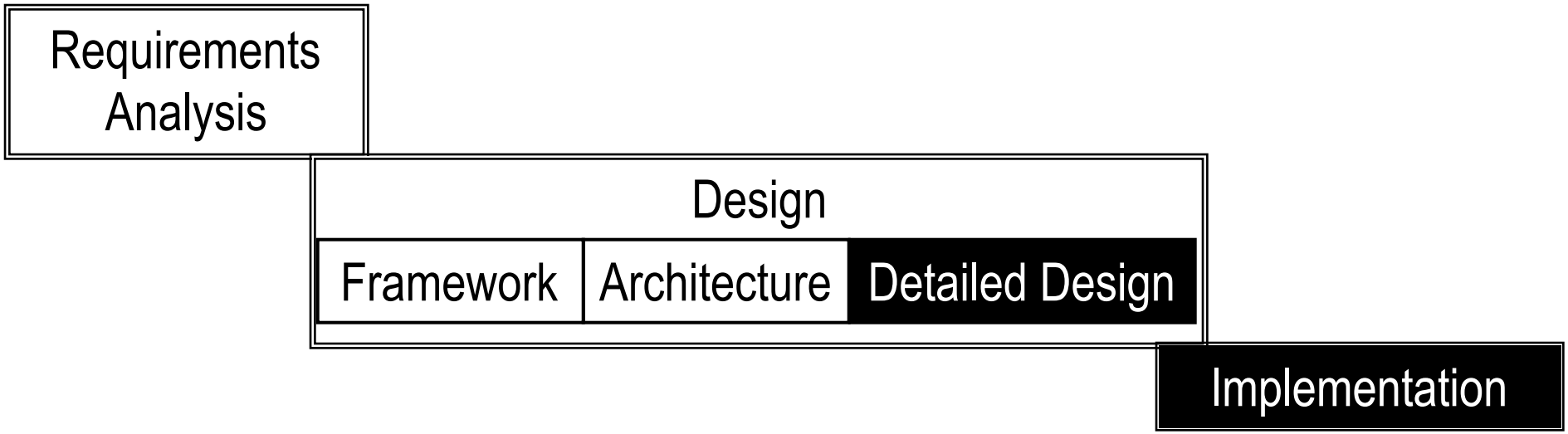
Chapter 5
Design Principles II:
Flexibility, Reusability, and Efficiency

Process Phase Affected by This Chapter



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

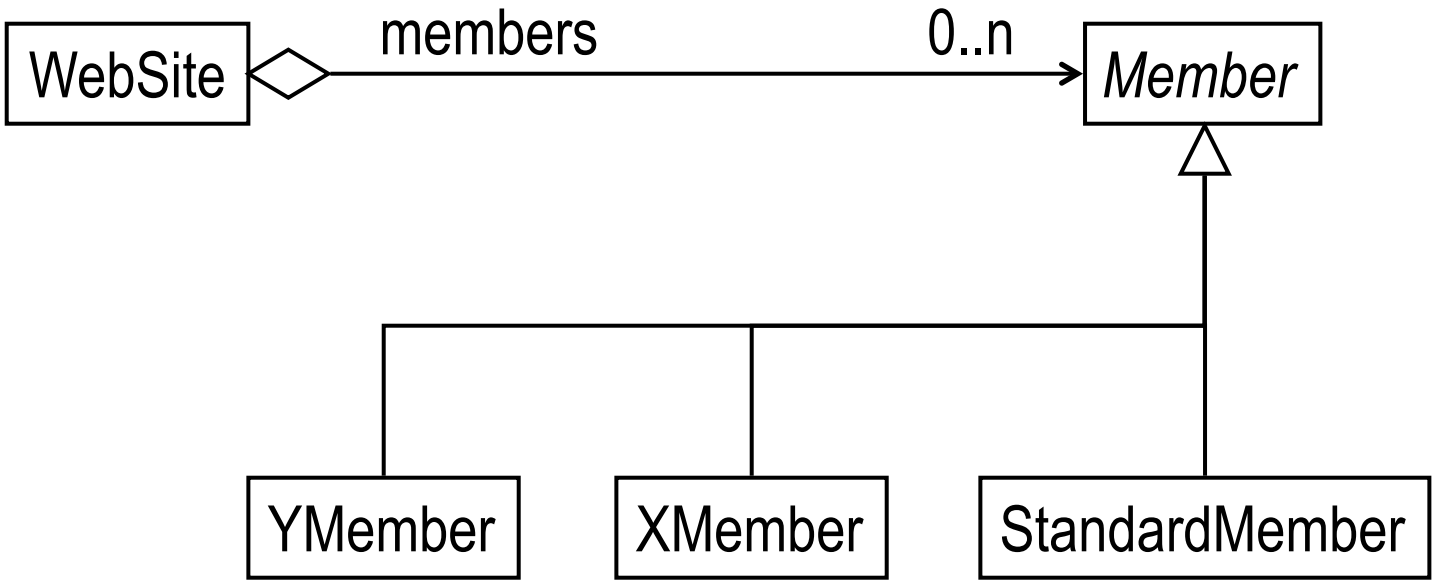
  Example: add *withdraw* function to existing *deposit* functionality

- *… changing* functionality

  Example: allow overdrafts

Registering Website Members

Registering Website Members Flexibly



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

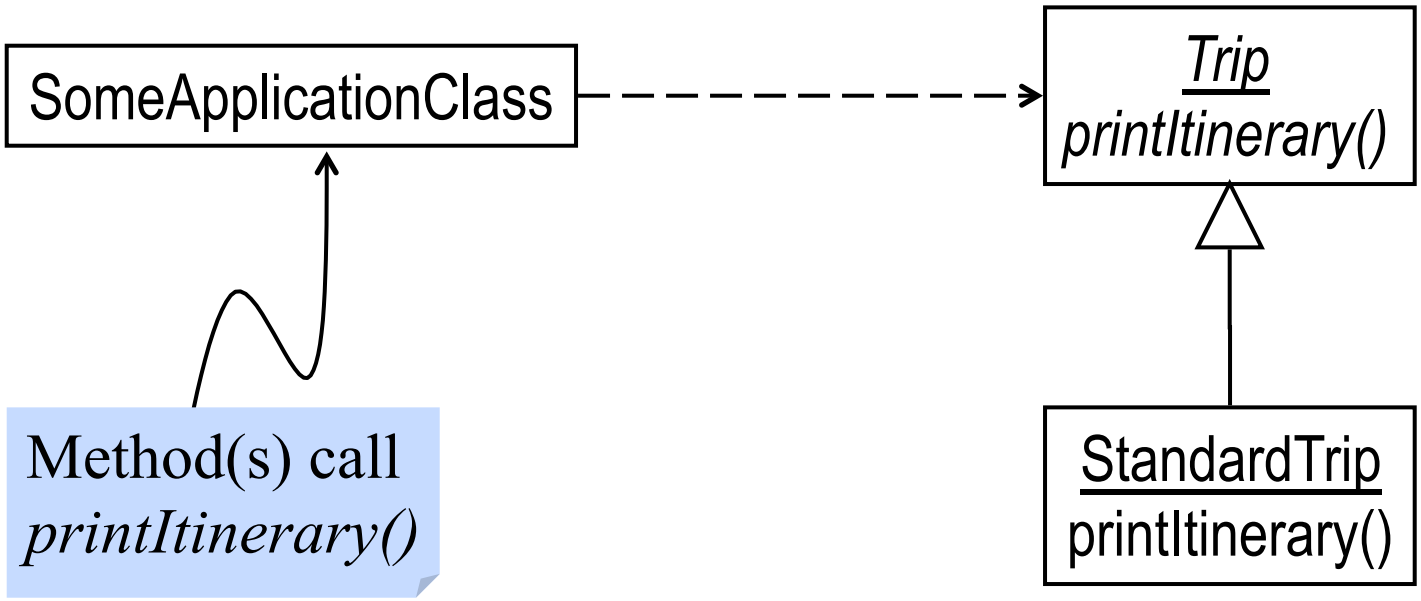
2. **... an existing base class**

   Example: add "print *road*- and *ship*- to air itinerary "

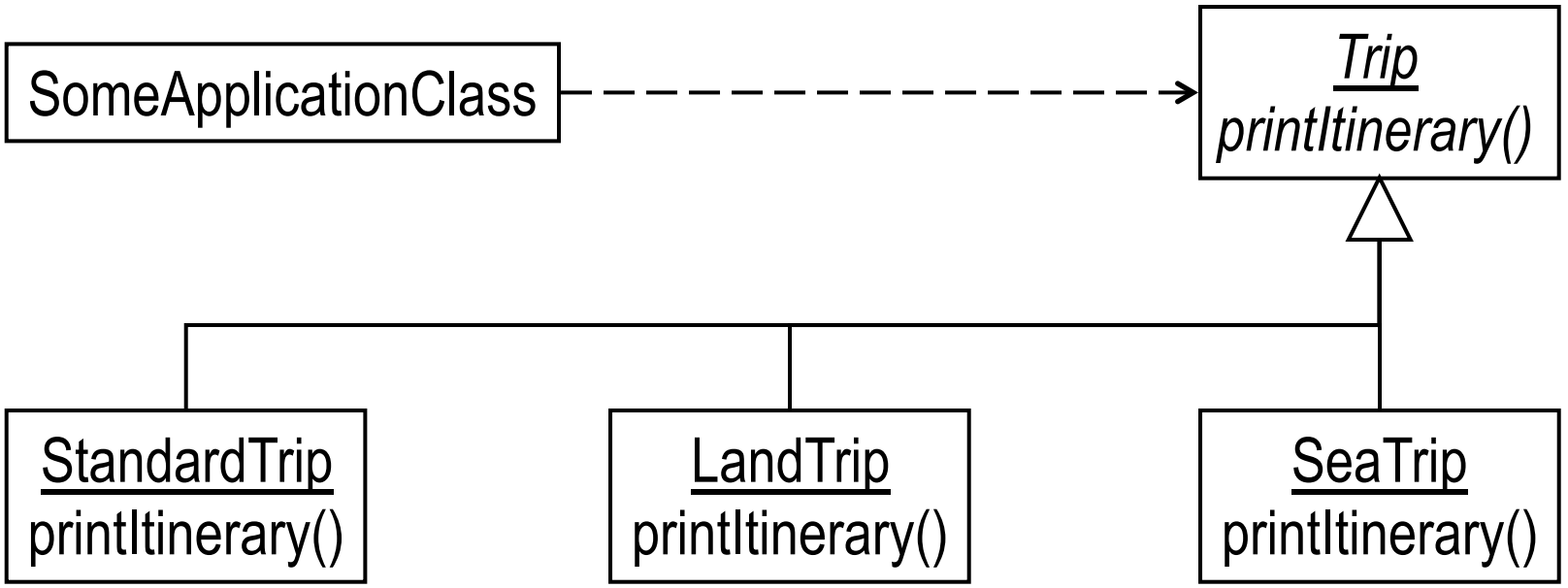
3. **... neither**

   Example: add "print itineraries for combinations of air, road and ship transportation"

Adding Functionality When a Base Class Exists



Adding Functionality Through a Base Class



Additional Type of Flexibility

Flexibility Aspect: ability to …	Described in …
… create objects in variable configurations determined at runtime	"Creational" design patterns –
… create variable trees of objects or other structures at runtime	"Structural" design patterns –
… change, recombine, or otherwise capture the mutual behavior of a set of objects	"Behavioral" design patterns –
… create and store a possibly complex object of a class.	Component technology –
… configure objects of predefined complex classes – or sets of classes – so as to interact in many ways	Component technology –

Key Concept: → Flexibility ←

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

Making a Class Re-usable

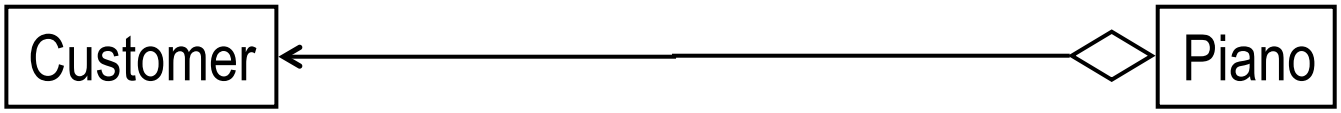
- Describe the class completely
- Make the class name and functionality match a real world concept

alternatives

- Define a useful abstraction
  - attain broad applicability
- Reduce dependencies on other classes
  - Elevate dependencies in hierarchy

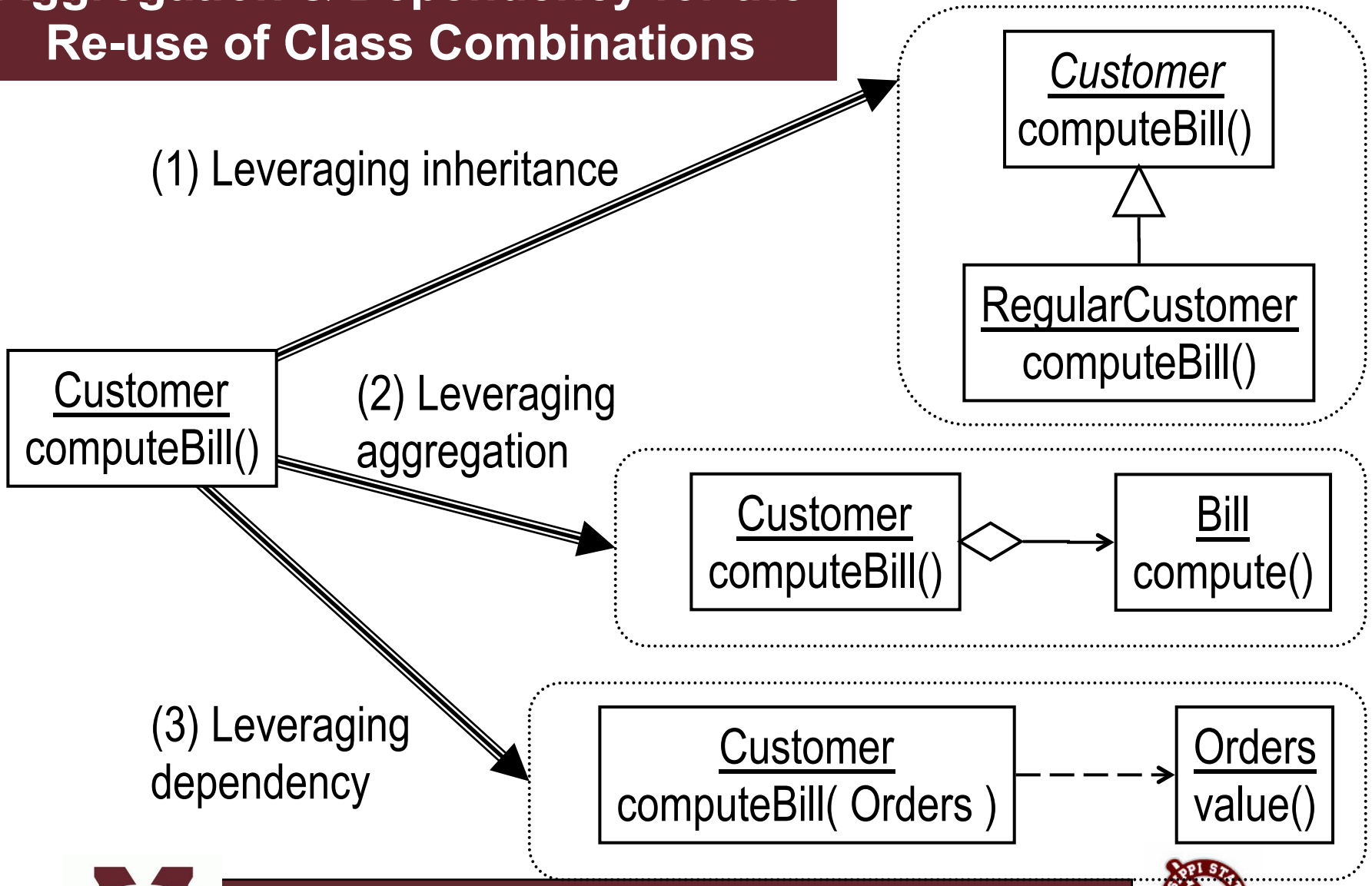
Reducing Dependency Among Classes

Replace …



with …

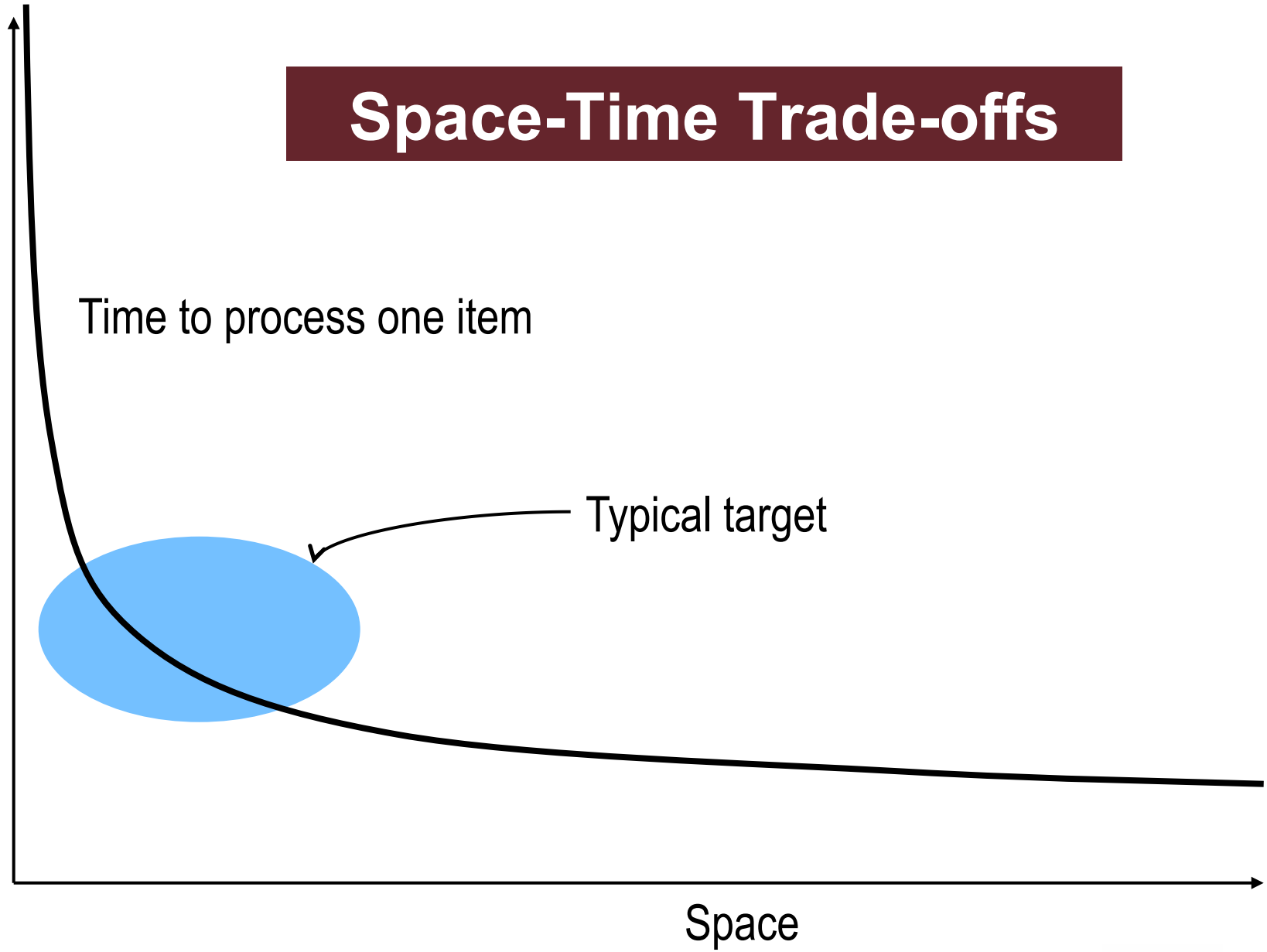
Leveraging Inheritance, Aggregation & Dependency for the Re-use of Class Combinations



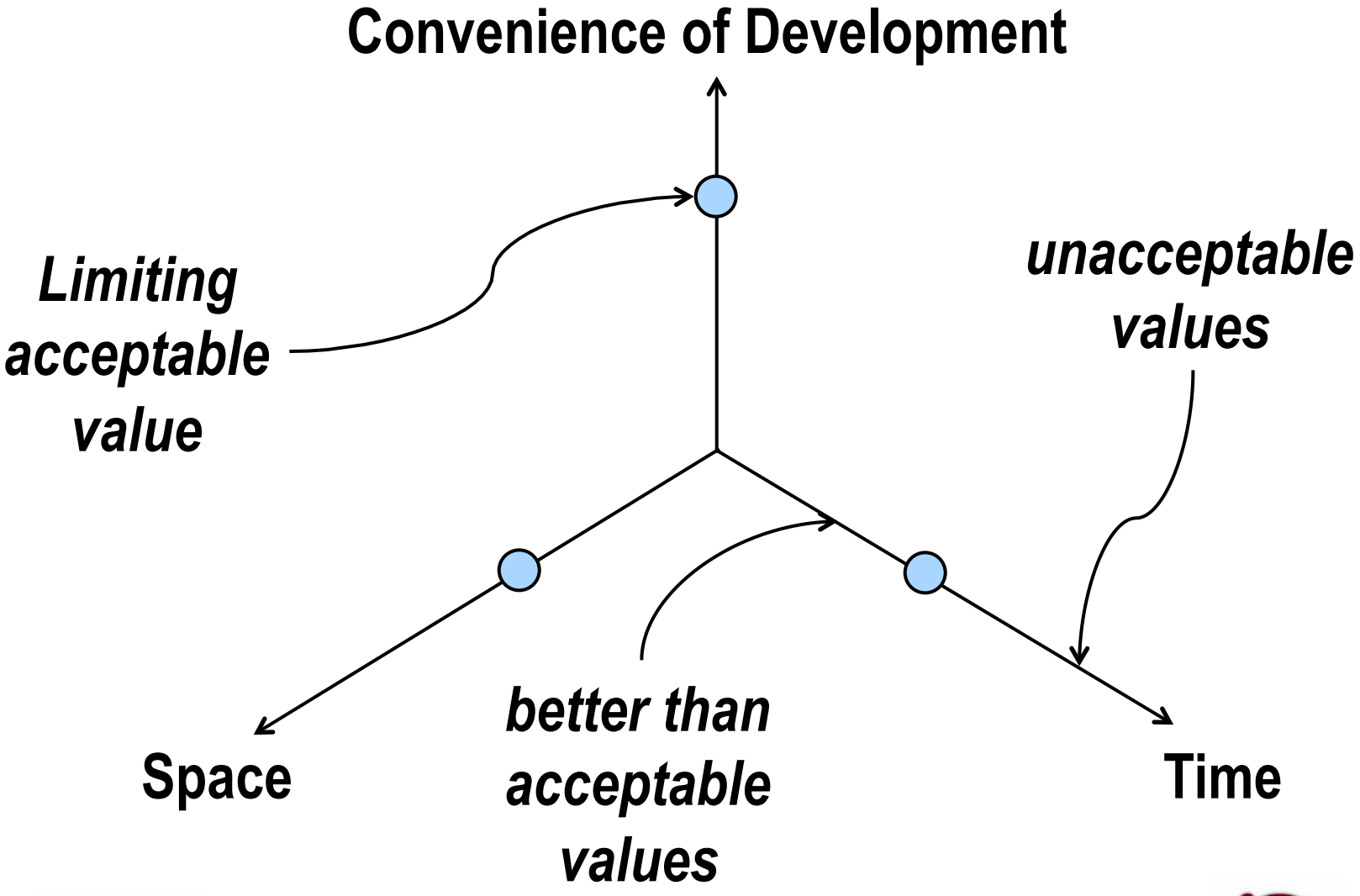
Basic Approaches to Time Efficiency

- ***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



Space-Time-Development Trade-offs

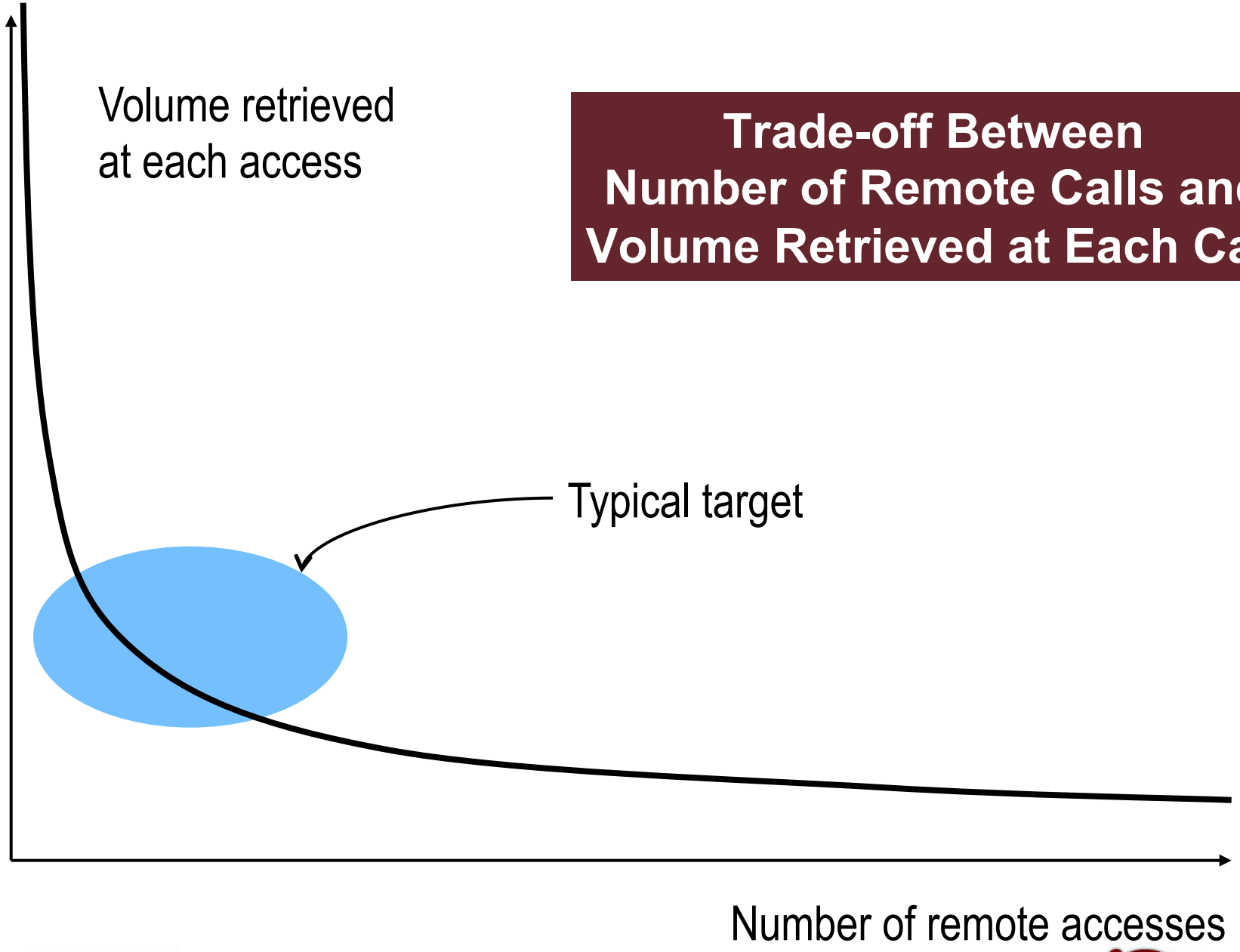


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

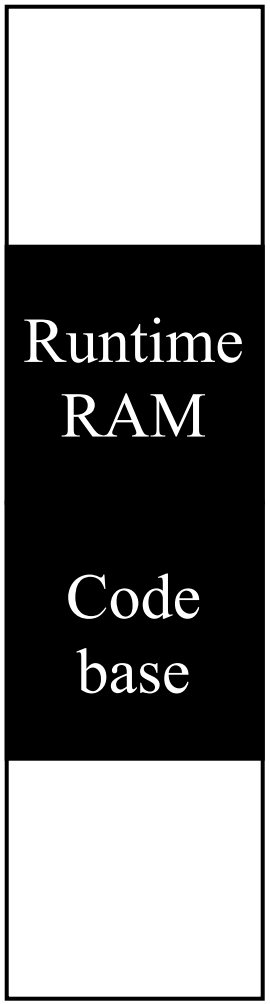
Impediments to Speed Efficiency

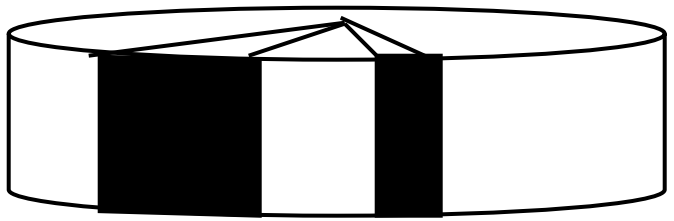
- *Loops*
  - while, for, do
- *Remote operations*
  - Requiring a network
    - LAN
    - The Internet
- *Function calls*
  - -- if the function called results in the above
- *Object creation*

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



Storage Locations





Disk storage required at runtime

Disk storage required between runtimes

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 - 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

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

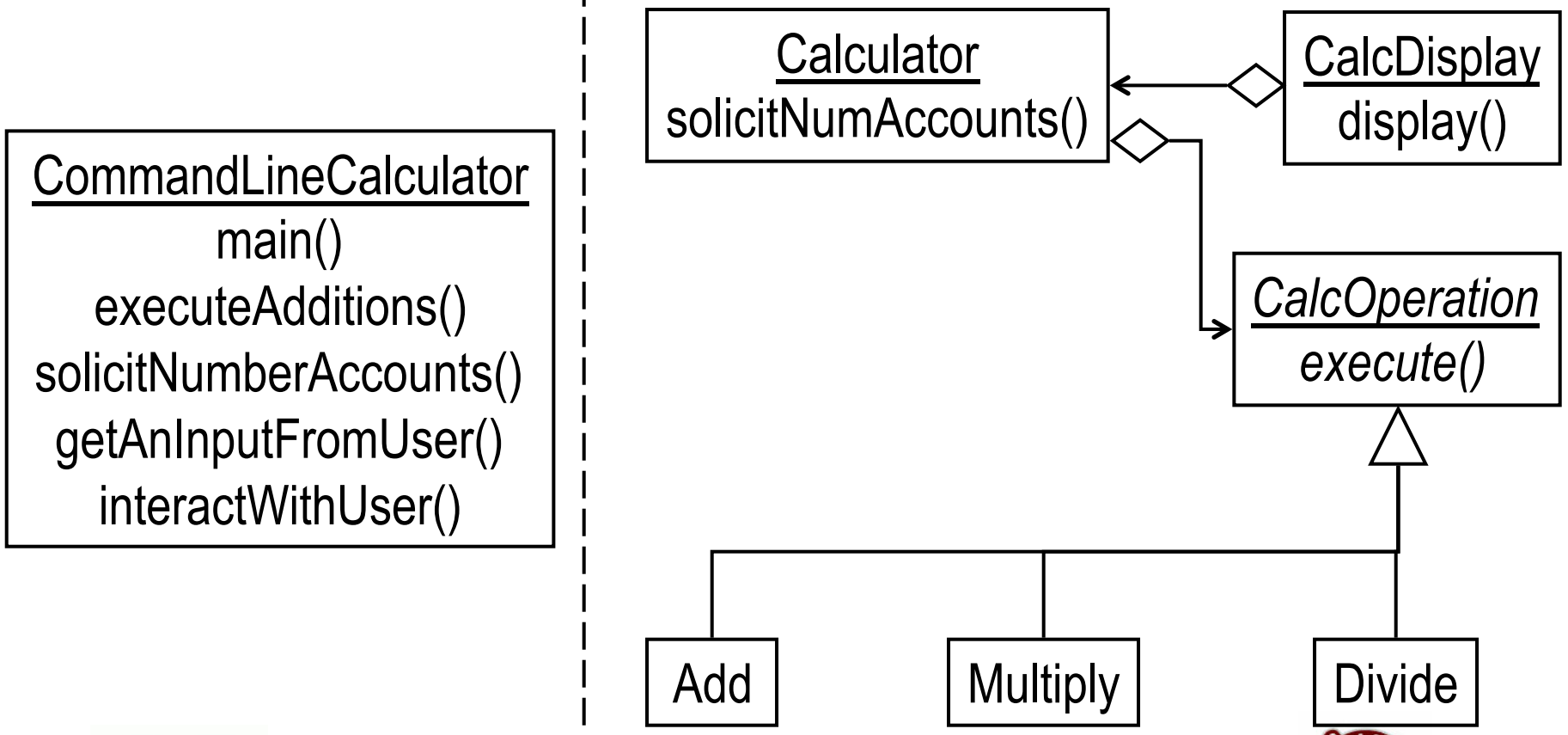
- Future applications less likely to use the work
- Refactoring for expanded requirements can be expensive

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

A More Flexible Design for *Calculator* Application

Existing Design

New Design



Summary of This Chapter

- ***Flexibility***
  - == readily changeable
- ***Reusability***
  - in other applications
- ***Efficiency***
  - in time
  - in space