Chapter 12 - Emergence

According to Kent Beck, a design is "simple" if it follows these rules

- Run all tests
- Contains no duplication
- Expresses the intent of programmers
- Minimizes the number of classes and methods

Chapter 13 - Concurrency

Concurrence is a decoupling strategy. It helps us decouple what gets fone from when it gets done. In single-threaded applications what and when are so strongly coupled that the state of the entire application can often be determined by looking at the stack backtrace. A programmer who debugs such a system can set a breakpoint, or a sequence of breakpoints, and know the state of the system by which breakpoints are hit.

Decoupling what from when can dramatically improve both the throughput and structures of an application. From a structural point of view the application looks like many lit- tle collaborating computers rather than one big main loop. This can make the system easier to understand and offers some powerful ways to separate concerns.

Miths and Misconceptions

- Concurrency always improves performance. Concurrency can sometimes improve performance, but only when there is a lot of wait time that can be shared between multiple threads or multiple processors. Neither situ- ation is trivial.
- Design does not change when writing concurrent programs. In fact, the design of a concurrent algorithm can be remarkably different from the design of a single-threaded system. The decoupling of what from when usually has a huge effect on the structure of the system.
- Understanding concurrency issues is not important when working with a container such as a Web or EJB container. In fact, you'd better know just what your container is doing and how to guard against the issues of concurrent update and deadlock described later in this chapter. Here are a few more balanced sound bites regarding writing concurrent software:
- Concurrency incurs some overhead, both in performance as well as writing additional code.
- Correct concurrency is complex, even for simple problems.
- Concurrency bugs aren't usually repeatable, so they are often ignored as one-offs instead of the true defects they are.
- Concurrency often requires a fundamental change in design strategy.

Chapter 14 - Successive Refinement

This chapter is a study case. It's recommendable to completely read it to understand more.

Chapter 15 - JUnit Internals

This chapter analize the JUnit tool. It's recommendable to completely read it to understand more.

Chapter 16 - Refactoring SerialDate

This chapter is a study case. It's recommendable to completely read it to understand more.

Chapter 17 - Smells and Heuristics

A reference of code smells from Martin Fowler's *Refactoring* and Robert C Martin's *Clean Code*.

While clean code comes from discipline and not a list or value system, here is a starting point.