Software Engineering Components

Product - result

Problem - requirements and specification

Process - organization and actions to get results

In developing a project which entails software engineering, I would organize the project in the order above.
Software evolves
⇒ Takes on a life of its own
⇒ New hardware, new systems, new needs, handle more complexity

Software experiences
⇒ Used to be many “horror” stories but they have subsided as we are better able to perform software engineering effectively.
Software characteristics

⇒ Software doesn’t wear out. Failures to “design” flaws.

⇒ Software is mostly custom built.
   New technology enables frameworks, components, etc.

⇒ Software manufacture is almost trivial

⇒ Software systems - is there a crisis?
   Yes and no
Myths can be strawmen used by “salesmen”.

⇒ A general statement of the objectives is sufficient to begin writing programs

⇒ Situations where problem, solution, and experience is lacking. Apollo is a typical example.

⇒ Software creates voluminous and unnecessary documentation.
   ⇒ This can be true or false.
   ⇒ “Good” software engineering recognizes documentation is communication and “corporate” memory.
Software Problem Definition

Requirements and specifications

⇒ System definition
   A set or arrangement of elements organized to accomplish some predefined goals by processing information

⇒ System elements may be:
   ⇒ Software - computer programs, data structures, documentation, etc.
   ⇒ Hardware - electronic computational systems, sensors, network and communication elements, actuators, etc.
   ⇒ People - operators, users, etc.
   ⇒ Documentation - description information concerning the system and how to use it.

⇒ Elements are combined to transform information, accomplish useful objectives
It should specify behavior, constraints on implementation, any thoughts about life cycle (use), responses to events, etc. Like other documents in engineering process, it should enable evolution.
Requirement and Specification Techniques

⇒ Scenario Analysis
  ⇒ Viewpoint, Service operation and service actions
  ⇒ Scope of viewpoints
    ⇒ Data sources & sinks
    ⇒ Frameworks - organization & structure
    ⇒ Users of service
    ⇒ Environment
  ⇒ Scope of service
    ⇒ data flow, event, hierarchy - you illustrate with charts & diagrams
⇒ Remember to consider the all parts of system.
Divide and conquer

⇒ Multiple level functions and/or objects

Component Eng.

Software Eng.

Product

Hardware

Software

Data

Function

Action
Information Flow Models

Flow charts (PSpec)

This should not be new to any of you
Control Flow Models

State Transitions Flow Charts (CSpec)

- Start Copying
  - Read Commands
    - Done
    - Empty tray
      - Make Copies
      - Malfunction
        - Diagnose Problems
          - Malfunction Fixed
    - Reloading Paper
      - Malfunction
        - Fixed
  - This loop could go to make copies

This could be a malfunction
Other Req. & Spec. Techniques

- Pre- & post- conditions
  - Operators - <, ==, not, or, for_all, there_exists, etc.
  - Predicates - a Boolean expression
- Algebraic

<table>
<thead>
<tr>
<th>Type:</th>
<th>name of spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>imports:</td>
<td>list of specification names</td>
</tr>
</tbody>
</table>

Informal description of type and its operations

Operation signatures setting out the names, parameters, and results of the operations

Axioms defining the behavior over the type. They relate the operations used to construct the entities with the operations used to inspect its values.
Structure Program Development Languages

⇒ PDL’s may be derived from actual programming languages
⇒ Use when:
  ⇒ sequence of steps like a program execution is needed
  ⇒ when specifying programatic operation
⇒ Difficulties
  ⇒ the PDL may not be rich enough linguistically
  ⇒ specifications at this detail may obscure a better design later
Objectives - Enable management and technical personnel to:

⇒ comprehend the problem and solution strategy

⇒ make reasonable estimates of resources needed

⇒ make the decisions to get and keep the project on track
  ⇒ risk analysis - chapter 6
  ⇒ setup initial scheduling and tracking - chapter 7
  ⇒ establish software configuration management - chapter 9
Resources

Goal - to get needs committed early

⇒ Human
  ⇒ Commitment, experience, environment, support, etc.
  ⇒ Estimation (Hind sight is 20/20 but costly)
    ⇒ Base on comparable projects, experience
    ⇒ Decompose and estimate subproject costs
    ⇒ Use empirical software estimation models
  ⇒ Make/buy decision
  ⇒ Relate to specs, designs, code, test data, etc.
⇒ Off-the shelf components
⇒ Full-experience or partial-experience components
⇒ New components - by far the highest risk decision
Project Estimation

- Based on the use of project metrics
- 1. Subdivide project into “logical” units (LU) Function points, object points, code units with LOC estimates, etc.
- 2. Weight the difficulty of each \( S = (S_{opt} + 4S_m + S_{pst})/6 \)
- Use Estimation model equations \( E = A \times B(S(LU)) \)
- Compare results to past project histories
- The logical units can be anything which makes sense to the project and people
- Makes estimates during project stages
  - COCOMO model of Boehm
- Correlate estimates with finished projects - feedback
- Estimate different things Cost, schedule, personnel needed (possibly by type), etc.
A very important part of project management planning

- Provides for an intelligent progression of project subtasks
- Provides insight into the inter-dependence of subtasks
- Compartmentalization - same principle as subdivision
  - Relate to metrics and project planning (FP, OP, etc.)
- Time & effort allocations - Determine the time & effort (resources) needed for each task
- Milestones & Deliverables
- Timeline chart or alternatively PERT and/or Critical Path charts can be used
Material covered in 2nd half

⇒ Specs, Design & Analysis Chpts 11 - 14
  ⇒ Req. & Spec. Analysis Chpts 12 & 13
  ⇒ Architectural D & A - Chpt 14
  ⇒ Interface D & A Chpt 15
⇒ Object-oriented Design
  ⇒ Object language features
  ⇒ Object design structuring - COM
  ⇒ Object D & A chpts 20 - 22
⇒ Testing Chpts 17 - 19
  ⇒ Techniques & Strategies - chpts 17 - 18
  ⇒ Metrics & Faults
Specifications

Bridge between requirements and “software” development

⇒ Methods
  ⇒ Formal - algebraic (pre- & post-cond.; ops & axioms, etc.)
  ⇒ Language (PDL, Executable Spec., High-level, 4 GLs, etc.)
  ⇒ Natural Language (understanding problem, outline format)
  ⇒ Frameworks (Visual tools)
  ⇒ Scenarios
    ⇒ Service - Viewpoint
    ⇒ Responsibility - Collaboration

⇒ Difficulties
  ⇒ People, communications, customer uncertainty, changing requirements, etc.
How to get them close?

⇒ Data structure
  ⇒ Data entity
    ⇒ Cardinality and modality

⇒ Data flow
  ⇒ Flow charts

⇒ Control Flow
  ⇒ Control charts
  ⇒ State transition diagrams
  ⇒ Data - state coupling diagrams
Design

First step in coding

⇒ Starting point
  ⇒ Requirements & Specification
  ⇒ Preliminary design & analysis

⇒ Decomposition
  ⇒ Top-down vs bottom-up
  ⇒ Modularity
    ⇒ functional independence - analyze “fan-out”
    ⇒ Minimize the interfaces between “objects”

⇒ Languages
  ⇒ Many proposed - Java & C++ are the preferred
  ⇒ Libraries - STL, COM objects, software development kits, etc.
Design Objectives

⇒ Modularity

⇒ Ease of editing

⇒ Readable (human & machine)

⇒ Maintainability

⇒ Effective data representation

⇒ Ability to easily translate into code
  ⇒ The project selected language(s)

⇒ Reuseability
Architectural Design

Objects are the central structure

⇒ Data architectures
  ⇒ Data libraries, access & dictionary

⇒ Data flow
  ⇒ Call and return - main and subprogram structure

⇒ Layered structure

⇒ Object-oriented structure
  ⇒ Functions & data
  ⇒ Effective interfacing
  ⇒ Layering, information hiding, etc.
Code Design

- Code Design Methods
- Structured Programming
  - Sequence, Selection & Repetition
  - Use constructs for graphing program

Flowchart:
- Sequence
- Switch (not shown)
- If then else
- F T
User Interface Concepts

⇒ Place User In Control
  ⇒ Mode vs modeless
    Mode – action complete before other actions.
    Modeless – action may be suspended and restarted without complications
  ⇒ Actions should be undoable
⇒ Make interface consistent
  ⇒ Provided by Macintosh – Apple
  ⇒ Action modes flow easily
⇒ Reduce user’s need to remember
  ⇒ Unavailable actions should be dimmed
  ⇒ Provide multiple means for actions
⇒ Actions should hide technical details
⇒ Actions tailored to user’s skill level
  ⇒ Multiple methods for controlling actions
Interface Model

Establish Environment

⇒ Users
  ⇒ Novices - Little syntactic or semantic knowledge - need help
  ⇒ Knowledgeable User
    ⇒ Intermittent - consistency
    ⇒ Frequent users – multiple methods for selecting action.
⇒ Location of interface?
  ⇒ External conditions – noise, interference, disturbances, etc.
⇒ User's considerations
  ⇒ Position and other related issues directly affecting the user
⇒ Other special human factors
  ⇒ Designs for disabilities, range of user abilities, range of user physical features
So many design issues it's almost impossible to discuss them.
⇒ Get as many opinions as possible
⇒ Try to formalize the evaluation
⇒ How many actions did a user employ in a session the learning time if possible
⇒ Compare with other interfaces, especially in the same area for consistency.
⇒ Try to get novice and knowledgeable users trials

⇒ Prototype – try to get early evaluation

⇒ Implementation Tools
  ⇒ frameworks – windows interface development tools
Object-oriented Programming

⇒ Support Characteristics of Language

⇒ Exam from a syntax & semantic aspect

⇒ Component Object Model

⇒ Using the object concept chpts 21 - 22
Language Features

C++ as example

⇒ class - starts definition of object
⇒ Data - provides both simple and complex structures
⇒ Functions
  ⇒ function name overloading, operator functions, pointers to functions, etc.
⇒ Data and functions can be available or hidden
⇒ Inheritance - base and derived classes
  ⇒ Implements decomposition
  ⇒ Enables significant reuse and upgrade
⇒ Polymorphism - dynamic (run-time) binding
⇒ Uniform exception handling
⇒ Properties - adds effective design & runtime data control
⇒ Templates
  ⇒ Standard Template Library
Component Object Model (COM)

DCOM is distributed COM
COM Interface (cont’d)

Structure for In-process Server
- Client
- COM Object
- Inprocess Proxy

Structure for Out-of-process server
- Network server
- Com RPC
- DCom
- Stub
- Inprocess Object

RPC - remote procedure call
Query features

- Client can always query interface to determine services
- Objects allow clients to request known interfaces
- With IDispatch, client can query about methods supported
- Server has no expectations about client using the object

Two general characteristics

- Controllers - request services and interact
  - Example - sending information to another client
- Container - enters into client process to provide control, display, etc.
  - Example - button in your window
- Provide general functionality
Provide design and user interface info

⇒ Contain *type information*
  ⇒ C++ augmented structures
  ⇒ Includes the identifiers to CoClasses (CLSID - GUID), interfaces (IIDs), dispatch identifiers (dispID)
  ⇒ References to other type libraries
  ⇒ Registry information access
  ⇒ Information necessary to compile for use

⇒ Library tools assist in implementation & use
Management of object-oriented projects

Similar to other design activities in projects

- Establish process “framework” for project
  - resource needs, deliverables, milestones, reuseability, risk, etc
- Analysis
  - Define functional and operational requirements
  - Describe end user
  - Provide a basis for validation testing
  - In the process analyze using scenarios
    - Responsibility - collaborator modeling
      - Similar to service - viewpoint
Responsibility - collaborator modeling

⇒ Responsibilities
  ⇒ attributes & operation relevant to the class
  ⇒ “anything the class does or knows”
  ⇒ break up - should be shared
  ⇒ “has a” and “is a” relate to sub or inherited class

⇒ Collaborators
  ⇒ classes which provide the class with information needed
  ⇒ “request for information or action”

⇒ Sec. 21.5 illustrates for SafeHome project.

<table>
<thead>
<tr>
<th>Class name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class characteristics:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsibilities:</th>
<th>Collaborators:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Testing Chpts 17-19

⇒ Different from previous project activity
⇒ Principles
  ⇒ Tests traceable to specifications (customer req.)
  ⇒ Tests (concepts) planned during previous project activity
  ⇒ Tests bottom-up
  ⇒ Exhaustive test isn’t possible
  ⇒ Test coordinated between designers & third party
⇒ Testability
  ⇒ Operability, Observability, Controllability, Decomposability, Simplicity, Stability, Understandability, Coupling
⇒ Test Case Design
  ⇒ White-box, black-box, interface
Test Case Design

⇒ White Box
  ⇒ tests designed with knowledge of internal structure
  ⇒ look for logic, process, condition, etc. errors
  ⇒ path and control structure testing

⇒ Black Box
  ⇒ tests designed without regard to internal structure
  ⇒ looks for inputs that cause error
  ⇒ looks at output that is incorrect or incomplete

Developers

Independent Testers

White-box

Interface

Black-box
White-box Testing

Techniques

⇒ Flow Graph Path Testing
  ⇒ Cyclomatic Complexity
  ⇒ Edges, nodes, regions
  ⇒ Connection Matrix

⇒ Control Structure Testing
  ⇒ Condition - Ej < relational operator > Ek

⇒ Error Conditions
  ⇒ Boolean operator error, variable, parenthesis (incorrect, missing, etc)
  ⇒ Relational - 3; <, ==, >
  ⇒ Relational operator or expression error
    Ej & Ek test and set over range and minimal relationship
    C1: Bj & Bk - {(T,T), T,F), (F,T)}
Data Flow Path Testing

- Test variables in the program - X
  - DEF(S) - statement S contains the definition of X
  - USE(S') - statement S' contains a use for X
  - DU chain - [X, S, S']

- DU chains strategies consider the logic and loop statements.

- Compiler error & warning can help
  - “Variable use without being set”
  - “Variable not defined”
Black Box Testing

Complimentary to white box testing

⇒ Usually applied after white box testing
⇒ Test questions:
  ⇒ How is functional validity tested?
  ⇒ How is performance & behavior tested?
  ⇒ What types of input for good test cases?
  ⇒ What features are sensitive to input parameters?
  ⇒ What are boundaries of the system?
  ⇒ What data rates & volume can tax the system?
  ⇒ What combinations of data can influence performance?
Black-Box Testing

BB testing related to design models

- Transaction flow models
- State transition models
- Timing models
- Data flow models
- Data entity models
- Equivalence Partitioning - input regions
- Comparison testing - repeat after changes & versions
Special testing

⇒ Testing GUIs
⇒ Testing Client/Servers
⇒ Testing Documentation and Help
⇒ Testing Real-time Systems
Approach is bottom-up

- Start by developing test strategies
- Unit testing - white box
  - Test and debug - use unit testing techniques
  - Profiling - determine code where runs take most time
- Integrated, interface and black-box
  - Regression testing - use older tests info and retest
  - Smoke testing - limits of design
- Recovery testing
- Verification & validation
  - Validation - are we building the right product?
  - Verification - is the product we are building correct?
Software Metrics

⇒ Problem
  ⇒ difficult to find metrics for all quality measures
  ⇒ some measures are highly subjective
  ⇒ different meaning (or utility) to different users

⇒ Effective metric
  ⇒ Simple, computable (measurable), Empirically and intuitively persuasive, Consistent & objective, Language independent, Feedback to software available
Design Model Metrics

Structural complexity: \( S(i) = fout(i)^2 \)
- \( fout(i) \) is the fan-out of module \( i \)

Data complexity: \( D(i) = \frac{v(i)}{fout(i)+1} \)
- \( v(i) \) is the I/O count for module \( i \)

High-level architectural complexity
- \( HKM = length(i) \times [fin(i)+fout(i)]^2 \)
  - \( fin = \) fan-in
  - \( length = \) language statements in module \( i \)

Simple morphology
- \( size = \) number of nodes + number of arcs

US AF Systems Command measures
Software does not wear out

⇒ Informal
  ⇒ How well does the software preform for the user?
  ⇒ Depend upon failure

⇒ Formal
  ⇒ Probability of failure free operation over a period of time
  ⇒ Removal of large % of faults doesn’t make a difference
  ⇒ Program has faults but perceived reliable by user
    ⇒ work arounds, fault tolerance, fault recovery

⇒ Formal development methods
  ⇒ no guarantee that software will be more reliable
  ⇒ specs may not reflect real user requirements
  ⇒ “proofs” may contain errors and/or not reflect user patterns
Reliability metrics

Measures

⇒ Number of system failures given a number of system inputs (POFOD)
⇒ Time between system failures (MTTF - 1/ROFOC) - rate of failure occurrence
⇒ Elapsed repair time or restart time given system is “continuously” available (AVAIL)
⇒ What is a “failure”? Some faults such as transient and those without serious consequences - Are they failures?
Fault Avoidance & Tolerance

Avoidance
⇒ Important strategy - design & implementation organized to enable

Tolerance
⇒ Assume faults remain - allow operation to continue after fault occurs
⇒ Proper use of exception handling very important here.

Detection
⇒ Testing and debugging - the very potential for enabling reliable systems

The objective of software engineering is embodied here.