Wrap-Up

EECS3311 A: Software Design
Fall 2018

CHEN-WEI WANG

What You Learned

- **Design Principles**:
  - *Abstraction* [contracts, architecture, math models]
    Think ***above the code level***
  - Information Hiding
  - Single Choice Principle
  - Open-Closed Principle
  - Uniform Access Principle
- **Design Patterns**:
  - Singleton
  - Iterator
  - State
  - Composite
  - Visitor
  - Observer
  - Event-Driven Design
  - Undo/Redo, Command [lab 4]
  - Model-View-Controller [project]

Why Java Interfaces Unacceptable ADTs (1)

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as `E`.
- A reasonably ***intuitive overview*** of the ADT.

Java 8 List API

Why Java Interfaces Unacceptable ADTs (2)

Methods described in a ***natural language*** can be ***ambiguous***:

`E`	**`set`**`(int index, E element)` Replaces the element at the specified position in this list with the specified element (optional operation).

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

`index` - index of the element to replace

`element` - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

`UnsupportedOperationException` - if the set operation is not supported by this list

`ClassCastException` - if the class of the specified element prevents it from being added to this list

`NullPointerException` - if the specified element is null and this list does not permit null elements

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this list

`IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())

Why Eiffel Contract Views are ADTs (1)

```
class interface ARRAYED_CONTAINER
feature -- Commands
  assign_at (i: INTEGER; s: STRING)
     -- Change the value at position 'i' to 's'.
    require
      valid_index: 1 <= i and i <= count
    ensure
      size_unchanged:
        imp.count = (old imp.twin).count
      item_assigned:
        imp [i] ~ s
      others_unchanged:
        across
          1 |..| imp.count as j
        all
          j.item /= i implies imp [j.item] ~ (old imp.twin) [j.item]
        end
  count: INTEGER
invariant
  consistency: imp.count = count
end -- class ARRAYED_CONTAINER
```

Why Eiffel Contract Views are ADTs (2)

Even better, the direct correspondence from Eiffel operators to logic allow us to present a *precise behavioural* view.

ARRAYED_CONTAINER

feature -- Commands
  assign_at (i: **INTEGER**; s: **STRING**)
    -- Change the value at position 'i' to 's'.
   require
    valid_index: 1 ≤ i ≤ count
   ensure
    size_unchanged: imp.count = (**old** imp.twin).count
    item_assigned: imp[i] ~ s
    others_unchanged: ∀j : 1 ≤ j ≤ imp.count : j ≠ i ⇒imp[j] ~ (**old** imp.twin) [j]

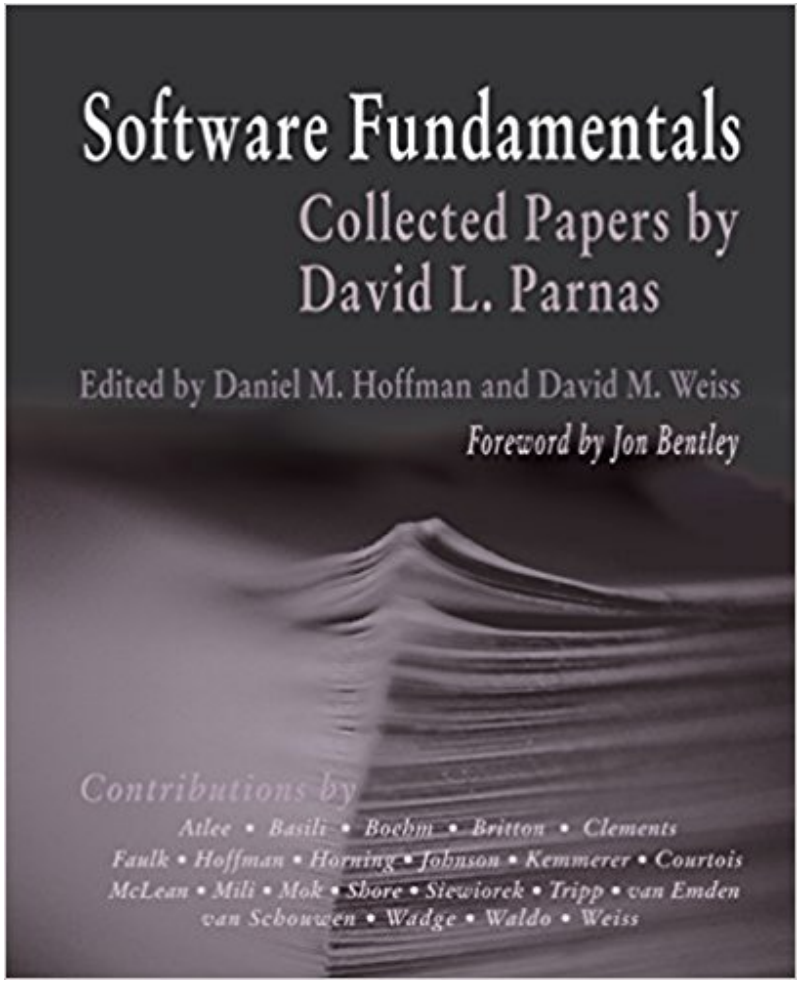
feature -- { **NONE** }
  -- Implementation of an arrayed-container
  imp: ARRAY[STRING]

invariant
consistency: imp.count = count

Beyond this course... (1)

- How do I program in a language not supporting *DbC* natively?
  - Document your *contracts* (e.g., JavaDoc)
  - But, it's critical to ensure (manually) that contracts are ***in sync*** with your latest implementations.
  - Incorporate contracts into your Unit and Regression *tests*
- How do I program in a language without a *math library*?
  - Again, before diving into coding, always start by *thinking above the code level*.
  - Plan ahead how you intend for your system to behaviour at runtime, in terms of interactions among *mathematical objects*.
  - Use *efficient* data structures to support the math operations.
    - `SEQ` refined to `ARRAY` or `LINKED_LIST`
    - `FUN` refined to `HASH_TABLE`
    - `REL` refined to a graph
  - Document your code with *contracts* specified in terms of the math models.
  - Test!

Beyond this course. . . (2)



- *Software fundamentals: collected papers by David L. Parnas*
- Design Techniques:
  - Tabular Expressions
  - Information Hiding