Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Class Design II
Design by Contract

I. Salomie, C.Pop
2017

Method Contract

- Specification of method behavior (method service specification)
- Contracts of methods
  - usually informally specified or
  - left unspecified
  - major problems in software development
- Potential problems of informal specifications of contracts
  - incomplete on some behavior aspects
  - ambiguity and multiple interpretations
  - contradictions with other contracts
- Advantages of formally specified contracts
  - precision and unambiguousness
  - run time checking
    - catch contract violations by the implementations as well as
    - misused of the services by the clients
  - facilitates reasoning about the behavior of implementation and clients

Method Contract

- Main contract goals
  – Ensures constraints on services are met
  – Ensures maintaining of certain conditions on implementation
- Contract specification
  – method pre-conditions
  – method post-conditions
- Pre-condition
  – boolean expression
    - Must hold when the method is invoked
    - Otherwise, exception is thrown
- Post-condition
  – boolean expression
  – holds when the method invocation returns

Method Contract

- Specification languages - support formally specifying the complete behavior of software systems
  – needs mathematical skills
  – algebraic specification: Larch
  – model based specification: Z and VDM
  – high costs => recommended for highly critical applications
  – limited use can be easy and very cost-effective

Design by Contract

- Introduced (and also his trademark) by Bertrand Mayer (ETH Zurich)
- The programmer is required to specify for each method (beside the functional code)
  - Method pre-conditions
  - Method post-conditions
- Programming language support for Design by contract
  - Eiffel – implement pre and post conditions in the code
    - Checked anytime during runtime
  - Java uses assertions

Design by Contract
Example

```
public interface List {
  // mutators
  public void addElement(Object le, int i);
  public void addFirst(Object le);
  public void addLast(Object le);
  public Object remove(int i);
  public Object removeFirst();
  public Object removeLast();

  // getters (accessors)
  public Object getFirst();
  public Object getLast();
  public Object getElement(int i);
  public int getSize();

  // test
  public boolean isEmpty();
}
```

Contract on Methods
Specification of pre and post conditions

/**
 * @ pre *precondition as boolean expr*
 * @ post *postcondition as boolean expr*
public void **m1**() { … }

Note.
- @pre and @post tags can be used as custom tags in javadoc starting with Java SDK 1.4
- pre and post conditions may occur multiple times
  – Meaning: conjunction of all boolean conditions

- pre and post specifications
  – Java expression syntax plus the following (for increasing expresive-ness)
  – @result
    - variable holding the return value of the method
  – @nochange
    - boolean expression saying that the state of the object is not changed by the method
  – boolean operators
    - => (logical implication)
    - <=> (logical equivalence)

Design by Contract
Contracts on Accessors

/**
* returns the # of list items
* @pre true
* @post @nochange
*/
public int getSize();

/**
* returns true if and only if the list is empty
* @pre true
* @post @result <=> getSize() > 0
* @post @nochange
*/
public boolean isEmpty();

- @pre true
  – Precondition is always satisfied
  – The methods can be invoked any time
- @post @nochange
  – The method is not changing the state of the object

Design by Contract
Contracts on Accessors

```
/**
* returns the i-the list item
* @pre    i >= 0 && i < getSize()
* @post  @nochange
*/
public Object getElement(int i);

/**
* returns the first list item
* @pre     !isEmpty()
* @post   @result == getElement(0)
* @post   @nochange
*/
public Object getFirst();

/**
* returns the last list item
* @pre     !isEmpty()
* @post   @result ==
    getElement(getSize() - 1)
* @post   @nochange
*/
public Object getLast();
```


Design by Contract
Contracts on Mutators

- Object state
  - before mutator invocation
  - after mutator invocation
- Default on post expressions
  - object state after method execution

Design by Contract
Contracts on Mutators

Quantified expressions
- Universally quantified expression
  - predicate holds on every object in the collection
- Existentially quantified expression
  - predicate holds on at least one object in the collection
- Specification extensions
  - Universally quantified expression
    @forall x : Range @ Expression
  - Existentially quantified expression
    @exists x : Range @ Expression
  - Meaning
    - x is variable name,
    - Range expression, specifies the collection
    - Expression, boolean expression

Range expressions
- [m .. n]
  - int range
  - m, n integer expressions
- Expression
  - Must evaluate to Collection, Enumerator or Iterator objects
  - Defines a range in the collection – contains all objects in the collection, enumeration or iterator
- ClassName
  - Defines a range consisting of all the instances of a class

Design by Contract
Contracts on Mutators

```
/**
* Adds a new element to the list on position i
* @pre    el != null && i>=0 && i <= getSize()
* @post  getSize() == getSize()@pre + 1
* @post  @forall k : [0 .. getSize() -1 ] @
*       (k <i  ==> getElement(k)@pre == getElement(k))
  &&
*       (k == i ==> el@pre == getElement(k) &&
*       (k > i ==> getElement(k-1)@pre == element(k)
*/
```
public void addElement (Object el, int i);

Design by Contract

Contracts on Mutators

```
/**
* Adds a new element at the head of the list
* @pre el != null
* @post getSize() == getSize()@pre + 1
* @post el@pre == getElement(0)
* @post @forall k : [0 .. getSize() -1 ] @
*       (getElement(k-1)@pre == getElement(k))
* /
public void addFirst (Object el);

/**
* Adds a new element at the end of the list
* @pre el != null
* @post getSize() == getSize()@pre + 1
* @post el@pre == getElement(getSize() - 1)
* @post @forall k : [0 .. getSize() -2 ] @
*       (getElement(k)@pre == getElement(k))
* /
public void addLast (Object el);
```


Design by Contract

Contracts on Mutators

```
/**
* Remove the element at the i-th position
* @pre getSize() > 0
* @pre i >= 0 && i < getSize()
* @post @result = getElement(i)@pre
* @post getSize() == getSize()@pre - 1
* @post @forall k : [0 .. getSize() -1 ] @
*       (k <i  ==> getElement(k)@pre == getElement(k)) &&
*       (k >= i ==> getElement(k+1)@pre == element(k)
*/
public void remove (int i);
```

Design by Contract
Contracts on Mutators

```
/**
* Remove the element at the list head
* @pre getSize() > 0
* @post @result = getElement(0)@pre
* @post @forall k : [1 .. getSize() -1 ] @
*         getElement(k+1)@pre == element(k)
*/
public void removeFirst(int i);

/**
* Remove the element at the end
* @pre getSize() > 0
* @post @result = getElement(getSize() - 1)@pre
* @post @forall k : [0 .. getSize() -1 ] @
*         getElement(k)@pre == element(k)
*/
public void removeLast (int i);
```


Design by Contract
Contracts and impementations

- All classes that implement List should fulfill the contract stipulations
- Pre and post conditions of the interface methods =>
  - Pre and post conditions of the class methods that implement the interface

Design by Contract
Class Invariants

- Object state
  - Stable
    - Initialized, but not being manipulated
  - Transient
    - The object is being manipulated, i.e. a method is executing on the object
- Invariants of a class
  - Formally specified condition
    - Always holds on any object of the class whenever it is in a stable state
  - Well-formed state

Design by Contract
Class Invariants

- Thinking about invariants
- Example - DLlist an implementation of the List interface
  - If the list is empty =>
    - head should be null **and** tail should be null
  - If the list is not empty =>
    - head points to the first list element **and** tail points to the last list element
  - **howMany** value should be equal to # of nodes reachable by following the next link from the list head
  - For each node reachable from head
    - **prev** field of its succeeding node should point to itself
    - **next** field of its preceding node should point to itself
  - The **prev** of the first node and the **next** of the last node is null
- DLlist representation is well formed if all the above conditions hold whenever a linked list object is in a **stable state**
- The conditions are invariants of the DLlist used to implement the List interface

Design by Contract

Class Invariants – well formed method for DLlist

```
protected boolean isWellFormed {
  int n = 0;
  for(DLnode p = head; p != null; p = p.next) {
    n++;
    if(p.prev != null) {
      if(p.prev.next != p) return false;
    }
    else {
      if(head != p) return false;
    }
    if(p.next != null) {
      if(p.next.prev != p) return false;
    } else {
      if(tail != p) return false;
    }
  }
  return n == howMany();
}
```


Design by Contract

Class Invariants

Documenting the invariants

- New tag @invariant booleanExpression

```
/**
* @invariant   isWellFormed()
* /
```

- @invariant tag can occur many times
- AND of all boolean expressions
- Entities used in invariants
  - implication operator
  - equivalence operator
  - quantified expressions
- Invariants deal with the objects at a given moment
- Class programmer must ensure
  - all public methods preserve the invariants

Design by Contract
Class Invariants

```
/**
 * @invariant    isWellFormed()
 * /
public class DLlist implements List() {
  // … other methods …
  protected boolean isWellFormed() {
     // … invariant implementation
  }

   protected DLnode head, tail;
   protected int count;
   static protected class DLnode {
      Object item;
       DLnode next, previous;
   }
}
```

- Class implementer must ensure that all public methods preserve the invariants

Design by Contract
Main Guidelines for Invariants

- The invocation of any method, in any order should always result in a well-formed state
- This is achieved if the following implementation obligations hold (for a given class)
  - For each public class constructor
    - The invariants must be **post-condition** of the constructor or implied by its post-condition
  - For each public method of the class
    - invariants can be
      - **assumed** as method preconditions and
      - **must** be method post-condition or implied by post-condition

Design by Contract
Assertions

- Run-time checking
  - pre and post conditions of methods and
  - class invariants
- Assertion
  - boolean condition at a given program location
  - Condition should be true whenever the execution flow reaches the assertion location
  - Java assessment statements (beginning with JDK 1.4) - check assertions at run time
- Syntax:

  assert assertionCondition;
- assertCondition - boolean expression
  - if true, assert has no effects
  - if false, AssertionError exception is thrown

Design by Contract
Assertions

- Assertions rules
  - Assertions on method preconditions
    - Should be placed at the entry point of each method
  - Assertions on method post-conditions
    - Should be placed at every exit point of each method
  - Assertions on class invariants
    - Should be placed at the entry point and at the exit point of each public method

Design by Contract
Assertions - Examples

```
/**
* returns the first list item
* @pre     !isEmpty()
* @post   @result == getElement(0)
*/
public Object getHead() {
  assert !isEmpty();
  Object result = (head != null) ?
    head.element : null);
  assert result == getElement(0);
  return result;
}
```

```
/**
* Adds a new element to the list on position i
* @pre    el != null && i>=0 && i <= getSize()
* @post   getSize() == getSize()@pre + 1
*/
public void addElement (Object el, int i) {
  assert el != null && i >= 0 && i <= getSize();
  assert isWellFormed();
  int sizePre = getSize();
  if(i<= 0) { addFirst(el); }
  else if (i >= count) { addLast(el) }
  else {
    // i >0 && i < count
    DLnode n = head;
    for(int j = 0; n != null; && j < i-1; j++) { n = n.next; }
    DLnode node = new DLnode();
    node.element = el node.next = n.next; node.prev = n;
    node.next.prev = node; n.next = node;
    count ++
  }
  int sizePost = getSize();
  assert sizePost == sizePre + 1;
  assert isWellFormed();
}
```


Design by Contract
Assertions – Examples

- Comments
  - Assertions derived from
    - pre and post conditions
    - class invariants are translated into assertions
  - Not all post-conditions are asserted
  - Post-conditions dealing with objects pre-state are more complicated to be asserted
- Note
  - Compiling with assertions
  - JDK 1.4 or higher

Design by Contract
Defensive Programming

- Objective
  - Preventing misuse
- How
  - Using assertions derived from the preconditions of all methods
  - An assertion failure of a pre-condition
    - indicates that the client has attempted to improperly use the service
  - Indications given by assertions in case of an error
    - Help to determine whether the error is caused by service implementation or improper service use
- Help in unit testing
  - assertions derived from post-conditions and class invariants - useful in unit testing and implementation debugging

Design by Contract
Pre and post condition in overriding methods

- Relates to overriding and implementing interfaces
  - Overriding subclass methods and implemented interface methods should honor the contracts
- No stronger preconditions in the subclasses
  - In other words: the subclass methods cannot be more restrictive than the corresponding methods of the super-class
- No weaker post-conditions in the subclasses
  - In other words: the subclass methods cannot be do less than the corresponding methods of the super-class

Design by Contract
Main Guidelines

- Each method should include assertions
  - on pre and post conditions
  - on class invariants