CSE 435
Nov 11, 2015

Software Quality
architecture design using agile?

Recall the 4 premises of agile methods from the agile manifesto:

- value individual and interactions over processes and tools
- value working software over comprehensive documentation
- value customer collaboration over contract negotiation
- value response to change over following plans
How to choose an architecture under these conditions?

- generate an initial design, then as new requirements emerge, **refactor**

Problems with this approach?

- minimize documentation
  - **NO**
- refactoring of the design often doesn’t happen
As fast as possible...

BUT NO FASTER!
General Design Concerns
(and, what is a quality design?)

- Modularity
- Abstraction
- Cohesion
- Coupling
- Information Hiding
Why?

● Modifiability
● change has a likelihood of introducing bugs
● want to minimize that likelihood
● make (bug free) modifications easier
Modularity

- Organize modules according to resources/objects/data types
- Provide cleanly defined interfaces
  - operations, methods, procedures, ...
- Hide implementation details
- Simplify program understanding
- Simplify program maintenance
Abstraction

- Control abstraction
  - structured control statements
  - exception handling
  - concurrency constructs

- Procedural abstraction
  - procedures and functions

- Data abstraction
  - user defined types
Abstraction (cont.)

- Abstract data types
  - encapsulation of data

- Abstract objects
  - subtyping
  - generalization/inheritance
Why generalization/inheritance?

- easier to think about impact of changes from a practical, application standpoint
- the more general, the more likely you can accommodate change by changing the inputs rather than the code itself
- can localize changes and reuse pieces (maybe!)
However...

Don’t overdo it!

If too general, too abstract, too many layers of inheritance, you have added complexity, violated modularity and increased the probability of introducing bugs during maintenance.

Also makes a steep learning curve for others’ understanding.
Cohesion

- Contents of a module should be *cohesive*
  - Somehow related
  - group “similar” things together
- Improves maintainability
  - Easier to understand
  - Reduces complexity of design
  - Supports reuse
(Weak) Types of cohesiveness

- Coincidentally cohesive
  - contiguous lines of code not exceeding a maximum size

- Logically cohesive
  - all output routines

- Temporally cohesive
  - all initialization routines
(Better) Types of cohesiveness

- Procedurally cohesive
  - routines called in sequence

- Communicationally cohesive
  - work on same chunk of data

- Functionally cohesive
  - work on same data abstraction at a consistent level of abstraction
Coupling

● **Connections** between modules

● **Bad coupling**
  - Global variables
  - Flag parameters
  - Direct manipulation of data structures by multiple classes
Coupling (cont.)

● **Good coupling**
  - Procedure calls
  - Short argument lists
  - Objects as parameters

● Good coupling improves maintainability
  - Easier to localize errors, modify implementations of an objects, ...
Information Hiding

● Hide decisions likely to change
  ▪ Data representations, algorithmic details, system dependencies

● Black box
  ▪ Input is known
  ▪ Output is predictable
  ▪ Mechanism is unknown

● Improves maintainability
Abstract data types

- Modules (Classes, packages)
  - Encapsulate data structures and their operations
  - Good cohesion
    - implement a single abstraction
  - Good coupling
    - pass abstract objects as parameters
  - Black boxes
    - hide data representations and algorithms
Attributes of Quality

- Modifiability
- Usability
- Security
- Performance
- Reliability / Robustness
How to Increase Performance?

- Buy better hardware!
- Identify opportunities for concurrency
  - make separate process threads for concurrent tasks
- Use efficient algorithms
Reliability and Robustness

**Reliability:** the software correctly performs its required functions under assumed conditions

**Robustness:** the software performs “well” in the presence of problems, unexpected or hostile conditions
Reliability and Robustness

Writing software to be both reliable and robust requires anticipating potential problems.

Handle that thing that “won’t ever happen”
Identify Trade-off Priorities

- Establish priorities for choosing between incompatible goals
- Implement minimal functionality initially and embellish as appropriate
- Isolate decision points for later evaluation
- Trade efficiency for simplicity, reliability, ...
Design Tradeoffs

- Section 5.7 in textbook. Illustration of several different architectures for the same problem, different trade-offs for each kind.