Contracts and Specifications

1. Recall: Testability
   1. Specifications
   2. Modularity
   3. Separation of concerns
   4. Separation interface from implementation
   5. Writing test first
   6. Good documentation
   7. Mocking frameworks
   8. Assertions
   9. Logging/Profiling
   10. Hooks
2. Recall: The Challenges of Complexity
   1. Software is structurally and dynamically complex
   2. Many “moving part”
   3. Lots of interconnections
   4. Surprise: behavior of the whole is quite different from that of any piece
3. Why Modularity
   1. As a way of managing complexity: allows decoupling of pieces of the system
      1. Separation of concerns in comprehension and reasoning
      2. Different types of concerns are separated out
      3. Can use elements without seeing details
      4. Sanity in modifications: can modify one element of a program without having to modify different areas all throughout the program
   2. Modularity allows divide and conquer strategies to work
   3. As a means to reuse
4. Example areas of benefit
   1. Code creation
   2. Modification (refactoring, evolution)
   3. Testing
   4. Debugging
   5. Review
   6. Staff specialization
5. Abstraction: key to modularity
   1. Abstraction is the process of putting aside certain details in order to treat many particular circumstances as the same
   2. Put another way, in abstraction we recognize that there is a large amount of commonality among different cases, and provide an interface that capture that commonality
   3. The differences in the details of the context are then dealt with in different ways
6. Two key means of Abstraction
   1. Abstraction by parameterization. We seek generality by allowing the same mechanism to be adapted to many different contexts by providing it with information on that context in the form of parameters
      1. Type parameterization
         1. C++: Type templates
         2. Java: Generics
         3. ArrayList<Pair>
   2. Abstraction by specification. We ignore the implementation details, and agree to treat as acceptable any implementation that adheres to the specification.
      1. Encapsulation: key to abstraction by specification
         1. Separation of interface from implementation facilitates modularity
         2. Details being hidden here are differences between implementations
            1. These implementations can be different, but can make use of the same interface
      2. Benefits
         1. Locality: separation of implementation: Ability to build one piece without worrying about details of how another will be implemented
            1. The person writing foo knows enough about bar that they can call it with confidence
         2. Modifiability: ability to change one piece of project without breaking other code
         3. Some reuse opportunities: abstract over mechanisms that differ in their details to only use one mechanism
            1. Same client code can use many implantations of the interface
      3. Key Motivator: Risk of Change
         1. Abstraction by specification helps lessen the work required when we need to modify the program
         2. By choosing our abstractions carefully, we can gracefully handle anticipated changes
            1. E.g. choose abstractions that will hide the details of things that we anticipated changes frequently
            2. When the changes occur, we only need to modify the implementations of those abstractions – not the code that uses them (calls them)
7. Key Problem
   1. How to define the interface
      1. Knowing the name or type signature of something we are using is necessary but grossly insufficient
         1. If could count only on the signature of something remaining the same, would be in tremendous trouble: could do something totally different
         2. We want some sort of way of knowing what this thing does
         3. We don’t want to have to look at the code
      2. We are seeking a form of contract
      3. We achieve this contact through the use of specifications, and preferably design by contract
8. Specifications: Modularity Benefits
   1. User of abstraction
      1. Specifies guarantees on which they can count
      2. By extension, delineates assumptions on which can’t count
      3. No need for
         1. Guesswork
         2. Access to codebase
         3. Reading the code
   2. Creator of abstraction
      1. Can write code against interface
      2. Clear understanding of what can be freely modified
      3. Knowledge on what features users could be counting
9. Specifications: indirection benefits
   1. Conceptual clarity
      1. Elimination of latent ambiguity
      2. Attention to exceptional/boundary conditions
   2. Easier integration
   3. Easier understanding of code
   4. Clear documentation of derived requirements
   5. In some cases, more aggressive optimization
   6. Easier quality assurance
   7. Despite costs, often very significant time saving in medium and long term – and potentially in short-term testing and debugging time
10. Specifications： QA Benefits
    1. Early opportunities for review of detailed design
    2. Direct derivation of many assertion check
       1. To check in calls to a mock
       2. To ensure on return values
    3. Easier test case creation before code is created
    4. Less risk of violation of Liskov Substitution Principle
11. Ease of Reasoning Affect
    1. Risk of Error
    2. Speed of
       1. Development
       2. Debugging
       3. Testing
    3. Performance: Lack of clarity about behaviour lower opportunities for optimization
       1. On part of users
       2. On part of providers of code
12. Contracts
    1. Contract defines agreement between implementation and usages
    2. Contracts are described by “specifications” implicit or explicit, formal or informal
    3. Several levels of contract (Mingins and Jezequel, as mentioned by Pugh)
       1. Type-based contract
       2. Semantic contract
       3. Performance contracts
       4. Quality of service contracts
13. Specifications: Balancing Tradeoff
    1. Detailed enough information that
       1. Can count on critical functionality
       2. Rules out implementations that don’t meet needs
    2. General enough that
       1. Have flexibility to change implementation
       2. Doesn’t unnecessarily rule out legal implementation
    3. Clear enough to understand
       1. To those from multiple backgrounds
       2. Discover misspecification
14. Specification Approaches
    1. Multiple ways of specifying
       1. Pre/Post/Invariants
       2. Pre/effects/invariants
       3. History constraints
    2. For safety in reasoning, need at least a minima set