Chapter 3: Measure Twice, Cut Once: Upstream Prerequisites

Core Prerequisites:

- **Problem Definition**
- **Requirements**
- **Architecture**

Importance of Prerequisites

- "A common denominator of programmers who build high-quality software is their use of high-quality practices. Such practices emphasize quality at the beginning, middle, and end of a project."
    - End = testing --> quality assurance
        - Also note, testing wont uncover flaws like software being built incorrectly or even building the wrong product
    - Middle = construction practices --> the focus of this book
    - Beginning = plan for, require and design a high-quality product
        - If you want a Rolls-Royce, you need to plan to build one from the very beginning
        - Also, this planning needs to be done when:
            - the problem is defined
            - the solution is specified
            - the solution is designed

The overarching goal of preparation is risk reduction:

- A good project planner clears out major risks as early as possible so the bulk of the project can proceed as smoothly as possible
- "By far" the most common project risks are poor requirements and poor project planning

Educating Non-techs

- Part of your job as a technical employee is to educate the nontechnical people around you about the development process
- Consumption hierarchy
    - Architects consume requirements
    - Designer consumes the architecture
    - Coders consume the design

Appeal to logic

- we need to understand what to build so time and money isn't spent on building the wrong thing

Appeal to analogy

- you don't start a fire until opening the flue

- you don’t put your shoes on before your socks

Appeal to data

- studies of the last 25 years have shown conclusively that it pays to do things right the first time

Types of Software:

1) Business Systems
2) Mission Critical Systems
3) Embedded Life-Critical Systems

- **Iterative** = build things in iterations as requirements change
    - requirements not well understood
    - design is complex, challenging or both
    - development team is unfamiliar with applications area
    - project contains a lot of risk
    - long term predictability is not important
    - cost of changing reqs, design and code downstream is low
- **Sequential** = prerequisites need to be satisfied up front
    - requirements are stable
    - design is straightforward and fairly well understood
    - dev team is familiar with application area
    - project contains little risk
    - long term predictability is important
    - the cost of changing reqs, design and code downstream is high

Hierarchy:

- Future improvements (top)
- System testing
- Construction
- Architecture
- Requirements
- Problem Definition (bottom)

Problem Definition = make sure you know what you’re aiming at before you shoot

Requirements = detailed description of what the software system is supposed to do

- Official documentation is huge too
- Without good requirements, you can have the right general problem but miss the mark on specific aspects of the problem
- Also, like everything else, the deeper into the project life cycle, the cost to fix continues to grow
- "Requirements are like water. They're easier to build on when they’re frozen"

- Also, remember the customer isn't stupid, they just haven't spent as much time digging through implementation issues as you have. Imagine if you looked at their area of expertise and were expected to be the smartest person in the room about it?
- Development is like driving, and requirements are the map(ish)
- **MAKE SURE CLIENTS KNOW THE COST OF REQUIREMENTS CHANGES**
  - "Since its not in the requirements document let me pull together a revised 'schedule' and 'cost' estimate so you can decide if you want to do it now or later"
  - 'schedule' and 'cost' are scary words lmao
- refer back to the business case to keep reqs under control

Architecture = design of constraints that apply system wide

- Quality of architecture determines the conceptual integrity of the system. This in turn determines the ultimate quality of the system.
- Without good software architecture, you may have the right problem but the wrong solution. It may be impossible to have successful construction.

Amount of Time to Spend on Upstream Prerequisites:

- Typically, for Requirements, Architecture and up-front planning
  - 10-20% of effort
  - 20-30% of schedule

Overall, get the prereqs and turn over every stone in creating a simple explanation(architecture) to describe the system solution

Architecture Components

Concepts:

- Program Organization
- Major Classes
- Data Design
- Business Rules
- User Interface Design
- Resource Management
- Security
- Performance
- Scalability
- Interoperability
- Internationalization/Localization
- Input/Output
- Fault Tolerance
- Architectural Feasibility
- Overengineering

- Buy-vs-Build Decision
- Reuse Decision
- Change Strategy
- General Architectural Quality

Program Organization:

- Made up of blocks
  - these blocks can either be a single task, or a collection of classes and routines
- Every feature listed in requirements should be covered by at least one building block.
  - If a function is claimed by two or more building blocks, their claims should cooperate, not conflict
- What each block is responsible for should be well defined. It should have one area of responsibility and have as little information about other blocks as possible
  - By minimizing what each block knows about other building blocks, you localize information about the design into a single building block
- Communication rules for each building block should be well defined.
  - Architecture should describe what other blocks can and cannot be accessed and how directly those that can be are allowed to be accessed
- Basically, need something that can be explained simply
- Also need something that was selected out of several options
  - there needs to be thought put into why what was selected is the best option
- Design RATIONALE is at least as important for maintenance as the design itself.

Major Classes

- Architecture should specify major classes to be used
- identify responsibilities and other class interactions
  - class hierarchies
  - states of transition
  - object persistence
  - subsystems
- "Specify the 20% of classes that make up 80% of system behavior"

Data Design

- describe major files and table designs to be used
- also note alternatives, and justify decisions
  - (sequential access silt vs random access or stack or hash)
- databases and their contents
  - why database over flat file
  - what interactions are better
  - what views are created
- basically, every decision about data

Business Rules

- If the architecture depends on specific business rules, these need to be identified and their impact explained
- Like lifespan of customer data forcing certain updates
    - mybama 30min timeout thing

User Interface Design

- Anywhere user touches it
    - GUI, Web, CLI
- Modularize architecture so new UIs can be plugged in easily

Resource Management

- Architecture should describe a plan for managing scarce resources such as:
    - Database connections
    - Threads
    - Handles
    - Memory
- Need to make estimates of all these with justification

Security

- Code level security
- Threat modeling
- Rules for handling buffers, untrusted data (user input, cookies, configs, other external sources)

Performance

- If performance is concern, need goals to be specified in reqs
- Architecture provides estimates and explain why goals are achievable
- Areas at risk of not making goals should be noted and elaborated on

Scalability

- How will the system address growth in:
    - number of users
    - number of servers
    - number of network nodes
    - number of database records
    - size of database records
    - transaction volume
    - etc.
- if system is not expected to grow and scalability isn't an issue, that assumption needs to be made explicit

Interoperability

- if system expected to share data or resources with other software or hardware, the architecture should describe how that's accomplished

Internationalization/Localization

- I18n/L10n
- I18n is prepping a program to support multiple locales
- L19n is translating a program to support a specific local language
- Basically, handling Unicode, utf8, c strings, etc.

Input/Output

- Specify
  - Look-ahead
  - Look-behind
  - Just-in-time reading scheme
- Also, where I/O errors are detected
  - field, record, stream or file level

Error Processing

- is processing corrective or detective?
  - if detective, program can continue processing as nothing happened, or quit
- is error detection active or passive?
  - like can the system anticipate errors, like user giving too big of a number, so program discarding it
- How does the program propagate errors?
  - discard data that caused error immediately?
  - enter error processing state?
  - wait until processing complete then notify?
- What error handling conventions are there?
  - need to specify a single, consistent strategy
- How will exceptions be handled?
  - architecture should address:
    - when code can throw an exception
    - where they will be caught
    - how they will be logged
    - how they will be documented
    - etc.
- Inside the program, at what level are errors handled?
  - point of detection or pass to error handling class or up call chain?
- What is the level of responsibility of each class for validating its input data?
  - Is each class responsible for validating its own?
  - Is there a group of classes responsible for validating data?
  - Can classes at any level assume the data they receive is clean?

- Do you want to use your environments built-in exception handling mechanism or build your own?
    - just cause env has one doesn't make it the best for your reqs

Fault Tolerance

- collection of techniques that increase systems reliability by detecting errors, recovering from them if possible, and containing their bad effects if not
- Examples for error in computing a square root
    - system back up to a point where it knew everything was all right and continue from there
    - system might have auxiliary code to use if it detects fault in the primary code. if first answer is wrong, use alternative routine to calc again
    - system use voting algorithm, where three routines answers are compared
    - system might replace erroneous value with phony value it knows to have benign effect on rest of the system

Architectural Feasibility

- Basically, need to demonstrate the system is technically feasible
- if any issues, the architecture should indicate how those issues have been investigated

Overengineering

- architecture should clearly indicate whether programmers should err on side of overengineering or on simplicity
- by setting expectations explicitly in the architecture, you can avoid cases where some classes are exceptionally robust, and others aren't

Buy-vs-Build Decisions

- Basically, if not using off the shelf, need to justify how building will result in better performance than ready-made libraries and components

Reuse Decisions

- If plan calls to reuse existing software, test cases, data formats, need to elaborate on how reused software will be made to conform to other architectural goals

Change Strategy

- Need to make architecture flexible enough to accommodate changes
- Likely changes include:
    - volatile data types
    - file formats
    - changed functionality
    - new features
    - etc
- Show anticipated changes and benefits of the changes
    - "possible enhancements"

General Architectural Quality

- good architecture specifications are characterized by
  - discussions of the classes in the system
  - information hidden in each class
  - rationales for including and excluding ALL possible design alternatives
- a good architecture should fit the problem
- **when you look at the architecture, you should be pleased by how natural and easy the solution seems**
  - **should NOT look like it was forced together with duct tape**