Agile Development Strategy – Dialectic Annealing

... project management for researchers.

We must accept the world for what it is.
Then dialectic debate may show its opposite...
This may be harmonized into that which includes them both
- Hegel

Index -

0. Abstract

We combine the process of Hegelian Synthesis with Agile development methods to address complications with developing projects with unknown scopes. Our method allows researchers to avoid being stuck in the details, and to foresee and navigate around complicated development problems by combining best practices from bottom-up, top-down, continuous-redesign, backtracking, and organic-growth strategies.

1.0) Introduction

Research and development is like drawing a map of a dark room while inside it.. Like the room, all technology has an inherent structure to it, referred to as the Archetypal shape of algorithm when one refers to code development. The complication with research (or bleeding-edge) development is that the technology's shape is unknown (and cannot be known) in the beginning. To uncover the shape in a reasonable amount of time, a clear development philosophy must focus the team development cycles. In other words, we need a method by which we can set about the task of mapping the (potentially giant) dark room...

If the complexity of the dark room is unknown, it is difficult to estimate a proper development time-line. Research requires more flexible time and task management systems than pure development projects. Using a flexible milestone tracking system allows developers to rotate priorities and build around complicated problems until the solution is found. It also frees the team to swap assignments easily with other team members. Inherently, this nurtures a merit-based development system, which empowers researchers to work on things they want to work on which encourages better quality. It also encourages the team develop gel, or cohesiveness, as a group because the team members work towards a common goal the same time they fulfill individually selected goals. In our time working on the DarkWynter project, we observed many impromptu co-development session amongst the group. In a merit-based fashion, the developers also tend to "police their own", making it easier to separate the hackers from the non-hackers.

2.0) Research as Thesis

Using a bottom-up approach, we have brainstorming sessions to form new milestones that add functionality to the codebase. While the new ideas are fresh in the mind, we develop a POC (Proof of Concept) model to explore the idea's potential. After letting the new tech settle in the minds of the developers, we redevelop it and transplant it to the best location in the main codebase.

```
 Invent <---> Integrate
        |
        L -> Backtrack
```

2.1) Bottom Up

After integrating the PoC, we take the base-level technology and grown it organically into a top-level architecture. This helps us find the architecture's most natural structure. Invent/Integrate works as a spiral model with production, however, the returns slow over time. Development cycles work outward from the base architecture. Programmers rapidly invent new functionality and apply it to the core spindle in prototype fashion. This is very efficient for brainstorming; however, coders must be rotated often to avoid divergence and stalling in development. Backtracking is time consuming, and produces an endless amount of miscellaneous code cleanup. Our measure of success is how far we can get before backtracking, thus coding endurance and discipline is key to our strategy.

2.2) Back Tracking

Backtracking specifically refers to starting with the last code developed and working backwards in the development timeline. Using the rapid-development approach, our codebase often reaches a point where forward progress is no longer feasible. When progress slows to a point where the forward rate has lost its return, we backtrack out of our development cycle.

Incremental Backtracking over time allows the base architecture to morph its inherent structure. As programmers integrate their functionality back into the core, the core grows organically in on itself, growing in layers. As development progresses, functionality crystallizes under the pressure of development.

Most of the time, we find that cleaning the code base pushes us through enough cycles to continue development. Each time the critical mass point occurs, we stop the forward progress and work in tandem to properly engineer our original work. Over time, several clean cycles produce a system architecture, and solidifies the code through a process like simulated annealing. Standard procedures for the group when stagnation occurs include general cleaning, optimizing algorithms, and Abstraction.

After the code has been cleaned up enough to provide room for forward development again, we go back to Inventing and Integrating

2.3) Transplanting

At certain times, the project (or parts of the project) must be transplanted out of the current architecture and into a new one. Before moving the code that we are keeping, we clean and modularize the code for better functionality. Once that is finished, we pull these components into the new project space, and attach controllers and camera first to get the base functionality online. Then we build in the new architecture and integrate components into the main procedural code-branch.

3.0) Development as Antithesis

We use top-down development along with our backtracking procedure to produce the final application.

3.1) Top-Down Perspective

Footprinting is the art of plotting the final software application to include all the viable technologies the programmers developed. As the final application scope begins to emerge, we reverse our procedure, from bottom-up to top-down. As this is a complicated procedure, we break away from the code to gain perspective before attempting to gain top down perspective on the code.

We begin by examining top-level descriptions of the application; this is where UML classes and sequence diagrams are priceless to the project. After taking the UML of the code, we begin to massage the structure, lining the code where it should be inside the structure. As we do this, the main components and procedural lines of the system's inherent structure become increasingly obvious as the UML structure is organized and reworked, producing a centerline for the application's architecture.

3.2) Forward Tracking

Once the footprint has been established, we work the codebase against the grain of our original bottom-up approach. Starting from the main entry point of the system, we work outward, checking for compatibility with the footprint. Conflicts are solved with interest to maximizing the functionality of the footprint, or final requirements, and no longer with the attitude of “just make it work.”

During this development time, integration, abstraction, and a general clean up of the code base will strengthen the inherent framework. Multiple iterations of cleaning are required, and developers should start at the strongest centerlines of the code and work outwards as far as progress is productive.

In forward tracking on this project, again we find ourselves continually bogged down in "currently unsolvable problems". Our solution to this is the same as for the backtracking; when our forward progress has slowed to diminishing returns, we track back into cleaner code and proceed forward with another component. Our unsolvable milestones have a tendancy to unravel themselves as other components mature and define their interfaces.

Remember, code has an inherent structure...and like a jigsaw puzzle, building around an unknown piece may reveal it.

4.0) Production forms Synthesis

Switching between bottom-up to top-down allows us to develop concepts rapidly. We can code with less precision initially, secure in the knowledge that the concepts will be revisited and industrialized. Our approach has saved a substantial number of work hours.

The speed up comes from _not_ getting lost in the details. In our experience, we find that programmers have a tendency to get bogged down in the implementation details that are often removed or recoded later anyway, due to broader structural changes. By leaving the research implementation in a vaguely "hacked-in" state, the codebase is more flexible to change as the research continues.

Industrialization of the code occurs during the development stage, but not until the full application scope has been hashed out. Our cross-grain approach, or switching to top-down passes, provide different perspectives on the codebase which helps to ensure that code is clean around the edges.

Finally, we maximize the talents of both the researcher and the developer by using separate passes. Researchers are given the freedom of creative coding, without having to worry obsessively about coding standards and best practices. Developers, on the other hand, are able to take a well defined footprint of the project and industrialize without getting bogged down with complicated research areas that may stall development.