CS1007: Object Oriented Design and Programming in Java

Lecture #7
Sept 27

Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

- Design considerations
- Testing
- Putting all together

Announcements

- Midterm date set
  - Will post review notes
  - Will be open book
  - No computers
- Homework due tonight at midnight
- Meet in clic lab on Thursday (Fairchild building) from 1pm-3pm (will take less)
  - Graded lab
  - Free to work outside of lab, but I will be there to answer questions

From last Time

- Encapsulation allows us to divide objects into logical parts and only present specific views of the object to outside manipulators
- Division of work
  - Accessors
  - Mutators

Idea:

- For real object oriented programming, should be minum of objects floating around between objects.
  - Using just methods to change objects
  - More responsibilities per object, but cleaner overall design
  - A.K.A Law of Demeter

Law of Demeter

- Example: Mail system in chapter 2

```
Mailbox currentMailbox =
  mailSystem.findMailbox(...);
```

- Breaks encapsulation
- Suppose future version of MailSystem uses a database
- Then it no longer has mailbox objects
- Common in larger systems
- Karl Lieberherr: Law of Demeter

- The law: A method should only use objects that are
  - instance fields of its class
  - parameters
  - objects that it constructs with new
- Shouldn't use an object that is returned from a method call
- Remedy in mail system: Delegate mailbox methods to mail system

```
mailSystem.getCurrentMessage(int mailboxNumber);
mailSystem.addMessage(int mailboxNumber, Message msg);
. . .
```

- Rule of thumb, not a mathematical law

Emphasis

- Some of the design choices come with experience
- No “One size fits all solution”
- Solution to balance decisions:
  - Documentation
  - Will talk about it soon

Designing projects

- Will now talk about what goes into designing a set of classes
- Remember
  - In general you will both give and be given only class files.
  - Along with the documentations (API) it is the only to know
    - What
    - How
    - Why

Quality of Class Interface

- Customers: Programmers using the class
- Criteria:
  - Cohesion
  - Completeness
  - Convenience
  - Clarity
  - Consistency
- Engineering activity: make tradeoffs

Cohesion

- Class describes a single abstraction
- Methods should be related to the single abstraction
- Bad example:

```
public class Mailbox
{
   public addMessage(Message aMessage) { ... }
   public Message getCurrentMessage() { ... }
   public Message removeCurrentMessage() { ... }
   public void processCommand(String command) { ...
  }
   ...
}
```

Completeness

- Support operations that are well-defined on abstraction
- Potentially bad example: Date

```
Date start = new Date();
// do some work
Date end = new Date();
```

- How many milliseconds have elapsed?
- No such operation in Date class
- Does it fall outside the responsibility?
- After all, we have before, after, getTime

Convenience

- A good interface makes all tasks possible . . . and common tasks simple
- Bad example: Reading from System.in before Java 5.0

```
BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
```

- Why doesn't System.in have a readLine method?
- After all, System.out has println.
- Scanner class fixes inconvenience

Be Clear

- Confused programmers write buggy code
- Bad example: Removing elements from LinkedList
- Reminder: Standard linked list class

```
LinkedList countries = new LinkedList();
countries.add("A");
countries.add("B");
countries.add("C");
```

- Iterate through list:

```
ListIterator iterator = countries.listIterator();
while (iterator.hasNext())
   System.out.println(iterator.next());
```

- Iterator between elements
- Like blinking caret in word processor
- add adds to the left of iterator (like word processor):
- Add X before B:

```
ListIterator iterator = countries.listIterator(); // |ABC
iterator.next(); // A|BC
iterator.add("France"); // AX|BC
```

- To remove first two elements, you can't just "backspace"
- remove does not remove element to the left of iterator
- From API documentation:
  `Removes from the list the last element that was returned by next or previous. This call can only be made once per call to next or previous. It can be made only if add has not been called after the last call to next or previous.`
- Huh?

Be Consistent

- Related features of a class should have matching
  - names
  - parameters
  - return values
  - behavior
- Bad example:

```
new GregorianCalendar(year, month - 1, day)
```

- Why is month 0-based?

Consistency

- Bad example: String class

```
s.equals(t) vs. s.equalsIgnoreCase(t)
```

- But

```
boolean regionMatches(int toffset,
   String other, int ooffset, int len)
boolean regionMatches(boolean ignoreCase, int
  toffset,
   String other, int ooffset, int len)
```

- Why not regionMatchesIgnoreCase?
- Very common problem in student code

Programming by Contract

- Spell out responsibilities
  – of caller
  – of implementer
- Increase reliability
- Increase efficiency

Preconditions

- Caller attempts to remove message from empty MessageQueue
- What should happen?
- MessageQueue can declare this as an error
- MessageQueue can tolerate call and return dummy value
- What is better?

- Excessive error checking is costly
- Returning dummy values can complicate testing
- Contract metaphor
  – Service provider must specify preconditions
  – If precondition is fulfilled, service provider must work correctly
  – Otherwise, service provider can do anything
- When precondition fails, service provider may
  – throw exception
  – return false answer
  – corrupt data

Preconditions

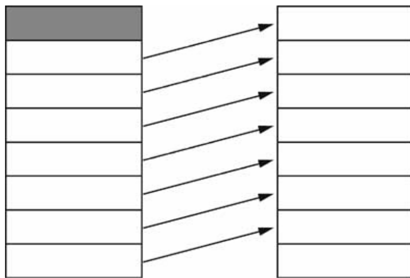
```
/**
   Remove message at head
   @return the message at the head
   @precondition size() > 0
*/
Message remove()
{
   return elements.remove(0);
}
```

- What happens if precondition not fulfilled?
- IndexOutOfBoundsException
- Other implementation may have different behavior

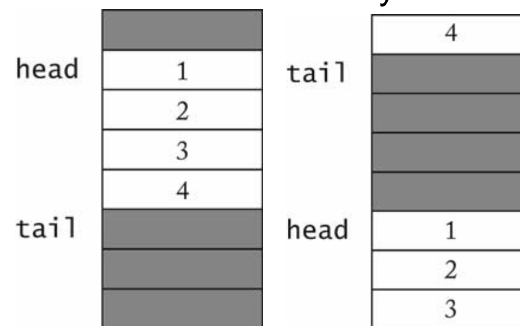
Circular Array Implementation

- Efficient implementation of bounded queue
- Avoids inefficient shifting of elements
- Circular: head, tail indexes wrap around

Problem with Array



Circular Array



Preconditions

- In circular array implementation, failure of remove precondition corrupts queue!
- Bounded queue needs precondition for add
- Naive approach:

```
@precondition size()< elements.length
```

- Precondition should be checkable by caller
- Better:

```
@precondition size() < getCapacity()
```

Java Assertion Command

- Mechanism for warning programmers
- Can be turned off after testing
- Useful for warning programmers about precondition failure
- Syntax:

```
assert condition;
assert condition : explanation;
```

- Throws AssertionError if condition false and checking enabled

Example

```
public Message remove()
{
    assert count > 0 : "violated precondition size()
   > 0";
    Message r = elements[head];
    . . .
}
```

- During testing, run with

```
java -enableassertions MyProg
```

- Or shorter, java -ea

Exceptions in contract

```
/**
   . . .
   @throws NoSuchElementException if queue is empty
*/
public Message remove()
{
   if (count == 0)
      throw new NoSuchElementException();
   Message r = elements[head];
   . . .
}
```

- Exception throw part of the contract
- Caller can rely on behavior
- Exception throw not result of precondition violation
- This method has no precondition

Postconditions

- Conditions that the service provider guarantees
- Every method promises description, @return
- Sometimes, can assert additional useful condition
- Example: add method

```
@postcondition size() > 0
```

- Postcondition of one call can imply precondition of another:

```
q.add(m1);
m2 = q.remove();
```

Class Invariants

- Condition that is
  - true after every constructor
  - preserved by every method (if it's true before the call, it's again true afterwards)
- Useful for checking validity of operations

- Example: Circular array queue

```
0 <= head && head < elements.length
```

- First check it's true for constructor
  - Sets head = 0
  - Need precondition size > 0!
- Check mutators. Start with add
  - Sets headnew = (headold + 1) % elements.length
  - We know headold > 0 (Why?)
  - % operator property:

```
0 <= headnew && headnew < elements.length
```

- What's the use? Array accesses are correct!

```
return elements[head];
```