Chapter 5
Design Principles II:
Flexibility, Reusability, and Efficiency
Process Phase Affected by This Chapter

Requirements Analysis

Design

Framework Architecture Detailed Design

Implementation

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

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

![Diagram showing the relationship between Website and Member with the method register() and 0..n members association.](image-url)
Registering Website Members Flexibly

[Diagram showing relationships between Website, Member, YMember, XMember, and StandardMember with a 0..n association]

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
Adding Functionality to an Application: Alternative Situations

Within the scope of …

1. … a list of related functions
   Example: add print to an air travel itinerary functions

2. … an existing base class
   Example: add “print road- and ship- to air itinerary ”

3. … neither
   Example: add “print itineraries for combinations of air, road and ship transportation”

Adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.
Adding Functionality When a Base Class Exists

SomeApplicationClass

Method(s) call printItinerary()

Trip
printItinerary()

StandardTrip
printItinerary()
Adding Functionality Through a Base Class

SomeApplicationClass

StandardTrip
  printItinerary()

LandTrip
  printItinerary()

SeaTrip
  printItinerary()

Trip
  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>“Creational” design patterns –</td>
</tr>
<tr>
<td>... create variable trees of objects or other structures at runtime</td>
<td>“Structural” design patterns –</td>
</tr>
<tr>
<td>... change, recombine, or otherwise capture the mutual behavior of a set of objects</td>
<td>“Behavioral” 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>
Key Concept: \[ \rightarrow \] Flexibility \[ \leftarrow \]

We design flexibly, introducing parts, because change and reuse are likely.
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

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

Alternatives:
- Define a useful abstraction
  - attain broad applicability

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

(2) Leveraging aggregation

RegularCustomer computeBill()

Customer computeBill()

(3) Leveraging dependency

Bill compute()

Customer computeBill()

Orders value()

Orders 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**
- **better than acceptable values**
- **unacceptable values**

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

Number of remote accesses

Volume retrieved at each access

Typical target

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

| Runtime RAM | Code base |

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.
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
   - 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

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

<table>
<thead>
<tr>
<th>Extreme</th>
<th>vs.</th>
<th>non-Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Job done faster (usually)</td>
<td></td>
<td>+ Future applications more likely to use parts</td>
</tr>
<tr>
<td>+ Scope clear</td>
<td></td>
<td>+ Accommodates changes in requirements</td>
</tr>
<tr>
<td>+ More likely to be efficient</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Future applications less likely to use the work
- Refactoring for expanded requirements can be expensive

- Scope less clear
- Potential to waste effort
- Efficiency requires more special attention

---

Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.
A More Flexible Design for *Calculator* Application

## Existing Design

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

## New Design

- **Calculator**
  - `solicitNumAccounts()`

- **CalcDisplay**
  - `display()`

- **CalcOperation**
  - `execute()`

- `Add`
- `Multiply`
- `Divide`

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