CS 430 - Lecture 09 - Tools of the Trade I

Collin Roberts

October 17, 2023

Outline

1. Stepwise Refinement
2. Cost-Benefit Analysis
3. Divide and Conquer
4. Separation of Concerns
5. Software Metrics

Stepwise Refinement

Definition 1

Stepwise refinement *is a technique by which we defer nonessential decisions until later, while focusing on the essential decisions first.*

Stepwise Refinement

1. This is a response to **Miller's Law**.
2. The text presents a mini case study in §5.1.1, about designing an updater for a master file.
3. The details of each step in the text example of stepwise refinement are not important. The important thing is to notice how decisions get deferred until they must be settled in later iterations.

Stepwise Refinement

4. I like the fact that the example in the text is a refinement of a design. I have found this technique extremely fruitful during my own design work. It would be less effective during the implementation workflow, for example.

Stepwise Refinement

5. **Key Challenge:** Decide which issues must be handled in the current refinement, and which can be deferred until a later refinement. There is **no algorithm** to decide! Experience and human intuition are required.
6. In my experience the technique can be effective when working on a problem either individually or in a group. In a group setting
    1. **brainstorming** can be used in the early stages, and
    2. more structured **reviews** can be used in the later stages.

Stepwise Refinement

7. Features of Brainstorming
    1. The problem to be solved may initially be unclear e.g. the team might start with a symptom, and understand the underlying cause through brainstorming.
    2. All team members are encouraged to speak, especially the shy ones.
    3. No editing in the first round(s), when ideas are being suggested. Editing happens after all ideas have been suggested.
    4. **Student Question:** Is brainstorming always top-down then?

        Instructor Answer: Brainstorming can be
        1. top-down for Intuitives, and
        2. bottom-up for Sensors.

        Either way can be productive.

Cost-Benefit Analysis

Definition 2

Cost-Benefit Analysis *is*

1. *a technique for determining whether a possible course of action would be profitable, in which we*
2. *compare estimated future benefits against estimated future costs,*
3. *often referred to as the "balance sheet view".*
4. *When selecting from among several options, the optimal choice maximizes the difference*

$$(estimated\ benefits) - (estimated\ costs).$$

Pitfalls

- We must quantify **everything** to start.
- Some things are easier to quantify than others.

Tangible benefits

Tangible benefits are easy to measure, e.g. estimated revenue from a new product.

Intangible benefits

Intangible benefits can be more challenging e.g. the reputation of your organization (think Facebook, recently).

Intangible benefits

To quantify intangible benefits, we must make **assumptions**, e.g. Facebook hacks will cause 5000 users to close their accounts - then we can estimate lost advertising revenue, using historical data.

1. Advantage: With better assumptions (say from improved historical data or from a new team member who brings new experiences) we can obtain more accurate quantifications of our intangible benefits.
2. As software engineering practitioners, we must gather all of our information by **ethical** means!

Divide and Conquer

Definition 3

To **divide and conquer** *is to break a large problem down into sub-problems, each of which is easier to solve.*

Divide and Conquer

Remarks:

1. Like Stepwise Refinement, Divide and Conquer is also common sense.
2. This is the “oldest trick in the book”.
3. This is a component of the Unified Process.

Divide and Conquer

Definition 4

An **analysis package** *is defined by:*
During the analysis workflow:

1. *Partition the software product into* **analysis packages**.
2. *Each package consists of a set of related* **classes** *that can be implemented as a single unit.*

Divide and Conquer

4. During the design workflow:
    1. Partition the implementation workflow into corresponding manageable pieces, termed **subsystems**.
5. During the implementation workflow:
    1. Implement each subsystem in the chosen programming language.

Divide and Conquer

6. **Key Problem:** There is **no algorithm** for deciding how to partition a software product into smaller pieces. Experience and human intuition are required.
7. **Example:** My last large project at SunLife (2003) was developing a new intranet site. The homepage consisted of four independent quadrants. Hence the home page naturally broke down into four analysis packages, and later, into four subsystems and four streams of implementation.

Separation of Concerns

Definition 5

A software product has **separation of concerns** *if it is broken into components that overlap as little as possible with respect to their functionalities.*

Separation of Concerns

Remarks:

1. Separation of concerns is a “new and improved” version of divide and conquer. The new guiding principle for how to divide up the components is to reduce or eliminate the overlaps in their functionalities.

Separation of Concerns

Motivation:

1. Minimize the number of regression faults! If separation of concerns is truly achieved, then changing one module cannot affect another module.
2. When done correctly, this also facilitates **re-use** of modules in future software products.

Separation of Concerns

3. Manifestations of separation of concerns:
    1. design technique of **high cohesion:** maximum interaction within each module (§7.2).
    2. design technique of **loose coupling:** minimum interaction between modules (§7.3).
    3. **encapsulation** (§7.4).
    4. **information hiding** (§7.6).
    5. **three tier architecture** (§8.5.4).
4. Tracking which modules were written by weaker programmers may facilitate more proactive maintenance work.

Separation of Concerns

Moral: Separation of concerns is desirable for Software Engineering.

Software Metrics

Definition 6

A **metric** *is anything that we measure quantitatively.*

Software Metrics

1. We need **metrics** to detect problems early in the software process before they become crises.
2. Examples:
   1. # LOC, lines of code (measures **size**)
   2. # faults / 1000 lines of code (measures **quality**)
   3. (after deployment) mean time between failures (measures **reliability**)
   4. number of person-months to build (measures **size**)
   5. staff turnover (high turnover affects budgets and timelines)
   6. cost

Software Metrics

3. Two types (Exercise: categorize the list above into one of these types):
   1. **product** metrics, e.g. # lines of code for a software product.
   2. **process** metrics, e.g.
      1. # lines of code for the organization.
      2. $\frac{\#\text{ of faults detected during product development}}{\#\text{ of faults detected during product's lifetime}}$, taken over all software products in the organization. (measures effectiveness of fault detection during development)
4. Some metrics are clearly tied to a certain workflow (e.g. we cannot count lines of code until implementation)

Software Metrics

5. Five essential, fundamental metrics for a software project:
    1. Size (e.g. in # Lines of Code)
    2. Cost to develop / maintain (in dollars)
    3. Duration to develop (in months)
    4. Effort to develop (in person-months; or as in my experience in person-days)
    5. Quality (in number of faults detected during the project)
6. There is no universal agreement among software engineers about which metrics are right, or even preferred.