John Guttag
MIT Electrical Engineering
& Computer Science

Abstract Data Types
“At most points in a program one is concerned solely with the behavioral characteristics of a data object... what one can do with it, not with how the various operations on it are implemented. I will use the term ‘abstract data type’ to refer to a class of objects defined by a representation-independent specification.”

*The Specification and Application to Programming of Abstract Data Types*, John Guttag, 1975
Recalling 1975
ADT = objects + operations

- Implementation hidden
- No operations on objects of the type except those provided by the abstraction
- Specifications allow one to “forget” implementation
Ideas Were in the Air

- Parnas
- Liskov
- Zilles
- Hoare
- Dahl
- Wirth

My Key Collaborator
- Jim Horning

- Palme
- Balzer
- Wulf
- Morris
- Wegbreit
- Standish
- ...

Abstract Data Types
Other Ideas Were Also in the Air

- Successive refinement
  - Algorithms + Data Structures = Programs
- Information exposure
  - All code belongs to everyone on the project

Interpretations of these and other prevalent ideas ran counter to assumptions underlying abstract types
Assumptions Underlying ADT’s

• Software is not fractal
  – High level design is qualitatively different from low level design
  – Counter to some views of successive refinement

• Knowledge is dangerous
  – Not possible to assimilate much information
  – Gives local decisions global ramifications
    – Change challenging, viz Y2K problem
  – Counter to aspects of Harlan Mills’ project management
Assumptions Underlying ADT’s, cont.

- Performance not governed by local decisions
  - Performance really comes from
    - Global understanding
    - Ease of modification
      - Late optimization crucial
- Inventing application domain-specific types is a good thing. (“The programming process is facilitated by introduction of new domains of discourse.”)
  - Fixed set of type abstractions not sufficient
What Matters In Software Development
What Matters In Software Development

• Time, cost, and quality matter
  – Truth and beauty are means, not ends

• Software systems
  – Cost too much to build
  – Take too long to build
  – Often don’t do what people want them to do
Why?

• Because programming is hard!
• Large software systems are among the most complex systems engineered by man
• People expect software to be malleable
• Programming is about
  – Coping with defects
    – Customers, engineers, environment, software
  – Managing complexity and change
Reducing and Ordering Complexity

Divide and rule is the key

- We do this using
  - Decomposition
  - Abstraction

- Decomposition creates structure
- Abstraction suppresses detail
  - Trick is to suppress the right details
Data Abstraction Hits the Mark

- Domain-specific types provide convenient decomposition
  - Concepts that will be relevant over lifetime (often decades) of program
- Representation independence through abstraction
  - Interface to module characterized by operations on that module

```
representation \( \downarrow A \) code \( \rightarrow \) representation \( \downarrow A \)
Abst. value \( \downarrow \) spec. \( \rightarrow \) Abst. value
```
Where Data Abstraction Led

- Shift of emphasis from decomposition to abstraction
  - Programming methods
  - Programming languages
- Increased emphasis on interfaces characterized by operations
- Deeper understanding of types
  - Rep. invariants, abstraction functions
  - Data type induction
Where Data Abstraction Led, cont.

• Centrality of data objects rather than procedures
• Focus on programming as a process of combining relatively large chunks
• Deeper understanding of role of specification
  – As a communication mechanism
• Flatter program structures
  – Increased code re-use
• Proliferation of useful domain-specific libraries
Can Be Misused, of Course

• Deep hierarchies of types
  – Encouraged by inheritance

• Delayed discovery of representation difficulties
  – Encouraged by excessive top-down design

• Too many highly specialized types
  – Many problems are not unique
Concluding Remarks

- From 1945 – 1975 almost all improvement in programmer productivity attributable to better tools, i.e., higher level languages and better machines
- In 1975, I thought that most future progress would stem from innovations in ways we thought about programs
I Was Not Completely Wrong

- Correctly pessimistic
  - Not much progress on programming languages since late 1970’s

- Correctly optimistic
  - Lots of conceptual progress

- Progress on hardware a critical factor in improved productivity
  - Vast memories, fast processors
  - Facilitated use of abstraction
What Next?

• Need a better handle on embedded software
• New tools crucial to continued progress
  – Static and dynamic analysis
    – Of designs and code
    – Principled testing
• Look to the new pioneers
  – Michael Ernst, David Evans, Daniel Jackson, Rustan Leino, David Notkin, ...

• For old times sake, pop(push(s, e)) = e