Computer Science 340

Software Design & Testing

Design By Contract

© Ken Rodham 2006

Design By Contract

- “Design by Contract” is a design technique that helps produce more reliable software (i.e., fewer defects)

Abstract Data Types

- Object-oriented design is based on the *theory of abstract data types*
- Domain and implementation concepts are modeled in software as ADTs
- *Interfaces* capture the notion of an ADT
  - Examples: NetworkDevice, Stack
- Classes provide implementations of ADTs

Abstract Data Types

- Even if a class doesn’t implement a pre-defined interface, it still defines an ADT as embodied in its public interface
- In this case, the ADT definition is combined with its implementation
- Explicit interfaces are used if we expect to have multiple implementations of the ADT
  - interface List, class ArrayList, class LinkedList, …
- If we expect an ADT to have only one implementation, we often combine the ADT definition and implementation in the same class
  - Example: Matrix

Defining ADTs

- ADT = syntax + semantics
- Each ADT operation has: name, parameter list, return type
- ADT clients must conform to this syntax, or they will fail to compile
- Each ADT operation also has semantics: What is the meaning of the operation? What does it do?
- Classes that implement ADTs must faithfully adhere to operation semantics as well as syntax
  - What would happen if ArrayStack's push implementation fires a nuclear missile rather than pushing a value on the stack?

Defining ADTs

- Source code precisely defines ADT syntax
  - Compilers enforce ADT syntax
- Source code does not precisely define semantics
  - Compilers cannot enforce semantics
- ADT semantics are typically defined by comments in the code, if they're defined at all
  - Comment each operation explaining what it does
  - Example: Stack, Matrix

Defining ADTs

- Imprecise or incomplete definitions of ADT semantics lead to reliability problems:
  - Clients and implementers have different ideas of what an operation does
  - Differing assumptions lead to defects

Design By Contract

- DBC is a technique for more precisely defining ADT semantics, thus preventing misunderstandings
- DBC is based on the real-word notion of a legal contract
  - A contract involves a “client” and a “supplier”
  - Each side has obligations and expected benefits, which are precisely defined in the contract
  - If a party performs their obligations, they are guaranteed to received the promised benefits

Defining operation semantics: Pre-conditions & Post-conditions

- An ADT is a contract between client and supplier
- Each operation has *Pre-conditions* and *Post-conditions*
- Pre-conditions are the client’s obligations
- Post-conditions are the supplier’s obligations
- Examples: Stack, Matrix

Defining operation semantics: Pre-conditions & Post-conditions

- If a client invokes an operation having satisfied all pre-conditions, the supplier must ensure that all post-conditions are met upon return

- If the client did not satisfy all pre-conditions, the supplier is under no obligation to satisfy the post-conditions
  - The supplier can do whatever it wants, including all manner of anti-social behavior (including crashing)

Defining operation semantics: Pre-conditions & Post-conditions

- The pre-conditions and post-conditions define the semantics of the operation

Exceptions

- What if the caller satisfied the pre-conditions, but for some reason the supplier is unable to satisfy the post-conditions?
- The supplier throws an exception
- Exception => Supplier breeched the contract
- Why might a supplier fail to satisfy the post-conditions?
  - Bug in supplier
  - External factors beyond supplier’s control (hard disk crash, Internet down, etc.)
- Exceptions are <u>not</u> thrown if the client breeches the contract (i.e., fails to meet pre-conditions)
  - Actually, the supplier can do whatever it wants in this case

DBC vs. Defensive Programming

- Defensive Programming says:
  - Operation implementations should be bullet-proof
  - Check all parameters for validity before using them
  - Return error or throw exception if parameters are invalid, but never crash
- Puts heavy burden on the supplier
- Results in lots of parameter checking code
- Client and supplier often have redundant checks
- Results in more code (harder to maintain)
- Slows programs down (too much redundant checking)

DBC vs. Defensive Programming

- DBC says:
  - Ensuring that pre-conditions are met is the client's job
  - Operation implementations should not contain code to verify that pre-conditions were met (e.g., no parameter checking code)
  - If pre-conditions are not met and something unseemly occurs, it is the client's fault, and they got what they deserved
  - Suppliers must throw an exception if post-conditions cannot be met
- Puts more burden on clients
- Results in less and more efficient code
- As a debugging tool, operation implementations may include `assert` statements to verify that pre-conditions were met, but this is optional, and all such should be turned off in the final release of the software

DBC vs. Defensive Programming

- Designers must make a conscious choice between Defensive Programming and DBC

Class Invariants

- Pre and Post-conditions apply only to single operations
- Some supplier obligations apply to the ADT as a whole, and are not specific to single operations
- *Class invariants* are class-level conditions that must always be satisfied by the supplier
  - Examples: Stack, Matrix

Class Invariants

- Constructors must establish all class invariants
  - When a constructor completes, all class invariants must be satisfied
  - If a constructor cannot establish the class invariants, it should throw an exception
- In addition to their post-conditions, public operations must also ensure that all class invariants are satisfied upon return
  - I.e., the class invariants are ANDed with the post-conditions of every public operation
- Class invariants may be temporarily violated while a public operation is executing, but they must be reestablished before the operation returns

Documenting ADTs with `javadoc`

- Interfaces and classes
  - Header comment
  - @invariant   (custom tag)
- Operations
  - Header comment
  - @pre   (custom tag)
  - @post   (custom tag)
  - @param
  - @returns
  - @throws

Documenting ADTs with `javadoc`

- `javadoc` will generate HTML documentation for your interfaces and classes

- Running `javadoc`:
  - `javadoc -tag invariant:t:"Class Invariants:" -tag pre:cm:"Pre-Conditions:" -tag post:cm:"Post-Conditions:" MyClass.java`