the end, the availability of Boost Graph was far superior to the alternative of ‘reinventing the wheel’.

For OS I went with Mac OS X development on a Macbook Pro, with secondary testing on a Linux Ubuntu 12.04 VM. The Mac platform (despite issues), with its BSD backend and rock-solid hardware makes for a powerful developer environment. Still, with more time available I would have aimed towards a Windows port as well.

# The Design

Four related aspects drove the chosen design and implementation. Firstly, minimize the upfront design and instead *design for change*. Secondly, utilize existing tools from STL/Boost much as possible. Thirdly, keep the implementation simple without over-engineering. Finally, use OOP in moderation and instead focus on adopting generic-programming conventions and referentially transparent code where possible.

These four motivations led to the following decisions:

* Keep the solution a simple command-line app.
* Use Boost graph as the backend
* Limit the algorithms to dijkstra. Better solutions exist yes, but given the current requirements are ‘user inputs a path and gets a result’, it’s premature to spend time on it.
* Create a single Solver strategy to calculate knight moves, specifically shortest paths.
* Separate the application controller logic from the backend and Solver.

The resulting, very basic class diagram is:



And a sequence diagram will have to wait I gotta hand this in ☺

# Fetchez la vache!

Not typically a section in a design document, but I just wanted to say thanks for reading, I hope you enjoyed it as much as I did writing it. After all, I think being casual when appropriate is an essential ‘spice’ as an engineer. I’ll try to refrain from more Monty Python quotes though.