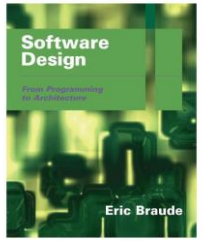
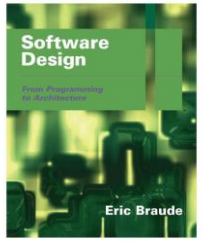


Lec 07 - Design Principles II

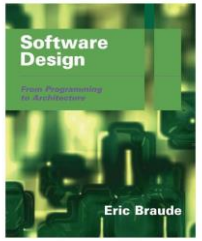
Topic covered



✧ Introducing design Principles:

- Flexibility
- Reusability
- Efficiency
- Reliability
- Usability

Flexibility



Flexible design implies a design that easily can accommodate the changes

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

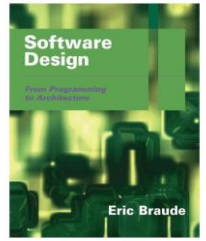
- ❑ *… adding different* functionality

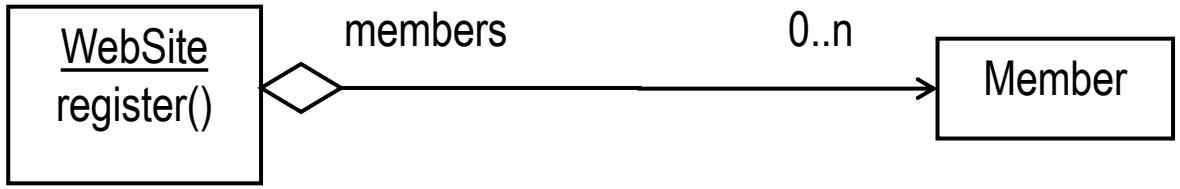
  Example: add *withdraw* function to existing *deposit* functionality

- ✧ *… changing* functionality

  Example: allow overdrafts

Registering Website Members

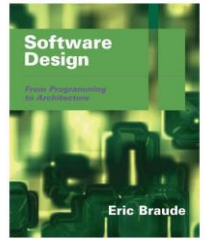


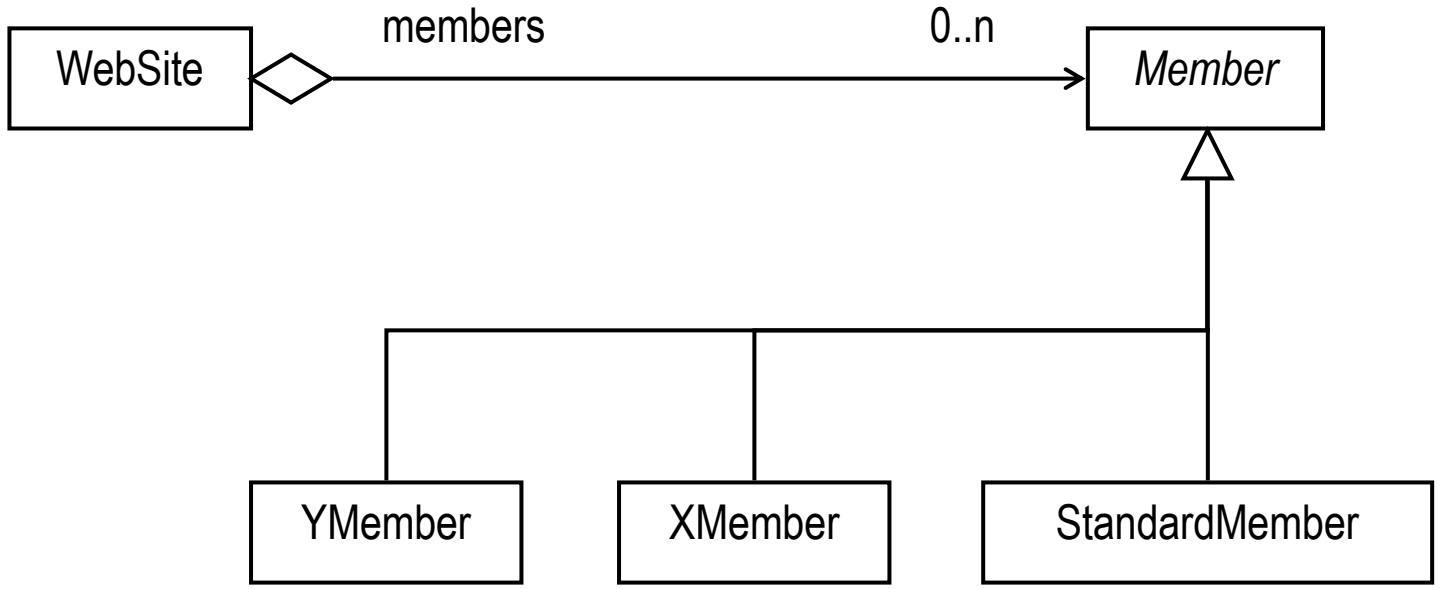


Example: Registering members at a web site (Design 1)

What's wrong with this design?

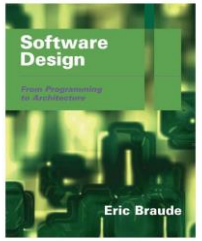
Registering Website Members Flexibly





Example: Registering members at a web site
(Design 2: a flexible design)

Adding Functionality to an Application



Alternative Situations

- ✧ Accommodate adding new kinds of functionality
- ✧ Adding functionality to an application :

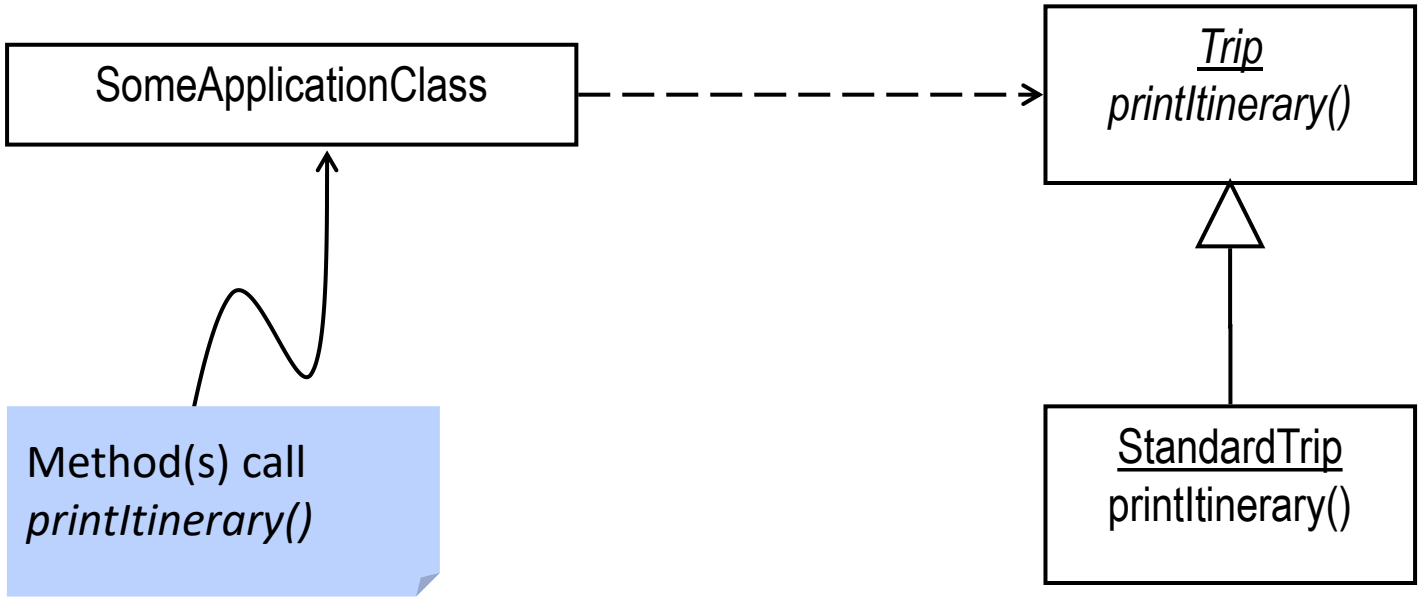
1. Within the scope of a list of related functions

   Example: add *print* to an air travel itinerary functions

2. Within the scope of an existing base class

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

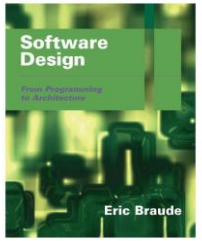
Adding Functionality When a Base Class Exists

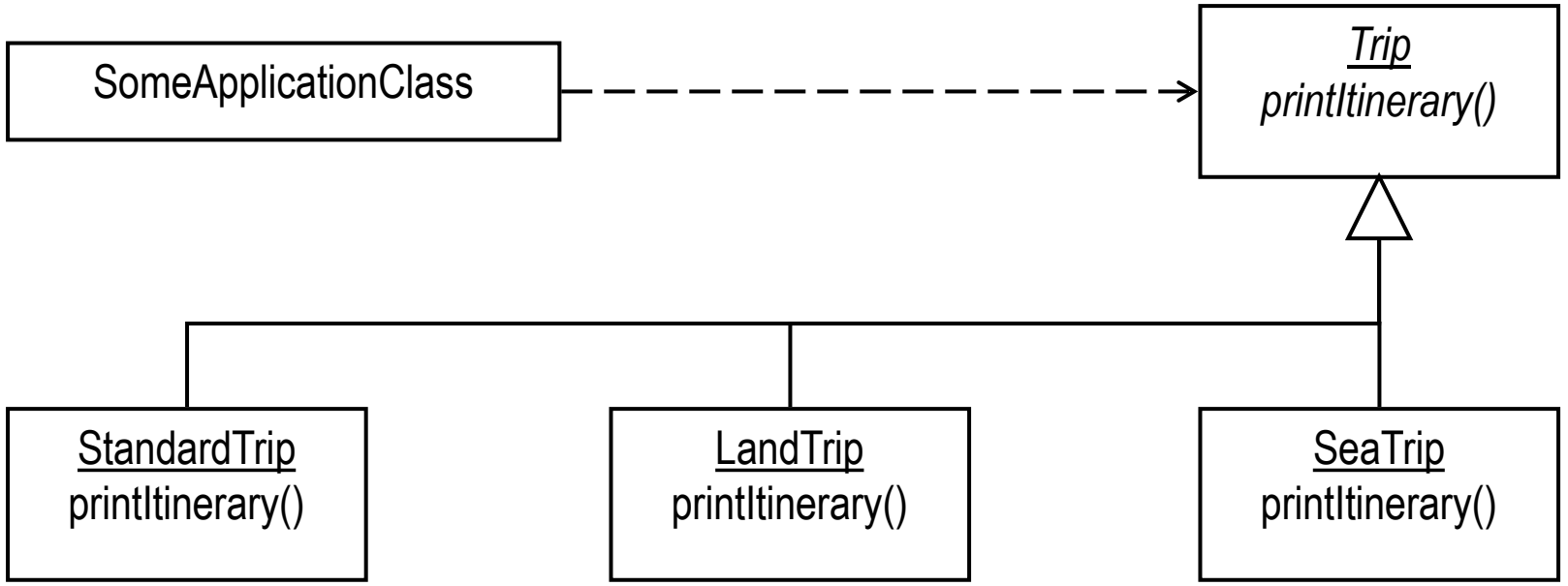


Case 1: Case 1 can be handled by adding the new method to an existing set of methods of a class

Add method print() to the class Itinerary.

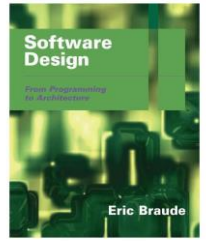
Adding Functionality Through a Base Class





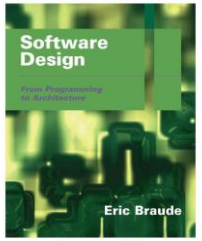
Case 2: Adding functionality when a base class exists

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 the architectural/design level (design patterns)
- ✧ Reusability at the implemented component level (design components)
- ✧ Our focus in this chapter is to cover:
  - Reusable functions
  - Reusable classes

Reusability of function design

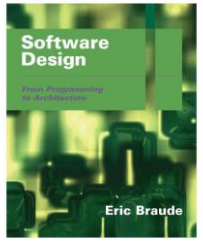


✧ Simple methods

✧ Guidelines for designing reusable methods

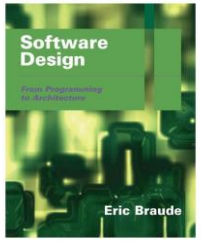
- Reusable methods must be defined completely
- Reusable methods must be independent of their environment, e.g., Static methods
  - Makes the design looser coupling, less coherent and less object-oriented.
- Name of a reusable method must be reflective of its functionality

Making a Method Re-usable



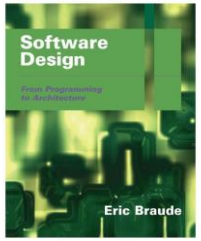
- Avoid unnecessary coupling with the enclosing class
  - Make static if feasible
  - Include parameterization
    - i.e., make the method functional
    - But limit the number of parameters
- Make the names expressive
  - Understandability promotes re-usability
- Explain the algorithm
  - Re-users need to know how the algorithm works

Making a Class Re-usable

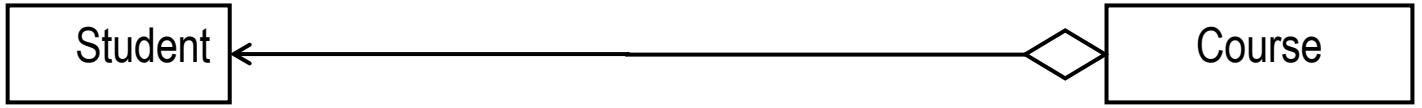


- Make the class name and functionality match a real world concept
- Reduce dependencies on other classes

Reducing Dependency Among Classes



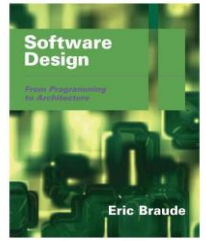
Replace …



With !!! What do you think?

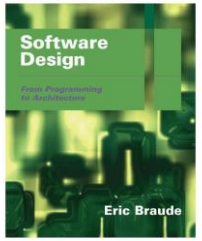
Increase the reusability of a class by reducing its dependencies.

Efficiency

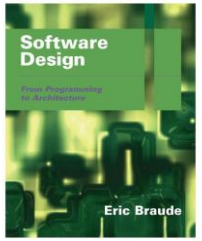


- Applications must execute required functionality within required time constraints
- Two dimensions of efficiency: time and space
- Ideally optimize a design for both time and space

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

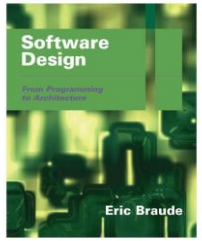


Speed efficiency

- ✧ Real-time applications are the most demanding in terms of speed

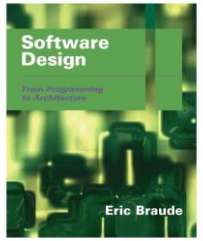
- ✧ Profiler--- an application that tracks resource usage during program execution

Impediments to Speed Efficiency



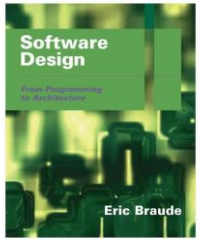
- *Loops*
  - while, for, do (think about sorting algorithms)
- *Remote operations*
  - Requiring a network
    - LAN
    - The Internet
- *Function calls*
  - -- if the function called results in the above
- *Object creation*

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

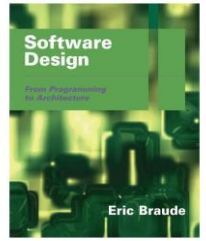
Design for extensive future requirements

Reuse usually a by-product

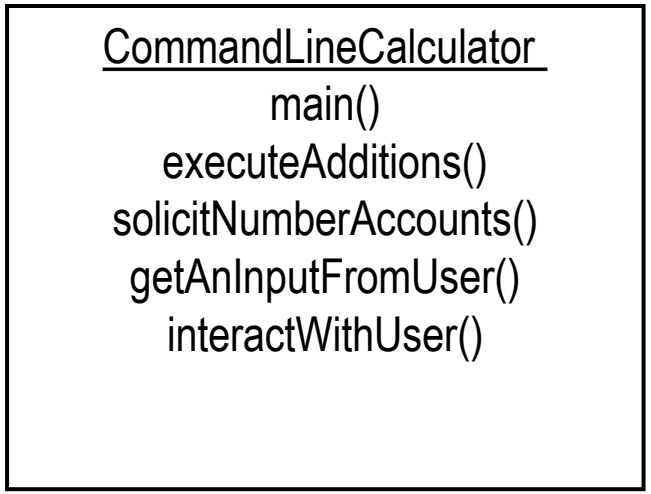
2. Ensure robustness

3. Provide enough efficiency

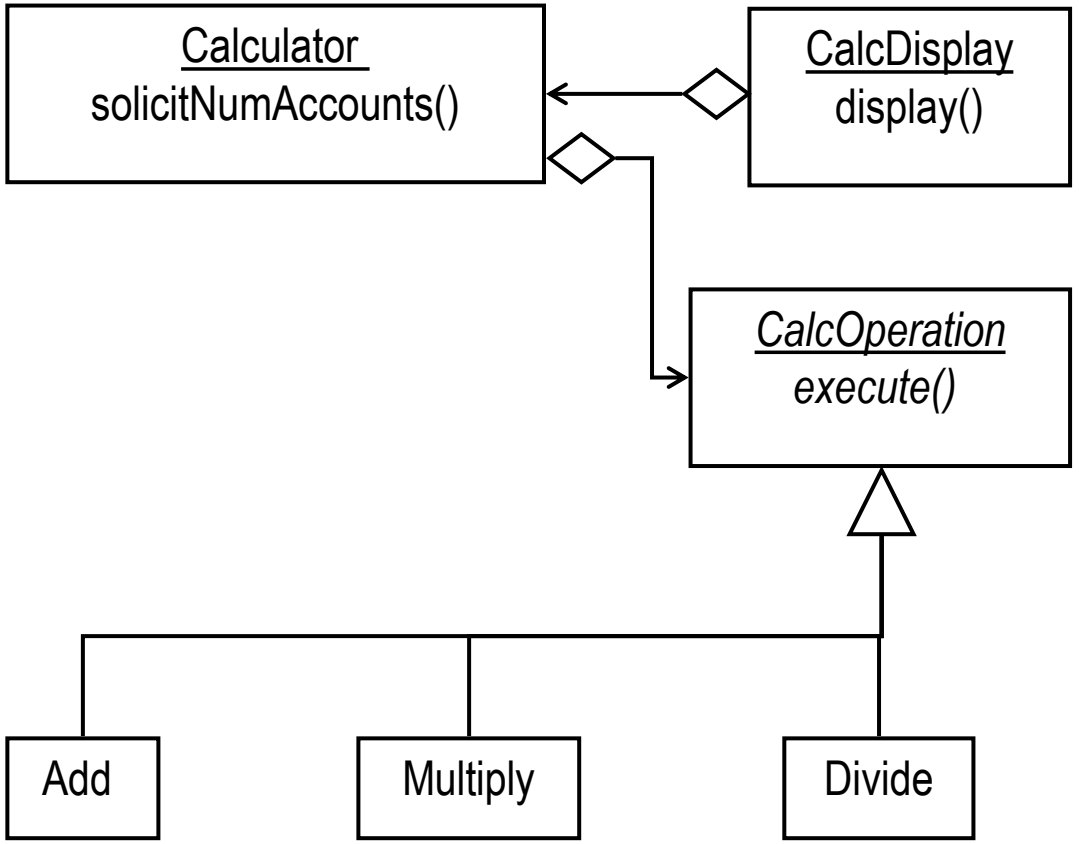
A More Flexible Design for *Calculator* Application



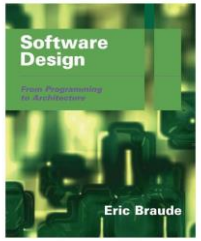
Existing Design



New Design



Summary



- *Flexibility*
  - == readily changeable
    We design flexibly, introducing parts, because change and reuse are likely.
- *Reusability*
  - in other applications
- *Efficiency*
  - in time
  - in space