Lecture 18 - Reusability

Collin Roberts

November 15, 2023

Outline

1. Re-Use Concepts
2. Impediments to Re-Use
3. Types of Re-Use
    1. Accidental (Opportunistic)
    2. Deliberate (Systematic)
4. Objects and Re-Use
5. Re-Use During Design and Implementation
    1. Library (toolkit)
    2. Application Framework
    3. Software Architecture
    4. Component-Based Software Engineering

Reusability

Importance of Re-usability

1. Advantages of Re-Use
    1. Save time/resources during development/testing. "Don't re-invent the wheel".
    2. Maintenance becomes cheaper.
    3. Library subroutines are tested, (supposedly) well–documented

Reusability

1. Pitfalls of Re-Use
    1. Depending too heavily on re-use can make us averse to writing new code, even where this is needed.
    2. Suppose that we need to extend/enhance an existing module before we can re-use it. This risks introducing regression faults for existing consumers of the module.
    3. Old modules might not be as "good" (efficient, secure, having good style, etc.) as new modules.
    4. If we view the re-used module as black-box, then we may struggle to confirm that our S/W product will actually match the spec; if a failure occurs in the re-used module after deployment, then we may be slow to diagnose the cause.

Reusability

1. Pitfalls of Re-Use
    5. Compatibility Issues:
        1. S/W versions, or
    6. Writing a module to handle multiple situations can make the module less efficient than if a separate module was written for each individual situation - but this would not be effective **abstraction**.
    7. If performance of the re-used module is not optimized, then all re-users will suffer a performance hit.
    8. Undetected faults get propagated.
    9. Documentation is often poor in practice.

Reusability

2. Other Aspects
    1. On average, 15% of any S/W product is written to serve a unique purpose.
    2. In theory, remaining 85% could be standardized and reused.
    3. In practice, only 40% reuse is achieved.
3. Re-use refers not only to code, but also to
    1. documents (e.g. design, manuals, SPMP, etc.)
    2. duration/cost estimates
    3. test data
    4. architecture
    5. etc.

Impediments to Re-Use

1. Sometimes, what is a candidate for being re-used is not obvious.
    1. Poor documentation (external, or internal, e.g. lack of comments in code) can contribute to this problem.
    2. If we abstract effectively during analysis/design workflows, then what to re-use becomes clearer.
2. SQA test cases: too outdated to use (if business rules change)
3. **Ego:** unwillingness to use someone else's code ("Not Written Here" syndrome)
4. **Quality Concerns:** sometimes justified, as above.

Impediments to Re-Use

5. Re-use can be expensive. It is costly to:
   1. develop reusable modules, and
   2. search the libraries and re-use the right module.
6. Legal issues with contract developers (possible intellectual property problems)
7. Commercial Of The Shelf (COTS): Developers do not provide the source code, so there is limited to no ability to modify and to re-use.
8. Etc.

Accidental (Opportunistic)

Idea: Developer of a new S/W product realizes that a previously developed module can be re-used as a subroutine in the new S/W product (e.g. re-use previously developed `Mean` function).

Deliberate (Systematic)

Idea: S/W modules are specially designed and constructed to be used in multiple S/W products.

Objects and Re-Use

Key Fact: OO classes are the best type of module that we know about so far for fostering re-use.

Re-Use During Design and Implementation

Remarks on Notation:

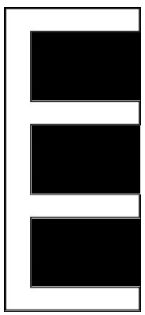
1. The diagrams for each type of re-use have
   1. shaded areas for the parts that are re-used, and
   2. whitespace for the parts that the re-user must supply.

We consider the following types of re-use.

Library (toolkit)

Assumes either the Classical or the OO paradigm.

Details:



Library (toolkit)

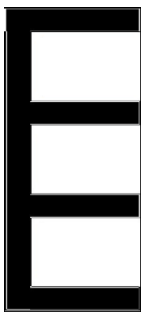
1. **What is Re-Used:** There is a **library**, a set of related re-usable operations e.g.
   1. A `Matrix` library contains many operations - +, *, `determinant`, `invert`, etc.
   2. `GUI` library contains different GUI classes - `window`, `menu`, `radio button`, etc.

   The re-user calls modules from the library.
2. **What is New:** The re-user must
   1. supply **control logic** of S/W product as a whole, and
   2. call library routines at the right moment using the control logic
   3. See Figure 8.2a in the text.

Application Framework

Assumes either the Classical or the OO paradigm.

Details:



Application Framework

1. **What is Re-Used:** Opposite to library approach: Control logic is re-used
2. **What is New:** The re-user must
   1. design application-specific sub-routines fitting inside the control logic.
   2. See Fig 8.2b in the text.

Application Framework

3. If the goal is to improve S/W development speed, then reusing a framework will be more effective than using libraries/toolkits WHY? It takes
    1. more effort to design control logic, and
    2. less effort to develop application-specific sub-routines, but
        1. in my experience, Library re-use is much more common than Application Framework re-use.
           Reason: It is rare to find two different S/W products with identical control logic.

Application Framework

4. **Examples of Application Framework Re-Use:**
    1. games
    2. Automated Teller Machines (ATMs)
        1. Suppose you are managing a team to develop S/W for ATMs, deployed by several banks.
        2. The control logic for an ATM deposit will be the same, regardless of the bank (note, we are over-simplifying a tiny bit here).
        3. However the details of how to carry out a deposit will depend completely on the choice of bank.
        4. A side comment here is that this would be an example of deliberate (systematic) re-use. We would design and build the control logic with the intent to re-use it at all of the banks.

Software Architecture

Remarks:

1. Software architecture encompasses a wide range of design issues, including
   1. organization of components (logical and physical)
   2. control structures
   3. communication / synchronization issues
   4. DB organization and access
   5. performance
   6. choice of design alternatives
2. Architecture can also be re-used.
3. A more detailed treatment of architecture will be beyond the scope of CS 430.

Component-Based Software Engineering

Goal: construct a standard library of re-usable components (i.e. for **Library** Re-Use). See §18.3 in the text if you want to read further.