Objectives

- To introduce the following design principles
  - Flexibility
  - Reusability
  - Efficiency
Flexibility

- Flexible design implies a design that can easily accommodate the changes.

- Aspects of flexibility:
  - Adding more of the same kind of functionality
    - Example (banking application): handle more kinds of accounts without having to change the existing design or code
  - Adding different functionality
    - Example: add withdraw function to existing deposit functionality
  - Changing functionality
    - Example: allow overdrafts
Flexibility: Designs for Adding More of the Same Kind of Functionality

- Registering members at a web site
  - Design 1:

```
WebSite
register()                   members  0..n
                       --------> Member
```

- Design 2: a flexible design

```
WebSite          members  0..n
                --------> Member
```

- YMember
- XMember
- StandardMember
Flexibility: Design for Adding Different Functionality

- Alternative situations - within the scope of:
  - A list of related functions
    - Example: add print to an air travel itinerary functions
  - An existing base class
    - Example: add “print road- and ship- to air itinerary”
  - Neither
    - Example: add “print itineraries for combinations of air, road and ship transportation”
Flexibility: Design for Adding Different Functionality (Cont’d)

- Case 1: Handled by adding the new method to an existing set of methods of a class
  - Add method `print()` to the class `Itinerary`.

- Case 2: Adding functionality when a base class exists

```plaintext
SomeApplicationClass

Method(s) call printItinerary()

Trip
printItinerary()

StandardTrip
printItinerary()
```
Flexibility: Design for Adding Different Functionality (Cont’d)

- Case 2: adding functionality through a base class

```
SomeApplicationClass  --- Trip
                     |   printItinerary()
                     v
StandardTrip         --- LandTrip    --- SeaTrip
printItinerary()    printItinerary() printItinerary()
```

- Case 3: covered later (chapter 9)
## Additional types of flexibility

<table>
<thead>
<tr>
<th>Flexibility Aspect: ability to ...</th>
<th>Described in ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>... create objects in variable configurations determined at runtime</td>
<td>“Creational” design patterns – chapter xx</td>
</tr>
<tr>
<td>... create variable trees of objects or other structures at runtime</td>
<td>“Structural” design patterns – chapter xx</td>
</tr>
<tr>
<td>... change, recombine, or otherwise capture the mutual behavior of a set of objects</td>
<td>“Behavioral” design patterns – chapter xx</td>
</tr>
<tr>
<td>... create and store a possibly complex object of a class.</td>
<td>Component technology – chapter xx</td>
</tr>
<tr>
<td>... configure objects of predefined complex classes – or sets of classes – so as to interact in many ways</td>
<td>Component technology – chapter xx</td>
</tr>
</tbody>
</table>
Reusability

- Designing a software system such that components of a developed system can be used again in developing new applications.

- Reusability is an important factor in producing low cost applications.

- Reusability at …
  - Architectural/design level
  - Implemented component level

- This chapter focuses on:
  - Reusable functions
  - Reusable classes
  - Reusable class combinations
Making a Method Reusable

- Specify completely
  - Preconditions etc.

- Avoid unnecessary coupling with the enclosing class
  - Make static if feasible
  - Include parameterization
    - But limit the number of parameters

- Make the names expressive
  - Understandability promotes re-usability

- Explain the algorithm
  - Re-users need to know how the algorithm works
Making a Class Reusable

- Describe the class completely

- Make the class name and functionality match a real world concept

- Define a useful abstraction
  - Attain broad applicability

- Reduce dependencies on other classes
  - Elevate dependencies in hierarchy
Reducing Dependency Among Classes

Example:

Replace …

```
Customer <-> Piano
```

with …

```
Customer <-> PianoOrder <-> Piano
```

Increase the reusability of a class by reducing its dependencies.
Reusability of Class Combinations

- Design patterns are reusable class combinations.
- Leveraging inheritance, aggregation and dependency for the reuse of class combinations
Leveraging Inheritance, Aggregation and Dependency for the Reuse of Class Combinations

1. Leveraging inheritance
2. Leveraging aggregation
3. Leveraging dependency
Efficiency

- Applications must execute required functionality within required time constraints.

- Two dimensions of efficiency:
  - Time
  - Space

- Ideally optimize a design for both time and space.
Space-Time Trade-offs

Time to process one item

Typical target

Space
Basic Approaches to Designing Efficiently

- Design for other criteria, then consider efficiency
  - Design for flexibility, reusability, …
  - At some point, identify inefficient places
  - Make targeted changes to improve efficiency

- Design for efficiency from the start
  - Identify key efficiency requirements up front
  - Design for these requirements during all phases

- Combine these two approaches
  - Make trade-offs for efficiency requirements during design
  - Address remaining efficiency issues after initial design
Speed Efficiency

- Real-time applications are the most demanding in terms of speed.
- Profiler—an application that tracks resource usage during program execution
- Impediments to speed efficiency
  - Loops
    - while, for, do
  - Remote operations
    - Requiring a network
      - LAN
      - The Internet
  - Function calls
  - Object creation
Trade-off Between Number of Remote Calls and Volume Retrieved at Each Call
Storage Efficiency

- Attaining Storage Efficiency
  - Store only the data needed
    - Trades off storage efficiency vs. time to extract and reintegrate
  - Compress the data
    - Trades off storage efficiency vs. time to compress and decompress
  - Store in order of relative frequency
    - Trades off storage efficiency vs. time to determine location
Trading off Robustness, Flexibility, Efficiency and Reusability

1A. Extreme Programming Approach
   Design for sufficiency only
or
1B. Flexibility-driven Approach
   Design for extensive future requirements
   Reuse usually a by-product

2. Ensure robustness

3. Provide enough efficiency
   Compromise re-use etc. as necessary to attain efficiency requirements

* Note that correctness is not included in trade-offs!
## Pros and Cons of Extreme vs. Nonextreme Design

<table>
<thead>
<tr>
<th>Extreme</th>
<th>NonExtreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Job done faster (usually)</td>
<td>+ Future applications more likely to use parts</td>
</tr>
<tr>
<td>+ More likely to be efficient</td>
<td>+ Accommodates changes in requirements</td>
</tr>
<tr>
<td></td>
<td>- Potential to waste effort</td>
</tr>
<tr>
<td>- Future applications less likely to use the work</td>
<td>- Efficiency requires more special attention</td>
</tr>
<tr>
<td>- Refactoring for expanded requirements can be expensive</td>
<td></td>
</tr>
</tbody>
</table>
A More Flexible Design for Calculator Application

**Existing Design**

```
CommandLineCalculator
  main()
  executeAdditions()
  solicitNumAccounts()
  getAnInputFromUser()
  interactWithUser()
```

**New Design**

```
Calculator
  solicitNumAccounts()

CalcDisplay
  display()

CalcOperation
  execute()

Add
Multiply
Divide
```
Summary

- **Flexibility**
  - readily changeable

- **Reusability**
  - in other applications

- **Efficiency**
  - in time
  - in space

(These add to Correctness & Robustness, covered in chapter 4)