Constructing Reliable C++ Classes

Jeff Kotula
Stratasys, USA

Present a six step method for creating reliable, trusted classes that are correct, robust, and usable. The six steps focus solely on the design and implementation of classes rather than an entire software lifecycle. Each step is elucidated through discussion of its goals, descriptions of concrete techniques and recommendations, and analysis of the decisions made during the step. An example of class design will be used to guide and illustrate: a character string class will be discussed, being something that most software engineers have experience using, and have alternate examples to compare against and evaluate.

LEVEL AND SCOPE

This is a practical session, directed toward working software engineers, on the process of constructing reliable C++ classes. It is not a Methodology, but a method, with roots in academia, but relevant to the everyday concerns of developers. Experience with software development and OO techniques and mechanisms is assumed. While the tutorial examples are geared toward C++, the basic ideas apply to most OO languages.

OUTLINE

1.0 Overview
1.1 Goals of the Method
   1.1.1 Reliability: Correct, robust, usable
   1.1.2 Human Efficiency: Creator, user
   1.1.3 Predictability: Reproducible process
1.2 Summary of six steps
1.3 Integrating with Lifecycle Processes

2.0 Step 1: Identify Classes
2.1 Sources
2.2 Evaluating Candidates
   2.2.1 Clear and unambiguous purpose
   2.2.2 Unique purpose
   2.2.3 Relevant
   2.2.4 Independent of implementation
   2.2.5 Non-trivial
2.3 Summary Sentence Rule

3.0 Step 2: Design (Concurrent with Step 3)
3.1 Design by Contract
3.2 Activities
   3.2.1 Creation functions
   3.2.2 State-querying functions
   3.2.3 Invariant
   3.2.4 Revisit constructors
   3.2.5 State-modifying functions
   3.2.6 For each function
      3.2.6.1 Preconditions
3.2.6.2 Postconditions
3.2.6.3 Virtual or not?

3.3 Verify
3.3.1 Two clients: users, inheritors
3.3.2 Minimal
3.3.3 Complete
3.3.4 Independent
3.3.5 Consistent

3.4 Required Functions
3.4.1 printDebug()
3.4.2 isValid()

4.0 Step 3: Document (Concurrent with Step 2)
4.1 Undocumented = Unspecified and Unknown
4.2 Myth: Code can be Self-Documenting

4.3 Class Documentation
4.3.1 Description
4.3.2 Use-cases
4.3.3 Deriving subclasses
4.3.4 Invariant
4.3.5 Warnings, caveats, limitations
4.3.6 External dependencies
4.3.7 Design tradeoffs/decisions

4.4 Member Function Documentation
4.4.1 Description
4.4.2 Pre-conditions for system/class/object
4.4.3 Post-conditions for system/class/object
4.4.4 For each argument and the return value
4.4.4.1 Description
4.4.4.2 Accepted input values (pre-conditions)
4.4.4.3 Output values, if relevant

(post-conditions)

5.0 Step 4: Review
5.1 Value of Peer Review
5.2 Ego Versus Ownership
5.3 Issues to Review
5.3.1 Understandability
5.3.2 Testability
5.3.3 Maintainability
5.3.4 Reliability
5.3.5 Reuse
5.4 Procedure Recommendations
5.4.1 Fit review to local processes
5.4.2 Small group
5.4.3 Tight focus
5.4.4 Face-to-face

6.0 Step 5: Implement (Concurrent with Step 6)
6.1 Mechanical Procedure
6.2 For Each Function
6.2.1 Assert pre-conditions and invariant
6.2.2 Code trivial cases
6.2.3 Code function body
6.2.4 Assert invariant at end
6.2.5 Validate documentation via walkthrough

6.3 Error Handling
   6.3.1 What could go wrong?
   6.3.2 Exceptions, return codes, or text output?

7.0 Step 6: Test  (Concurrent with Step 5)
7.1 Unit Testing Framework
7.2 Unit Testing
    7.2.1 Each constructor, at various scopes
    7.2.2 Destructor, at various scopes
    7.2.3 Each member function
       7.2.3.1 Each argument
       7.2.3.2 Each postcondition
7.3 Equivalence Categories
    7.3.1 Segment the range of values
    7.3.2 Test using each segment
    7.3.3 Use preconditions to limit categories
7.4 Regression Tests
    7.4.1 Use unit testing framework
    7.4.2 Baselining results
       7.4.2.1 Text
       7.4.2.2 Graphics
    7.4.3 Executing

Jeff Kotula is a Principal Software Engineer at Stratasys, Inc. in Minnesota. Stratasys is the leading manufacturer of office prototyping machines supporting industrial CAD/CAM. My thirteen years of experience includes work in graphics, numerical algorithms, user interface design and implementation, software architecture and re-architecting, development process improvement, and systems design.

He received his Ph.D. in Computer Science from the University of Minnesota, with an emphasis in software engineering and programming languages. His doctoral thesis was titled, "Patterns of Object-Oriented Software Component Documentation" and fused the then-burgeoning field of pattern languages with software engineering principles of component documentation. He has published papers in IEEE Software (1998 and pending) and Software Practice and Experience (1996).