Chapter 9 Notes

After a system has been developed, it inevitably has to change if it is to remain useful.

* To fix errors
* Adapt it to changes in hardware and software
* Improve performance
* To adapt to requirements changes

Most companies spend more money on software maintenance than SW development. (85%-90% of SW costs are evolution costs)

Brownfield SW Development = situations in which SW systems have to be developed and managed in an environment where they are dependent on, on many other SW systems.

Large software systems have very long lifetimes.

SW Engineering should be seen spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system. You start by creating release 1 of the system. Once delivered changes are proposed and the development of release 2 starts almost immediately.

Evolution Processes (Section 9.1)

* Evolution may be an informal process between the system users and developers. In other companies, it is a formalized process with structured documentation produced at each stage in the process
* System change proposals are the driver for system evolution in all organizations.
  + They come from existing requirements that have not been implemented, request for new requirements, bug reports, and new ideas for SW improvement.
  + The process of change identification and system evolution are cyclic and continue throughout the lifetime of a system
* Software Evolution Process
  + Change Requests
  + Impact Analysis
  + Release Planning (covers all proposed changes below… and decides which one to implement)
    - Fault Repair
    - Platform Adaptation
    - System Enhancement
  + Change Implementation (an iteration of development)
    - Proposed Changes
    - Requirements Analysis
    - Requirements Updating
    - Software Development (including testing)
  + System Release
* Change Requests sometimes relate to system problems that have to be tackled urgently. These urgent changes can arise for 3 reasons:
  + If a serious system fault occurs that has to be repaired to allow normal operation
  + If changes to the system operating environment have unexpected effects that disrupt normal operation
  + If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.
* Urgent Change Requests can be problematic because it means that you may not be able to follow the formal change analysis process of modifying the requirements and design. So the danger comes in to play when you make changes to code and never update the documentation. They are also problematic because you use a quick solution to a problem and not the best solution.
* Evolution pretty much involves continuing the agile development process.
* 2 Problems that arise when there is a handover from a development team to a separate team responsible for evolution
  + Where the development team has used an agile approach but the evolution team is unfamiliar with the agile methods and prefers a plan-based approach.
    - The team may expect detailed documentation to support evolution but this is not commonly produced in agile processes
  + Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

Program Evolution Dynamics (Section 9.2)

* Is the study of system change.
* Lehman’s Laws
  + Continuing Change
    - A program that is used in a real-world environment must change or else it will become increasing less useful.
  + Increasing complexity
    - As more change occurs, the structure of the system tends to become more complex.
  + Large Program Evolution
    - Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release
  + Organizational Stability
    - Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
  + Conservation of familiarity
    - Over the lifetime of a system, the incremental change in each release is approximately constant
    - The more functionality that there is in an increment, the more faults that there will be.
  + Continuing Growth
    - Functionality offered by system has to continually increase to maintain user satisfaction
  + Declining Quality
    - Quality of systems will decline unless they are modified to reflect changes in their operational environment
  + Feedback System
    - Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback system to achieve significant product improvement

Software Maintenance (Section 9.3)

* Is the general process of changing a system after it has been delivered.
* This term is usually applied to custom software in which separate development groups are involved before and after the delivery.
* Changes made to software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or accommodate new requirements.
* 3 Types of SW Maintenance
  + Fault Repairs
    - Coding errors are usually relatively cheap to correct
    - Design errors are more expensive as they involve rewriting several program components.
    - Requirement errors are the most expensive to repair because of the extensive system redesign with may be necessary.
  + Environmental Adaptations
    - Is required when some aspect of the system’s environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes
  + Functionality addition
    - Is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the SW is often much greater than for the other types of maintenance.
* More of the maintenance budget is spent on implementing new requirements than on fixing bugs.
* Maintenance Effort Distribution
  + 65% = Functionality Addition or Modification
  + 17% = Fault Repair
  + 18% = Environmental Adaptation
* It is cost effective to spend time in design and development to make the system more maintainable and easier to modify in the future.
  + How to reduce Maintenance Cost
    - Use precise specifications
    - Use of Object Oriented Development
    - Configuration Management
  + This is the idea behind Agile Methods
* It is more expensive to add functionality after a system is in operation than it is to implement the same functionality during development. The reasons for this are:
  + Team Stability
    - After a sytem has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before implementing changes to it.
  + Poor development practice
    - The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write maintainable software. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in the future
  + Staff skills
    - Maintenance staffs are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. Also, the system maybe implemented in obsolete languages that the maintenance team do not know.
  + Program age and structure
    - As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change.
  + \*\*NOTE
    - The first 3 issues above deal with how organization still views development and maintenance as separate processes with no incentive to develop code that is easier to maintain.
* Maintenance Prediction (Section 9.3.1)
  + You should try to predict what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain.
  + Stages
    - Predicting System Changes
      * How many change requests can be expected?
      * What parts of the system are most likely to be affected by change requests?
    - Predicting Maintainability
      * What parts of the system will be the most expensive to maintain?
    - Predicting Maintenance Costs
      * What will be the lifetime maintenance costs of this system?
      * What will be the costs of maintaining this system over the next year?
  + To evaluate the relationships between a system and its environment, you should assess:
    - The number and complexity of system interfaces
      * The more interfaces and the more complex these interfaces, the more likely changes will be required as new requirements are proposed
    - The number of inherently volatile system requirements
      * Requirements that reflect organizational policies and procedures are likely to be more volatile
    - The business processes in which the system is used
      * As business processes evolve, they generate system change requests. The more processes that use a system, the more the demands for system change
  + The more complex a system, the more difficult it is to maintain
  + To reduce maintenance costs, you should try to replace complex system components with simpler alternatives
  + After a system has been put into services, you can use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:
    - Number of requests for corrective maintenance
      * an increase in bug and failure reports
    - Average time required for impact analysis
      * this reflects the number of program components that are affected by the change request. If time increases, it implies more and more components are affected and maintainability is decreasing
    - Average time taken to implement a change request
      * This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected
    - Number of outstanding change requests
      * An increase in this number over time may imply a decline in maintainability.
* Software Reengineering (Section 9.3.2)
  + To make legacy systems easier to maintain, you can reengineer them to improve their structure and understandability.
  + Reengineering may involve:
    - documenting the system
    - refactoring the system architecture
    - translating programs to a modern programming language
    - Modifying and updating the structure and values of the system’s data.
  + 2 Important benefits from reengineering rather than replacement
    - Reduced Risk
      * There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.
    - Reduced Cost
      * The cost of reengineering may be significantly less than the cost of developing new software.
  + Activities of the Reengineering Process
    - Source Code Translation
      * Program is converted from an old programming language to a more modern version of the same language or to a different language.
      * SW Tools may be used here
    - Reverse engineering
      * Program is analyzed and information extracted from it. This helps to document its organization and functionality
      * Usually is automated
    - Program Structure Improvement
      * The control structure of the program is analyzed and modified to make it easier to read and understand.
    - Program Modularization
      * Related parts of the program are grouped together and, where appropriate, redundancy is removed. This may also involve Architectural Refactoring
    - Data Reengineering
      * The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure.
  + Preventative Maintenance by Refactoring (Section 9.3.3)
    - Refactoring is the process of making improvements to a program to slow down degradation through change.
      * It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand.
    - Refactoring can be seen as “Preventative maintenance” that reduces problems of future change
    - Reengineering takes place after a system has been maintained for some time and maintenance costs are increasing. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases costs and difficulties of maintaining a system.
    - Refactoring is an inherent part of agile methods
    - Situation in which programs can be improved
      * Duplicate Code
        + Fix this by replacing repeated code with a call to a function
      * Long methods
        + Fix this by separating the methods out into smaller methods
      * Switch (Case) Statements
        + These often involve duplication around multiple parts of the program
        + This can be fixed through polymorphism
      * Data Clumping
        + This occurs when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program.
        + These can be fixed by having an object encapsulate all of the data
      * Speculative Generality
        + Occurs when developers include generality in a program in case it is required in future
        + This can often be fixed by simply removing it.

Legacy System Management (Section 9.4)

* Many companies have many legacy systems in which they must decide whether to proceed with maintain or not.
* 4 Strategic Options for Realistic Assessments of Legacy Systems
  + Scrap the system completely
    - Should be chosen when the system is not making an effective contribution to business processes.
  + Leave the system unchanged and continue with regular maintenance
    - Should be chosen when the system is still required but it is fairly stable and the system users make relatively few change requests
  + Reengineer the system to improve its maintainability
    - Should be chosen when the system quality has been degraded by change and where a new change to the system is till be proposed.
  + Replace all or part of the system with a new system
    - Should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation or where off-the-shelf system would allow the new system to be developed at a reasonable cost.
* When you access a legacy system, you must do so from a business perspective and technical perspective
  + Business Perspective
    - Decide if the business really needs the system.
  + Technical Perspective
    - Assess the quality of the application software and the system’s support software and hardware.
  + Then use a combination of the business value and system quality to inform your decision on what to do with legacy system.
  + 4 Clusters of Systems
    - Low Quality, Low Business Value
      * Keeping these systems will be expensive and the rate of return to the business will be small. These should be scrapped.
    - Low quality, High Business Value
      * These systems cannot be scrapped since they have value.
      * Their low quality means that it is expensive to maintain them. These system should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf system is available.
    - High Quality, Low Business Value
      * Don’t contribute much to the business but they may not be very expensive to maintain.
      * It is not worth replacing these systems so noramml system maintenance may be continued if expensive changes are not required if expensive changes are not required and the system hardware remains in use. If expensive changes are required, the software should be scrapped
    - High Quality, High Business Value
      * These system have to be kept in operation.
      * Their high quality means you don’t need to invest much in transformation or system replacement.
      * Normal system maintenance should be used.
  + To asses business value of a system you must discuss the following info with stakeholders:
    - The use of the system
      * How much is it used?
      * Does it serve an importan function?
    - The business processes that are supported
    - The system dependability
      * Is the system dependable and reliable?
    - Thee system outputs
      * Is whatever the system outputs/perform important and used by customers?
  + To asses value of a system from the technical perspective you must consider the application system and its environment
    - Environment Assessment Factors
      * Supplier Stability
        + Is the supplier still in existence?
        + Is the supplier likely to still exist?
      * Failure Rate
        + Does the hardware have a high rate of reported failures?
        + Does the support software crash and force system restarts?
      * Age
        + How old is the hardware and software?
      * Performance
        + Is the performance of the system adequate?
      * Support Requirements
        + What local support is required by the hardware and software?
      * Maintenance Costs
        + What are the costs of hardware maintenance and support software licenses?
      * Interoperability
        + Are there problems interfacing the system to other systems?
    - Application Assessment Factors
      * Understandability
        + How difficult is it to understand the code?
        + How complex are the control structures used?
      * Documentation
        + What system documentation is available?
        + Is it consistent, complete, and current?
      * Data
        + Is there an explicit model for the system?
        + To what extent is data duplicated across files?
      * Performances
        + Is the performance of the application adequate?
      * Programming Language
        + Are modern compilers available for the programming language used to develop the system?
      * Configuration Management
        + Are all version of all parts of the system managed by a configuration management system?
      * Test Data
        + Does test data for the system exist?
      * Personnel Skills
        + Are there people available who have the skills to maintain the application?
  + Data that may be helpful in quality assessment are:
    - The number of system change requests
      * The higher this number is, the lower the quality of the system since each change damages the system structure
    - The number of user interfaces
      * More interfaces means that there is a higher chance that there are redundancies in them
    - The volume of data used by the system
      * High volume of data means that it is more likely that there will be data inconsistencies that reduce system quality.