CHAPTER ONE

Software Engineering Practice

Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed. It represents the details—the technical considerations and how to's—that are below the surface of the software process—the things that you'll need to build high-quality computer software.

The Essence of Practice

This section lists the generic framework (communication, planning, modeling, construction, and deployment) and umbrella (tracking, risk management, reviews, measurement, configuration management, reusability management, work product creation, and product) activities found in all software process models.

George Polya, in a book written in 1945 (!), describes the essence of software engineering practice…

1. ***Understand the problem*** (communication and analysis).
    - *Who are the stakeholders?*
    - *What are the unknowns? "Data, functions, features to solve the problem?"*
    - *Can the problem be compartmentalized? "Smaller that may be easier to understand?*
    - *Can the problem be represented graphically? Can an analysis model be created?*
2. ***Plan a solution*** (modeling and software design).
    - *Have you seen a similar problem before?*
    - *Has a similar problem been solved? If so, is the solution reusable?*
    - *Can sub-problems be defined?*
    - *Can you represent a solution in a manner that leads to effective implementation?*
3. ***Carry out the plan*** (code generation).
    - *Does the solution conform to the plan?*
    - *Is each component part of the solution probably correct?*

4. ***Examine the result for accuracy*** (testing and quality assurance).
    - *Is it possible to test each component part of the solution?*
    - *Does the solution produce results that conform to the data, functions, features, and behavior that are required?*

Core Principles

- ***The Reason It All Exists:*** Provide value to the customer and the user. If you can't provide value, then don't do it.
- ***KISS—Keep It Simple, Stupid!*** *All designs should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system.
- ***Maintain the product and project "vision."*** *A clear vision is essential to the success of a S/W project.*
- ***What you produce, others will consume.*** *Always specify, design, and implement knowing someone else has to understand what you are doing.*
- ***Be open to the Future.*** *Never design yourself into a corner*. Always ask "what if," and prepare yourself for all possible answers by creating systems that solve the general problem, not just the specific one.
- ***Plan Ahead for Reuse.*** *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.*
- ***Think!*** Placing clear, complete thought before action almost always produces better results.

1. **Communication Practices**

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a *communication* (also called *requirement elicitation*) activity.

Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront S/W engineers.

In this context, the following are communication **principles** and concepts that apply to customer communication:

- ***Listen***: focus on the speaker's words, rather than formulating your response to those words. Be a polite listener.
- ***Prepare before you communicate***: Spend the time to understand the problem before you meet with others "research".
- ***Someone should facilitate the communication activity***. Have a leader "moderator" to keep the conversation moving in a productive direction.
- ***Face-to-face communication is best***.
- ***Take notes and document decisions***.
- ***Collaborate with the customer***. Each small collaboration serves to build trust among team members and creates a common goal for the team.
- ***Stay focused, modularize your discussion***. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.
- ***Draw pictures when things are unclear***.

(a)Once you agree to something, move on; (b) if you can't agree to something, move one; (c) if a feature or function is unclear and can't be clarified at the moment, move on.

- ***Negotiation is not a contest or a game. It works best when both parties win.***

2. Planning Practices

The *planning* activity encompasses a set of management and technical practices that enable the S/W team to define a road map as it travels toward its strategic goal and tactical objectives.

Regardless of the rigor with which planning is conducted, the following principles always apply:

Understand the project scope. Scope provides the S/W team with a destination.

Involve the customer (and other stakeholders) in the planning activity. The customer defines priorities and establishes project constraints. S/W engineers must often negotiate orders of delivery, timelines, and other related issues.

Recognize that planning is iterative. A plan must be adjusted to accommodate changes.

Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

Consider risk as you define the plan.

Be realistic. Even the best S/W engineers make mistakes.

Adjust granularity as you plan. A fine granularity plan provides significant work task detail that is planned over relatively short time increments. A coarse granularity plan provides broader work tasks that are planned over longer time periods.

Define how quality will be achieved.

Define how you'll accommodate changes. "Can the customer request a change at any time?"

Track what you've planned and make adjustments as required.

Barry Boehm states: "You need an organizing principle that scales down to provide simple plans for simple projects."

Boehm suggests an approach that addresses project objectives, milestones, and schedules, responsibilities, management and technical approaches, and required resources.

Boehm calls it the ***W^5HH*** *principle*, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.

Why is the system being developed? Does the business purpose justify the expenditure of people, time, and money?

What will be done? Identify the functionality to be built.

When will it be accomplished? Establish a workflow and timeline for key project tasks and identify milestones required by the customer.

Who is responsible for a function? Define members' roles and responsibilities.

Where are they located (organizationally)? Customers also have responsibilities.

How will the job be done technically and managerially? Once a scope is defined, a technical strategy must be defined.

How much of each resource is needed? The answer is derived by developing estimates based on answers to earlier questions.

3. Modeling Practices

The process of developing analysis and design models is described in this section. The emphasis is on describing how to gather the information needed to build reasonable models, but no specific modeling notations are presented in this chapter. UML and other modeling notations are described in detail later in the text.

In S/W Eng. work, two models are created: analysis models and design models.

Analysis models represent the customer requirements by depicting the S/W in three different domains: the information domain, the functional domain, and the behavioral domain.

Design models represent characteristics of the S/W that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

3.1 Analysis Modeling Principles

The information domain of a problem must be represented and understood. The *information domain* encompasses the data that flow into the system (end-users, other systems, or external devices), the data that flow out of the system and the data stores that collect and organize persistent data objects.

Represent software functions. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Represent software behavior. The behavior of the S/W is driven by the interaction with the external environment.

The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion (or hierarchical).

The analysis task should move from essential information toward implementation detail. Analysis begins by describing the problem from the end-user perspective. The"essence" of the problem is described without any consideration of how a solution will be implemented.

3.2 Design Modeling Principles

The software design model is the equivalent of an architect's plans for a house.

Set of principles used:

Design must be traceable to the analysis model. The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them.

The design model translates this information into an architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis class.

Always consider architecture. S/W architecture is the skeleton of the system to be built. Only after the architecture is built should the component-level issues should be considered.

Focus on the design of data as it is as important as a design. Data design is an essential element of t architectural design.

Interfaces (both user and internal) must be designed. A well designed interface makes integration easier and assists the tester in validating component functions.

User interface design should be tuned to the needs of the end-user. "Ease of use."

Component-level design should exhibit functional independence. The functionality that is delivered by a component should be *cohesive*- that is, it should focus on one and only one function.

Components should be loosely coupled to one another and to the external environment. *Coupling* is achieved in many ways – via a component interface, by messaging through global data. Coupling should be kept as low as is reasonable. As the level of coupling increases, error propagation also increases and the overall maintainability of the system decreases.

Design representation (models) should be easily understood.

The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity.

4. Construction Practices

In this text "construction" is defined as being composed of both **coding and testing.** The purpose of testing is to uncover defects. Exhaustive testing is not possible so processing a few test cases successfully does not guarantee that you have bug free program. Unit testing of components and integration testing will be discussed in greater later in the text along with software quality assurance activities.

4.1 coding principles

Preparation Principles: Before writing one line of code, be sure of:

1. Understand the problem you are trying to solve.
2. Understand the basic design principles.
3. Pick a programming language that meets the needs of the S/W to be built and the environment in which it will operate.
4. Select a programming environment that provides tool that will make your work easier.
5. Create a set of unit tests that will be applied once the component you code is completed.

Coding Principles: As you begin writing code, be sure you

1. Constrain your algorithm by following structured programming practice.
2. Select the proper data structure.
3. Understand the software architecture.
4. Keep conditional logic as simple as possible.
5. Create easily tested nested loops.
6. Write code that is self-documenting.
7. Create a visual layout.

Validation Principles: After you've completed your first coding pass, be sure you

1. Conduct a code walkthrough.
2. Perform unit test and correct errors.

3. Refactor the code.

4.2 Testing Principles

- Testing is a process of executing a program with the intent of finding errors.
- A good test is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

5. Deployment Practices

Customer Expectations for the software must be managed. "Don't promise more than you can deliver."

A complete delivery package should be assembled and tested.

A support regime must be established before the software is delivered.

Appropriate instructional materials must be provided to end-users.

Buggy software should be fixed first, delivered later.

Code generators

In general, there are two main categories of Automatic code generation: passive or active. Passive code generators build the code once, then have nothing more to do with the code. Active code generators, on the other hand, keep track of the code during its lifecycle. Active code generators are run on code multiple times during the lifecycle. With Active Code generators, there is code you can modify, and code that should only be modified by the code generator.

1. Code generation approaches

The main code generation approaches can be singled out: visitor-based and template-based.

- The point of the ***visitor-based*** approach is that the code is being generated while iterating through textual representation of the model.
- The ***template-based*** approach is more commonly used because it could provide more complex and "nice" code generation. This approach implies writing special textual templates, which basically are a set of rules. These rules specify the way of generating code from some specific model.

2. Code generation types:

- **Code mungers** takes existing code and generates new code. Xdoclet is an example of a code munger. Instead of rewriting the code by hand, a program could translate the C source code into java source code. I personally have used a program to turn BASIC code into QBASIC code.

- **Inline code expanders** take existing code and elaborate certain sections of that code. For example, Pro*C takes SQL code in a program and expands it to c code.
- **Mixed code generators** take model use code as input then build new code and inserts it back into the original code. Wizards are often used to implement this new code. I would consider the majority of HTML page creators to be mixed code generators.
- A **partial class generator** is different from the previous code generators. It takes an abstract definition as input instead of code. It then outputs code which the user extends by creating derived classes and extending methods. For example, Expert coder is a toolkit that can generate expert systems that convert UML models to CodeDOM objects.
- **Tier generators** seem to be the most common type of code generator. Tier generators build complete output code from an abstract definition. For example, there are many SDK's that allow a user to create a GUI by pointing and clicking. The SDK then transforms the model of the interface to code.

3. **Code generation techniques**

- Code generation techniques as: ***Templates and filtering*** **technique** defines that generalized source code fragments are represented as textual templates. During code generation process, the variables of these templates are associated with the system model.

- ***Templates and meta-model*** **technique** is very similar to the previous one, except the new meta-model of the system representation is being created before the template is executed in a user defined way. It means that templates are not dependent on the model syntax and could be written in any programming language.

- ***Frame processors*** have the following principle of operation,the code is being specified with the help of objects -like frames. Each frame owns a group of attributes called slots. When a new frame is being created, its slots are connected with specific values. They may represent different aspects of the system model and also may contain references to other frames. Thus, during code generation, the frames form the tree structure of the system source code.

- ***API-based generator*** is usually connected with a single programming language. Thus it provides the user with a special framework, which makes the code generation process more intuitive. The source code is commonly specified with the help of templates. API-based generators often have built-in compilers that are able to evaluate the generated code at once.

- ***In-line generation*** **technique** determines that the program source code contains text fragments, which at the time of compilation, depending on the conditions, can be extended or not included into the program at all. Such code generation technique is quite primitive, therefore it cannot be used as the main one.

- ***Code attributes*** **technique** defines that special text such as comments can be placed in the existing source-code. In fact, these comments are specific instructions to the code generator. This technique is commonly used to automatically generate program documentation.

- ***Code weaving*** **technique** states the following: some independent and non-related pieces of text are manually written into the input file. At the same time it is also defined how all these fragments should be joined together during the code generation process. Finally, all these pieces would be processed and mixed to form the program source code.