Abstraction and modularity

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 1

Stepwise refinement

- Begin with a high level description of the task

```
begin
  print first thousand prime numbers
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Stepwise refinement

- Begin with a high level description of the task
- Refine the task into subtasks
- Further elaborate each subtask
- Subtasks can be coded by different people
- Program refinement — focus on code, not much change in data structures

```
begin
  print first thousand prime numbers
end
```

```
begin
  declare table p
  fill table p with first thousand primes
  print table p
end
```

```
begin
  integer array p[1:1000]
  for k from 1 through 1000
    make p[k] equal to the kth prime number
  for k from 1 through 1000
    print p[k]
```

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances

Data refinement

- Banking application
  - Typical functions: `CreateAccount()`, `Deposit()/Withdraw()`, `PrintStatement()`
- How do we represent each account?
  - Only need the current balance
  - Overall, an array of balances
- Refine `PrintStatement()` to include `PrintTransactions()`
  - Now we need to record transactions for each account
  - Data representation also changes
  - Cascading impact on other functions that operate on accounts

Modular software development

- Use refinement to divide the solution into components

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour

Modular software development

- Use refinement to divide the solution into components
- Build a prototype of each component to validate design
- Components are described in terms of
  - Interfaces — what is visible to other components, typically function calls
  - Specification — behaviour of the component, as visible through interface
- Improve each component independently, preserving interface and specification
- Simplest example of a component: a function
  - Interfaces — function header, arguments and return type
  - Specification — intended input-output behaviour
- Main challenge: suitable language to write specifications
  - Balance abstraction and detail, should not be another programming language!
  - Cannot algorithmically check that specification is met (halting problem!)

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements

Programming language support for abstraction

- Control abstraction
  - Functions and procedures
  - Encapsulate a block of code, reuse in different contexts
- Data abstraction
  - Abstract data types (ADTs)
  - Set of values along with operations permitted on them
  - Internal representation should not be accessible
  - Interaction restricted to public interface
    - For example, when a stack is implemented as a list, we should not be able to observe or modify internal elements
- Object-oriented programming
  - Organize ADTs in a hierarchy
  - Implicit reuse of implementations — subtyping, inheritance

Summary

- Solving a complex task requires breaking it down into manageable components
  - Top down: refine the task into subtasks
  - Bottom up: combine simple building blocks
- Modular description of components
  - Interface and specification
  - Build prototype implementation to validate design
  - Reimplement the components independently, preserving interface and specification
- PL support for abstraction
  - Control flow: functions and procedures
  - Data: Abstract data types, object-oriented programming