Design Principles

• Data Structures and Algorithms
• Design Goals
• Implementation Goals
• Design Principles
• Design Techniques

Data Structures

• Data Structure - A systematic way of organizing and accessing data.
• In the most general sense, a data structure is a data container into which we can store data.
• However, to access the data in the container, we must follow certain rules …
Abstract Data Types

- The rules that we follow to access data in a data structure is part of the abstract data type (ADT) for that data structure.
- In the object-oriented environment, the ADT also defines what messages the data structure must respond to.

Example - the Simplest Data Structure

- An array is the simplest data structure that almost all programming languages support.
- An array is a data container.
- Rules to access an array:
  - An array has a pre-defined, non-changeable capacity $N$.
    - Example: `int a[] = new int[100];`
  - Elements in the array are indexed by an integer from 0 to $N$-1.
  - When putting data into the array, the index must be in range.
    - Example: `a[3] = 100;`
  - When access data from the array, the index must be in range.
    - Example: `int k = a[3];`
Algorithms

• Algorithm: A step-by-step procedure for performing some tasks in a finite amount of time.
  – This implies that an algorithm should always terminate.
• Algorithms are used with data structures to ensure the rules for the ADT are enforced.

Design Goals

• Using Data Structures and Algorithms to achieve:
  – Correctness: Data structures/Algorithms are designed, at a high level, to always work correctly.
  – Efficiency: Data structures/Algorithms should be
    • as fast as possible;
    • use as little resources (e.g. memory) as possible.
Implementation Goal

- The code we write should have the following properties:
  - Robustness
  - Adaptability
  - Reusability

Robustness

- The code should handle all possible inputs (i.e., never “stall”, “crash” or “quit”).
- The code should recover gracefully from hardware or system failure (e.g., printer out-of-paper, file missing).
- The code should handle gracefully for system limitations (e.g., “out-of-memory” errors).
Adaptability

- Software should facilitate changes over time (in response to changing conditions).
- The code should be able to adapt to unexpected events.
- The “Y2K” problem was a very good counter example that how much it would cost us for lack of adaptability.

Reusability

- Software should facilitate reuse of components in other software systems.
- Example: the user interface code in every office suite can be reused for every component in the suite.
Design Principles

- Abstraction
- Encapsulation
- Modularity and Hierarchy

Abstraction

- Distill a complicated system down to its most fundamental parts.
- Describe these parts in simple and precise language.
- Specify interface but not implementation.
- Example: Abstract Data Types (ADT)
  - Specify types of data stored.
  - Specify operations and parameters.
Encapsulation

- Encapsulation = Information Hiding.
- Each software component should implement an abstraction without revealing the internal details of the implementation.
- Example: A stack can be implemented with either an array or a linked list, the user does not need to know about the details in order to use the stack.

Modularity and Hierarchy

- Modularity: Organizing the structures in which the different components are divided into separate functional units.
- Hierarchy: The relationship among the different modules.
### Hierarchy Example

- Building
  - Apartment
    - Low-Rise Apartment
    - High-Rise Apartment
  - House
    - Two-Story House
  - Commercial
    - Ranch
    - Skyscraper

### Design Techniques

- Interfaces and Typing
- Inheritance and Polymorphism
- Classes and Objects
- Design Patterns
**Interfaces and Strong Typing**

- All instances of one class have the same interface: a set of messages to which they respond in a prescribed way.
- This is called Application Programming Interface (API) or just short for Interface.
- Strong Typing: All types of parameters passed to methods MUST conform with the interface.
- Type checks of a strong typing language are done at compile time.
- To convert objects among different types, casting must be used.

**Inheritance**

- Subclass inherits all instance variables and methods from superclass.
- Object of Class S has instance variable x.
- Object of Class T has instance variables x and y.
- Object of Class S can respond to methods a, b and c.
- Object of Class T can respond to methods a, b, c, d and e.
- Message: “o.a()” means sending “a()” message to object “o”.

```
Class: S
Fields: x
Methods: a ()
         b ()
         c ()

extends

Class: T
Fields: y
Methods: d ()
         e ()
```
Method Overridden

- Subclass can also change (override) the way how it responds to a message.
- Object of Class T can still respond to methods a, b and c.
- However, the way it responds to the method a and b is different than its superclass.

Polymorphism

- Polymorphism - an object of Class T will respond to the “a ()” message using the overridden implementation, whether it has been typecast to S or not.
- This is also called late binding or dynamic binding as the determination of which implementation to execute is deferred until runtime.
- In Java, methods are dynamically bound by default. Only final methods use static binding.
Classes and Objects

- Classes define the behavior of objects.
  - What variables to hold data
  - What messages to respond to
- Objects are the actual actors in application programs
  - They actually hold variables (in memory)
  - They actually respond to messages

Static methods and variables

- The definition of a class is held in a Class object.
- The Class object also holds the static variables and responds to the static methods.
- When we send a message to a static method, we are sending the message to the Class object of that class, not any instance object of that class.
Design Patterns

- A design pattern describes a solution to a “typical” software design problem.
- A pattern provides a general template for a solution that can be applied in many different situations.
- It describes the main elements of a solution in an abstract way that can be specialized for the specific problem at hand.
- Design patterns we will be studying in this course:
  - Adapter (Chapter 4)
  - Iterator (Chapter 5)
  - Position (Chapter 5)
  - Composition (Chapter 7)
  - Comparator (Chapter 7)

Summary

<table>
<thead>
<tr>
<th>Design Goals</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Efficiency</td>
</tr>
<tr>
<td>Implementation Goals</td>
<td>Robustness</td>
</tr>
<tr>
<td></td>
<td>Adaptability</td>
</tr>
<tr>
<td></td>
<td>Reusability</td>
</tr>
<tr>
<td>Design Principles</td>
<td>Abstraction</td>
</tr>
<tr>
<td></td>
<td>Encapsulation</td>
</tr>
<tr>
<td></td>
<td>Modularity</td>
</tr>
<tr>
<td>Design Techniques</td>
<td>Interfaces and Typing</td>
</tr>
<tr>
<td></td>
<td>Inheritance and Polymorphism</td>
</tr>
<tr>
<td></td>
<td>Classes and Objects</td>
</tr>
<tr>
<td></td>
<td>Design Patterns</td>
</tr>
</tbody>
</table>