After Deployment

Evolution - Maintenance 9.1-9.3
Configuration management 25
Legacy systems 9.4
Re-engineering 9.3.2

Emergency code repair Fig 9.6

- Leave change request open
- Follow normal change request routine

Impact analysis

- Change Control Board
  - benefits of the change
  - number of users affected
  - what if no change?
  - cost
- If change:
  - priority
  - fit in release cycle
Configuration items

- requirements
- design documents
- code - modules
- test suites
- documentation
- installation files/routines

Terminology

- **Version**
  - of an item
  - unique identifier
- **Baseline**
  - collection that cannot be changed (fall-back)
- **Release**
  - delivered to customer

Tool support

- **Database**
- **Editing**: check out ... check in
- **System build**
- **Regression test**
- Change reports, documentation

**Figure 25.12 Continuous integration**

**Figure 9.1 A spiral model of development and evolution**
Evolution Dynamics (Lehman)

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Continuing change</td>
<td>A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.</td>
</tr>
<tr>
<td>7. Declining quality</td>
<td>The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.</td>
</tr>
<tr>
<td>3. Continuing growth</td>
<td>The functionality offered by systems has to continually increase to maintain user satisfaction.</td>
</tr>
<tr>
<td>2. Increasing complexity</td>
<td>As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.</td>
</tr>
</tbody>
</table>

Constant pace of change

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8. Feedback system</td>
<td>Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.</td>
</tr>
<tr>
<td>3. Large program evolution</td>
<td>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.</td>
</tr>
<tr>
<td>4. Organisational stability</td>
<td>Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.</td>
</tr>
<tr>
<td>5. Conservation of familiarity</td>
<td>Over the lifetime of a system, the incremental change in each release is approximately constant.</td>
</tr>
</tbody>
</table>

Maintenance costs

- Maintenance costs more than development
  - loss of information
    - time
    - handovers
  - less skilled people
  - structure gets worse
- It pays to invest in maintainability
  - refactoring

Refactoring 9.3.3

- During development (evolutionary, incremental, agile)
- During maintenance
  - "code smells"
  - design patterns
  - documentation

Legacy systems

- Old systems
  - > 10^11 LOC
  - date back to 70’s
- Hardware no longer available
  - "don’t touch it" not an option
- Business rules implicit in software
- Data – a lot of it!
  - only accessible through this system
Legacy software

- Documentation lost (not maintained)
- Design – not modular
  - overoptimized
  - user interface (command line)
- Code – source code lost
  - old language
  - unstructured
  - badly patched

Re-engineering

- Goal:
  extract what we must / can reuse:
  - knowledge: business rules
  - data: conversion
  - design, code?
- Why?
  - reduce risk
  - reduce cost

Legacy system wrapper

- Even if you keep the legacy system, ...
- how to interface with new systems

Figure 9.13 An example of a legacy system assessment

Figure 9.11 The reengineering process

Legacy system wrapper provides interfaces