The Smalltalk Environment, SUnit, and Inheritance
Names

• Capitalization conventions
  ▶ local variables start with a lower-case letter
  ▶ non-locals start with an upper-case letter
  ▶ new words are capitalized
    ◦ pariwise + product => pairwiseProduct
    ◦ with + all + subclasses => withAllSubclasses
Naming Guidelines

• Name methods after what they accomplish
  ▶ … not after the mechanism used in the implementation
  ▶ imagine a very different implementation.
    ◦ could you name this imagined method the same?

• Use the *same* name as the method in the other class that does a similar thing
Naming Guidelines

• Name variables after their roles
  ▶ instance variables and temporary variables should be named after their role
    sum     result     bounds
  ▶ don’t add a temporary variables unless there is a reason to do so!
    b := self bounds.
    children do: [ :each l ... b topLeft ... b bottomRight ...]
The Smalltalk Collections

Q: What is a Collection?

A: An object that understands (some of) the following methods:

- isEmpty
- size
- includes:
- occurrencesOf:
- do:
- select:
- collect:
- reject:
- detect:
- detect:ifNone:
- inject:into:
- asSet
- asBag
- asOrderedCollection
- asSortedCollection
**Collections (cont.)**

Q: Which classes have these methods?

A: Lots! In particular, most subclasses of Collection

<table>
<thead>
<tr>
<th>Set</th>
<th>Bag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Array</td>
<td>SortedCollection</td>
</tr>
<tr>
<td>Ordered Collection</td>
<td>LinkedList</td>
</tr>
<tr>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>
What’s the Difference?

Each of these classes has some interesting *refinement* of the basic protocol

- **Indexed Collections**
  - map an index to a value with *at:* (also *at: put:*)

- **Extensible Collections**
  - size can be changed with *add:* (and *remove:*)

- **Sequenceable Collections**
  - Indexed Collections on which we can sequence through the index set; supports *first:*, *do:*, *collect:* …
• Ordered Collections
  – access, insertion and removal based on the order are allowed: after: before: add: before: add:beforeIndex:

• Sorted Collections
  – The order is maintained by a relation (block) supplied explicitly with sortBlock:. at:put: is not understood.

If we regard these classes as a way of specifying *interfaces* (aka *protocols*) we can arrange them in a lattice by inclusion.
Interfaces of the Collections

Abstract class

Collection

Indexed Collection

Sequencable Collection

Updatable Collection

Set

Bag

Sorted Collection

Dictionary

Array

Interval

Ordered Collection
Abstract Classes in Smalltalk

Smalltalk classes are sometimes used to group behavior that is not complete enough to build an object! Such classes are called:

- abstract classes, or abstract superclasses

  - `collection>>add: newObject`
    “include `newObject` as one of my elements. Answer `newObject`...”
    self subclassResponsibility

  - `collection>>addAll: aCollection`
    aCollection do: [:each | self add: each].
    ↑ aCollection
Inheritance in Action!

• Subclasses of Collection don’t need to implement `addAll`:
  – it will be “inherited”
  – it will work if and only if they implement `add`:
• Partially abstract superclasses are a convenient place to put common code
• It can be hard to know if a class is abstract or concrete
  – **Hint**: try sending `new` or `new:` to the class
What about Parsing Numerals?

• Where should the methods go in the class hierarchy?

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>parseAsNumeral</td>
</tr>
<tr>
<td>digitValue</td>
</tr>
<tr>
<td>reversed</td>
</tr>
<tr>
<td>pairedWithPowersOf10</td>
</tr>
<tr>
<td>pairwiseProduct</td>
</tr>
<tr>
<td>sum</td>
</tr>
</tbody>
</table>
Unit Testing

• Code that isn’t tested doesn’t work
  ▶ Well, it’s true of my code — with the exception of simple accessors

• Two kinds of testing
  ▶ Unit testing
  ▶ Functional testing
What are test for?

• Tests are an executable specification of the functionality that they cover
  - always synchronized with the code

• Tests increase the likelihood that the code is correct.
  - When you introduce a bug, you are more likely to find it very quickly, while it is still easy to fix

• Writing “tests first” improves your interfaces
  - you see your code from the client’s point of view

• The presence of tests gives you the courage to make structural changes to the code: refactoring
  - refactoring is essential to prevent creeping entropy
Test-driven Development

• When creating fresh code:
  ▶ First write a test
    ○ only then write the code that makes the test run

• When maintaining old code
  ▶ First write a (failing) test to isolate the bug
    ○ then fix the bug
    ○ … and run the whole test suite
A problem due to Hamming

From Dijkstra's *A Discipline of Programming*.

Problem: construct an ordered set such that:

A1. 1 is in the set

A2. if x is in the Set, then x*2, x*3 and x*5 are in the set.

A3. there are no other values in the set.

Dijkstra limits himself to a proper prefix
Dijkstra’s Solution

Transcribed into Smalltalk, and parameterized by \( n \):

```smalltalk
dijkstrasSolution: n
    "This is a Smalltalk translation of Edsger Dijkstra's solution to Hamming's problem, as presented in chapter 17 of 'A Discipline of Programming'.
    P0 means that aq represents the first (aq size) elements of the sequence.
P1 means that (x2 is the minumum value > aq last s.t. x2 = 2 * x for some x in aq) and
    (x3 is the minumum value > aq last s.t. x3 = 3 * x for some x in aq) and
    (x5 is the minumum value > aq last s.t. x5 = 5 * x for some x in aq)

    The the loop termination test has been moved to the middle of the loop to make it possible to generate the sub-sequence of element <= n"

    | i2 i3 i5 x2 x3 x5 aq nextElement |
    n isInteger ifFalse: [self notify: 'dijkstrasSolution: argument ', n printString, ' must be an integer.'].
aq := OrderedCollection with: 1.
i2 := i3 := i5 := 1.
x2 := 2.
x3 := 3.
x5 := 5.
“{ P1 established }"
```
Dijkstra’s Solution

"{ P1 established }"

nextElement := (x2 <= x3 and: [x2 <= x5]) ifTrue: [x2]
ifFalse: [
    (x3 <= x2 and: [x3 <= x5]) ifTrue: [x3]
    ifFalse: [
        (x5 <= x2 and: [x5 <= x3]) ifTrue: [x5]].

nextElement > n ifTrue: [^aq asArray].
aq addLast: nextElement.

"{aq size has been increased by 1 under the invariance of P0}"

[x2 <= aq last]
    whileTrue: [i2 := i2 + 1. x2 := 2 * (aq at: i2)].

[x3 <= aq last]
    whileTrue: [i3 := i3 + 1. x3 := 3 * (aq at: i3)].

[x5 <= aq last]
    whileTrue: [i5 := i5 + 1. x5 := 5 * (aq at: i5)] repeat
How to solve this without being brilliant?

- *Generalize* from the given problem
- We start with a set, and add new elements:
  - based on the existing contents of the set
    - if $x$ is already in the set add $x \times 2$, $x \times 3$, …
    - if $x$ and $y$ are in the set, add $x \cup y$
  - subject to some condition ($x < 100$)