* Maintenance activities
  - Corrective
  - Adaptive
  - Perfective

* Evolution: creating new but related designs → change in requirements [continuous change from simpler → better]
* Maintenance: means fixing bug: planned activity [does not involve making major changes to the arch] → post delivery

* Laws of Lehman:
[1] continuing change [2] increasing complexity [3] self-regulation [4] conservation of organizational stability
[5] conservation of familiarity [6] continuing growth [7] Declining quality [8] feedback system

* Why modifications are required?
- Market conditions
- Client requirements
- Host modification (change in hardware/software)
- organizational change

* Software maintenance
  - activities to make corrections
  - activities to make enhancements
    - change existing requirement
    - add new system requirement
    - change the implementation

* Software maintenance life cycle, [SMLC]
1. iterative model
2. change mini-cycle model
3. Staged model → long-lived software

* Software Configuration Management: (managing & controlling change)
  - reduce communication errors among personnel working on different aspects of s/w

* regression testing: we don't test the system as a whole
only small (main) functionalities are tested

Software Maintenance Activities

Intention-based

- **Corrective** (reactive)
  correct failures
  - processing failures
  - performance failure
- **Adaptive**
  adapt to changes
- **perfective** (re-engineering)
  make variety of improvements
- **preventive**
  Prevent problems from occuring
  (software rejuvenation)
  - resulting from continuously running the software system

Activity Based

- **corrections**
  fix defects in the system
- **Enhancements**
  effect changes to the system

Evidence based

Business Rule

- **Enhancive**
  adding & modifing business rules
- **corrective**
  correcting identified bugs
- **Reductive**
  * drop some data generated
  * decreasing the amount of data input

Software properties

- **Adaptive**
  Port the software to a different exec platform
- **performance**
  results that impact the user
- **preventive**
  Enhance maintainability
- **Groomative**
  substituting components with more efficient & simpler one

Documentation

- **updative**
- **Reformative**
  improve the readability of the documentation

Supportinterface

- **Evaluation**
  review the program code
- **consultive**
  cost & length of time of maintenance work
- **training**
  training the stakeholders

With respect to:

* the whole software system
* the program code
* the functionality experienced by the customer
* the external documentation

=> The default type is EVALUATING, if there are ambiguities in an activity.

Software maintenance has unique characteristics:

↳ constraints of an existing system
↳ shorter time frame
↳ available test data

* Maintenance process

↳ Maintained product:

↳ product: coherent collection of several artifacts.
↳ product upgrade: any change made to a software product.
↳ artifact: number of different artifacts are used in the design of software product

↳ Activity type:

↳ Activity: artifacts I/O → new/changed artifacts I/O
↳ Investigation activity: Evaluate the impact of making a certain change
↳ modification activity: change the system's implementation
↳ management activity: configuration control
↳ quality assurance activity: modifications performed don't damage the integrity of the system

↳ Maintenance organization process:

↳ Event management: CR are handled in an event management process
↳ change control: Evaluation of results of investigation
↳ configuration management: Integrity is maintained in terms of modification status & version number.

Maintenance models.

[1] Reuse-oriented Model

[a new version of an old system by modifying one or more components]

1. Quick fix model
   → necessary changes are quickly made [no impact analysis]
2. Iterative enhancement model
   → revises the highest-level document and propagates the changes down through the lower-level document.
3. Full reuse model
   → modified components + newly developed system.

[2] The staged model : [descriptive model]

→ improve understanding of how long-lived software evolves.

① Initial development
② Evolution
③ Servicing
④ phase out

[3] Change mini-cycle model

① CR
② analyze & plan change
③ Implement change
④ verify & validate
⑤ Documenting change

* Software Maintenance standart
  * IEEE/EIA 1219
    ↳ maintenance as a fundamental life cycle process
  * ISO / IEC 14764
    ↳ maintenance as an iterative process for managing & executing software maintenance

→ Analysis → design → implementation → system test
acceptance test → delivery

→ process implementation → problem & modification analysis → modification implementation → maintenance review & acceptance migration → retirement.

Software Configuration management functionality

↳ product → identification: items whose configurations need to be managed
→ version control:
master copy → centralized repository
working copy
conflicts can be resolved by: (1) lock-modify-unlock
(2) copy-modify-merge

↳ Tools → workspace.
① sandbox checked out files
② Building stores the differences
③ Isolation at least one workspace
→ building

↳ process → change management:
① understand the impact of modifications
② identify the products
③ provide tools
→ status accounting
→ Auditing
① audit for functional configuration
② audit for physical configuration

* change management request workflow

* Impact analysis: enables understanding impact of change via identifying the components that are impacted by the change Request (CR)
Reasons
→ 1. cost
2. Some critical portions of the system are going to be impacted or not.
3. Record the history of change
4. how items of change are related to the structure of the software
5. Determine the portions of the software that need regression testing.

* Traceability: is the ability to trace between software artifacts generated & modified
* Ripple effect analysis: a modification to a single variable may require several parts of the software system to be modified.
- It reveals what & where changes are occuring
- between successive versions & when a new module is added, measurment of ripple effect will tells us the complexi
* Error flow analysis: definitions of program variables involved in a change are considered to be potential source of errors
* Stability reflects the resistance to the potential ripple effect which a program would have when it is changed.
* change propagation: if any entity is changed, then all related entities in the system are cha
* Starting impact set (SIS): initial set of objects presumed to be impacted
* Candidate impact set (CIS): estimated to be impacted
* Actual impact set (AIS): set of objects actually impacted
* Discovered impact set (DIS): set of objects discovered to be impacted
* False positive impact set (FPIS): (CIS ∪ DIS) − AIS
* It is important to minimize the differences between AIS and CIS
* Recall: $\frac{|CIS \cap AIS|}{AIS} = \frac{3}{4}$ ✓ (3 ∈ CIS, 4 ∈ AIS; → 1 in DIS) Recall = 1 if DIS is empty
* Precision: $\frac{|CIS \cap AIS|}{CIS} = 1$ (3 ∈ CIS, 4 ∈ AIS) precision = 1 if FPIS is empty
* Adequacy is represented in terms of a performance metric called inclusiveness

$$\text{inclusiveness} = \begin{cases} 1 & \text{if } AIS \subseteq CIS \\ 0 & \text{otherwise} \end{cases}$$

* Effectiveness : is the ability of an impact analysis technique to generate results that actually benefit the maintenance tasks.

1. Ripple - sensitivity :

* DISO [directly impacted set of objects]
* IISO [indirectly impacted set of objects]
* Amplification is used as a measure of Ripple - sensitivity.

$$= \left| \frac{IISO}{DISO} \right| \rightarrow 1$$

2. Sharpness :

* Avoid having to include objects in the CIS that are not needed to be changed
* Sharpness is expressed by means of change Rate

$$\text{change Rate} = \frac{|CIS|}{|System|}$$

sharpness ↑ when change rate < 1

3. Adherence

* Adherence is expressed by S-Ratio

$$\text{S-Ratio} = \frac{|AIS|}{|CIS|}$$

range from 0 to 1, ideally 1

* The complexity of software can be decreased by improving its internal quality by restructuring the software
* Source code is restructured to improve some of its non-functional requirements, without modifying its functional requirements
* Restructuring applied on object-oriented software is called Refactoring
* Restructuring means reorganizing software (source code + documentation)
* External software value: fewer faults in software is seen to be better by customers
* Internal software value: well-structured system is less expensive to maintain (→ effort, cost, time)
* Activities in a refactoring process:
  1. Identify what to refactor
  2. Determine which refactoring(s) to apply
  3. Ensure that refactoring preserve the software's behavior (functional requirement)
  4. Apply the refactoring(s) to the chosen entities
  5. Maintain consistency
* Code smell: is any symptop in source code that possibly indicates a deeper problem
* Encapsulation: ① Increased modularity of the system
  ② It can be modified without modifying the classes

- Some refactorings must be applied together
- Some refactorings must be applied in a certain way
- Some refactorings can be individually applied, but, must follow an order if applied together
- Some refactoring are mutually exclusive

* The following two techniques can be used to analyze a set of refactorings, to select a feasible subset
  ① Critical pair analysis:
  Given a set of refactorings, analyze each pair for conflicts. A pair is said to be conflicting if both of them cannot be applied together
  ② Sequential dependency analysis:
  - If one refactoring has already been applied, a mutually exclusive refactoring cannot be applied anymore

* A Non-exclusive list of such non-functional
  1. Temporal constraints is that the operation occur in a certain order
  2. Resource constraints
  3. Safety constraints

* Two pragmatic ways of showing that refactoring preserves the software's behavio
  1. Testing [before & after]
  2. Verification of preservation of call sequence

* Refactorings impact both internal & external qualities of software
  * internal: size, complexity, ..
  * External: non-functional requirements

* If one kind of artifact is changed, then it is important to change some or all of the other artifacts so that consistency is maintained accross the artifacts

* The concept of change propagation is used to deal with inconsistency

* push down method: method to be put in child class

* push up method: method to be put in parent class

* Replace algorithm x with y:
  ① clearer ② Standards ③ easier

* Replace parameters with methods:
  ① Reduces the number of parameters
  ② Avoid errors while passing long parameter lists

- Software reuse
  1. Software development with reuse
  2. Software development for reuse

* "Development-for-reuse" process is used to create reusable software assets (RSA)

* What is typically reused?
  1. System reuse
  2. Application reuse
  3. Component reuse
  4. object and function reuse

* program families are a set of programs with several common attributes & features
* Domain and domain analysis: finding similar software systems in a specific problem
* The reusability property indicates the degree to which the asset can be reused
* For a software component to be reusable:
  1. Environmental independence
  2. High cohesion — single objective
  3. Low coupling — few or no dependencies
  4. Adaptability
  5. understandability
  6. Reliability
  7. portability

* Reuse models are classified as:
1. proactive: All conceivable variations
2. Reactive: reusable assets are developed if a reuse opportunity arises
   ↳ solid product architecture in a domain → more expensive
3. Extractive: reuses some operational & software products
   ↳ organizations have accumulated artifacts & experience in a domain

* Factors influencing Reuse:
  1. Management support
  2. Legal
  3. Economic
  4. Technical

* Reuse library: stores reusable assets and provides an interface to search the repository
  ↳ can be collected by: reengineer, design & build, purchase

* Reengineering is the examination, analysis & restructuring of an existing software system to reconstitute it in a new form & the subsequent implementation of the new form

Why do we do reengineering?

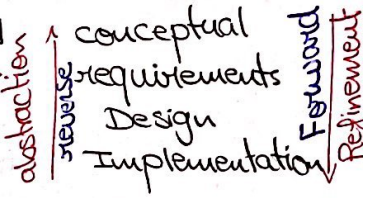
① understand existing software ② Improve functionality

• Reengineering concepts:

* Principle of abstraction: will gradually increased by successively replacing the details with abstract information
  ↑ reverse

* Principle of refinement: will decrease gradually by successively replacing some aspects of the system with more details
  ↳ does not involve any change in abstraction [no modification, addition, deletion of info]

* Principle of alteration: the making of some changes in representation..
  ↳ restructuring (refactoring)



Abstraction is represented by a "up-arrow"

Alteration is represented by a horizontal arrow

Refinement is represented by a down-arrow

* Reengineering is a sequence of 3 activities:

① Reverse engineering ② Re-design [Δ] ③ forward engineering
↳ alteration

* Alteration → change of functionality ⟹ Business rules
  ↳ change in implementation technique

• Types of changes

1. Recoding (rehosting) means reengineering of source code, with no addition or reduction
2. Redesign, restructure the architecture, modify data model, replace an algorithm with efficient one
3. Respecify, change the form or the scope of the requirements
4. Rethink, change the system's problem domain

* Software Reengineering Strategies:

1. Rewrite
2. Rework
3. Replace

* Reengineering approches.

The five approaches are different in:
1. Extent of reengineering
2. Rate of substitution with the new one

1. Big Bang
→ replacing the whole system at once
→ used if reengineering cannot be done in parts
→ Disadvantage: that the reengineering project becomes a monolithic task (not desirable)
↳ consumes too much resources

2. Incremental approach
→ reengineering gradually
→ large systems, several new interim versions are produced & released
→ Advantages: ① Locating errors
② Easy for the customer to notice progress
→ Disadvantages: ① Take much longer time to complete [due to documentations]
② The architecture cannot be changed

3. Partial approach
→ only a part of the system is reengineered and then it is integrated with the non-engineered
→ Advantages: Less time & cost
→ Disadvantages: the interface between the reengineered & the non can't be modified

4. Iterative approach
→ few procedures at a short time
→ Advantages: ① continued operation of the system
② The maintainers & the users are still familiar with the system

5. Evolutionary approach
→ functional objects
→ components of the original system are substituted with re-engineered components
→ Advantages: ① More cohesive
② Scope of components is reduced
→ Disadvantages: ① similar functions are identified & refined as one unit in the new system

* Data base Reverse engineering (DBRE)
1. Conceptual phase — user req. → conceptual schema — ① Logical schema: describes the data structure ② physical schema: implements the logical schema
2. Logical phase — conceptual schema → simple model → optimization reasoning
3. Physical phase — logical schema → data description language

* DBRE → Backword execution of the logical & physical phase
1. Data structure extraction
2. Data structure conceptualization