# Software Change

* Software must change to remain viable
* Software change is inevitable
  + New requirements emerge when the software is used
  + The business environment changes
  + Errors must be repaired
  + New computers and equipment is added to the system
  + The performance or reliability of the system may have to be improved
* A key problem for all organizations is implementing and managing change to their existing software systems.

# Importance of Evolution

* Organizations have huge investments in their software systems - they are critical business assets.
* To maintain the value of these assets to the business, they must be changed and updated.
* Since maintenance consumes ~60% of software costs, it is probably the most important life cycle phase of software . . .
  + The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

# Evolution and servicing

* Evolution
  + The stage in a software system’s life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
* Servicing
  + At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software’s environment. No new functionality is added.
* Phase-out
  + The software may still be used but no further changes are made to it.

# Evolution Process

* Software evolution processes depend on
  + The type of software being maintained;
  + The development processes used;
  + The skills and experience of the people involved.
* Proposals for change are the driver for system evolution.
  + Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
* Change identification and evolution continues throughout the system lifetime.

# Change implementation

* Iteration of the development process where the revisions to the system are designed, implemented and tested.
* A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.
  + During this phase, you must understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

# Urgent Change Requests

* Examples:
  + If a serious system fault must be repaired to allow normal operation to continue
  + If changes to the system’s environment (e.g. an OS upgrade) have unexpected effects
  + If there are business changes that require a very rapid response (e.g. the release of a competing product).

# Software maintenance

* The term is mostly used for changing custom software.
* Generic software products are said to evolve to create new versions.

# Types of maintenance

* Maintenance to repair software faults
  + Changing a system to correct deficiencies in the way meets its requirements.
* Maintenance to adapt software to a different operating environment
  + Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
* Maintenance to add to or modify the system’s functionality
  + Modifying the system to satisfy new requirements.
* Which type do you think requires the most effort?

# Maintenance Costs

* Usually greater than development costs: 2 to 100-times depending on the application.
* Increases as software is maintained.
  + Maintenance corrupts the software structure so makes further maintenance more difficult.
  + Aging software can have high support costs
    - For example: old languages, compilers etc.

# Maintenance Cost Factors

* Team stability
  + Maintenance costs are reduced if the same staff are involved with them for some time.
* Contractual responsibility
  + The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
* Staff skills
  + Maintenance staff are often inexperienced and have limited domain knowledge.
* Program age and structure
  + As programs age, their structure is degraded and they become harder to understand and change.

# System Re-engineering

* Applicable where some but not all sub-systems of a larger system require frequent maintenance.
* Re-engineering involves adding effort to make them easier to maintain.
  + The system may be re-structured and re-documented.

# Advantages

* Reduced risk
  + There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
* Reduced cost
  + The cost of re-engineering is often significantly less than the costs of developing new software.

# Process

* Source code translation
  + Convert code to a new language.
* Reverse engineering
  + Analyze the program to understand it;
* Program structure improvement
  + Restructure automatically for understandability;
* Program modularization
  + Reorganize the program structure;
* Data reengineering
  + Clean-up and restructure system data.

# Cost factors

* The availability of expert staff for reengineering can be a problem with old systems based on technology that is no longer widely used.

# Refactoring

* Refactoring is the process of making improvements to a program to slow down degradation through change.
  + You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
* Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
* When you refactor a program, you should not add functionality but rather concentrate on program improvement.

# Refactoring vs. Re-engineering

* Re-engineering takes place after a system has been maintained for some time and maintenance costs are increasing.
  + You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable.
* 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 the costs and difficulties of maintaining a system.

# Potential problems to address in refactoring

* Duplicate code
  + The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
* Long methods
  + If a method is too long, it should be redesigned as a number of shorter methods.
* Switch (case) statements
  + These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program.
  + In object-oriented languages, you can often use polymorphism to achieve the same thing.
* Data clumping
  + Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
* Speculative generality
  + This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

# Legacy Systems

* It may not be cost effective to replace these systems because
  + They are implemented using outdated languages and technologies
  + The structure may have been degraded by change
  + The documentation might be missing

# Legacy system management

* Organizations that rely on legacy systems must choose a strategy for evolving these systems
  + Scrap the system completely and modify business processes so that it is no longer required
  + Continue maintaining the system
  + Transform the system by re-engineering to improve its maintainability
  + Replace the system with a new system.

# Legacy system categories

* Low quality, low business value
  + These systems should be scrapped.
* Low-quality, high-business value
  + These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
* High-quality, low-business value
  + Replace with OTS, scrap completely or maintain.
* High-quality, high business value
  + Continue in operation using normal system maintenance.

# Business value assessment

* Assessment should take different viewpoints into account
  + System end-users
  + Business customers
  + Line managers
  + IT managers
  + Senior manager

# Issues in business value assessment

* The use of the system
  + If systems are only used occasionally or by a small number of people, they may have a low business value.
* The business processes that are supported
  + A system may have a low business value if it forces the use of inefficient business processes.
* System dependability
  + If a system is not dependable and the problems directly affect business customers, the system has a low business value.
* The system outputs
  + If the business depends on system outputs, then the system has a high business value.

# System quality assessment

* Business process assessment
  + How well does the business process support the current goals of the business?
* Environment assessment
  + How effective is the system’s environment and how expensive is it to maintain?
* Application assessment
  + What is the quality of the application software system?

# Quality Measurement

* You may collect quantitative data to assess the quality of the application system
* Assess items such as
  + the number of system change request;
  + the number of different user interfaces used by the system
  + the volume of data used by the system

# Agile Approach

* Change points are place where you need to make changes in the legacy system

# Exploration

* Goal is to understand the app from both the customers' and developers' point of view
* Operating on a live app could endanger customer data or user experience
  + Checkout a scratch branch that you never intend to check back in and can used for experimentation
  + Clone the database
* Have customers demonstrate how they use the app, indicating the changes that they have in mind
* Use rake stats to get the total number of lines of codes which can tell you which classes are most complex and probably most important.
* Can also read design documents and tests

# Establish Ground Truth with Characterization Tests

* Characterization tests describe the actual, current behavior of a pieces of software
* Integration-level characterization tests are often easier to start with since they capture visible app behavior
* Unit and functional tests may be harder since we don't fully understand the code

# Bad Comments

* Most software is poorly documented
* A lot of code has no comments at all
* Many open source projects set a very bad example
* When present, comments often not helpful
* Many people think comments are a bad idea
  + Not useful
  + Out of date
  + Clutter the code
  + "Just let me read the code!"

# Comments

* The biggest problem is that most people don't know how to write comments.
* "My code is self-documenting" (one of the great lies)
* There are many other facets of writing good documentation, but this one rule will get you most of the way.

# Beyond Correctness

* Beautiful code is easier and less expensive to maintain
* How do you tell when code is less than beautiful and how do you improve it?
* Identify problems in two ways
  + Quantitatively using software metrics
  + Qualitatively using code smells

# Qualitative – Code Smells

* Code smell alerts you that something may be wrong
* There are 20 to 60 code smells depending on authority
* Methods that don’t follow this advice will often give off several of the smells
* Short – depends on language (More than 1 screenful for old monitors) – try to do one thing, and quickly grasp the one thing – may just call a bunch of methods do this
* Why are lots of arguments bad?
  + Hard to get good testing coverage
  + Hard to mock/stub while testing
* Single level of abstraction
  + Concentrate on Abstraction since correlated and Most rapidly makes your code start to make ugly, correlates well with other smells
  + Divide and Conquer: Don’t have one method that does everything. Divide into understandable pieces, and have methods call others
    - One that orchestrates all the work

# Quantitative

* ABC complexity
  + Branching terms are :and, :case, :else, :if, :or, :rescue, :until, :when, :while.
  + NIST – standard for gov't contracts
* Cyclomatic complexity
  + a graphical measurement of the number of possible paths through the normal flow of a program
  + Rails tool saikuro calculates cyclomatic complexity

# Leap Year Refactor

* Original
  + ABC score of 23   
    (>20 so a problem))
  + Gets code complexity score of 4 (≤ 10 so not a problem)
* Revised
  + Reek: No Warnings
  + Flog (ABC):
  + TimeSetter.convert = 11
  + TimeSetter.leap\_year? = 9
  + Saikuro (Code Complexity) = 5