CPSC 5011: Object-Oriented Concepts

Lecture 3: Programming by Contract

Academic Version of Design by Contract (proprietary)

Betrand Meyers, Eiffel

Defensive Programming

- Few assumptions made about data, state or environment

$\Rightarrow$ Little assumed to be correct (form, content, value,…)

$\Rightarrow$ Implicit assumption is that errors abound

- Extensive testing needed to ensure correct execution
- Significant overhead (and clutter code)
- Effectively contains error

Contractual Design

- Software viewed as a contract between the software designer and the application programmer
- Using published pre and postconditions, contractual design places responsibility for correct execution on the client
  - Client assumes that the software has been implemented correctly
  - Client must satisfy preconditions
  - Client tracks postconditions for potential state change(s)
- Efficiency regained by assuming that all preconditions met
  - No need to check data for form or value
- May not be appropriate for safety-critical software

Programming by Contract

- ***Alternative to Defensive Programming***
  - – avoid overhead of defensive programming
  - –retain safety of defensive programming
  - –responsibility for secure code shared between class designer and application programmer
  - – alleviates demand for extensive testing

- ***Documentation Convention => Communicates***
  - – *ASSUMPTIONS about environment & use*
    - •*Must be met by application programmer*
    - •*Increased need to indicate state change*
    - •*Decreased responsibility for error*
  - – *RESTRICTIONS*
    - •*Must be followed, as in contract, when using class*

Encapsulation => Reliable Code

Class Design dependent on Class Designer -- IDEALS

- No function can modify (class, object) data
  WITHOUT permission
- All constructors create objects in an initial valid state
- All mutator operations preserve validity of state
  - Insure legal state transitions

$\Rightarrow$ Data Integrity

$\Rightarrow$ Application Programmer Need not check for valid state

Proper Use Dependent on Application Programmer

Programming by Contract

- Bertrand Meyer (architect of Eiffel)
- Operations viewed as agents FULFILLING a contract
  - Defensive programming possible but NOT default
- Formal agreement between

  class designer and application programmer

If Application Programmer meets preconditions
Class Designer guarantees postconditions

Tradeoffs EXPLICIT: efficiency versus security

Efficiency vs. Safety

Example: STACK POP() operation

Application Programmer pops off EMPTY stack: ILLEGAL

STACK class response choices:

1) Undefined Behavior – no internal check

precondition for pop(): STACK object may NOT be empty

pop() does not check for empty state – no overhead

Application Programmer deals with consequences of violation of precondition

data corruption

delayed failure (hard to trace)

2) Return default value (zero) – internal check

no precondition – defensive programming

check for empty layered on top of all pop() calls

All users penalized by overhead

3) EXCEPTION -- a viable option? How is state known without check?

Programming by Contract

- Preconditions
  - state required expectations for correct function call
- Postconditions
  - record potential and actual state changes after function execution.
- Interface invariants
  - document stable conditions for use of the class.
- Implementation invariants
  - document internal design decisions
- Class invariants
  - describe key characteristics of the type defined by the class.

Programming by Contract

- PreConditions
  - Must be met before call made
  - Can be "none"
- PostConditions
  - Guarantee of state after call processed
  - Used to record potential and actual state changes

- Interface Invariants
  - Only PUBLIC interface
  - Informs application programmer of constraints
- Implementation Invariants
  - Internal, for software maintenance
  - identifies design constraints
- Class Invariants

Table 5.2 Programming By Contract

Specification	*Intent*	*Characteristics*
Precondition	Safe entry into function	Required incoming state
Postcondition	Identify state changes	Possible altered state
Interface Invariant	Promote consistent use	Services supported
Implementation Invariant	Software maintenance	Design specifications
Class Invariant	Communicate Type & Use	Designed functionality

Preconditions

- Removes need for class operation to verify precondition
- Published so understood by those requesting service
  - Potentially SEVERE consequences (e.g. pop() off empty stack)
- Describe required state necessary for correct behavior
- If not met, no guarantee about resulting behavior
- Must be verifiable
  - Application programmer must be able to verify precondition
  - e.g. can check if stack empty, AP can choose or avoid overhead
- Define compatibility between object state & operation
  - Not always easy to verify long-term
  - e.g. file open before reading (file existence is one time check)
- Define validity of argument
  - Acceptable values (not type: compiler checks type)

Postconditions

- Identifies potential and actual state changes
- Published so application programmer can
  - track state changes
  - verify subsequent preconditions
- Describe state object is left in after function exited
  - NOT a description of operation

EXAMPLE:

push() stack non-empty

pop() stack *may* be empty

Table 5.3 Common Pre and PostConditions

Pre condition	State satisfied Icon active Stack non-empty	Resource held File handle valid Memory allocated	Data valid Values in range	Ownership Callee is owner
Post condition	State altered Icon inactive Stack empty Stack non-empty	Resource released File closed Memory deallocated	Data stored Icon inactive Stack full Object updated	Ownership transferred Caller no longer owner

Interface Invariants

- Published at top of class: higher level than preconditions
  - Interface is public face of class
  - Provision/guarantee of behavior
- Describe restrictions on use of objects
  - Assignment operator private => copying suppressed
  - Copy constructor private => call by value not supported
- Describe preconditions that apply to all public functions
  - *e.g. threshold affects magnitude of functionality*
  - True upon entry to all operations
    - *If object has been manipulated correctly (preconditions met)*
  - Reduce need for internal testing
- Define relationship between two (or more) mutators
- Restrict state and state transitions

Implementation Invariants

- Design details for class designer(s) and software maintenance
- Published at top of implementation file

⇒ Class integrity maintained when subsequent designers
  implement changes/upgrades
  add new operations

- Describe design choices
  - Hash table collisions handled via chaining
  - Priority queue items aged so as to minimize starvation
  - Interface of subobject echoed (i.e. wrapper)

- Describe bookkeeping details – ownership, etc.
- Define legal values of data fields
- Define relationships between fields
  - e.g. inventory value drives commission percentage

Class Invariants

- Less common than other invariants
  - Intersection of interface and implementation invariants

- Describe conditions that hold true after
  - every constructor
  - before and after every operation

e.g. *set contains no duplicate values*
object id is unique
ownership relative to subobjects
owned, shared, transferable???

- All operations designed to preserve invariant
- Closed nature of class

=> designer complete control over all operations that modify data fields

Table 5.4 Common Invariants for Programming By Contract

Interface Invariant	*Implementation Invariant*	*Class Invariant*
Constraints	Internal design data structures utility functions	Relationships Association Cardinality
Expected use	Interface (portion echoed)	Environment
Data validity	Data dependencies	Ownership Transferable?
Error response	Error response	Error response

```
class PriorityQ              // specify invariants
{    ...                     // private data members
   // copying suppressed
   PriorityQ(PriorityQ&);
   PriorityQ& operator=(const PriorityQ&);

   void resize();
   void age();
 public:
   PriorityQ(unsigned SIZE = DEFAULT_CAPACITY);
   ~PriorityQ();
   …
   int   count() const;
   bool  isStored(const Item&) const;
   void  enQ(const Item&);
   void  deQ(const Item&);
   bool isEmpty() const;

   // possible supplemental public functions
   void  clear();
   Item& getFirst() const;
   Item& getLast() const;
};               // FIRST, must determine characteristics of the class
```

Interface Invariants
(Application Programmer)

- **Minimal**: illegal calls (unspecified behavior)
  - Call by value not supported
  - Copying via assignment not supported
  - Cannot extract (`deQ()`, `getFirst()`, `getLast()`) if PriorityQ empty
- **Problematic**: inconsistent with default behavior or internal response
  - Cannot add beyond capacity (`resize()`)
  - Starvation prevented (`age()` may or may not prevent all starvation)
  - `deQ()` highest priority item
    - External perception of priority may differ from internal
    - `age()` may interfere with presumed priority
- **Unnecessary**: condition enforced by compiler
  - Constructor cannot pass a negative number
  - Valid type (`Item&`) passed
  - Constructor must provide initial size (default defined by constructor)

Implementation Invariants
(Class Designer/Modifier)

- **Minimal:** implied by interface, internal data structures, private utility functions
  - Dynamically allocated array used to implement PriorityQ
    - Underlying heap data structure (with heapify() etc). Why?
      - Efficient access and resizing
      - Ordered collection, with efficient reordering
  - Copying via assignment not supported
  - Call by value not supported
  - No default behavior for accessors
  - Nop if `clear()` called on empty PriorityQ
  - `age()` strives to avoid starvation
    - Age factor associated with data value for internal priority
    - Outline aging algorithm: linear or proportional scaling?
  - `resize()` will double internal array when capacity reached

Implementation Invariants
(Class Designer/Modifier)

- **Problematic**: of questionable validity or relevance
  - Ordered Array (NO!: minheap or maxheap supports item extraction via root index)
  - No starvation (may be difficult to guarantee)

- **Unnecessary**: implied by function prototype
  - `isEmpty()` non-destructive (implied by `const`)
  - `enQ()` and `deQ()` trigger reordering (implied in heap functionality)

Preconditions

- **Minimal**: implied by above invariants
  - Extract (`deQ()`, `getFirst()`, `getLast()`) only from non-empty PriorityQ
- **Problematic**: of questionable validity
  - `isStored()` cannot be called with empty PriorityQ
- **Unnecessary**: implied by function prototype
  - `enQ()` has valid Item

PostConditions

- **Minimal**: describes STATE after operation done (or potential for state change)
  - PriorityQ may be empty after `deQ()`
  - PriorityQ empty after `clear()`
  - PriorityQ non-empty after `enQ()`
- **Problematic**: of questionable validity
  - DEFAULT_CAPACITY is public
- **Unnecessary**: describes what functions does
  - PriorityQ object exists after constructor fires
  - PriorityQ unchanged by `getFirst()`, `getLast()` (const functions)
  - One fewer item in PriorityQ after `deQ()`

Class Invariants

- **Minimal**: implied by above invariants
  - Container stores data
    - Highest priority item dequeued first
    - Priority is combination of age and value
    - Items aged (internally) to avoid starvation
  - Container capacity
    - Default capacity
    - Size may be specified upon instantiation
    - Internal resizing averts capacity overflow
  - Call by value not supported
  - Copying via assignment not supported
  - Extract (`deQ()`, `getFirst()`, `getLast()`) only from non-empty PriorityQ

Class Invariants

- **Problematic**: of questionable validity
  - No starvation
  - PriorityQ objects do not contain duplicate values
    - Perspective: aging changes compositive value
    - Application dependent
- **Unnecessary**: implied by functions provided
  - Starvation possible

OOD

- Dual Perspective
  - Client (application programmer) programs to interface
  - Class designer controls private implementation
- Encapsulation
  - Preserves internal control of state
- Programming by Contract
  - Capitalizes on encapsulation and dual perspective
  - Removes need for extensive testing with publication of pre & post conditions
- Documentation is Contract!
- Documentation is Specification of Design

Single Responsibility Principle

- *Every object should have a single encapsulated responsibility*
  - *Thus, there is only ever one reason to modify a class*
- Emphasizes cohesion and promotes software maintenance.
- Class functionality focuses on primary goal
  - class designer precisely targets use, and potential reuse.
  - Class integrity is easier to preserve.
- Priority queue example exemplifies this principle
  - Priority queue stores items in order of importance
  - Priority queue does not do anything else.
- When preservation of state (implementation invariant) is consistent with expectations of use (interface invariant), *the single responsibility principle holds*

Responsibility Driven Design

- *Identify responsibilities (functionality) and required information.*
- Works in tandem with Programming By Contract
  - specify design and all contractual expectations as to use
  - implementation invariant specifies the design of the object, with a focus on functionality and internal responsibility for state.
  - interface invariant specifies the public functionality and any client responsibility for consistent use.
- Clear and cohesive interfaces reinforce class
- For example, priority queue provides public functionality to store, retrieve and check for data.
  - Internally, priority queue implements functions to resize container when needed and to periodically age stored data items to prevent starvation.