## Operation and maintenance

> “Normally, this is the longest life cycle phase”

### Usually involves :

- **Correcting** errors which were not discovered in earlier stage of the life cycle
- **Improving** the implementation of system units
- **Enhancing** the system’s service as new requirements are discovered

## Change is Inevitable

- A client’s manager might move on, get a new job
- Their business strategy might move and needs might change
- New technology updates might get released
- Prototypes help anticipate changing needs
- Incremental delivery helps accommodate changing needs

Good big software may get used for 10-30 years

- In that time, markets change, technology improves
- Company has to update its software to stay ahead

## 85-90% of organisation software costs are evolution costs

Companies are more likely to update their current software

- Than fully transfer to a whole new platform
- If you do, you have to move data,use unfamiliar software, train people.
- Building a whole new piece of software is a risk
  - Errors might be made in specifications for new software
  - New software might take longer to arrive
  - A whole new project might be more expensive

### Why so costly

- Team changes
- Staff skills
  - maintenance is often given to junior staff
- Program Age and structure
  - Older software has changed / been redesigned more
- Poor development practices
  - Cutting corners, inadequate docs, etc.
  - Makes it hard to maintain.

### Poor development practices

- Undocumented requirements mean you can’t tell what’s changed
- Undocumented designs mean you can’t understand the plan
- Low readability of code makes it hard to figure out
  - Variable names that don’t mean anything
  - No comments
  - Complex methods – lots of ifs and for loops
- Poor code means limited documentation

## Types of change

### Fault repairs – to fix coding errors

- usually cheap to fix
- don’t involve much redesign

### Environmental adaptation - e.g. updates for new OS

- A bit more expensive to fix
- Doesn’t involve much redesign

### Functional Addition to meet business changes

- Much more expensive
- Often involves **redesign**

*“Once a system has been installed and is regularly used, new requirements inevitably emerge”*

### Changes are new requirements

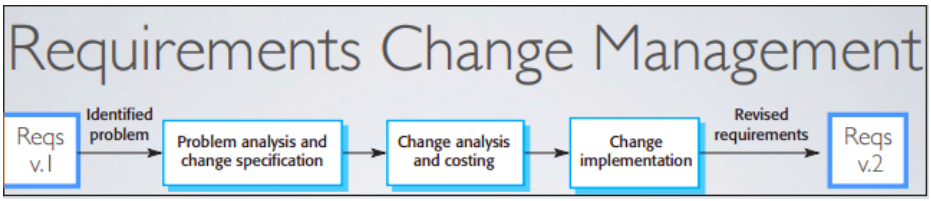
- Which affect specifications
- Which affect system designs
- Which affect code architecture
- Which affects code
- Which affects whether acceptance tests are passed

You have to start with **Requirements change management**

## Requirements Change Management

(A solution to handling change)

- The process of understanding and controlling change to requirements
- Tracing which requirements have changed in light of new ones
- Requirements need unique Ids, so they can be cross-referenced
- The impact on Specification/Designs can only be understood if you specified which requirements each specification supported in the fist place




### Problem Analysis & Change Specification

- That’s specification of Changes (not creating new specifications)
- Analysing a specific proposed change request
  - analyse how affects specifications
  - confirm understood changes/impact with requester

### Change analysis and costing

- Trace changes to specifications and estimate size of change
- Estimate a cost with the project & design manager
- Decide whether to proceed, given the cost

**Tips :**

- Don’t change arbitrarily, don’t start with the code
- Start by understanding the change and cost.
  - Is it big, small, easy, hard.
- Decide who’s paying, if outside of current contract

**Work to the new plan**

- It is better to make an intelligent change to the code
  - Perhaps, making larger number of changes
- Than to just make a fix/hack in the bit that’s wrong
- **Refractoring** code is the process of making improvements to the program to slow down degradation trough change
- Often considered ‘preventative maintenance’ that reduces the challenges for future changes.

**Things that can be improved by refractoring**

- Duplicate/similar code, which should be a method to call
- Long methods,which should be broken down
- Data clumping when patterns of function should be in a new class
- Speculative generality (code was written for future use,but was never used), just remove it

**Planning isn’t always realistic**

- Often problems occur that require *fast* or *emergency* changes
  - In this case updating the code takes priority over documentation
- The danger is that the requirements, the software design and the code become inconsistent

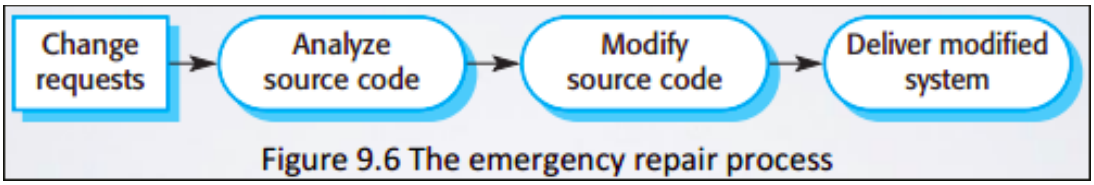


Figure 9.6 The emergency repair process

### *Handling emergency cases*

- It’s best to document emergency changes
- Later, ‘proper’ solutions to changes should be designed
  - For the next proper release version
- It’s the most dangerous when
  - Multiple subsequent emergency repairs occur
  - Repairs stack masking the fixes of the original repair
  - Code becomes more unmanageable / unmaintainable

## When is it better to start again

At some point, the cost of change gets high

- The client notes that they have spent a lot over time
- The software company realises its getting hard to fix

**Options :**

- Rebuild the software – new design, easier to maintain
- Build different software for new jobs
- Requirements are so different, you need a different software

Software company & client need to discuss long-term strategy

## Software evolution

- If you consider that the first release was based on
  - A prototype from a previous phase
  - An architecture design from another app you released
  - Reusing successful code segments
- Then evolving from V1 to V2 is just another iteration
  - Specification
  - Implementation
  - Testing
  - Release