Final Stages

- Initial development
  - first functioning version
- Evolution
  - software changes
  - evolution ends
- Servicing
  - servicing patches
  - servicing discontinued
- Phase-out
  - switch-off
- Close-down
End of Software Evolution

- Software stabilization
- Code decay
- Business decision
Software stabilization

• Software domain and technology of the project are stable
• Volatility of stakeholder knowledge is the only reason for evolution
  – stakeholders explore the problem by trial-and-error
• Stakeholder knowledge reaches a point of stability
  – further substantial changes are unnecessary
Example stabilization

- Real-time systems
  - for example a car ignition
- Domain is the specific car engine
  - once produced, it does not change
  - domain is completely stable
- Stakeholders learning is the only reason for evolution
  - once the learning is completed there is no reason for further evolution.
Code decay

• Software is not material
  – not subject to wear and tear
• Its structure decays under the impact of software changes.
Symptoms of code decay

• Increased difficulty of software changes
  – confused contracts between suppliers and clients
  – concept extensions delocalized into several classes
  – unexpected or illogical coordination among the classes

• Decreased quality
  – an increase in the presence of bugs
Decayed structure of software

- Does not support concept location
  - code does not have meaningful identifiers for grep
  - does not have clean contracts for the dependency search
- Does not support unit testing, code inspection
  - the changes become increasingly difficult and risky
Loss of knowledge

• Programmers must understand the domain
• They also must understand the architecture, the algorithms, and data structures of the code
• Refactoring also requires a thorough knowledge of software
• If that knowledge is lost, even refactoring is no longer possible and the system further decays
Knowledge transfer

• Part of the knowledge is recorded in program documentation,
• Knowledge is of such a size and complexity that a complete recording is impractical
• Tacit individual or group knowledge
  – tacit knowledge is constantly at risk.
  – software engineers who are leaving the project are taking their knowledge with them
Loss of knowledge through cultural change

• There are programs still in use that were created a half century ago.
  – created in a context of old hardware
  – written in obsolete languages
  – elaborate algorithms deal with limitations
  – programmers who created these programs are no longer available
  – new programmers may not understand old techniques and old culture
Knowledge gap

• Decay makes the code more complicated
• Knowledge necessary for evolution increases
• Gap between the knowledge of the team and the growing complexity of software may become too large
  – the software evolvability is lost
  – the system slips into servicing
Management decision

• Managers decide that software does not deserve new investments
• It still retains a value to its current users
  – the purpose of the servicing is to protect that residual value
• Threats to that value can come from
  – the bugs that surface during the use
  – changes in the technology or domain to which the software must adapt
Example Y2K

- Wrong assumption about the life-span
  - lifespan will end before January 1, 2000
- Year represented by two digits
  - February 23, 1978 = 02, 23, 78
  - January 1, 2000 = 01, 01, 00
    - the programs interpreted this as January 1, 1900
    - correction became a huge worldwide concern
    - The change protected software value beyond January 1, 2000 deadline.
Wrappers

- Wrapping deficient functionality
- Structure further deteriorates
Processes

- SIP, AIP, DIP, CIP
- No need for postfactoring
  - Changes are corrective and adaptive
Phase-out

• No new updates to the software
  – if software is still in use, the users must work around the deficiencies.
    • work-around.
    • owners of the software still may provide a help
    • the bulletins that describe how to work around the known software bugs

• Management may slowly withdraw servicing
  – Gradual start of phase-out
Close-down

• Users stop using the software
  – work done by the system may no longer be needed
  – software was already in the stage of phase-out but a new problem appeared and pushed the software over into close-down
  – a new and better software replaced the old one.
Phases of close-down

• Users who used the old system must be retrained to use the new system

• Persistent data need to be preserved
  – customer records, birth certificates, student transcripts
  – the old data are migrated to the new system
    • may involve data conversion
    • the quality and integrity of this conversion must be evaluated
Management options

- Decayed code
- Maintain status quo
- Retire system
- Reengineer
Reengineering

- Reengineering = reverse engineering + forward engineering

- Reverse engineering
  - Code $\rightarrow$ abstraction
    - Design
    - Knowledge

- Forward engineering
  - Abstraction $\rightarrow$ code
Reengineering

extracted knowledge

old code

old

new design

new code

new
Reengineering process

• Whole-scale
  – Puts big resources at risk
  – Tolerance of the users is low
    • Users expect the same functionality
  – Systems are huge
    • Whole-scale reengineering takes too long
    • Maintaining two systems in parallel
    • Expensive
    • May be obsolete before being used

• Incremental
Incremental reengineering

• Results are immediate and concrete
  – less risk
• Flexible
  – efforts can be redirected
• Justification is easier
• Disadvantages
  – preserves some of the old structure
Heterogeneous software

- Cashiers
- Store
- Inventory
- CashierRecord
- Session
- Sale
- Item
- Payment
- SaleLineItem
- Price
- Cash
- Check
- Charge
- PromoPrice
Reengineering interleaved with evolution

• The product backlog consists of the following tasks:
  – servicing requests
    • require only small changes
  – evolutionary requests
    • involve evolvable modules
    • involve decayed modules
  – requests to reengineer decayed modules

• Processes: SIP, AIP, DIP, CIP
Advantages

• Avoids disruption of the user’s routine
  – results of the reengineering are immediate, concrete, and limited in scope
  – if anything goes wrong, the efforts can be reversed and redirected

• Justification to reengineer is easier
  – the expenses of the individual reengineering tasks are limited.
Four classes of components

- Software = business rules
- Data
- Platforms
  - hardware, operating systems, compilers …
- Support for interaction with the program
  - GUI, sensors, …
Gateways

• Support coexistence of the old and new parts in the same system
• Map the functionality and format of messages between new and old
Example

- Interface
- Gateway
- Old software
- New software
- Gateway
- Old data
- New data
Redevelopment

• Code is rewritten by the programmer
  – the old code is no longer usable
• unit or functional tests are preserved
  – guarantee that the old and new code has the same functionality.
Preserve the knowledge

• The knowledge of the domain is often collected in the code
• It is not recorded anywhere else
  – example of such knowledge is the knowledge how to calculate insurance premiums
• This knowledge must not be lost
  – it is extracted from the old program and used in the new one