Chapter 8
Structural Design Patterns

- Facade
- Decorator
- Composite
- Adapter
- Flyweight
- Proxy

Façade Design Pattern

Facade Design Purpose

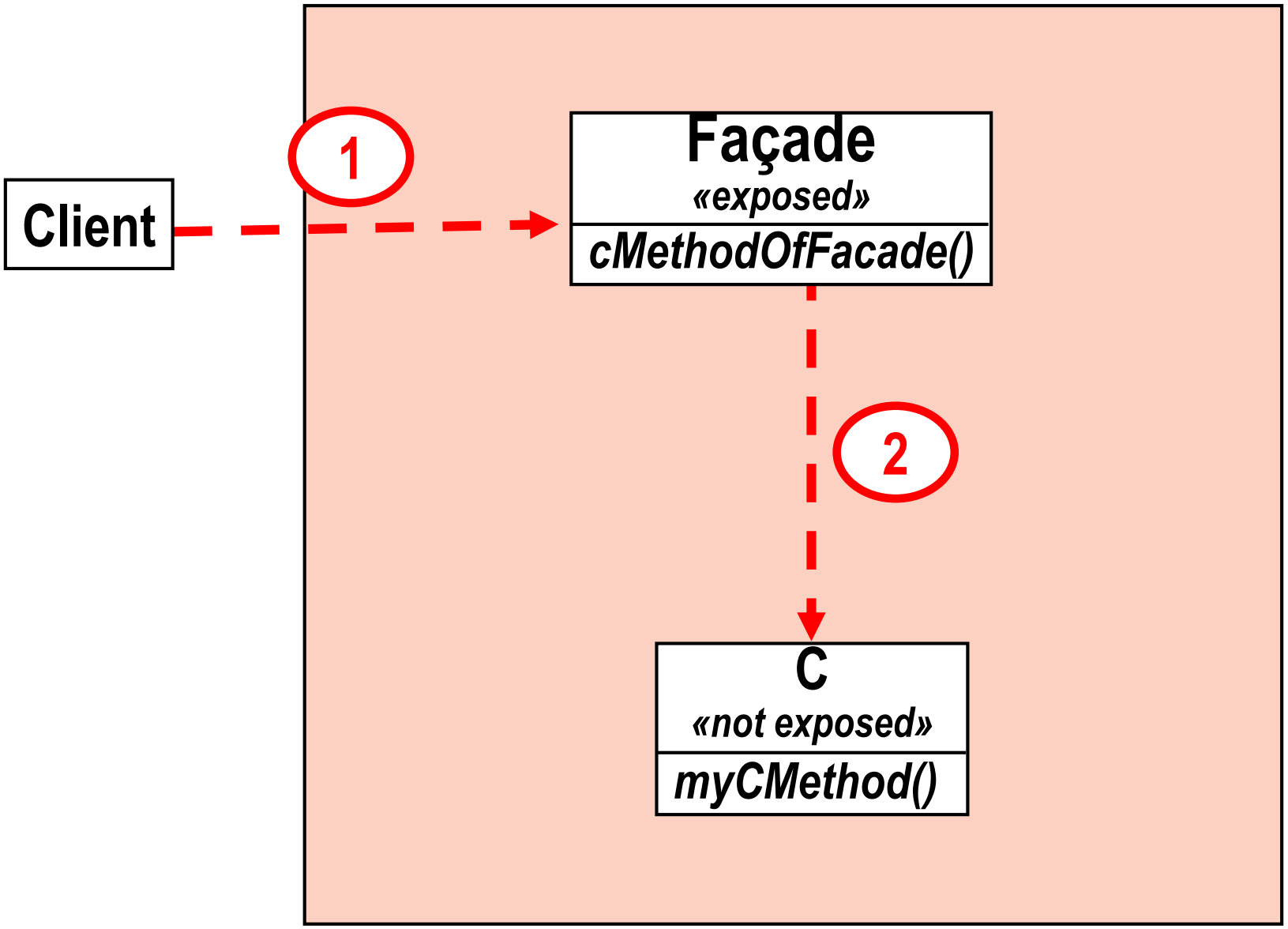
Provide an interface to a package of classes

Design Pattern Summary

Define a class (typically a singleton) that is the sole means for obtaining functionality from the package.

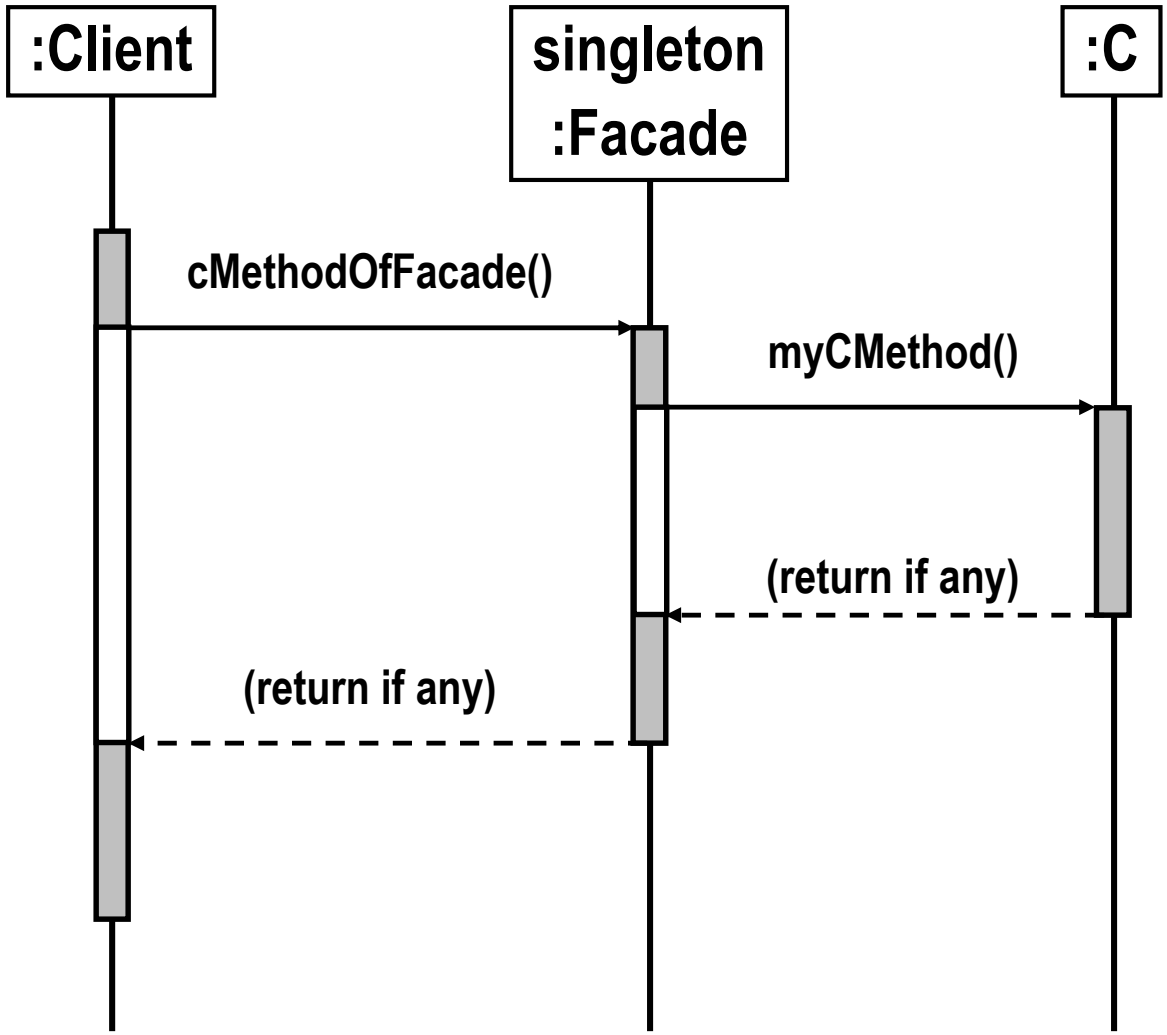
***Notes*: the classes need not be organized as a package; more than one class may be used for the façade.**

Facade Design Pattern Structure

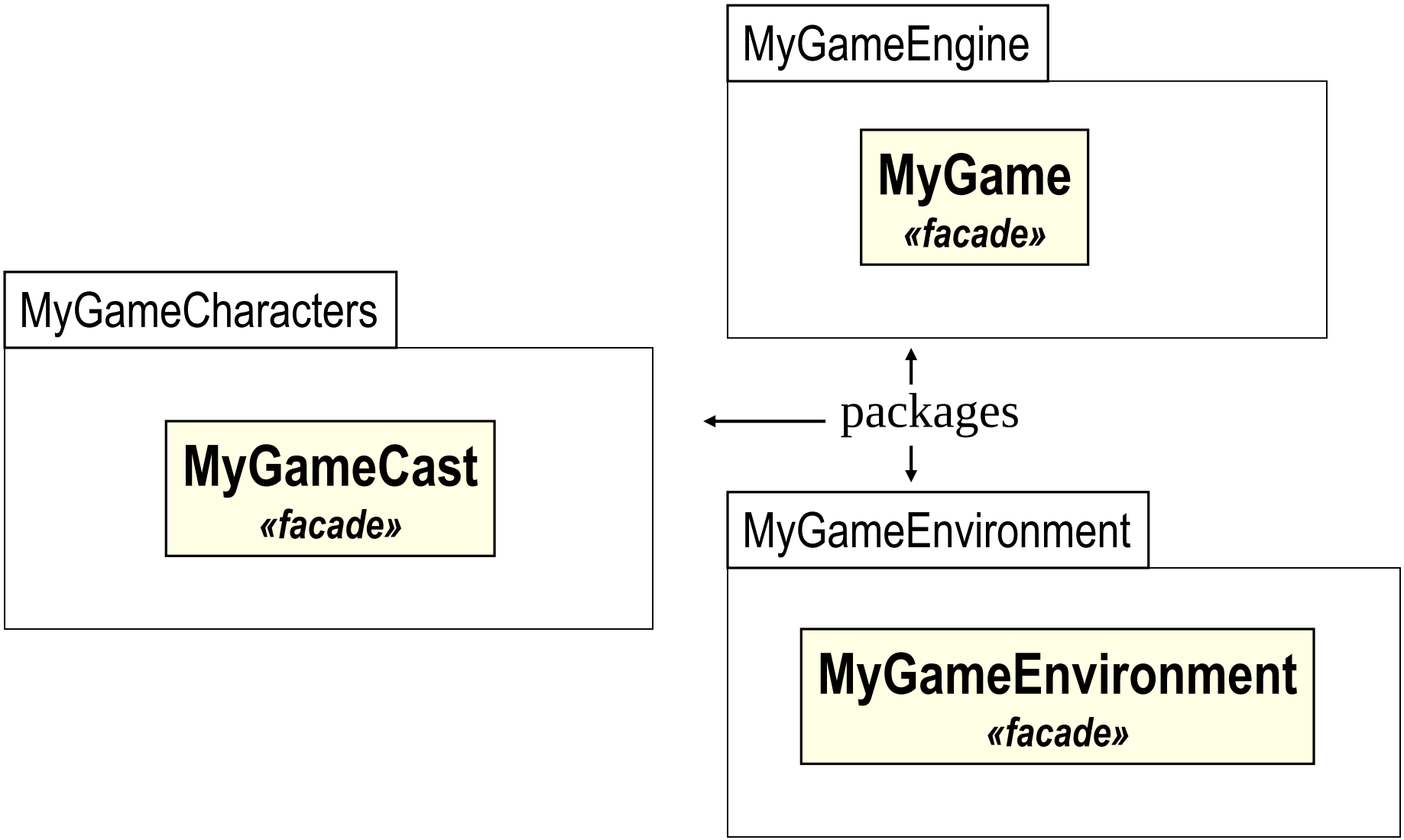


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for Façade



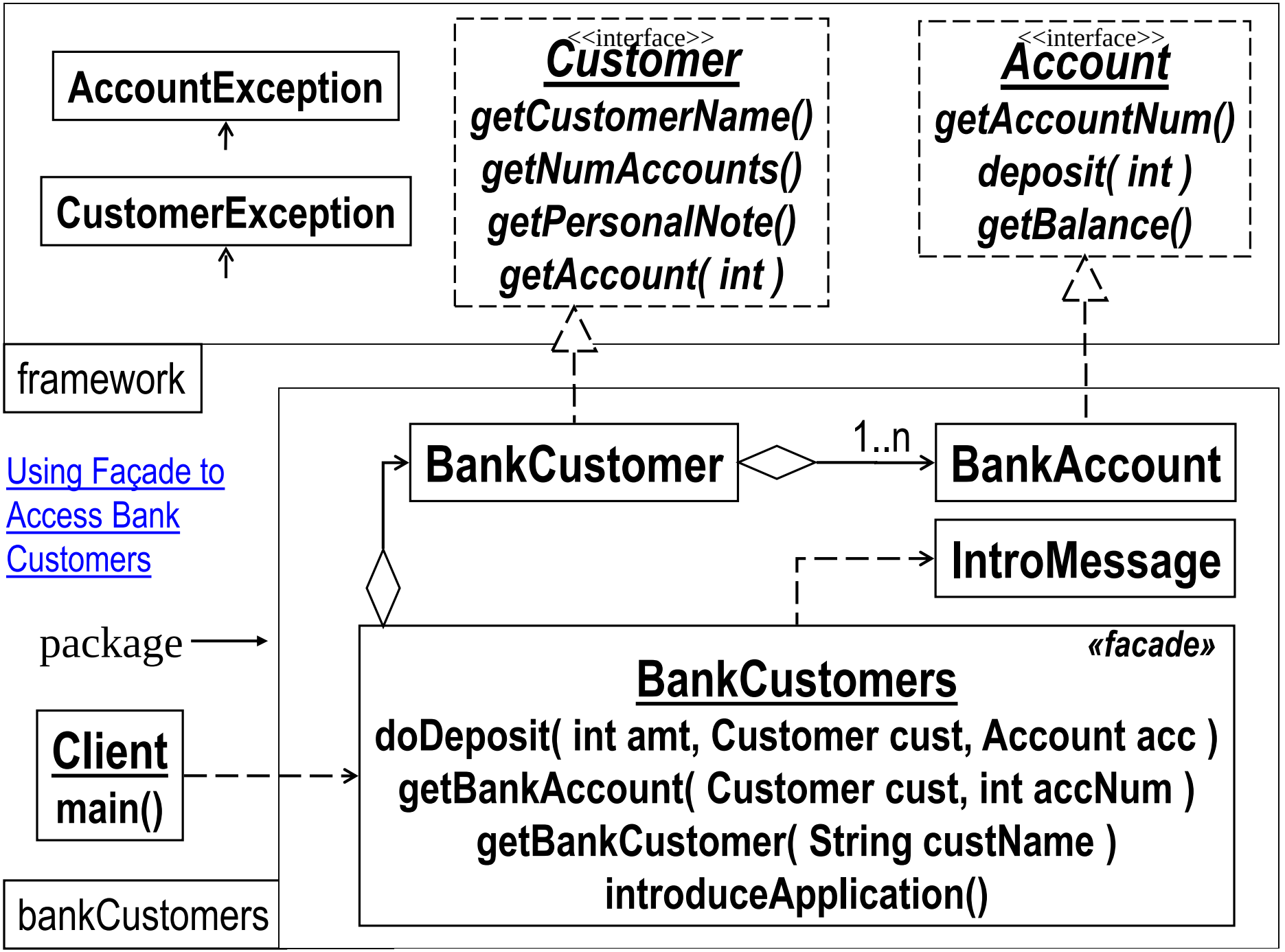
Using Façade for Architecture of a Video Game



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

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

Collecting customer-related classes in a package with a clear interface clarifies the design, allows independent verification, and makes this part reusable.



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

Key Concept: → *Facade* Design Pattern ←

-- modularizes designs by hiding complexity (similar to the web services provided by a web site)

Decorator Design Pattern

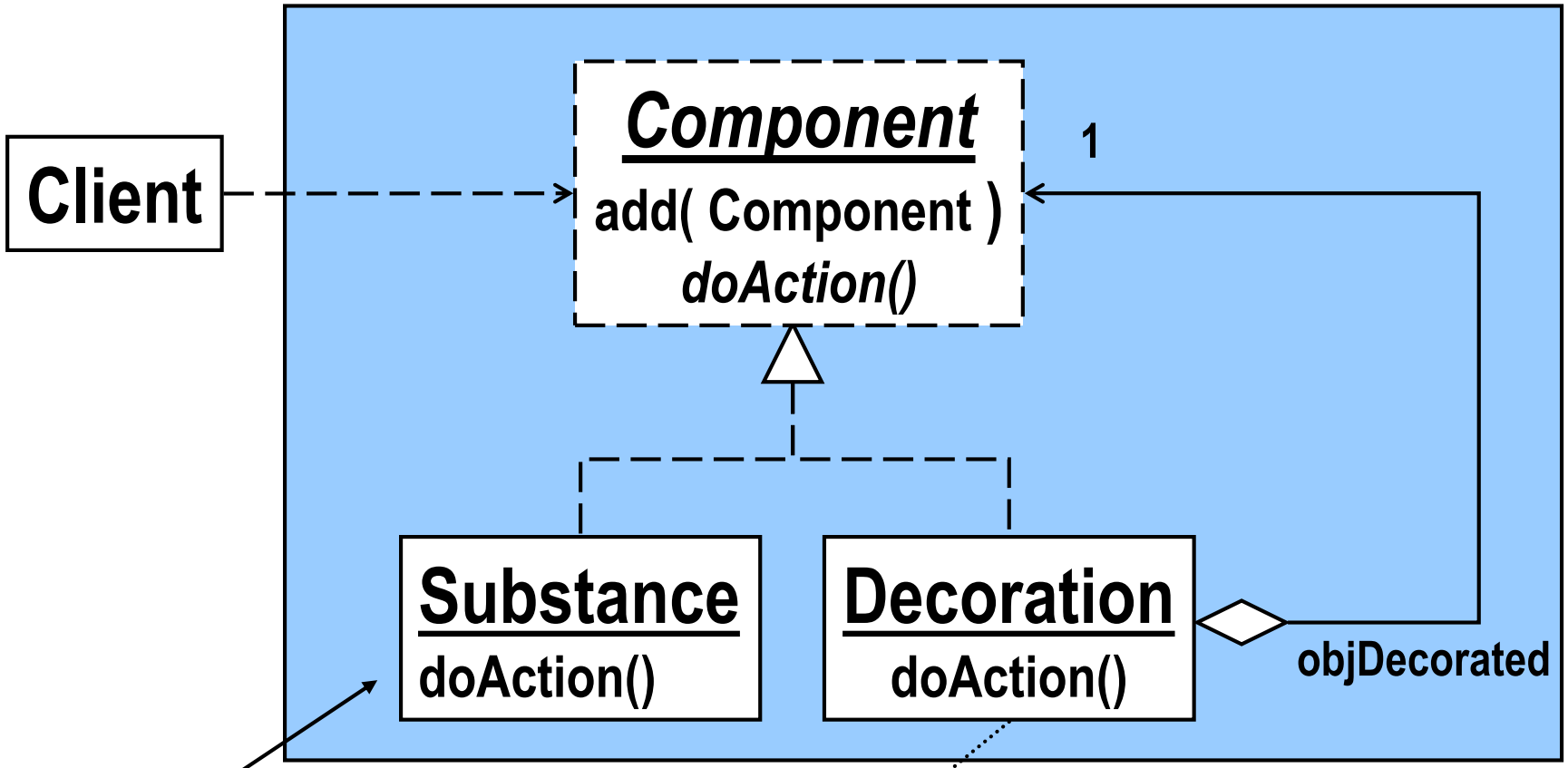
Decorator Design Purpose

Add responsibilities to an object at runtime.

Design Pattern Summary

Provide for a linked list of objects, each encapsulating responsibility.

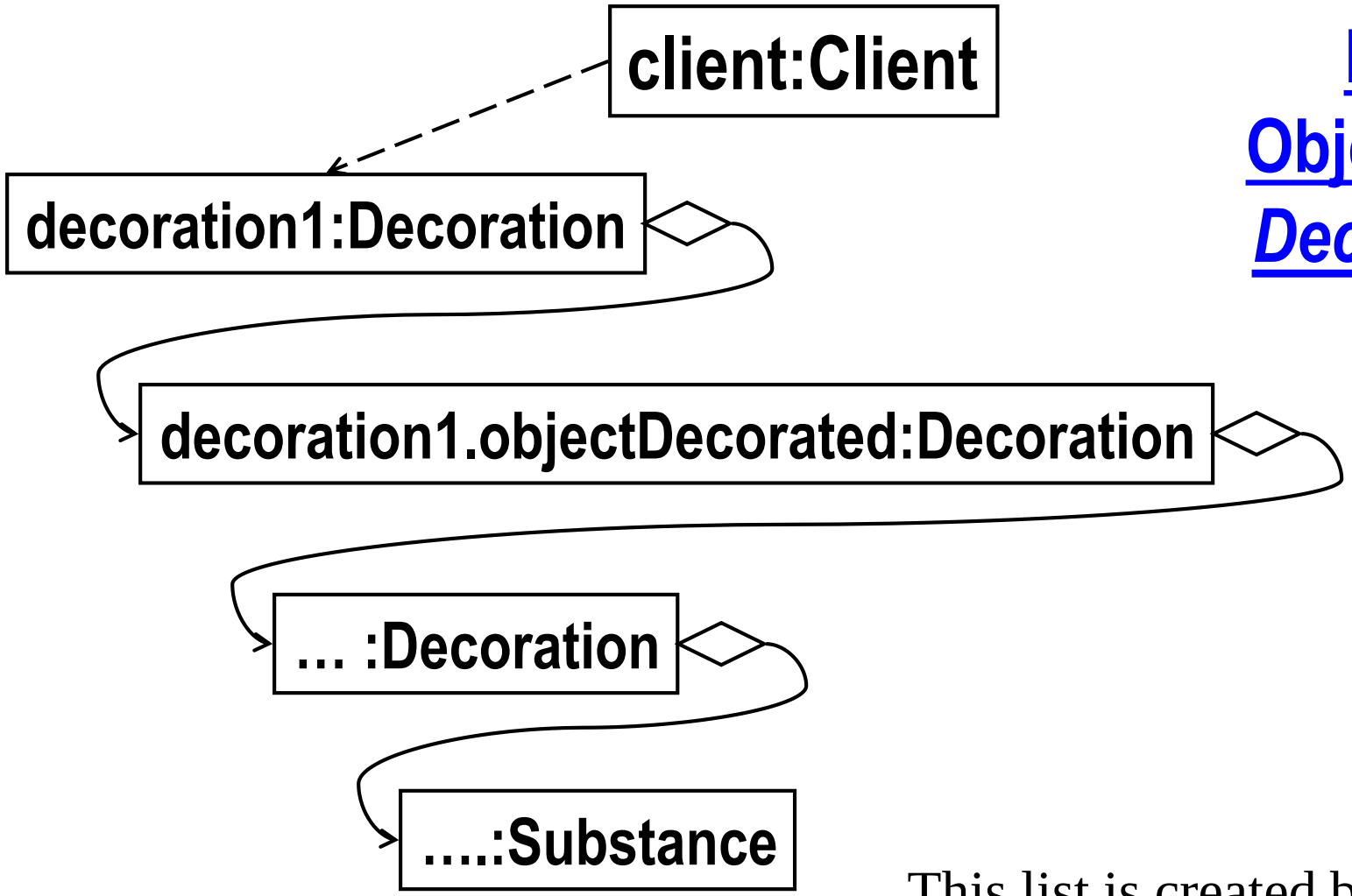
Decorator Class Model



Undecorated class

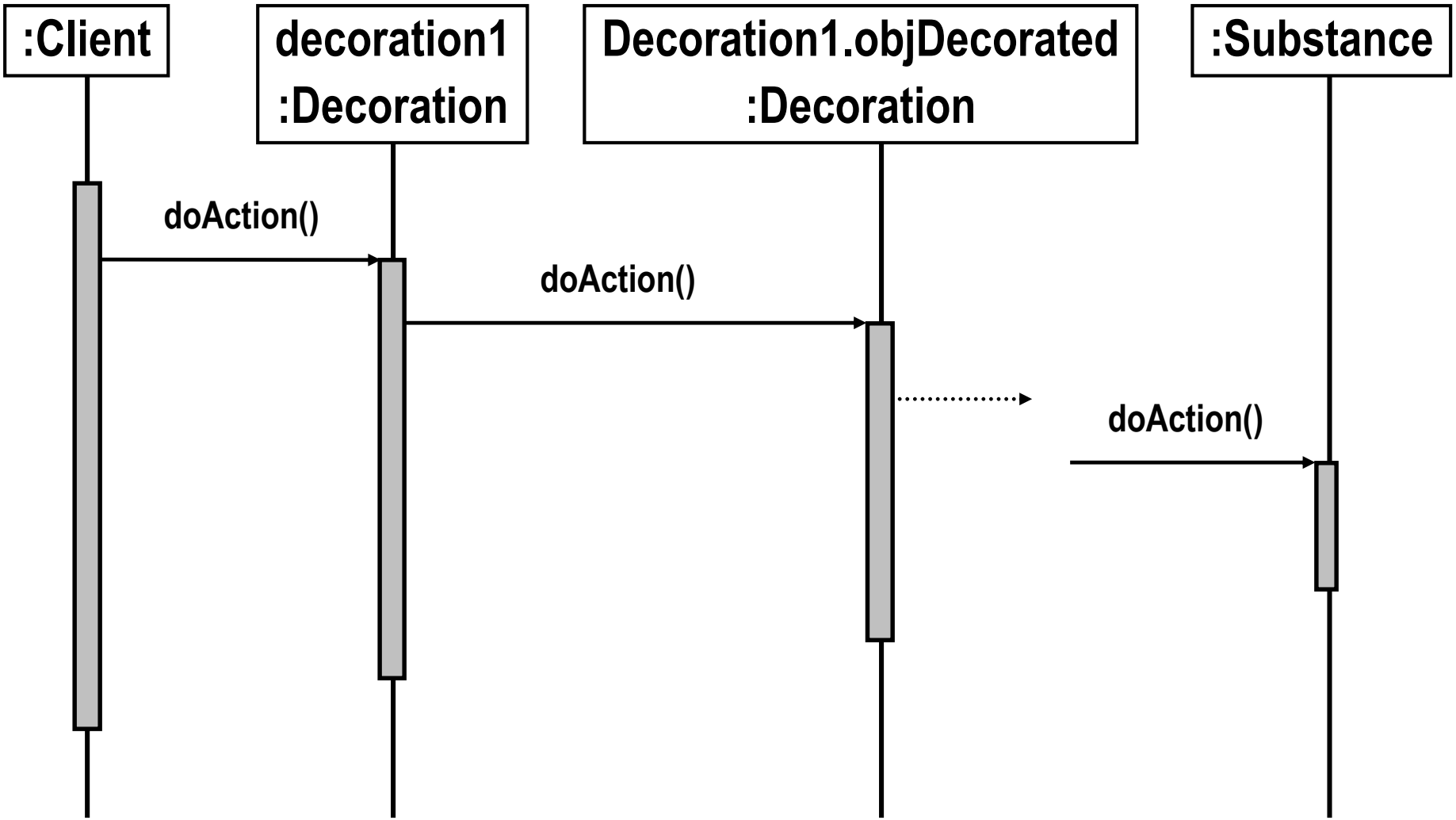
```
void doAction()
{    ..... // do actions special to this decoration
     objDecorated.doAction();  // pass along
}
```

Linked Objects in *Decorator*



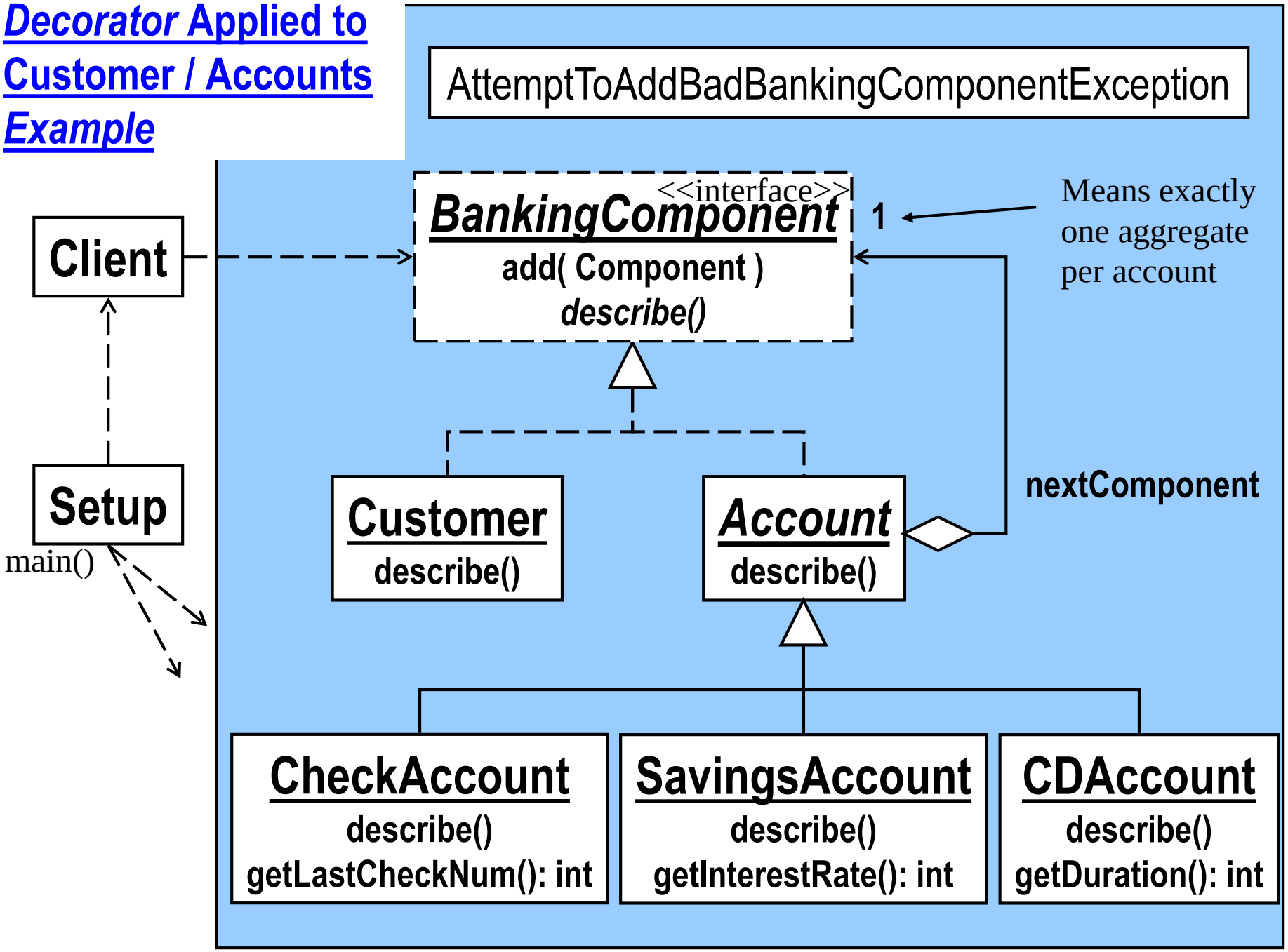
This list is created backwards so that the last decoration added is the first one to be executed

Sequence Diagram for *Decorator*

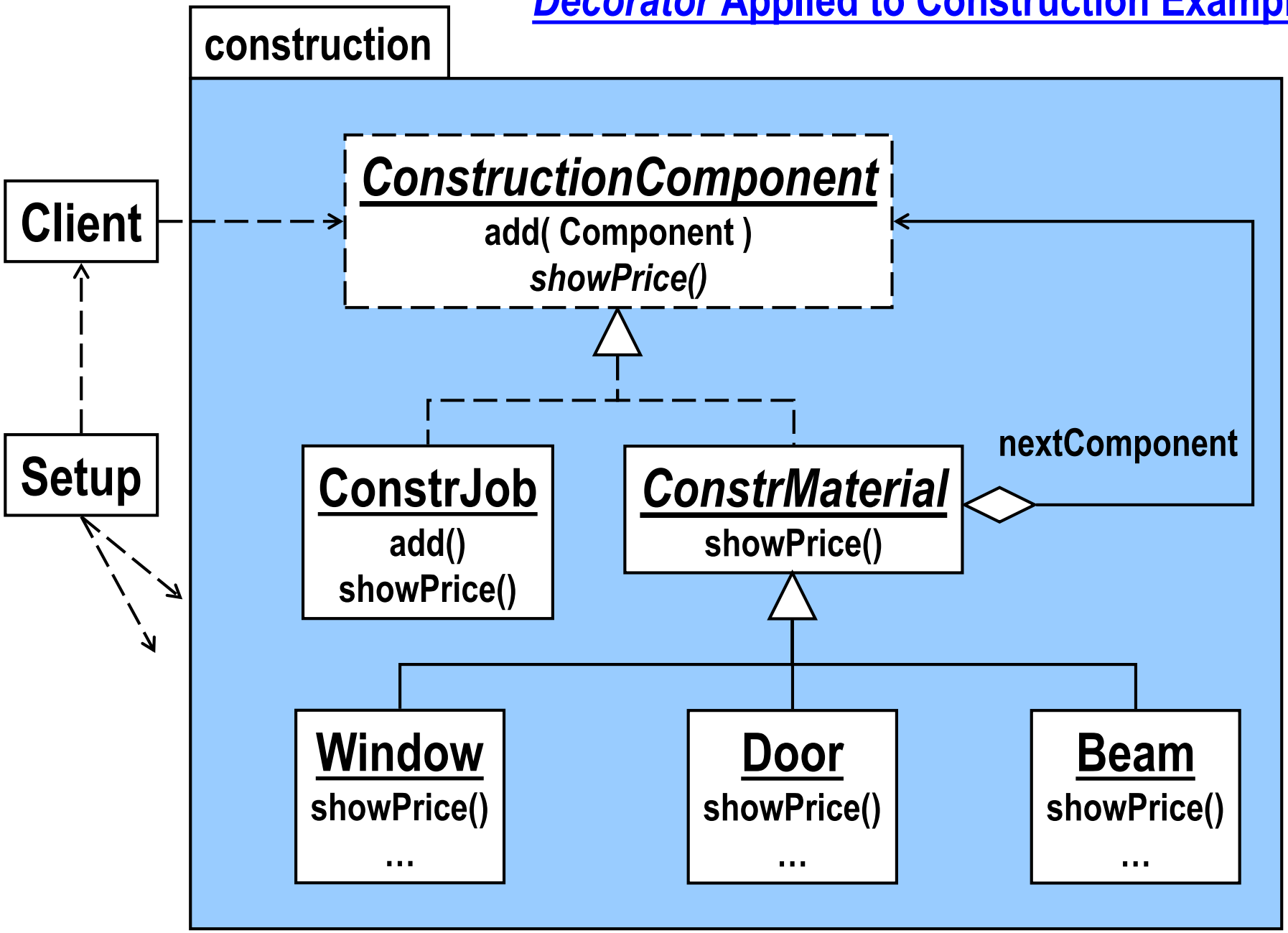


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Decorator Applied to Customer / Accounts *Example*

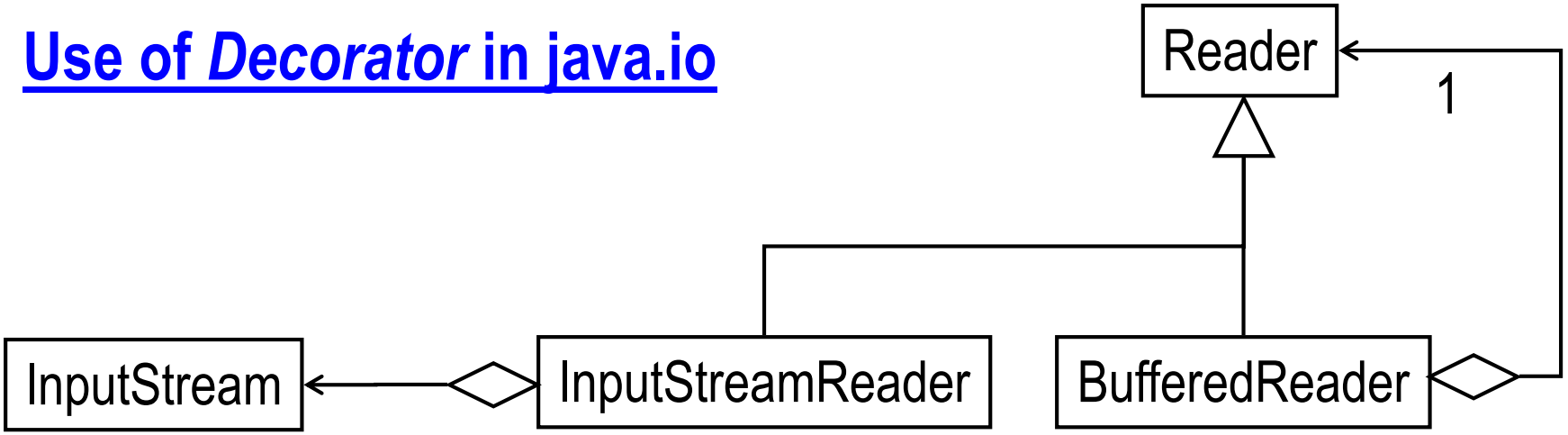


Decorator Applied to Construction Example



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

Use of *Decorator* in java.io




```
BufferedReader bufReader = new BufferedReader
    (new InputStreamReader(System.in) );
```

Key Concept: → *Decorator* Design Pattern ←

-- allows addition to and removal from objects at runtime

-- represents an object version of a linked list where a client does not need to know about the subclasses in the list

-- emphasizes the structural properties of the substance in the list

Composite Design Pattern

Composite

Design Purpose

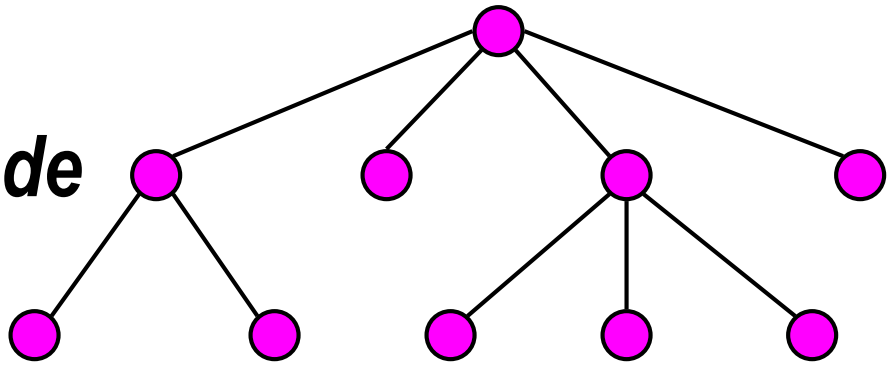
Represent a Tree of Objects

Design Pattern Summary

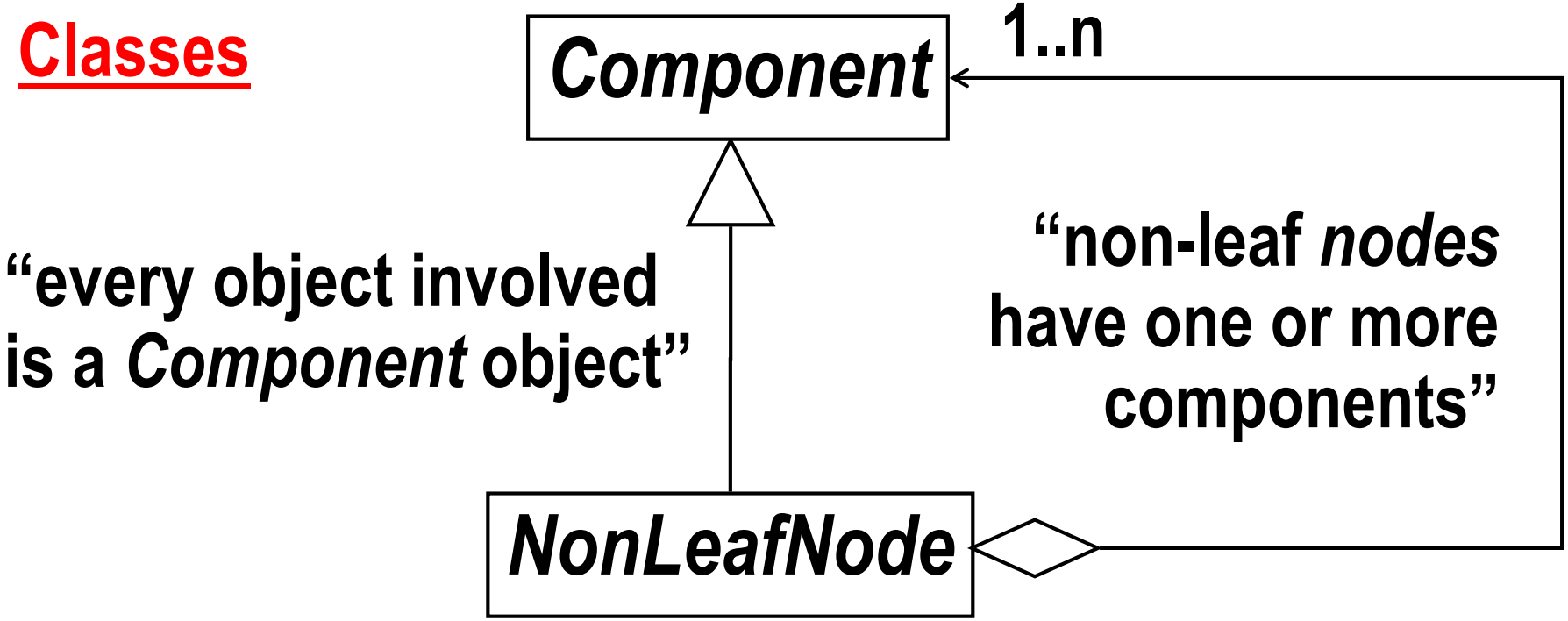
Use a recursive form in which the tree class aggregates and inherits from the base class for the objects.

Basis for *Composite* Class Model

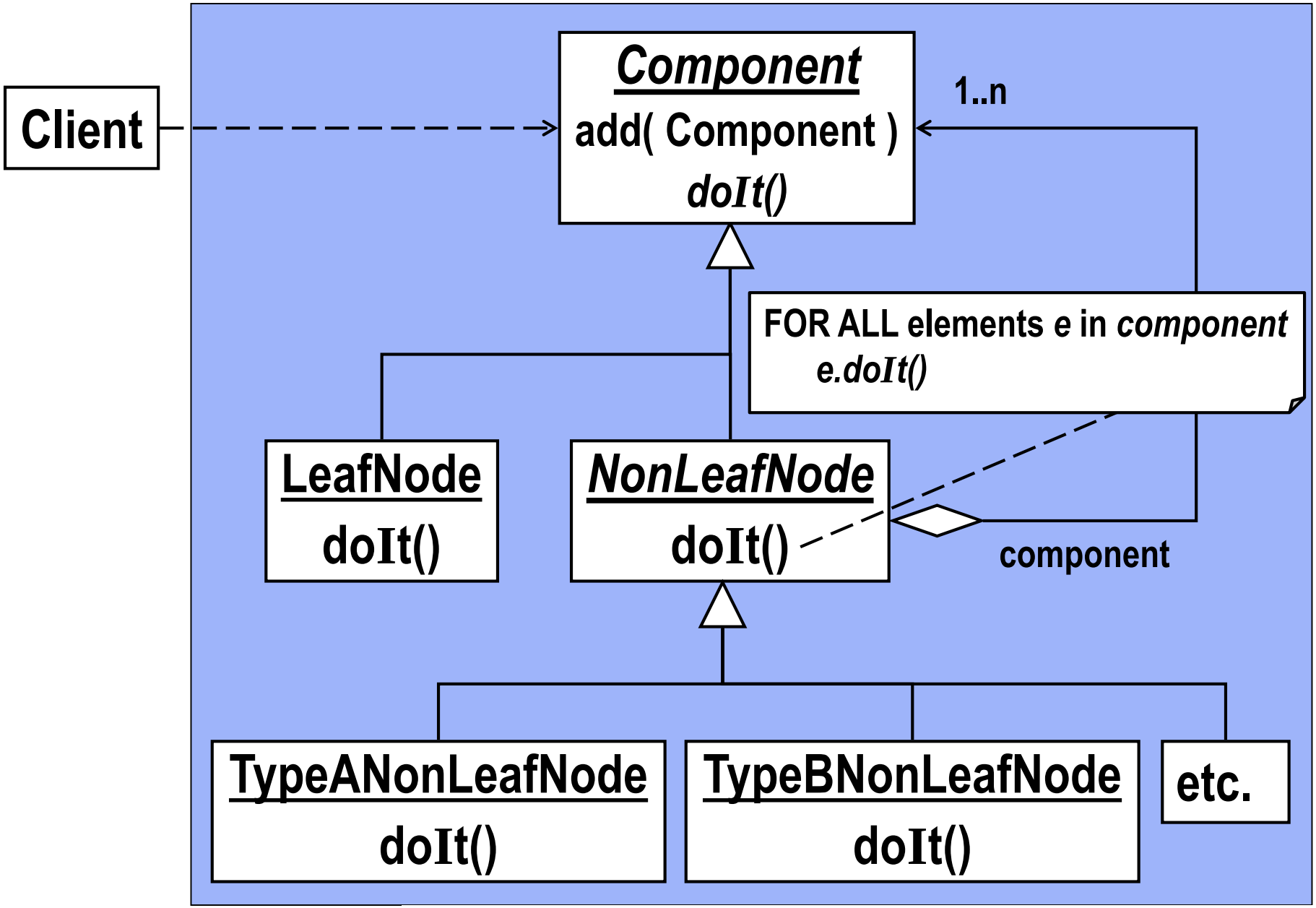
Objects



Classes

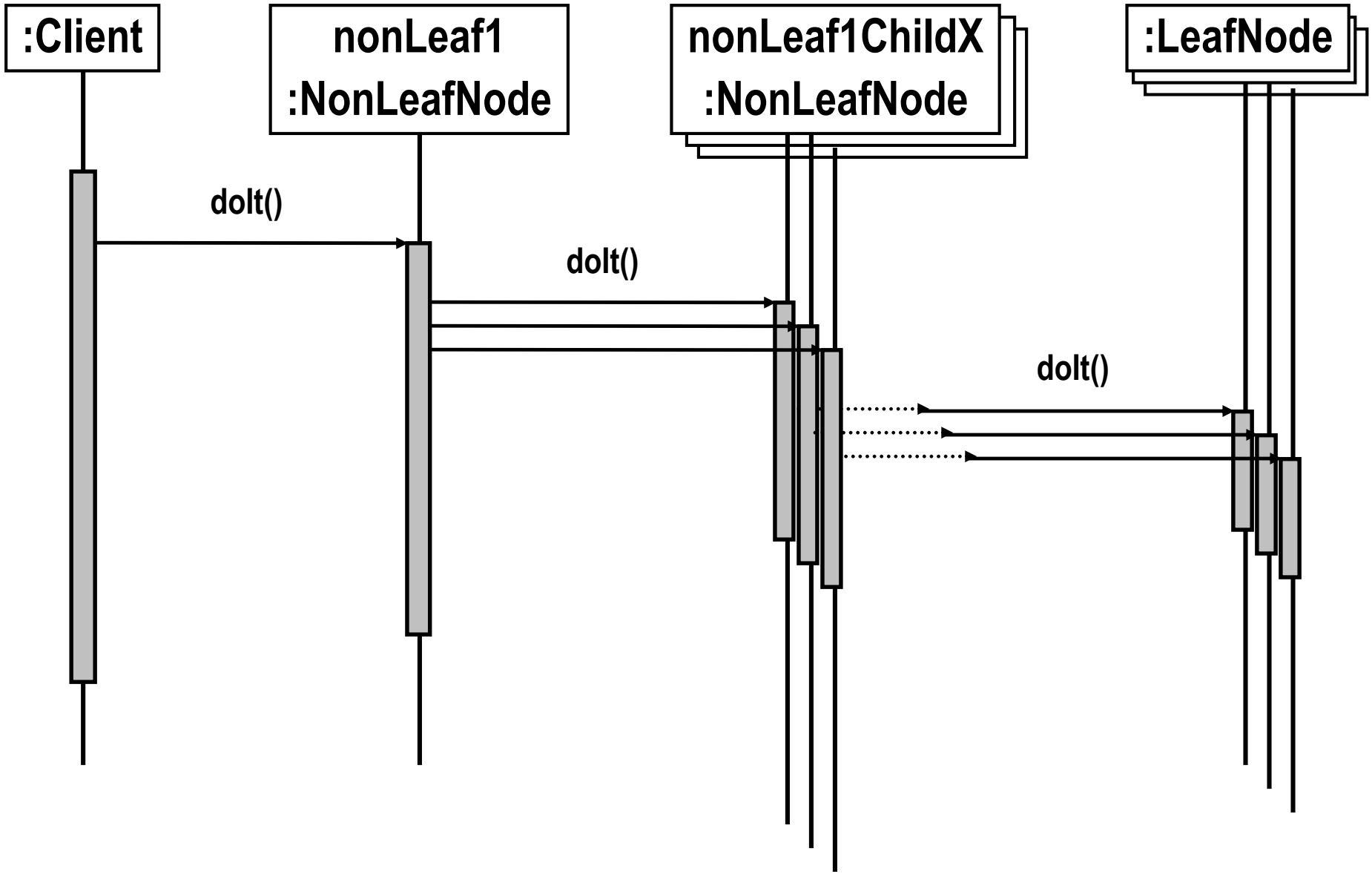


Composite Class Model



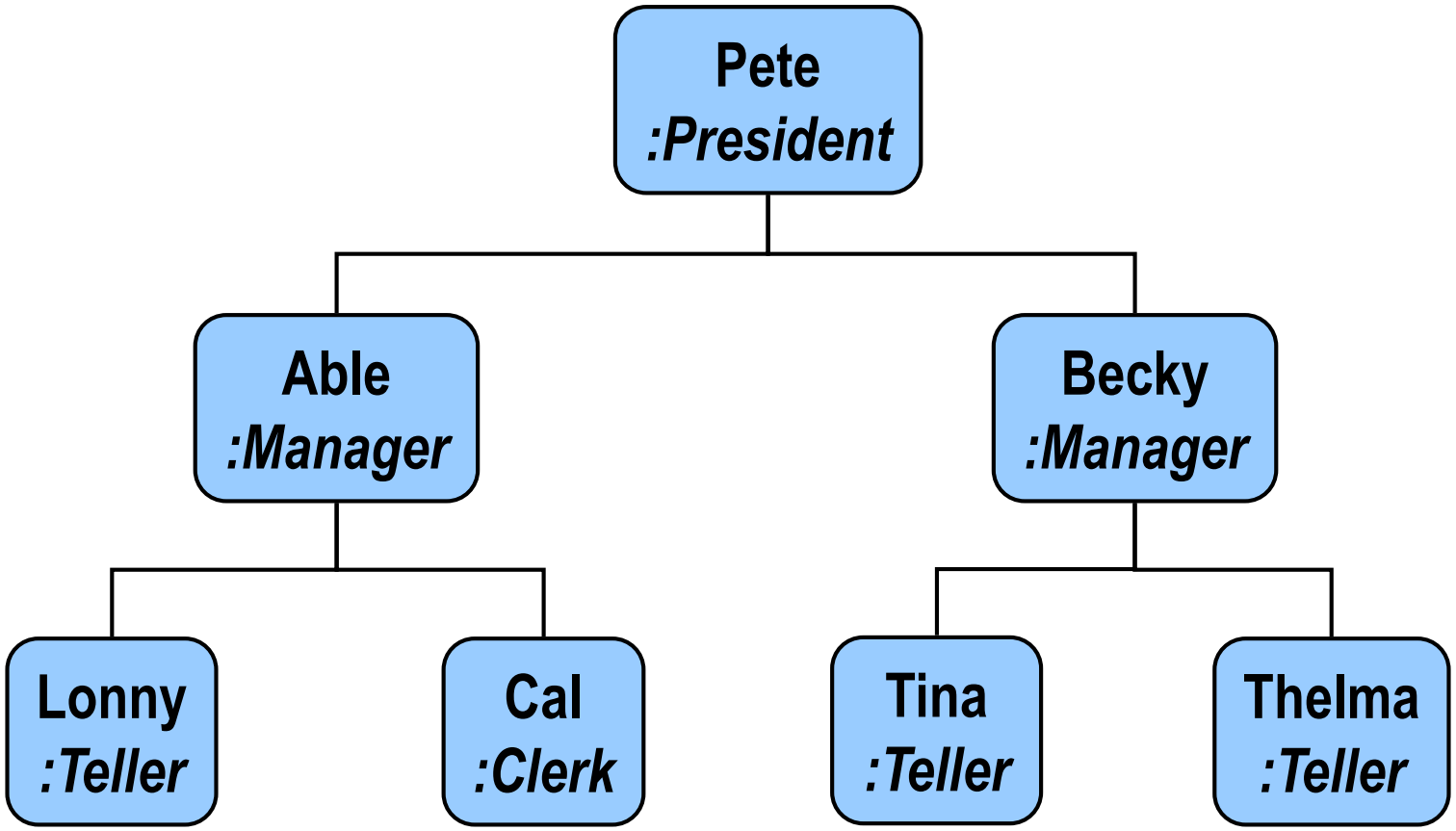
Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Sequence Diagram for *Composite*



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

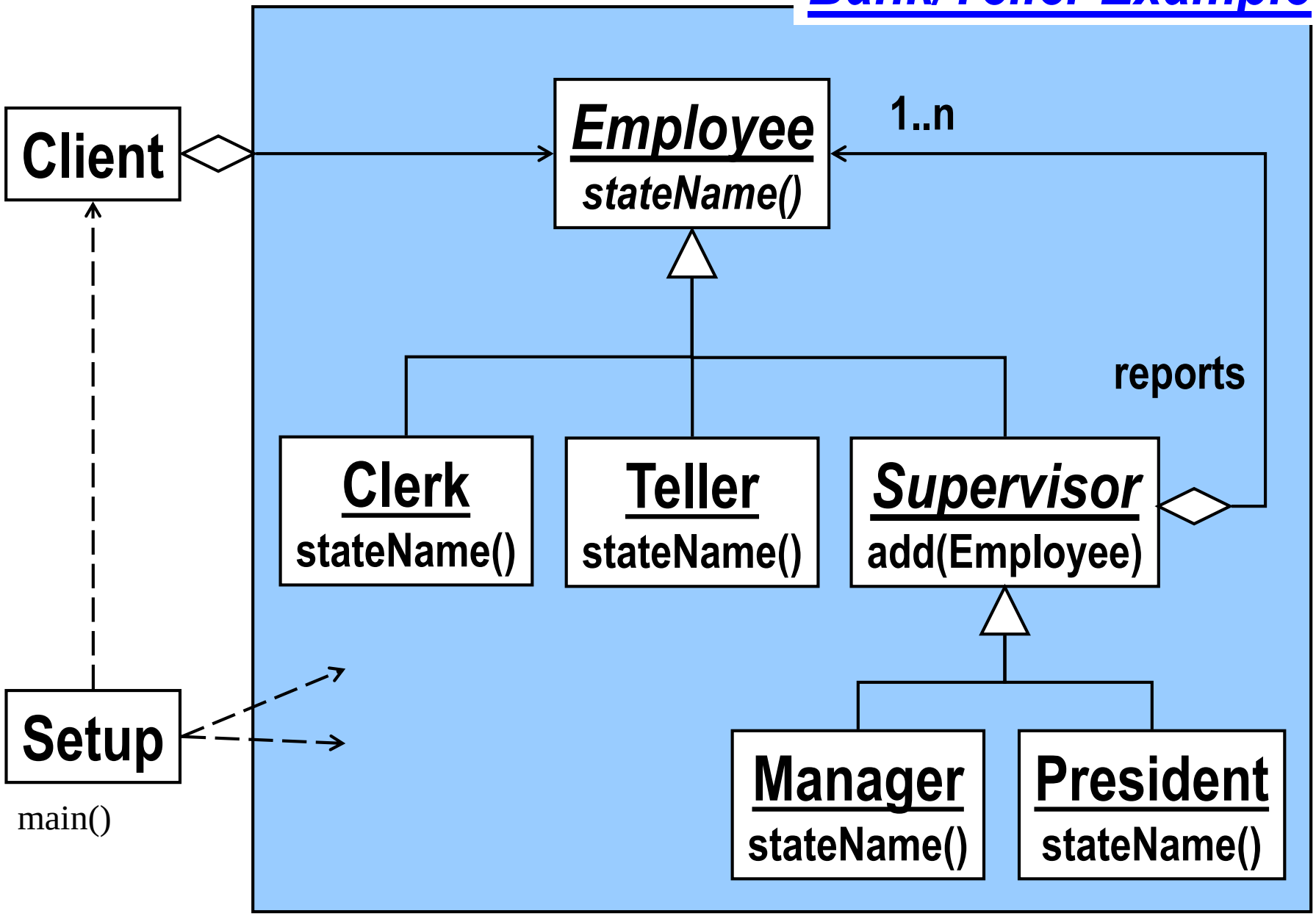
Employee Hierarchy



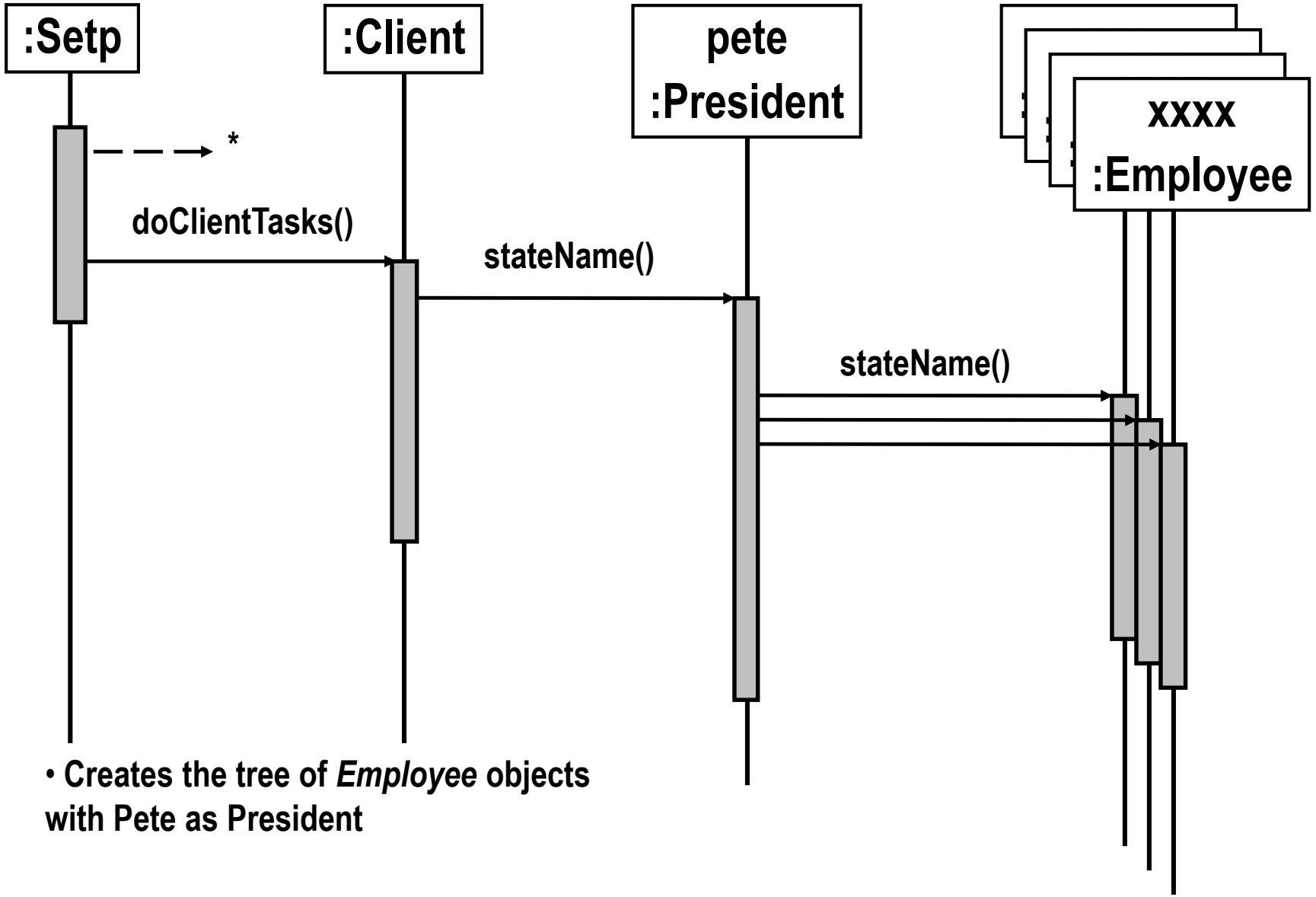
Design Goal At Work: → Flexibility, Correctness ←

We need to add and remove employees at runtime and execute operations on all of them.

Bank/Teller Example

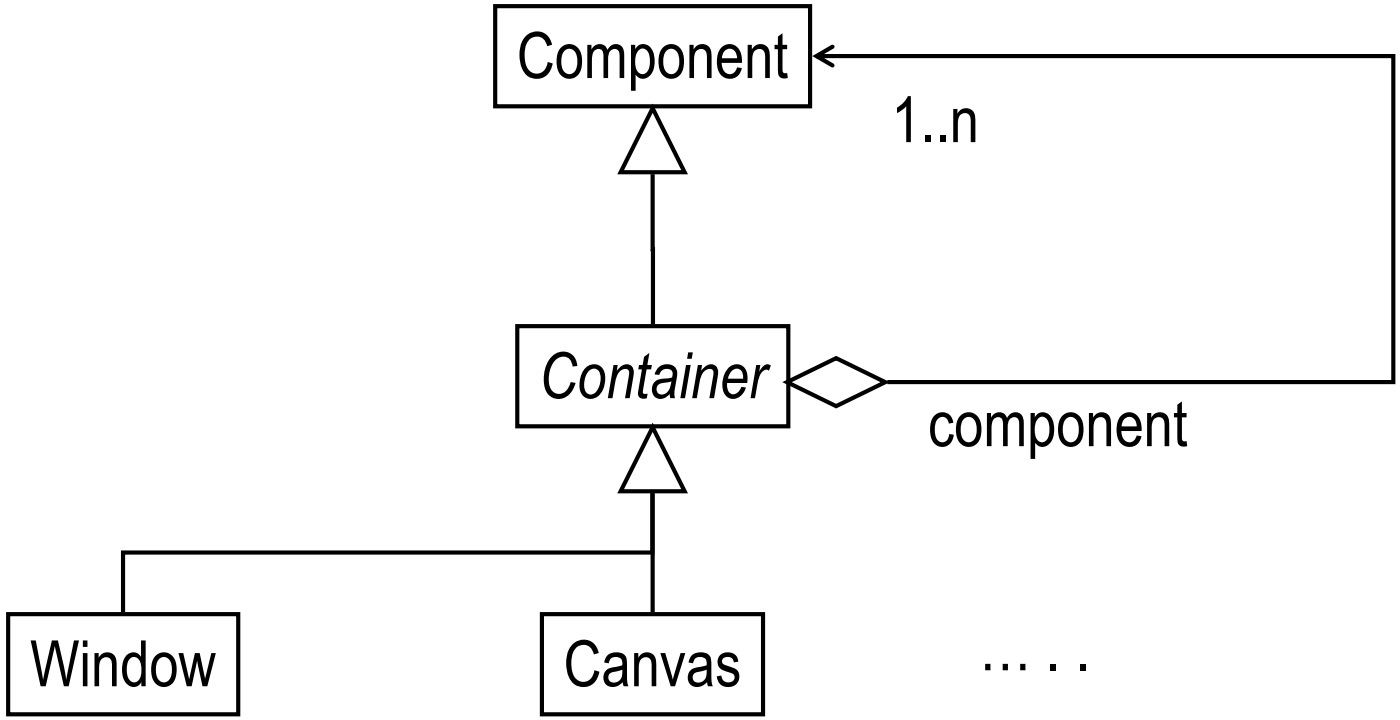


Sequence Diagram for *Bank/Teller Example*



- Creates the tree of *Employee* objects with Pete as President

Composite in *java.awt*



Key Concept: → *Composite* Design Pattern ←

-- used to represent trees of objects.

Adapter Design Pattern

Adapter

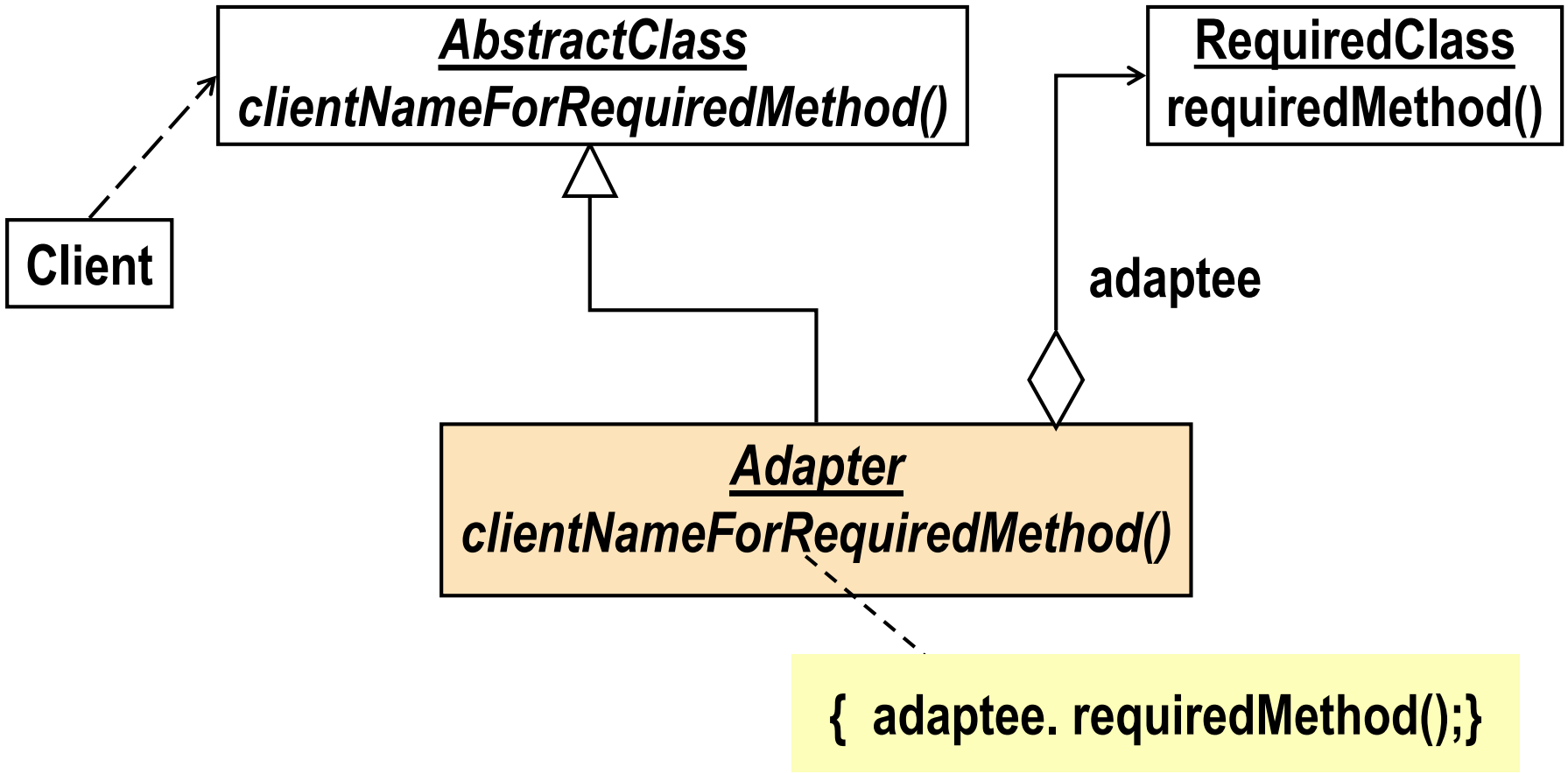
Design Purpose

Allow an application to use external functionality in a retargetable manner.

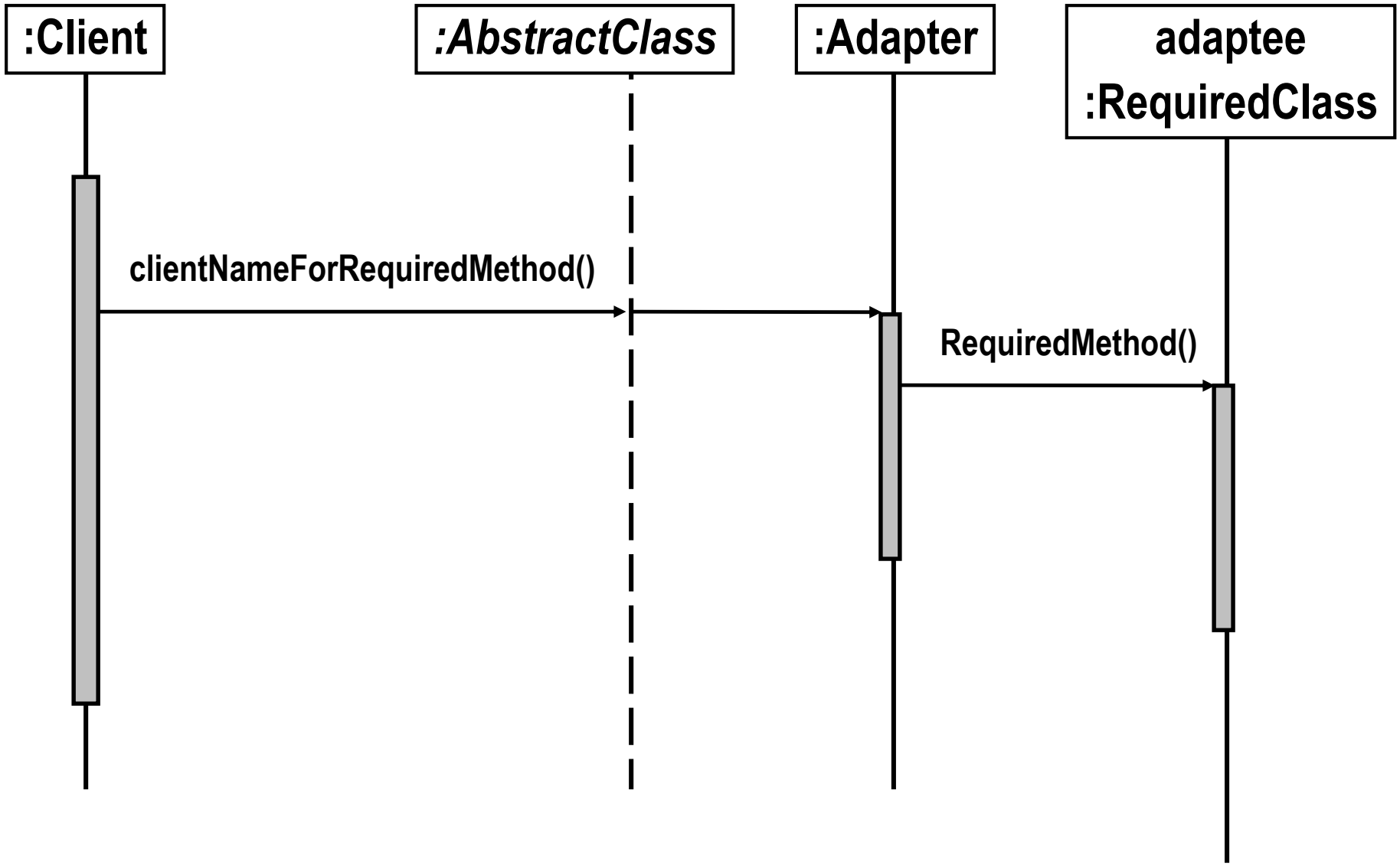
Design Pattern Summary

Write the application against an abstract version of the external class; introduce a subclass that aggregates the external class.

Adapter Example



Sequence Diagram for *Adapter*

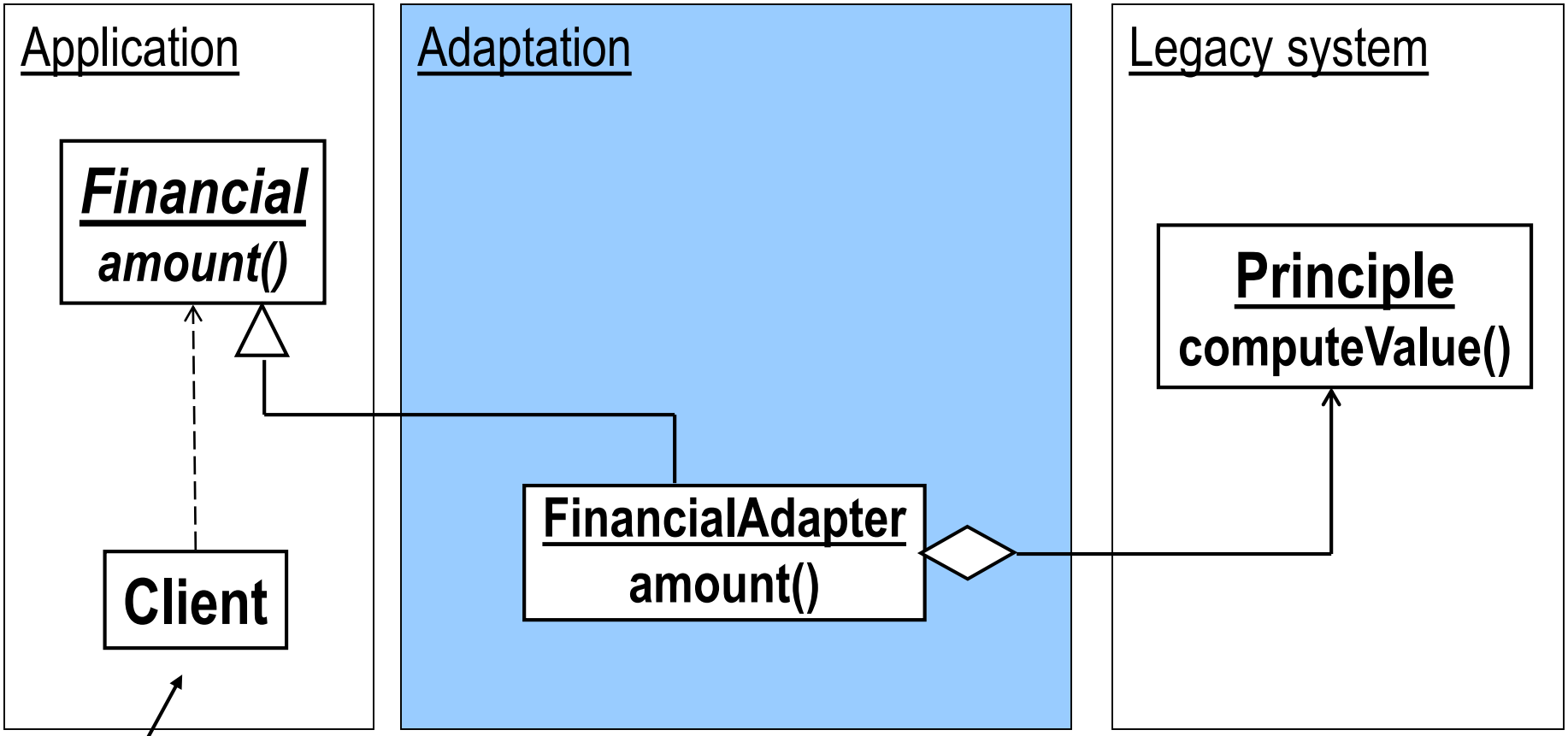


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

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

We want to separate the application as a whole from financial calculations which will be performed externally.

Adapter Design Pattern



Setup code in the client instantiates the Financial object as a FinancialAdapter instance

Code Example

```
class FinancialAdapter extends Financial
{
   Principal legacyAdaptee = null;

   // Constructor goes here . . .

   // This method uses the legacy computeValue() method
   float amount(float originalAmount, float numYears, float intRate)
   {
   return legacyAdaptee.computeValue(orginalAmount, numYears, intRate);
   }
}
```

```
executeFinanceApplication(new FinancialAdapter() );
```

Key Concept: → *Adapter* Design Pattern ←

-- to interface flexibly with external functionality.

Flyweight Design Pattern

Flyweight

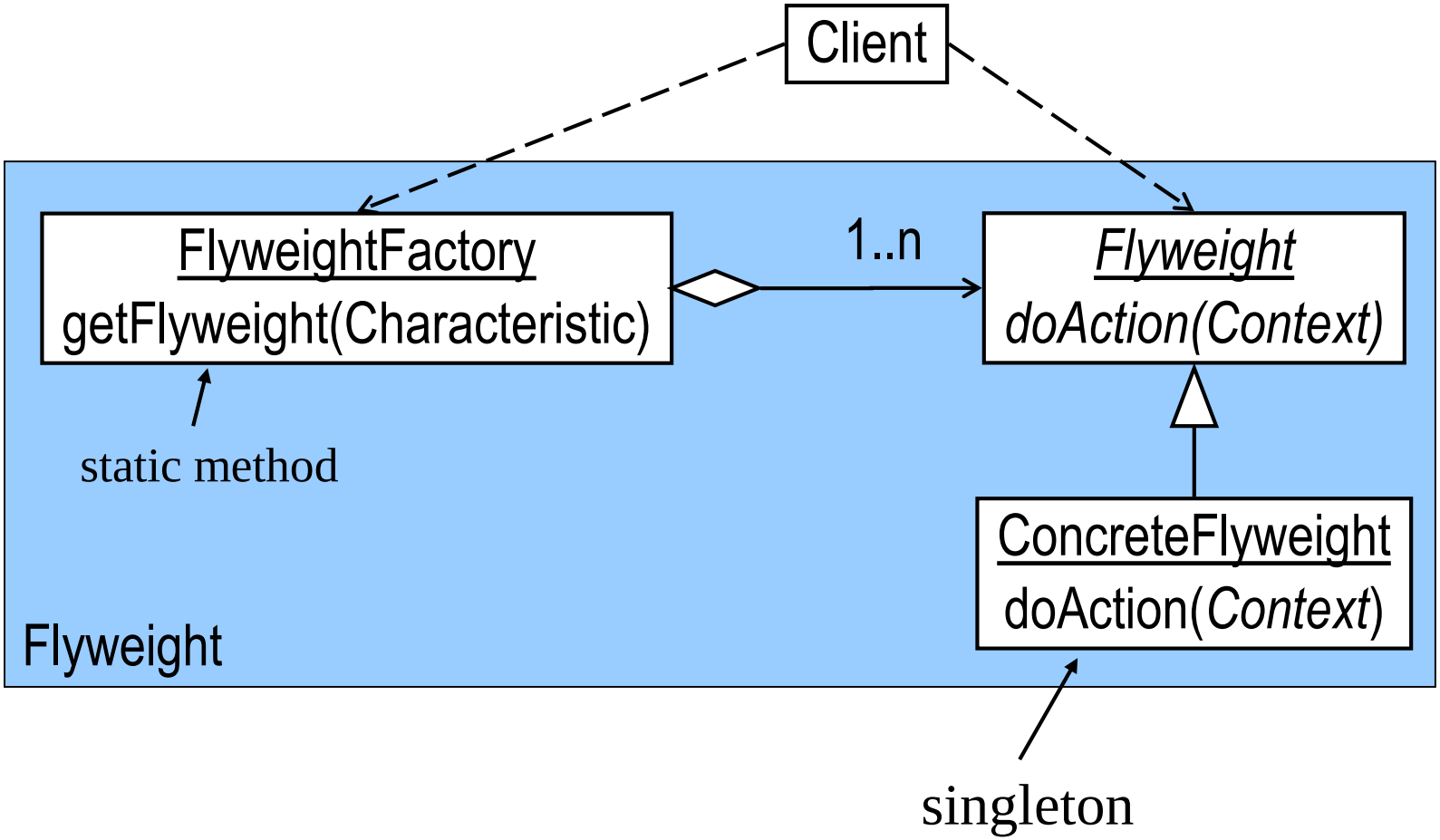
Design Purpose

Manage a large number of almost indistinguishable objects without constructing them all at once.

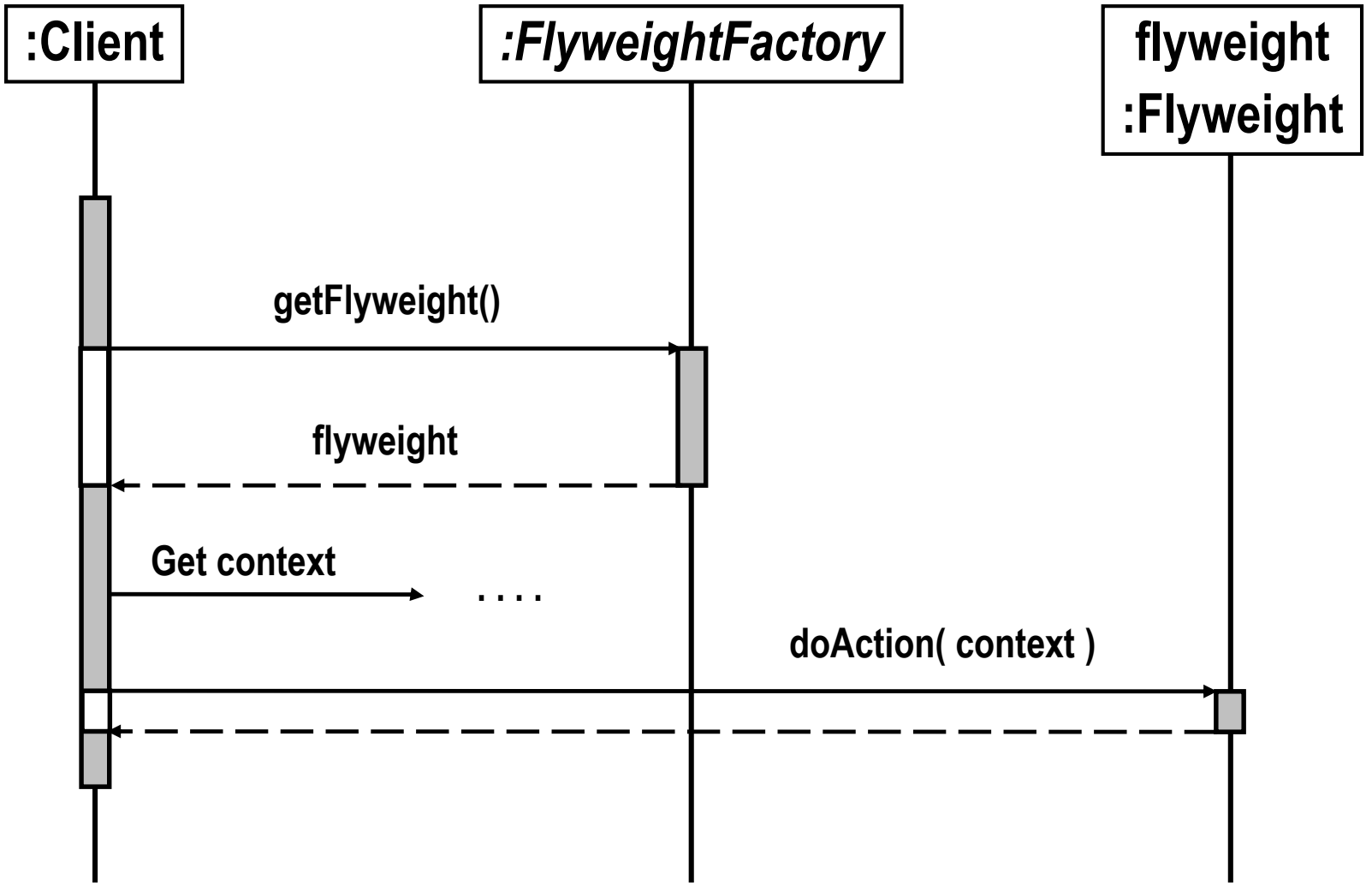
Design Pattern Summary

Share representatives for the objects; use context to obtain the effect of multiple instances.

Flyweight Class Model

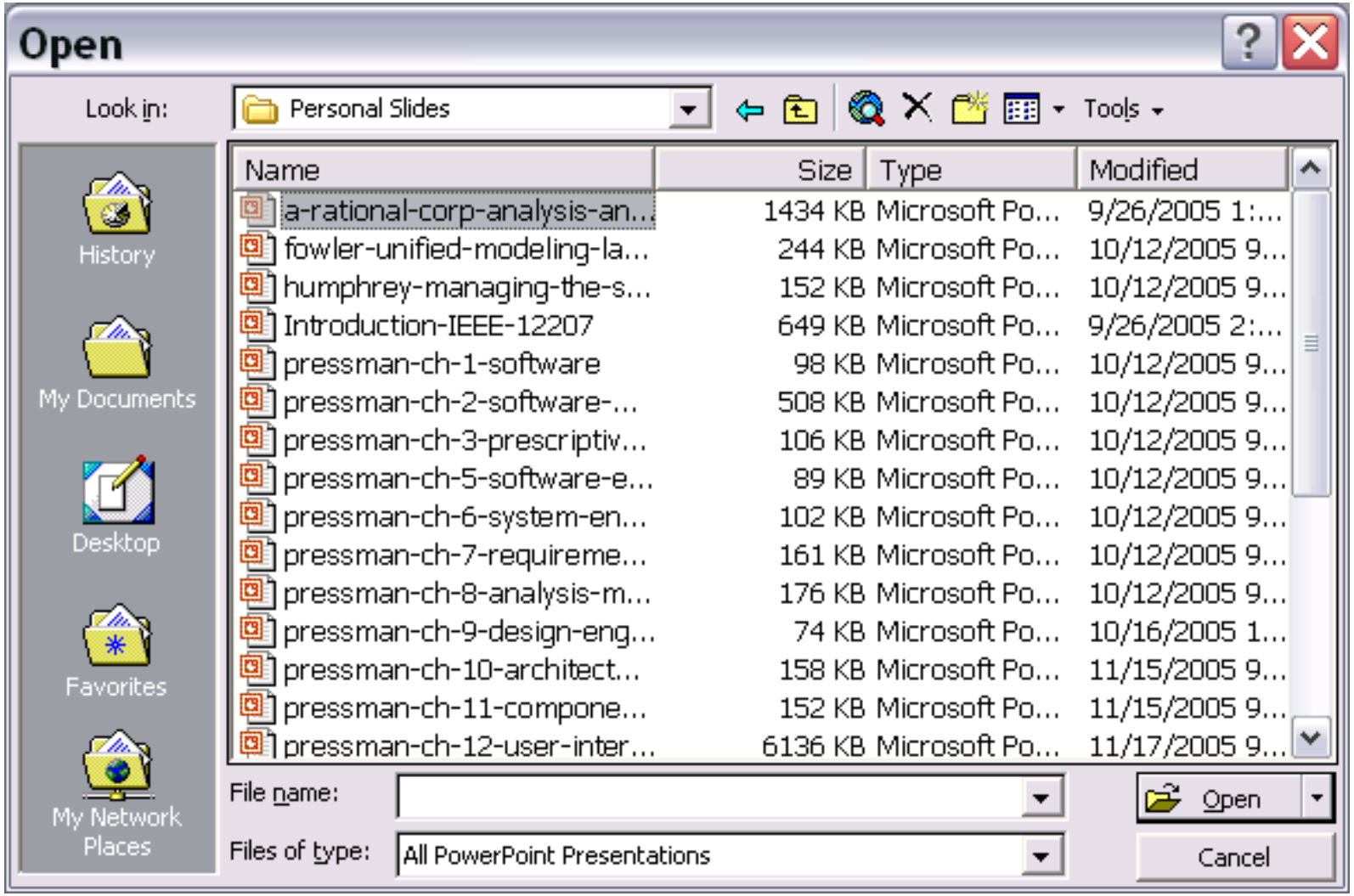


Sequence Diagram for *Flyweight*

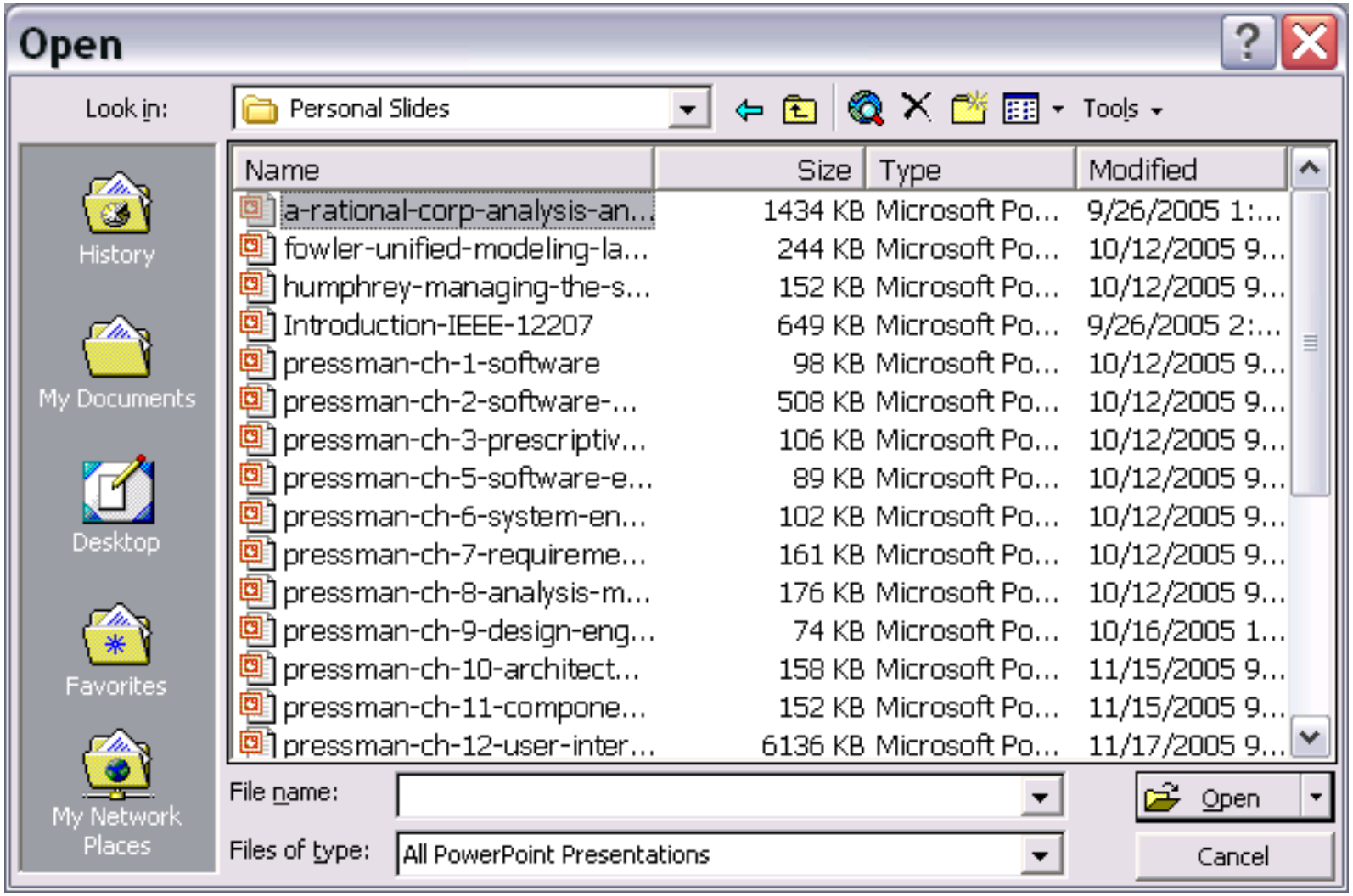


Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Example A: Window size of 15; folder contains 20 items



Example B: Window size of 15; folder contains over 500 items



Each line item had a drawing window associated with it

Design Goal At Work: → Space Efficiency ←

We want to avoid proliferating an object for every item to be displayed.

Key Concept: → *Flyweight* Design Pattern ←

-- to obtain the benefits of a large set of individual objects without efficiency penalties.

Proxy Design Pattern

Proxy

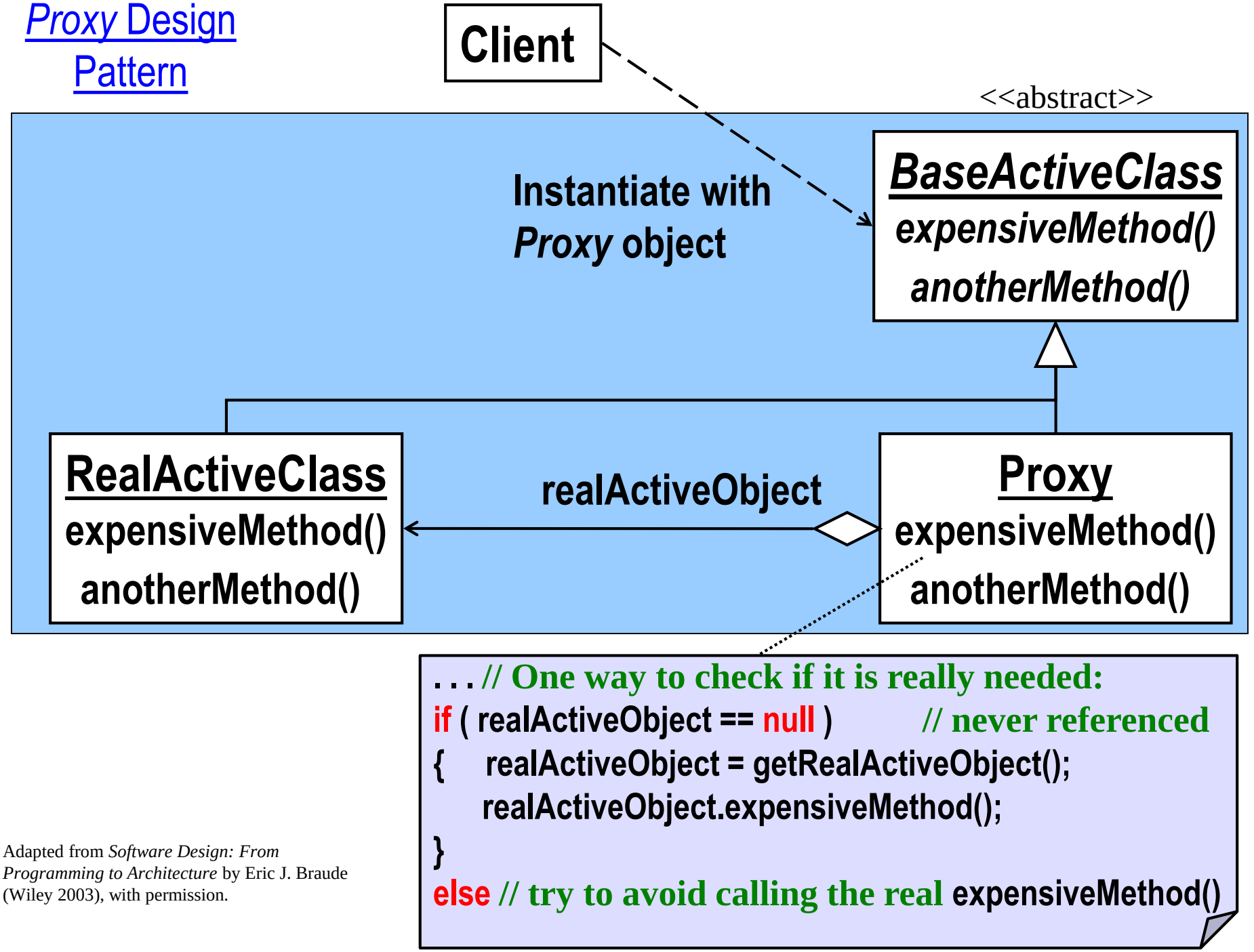
Design Purpose

Avoid the unnecessary execution of expensive functionality in a manner transparent to clients.

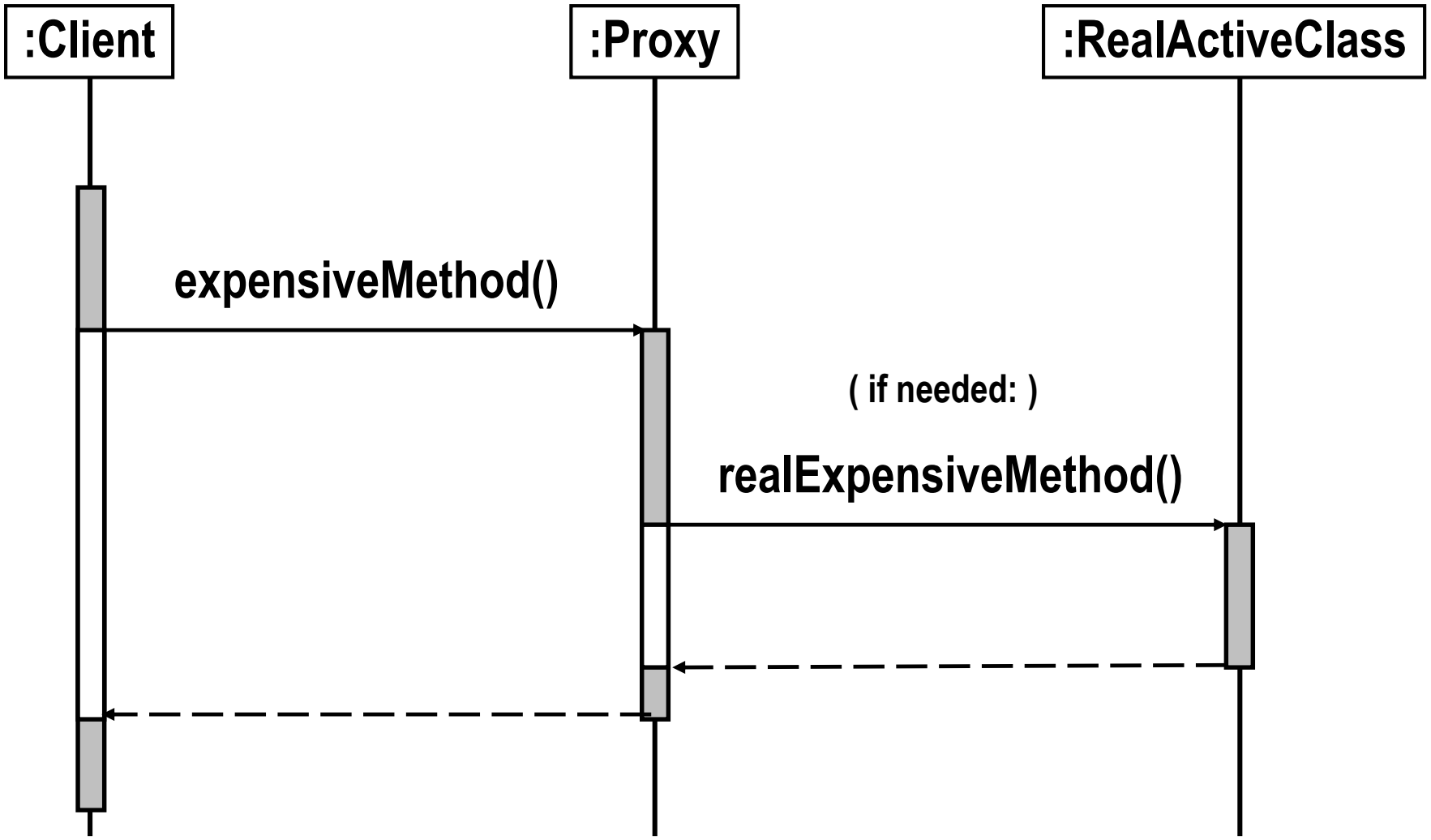
Design Pattern Summary

Interpose a substitute class which accesses the expensive functionality only when required.

Proxy Design Pattern



Sequence Diagram for *Proxy*



Console

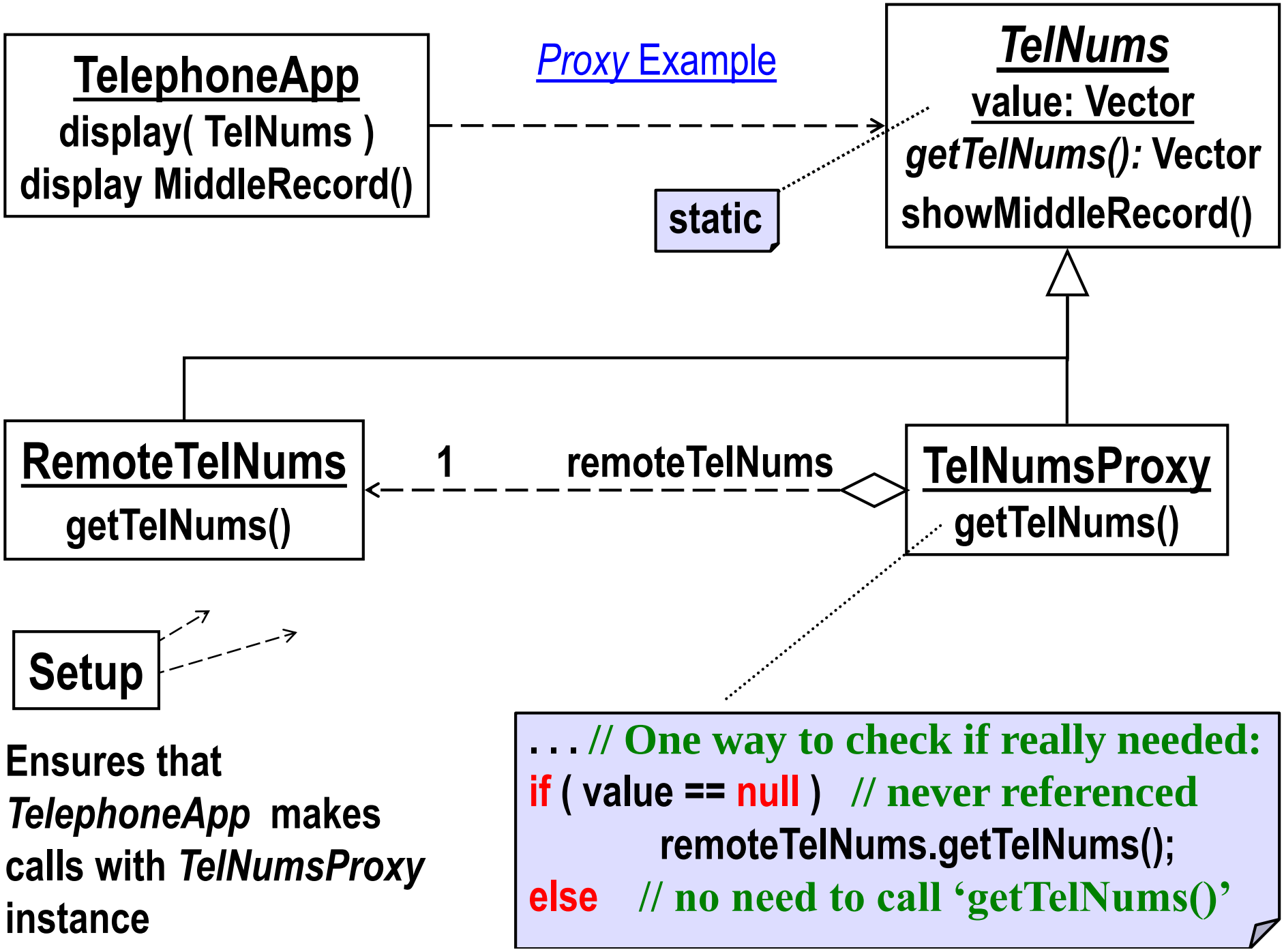
```
Please pick a command from one of the following:
quit
middle
all

> all
================== Retrieving from the Internet ==================
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all

> middle
=== No need to retrieve from the Internet ===
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all

> all
======== No need to retrieve from the Internet ========
9049031 John Dom
9049032 John Dol
9049033 John Don
9049034 John Dop
9049035 John Dor
9049036 John Dos
```

I/O of Telephone Record Proxy Example

Design Goal At Work: → *Efficiency* and *Reuse* ←

Avoid unnecessary data downloads.

Proxy Example



Ensures that *TelephoneApp* makes calls with *TelNumsProxy* instance

Key Concept: → *Proxy* Design Pattern ←

-- to call expensive or remote methods.

Summary of *Structural Design Patterns*

Structural Design Patterns **relate objects (as trees, lists etc.)**

- ***Facade*** **provides an interface to collections of objects**
- ***Decorator*** **adds to objects at runtime (in a list structure)**
- ***Composite*** **represents trees of objects**
- ***Adapter*** **simplifies the use of external functionality**
- ***Flyweight*** **gains the advantages of using multiple instances while minimizing space penalties**
- ***Proxy*** **avoids calling expensive operations unnecessarily**