Chapter 9
Behavioral Design Patterns

- □ Interpreter
- □ Iterator
- □ Mediator
- □ Observer
- □ State
- □ Chain of Responsibility
- □ Command
- □ Template

Interpreter Design Pattern

Interpreter

Design Purpose

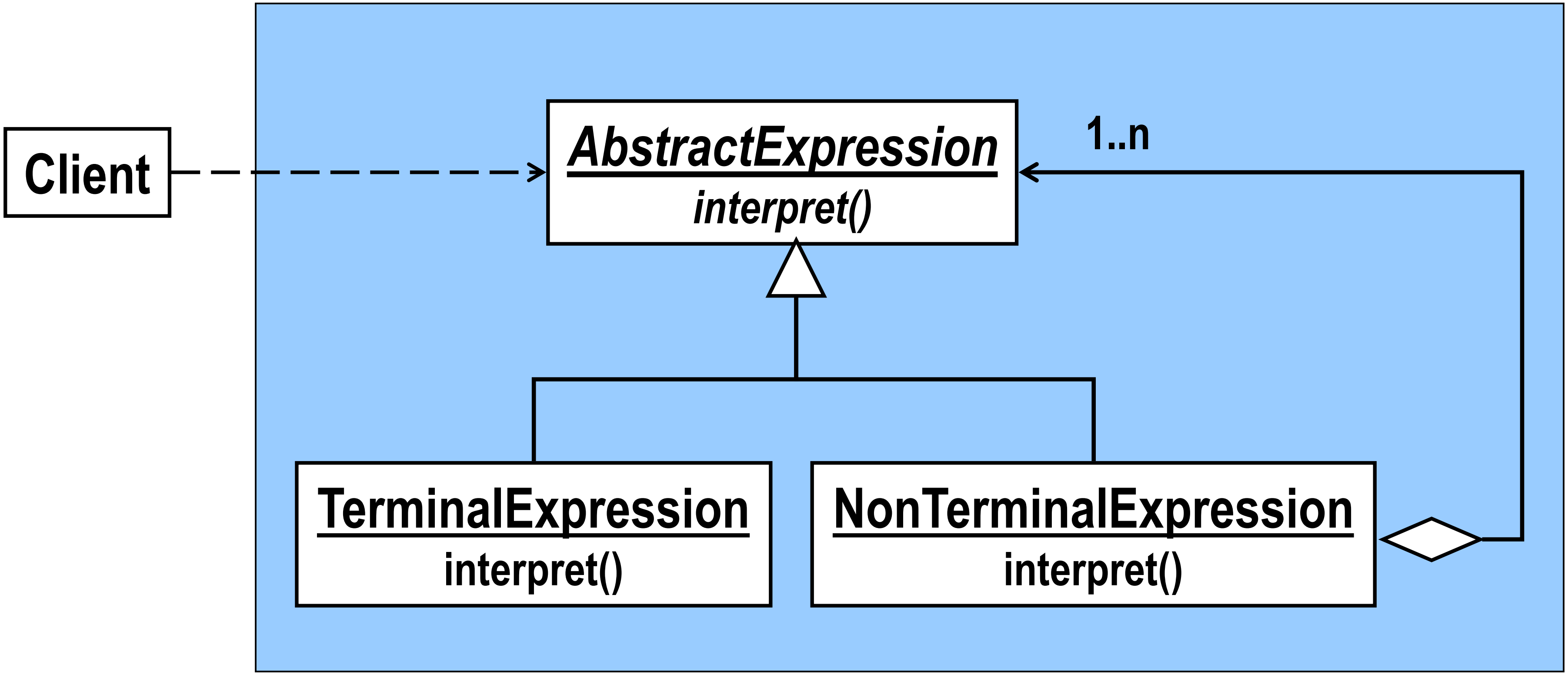
Interpret expressions written in a formal grammar.

Design Pattern Summary

Represent the grammar using a recursive design pattern form: Pass interpretation to aggregated objects.

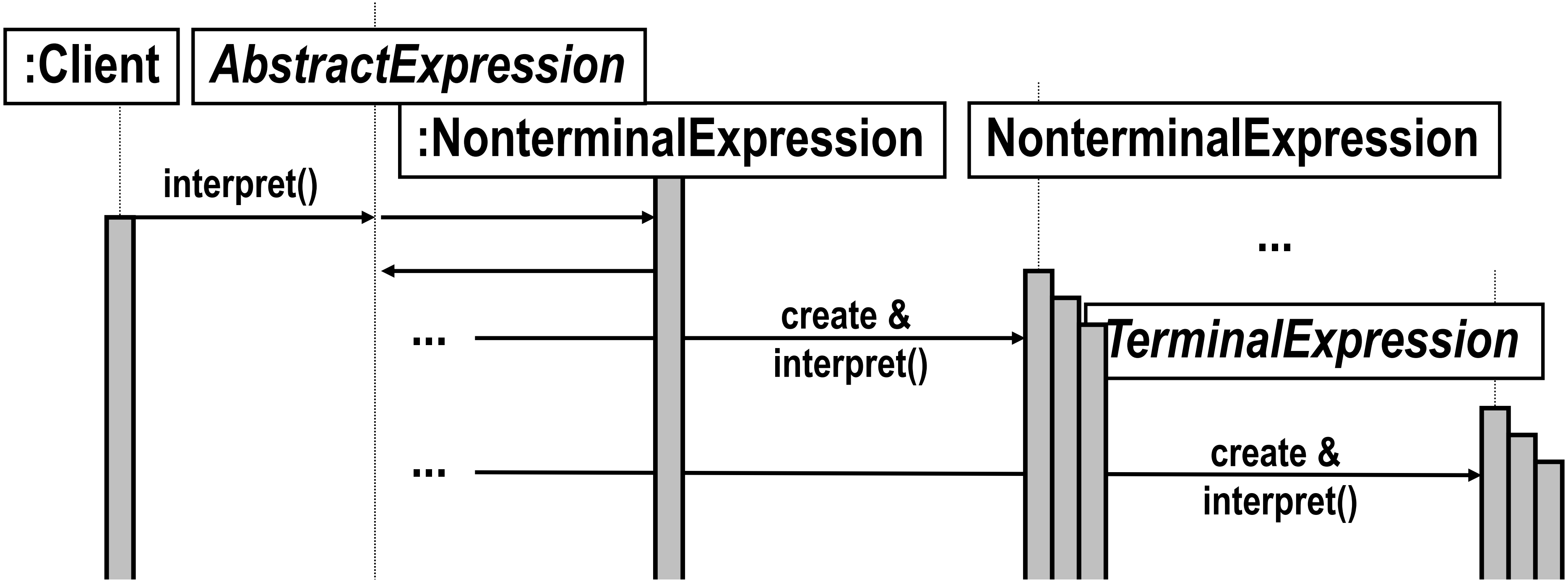
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Interpreter Design Pattern



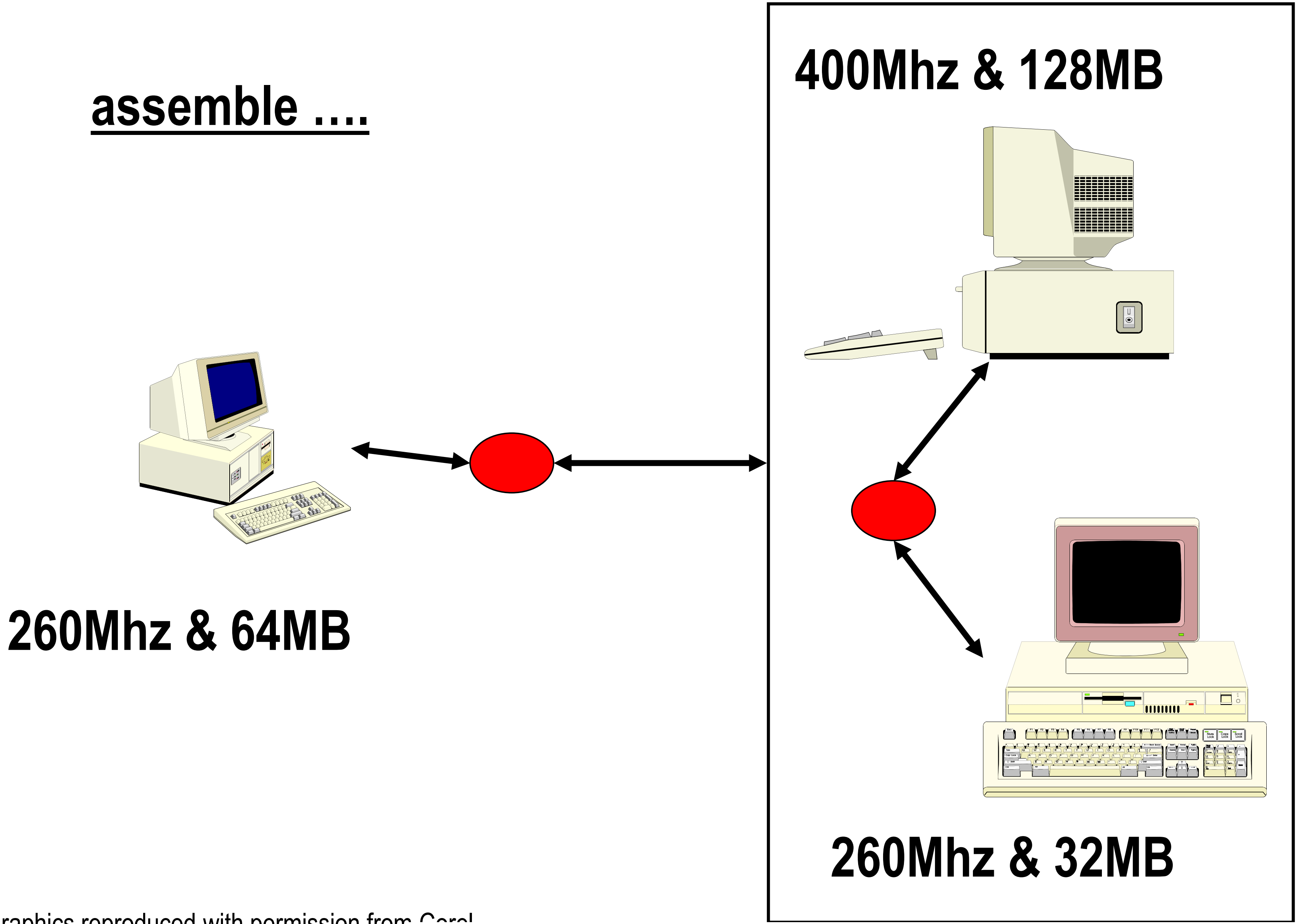
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Interpreter Sequence Diagram



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

Example Interpreter Application: Network Assembly



assemble ....

400Mhz & 128MB

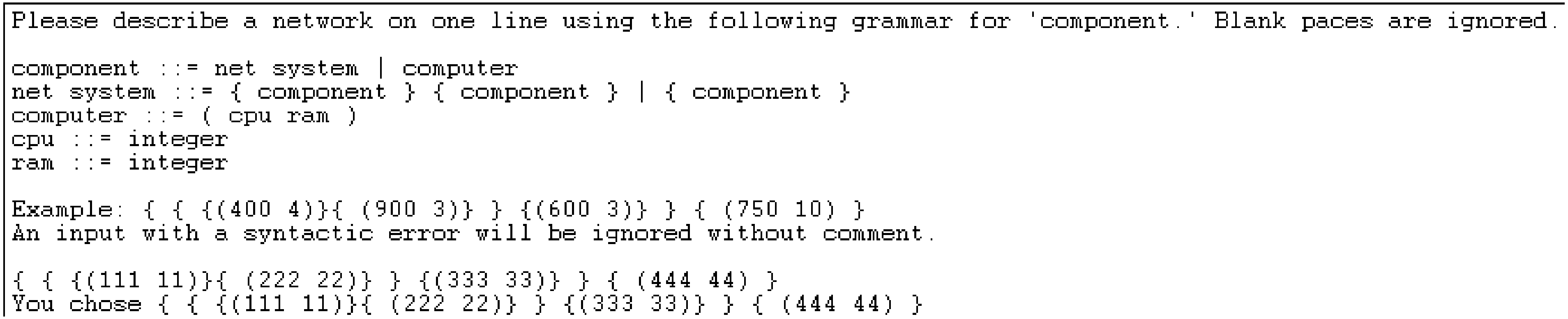
260Mhz & 64MB

260Mhz & 32MB

Graphics reproduced with permission from Corel

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

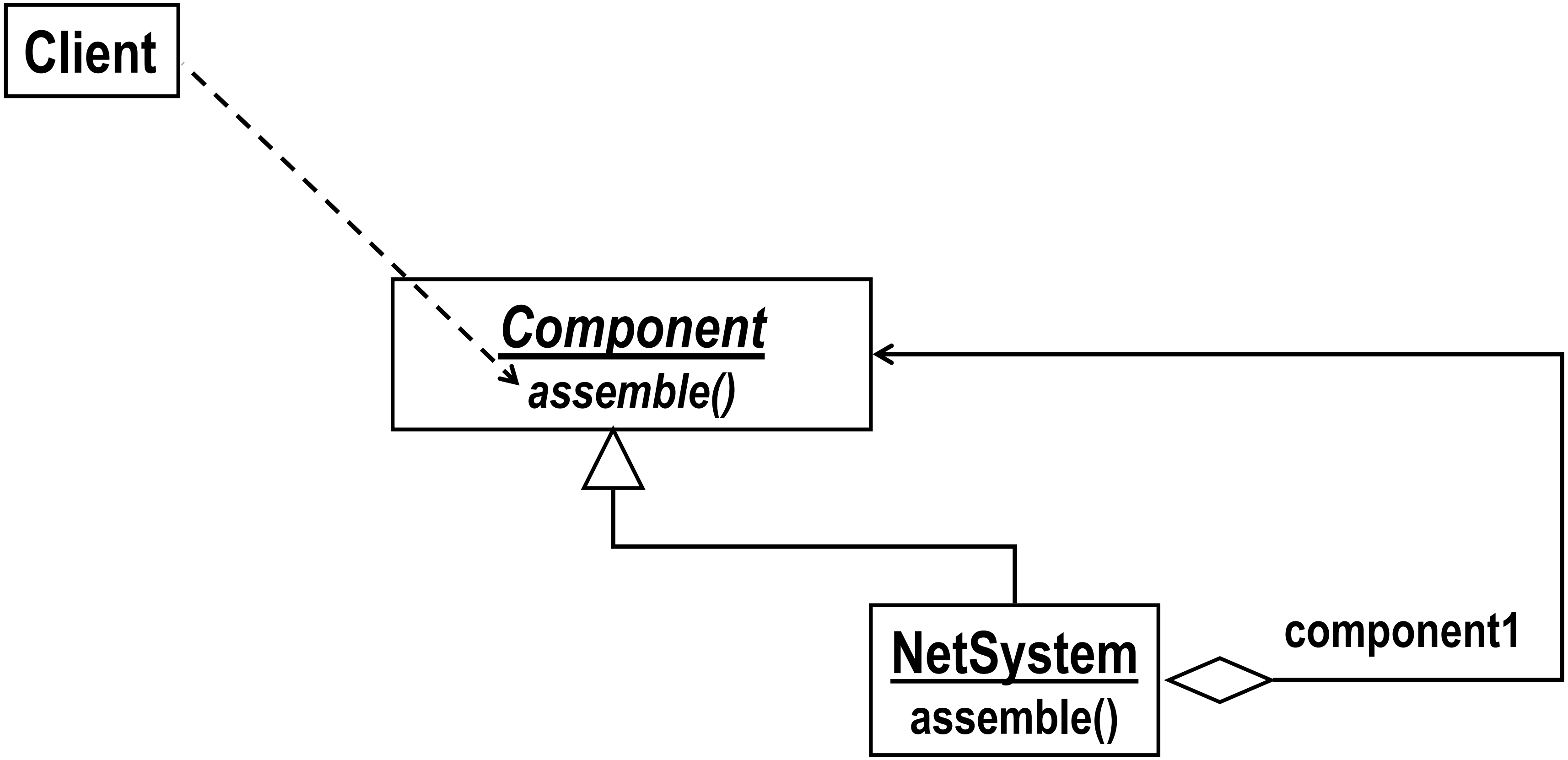
Input For Network Assembly Example



```
Please describe a network on one line using the following grammar for 'component.' Blank paces are ignored.

component ::= net system | computer
net system ::= { component } { component } | { component }
computer ::= ( cpu ram )
cpu ::= integer
ram ::= integer

Example: { { {(400 4)}{ (900 3)} } {(600 3)} } { (750 10) }
An input with a syntactic error will be ignored without comment.

{ { {(111 11)}{ (222 22)} } {(333 33)} } { (444 44) }
You chose { { {(111 11)}{ (222 22)} } {(333 33)} } { (444 44) }
```

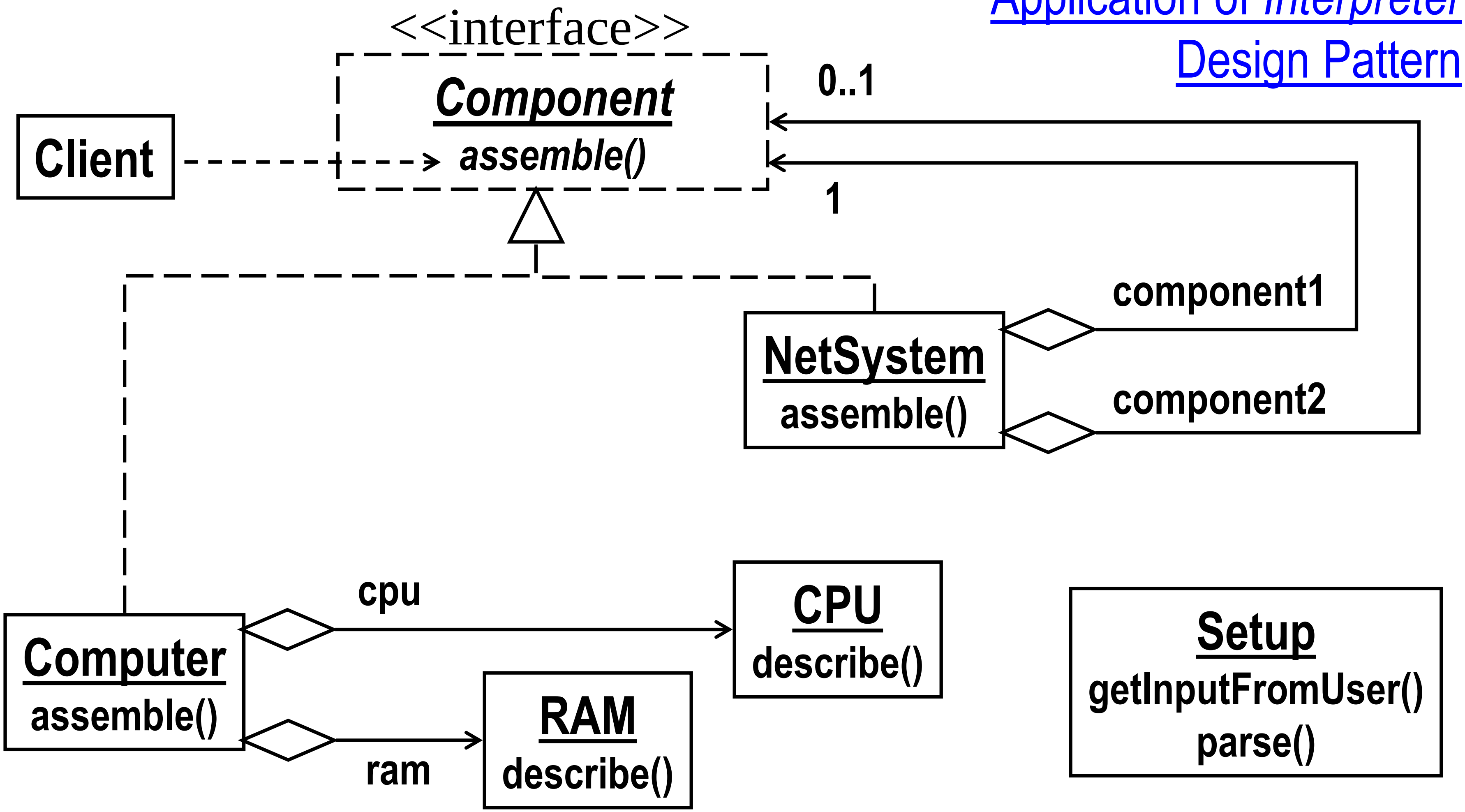
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Interpreter Design Pattern



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

Application of *Interpreter* Design Pattern



```
component ::=  net_system | computer
net_system ::= { component } { component } | { component }
computer ::= {cpu ram }
cpu ::= integer
ram ::= integer
```

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

Source code example

```
interface Component
{

}

class Computer implements Component
{

}

class NetSystem implements Component
{

}
```

```
class Client
{
. . .
Component networkOrder;
. . .
networkOrder.assemble();
. . .
}
```

```
class CPU
{

}



class RAM
{

}


class Setup
{

main()

}
```

Key Concept: → *Interpreter* Design Pattern ←

-- a form for parsing and a means of processing expressions.

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

Iterator Design Pattern

Iterator

Design Purpose (Gamma et al)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Design Pattern Summary

Encapsulate the iteration in a class pointing (in effect) to an element of the aggregate.

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

Purpose of *Iterator*

- - given a collection of objects

  e.g.,

  - the videos in a video store
  - a directory

Aggregate object

- - having specified ways to progress through them

  e.g.,

  - “list in alphabetical order”
  - “list all videos currently on loan”

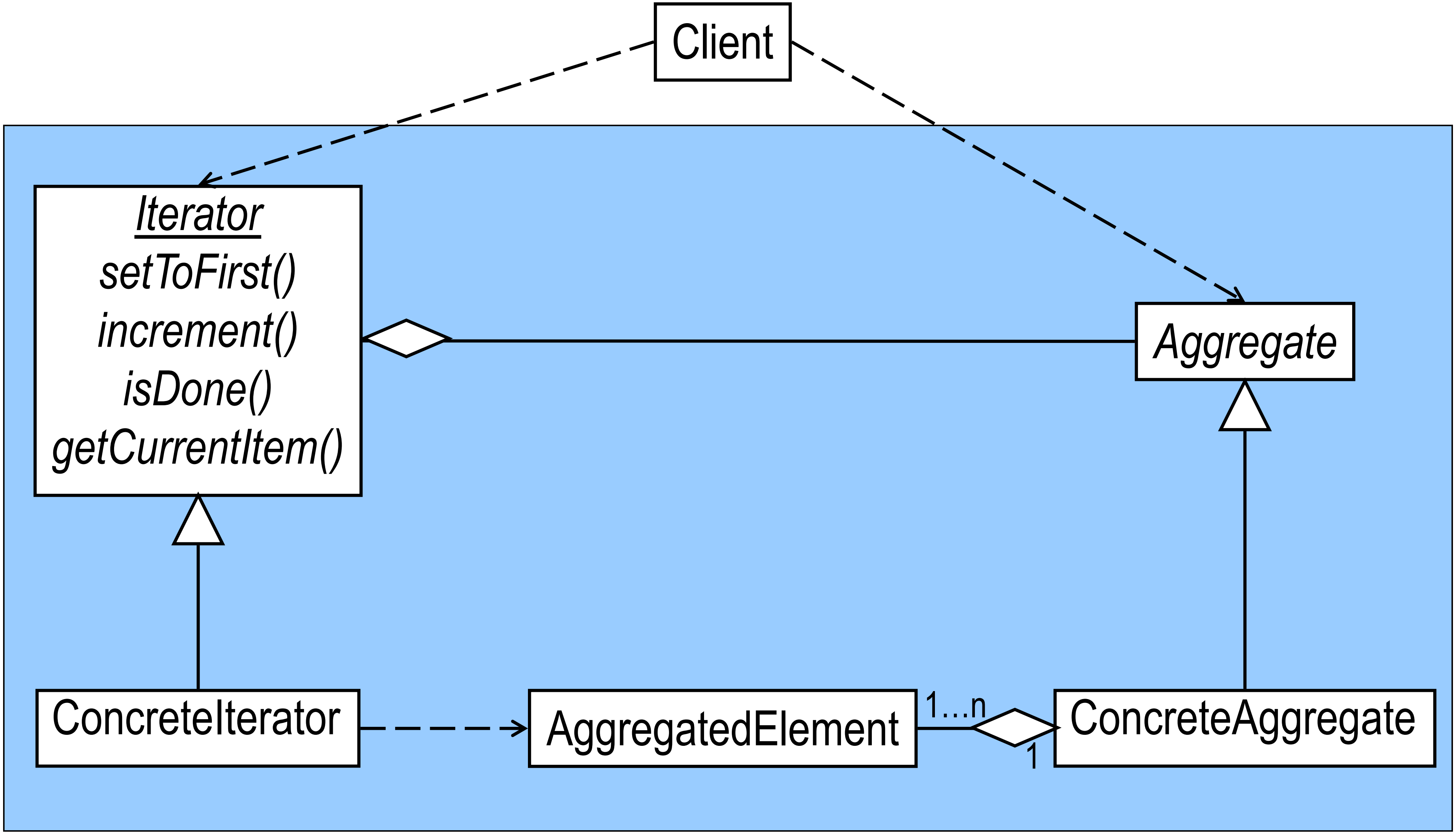
iterator2

iterator7

- ... *encapsulate* each of these ways

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

Iterator Class Model



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

Using *Iterator* Functions

```
/*
To perform desiredOperation() on elements of
the aggregate according to the iteration (order) i:
*/
for( i.setToFirst(); !i.isDone(); i.increment() )
    desiredOperation( i.getCurrentElement() );
```

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

Functions for *Iterator*

```
// Iterator "points" to first element:
void setToFirst();
// true if iterator "points" past the last element:
boolean isDone();
// Causes the iterator to point to its next element:
void increment();
// Return the element pointed to by the iterator:
C getCurrentElement();
```

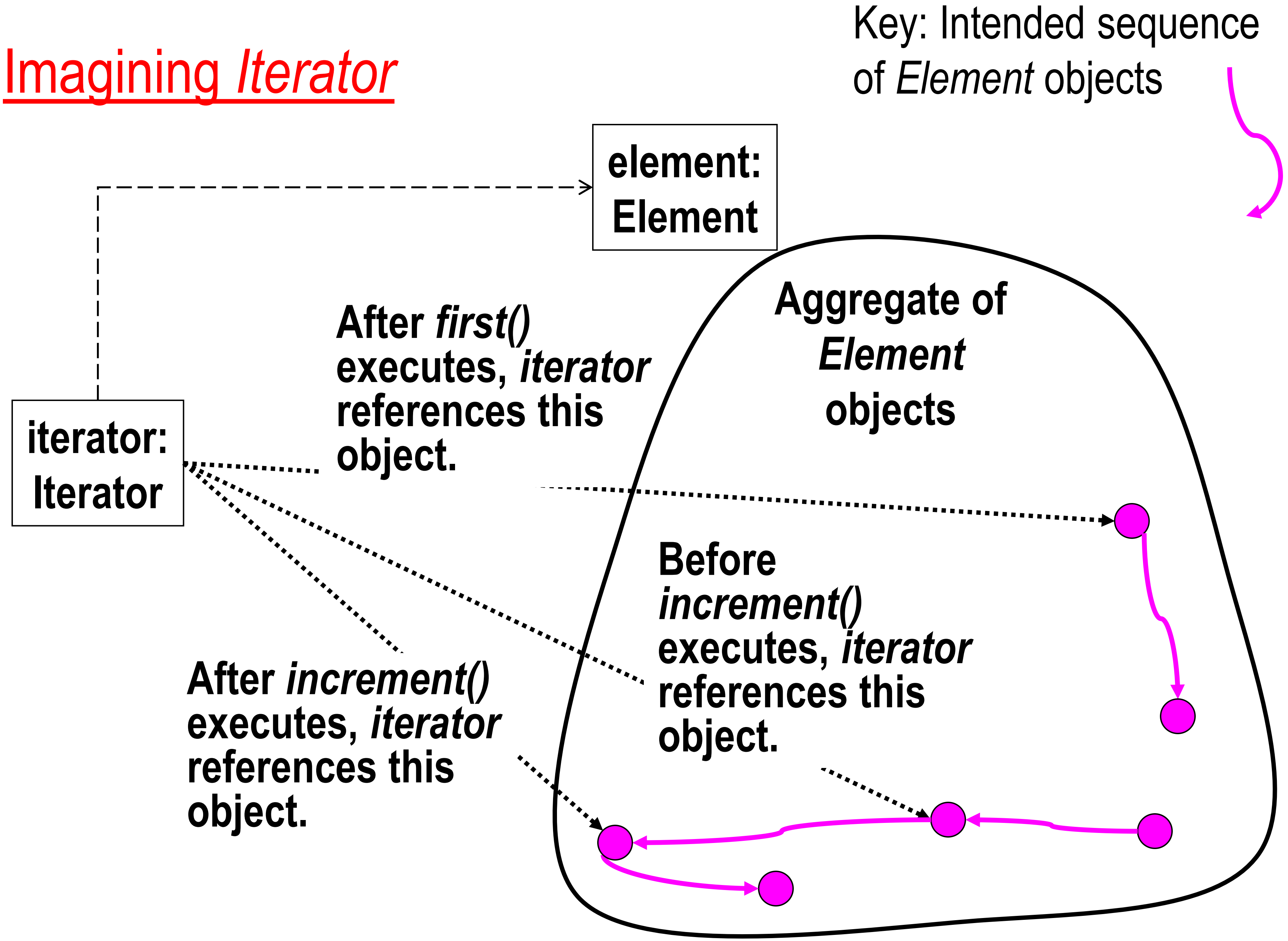
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Iterator in Arrays, *Vector*, and in General

		Iterator Operations			
		Set to beginning	Increment	Get current element	Check not done yet
The Iterator	Index (integer) *i* *on* array *myArray*	*i* = *0*	++*i*	*myArray[i]*	*i* < *myArray .length*
	Index (integer) *j* *on* Vector *myVector*	*j* = *0*	++*j*	*myVector .get(j)*	*j* < *myVector .size()*
	Iterator (object) *myIterator*	*myIterator .setToFirst()*	*myIterator .increment()*	*myIterator .getCurrent Element()*	*! myIterator .isDone()*

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

Imagining *Iterator*



Key: Intended sequence of *Element* objects

element: Element

iterator: Iterator

Aggregate of *Element* objects

After *first()* executes, *iterator* references this object.

Before *increment()* executes, *iterator* references this object.

After *increment()* executes, *iterator* references this object.

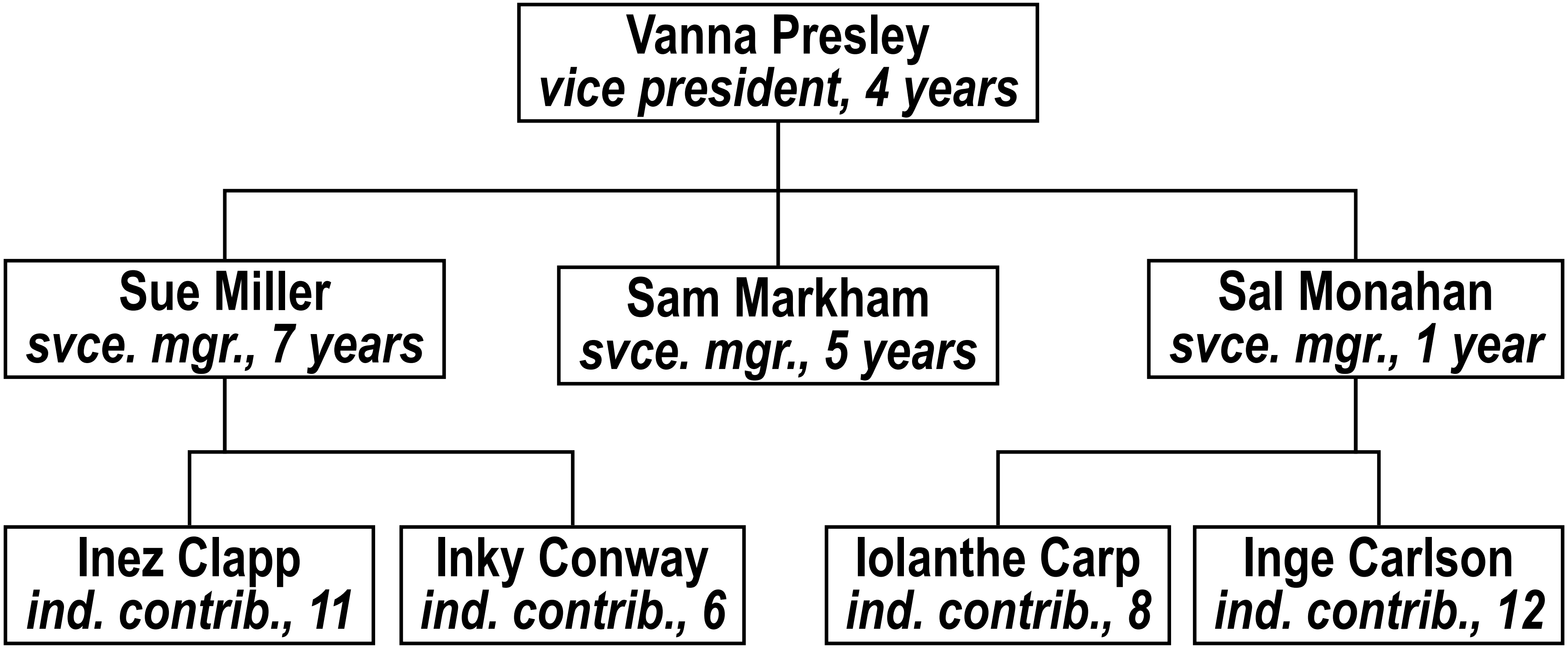
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Iterator Example Setup Code

```
// Suppose that we have iterators for forward and
// backward order: we can re-use print_employees()

List employees = new List();
ForwardListIterator forward                  // to go from front to back
  = new ForwardListIterator ( employees );
ReverseListIterator backward                 // to go from back to front
   = new ReverseListIterator ( employees );

client.print_employees( forward );           // print from front to back
client.print_employees( backward );          // print from back to front
```

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

An Organizational Chart Example



Vanna Presley
vice president, 4 years

Sue Miller
svce. mgr., 7 years

Sam Markham
svce. mgr., 5 years

Sal Monahan
svce. mgr., 1 year

Inez Clapp
ind. contrib., 11

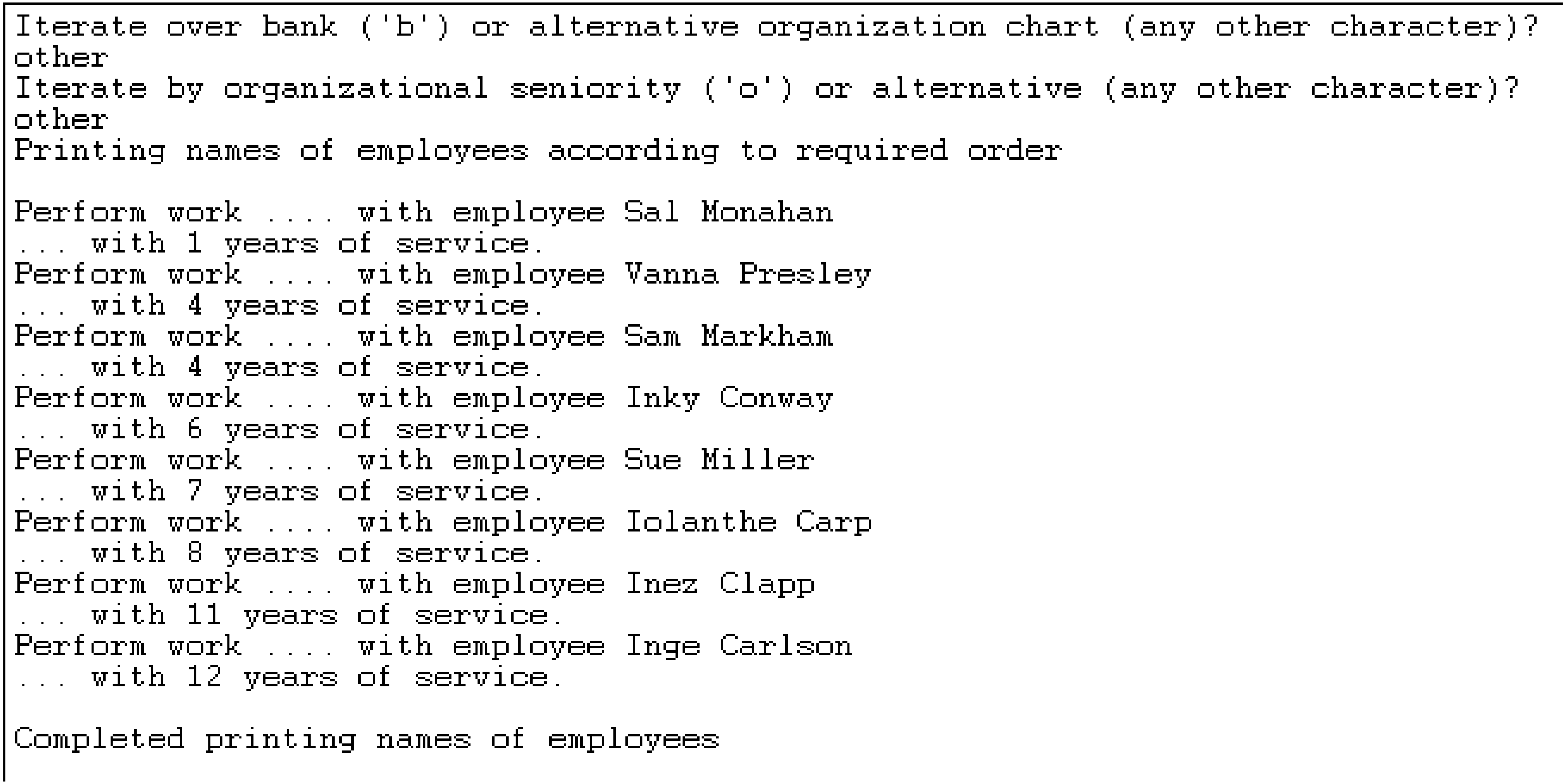
Inky Conway
ind. contrib., 6

Iolanthe Carp
ind. contrib., 8

Inge Carlson
ind. contrib., 12

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

Iterating by Years of Service Over an Organization Chart



```
Iterate over bank ('b') or alternative organization chart (any other character)?
other
Iterate by organizational seniority ('o') or alternative (any other character)?
other
Printing names of employees according to required order

Perform work .... with employee Sal Monahan
... with 1 years of service.
Perform work .... with employee Vanna Presley
... with 4 years of service.
Perform work .... with employee Sam Markham
... with 4 years of service.
Perform work .... with employee Inky Conway
... with 6 years of service.
Perform work .... with employee Sue Miller
... with 7 years of service.
Perform work .... with employee Iolanthe Carp
... with 8 years of service.
Perform work .... with employee Inez Clapp
... with 11 years of service.
Perform work .... with employee Inge Carlson
... with 12 years of service.

Completed printing names of employees
```

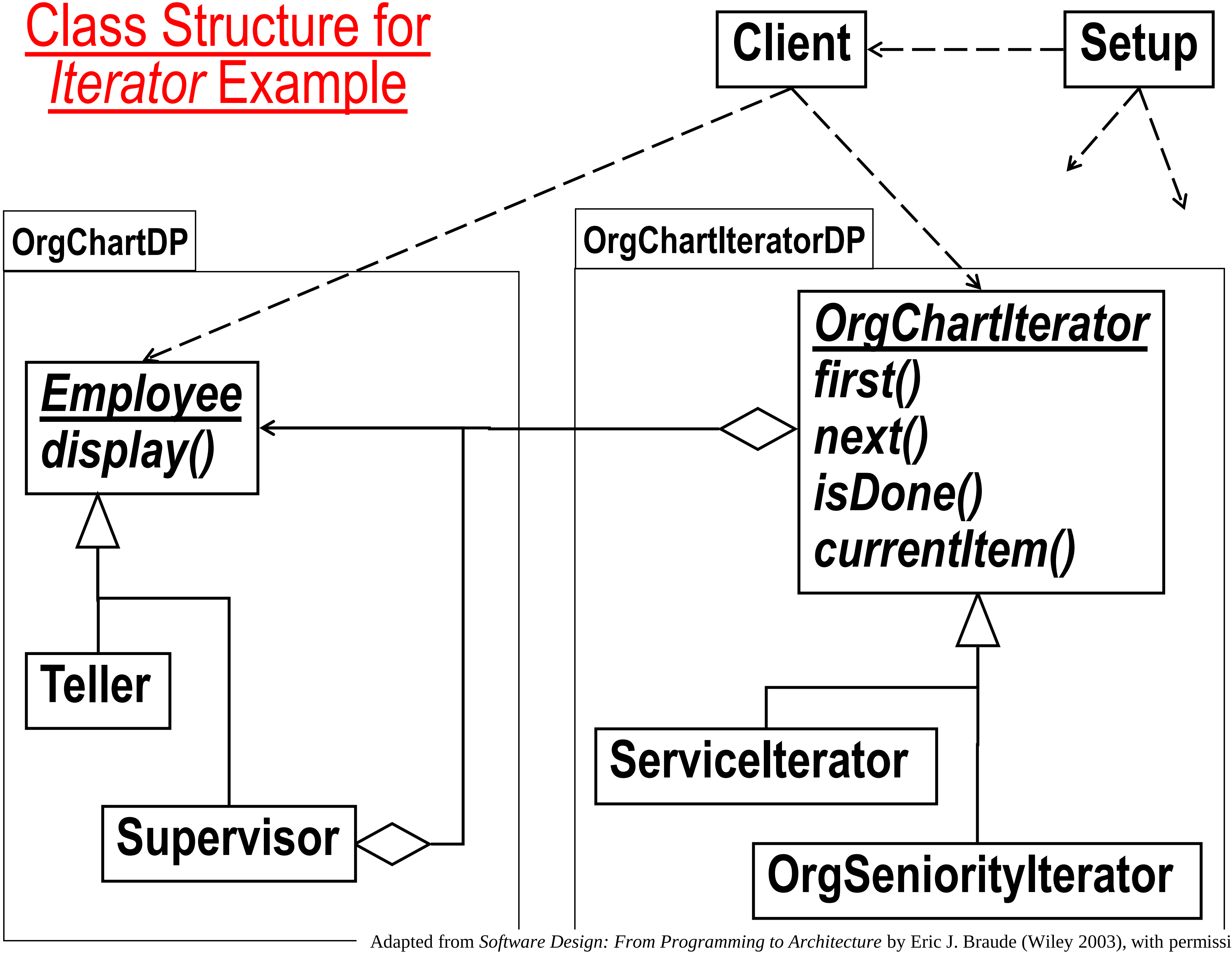
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Flexibility*, *Correctness* ←

Separate the "visiting" procedure from the processing of individual employees.

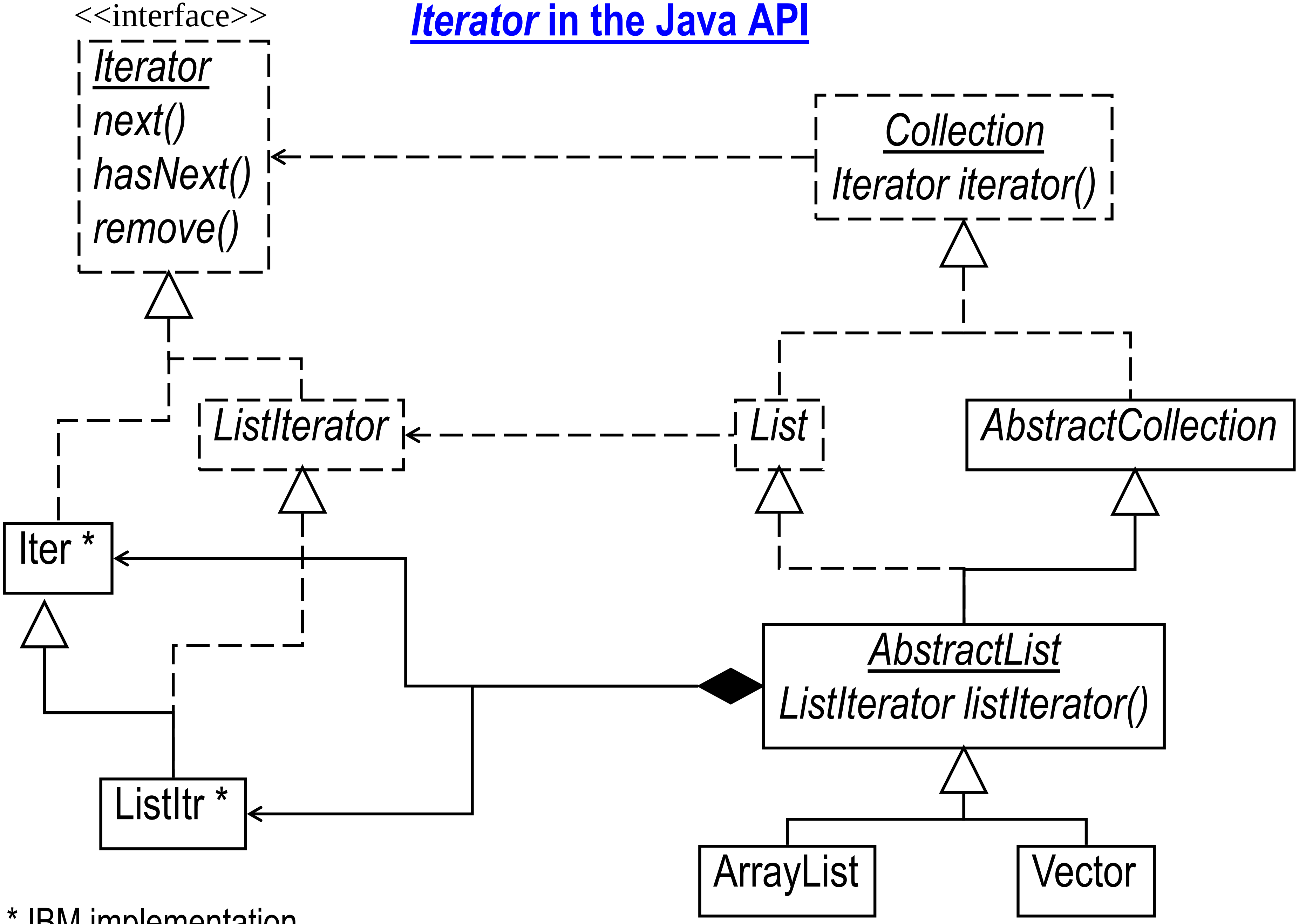
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Class Structure for *Iterator* Example



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

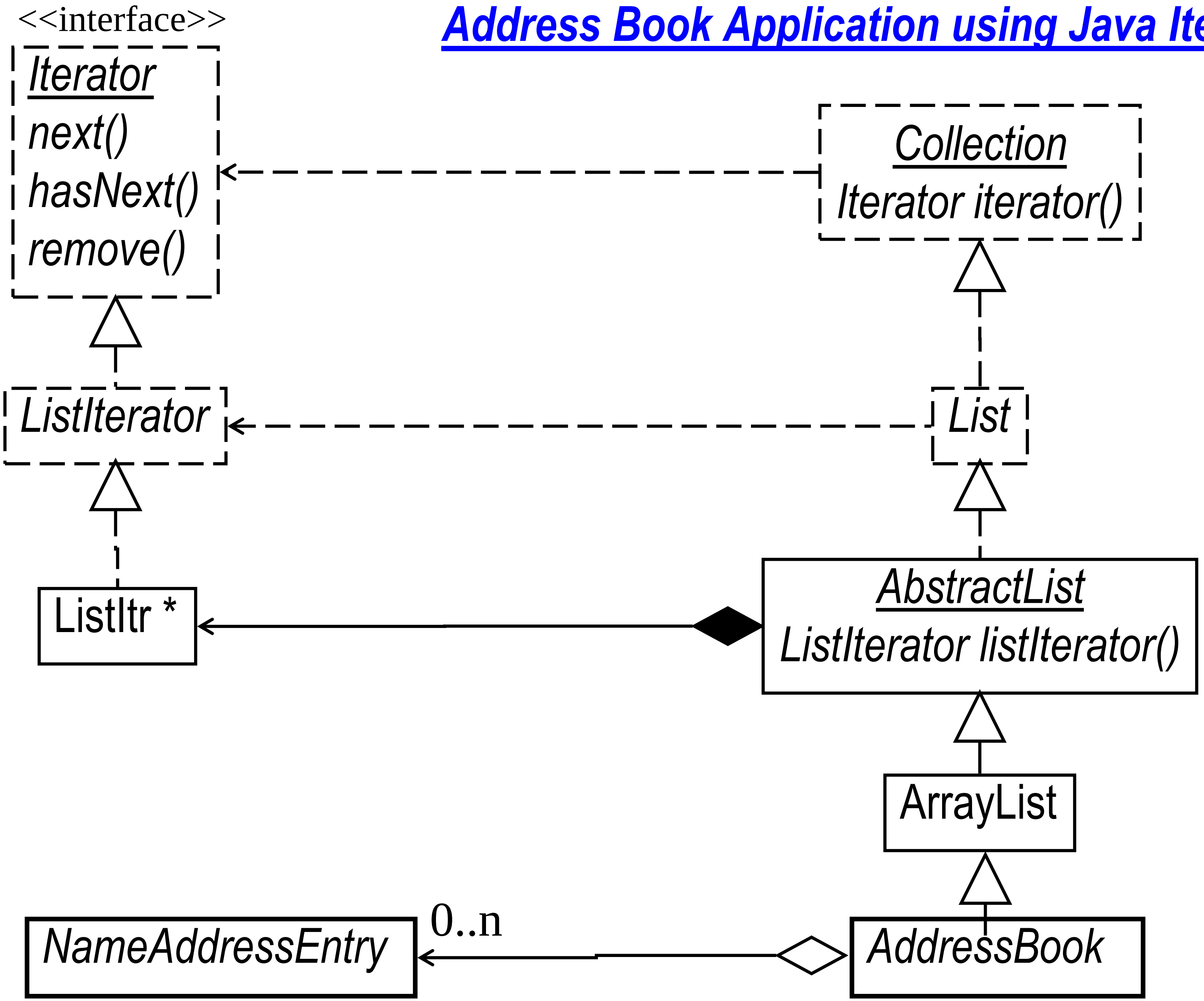
Iterator in the Java API



* IBM implementation

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

Address Book Application using Java Iterator



* IBM implementation

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

Key Concept: → *Iterator* Design Pattern ←

-- to access the elements of a collection.

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

Mediator Design Pattern

Mediator

Design Purpose

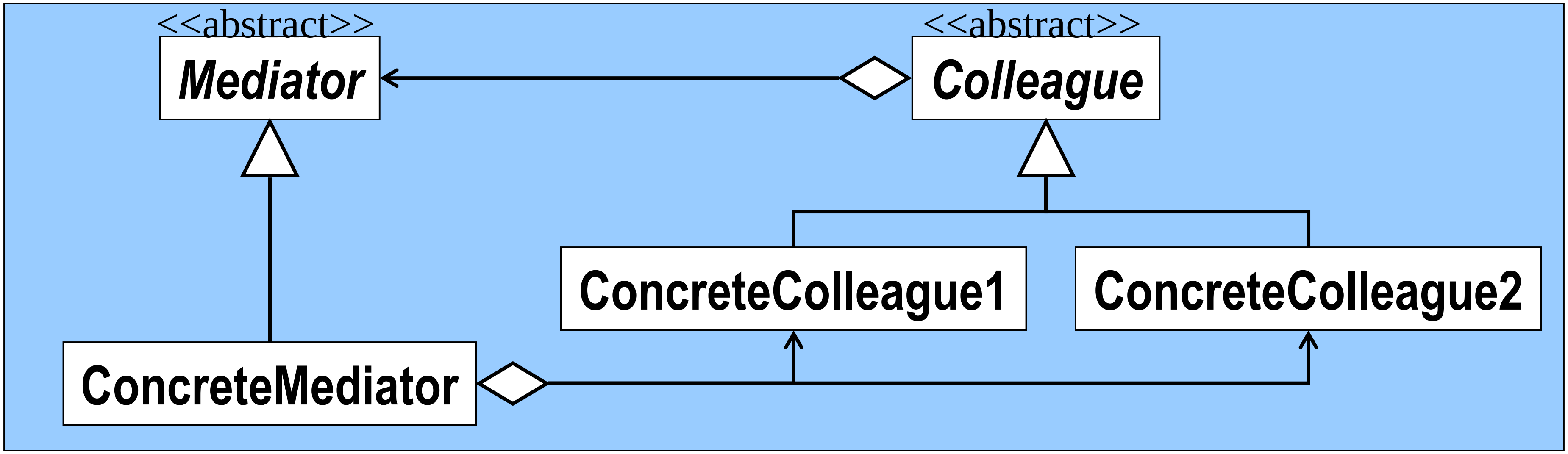
Avoid references between dependent objects

Design Pattern Summary

Capture mutual behavior in a separate class

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

The *Mediator* Class Model



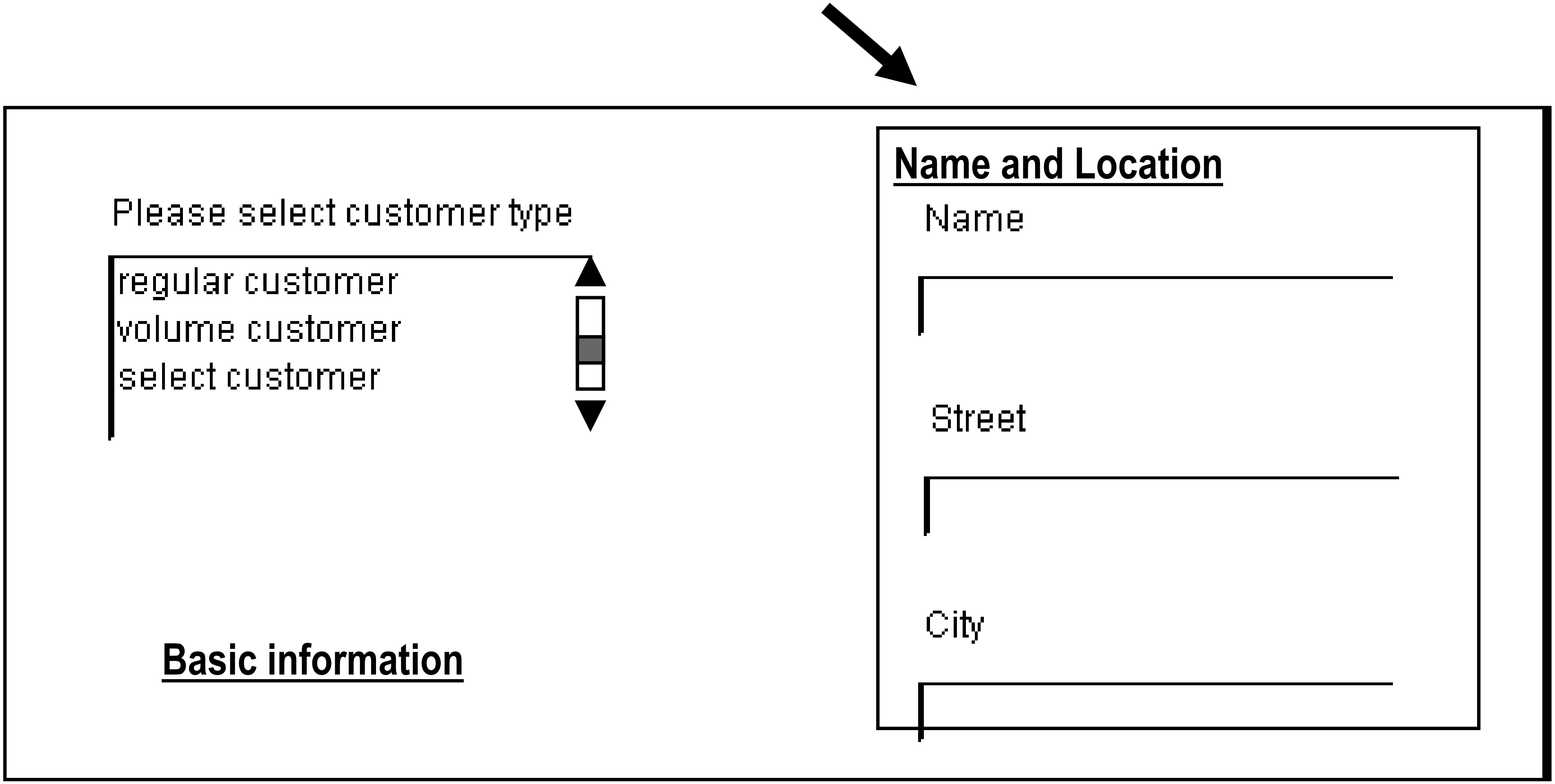
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Reusability* and *Robustness* ←

Avoid hard-coded dependencies among the game's GUI classes, enabling their use in other contexts.

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

Solicitation of Customer Information (*1 of 2*)



Please select customer type

regular customer
volume customer
select customer

Basic information

Name and Location

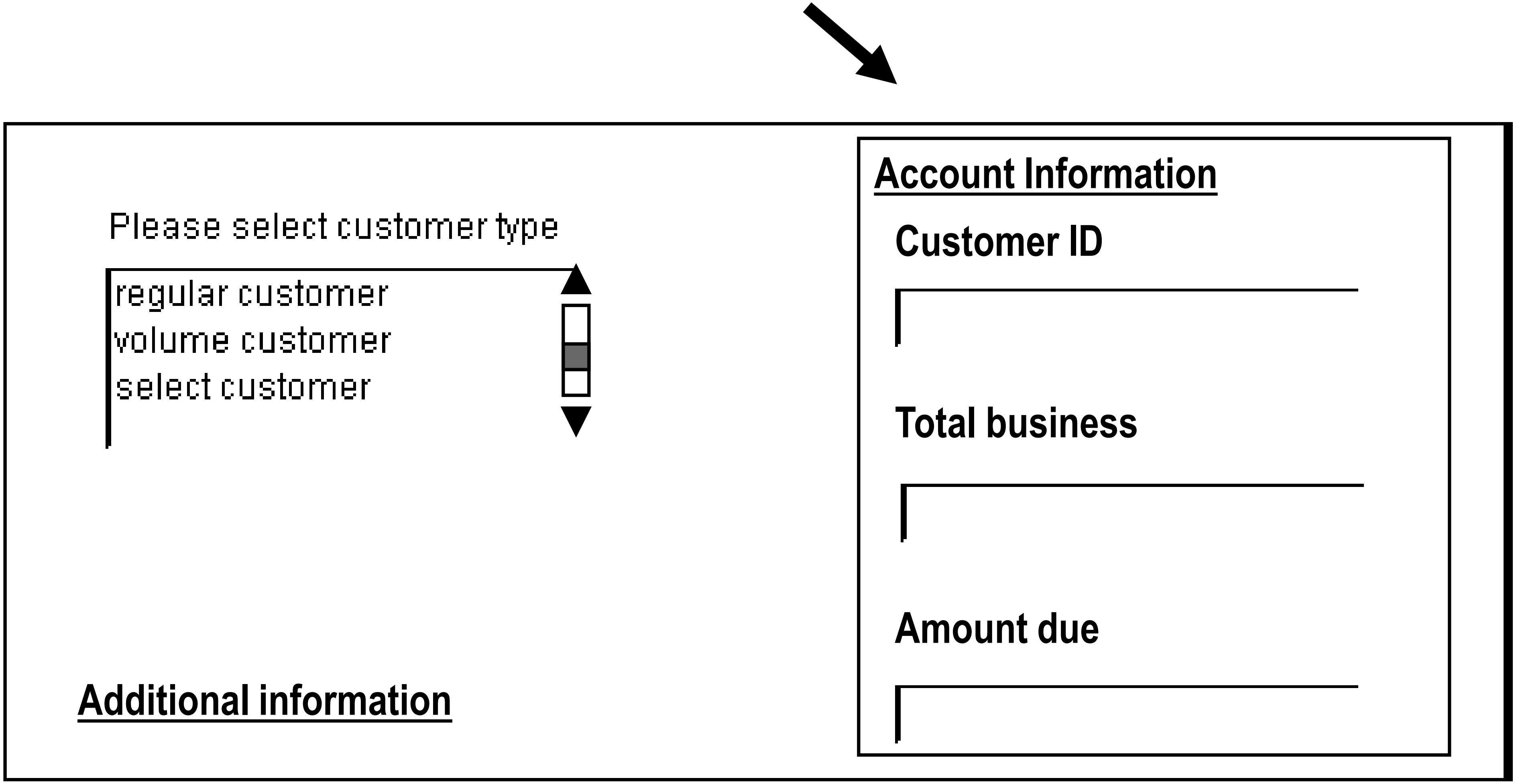
Name

Street

City

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

Solicitation of Customer Information (2 *of 2)*



Please select customer type

regular customer

volume customer

select customer

Additional information

Account Information

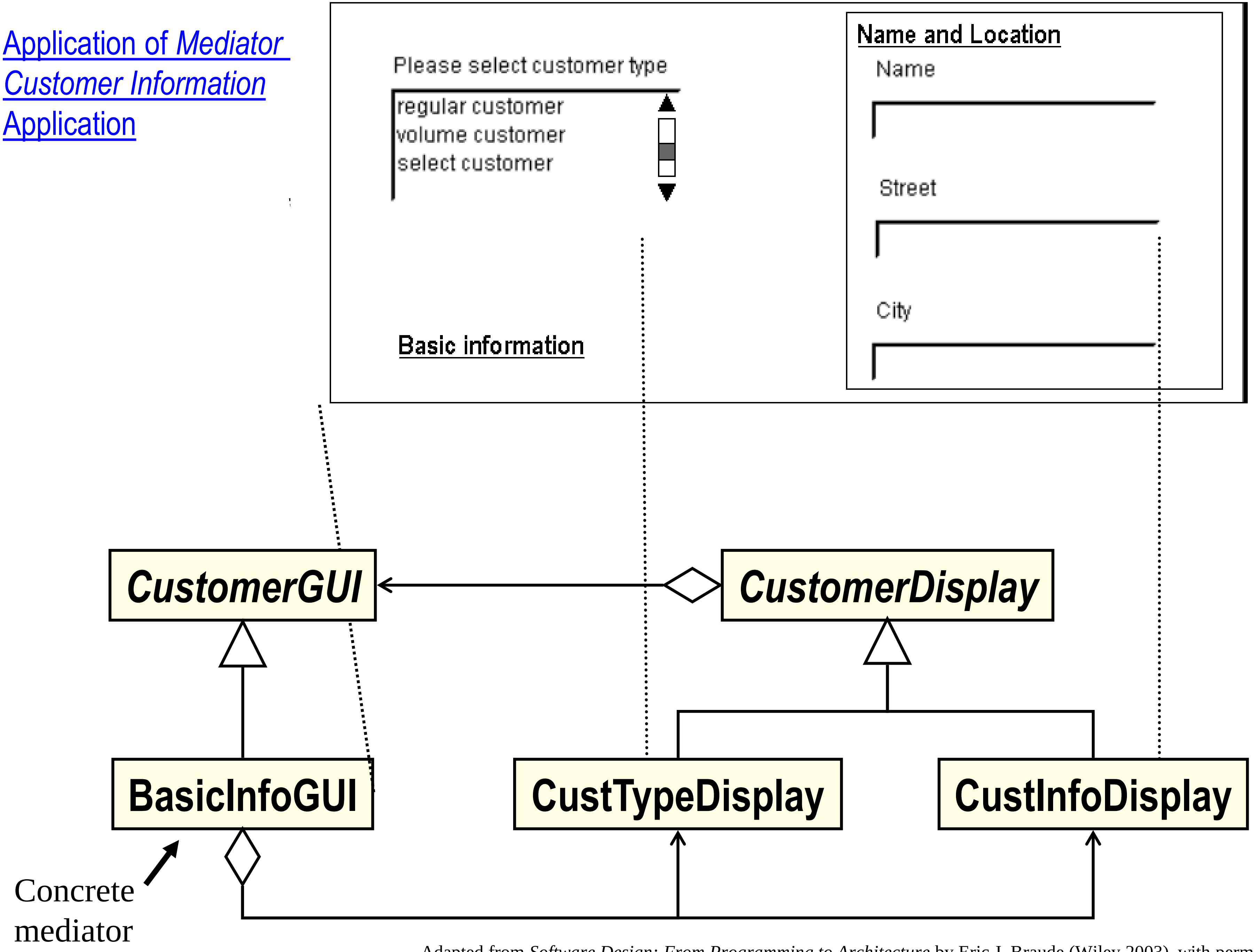
Customer ID

Total business

Amount due

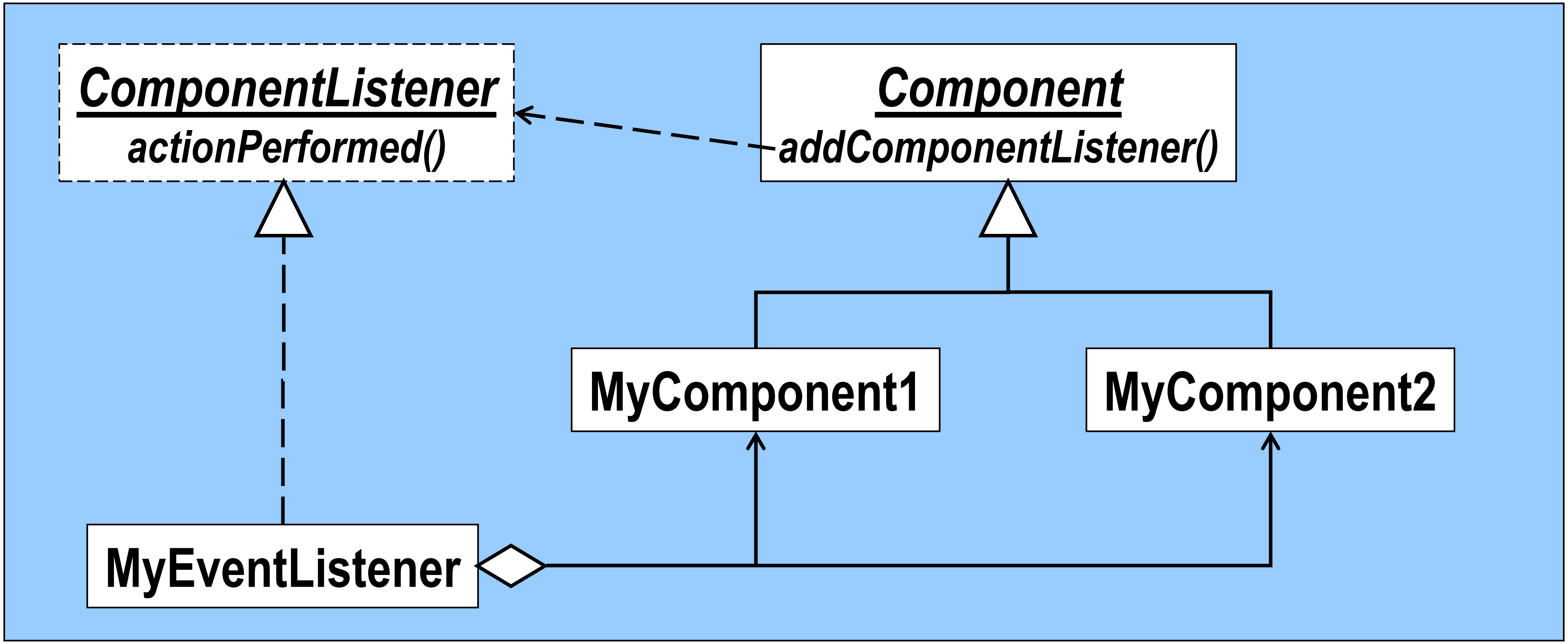
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Application of *Mediator* *Customer Information* Application



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

The *Mediator* Class Model in the Java API



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

Key Concept: → ***Mediator*** Design Pattern ←

-- to capture mutual behavior without direct dependency.

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

Observer Design Pattern

Observer

Design Purpose

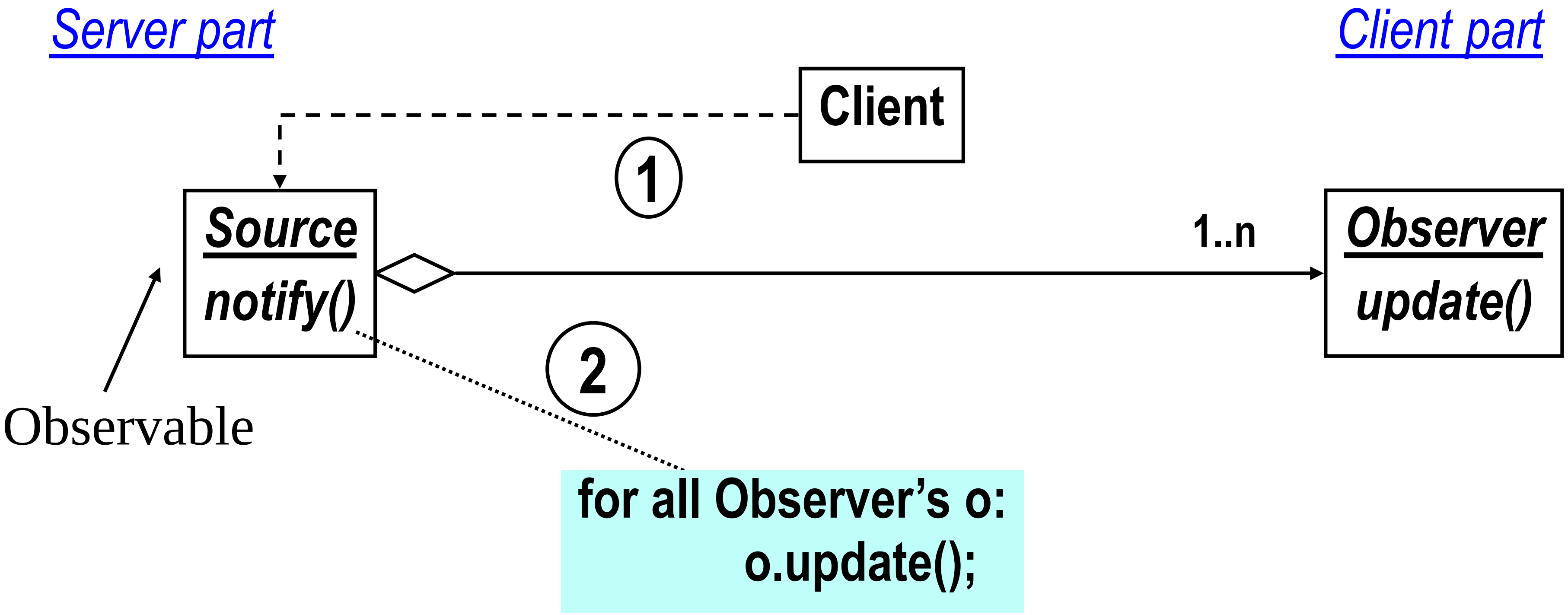
Arrange for a set of objects to be affected by a single object

Design Pattern Summary

The single object aggregates the set, calling a method with a fixed name on each member

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

Observer Design Pattern

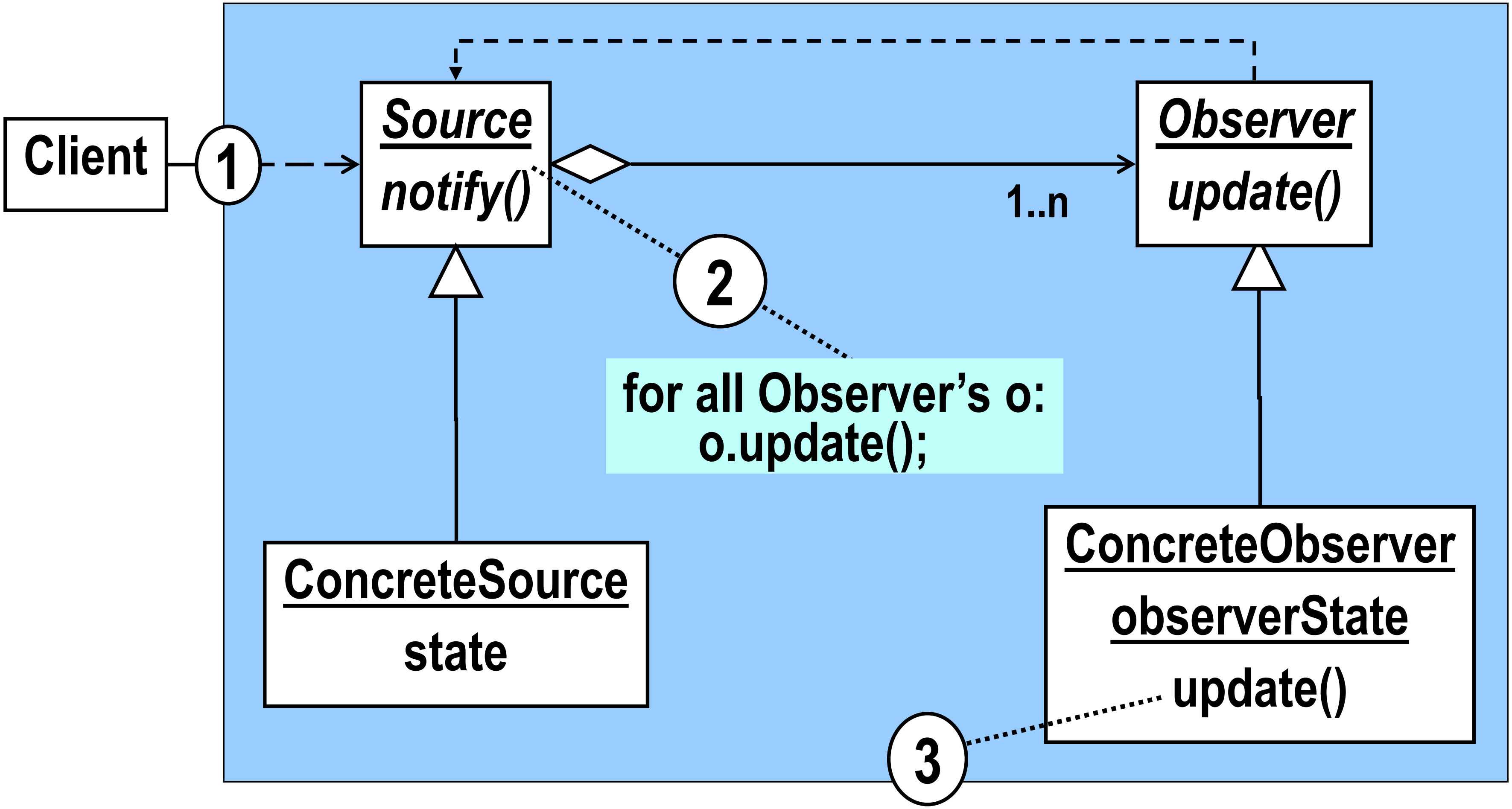


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Observer Design Pattern

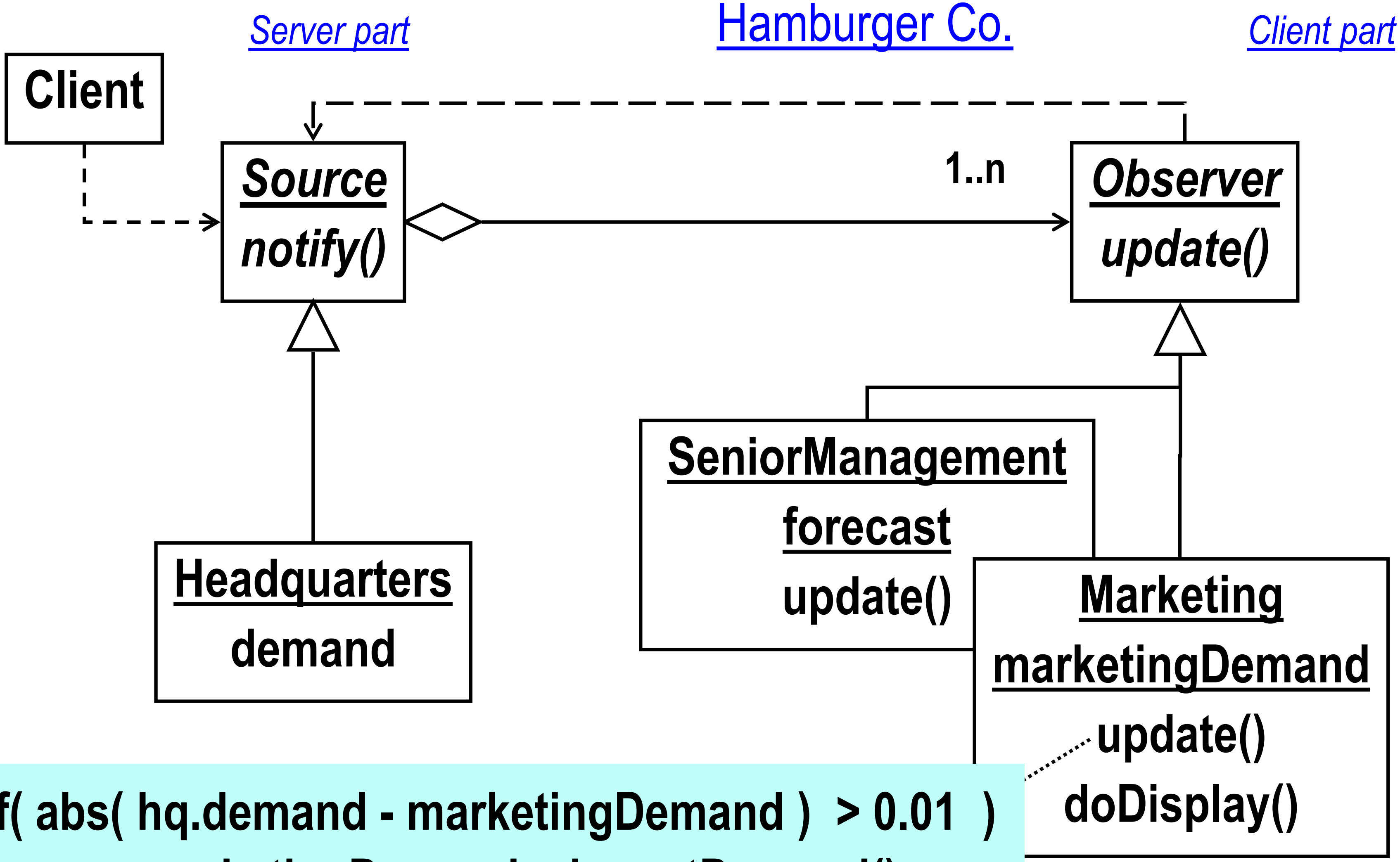
Server part

Client part



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

Observer Applied to International Hamburger Co.



```
if( abs( hq.demand - marketingDemand )  > 0.01  )
{         marketingDemand = hq.getDemand();
          doDisplay();
}
```

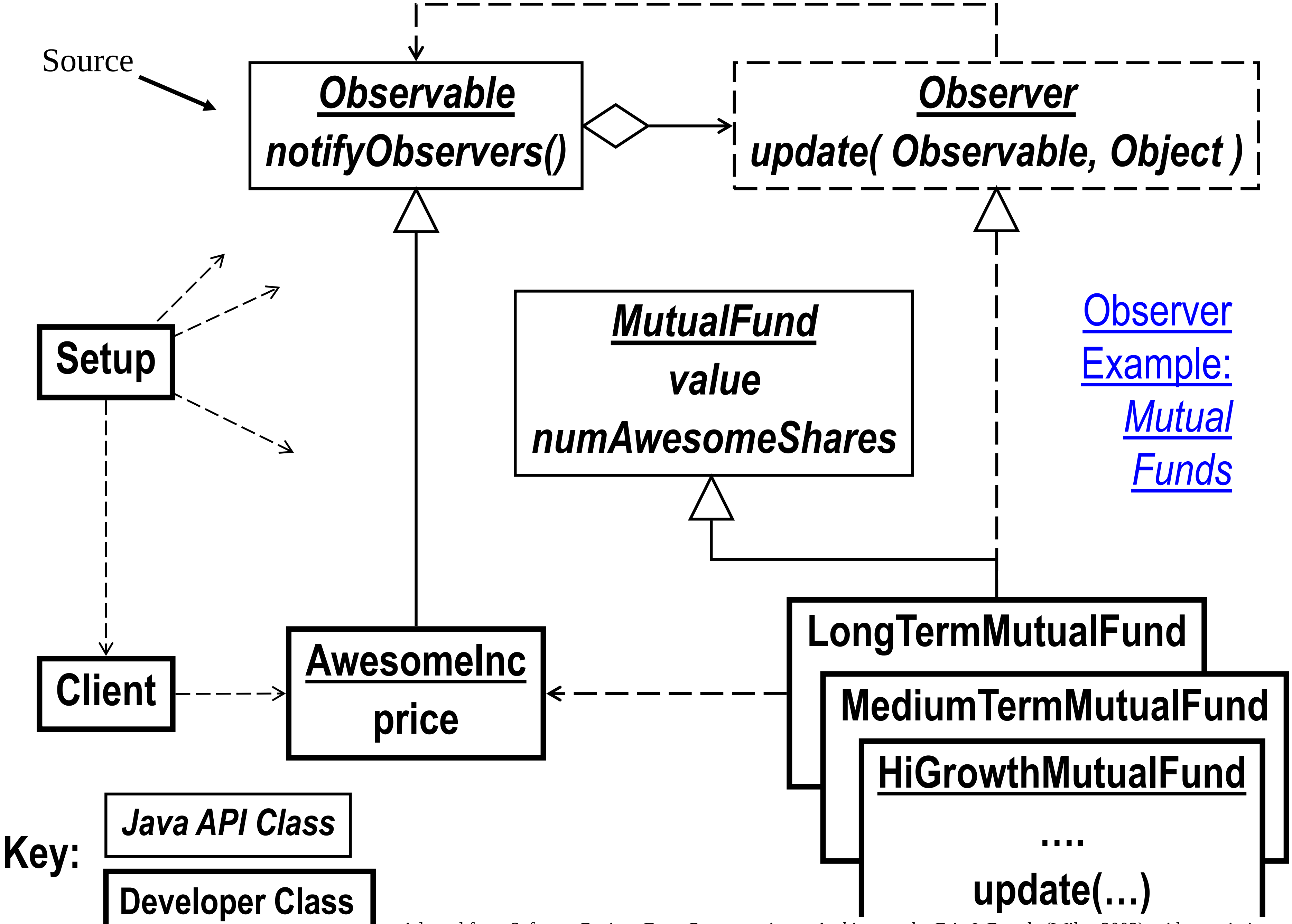
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Flexibility* ←

Allow mutual funds objects to easily acquire or divest of stocks.

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

Observer Example: *Mutual Funds*



Source

Observable

notifyObservers()

Observer

update(Observable, Object)

MutualFund

value

numAwesomeShares

Setup

Client

AwesomeInc

price

LongTermMutualFund

MediumTermMutualFund

HiGrowthMutualFund

….

update(…)

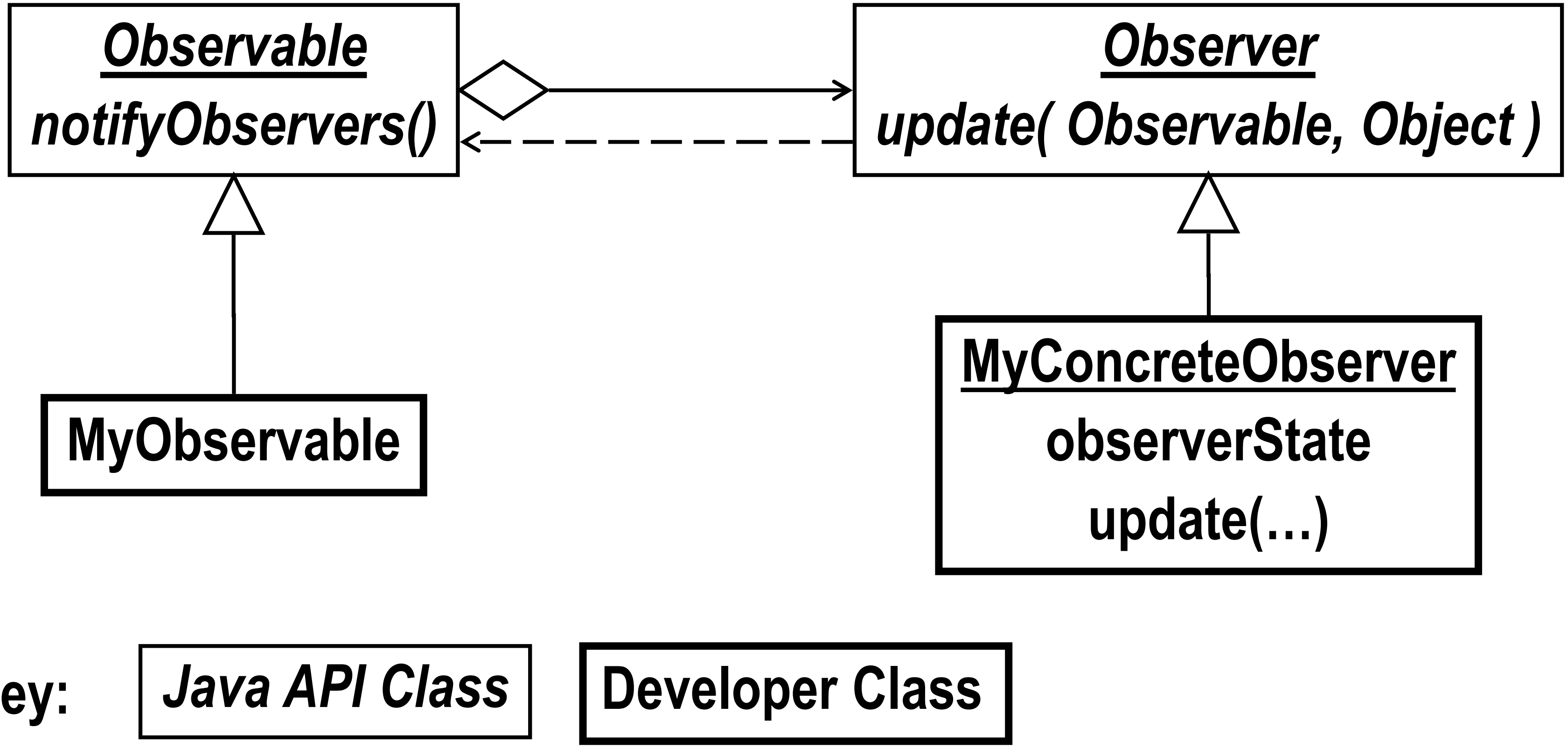
Key:

Java API Class

Developer Class

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

Observer in the Java API



Observable

notifyObservers()

Observer

update(Observable, Object)

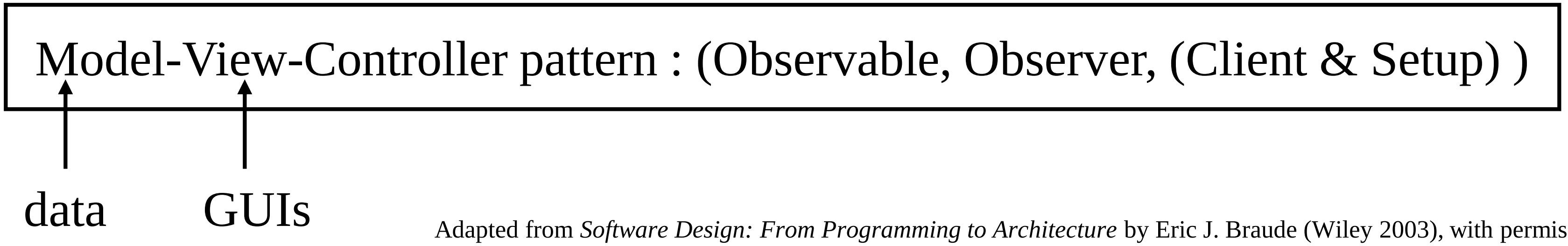
MyObservable

MyConcreteObserver

observerState

update(…)

Key: *Java API Class* **Developer Class**



Model-View-Controller pattern : (Observable, Observer, (Client & Setup))

data

GUIs

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

Key Concept: → *Observer* Design Pattern ←

-- to keep a set of objects up to date with the state of a designated object.

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

State Design Pattern

State

Design Purpose

Cause an object to behave in a manner determined by its state

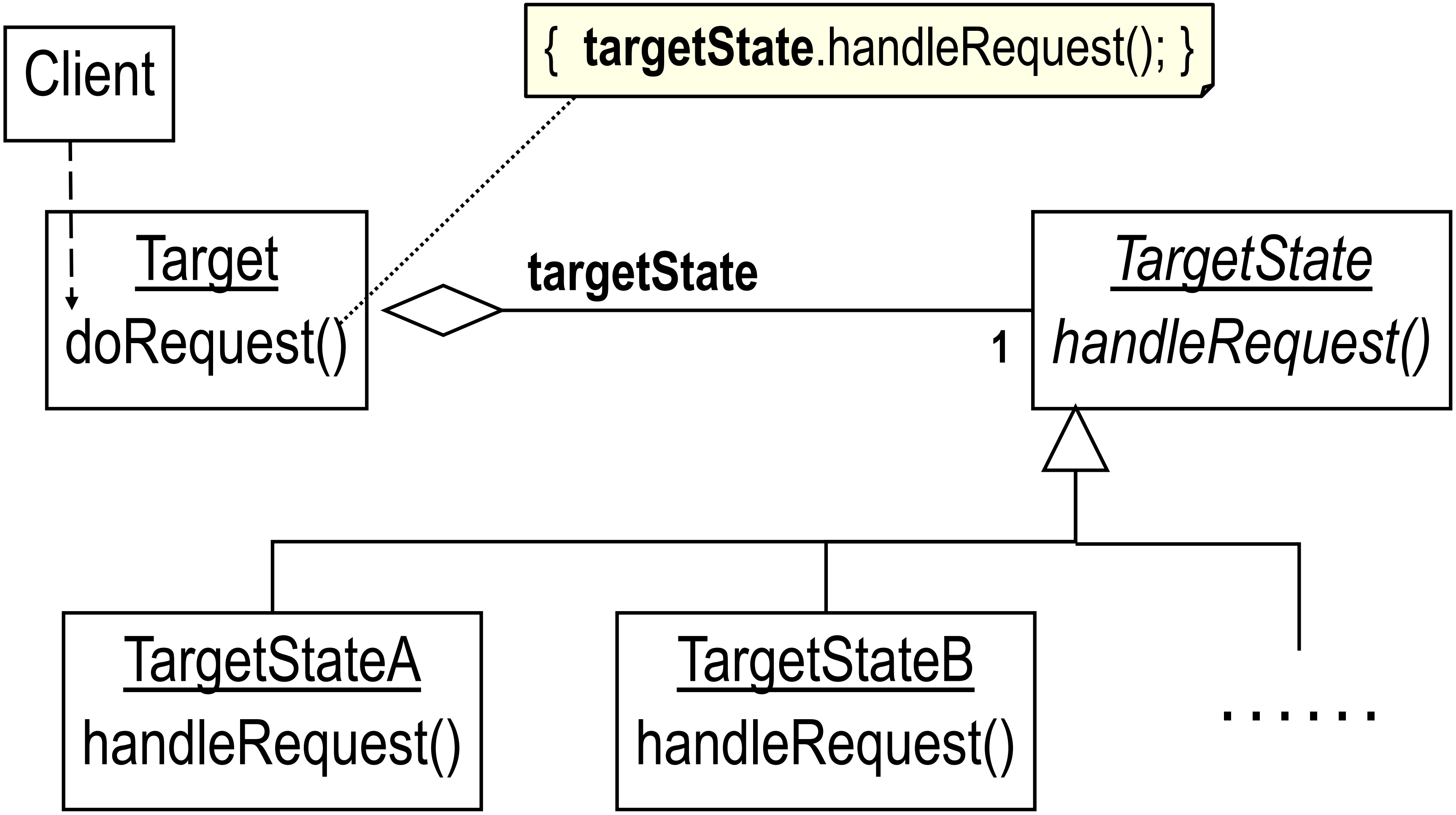
Design Pattern Summary

Aggregate a *state* object and delegate behavior to it

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

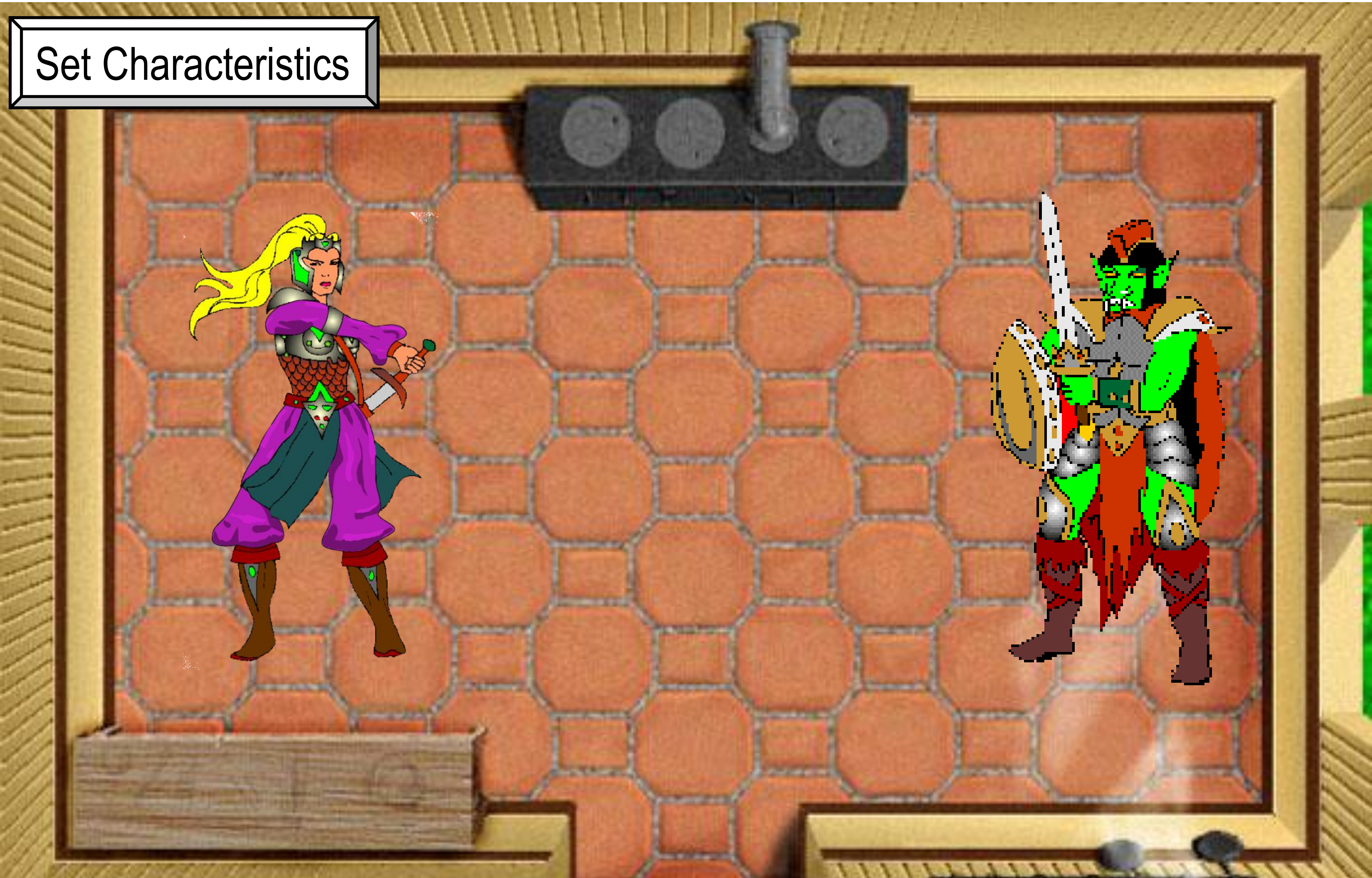
State Design Pattern Structure:

doRequest() behaves according to state of *Target*



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

GUI For a Role-Playing Video Game



Courtesy Tom VanCourt and C

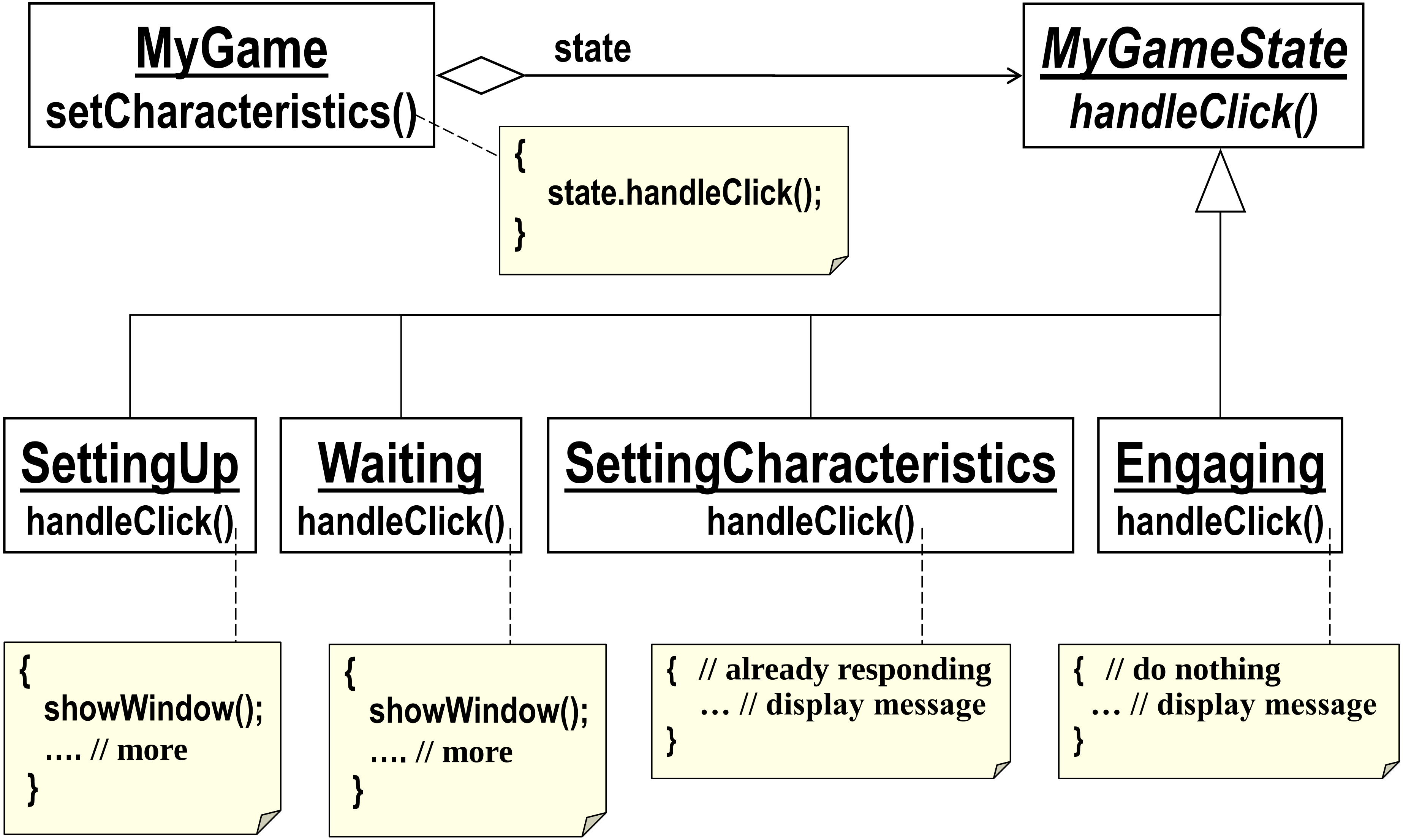
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Correctness* and *Reusability* ←

Separate the generic code for handling button clicks from the actions which depend on the game's status at the time.

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

State Design Pattern Applied to Role-Playing Game



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

Key Concept: → *State* Design Pattern ←

-- to cause a object’s functions to behave according to the state it’s in.

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

Chain of Responsibility Design Pattern

Chain of Responsibility

Design Purpose

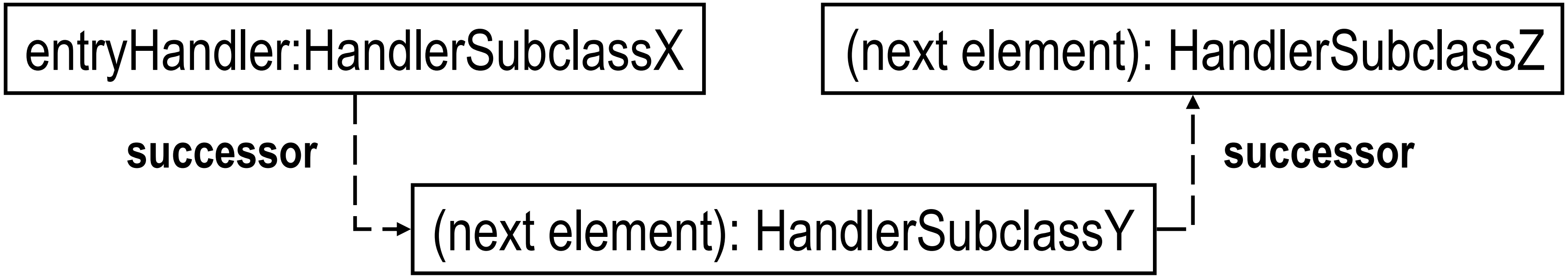
Allow a set of objects to service a request. It presents clients with a simple interface

Design Pattern Summary

Link the objects in a chain via aggregation, allowing each to perform some of the responsibility, passing the request along

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

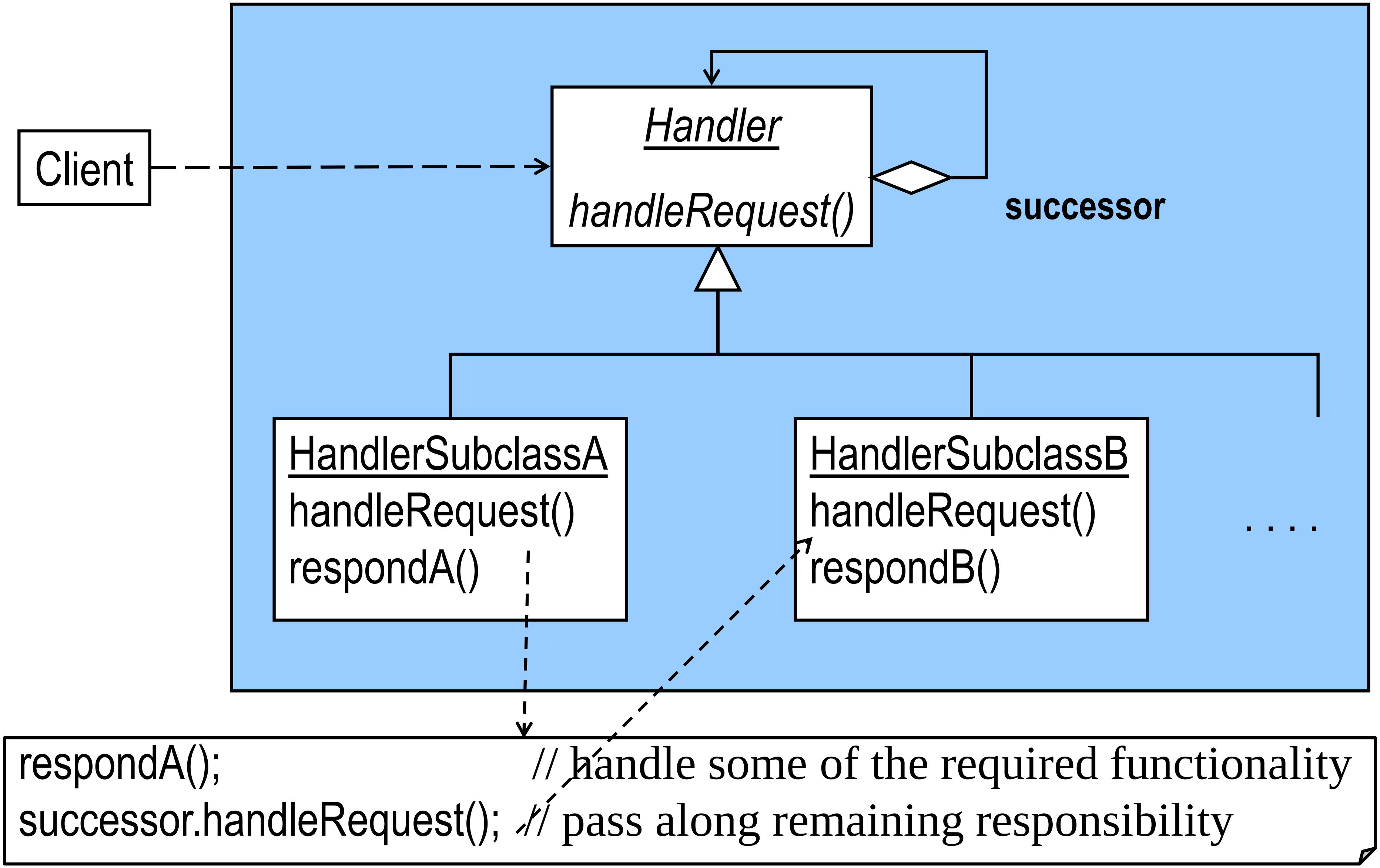
Chain of Responsibility: Object Model



entryHandler:HandlerSubclassX
(next element): HandlerSubclassZ
successor
successor
(next element): HandlerSubclassY

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

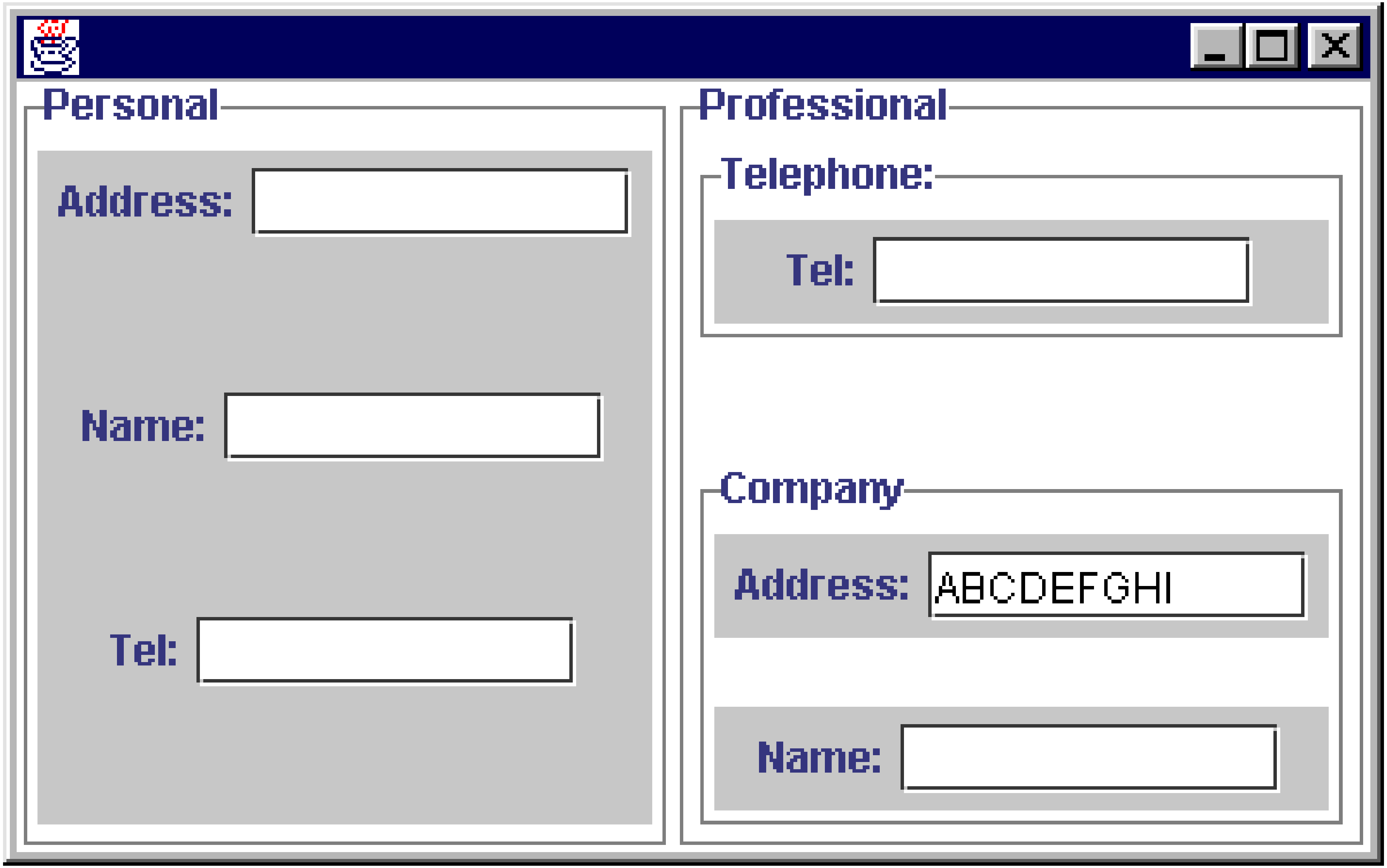
Chain of Responsibility Class Model



```
respondA();                    // handle some of the required functionality
successor.handleRequest();     // pass along remaining responsibility
```

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

GUI For *Customer Information* Application



Personal

Address:

Name:

Tel:

Professional

Telephone:

Tel:

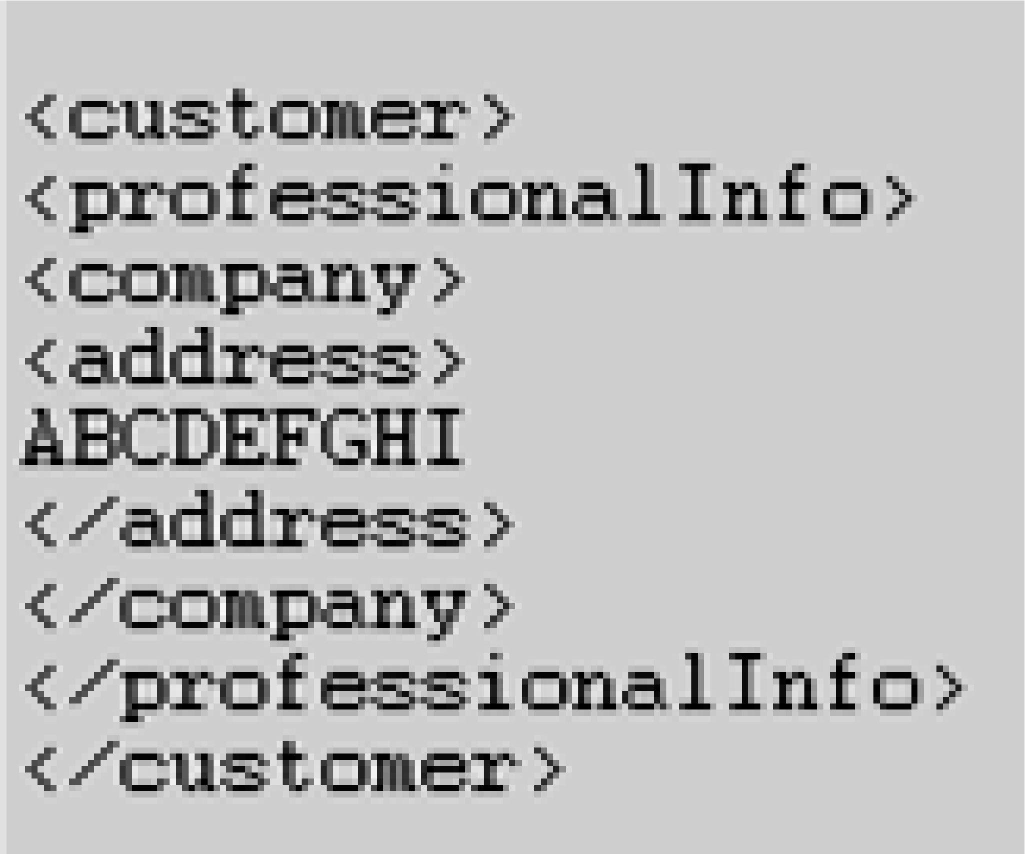
Company

Address: ABCDEFGHI

Name:

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

XML Output for *Customer Information* Application



```
<customer>
<professionalInfo>
<company>
<address>
ABCDEFGHI
</address>
</company>
</professionalInfo>
</customer>
```

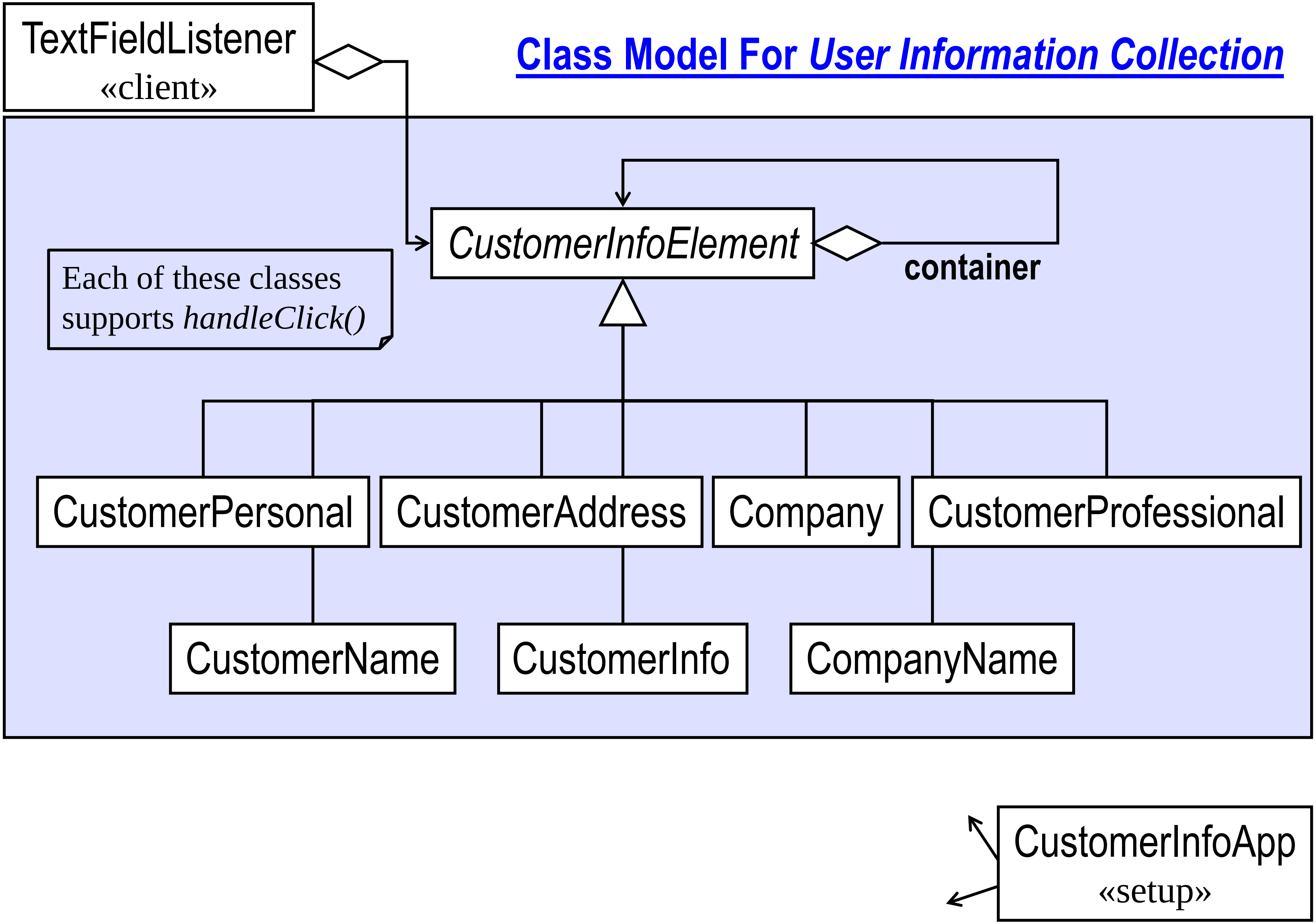
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Flexibility* ←

Isolate the responsibilities of each part of the input form to generate its XML.

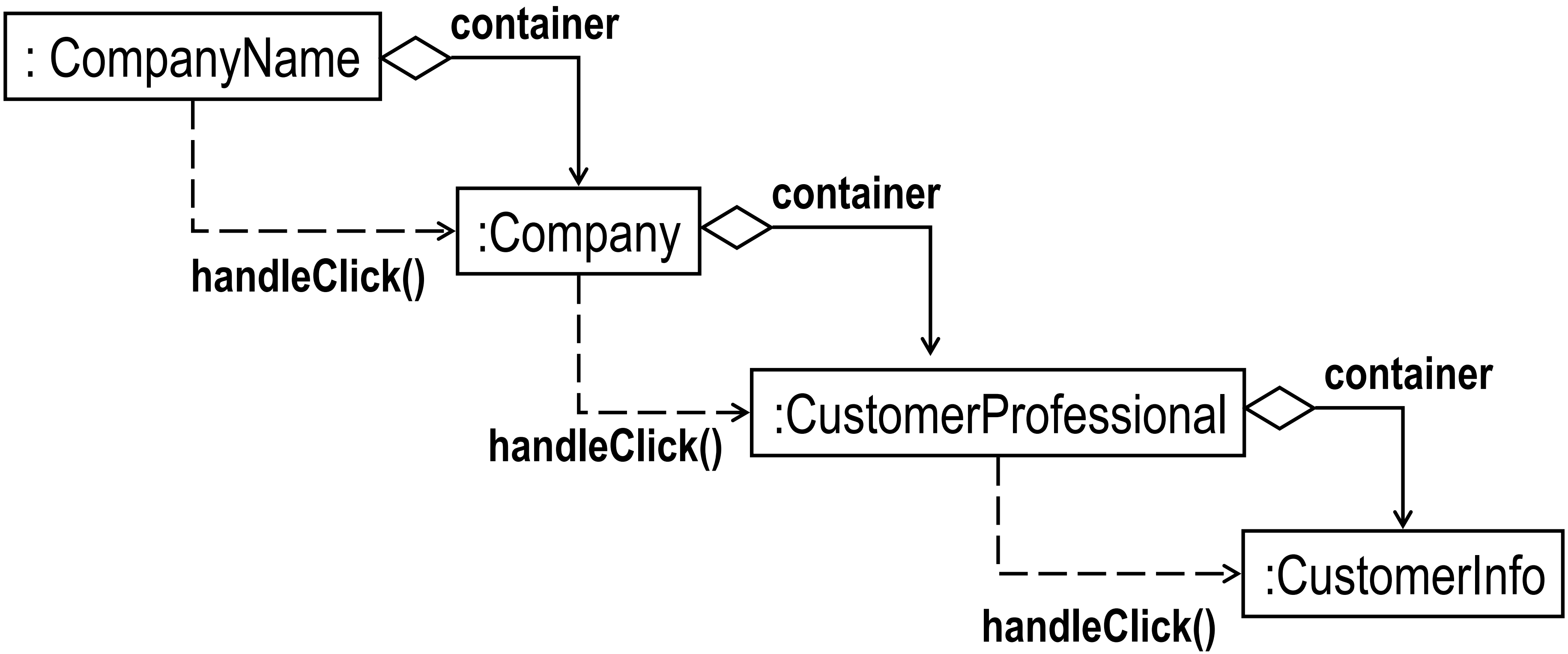
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Class Model For *User Information Collection*



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

Object Model Fragment for *Customer Information* Example



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

Key Concept: → ***Chain of Responsibility*** **Design Pattern** ←

-- to distribute functional responsibility among a collection of objects.

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

Command Design Pattern

Command

Design Purpose

Increase flexibility in calling for a service

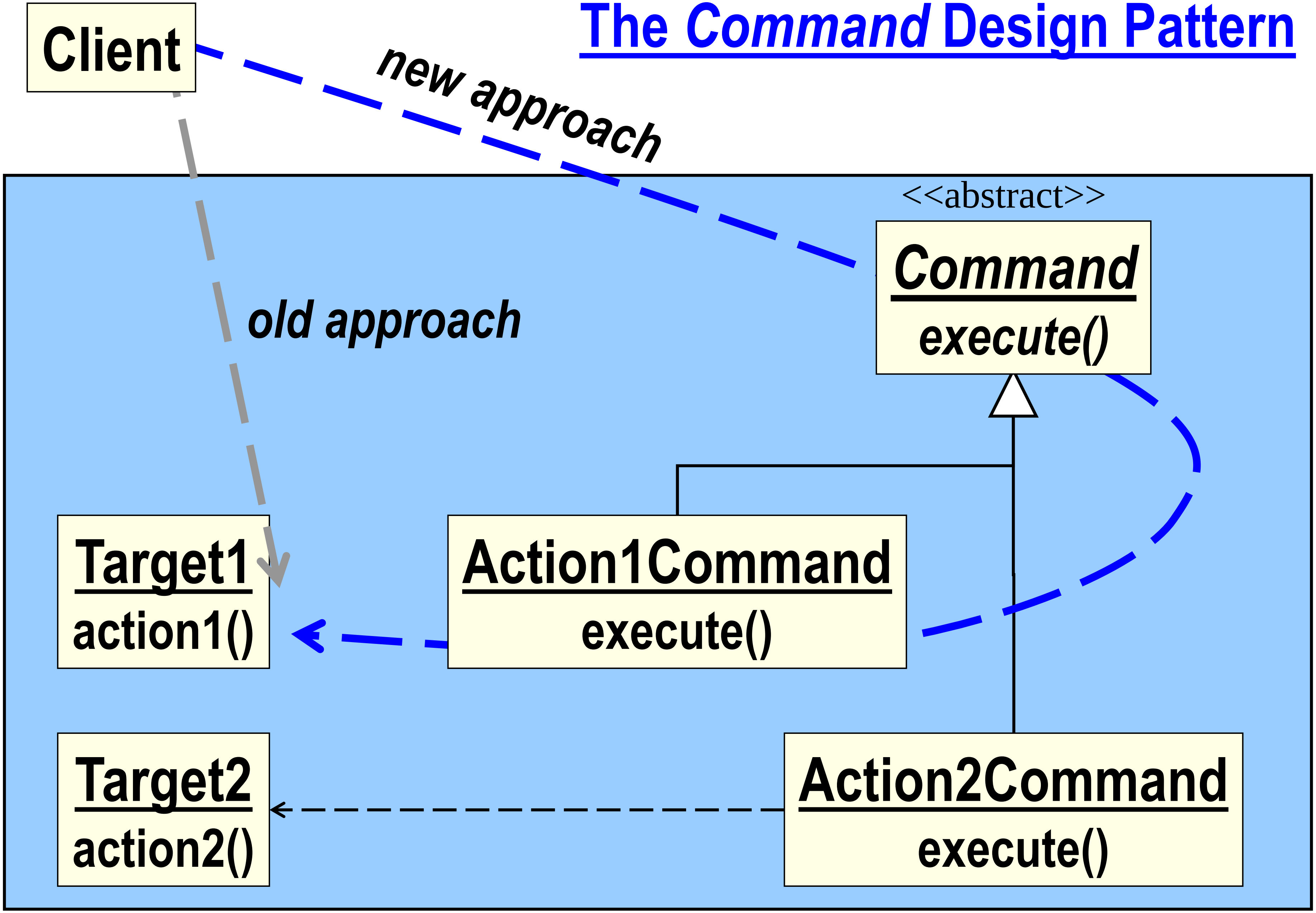
e.g., allow undo-able operations

Design Pattern Summary

Capture operations as classes

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

The *Command* Design Pattern

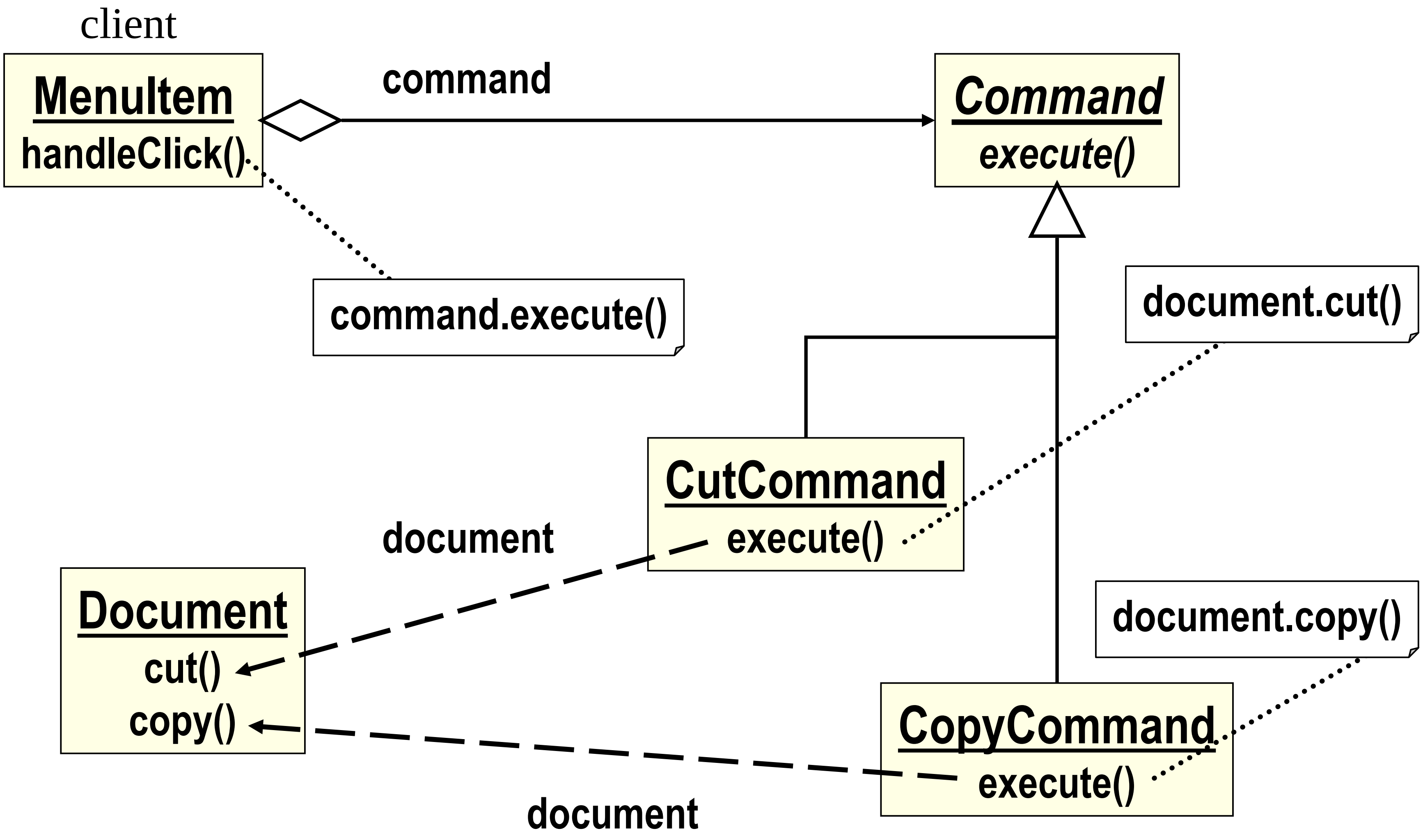


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Source Code Example

```
Command myCommand = Command.getCommand(. . .);

myCommand.execute();
```

The *Command* Design Pattern: Example



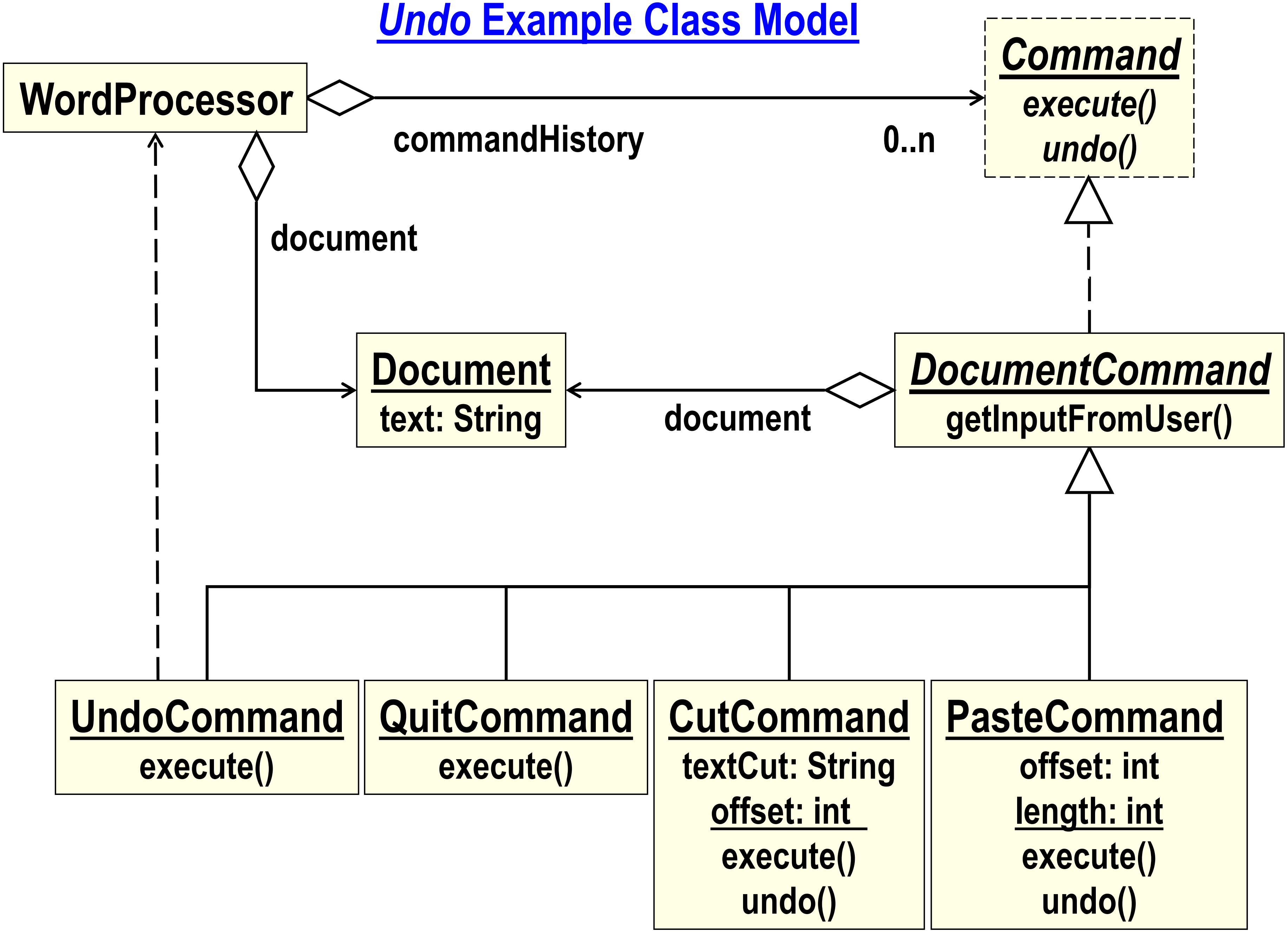
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → *Flexibility* and *Robustness* ←

Isolate the responsibilities of the Word Processor commands, making them save-able and reversible.

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

Undo Example Class Model



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

Key Concept: → *Command* Design Pattern ←

-- to avoid calling a method directly (e.g., so as to record or intercept it).

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

Template Design Pattern

Template

Design Purpose

Allow runtime variants on an algorithm

Design Pattern Summary

Express the basic algorithm in a base class, using method calls where variation is required

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

Example of *Template* Motivation

- Required to solve equations of the form

$$ax^2 + bx + c = 0.$$

- Must be able to handle all input possibilities for *a*, *b*, and *c*.

- This is a tutorial application that must provide full explanations to users about the solutions for all values for *a*, *b*, and *c*.

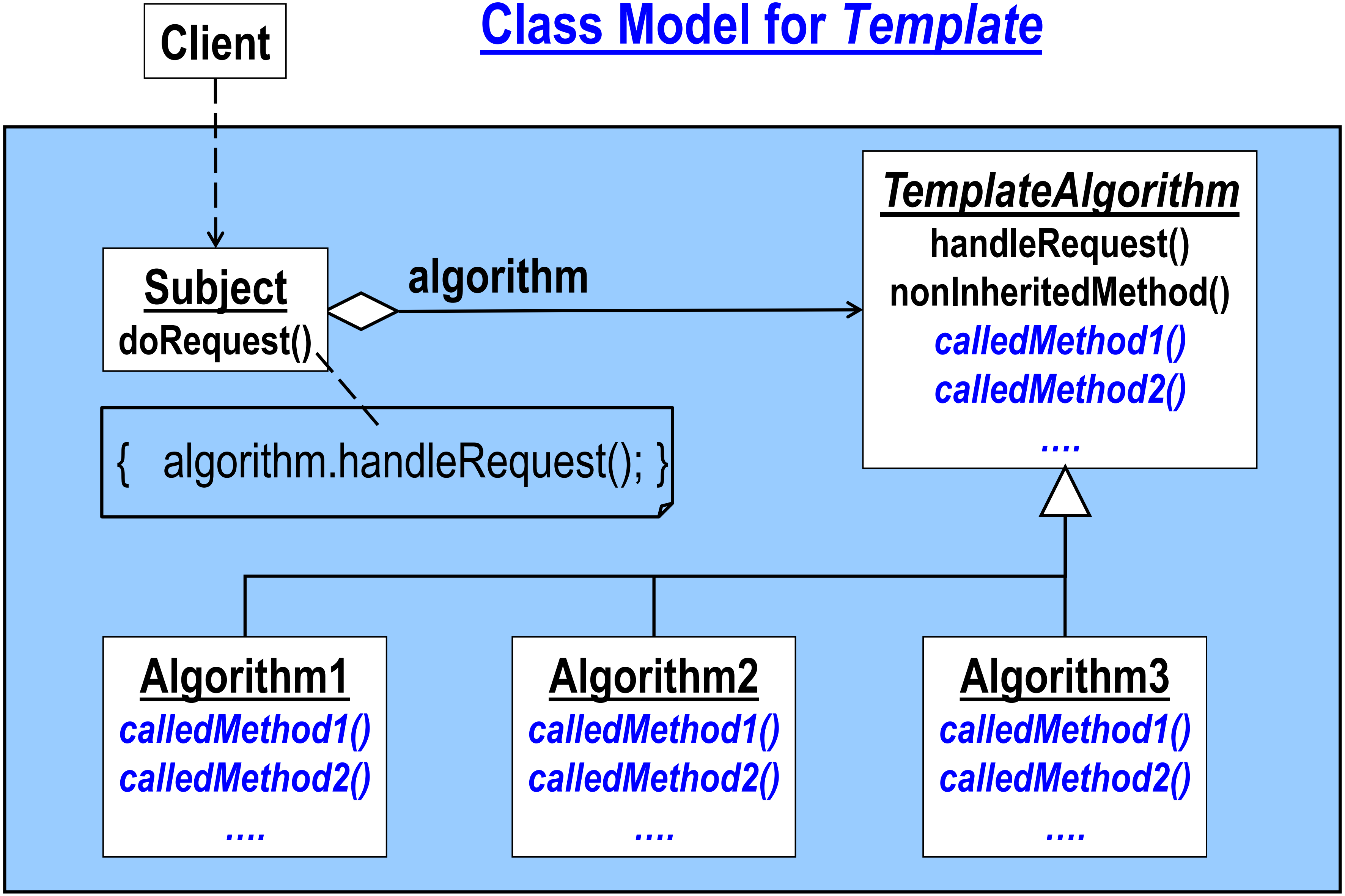
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

A Basic Quadratic Algorithm

1. Report progress
2. Display number of solutions
3. Display first solution, if any
4. Display second solution, if any

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

Class Model for *Template*



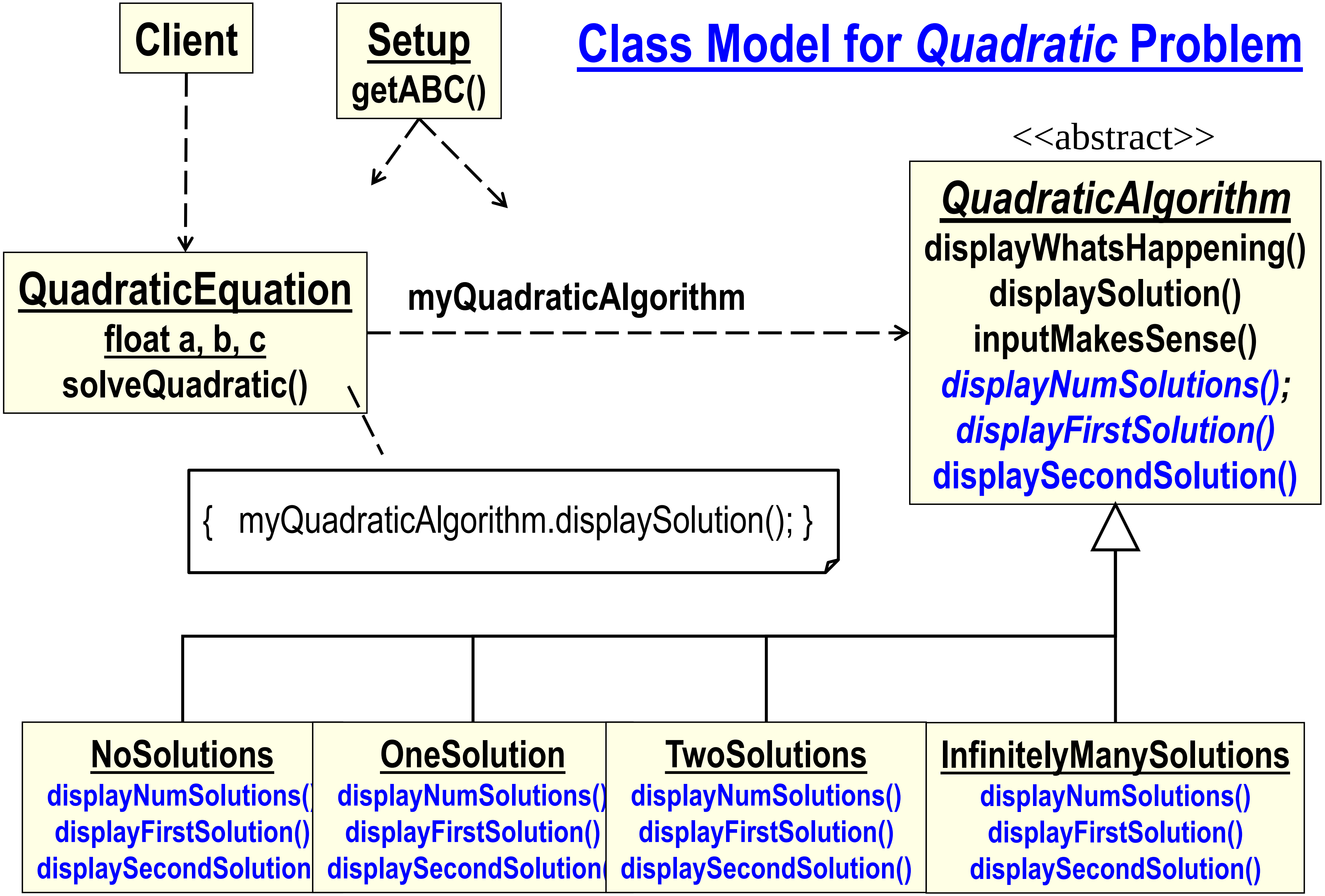
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Design Goal At Work: → ***Flexibility*** and ***Robustness*** ←

Isolate the main algorithm for quadratic solution display.
Isolate the variants that depend on the coefficients.

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

Class Model for *Quadratic* Problem



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

Source Code Example

```
QuadraticEquation myQuadraticEquation;

myQuadraticEquation.solve();
```

```
abstract class QuadraticAlgorithm
{

}


class OneSolution extends QuadraticAlgorithm
{

}


class QuadraticEquation
{
   QuadraticAlgorithm myQuadraticAlgorithm;
}
```

Key Concept: → *Template* Design Pattern ←

-- to capture a basic algorithm and its variants.

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

Summary of *Behavioral Design Patterns*

Behavioral Design Patterns **capture behavior among objects**

- □ ***Interpreter*** **handles expressions in grammars**
- □ ***Iterator*** **visits members of a collection in a sequential fashion**
- □ ***Mediator*** **captures behavior among peer objects without building a dependency between the objects**
- □ ***Observer*** **updates objects affected by a single object**
- □ ***State*** **allows method behavior to depend on current status**
- □ ***Chain of Responsibility*** **allows a set of objects to provide functionality collectively**
- □ ***Command*** **captures function flexibly (e.g. undo-able)**
- □ ***Template*** **captures basic algorithms, allowing variability**

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