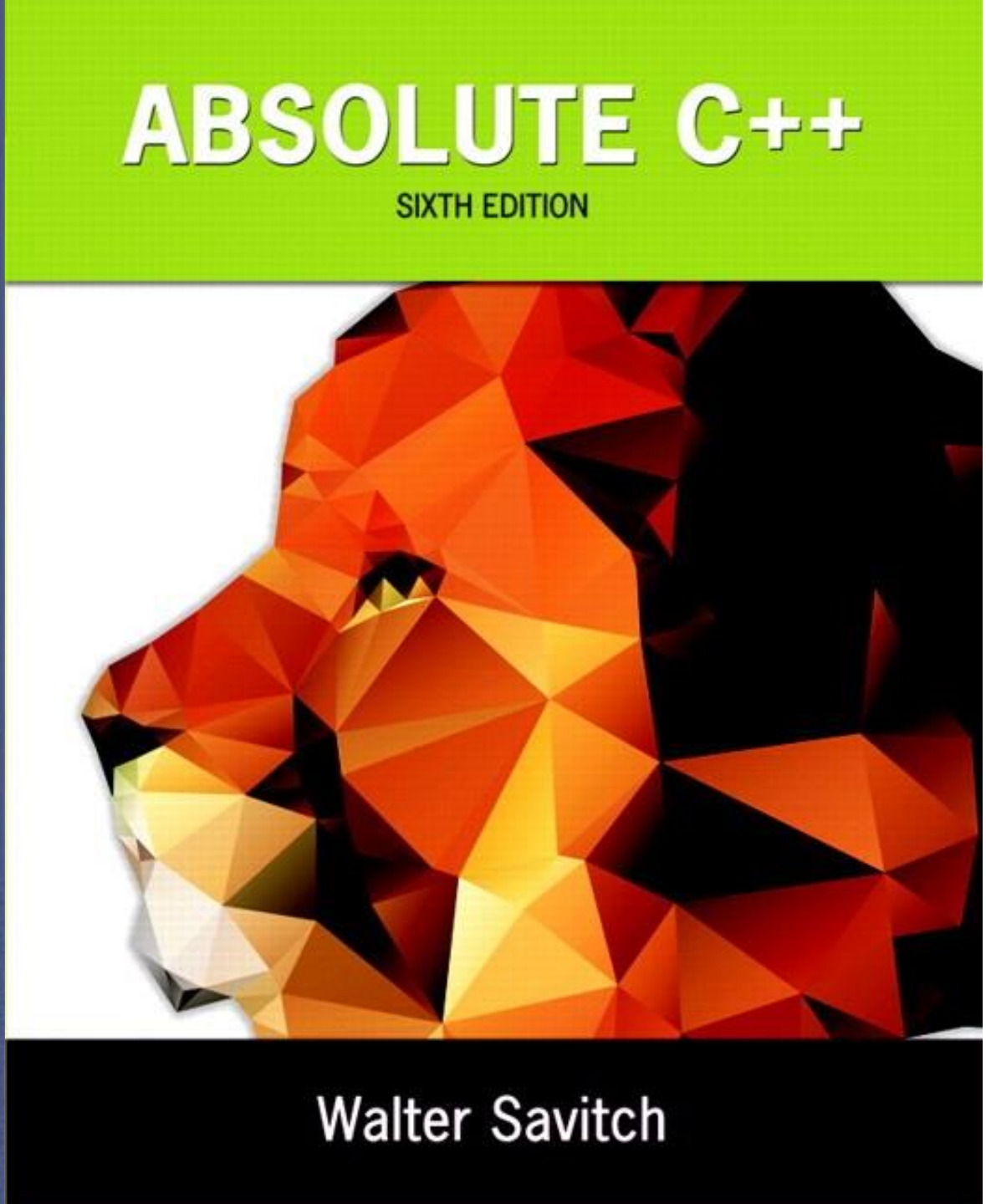


Chapter 20

Patterns and UML

PEARSON

Learning Objectives

- Patterns
  - Adapter pattern
  - Model-View-Controller pattern
  - Sorting pattern and it's efficiency
  - Pattern formalism
- UML
  - History of UML
  - UML class diagrams
  - Class interactions

Introduction

- Patterns and UML
  - Software design tools
  - Programming-language independent
    - Assuming object-oriented-capable
- Pattern
  - Like "ordinary" pattern in other contexts
    - An "outline" of software task
  - Can result in different code in different but similar tasks
- UML
  - Graphical language for OOP design

Patterns

- Patterns are design principles
  - Apply across variety of software *applications*
  - Must also apply across variety of *situations*
  - Must make assumptions about application domain
- Example:
  Iterator pattern applies to containers of almost any kind

Pattern Example: Iterators

- Recall iterators
- Iterator pattern applies to containers of almost any kind
- 1st described as "abstract"
  - As ways of cycling thru any data in any container
- Then gave specific applications
  - Such as list iterator, constant list iterator, reverse list iterator, etc.

Consider No Patterns

- Iterators
  - Imagine huge amount of detail if all container iterators presented separately!
  - If each had different names for begin(), end()
  - To make "sense" of it, learners might make pattern themselves!
- Until pattern developed, all were different
  - "Seemed" similar, but not organized
- Consider containers as well
  - Same issues!

Adapter Pattern

- Transforms one class into different class
  - With no changes to underlying class
  - Only "adding" to interface
- Recall stack and queue template classes
  - Both can choose underlying class used to store data:
    stack<vector<int>> -- int stack under vector
    stack<list<int>> -- int stack underlying list
  - All cases underlying class not changed
    - Only interface is added

Adapter Pattern Interface

- How to add interface?
  - Implementation detail
  - Not part of pattern
- But… two ways:
  - Example: for stack adapter:
    - Underlying container class could be member variable of stack class
    - Or stack class could be derived class of underlying container class

Model-View-Controller Pattern

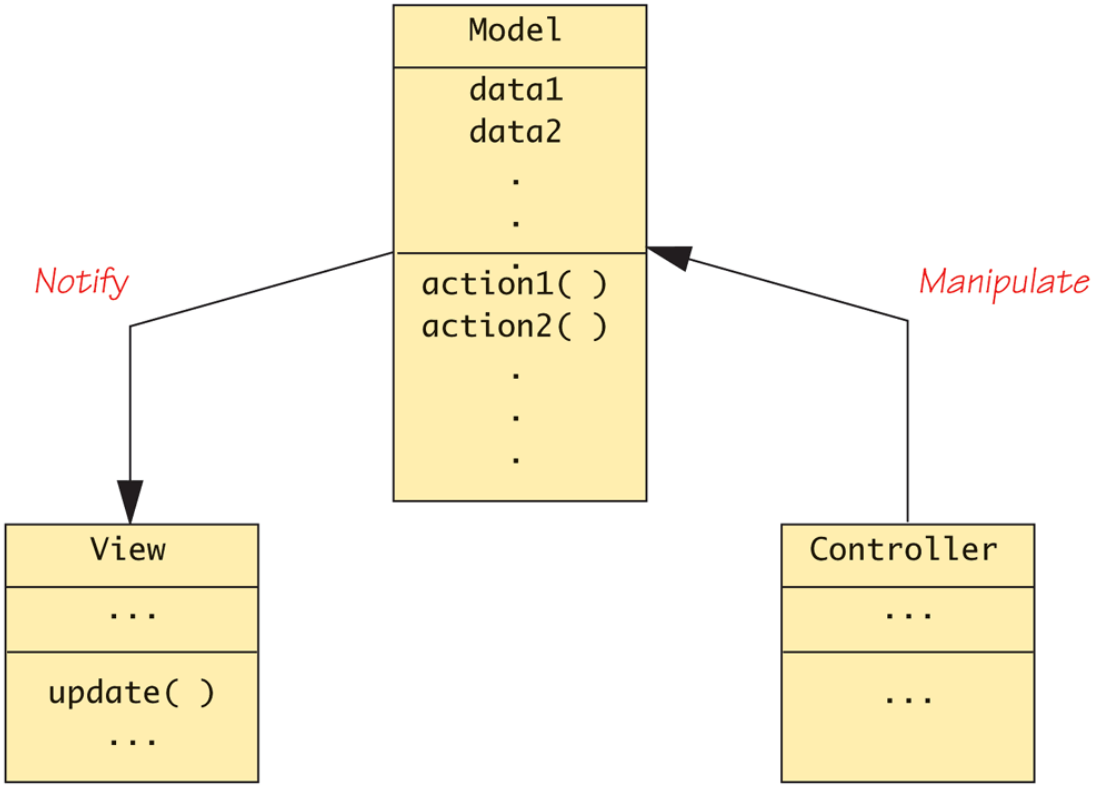
- Way of dividing I/O task out
  - Model part: heart of application
  - View part: output
    - Displays picture of model's state
  - Controller part: input
    - Relays commands from user to model
- A divide and conquer strategy
  - One big task → three smaller tasks
    - Each with well-defined responsibilities

Model-View-Controller Pattern

- Any application can fit
- But particularly suited to GUI design projects
  - Where view can actually be visualization of state of model

Display 20.1
Model-View-Controller Pattern

Display 20.1 Model-View-Controller Pattern



A Sorting Pattern Example

- Similar pattern among "most-efficient" sorting algorithms:
  - Recursive
  - Divide list into smaller lists
  - Then recursively sort smaller lists
  - Recombine two sorted lists obtaining one final sorted list

Sorting Pattern

- Clearly a divide-and-conquer strategy
- Heart of pattern:

```
int splitPt = split(a, begin, end);
sort(a, begin, splitPt);
sort(a, splitPt, end);
join(a, begin, splitPt, end);
```

- Note no details on how split and join are defined
  - Different definitions will yield different sorting algorithms

Function split

- Rearranges elements
  - In interval [begin, end]
- Divides interval at split point, *splitPt*
- Two new intervals then sorted
  - [begin, splitPt) – first half
  - [splitPt, end) – second half
- No details in pattern
  - Nothing about how rearrange and divide takes place

Function join

- Combines two sorted intervals
  - Produces final sorted version
- Again, no details
  - join function could perform many ways

Sample Realization of Sorting Pattern: Mergesort

- Simplest "realization" of sorting pattern is mergesort
- Definition of split very simple
  - Just divides array into two intervals
  - No rearranging of elements
- Definition of join complex!
  - Must sort subintervals
  - Then merge, copying to temporary array

Mergesort's join Function

- Sequence:
  - Compare smallest elements in each interval
  - Smaller of two → next position in temporary array
    - Repeated until through both intervals
  - Result is final sorted array

Sort Pattern Complexity

- Trade-off between split and join
  - Either can be simple at expense of other
  - e.g., In mergesort, split function simple at expense of complicated join function
  - Could vary in other algorithms
- Comes down to "who does work?"

Consider Quicksort

- Complexity switch
  - join function simple, split function complex
- Library files
  - Include files "mergesort.cpp", "quicksort.cpp" both give two different realizations of same sort pattern
  - Provide same input and output!

Quicksort Realization

- A sophisticated split function
  - Arbitrary value chosen, called "splitting value"
  - Array elements rearranged "around" splitting value
    - Those less than in front, greater than in back
    - Splitting value essentially "divides" array
  - Two "sides" then sorted recursively
- Finally combined with join
  - Which does nothing!

Sorting Pattern Efficiency

- Most efficient realizations "divide" list into two chunks
  - Such as half and half
  - Inefficient if divided into "few" and "rest"
- Mergesort: O(N log N)
- Quicksort:
  - Worst case: $O(N^2)$ (if split uneven)
  - Average case: O(N log N)
    - In practice, one of best sort algorithms

Pragmatics and Patterns

- Patterns are guides, not requirements
  - Not compelled to follow all fine details
  - Can take "liberties" and adjust for particular needs
    - Like efficiency issues
- Pattern formalism
  - Standard techniques exist for using patterns
  - Place of patterns in sofware design process not yet clear
    - Is clear that many basic patterns are useful

UML

- Unified Modeling Language
- Attempt to produce "human-oriented" ways of representing programs
  - Like pseudocode: think of problem, without details of language
- Pseudocode very standard, very used
  - But it’s a linear, algebraic representation
- Prefer "graphical" representation
  - Enter UML

UML Design

- Designed to reflect/be used with object-oriented programming philosophy
- A promising effort!
- Many companies have adopted UML formalism in software design process

History of UML

- Developed with OOP
- Different groups developed own graphical representations for OOP design
- 1996:
  - Booch, Jacobsen, Rumbaugh released early version of UML
  - Intended to "bring together" various other representations to produce standard for all object-oriented design

UML Lately

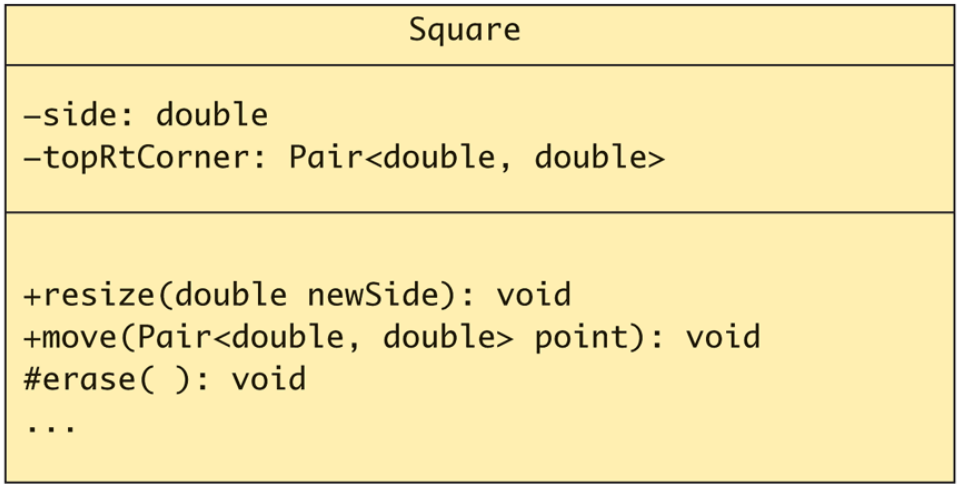
- Since 1996:
  - Developed and revised with feedback from OOP community
- Today:
  - UML standard maintained and certified by Object Management Group (OMG)
    - Non-profit organization that promotes use of object-oriented techniques

UML Class Diagrams

- As classes are central to OOP…
- Class diagram is simplest of UML graphical representations to use
  - Three-sectioned box contains:
    - Class name
    - Data specifications
    - Actions (class member functions)

Class Diagrams Example: **Display 20.6** A UML Class Diagram

Display 20.6 A UML Class Diagram



Class Diagrams Example Notes

- Data section:
  - + sign indicates public member
  - - sign indicates private member
  - # indicates protected member
  - Our example: both private (Typical in OOP)
- Actions:
  - Same +, -, # for public, private, protected
- Need not provide all details
  - Missing members indicated with ellipsis (…)

Class Interactions

- Class diagrams alone of little value
  - Just repeat of class interface, often "less"
- Must show how objects of various classes interact
  - Annotated arrows show information flow between class objects
    - Recall Model-View-Controller Pattern
  - Annotations also for class groupings into library-like aggregates
    - Such as for inheritance

More Class Interactions

- UML is extensible
  - If your needs not in UML, add them to UML!
- Framework exists for this purpose
  - Prescribed standard for additions
  - Ensures different software developers understand each other's UML

Summary

- Patterns are design principles
  - Apply across variety of software applications
- Pattern can provide framework for comparing related algorithms" efficiency
- Unified Modeling Language (UML)
  - Graphical representation language
  - Designed for object-oriented software design
- UML is one formalism used to express patterns