Software Maintenance and Evolution

CS3 / SEOC1

Note 15

Maintenance and Evolution: From CS2 notes

- Types of maintenance:

  corrective: correcting faults in system behaviour. Caused by errors in coding, design or requirements

  adaptive: due to changes in operating environment (e.g. different hardware or OS)

  perfective: due to changes in requirements. Often triggered by organisational, business or user learning

- Also **preventive** maintenance; e.g. dealing with legacy systems
- Software re-engineering an approach to dealing with legacy systems through re-implementation.

Some Maintenance Statistics

- maintenance consumes 40% – 80% of total costs
- typical developer's activity (from *Lientz and Swanston*'s review of 487 companies):

  48% maintenance

  46.1% new development

  5.9% other

- huge quantities of legacy code:
  - US/DoD maintains more than 1.4 *billion* LOC for non-combat information systems, over more than 1700 data centres. Estimated to cost $9 billion per annum.
  - (in 1999) Boeing payroll system: approx 22 years old; 650K LOC COBOL
  - Bell Northern Research's *entire* operation is maintenance of one system – telephone switching product line. 12 million LOC (assembly and "higher-level" languages), approx. 1 million lOC revised annually

Distribution of Maintenance Effort: *Vliet* and *Lientz and Swanston*

- corrective (approx. 21%):

  12.4% emergency debugging

  9.3% routine debugging

- adaptive (approx. 25%):

  17.3% data environment adaption

  6.2% changes to hardware or OS

- perfective (approx. 50%):

  41.8% enhancements for users

  5.5% improve documentation

  3.4% other

- preventive (approx. 4%):

  4.0% improve code efficiency

Maintenance is hard because:

- key design concepts not captured
- systems not robust under change
- poor documentation
  - of code
  - of design process and rationale
  - of system's evolution...
- "stupid" code features may not be so stupid
  - work-arounds of artificial constraints, may no longer be documented (e.g. OS bugs, undocumented features, memory limits)
- poor (management) attitudes
  - maintenance not "sexy"
  - it's just "patching code"
  - easier/ less important than design (does not need similar level of support – tools, modelling, documentation, management)
- SEOC addresses all these issues. *How?...*

Managing Maintenance

Corrective: requires maintenance *strategy*, preferably negotiated contract between supplier and customer(s)

- policies for reporting and fixing of errors; auditing of process

Perfective: should be treated as *development* (i.e. requirements, specification, design, testing, ...

- iterative (or evolutionary) development approach best suited
- risks: drift, shift, creep, ooze, bloat, ...
- when does design or development *stop*?

Adaptive and Preventive: can anticipate, schedule, estimate, monitor and manage...

Maintenance Management Case Study (1)

- Spring Mills Inc.: early 1970's
  - programming shop runs 24 hours a day, 6 days a week
  - 3000+ programs in production
  - approx. 700 new programs per year
- 1972, John Mooney assessed operation as:
  - overworked programmers operating under stress
  - new systems typically over budget and late
  - no designated maintenance staff
  - approx. 75 maintenance requests per week
  - no maintenance strategy or planning
  - developers time: 30% maintenance; 45% new development; 10% special; 14% admin

Maintenance Management Case Study (2)

- 1973, Mooney reorganises shop and creates maintenance team
  - management strategy: requests logged, classified, evaluated, prioritised and assigned
  - team responsibilities: fast; good programming standards; regression testing of modified programs
  - numerous incentives, including financial
  - team responsible for *all* existing programs
  - new programs "signed over" to team when error- and change-free for 90 days
    - sign-over activity becomes significant project landmark

Maintenance Management Case Study (3)

- Outcome:
  - maintenance team becomes "highly skilled, elite corps of multi-lingual experts"
  - deep understanding of company's systems
    - particularly troublesome dependencies
  - offers services as "system auditors" or "consultants" on difficult problems
  - de facto *quality assurance* stakeholders
- leads to overall developers time:
  20% maintenance; 57.9% new development; 21.3% special and admin
- *previously, developers time:*
  30% maintenance; 45% new development; 24% special and admin
- everybody happy...

Preventive Maintenance

- accounts for 4% of maintenance requests, but
  - Pareto Principal applies
  - legacy systems *increasing* problem
- Software Migration approaches:

  Redevelopment: rebuild system from scratch. Easier problem (initially) but costly and very high risk

  Transformation: to (typically) new language/ paradigm:

  restructure c.f. *refactoring*

  re-engineer typically reverse-engineering followed by forward-engineering

  design recapture recreate design abstractions from code, documentation, personal experience, general problem and domain knowledge

  Encapsulation: "Software Wrapping" – wrap up existing code as components

Software Wrapping Case Study (1): Sparkasse: German savings and loan organisation

- 7 regional computing centres; client-server batch processing on conventional mainframe systems; code (variously) in Assembler, PL/1, COBOL and NATURAL
- legacy host systems highly integrated
- desired to introduce OO and components
- wrapping approach taken:
  - reuse S/W by encapsulating and controlling access via API's (Application Program Interfaces)
  - reuses existing S/W without moving it to new environment
  - legacy S/W remains, with minor changes, in native environment – yet is accessible to newer distributed OO components

Software Wrapping Case Study (2)

- 1997: Wrapping pilot-project undertaken
- 5 encapsulation levels:

  Job: remotely invoked batch-type job control procedures

  Transaction: client-server transactions

  Program: remotely invoked batch program

  Module: native code modules (easiest to wrap – already "component-ish")

  Procedure: individual procedure within legacy code (hardest to wrap)

Sparkasse: Issues/ lessons learned

- adaption of *all* subprograms necessary
- server to host communication weakest link
  - character conversion, ASCII to EBCDIC, common
  - constant translation and re-translation
- testing time-consuming due to high number of dependencies
- 5 step, bottom-up testing strategy:
  1. test adapted program in controlled test-harness
  2. test wrapper software with driver for client and stub for wrapped code
  3. test wrapper and wrapped code
  4. integration test: complete client-server transaction
  5. system test: multiple transactions to test reentrancy of wrapper and wrapped code

Summary

- Maintenance
  - is important
  - is difficult and costly
  - can, and should, be managed
  - has a bad reputation, but can and should be challenging and rewarding
- legacy systems a significant and increasing problem
  - number of approaches to dealing with this
  - many involve transforming to OO and/ or component based paradigm
    * abstraction/ high cohesion; encapsulation/ low coupling
- SEOC helps. How?
  - *You really should be able to work this out for yourself by now...*