# Software Maintenance

## PSU CS 300 Lecture 10-2b

**Bart Massey**
**Assoc Prof Computer Science**
**Portland State University**
**<bart@cs.pdx.edu>**

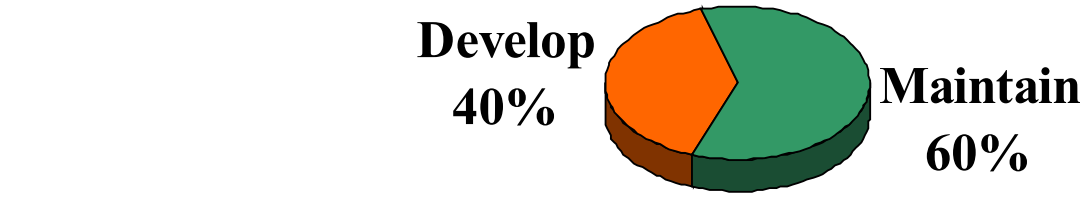
# Message

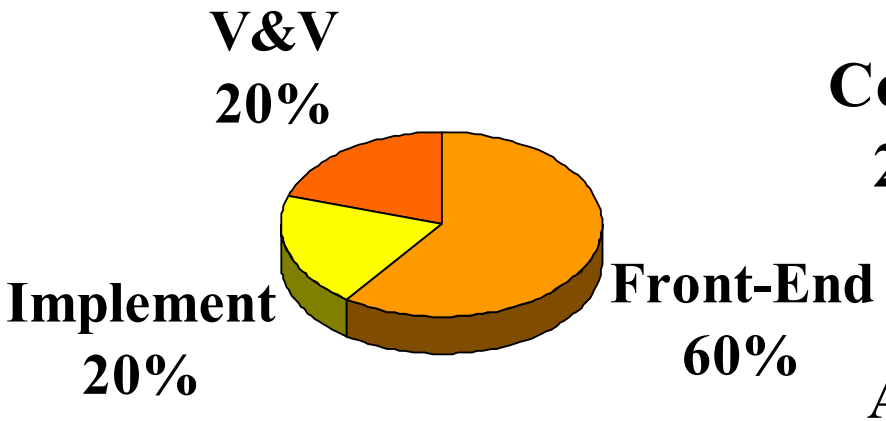
- **Maintainability is a crucial component of *quality***
- **Maintenance must *preserve* quality**

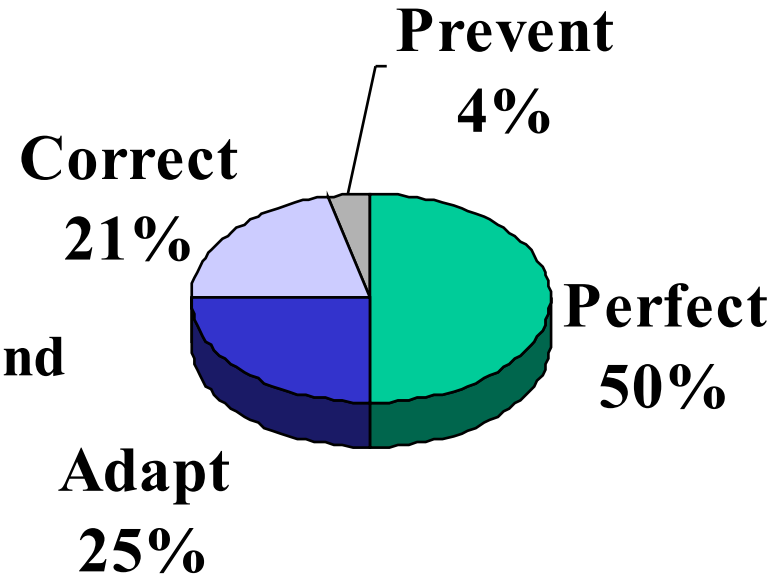
# Software maintenance

- Maintenance effort
- Maintenance activities
- Construction for maintenance
- Managing maintenance

# Maintenance effort



# Kinds of maintenance

- **Corrective: post-failure defect repair**
- **Perfective: upgrade or improve function**
- **Adaptive: accommodate other changes**
- **Preventative: repair faults before error**
- **Re-engineering**

# Faulty software

- **Software does not “break”: software defects are caused by human error**
  - **during dev, maintenance**
- **High *initial* corrective maintenance = poor quality system**
- **Corrective maintenance should be small portion of total development**

# Corrective maintenance

- "Bug-fixing" after failure
  - identify fault causing failure
  - identify fault injection point
  - correct problem *at this point*
- "Heisenbugs" happen anyhow
  - regression test
  - preventative maintenance

# Software grows or dies

- **No requests for improvement = unused (probably useless) system**
- **System growth can be blessing or curse**

# Perfective maintenance

- **"Just like" corrective maintenance!**
  - **identify feature injection point**
  - **build in new functionality from this point**
- ***Important:* you will do this a lot**
  - **inject no new faults**
  - **organize and simplify**
  - **rework now rather than later**

# Disorder and complexity

- The “KISS” principle:
  “Keep It Simple, Stupid”
- Some systems inherently complex:
  unavoidably complex designs
- Maintenance leads to different complexity:
  entropy = increasing disorder
- Disorder eventually kills software

# Adaptive maintenance

- Adapt software to change
- After corrective or perfective maintenance
  - natural given correct level
  - counters tendency to disorder
- After change in environment
  - platform change, tool change

# Preventive maintenance

- **Prevent failures**
  - **removing defects before failure**
  - **anticipating and “vaccinating” against faults**
- **Robustification**
- **Oft-neglected**

# Re-Engineering

- **Eventually, software dies from**
  - **disorder**
  - **dramatic requirements change**
- **Goal: reuse legacy materials in new systems**
  - **automated tool assistance**
  - **all work products: requirements, designs, code, tests**

# Construction for maintenance (1)

- **Planning for maintenance**
  - **resources, costs**
  - **facility**
- **Requirements**
  - **prioritized optional requirements**
  - **complete set of system tests**
  - **traceability to tests**

# Construction for maintenance (2)

- **Architectural Design**
  - **modularity, low coupling, information hiding**
- **Detailed Design**
  - **solid set of unit tests**
  - **modularity**
- **Implementation**
  - **readable, modifiable code**

# Managing maintenance

- **Configuration Management**
- **Maintenance CM**
- **Impact Analysis**

# Configuration management

- **Configuration: set of all baseline work products**
  - **must identify**
  - **must protect**
- **Configuration Management (CM) uses**
  - **Change Control Board (CCB)**
  - **software assistance**

# Internal and external CM

- **Internal CM: during product dev**
  - **control evolving work products**
  - ***project* CCB includes: project, marketing manager, project architect, team leads**
- **External CM: after product release**
  - **control released version of product**
  - ***product* CCB includes: product, marketing, maintenance manager, project architect**

# Maintenance CM

- Two sources of change
  - discovered defects
  - requirements changes
- Handled using specified workflow (process)
  - via CCB
- Tracked using CM

# Impact analysis

- Given: traceability info.
- Find: what must change
- Injects change at proper point
- Enables estimation of repair costs and problems
  - deliberate programming: *e.g.*, search failed
- Must design to avoid or handle

# Principled maintenance pays

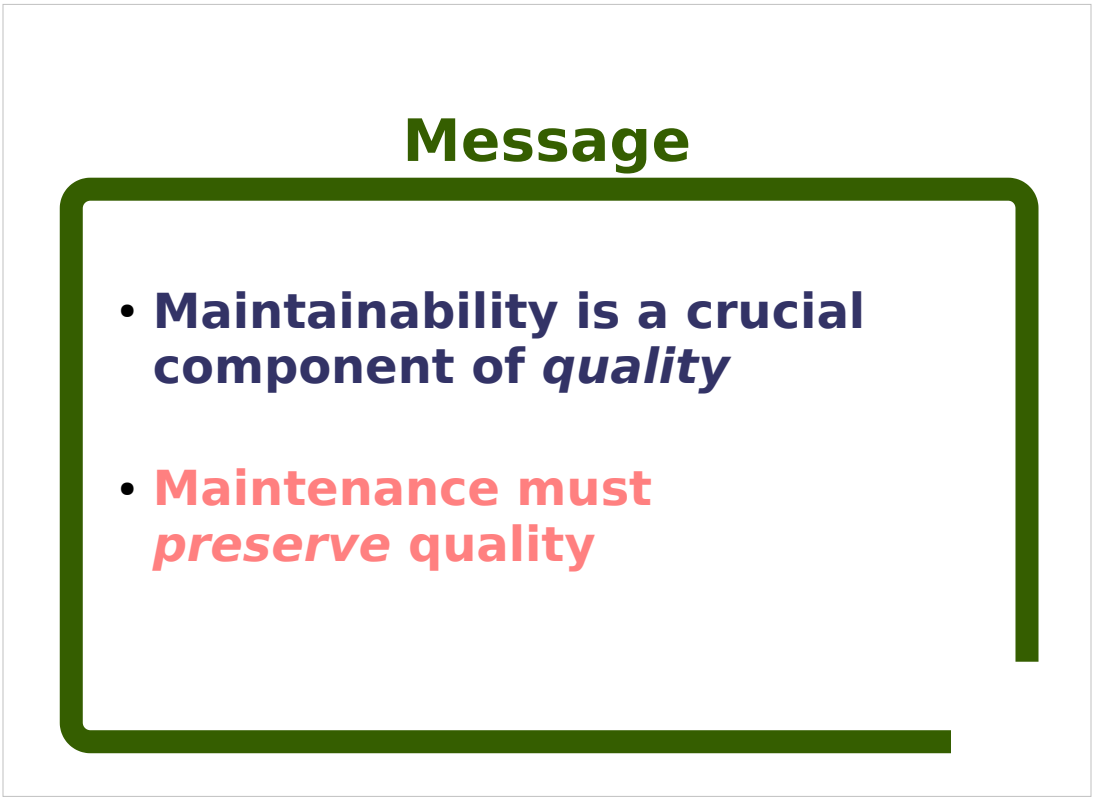
- **Good maintenance practices:**
  - **root cause analysis**
  - **impact analysis**
  - **controlled change**

  **may make the difference between**
  - **successful software**
  - **temporarily useful artifact**

# Software Maintenance

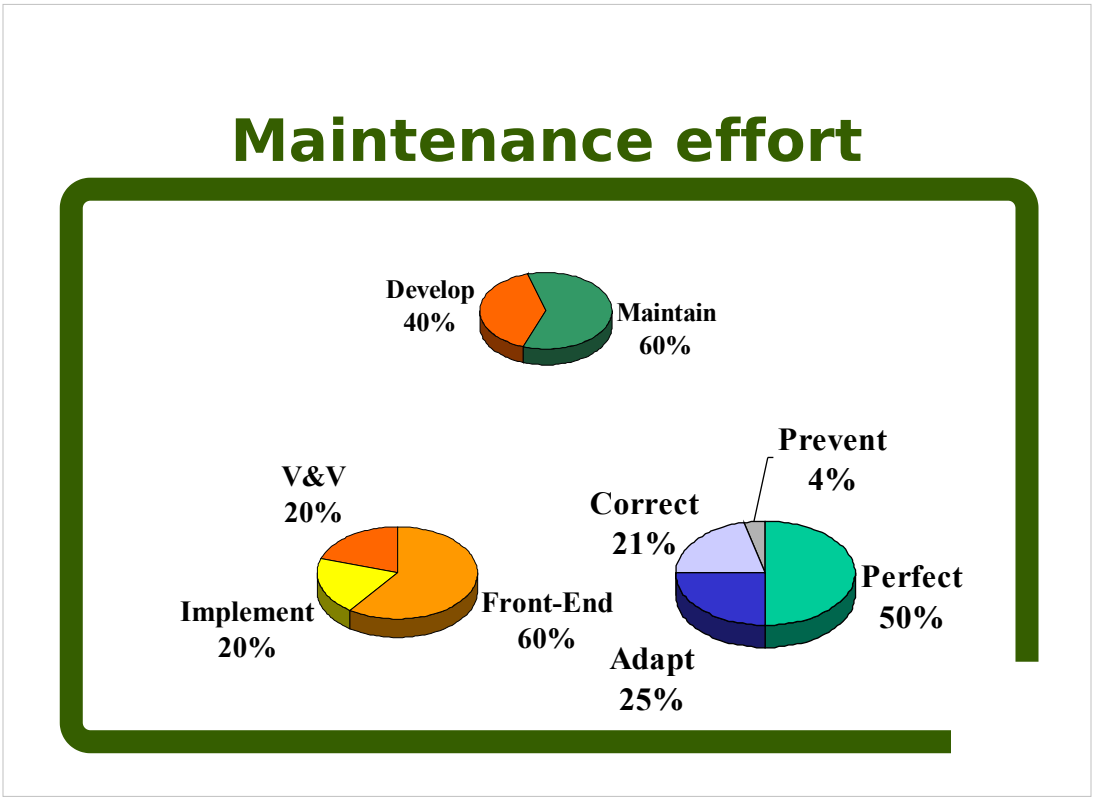
## PSU CS 300 Lecture 10-2b

**Bart Massey**
**Assoc Prof Computer Science**
**Portland State University**
**<bart@cs.pdx.edu>**



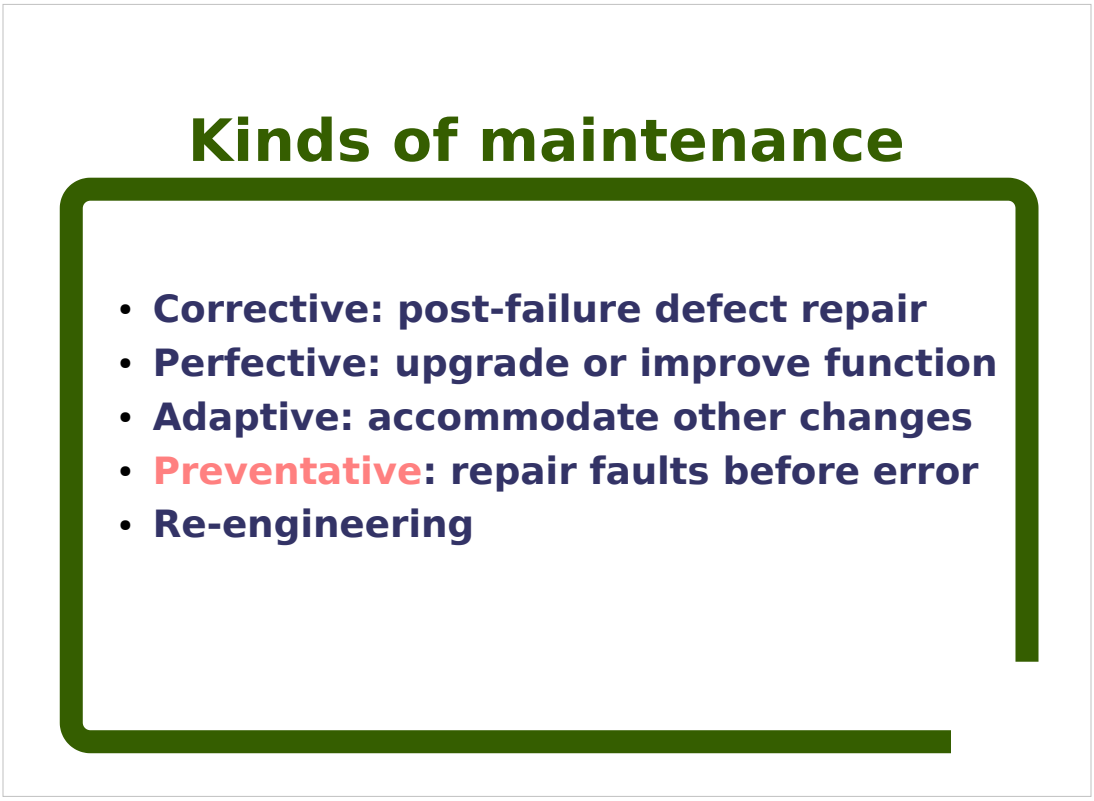

An unmaintainable piece of software is a useless piece of software. There are two ways for a piece of software to become unmaintainable: (1) have the unmaintainability built in, or (2) have it added after the fact. Both are common.


Maintenance programming = programming under strong constraints. A "Project BIFF" situation: improve the program without spending any effort or changing anything.

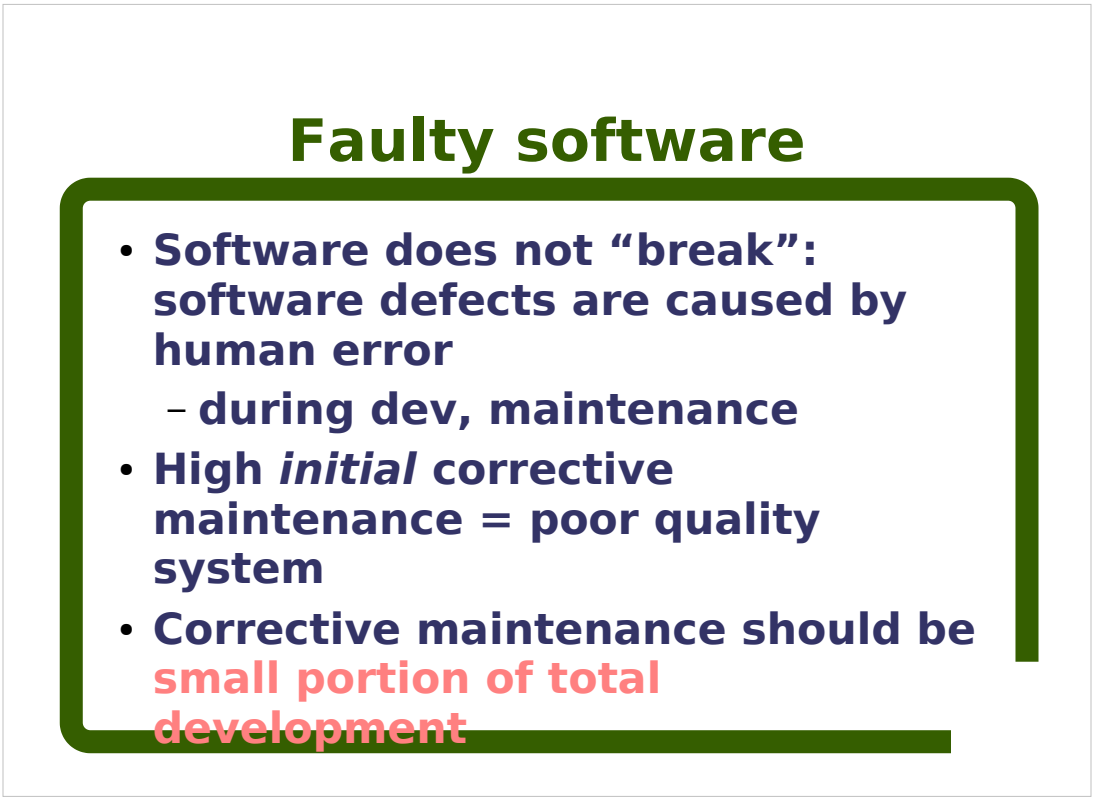



Note that development is only 40% of the effort, yet we only spend part of one lecture on this course on maintenance. This is because less is known about maintenance, because good development will produce projects which are easy (but onerous) to maintain, and because "perfective maintenance" resembles incremental development so strongly anyhow.

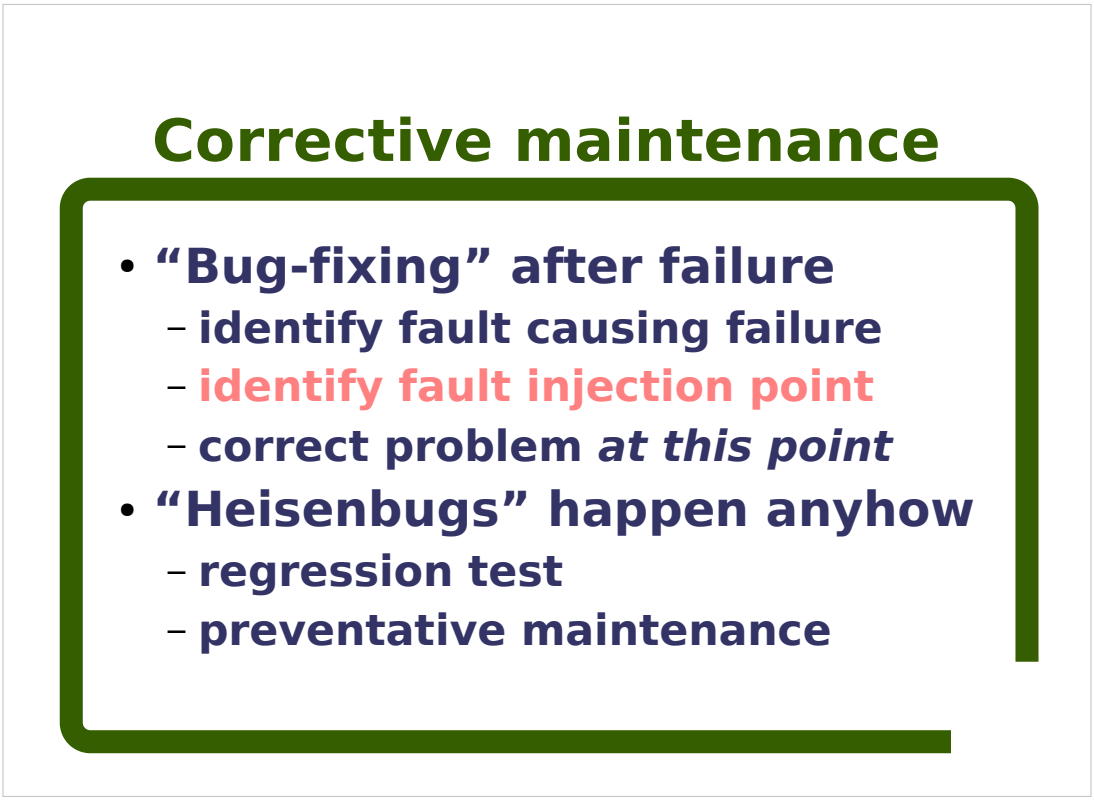
The maintenance numbers are "typical" numbers from the study in the Pfleeger book, not "best-practice" numbers. It is hard to say what best-practice should be: surely preventative maintenance should be larger, and ideally corrective maintenance should be 0.




Re-engineering is not really a kind of maintenance, of course, but many of the considerations are common.



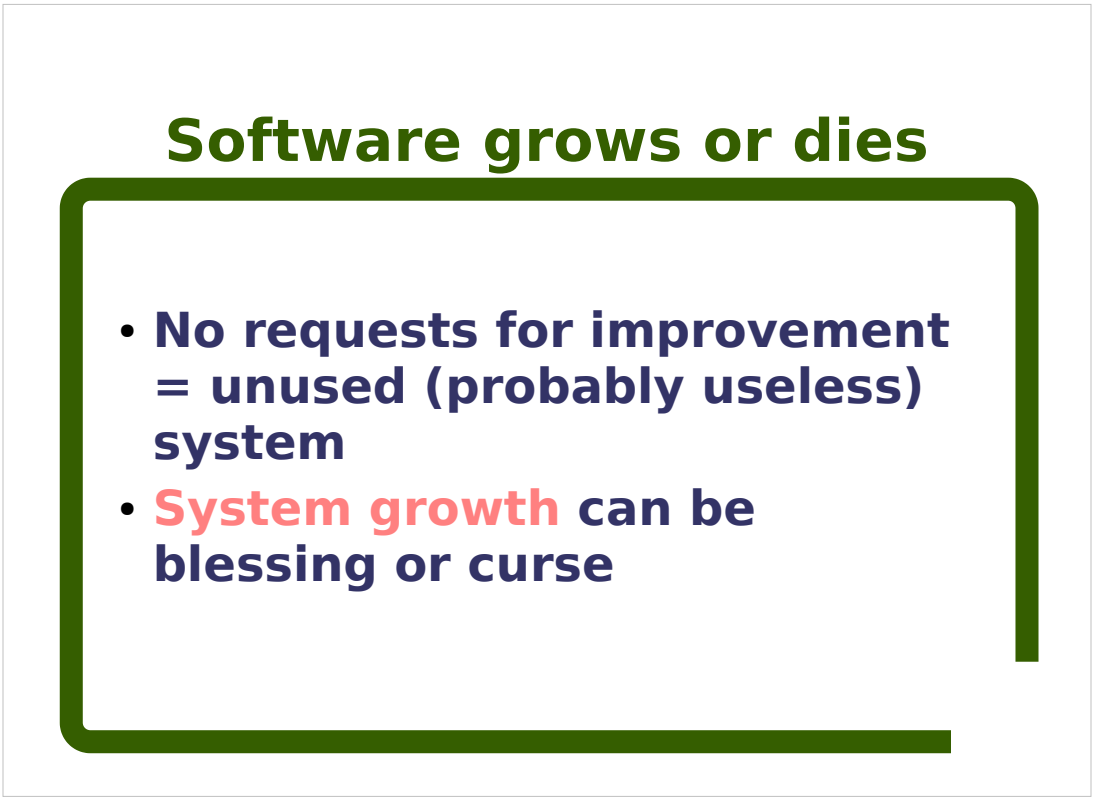

There is great danger that the design will creep horribly in response to major early bug-fixing.




If you are trying to fix, *e.g.,* a design bug, it should be fixed in the design, then propagated downward.

"Heisenbug" = fault expressed by perturbing the system while trying to debug it.

We will talk about regression testing next week.

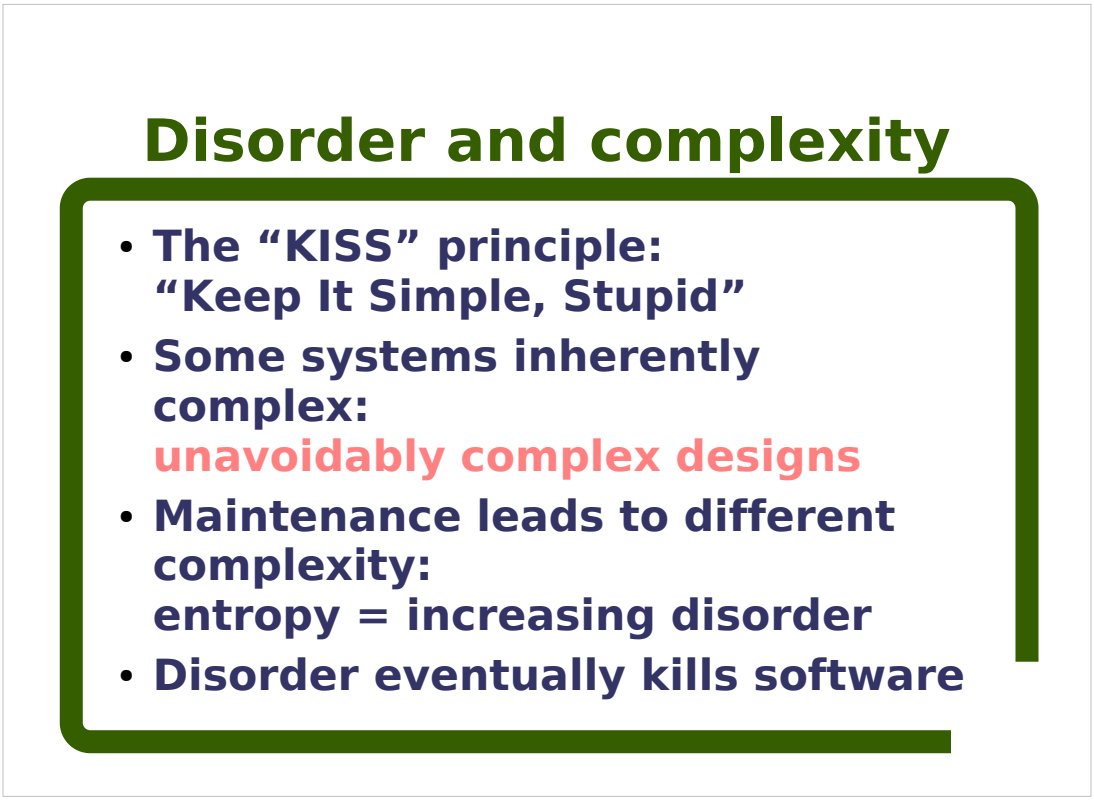



A properly-grown system in a stable environment will eventually match that environment beautifully. An improperly-grown one will be a mismatch and an internal mess.

## Perfective maintenance

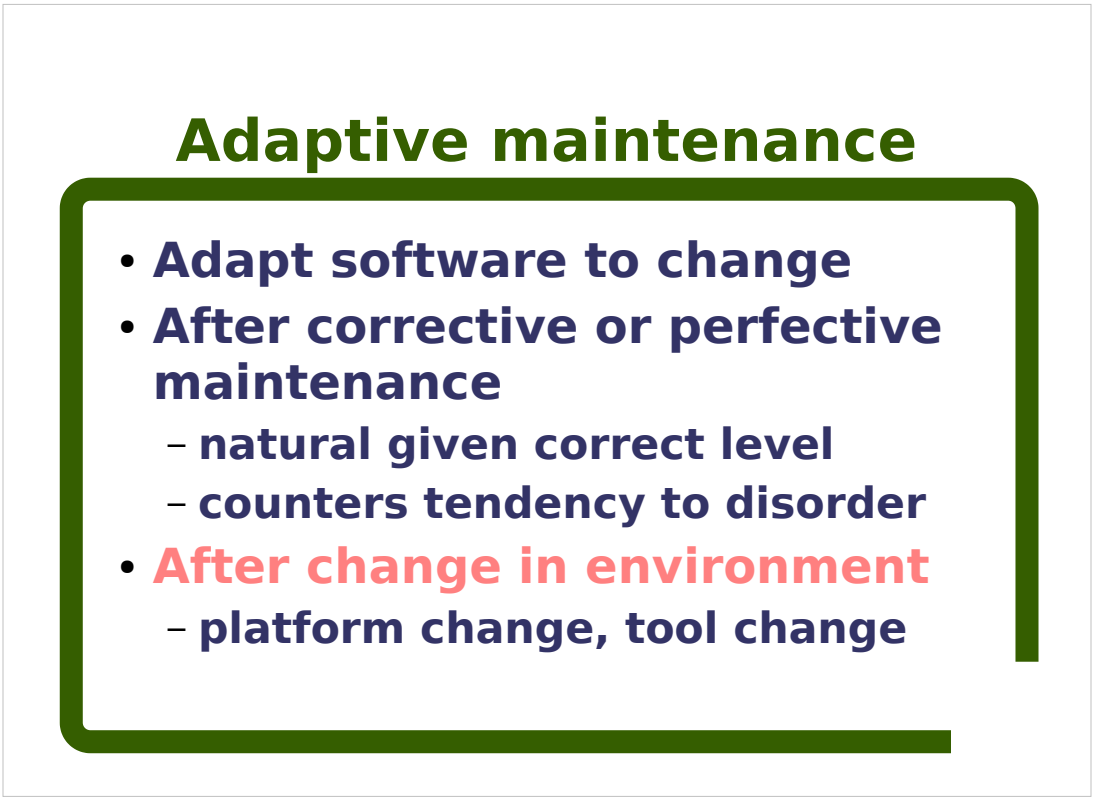
- "Just like" corrective maintenance!
  - identify feature injection point
  - build in new functionality from this point
- *Important:* you will do this a lot
  - inject no new faults
  - organize and simplify
  - rework now rather than later

This is where loose coupling, reasonable unit tests, and good higher-level work products really pay off.

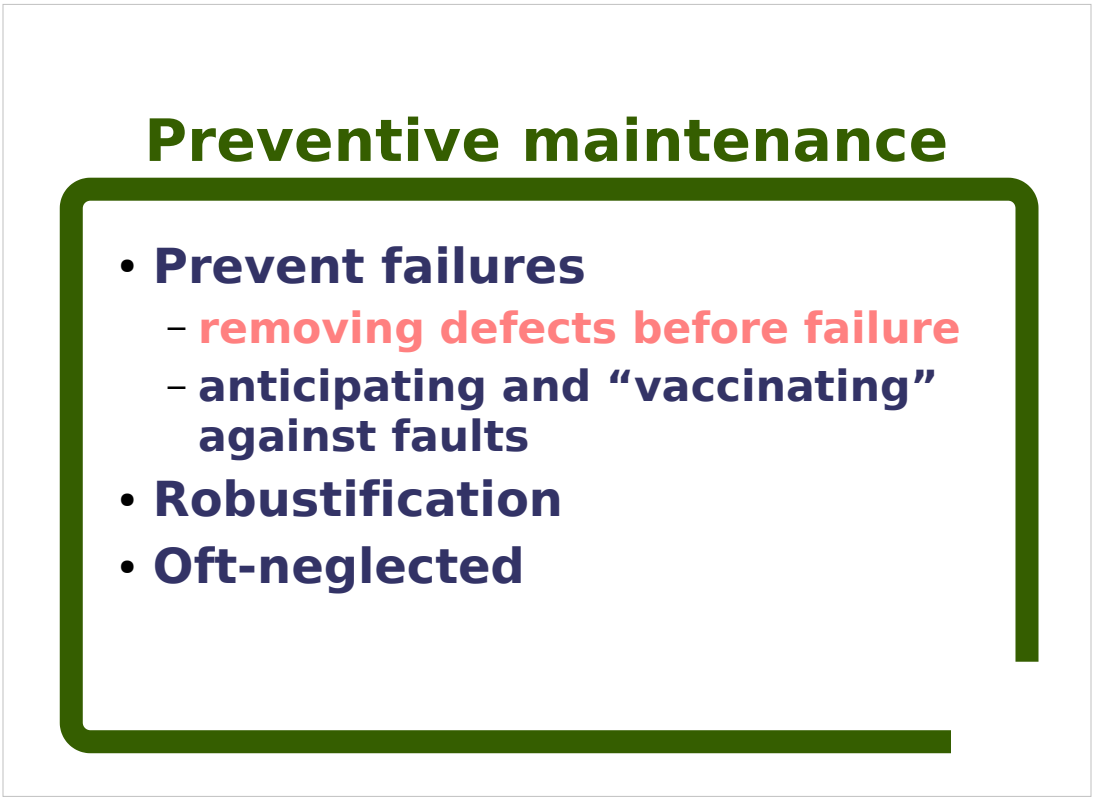



Not, as I recently overheard, “Keep It Simple and Stupid.”

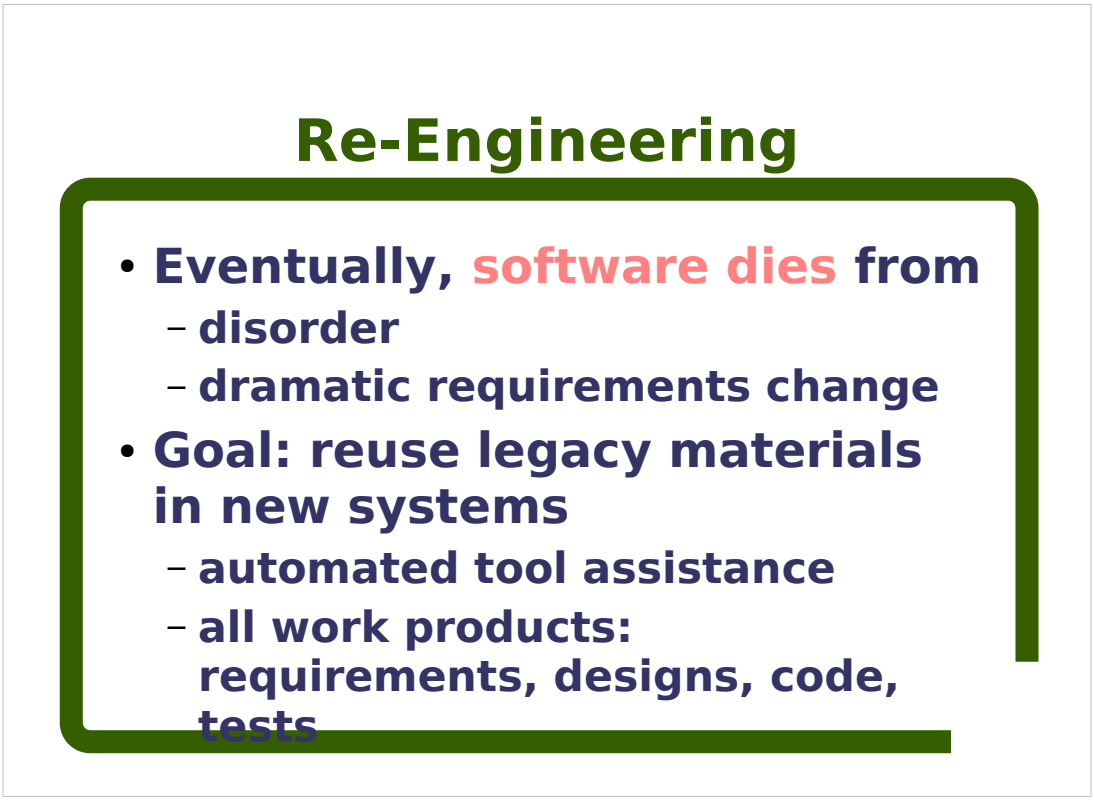
Complexity is almost always bad, but if a system must be complex, it should at least be designed complex, so that one has a chance of understanding it.




Robustness in design and implementation limits the need for adaptive maintenance. However, you may want to still do preventive maintenance to maintain this robustness.

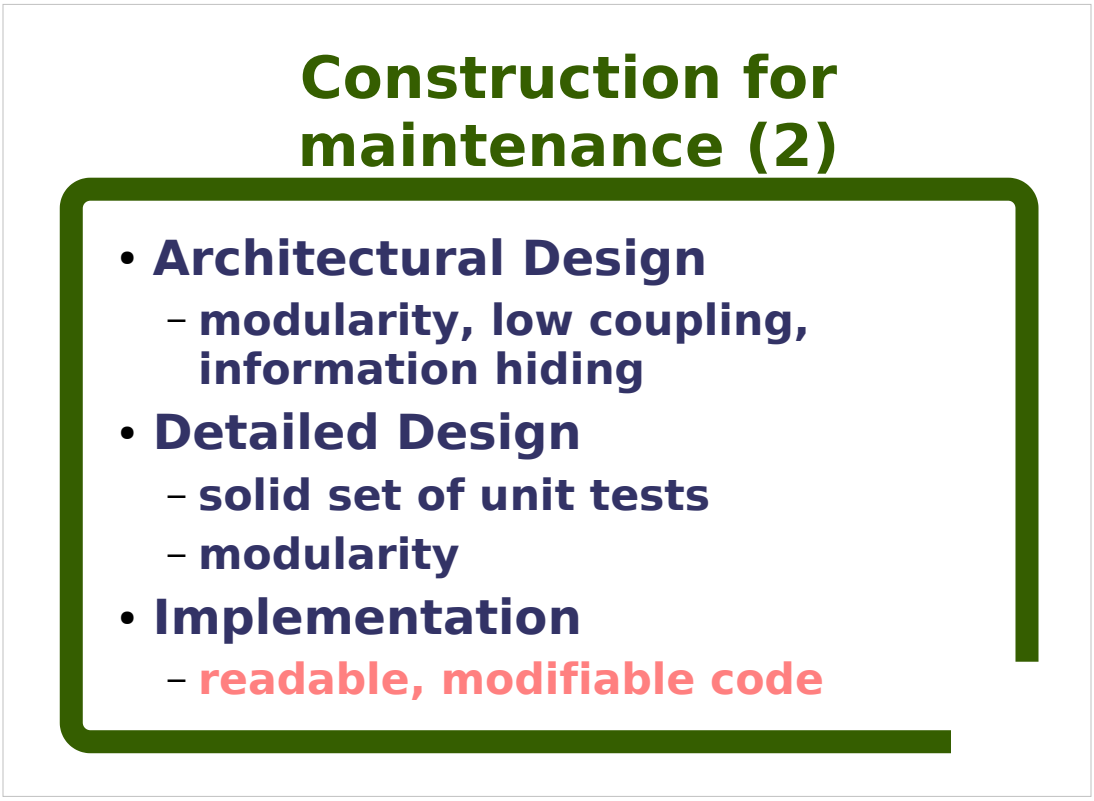



Management has a responsibility to allot appropriate time to this activity, and to listen to engineers who express needs for it. An ounce of prevention is worth a pound of cure in software maintenance too.

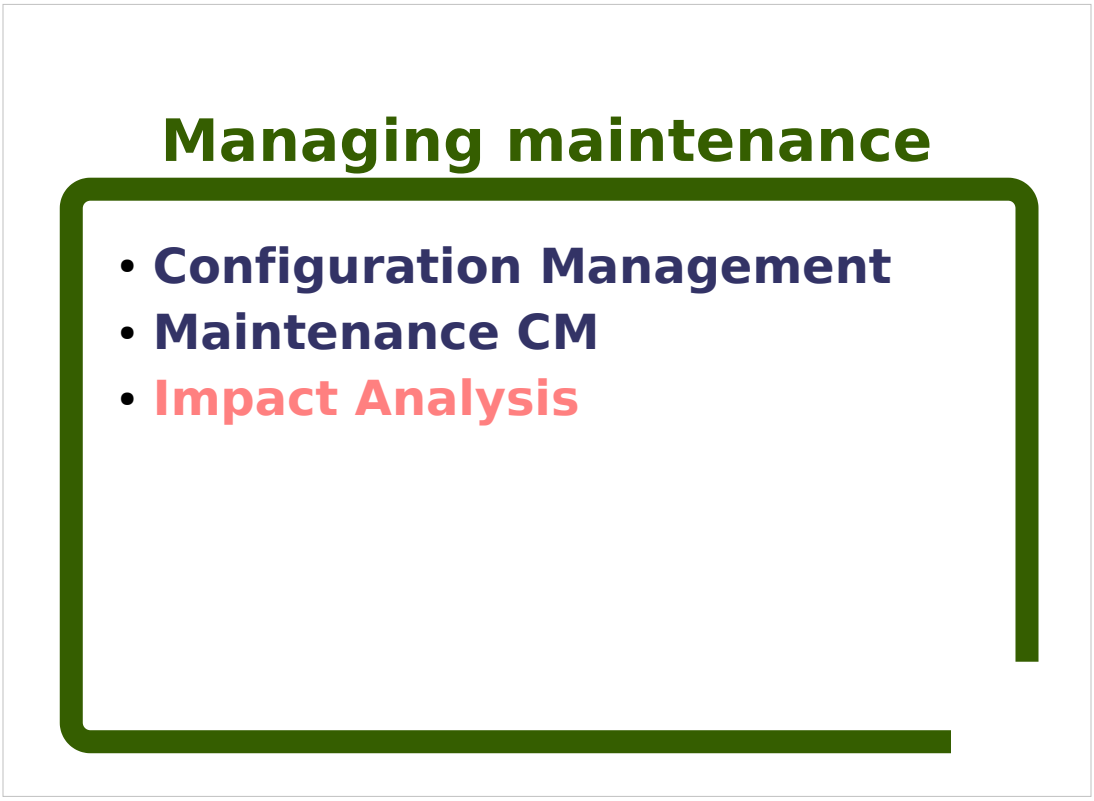



It is often hard to say when a product should be discarded. Having automatic ways to salvage work makes it more attractive to do these analyses.

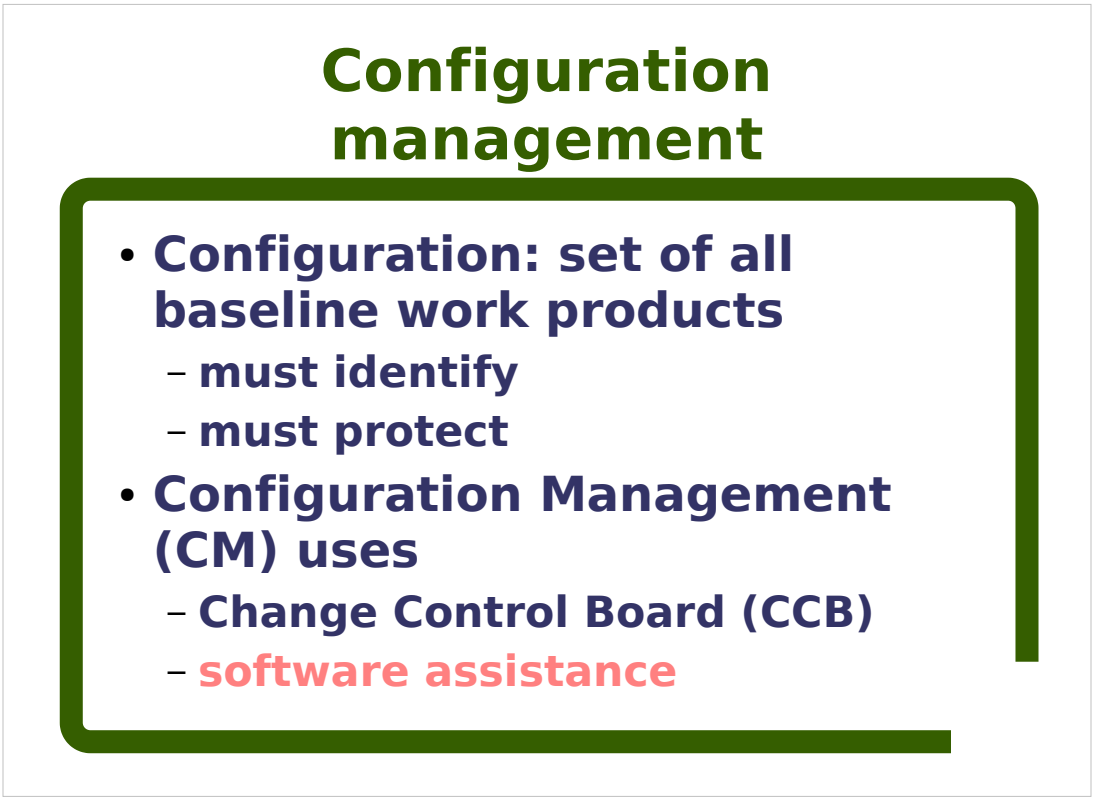

These are just some of the obvious things that can be done in this regard. Most of them are good development practice anyhow.



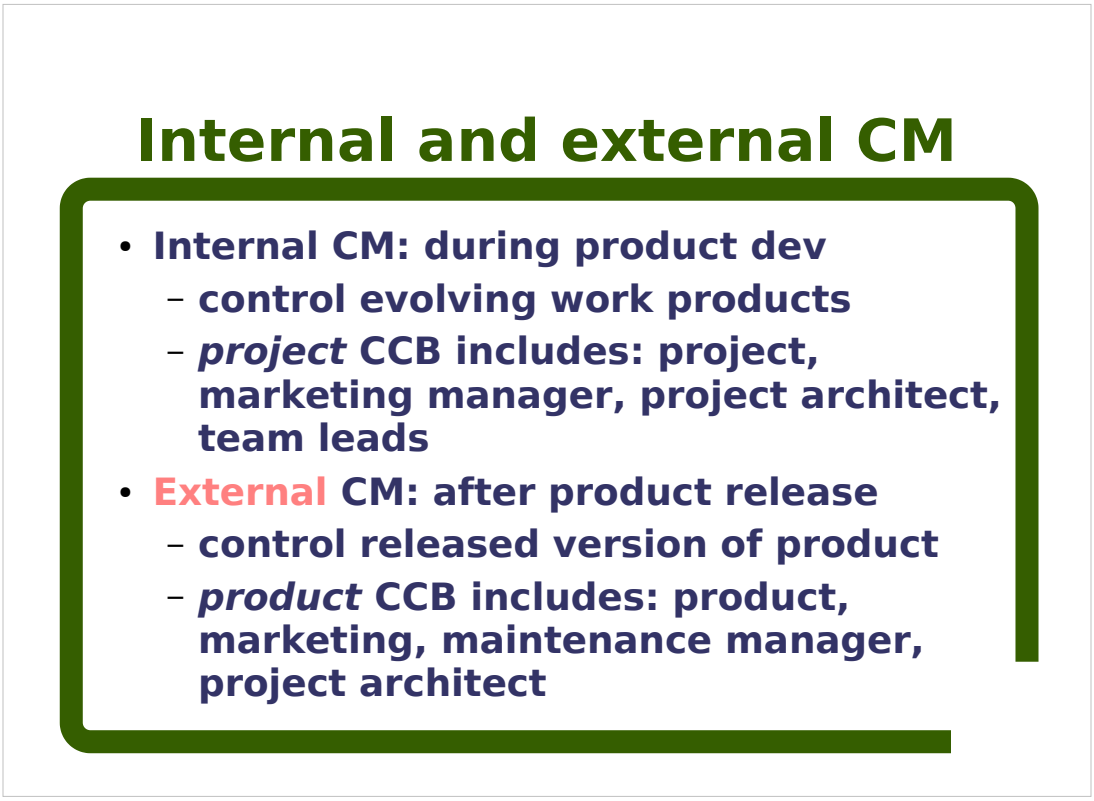

Building architects and designers spend a lot of time thinking about maintainability...



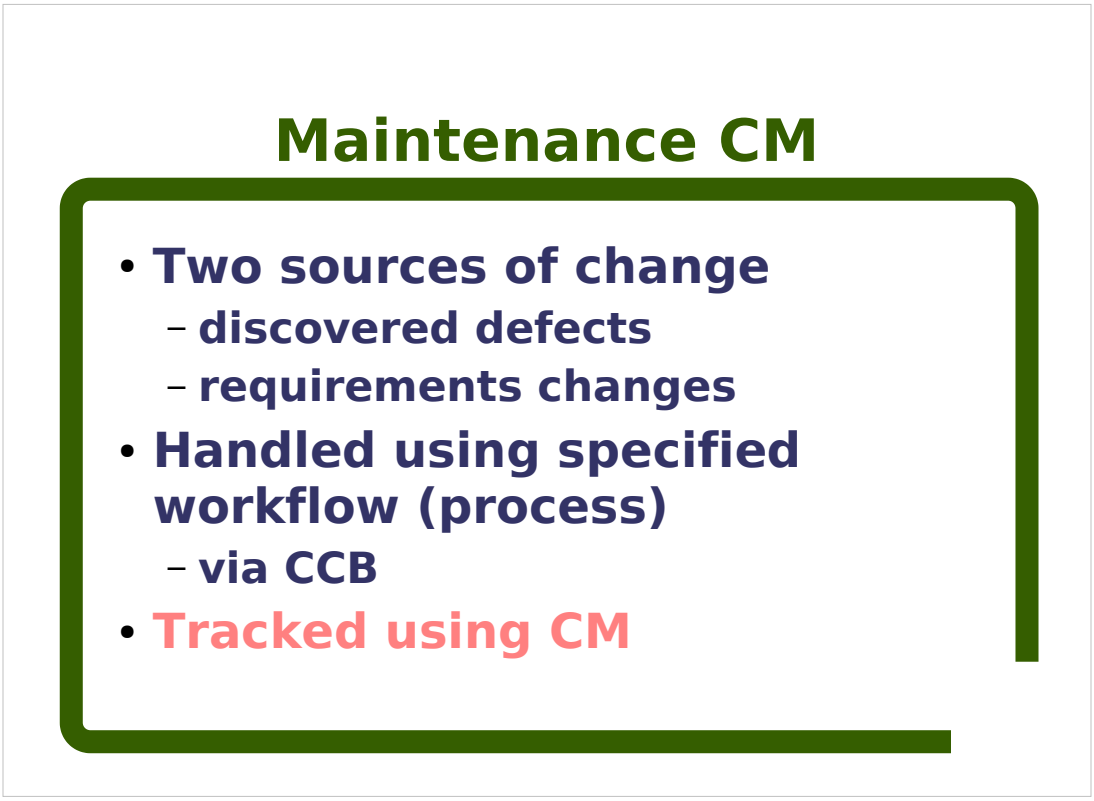

“Configuration” here means something different than “product version”. I try to avoid the word “version” because it sometimes means “revision” and sometimes “build”. All should be clearer in a moment.




The CCB should be explicitly represented in the overall workflow model of a software development project.




These activities are more different than first thought might indicate. In particular, the external CM includes “build” elements...




One can often distinguish good from bad organizations by looking at their mechanism for dealing with ECPs: most organizations these days deal with SPRs reasonably, but in my experience few extend the same discipline to changes.

## Impact analysis

- Given: traceability info.
- Find: what must change
- Injects change at proper point
- Enables estimation of repair costs and problems
  - deliberate programming: *e.g.*, search failed
- Must design to avoid or handle

We have already discussed this some.

## Principled maintenance pays

- **Good maintenance practices:**
  - **root cause analysis**
  - **impact analysis**
  - **controlled change**

  **may make the difference between**
  - **successful software**
  - **temporarily useful artifact**