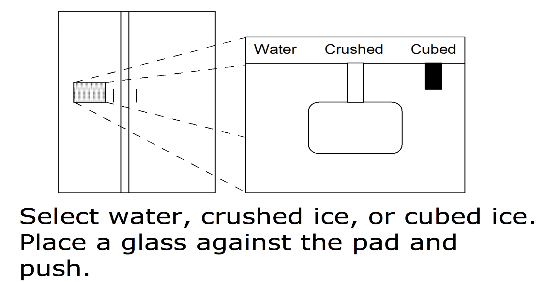
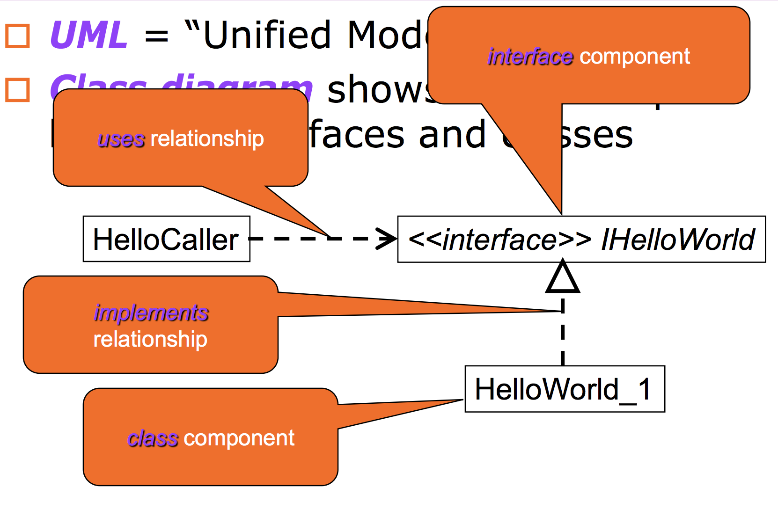
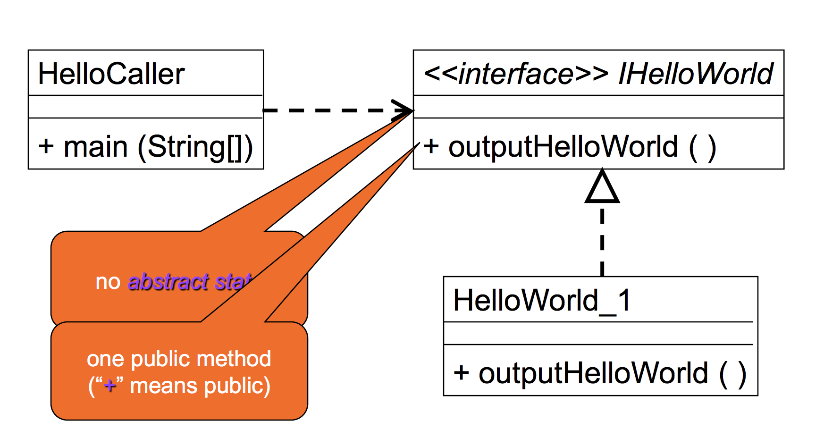
Exam 2, Topical Outline

Lecture 12: Intro to Design & Specification (10/9)

1. Vocabulary:
   * System:
     + Any part of anything you want to think about as an individual unit
   * Interface:
     + A description of the “boundary” between a system and everything else, that also explains how to think about that system as a unit
     + A contract between component users (clients) and developers (implementers)
     + Typically describes demands on users and responsibilities for implementers
     + Should present essentials in abstract user-oriented terms and hide inessential info
   * Subsystem/component:
     + a system that is used inside, i.e., as a part of, another system (a relative notion!)
   * Client:
     + An agent viewing a system from outside as an indivisible unit
   * Implementer
     + An agent viewing a system from the inside as an assembly of components
   * information hiding
     + Technique where info is intentionally left out that is merely an implementation detail
   * Abstraction
     + Complementary technique in which one creates a “cover story” to mask implementation details
2. UML class diagrams: modeling interfaces and classes, implements relationship, uses relationship
   * Class diagram shows relationships between interfaces and classes





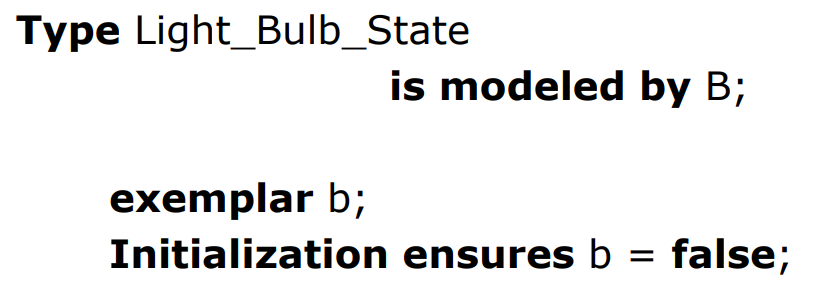
1. Rationales for separation of interface and implementation; examples and benefits (e.g., Data Dictionary interface with multiple implementations)
2. Modular Design – motivations
   * Seperation of concerns
   * Independent development by team members
   * Bug localization
   * Ease of evolution and maintenace
3. Design by Contract
   * Known as programming-to-the-interface
   * First clearly articulated in the 80s
   * Standard policy governing governing seperation of concerns rationale
   * Contract:
     + Facilitates communication between implementers and clients
     + Hide information by design, as its irrelevent to clients
   * Requires clause (precondition):
     + Characterizes responsibility of the caller
     + Not necessarily needed
       - Without it, implementations require error checking
   * Ensures clause (postcondition)
     + Effectively characterizes responsibility of the implementation code without
     + Implementer assumes requires clause
       - Guaranteed only if caller satisfies requires clause

Lecture 13: Design & Specification / Formal Methods (10/11)

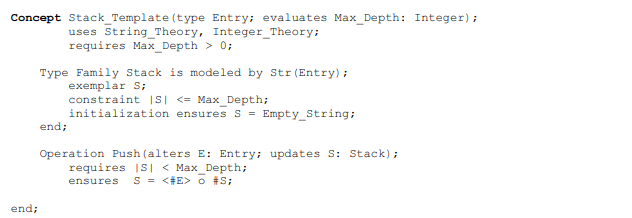
1. Requirements vs.specifications
   * Requirements:
     + Intended for customers and developers
     + Informal descriptions necessary
   * Specification:
     + Intended for development team
     + Formal descriptions necessary
     + Interface specification:
       - Contract between component users and developers
       - Describes demands on users and responsibilities for implementers
       - Presents essentials with abstraction, hiding inessentials
2. Informal specification
   * Straightforward descriptions
     + “Push pushed object on a stack”
     + Often unhelpful
   * Use of metaphors
     + A queue is like a line for fast food
     + Often generalizes
   * Use of implementation details
     + Push behaves like AddElement method on victor
     + Often unappropriated for user-oriented information abstraction
   * Seven Sins (from Meyer Paper):
     + Noise
       - Presence in text of element that doesn’t carry relevant information
     + Silence
       - Existence of feature not covered by text
     + Over-specification
       - Presence in text that relates to feature of problem not to feature of solution
     + Contradiction
       - Two or more text elements that define feature incompatibly
     + Ambiguity
       - Text that can be interpreted in at least two different ways
     + Forward Referencing
       - Uses features of problem not yet defined
     + Wishful Thinking
       - Text that defines feature of problem in way that a candidate solution cannot realistically be validated with respect to feature
3. Formal specification (see B. Meyer paper andslides)
   * Precisely communicates demands and responsibilities
   * Allows for parallel, independent development of client and implementation components within team environment
   * Minimizes integration costs
   * Specification Languages:
     + Often designed for particular programming language, or for general purpose
     + Some integrated with programming constructs
     + Few integrate ability to perform mathematical reasoning
   * Allows formal reasoning:
     + May be manual or automatic
4. Methods for checking correctness
   * Testing
   * Tracing / Inspection
   * Mathematical reasoning
     + Proves correctness and absence of bugs

Lecture 14: Mathematical Modeling (10/16)

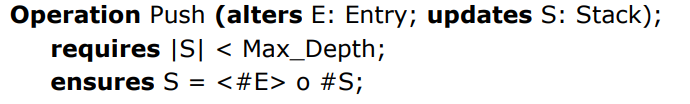
1. Programming types (i.e., integers, reals, etc.) viewed as mathematical types (i.e., Z, Boolean, Cartesian product of Booleans,strings)
   * Z: -5, 0, 15
   * B: true, false
   * Str(Z): < -5, 10, -15 >
   * Powerset(Z): { 10, 15, -5 }
2. Elements of mathematical model (type, modeled by, exemplar, constraints, initialization, operations with ensures / requires clauses, parameterstypes)
3. Meaning of various parameters types (updates, restores, clears,alters)
   * **Updates**: parameter is changed to a value specified by the ensures clause
   * **Alters**: parameter is changed to an unspecified value
   * **clears**: parameter reset to default type value on completion
   * **replaces**: initial value of parameter not used; replaced with a value specified by ensures clause
4. Simple models (e.g., light bulb, traffic light)

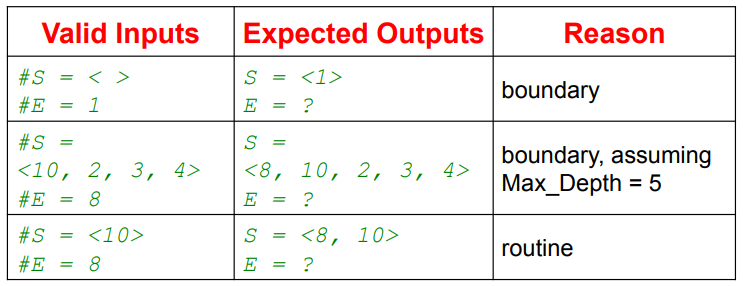


Lecture 15: Specification, continued (10/18)

1. Mathematical string theory/notation (Str(Z), empty\_string, concatenation, length,)
   * |S|: Length of S
   * S1 o S2: Concatenation of S1 and S2
   * <e>: String containing item e
   * Reverse(S): Reverse of S
   * Prt\_Btwn(start, end, S): Susbstring of S from start to end-1, inclusive
   * DeString(S): If |S| = 1, the first element
2. Specifying stacks, queues, and their operations using string theory
3. Operation specifications – correctness and style
   * See slides
4. Interface design: choice of operations (too many vs. too few)
   * Too few – Interface not sufficiently functional
   * Too many – Implementers unnecessarily burdened
5. Using reusable components (why, how)
6. Multiple implementations (why?)
   * Provide same functionality
   * Possible trade-offs:
     + Time vs space
     + Average case vs worst case
     + Efficiency vs Predictability
   * Pick ones best fitting their requirements

Lecture 16: Using Specifications for Testing and Verification (10/25)

1. Testing : goals, white-box, black-box
   * **Goal**: Find inputs where code fails to behave as specified
   * **Black**-**box**: Based on contracts, don’t change when code changes
   * **White**-**box**: Based on code
2. Designing a test plan: choosing test cases
   * **Boundary** cases: Smallest, largest, special values
   * **Challenging** cases: *You* find difficult or error-prone
   * **Routine**
3. Writing a test plan for a given operation (valid inputs, expected outputs, reason selected)



1. Tracing and debugging
   * Locate error using sample inputs
   * Analyze code n sample inputs to understand why code fails to behave as specified
   * **Tracing**: Analyze, but do not execute code (form of inspection)
   * **Debugging**: Execute code on selected inputs, following execution
2. Formal reasoning; how does it differ from testing, tracing,debugging
   * Prove using math that code functions as specified in contract on all precondition satisfying inputs
   * Approach:
     + Verifier
     + Analyze
     + Show logical errors when proof fails

Lecture 17: Iteration and Loop Invariants (10/30)

1. Justifications for correctness of code
   * Assertion developer claims to be true that will be checked by verifier
   * Easier for mechanical verifier than inventing it
   * Requires education and experience
2. loop invariants
   * Property that is true every time execution reaches a certain point
3. Writing loop invariants
   * tracing approach:
     + Trace for multiple iterations and note values of variables at beginning of each iteration
     + Find relationship that always holds true
   * logical approach:
     + Suppose Q is assertion to be proved after loop
     + Suppose B is loop condition
     + Find invariant Inv so that: not B and Inv => Q
   * Needs to confirm ensures clause
4. Progress metric for termination – an ordinal number that is decreasing
   * Programmer specifies progress metric for loop termination and verifier checks validity, using it in proof