Chapter 5
Design Principles II:
Flexibility, Reusability, and Efficiency

Aspects of Flexibility

Anticipate ...

- … 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

Registering Website Members

<table>
<thead>
<tr>
<th>WebSite</th>
<th>register()</th>
</tr>
</thead>
<tbody>
<tr>
<td>members</td>
<td>0..n Member</td>
</tr>
</tbody>
</table>
Registering Website Members Flexibly

WebSite → members 0..n → Member

YMMember XMember StandardMember

Adding Functionality to an Application: 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”

Adding Functionality When a Base Class Exists

SomeApplicationClass

Method(s) call printItinerary()

StandardTrip printItinerary()

Adding Functionality Through a Base Class

SomeApplicationClass

Trip printItinerary()

StandardTrip printItinerary()

LandTrip printItinerary()

SeaTrip printItinerary()
### Additional Type 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>“Creationa&quot; design patterns –</td>
</tr>
<tr>
<td>... create variable trees of objects or other structures at runtime</td>
<td>“Structural&quot; design patterns –</td>
</tr>
<tr>
<td>... change, recombine, or otherwise capture the mutual behavior of a set of objects</td>
<td>“Behavioral&quot; design patterns –</td>
</tr>
<tr>
<td>... create and store a possibly complex object of a class.</td>
<td>Component technology –</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 –</td>
</tr>
</tbody>
</table>

#### Making a Method Re-usable

- Specify completely
  - Preconditions etc
  - Avoid unnecessary coupling with the enclosing class
  - Make static if feasible
  - Include parameterization
    - i.e., make the method functional
    - 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 Re-usable

- 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

---

**Key Concept: Flexibility**

We design flexibly, introducing parts, because change and reuse are likely.
Reducing Dependency Among Classes

Replace ...

Customer --> Piano

with ...

Customer --> PianoOrder --> Piano

Leveraging Inheritance, Aggregation & Dependency for the Re-use of Class Combinations

(1) Leveraging inheritance

Customer

computeBill()

RegularCustomer

computeBill()

(2) Leveraging aggregation

Customer

computeBill()

Bill

compute()

(3) Leveraging dependency

Customer

computeBill(Orders)

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.

Basic Approaches to Time Efficiency

- 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

Space-Time Trade-offs

Time to process one item

Typical target

Space

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Space-Time-Development Trade-offs

Convenience of Development

- Limiting acceptable value
  - unacceptable values
  - better than acceptable values

Space

Time

Impediments to Speed Efficiency

- **Loops**
  - while, for, do

- **Remote operations**
  - Requiring a network
    - LAN
    - The Internet

- **Function calls**
  - if the function called results in the above

- **Object creation**

Trade-off Between Number of Remote Calls and Volume Retrieved at Each Call

- Volume retrieved at each access
- Typical target
- Number of remote accesses

Storage Locations

- Runtime
  - RAM
  - Disk storage required at runtime
  - Disk storage required between runtimes

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Adapted from *Internet-based Application Development*, copyright (c) 1998-2001 by Eric J. Braude.
Attaining Storage Efficiency

- Store only the data needed
  - Trades off storage efficiency vs. time to extract and re-integrate
- 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
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

Trading off Robustness, Flexibility, Efficiency and Reusability

Extreme vs. non-Extreme

+ Job done faster (usually)
+ Scope clear
+ More likely to be efficient
- Future applications more likely to use parts
- Accommodates changes in requirements
- Scope less clear
- Potential to waste effort
- Efficiency requires more special attention

A More Flexible Design for Calculator Application

Existing Design

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

New Design

```
Calculator
solicitNumAccounts()
execute()
display()
```

Adapted from "Software Design: From Programming to Architecture" by Eric J. Braude (Wiley 2003), with permission.
Summary of This Chapter

- **Flexibility**
  - readily changeable

- **Reusability**
  - in other applications

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

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.