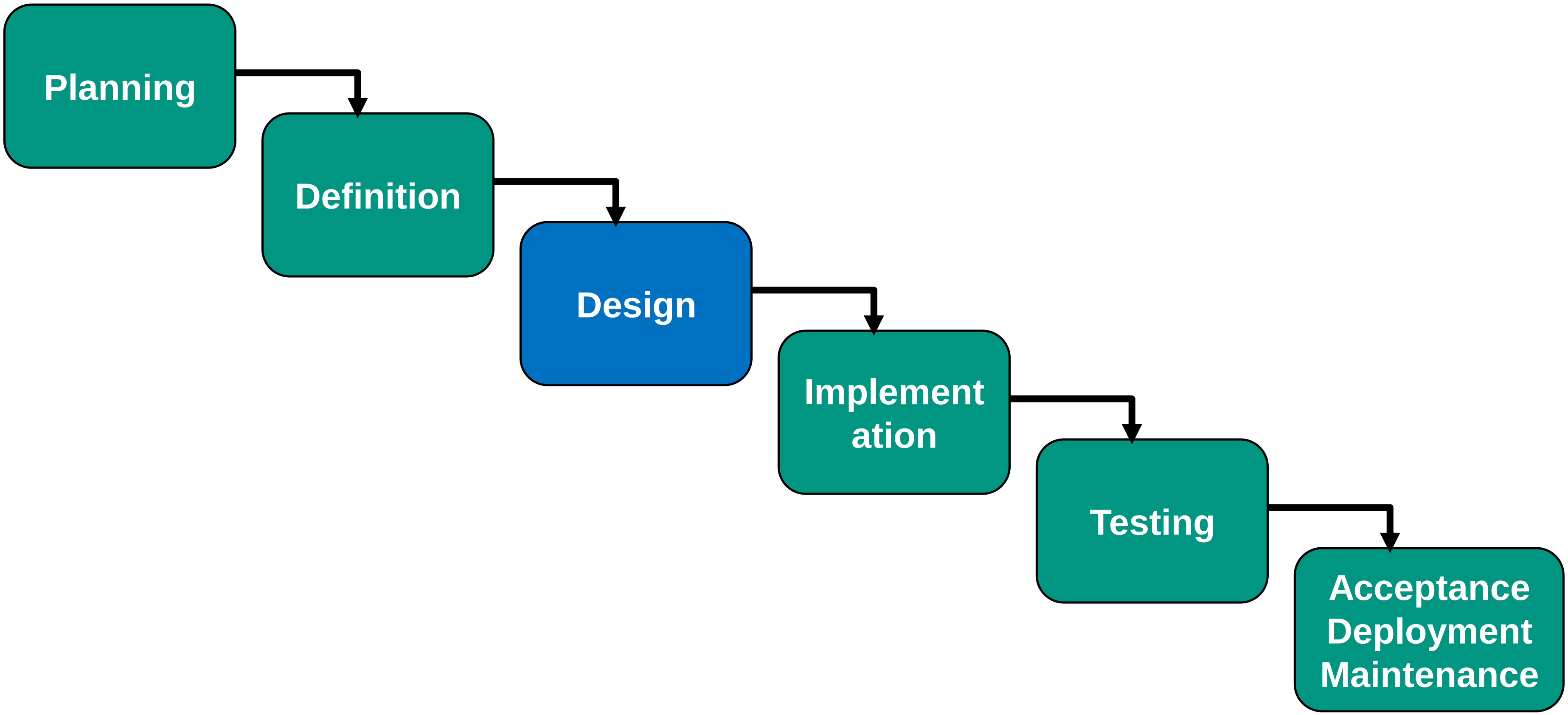
Design Patterns

CMPS115 – Summer 2017
Richard Jullig

Waterfall of Activities



Origin of design patterns: Architecture and Urban Planning

- Christopher Alexander, Architect, Berkeley
  - Book:
    A Pattern Language, Oxford University Press, New York 1977

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

- Each pattern is a
  - Three-part rule
  - Expresses relation between a context, a problem, and a solution

Sample Pattern: Web of Shopping

- **Context:**
  Physical distribution of shops and customers relative to each other

- **Conflict**:
  shops rarely placed where they best serve people’s needs and guarantee own stability

- **Resolution**: Locate a shop using these steps:
  - Identify and locate all shops offering the same service.
  - Identify and map the location of potential customers.
  - Find the biggest gap in the web of similar shops with potential customers.
  - Within the gap locate your shop next to the largest cluster of other kinds of shops.

Design Patterns

- Dfn. A software design pattern describes a family of solutions for a (recurring) software design problem.

- Design patterns are for design (programming in the large) what algorithms are for programs (programming in the small)

- Key purpose: capture design knowledge for reuse
  - prevent reinventing the wheel

History of Design Patterns

1987: Ward Cunningham and Kent Beck develop a pattern language with five Smalltalk patterns

1991: Erich Gamma and Richard Helm start jotting down catalog of patterns; first presentation at TOOLS

1991: First Patterns Workshop at OOPSLA

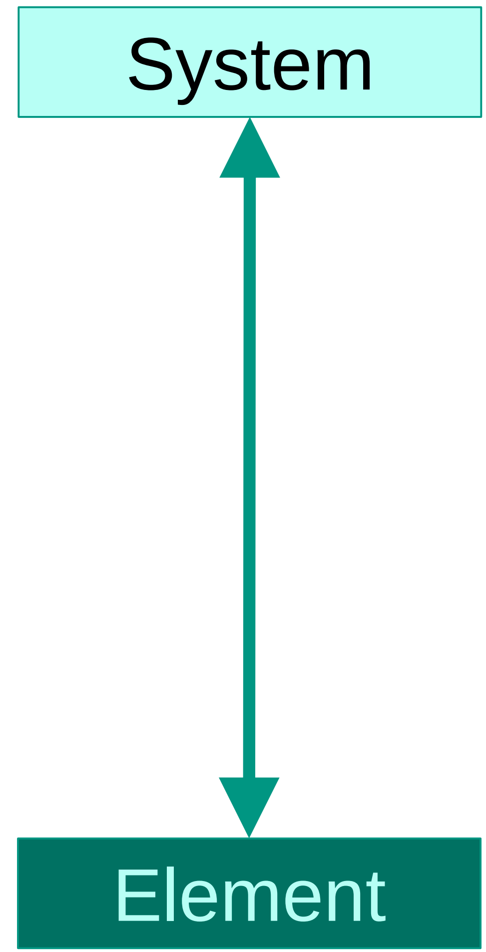
1993: Kent Beck and Grady Booch sponsor the first meeting of the Hillside Group

1994: First Pattern Languages of Programs (PLoP) conference

1994: The Gang of Four (GoF: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) publish the Design Patterns book.

Patterns at different levels

- Architectures:
  entire application/subsystem
- Design Patterns:
  design problem in particular context
  at component level
- Algorithms & Data structures



Why are Design Patterns Important?

- Patterns improve team communication
  - provide useful concepts
  - establish shared terminology
  - facilitate discussion between developers about complex designs

- Patterns capture essential concepts in (re-)usable form
  - help understand designs
  - document design, precise and concise
  - prevent architecture drift (preserve architecture integrity)
  - clarify design knowledge
  - are technology independent (for the most part)

Why are Design Patterns Important? (cont'd)

- Patterns document and advance state of the art
  - prevent reinventing the wheel
  - help less experienced designers
    - however: patterns are rather suggestions than prescriptions;
    - require adaptation to context

- Patterns help improve code quality and structure
  - patterns come with constructive and instructive examples
  - patterns have internal consistency, balance, and completeness

Design Patterns: Categories

- Decoupling Patterns
  - divide a system in several units so that
    units can be independently produced, changed, exchanged, and reused
  - Advantage: changes are local;
    change impact on system contained (minimized)
  - Many decoupling patterns contain a coupling component
    - provides the interface between loosely coupled units.
    - These coupling components are often useful for connecting separately produced units.

Decoupling Patterns (Structural Patterns)

- Adapter
- Observer
- Bridge
- Iterator
- Proxy
- Mediator

Design Patterns: Categories (2)

- Variant Patterns (Factoring Patterns)
  - extract commonalities from related units
  - provide single description of shared part
  - avoids duplication of same code in different places
  - enables uniform use of different components

Variant Patterns

- Abstract factory
- Visitor
- Builder
- Factory
- Composite
- Template
- Strategy
- Decorator

Design Patterns: Category (3)

- State management patterns
  - deal with state of objects independent of purpose

- Singleton
- Flyweight
- Memento
- Prototype
- State

Design Patterns: Categories (4)

- Control Patterns
  - help represent and manage control flow
  - ensure invocation of right methods at the right time

- Command
- Master/Worker

Design Patterns: Categories (5)

- Virtual machines
  - take programs and data as input
  - independently execute program for given data
  - implemented in software (not hardware)
- Interpreter

Design Patterns: Categories (6)

- Convenience
  - save work by customizing classes, methods, and interfaces
  - tailor access to specific context
- Façade
- Null-Object

Design Patterns: Categories (7)

Scope	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

- Pattern classification by GoF (Gang of Four): by scope and purpose

Combining Patterns

- Patterns often occur in combination

- For a fun but complex example, see
  - Chapter 12 in Head First Design Patterns
    - Excellent resource (strongly suggest you take a look at it)
    - (on Piazza>Resources>Reading Material)
    - Freeman_Head_First_Design_Patterns.pdf
  - read, understand, work problems/questions
  - maybe even: implement

- Combines
  - Adapter
  - Decorator
  - Abstract Factory
  - Iterator
  - Observer

References

- Gamma, Helm, Johnson, Vlissides. Design Patterns (1995)
- Freeman. Head First Design Patterns (2004)
  - Examples in Java
- Martin. Agile Principles, Patterns and Practices in C#
  - Examples in C#
- See Piazza > Resources > Reading Material

Design Patterns: Decoupling Patterns

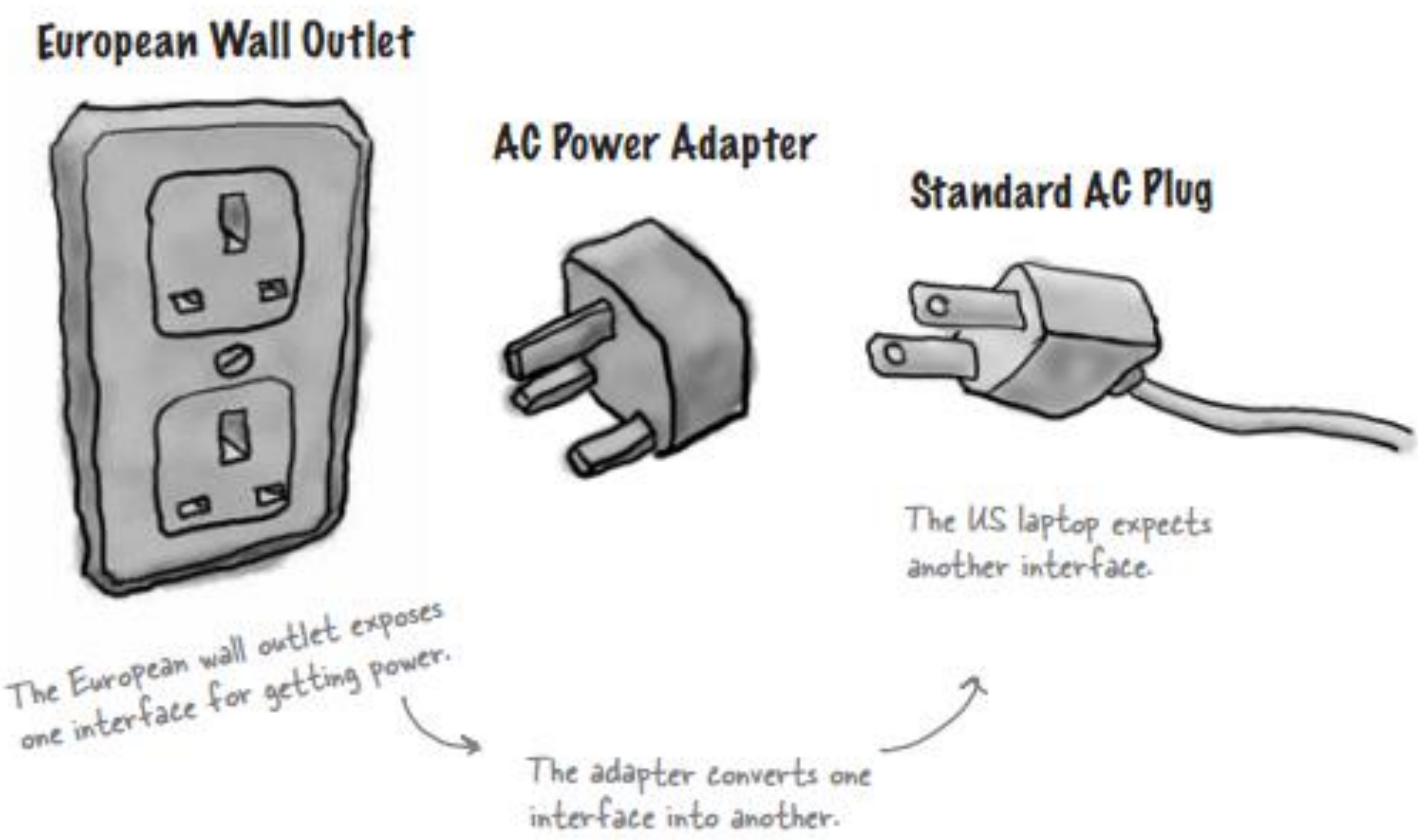
CMPS115 – Summer 2017
Richard Jullig

Adapter

- Purpose
  - Adapt the interface of a class to the one expected by clients
  - Allows classes to interoperate across incompatible interfaces

- Synonyms
  - wrapper

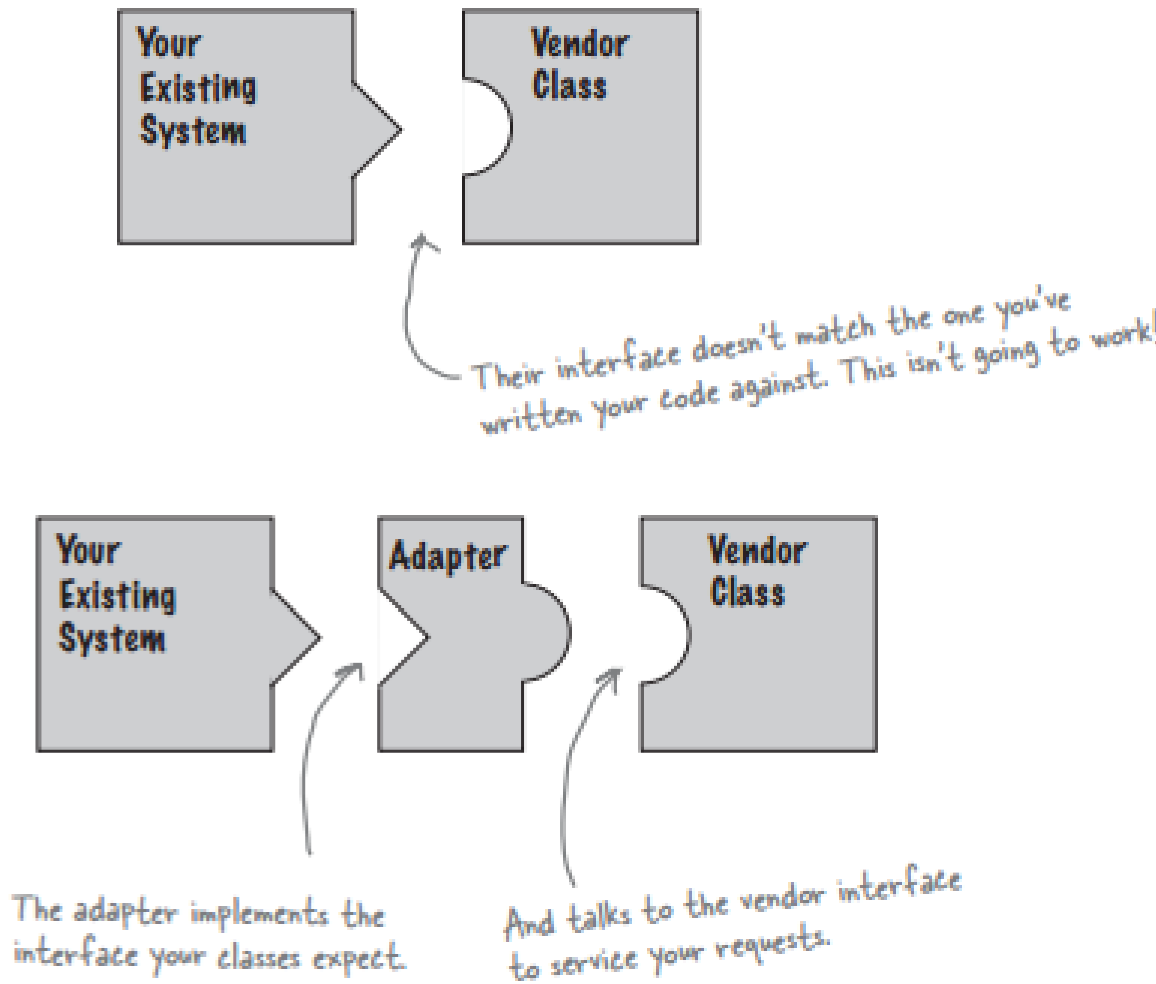
Physical Adapter Example



- Note: instead of *European* read *British*
  - Germany, France have other (mutually incompatible) standards
  - As of 2016, the UK is no longer part of the EU

Software Adapter: Problem context

- Your system assumes an interface not directly supported by vendor
- You have reason not to change your system
- You cannot change the vendor's interface
- Reify the necessary change as an adapter



Software Adapter – Example (1)

```
public interface Duck {
    public void quack();
    public void fly();
}
```

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing.

- Duck interface
- Concrete Duck class (Mallard)

Software Adapter – Example (2)

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble.

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions.

- Turkey interface
- Concrete turkey class (Wild turkey)

Software Adapter – Example (3)

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

- Substituting turkey for duck
- For Thanksgiving: try the reverse

- Note:
  dependency injection (constructor)

Software Adapter – Example (4)

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says...");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says...");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck...

and a Turkey.

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck.

Then, let's test the Turkey: make it gobble, make it fly.

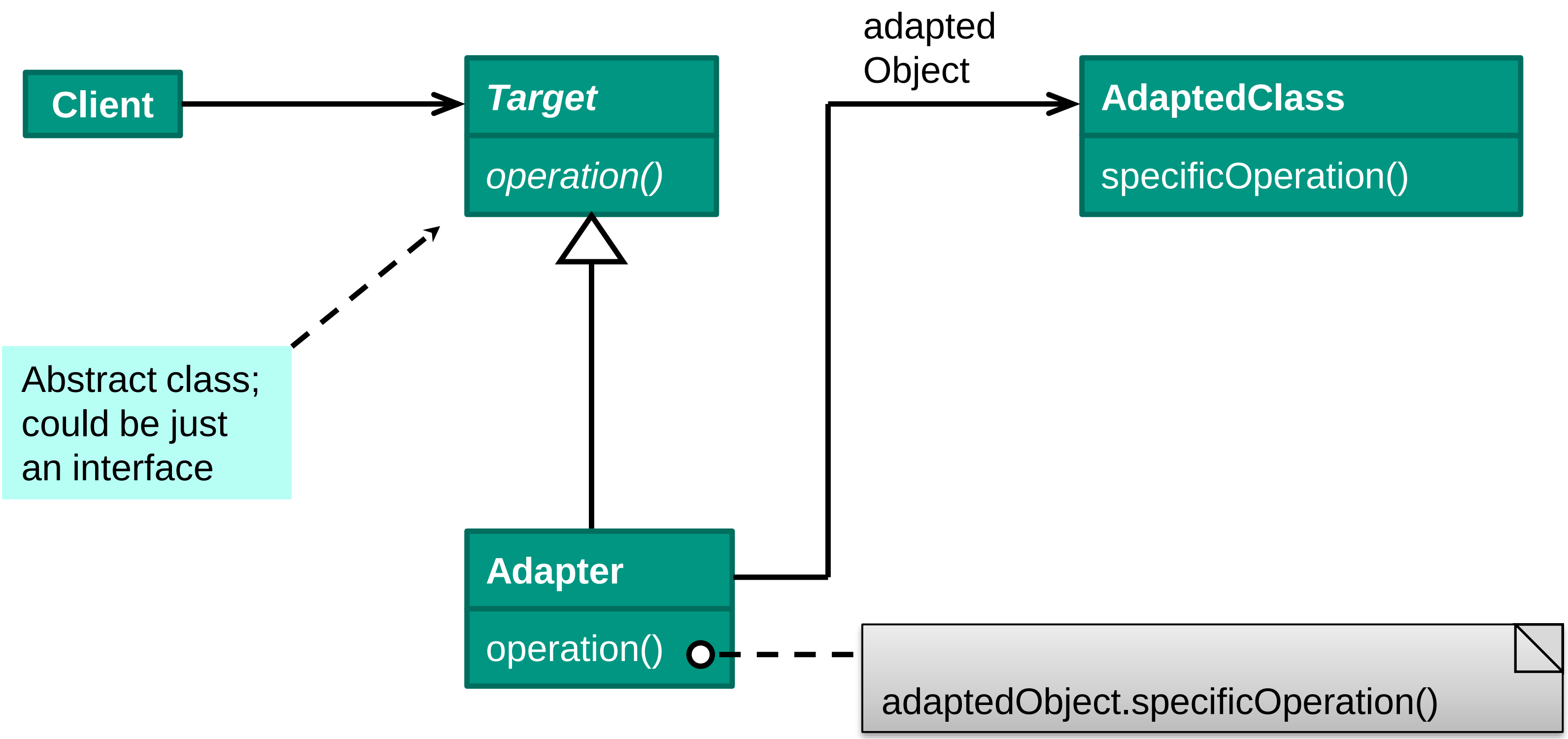
Now let's test the duck by calling the testDuck() method, which expects a Duck object.

Now the big test: we try to pass off the turkey as a duck...

Here's our testDuck() method; it gets a duck and calls its quack() and fly() methods.

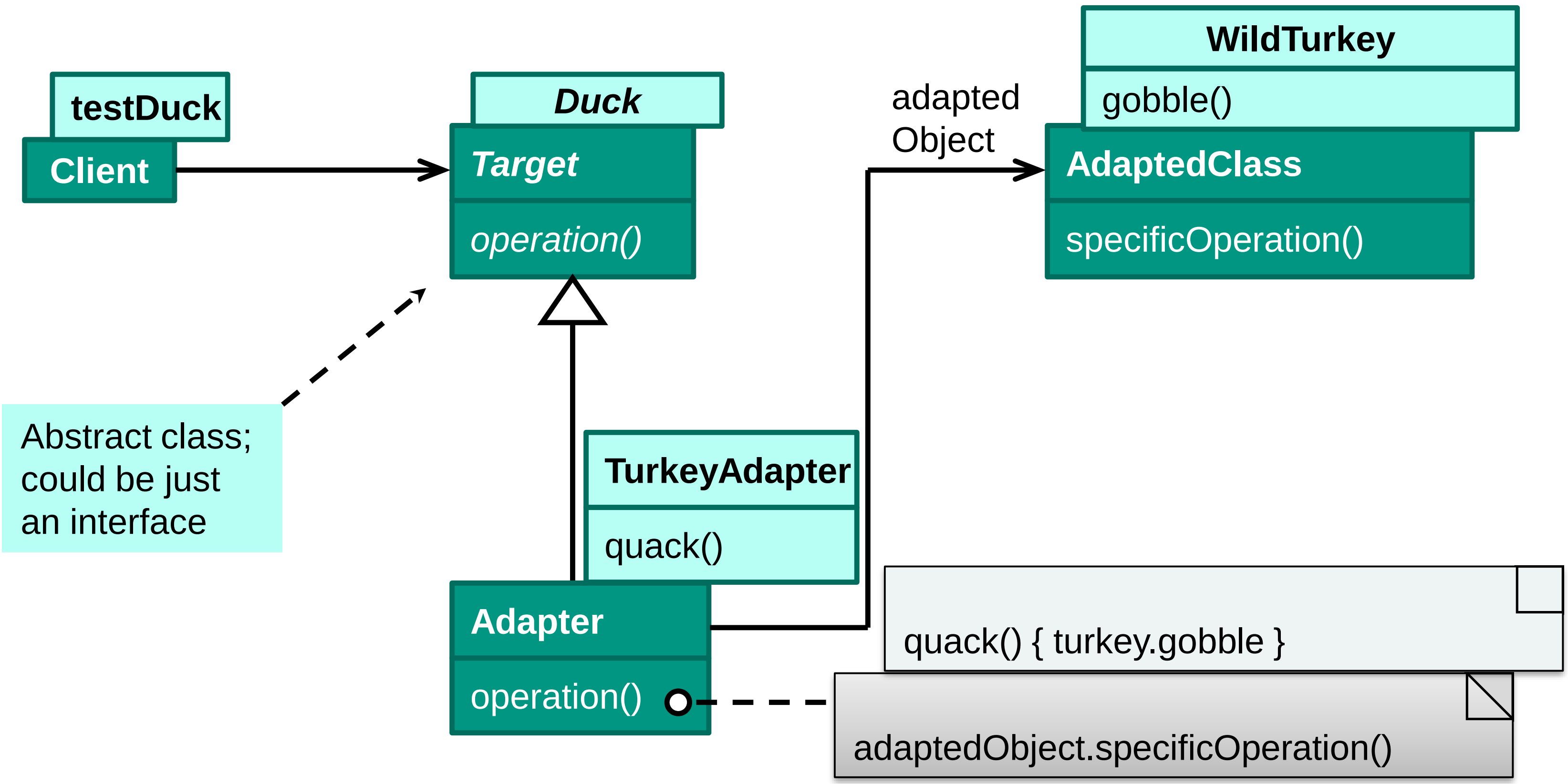
Adapter: Structure (1)

- Object adapter (single inheritance)



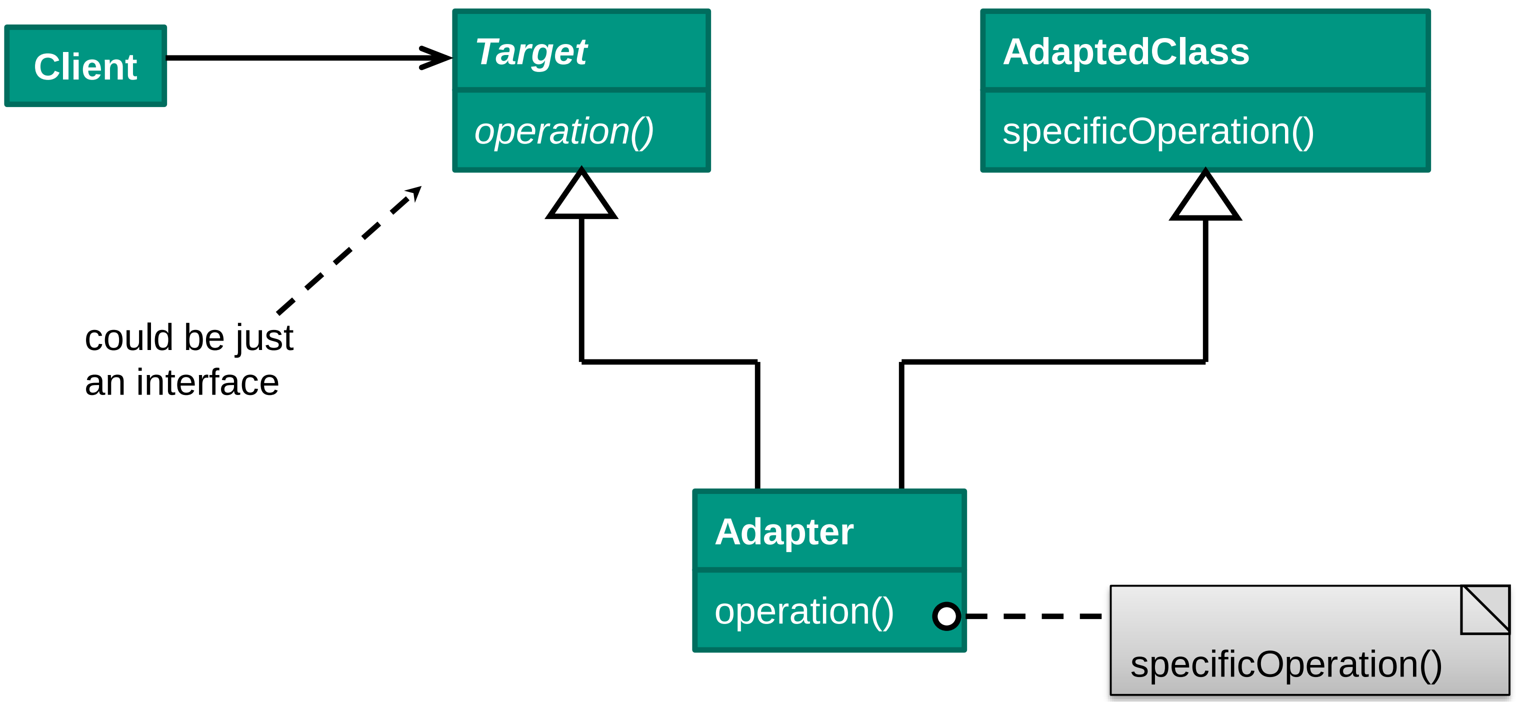
Turkey Adapter: Structure

- Object adapter (single inheritance)



Adapter: Structure (2)

- Class adapter (with multiple inheritance)



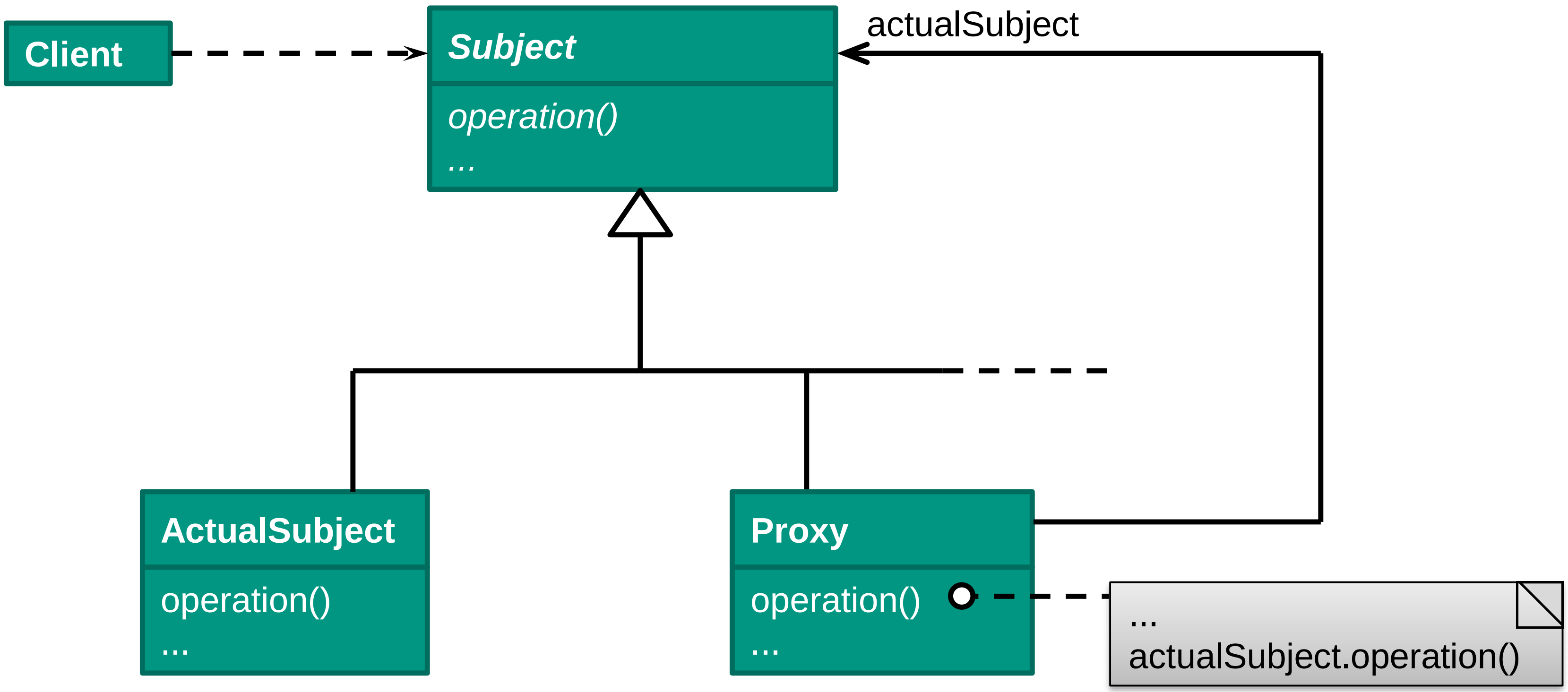
Adapter: Applicability

- When there is a mismatch between the interface of a class and the interface expected by the client (and neither one can/should be changed)
- When we want to create a reusable class which can work with classes with (somewhat) incompatible interfaces

Proxy

- Purpose
  - Control access to object by means of “stand-in”
- Synonym: Surrogate

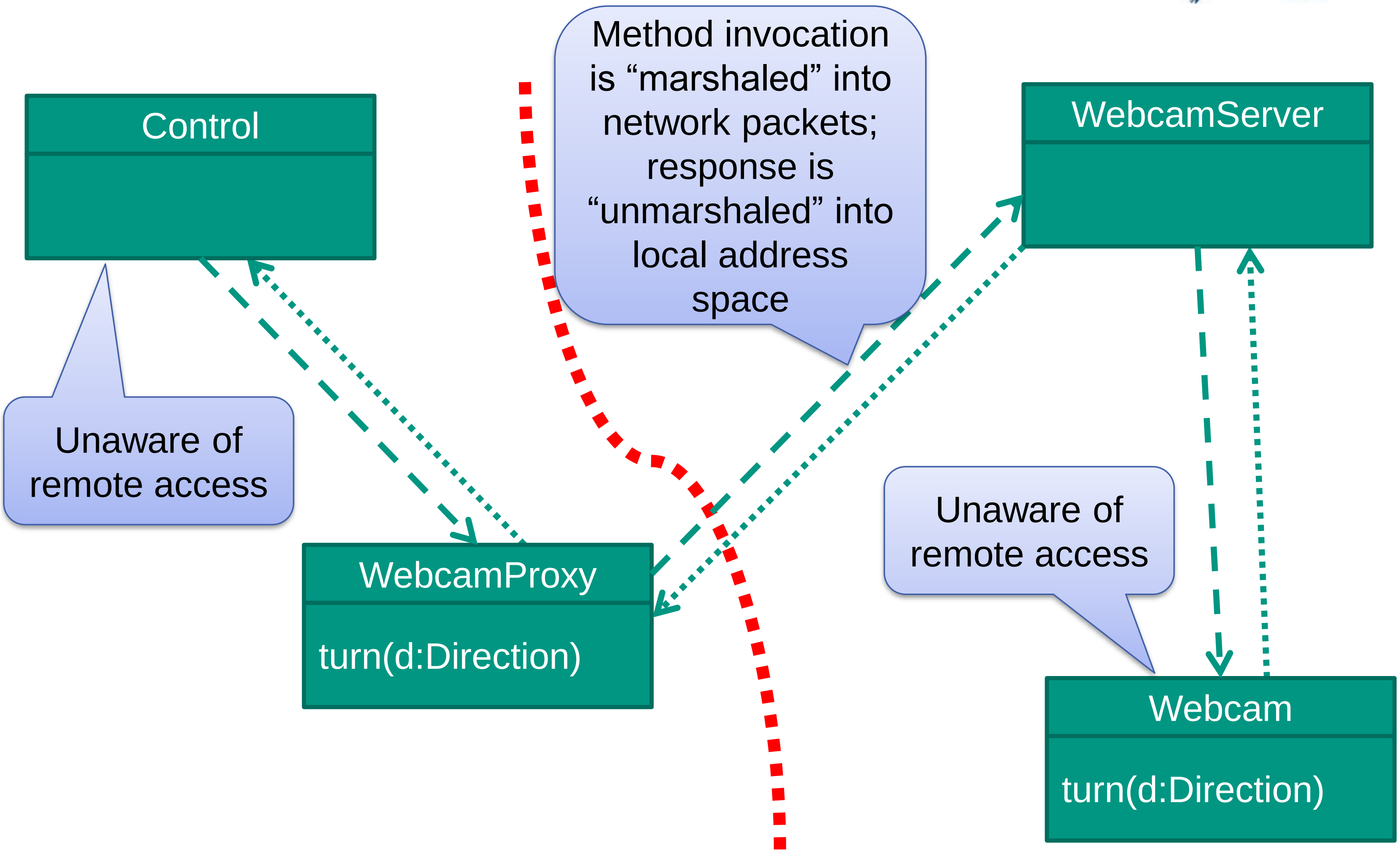
Proxy: Structure



Proxy: Applicability (1)

- Need for more sophisticated, adaptable reference than bare pointer is needed
- Typical examples
  - Protocol-keeping proxy
    - Counts references to actual object to support garbage collection
    - May maintain other access information
  - Caching proxy
    - Supports storage management, e.g.
    - Load actual object when (first) needed
    - May maintain buffer for storage management of multiple objects
  - Remote proxy
    - local stand-in for object in different address space
    - Manages “marshaling” and “unmarshaling” of method invocation and responses

Remote Proxy: Webcam Example



Proxy: Applicability

- Virtual proxy
  - generates “expensive” objects when needed (delayed generation and/or loading)
- Firewall
  - Controls access to actual object; manages access rights
- Synchronization proxy
  - Coordinates concurrent access to an object
- Decorator
  - Adds additional capabilities to existing object
  - Decorators can be decorated (decoration cascade)

Read more: Head First Design Patterns, page 460 ff.

Observer

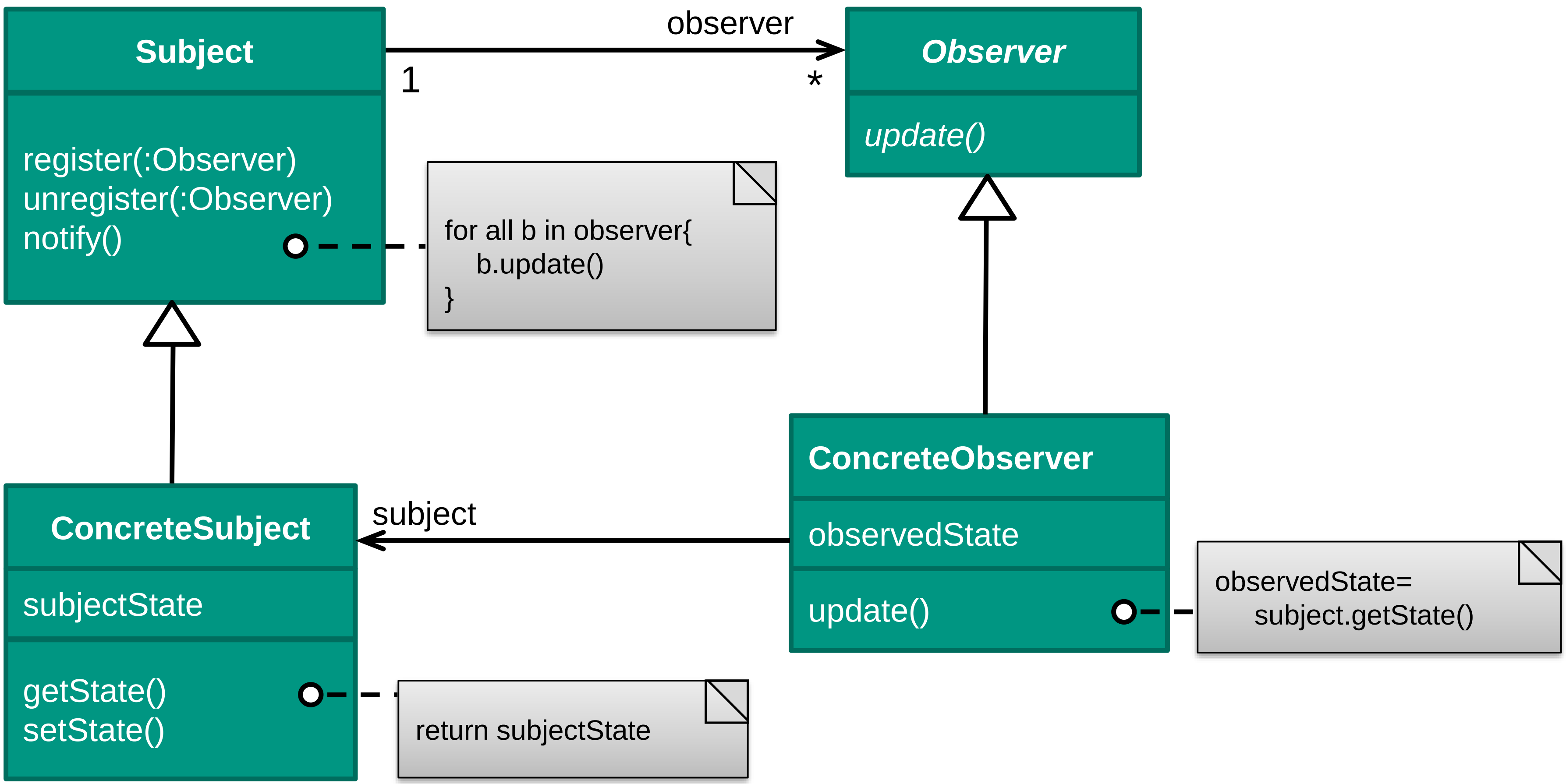
- Purpose/Intent
  - Define a one-to-many dependency between (one) observed object and (many) observing objects. When the state of the observed object changes the observers are notified and updated automatically.

- Synonyms
  - Dependents
  - Publish-subscribe

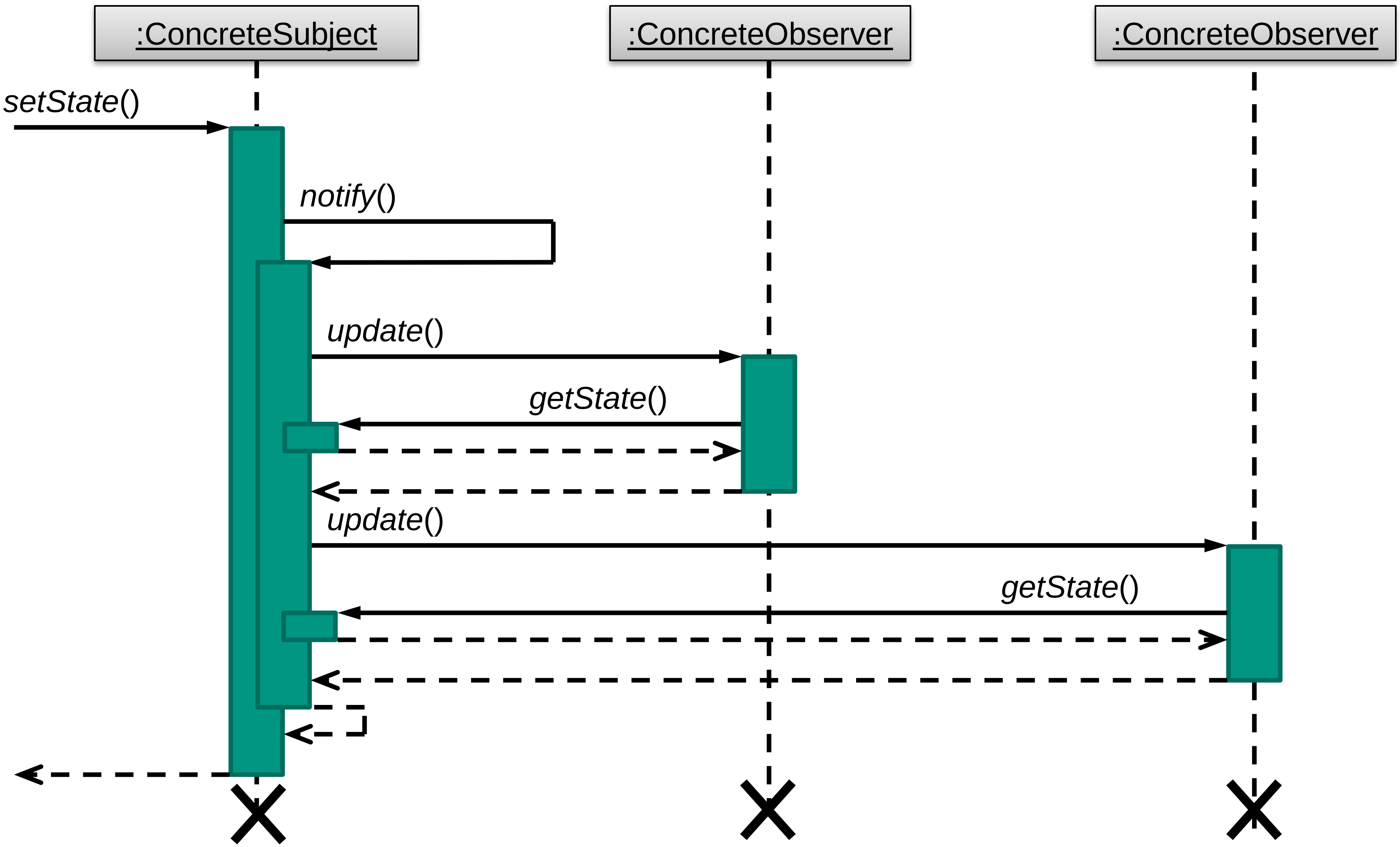
Observer: Motivation

- The state of observed and observing objects must be kept consistent (observers must be up-to-date).
- Nevertheless, observed and observing objects should not be tightly coupled. Loose coupling allows for independent evolution and reuse.

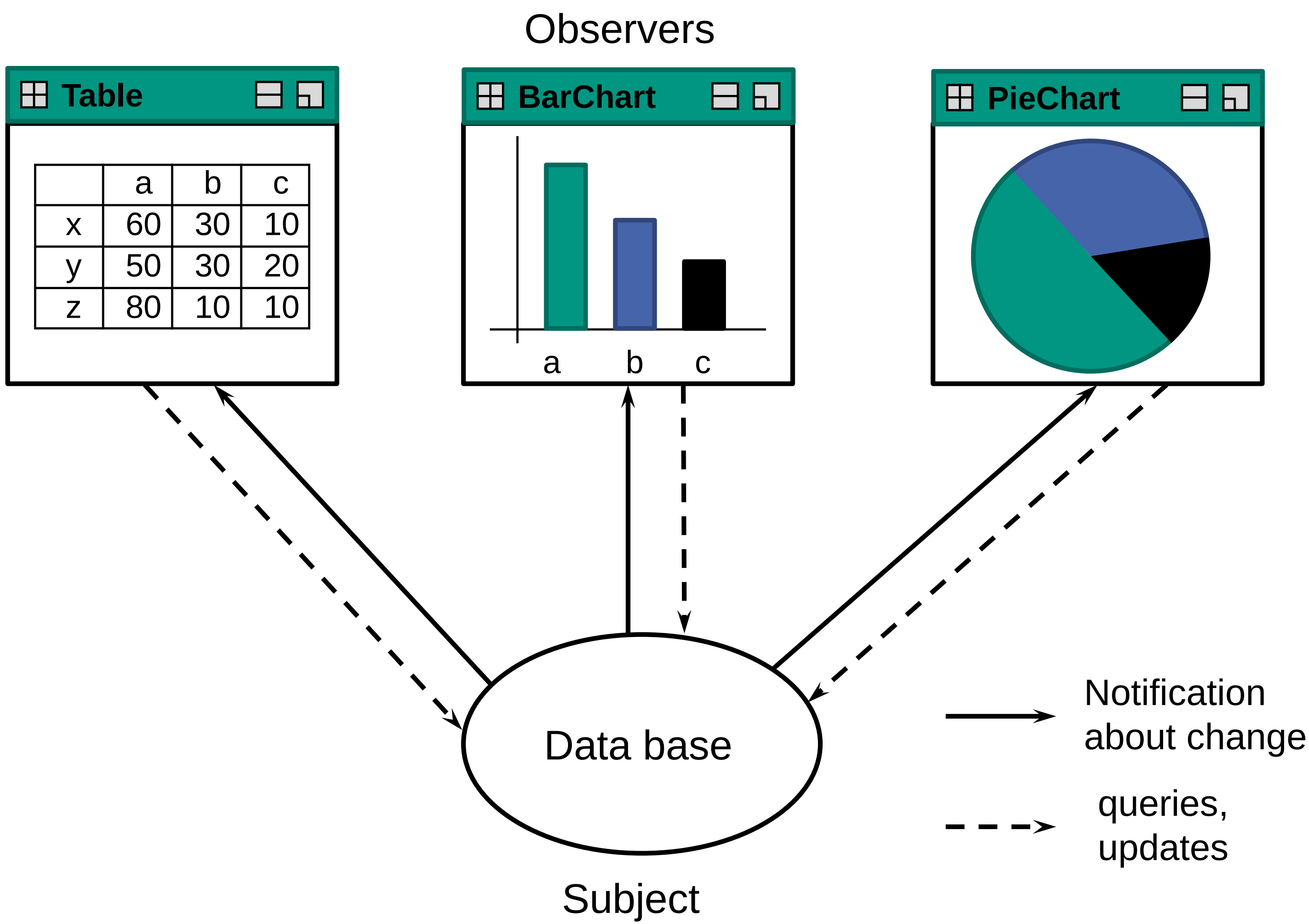
Observer: Structure



Observer: Interaction diagram (Sequence diagram)

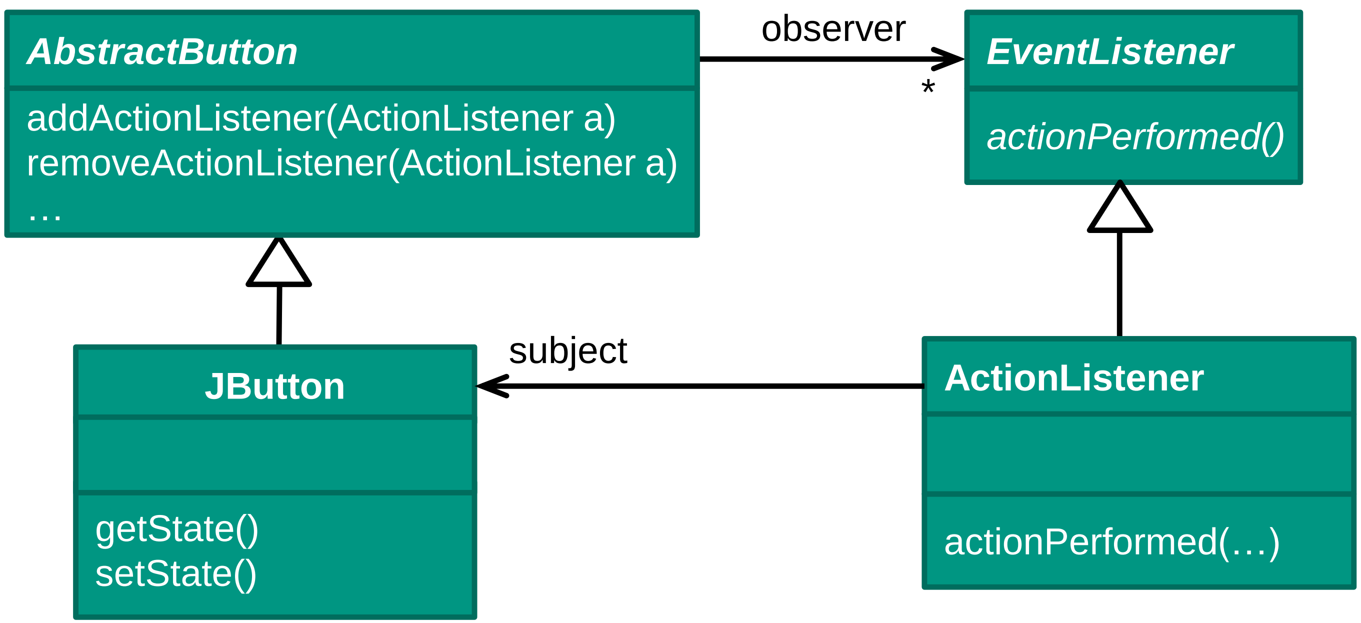


Observer: Example



Observer: Example in Java

- In Java (and other languages, e.g. Javascript): events are handled by observers:



Observer: Applicability

- When changes to the state of one object need to be propagated to the state changes in other objects but it is not known which object or how many.
- When one object needs to notify other objects without knowing anything about the other objects.
- When one basic fact can be interpreted or presented in different ways. Keeping these aspects separate allows for independent reuse and evolution.

Observer: Consequences (1)

- Subjects and observers can be reused independently.
- Observers can be added and removed without changes to the subject or other observers.
- Subject and observers are coupled by notifications.
- Subject and observers belong to different layers in uses-hierarchy (without cycles)
  - Observers use subject, but not vice versa.

Observer: Consequences (2)

- Automatic broadcast of change notifications
- Observers decide whether or not to respond to notification.
- Updates may require non-obvious effort
  - notification may lead to cascade of updates
  - notification indicates only that subject’s state changed, not how
    - protocol could be extended to provide information about nature of change

Observer: Implementation (1)

- When observing more than one subject, provide subject as parameter in update:
  - `update(s :Subject)`

- Triggering update
  - `setState()` calls `notify()`, or
  - state-changing clients call `notify()` directly
    - allows clients to batch state changes (avoid spurious updates)

Observer: Implementation (2)

- Subject must be in consistent state before notifying observers
  - When a subject subclass calls inherited operations, notifications may be sent before the subclass object is in a consistent state
  - alternative: use template method for update
    (see Gamma et al., Design Patterns)
- Update: Push or Pull?
  - Pull model:
    Observers retrieve data from subject (could be inefficient)
  - Push model:
    Subject sends data in update message
    (could interfere with reusability)

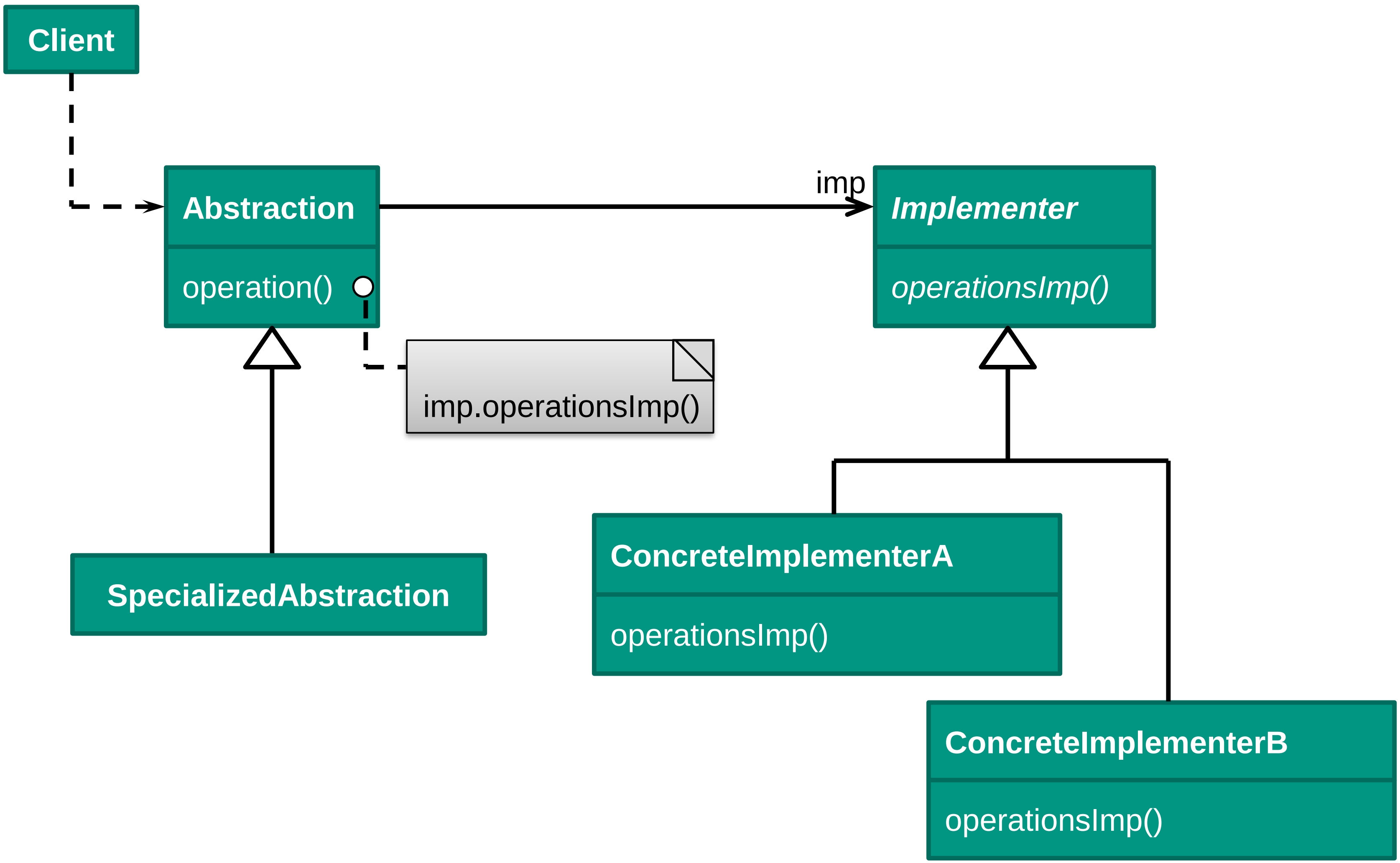
Observer: Implementation (3)

- Change Manager between subject and observers
  - Maps Subject to Observers; offers interface for managing mapping
  - Updates upon request of Subject all dependent Observers
  - Avoids multiple updates
  - Encapsulates complex update semantics

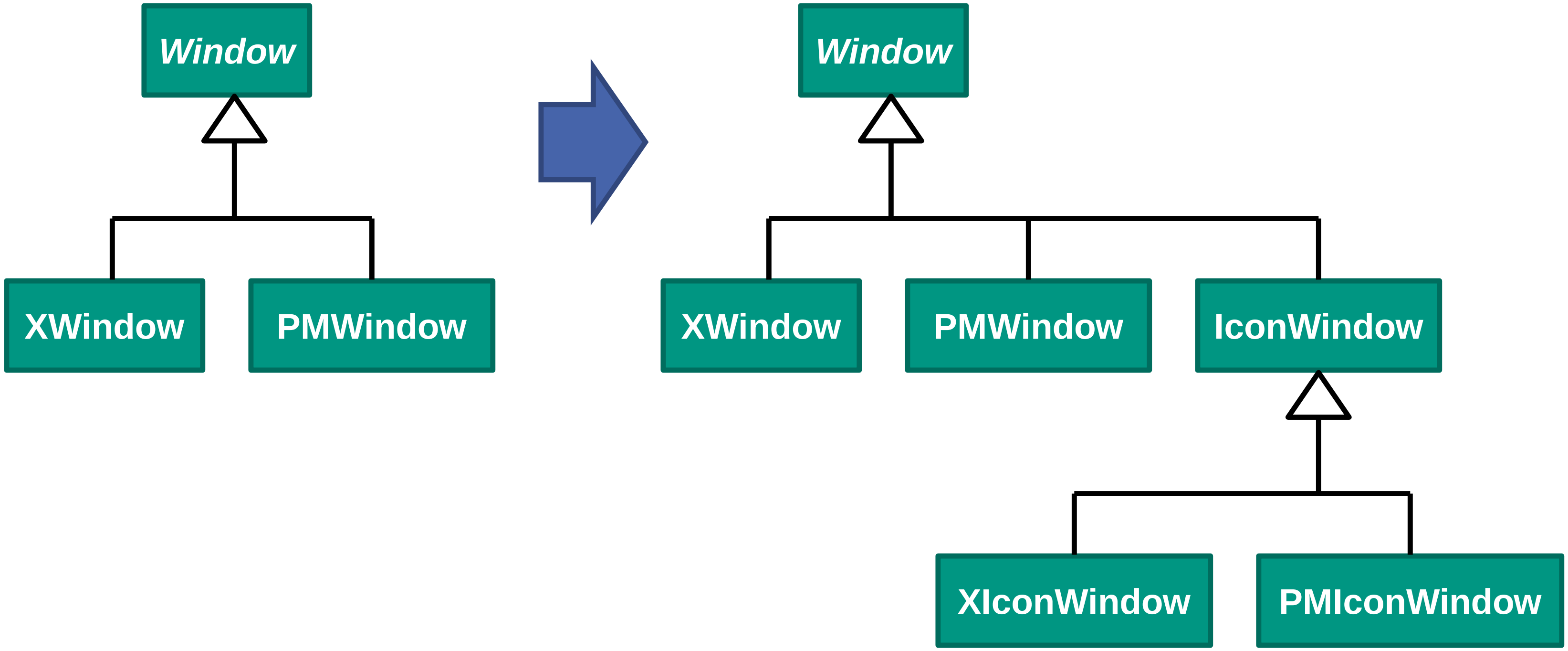
Bridge

- Purpose
  - Decouple abstraction from implementation
  - Make both independently variable
- Synonym: Handle, Body

Bridge: Structure

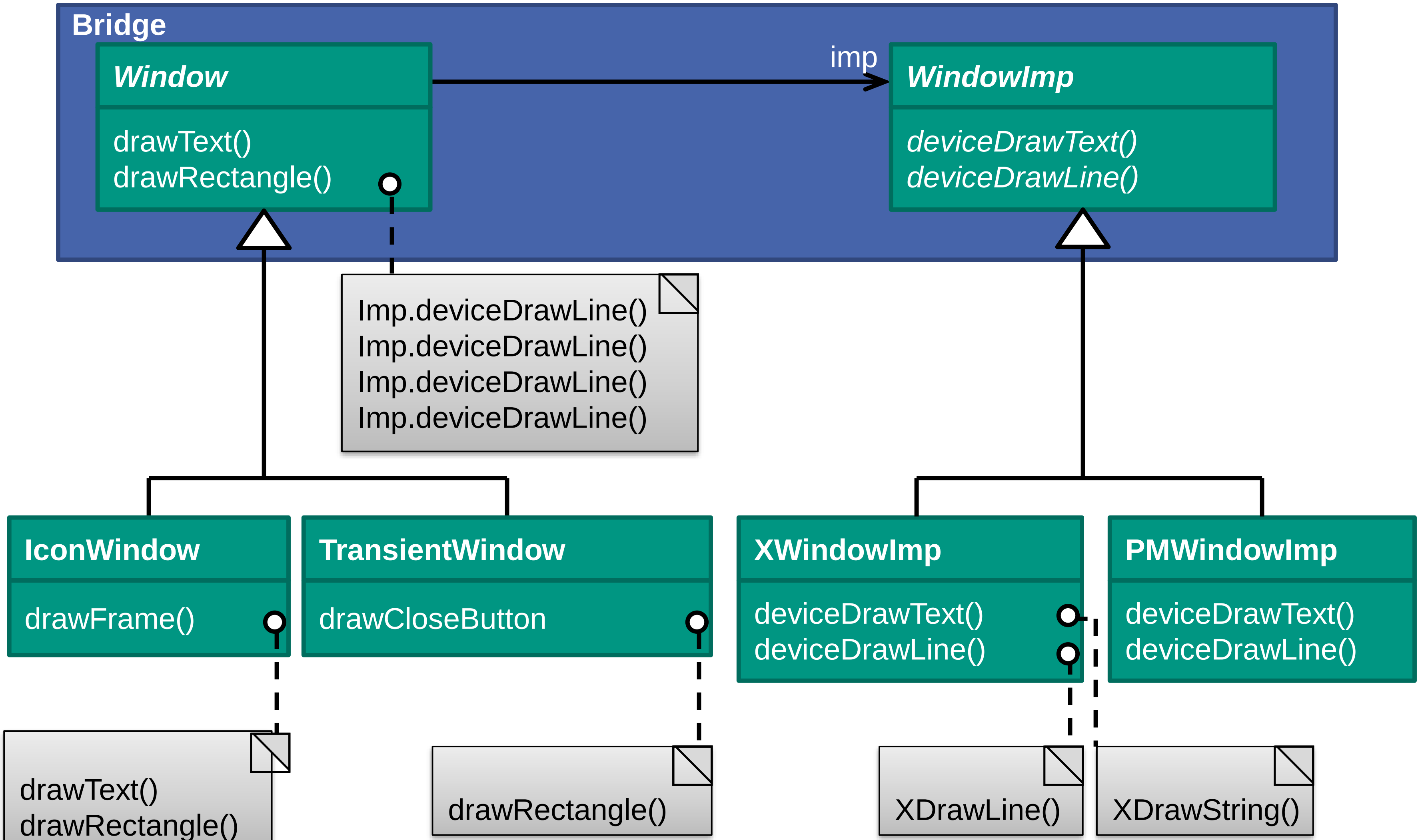


Bridge: Motivating example

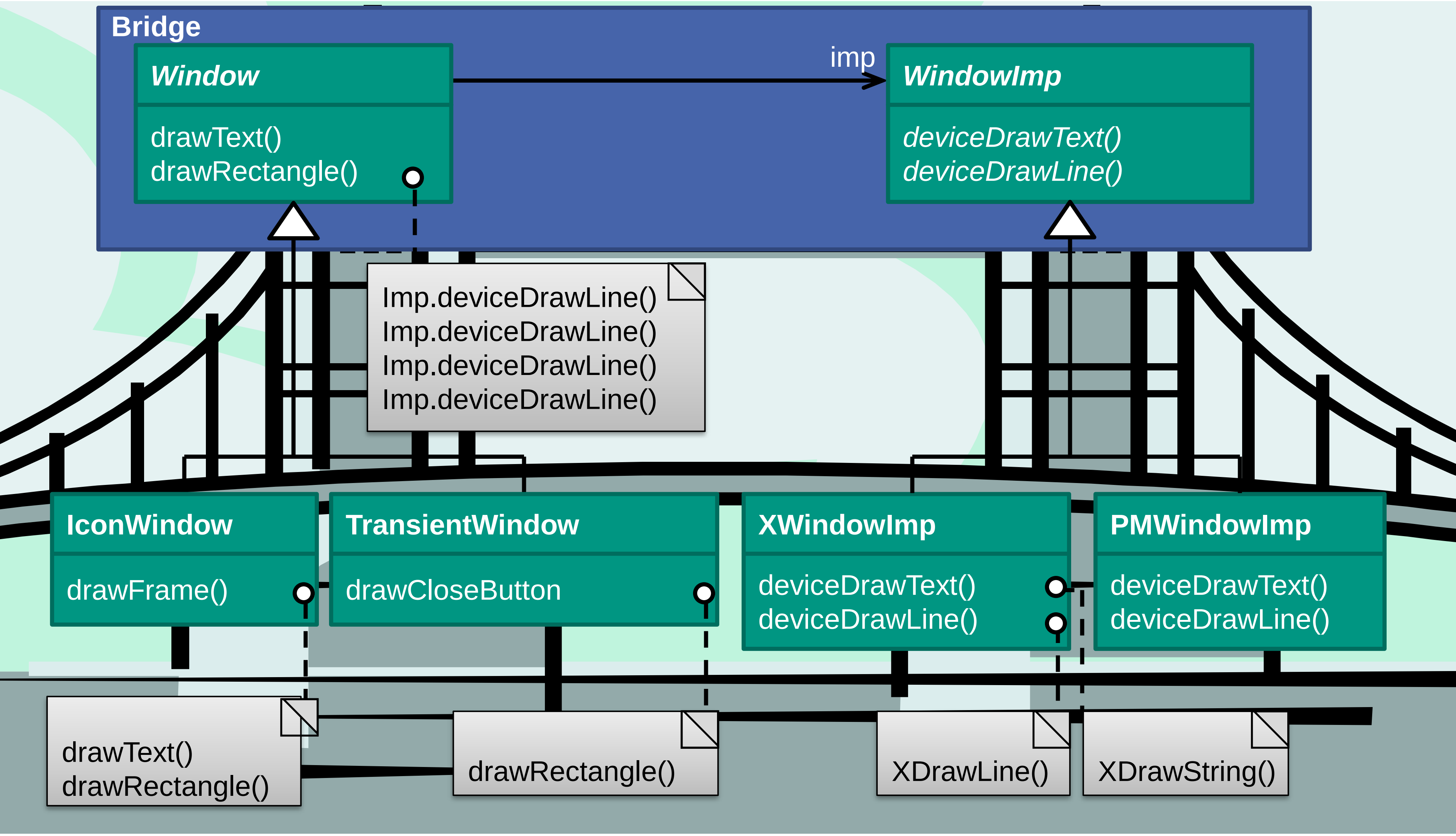


What is cumbersome and ugly here?

Bridge: Example



Bridge: Example



Bridge: Application example

- Installation of multiple Java versions simultaneously
- Version to be used can be chosen before each application execution.

Bridge: applicability (1)

- Avoid permanent link between abstraction and implementation
- Abstraction and implementation need to be extensible to subclasses
- Changes in implementation should not propagate to changes in clients

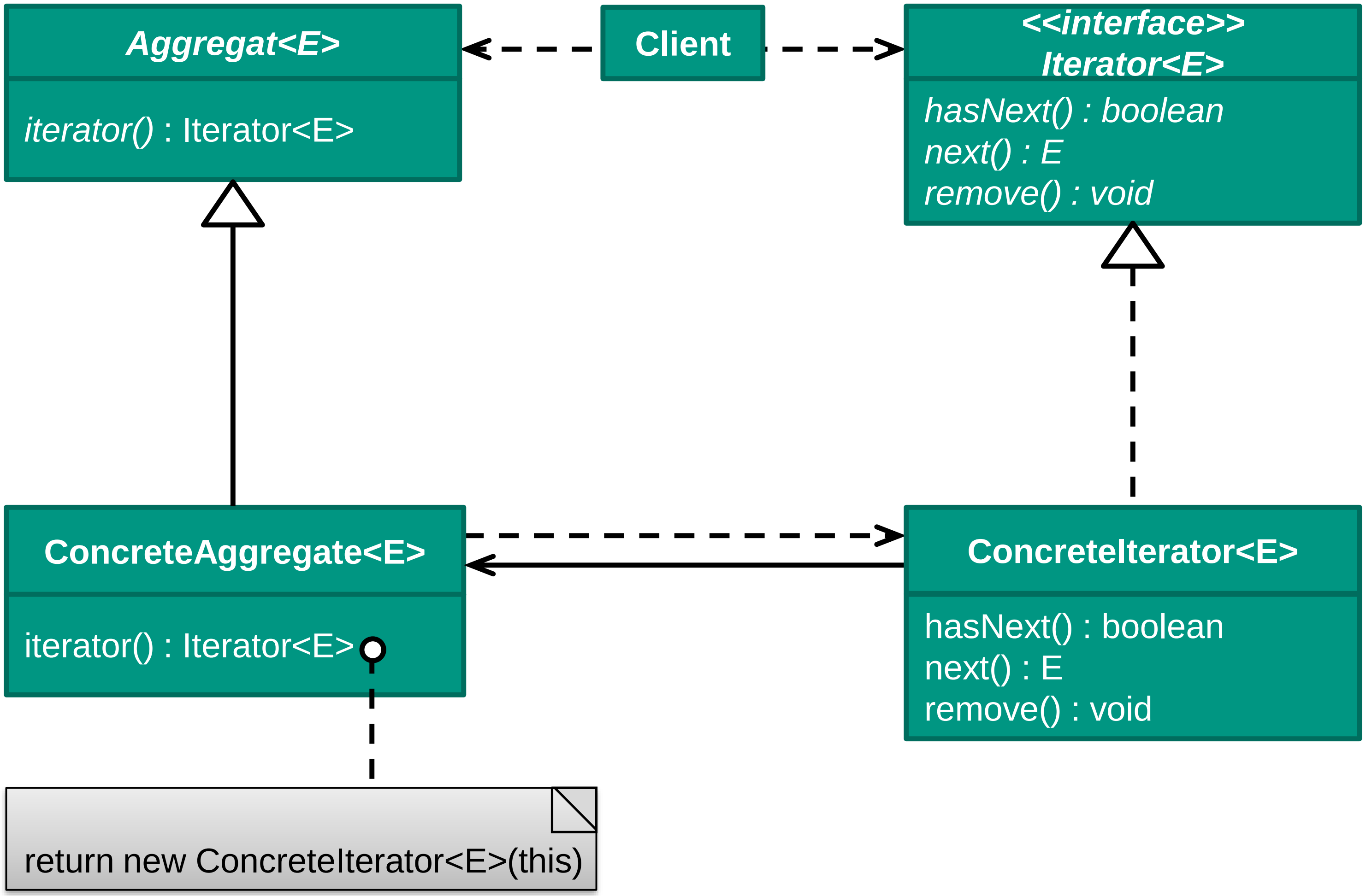
Bridge: applicability (2)

- Implementation should be completely hidden from clients
- Avoid rapid growth of subclasses
- Allow common use of implementation by several objects

Iterator

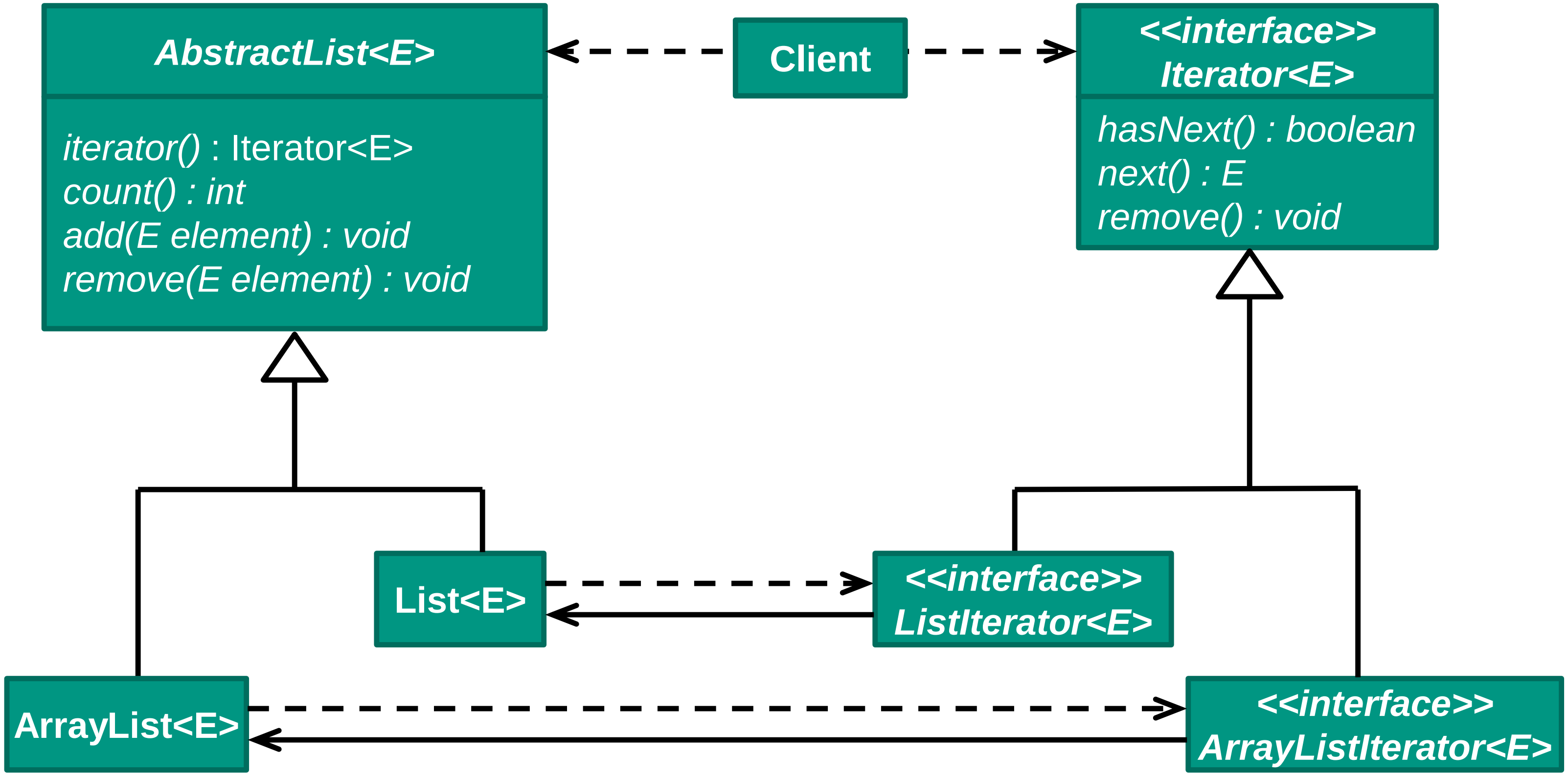
- Purpose
  - Enable sequential access to elements of composite object without exposing underlying representation
  - E.g. iterate through sequence independent of implementation as doubly-linked list or array
- Synonyms: Enumerator, robust iterator

Iterator: Structure



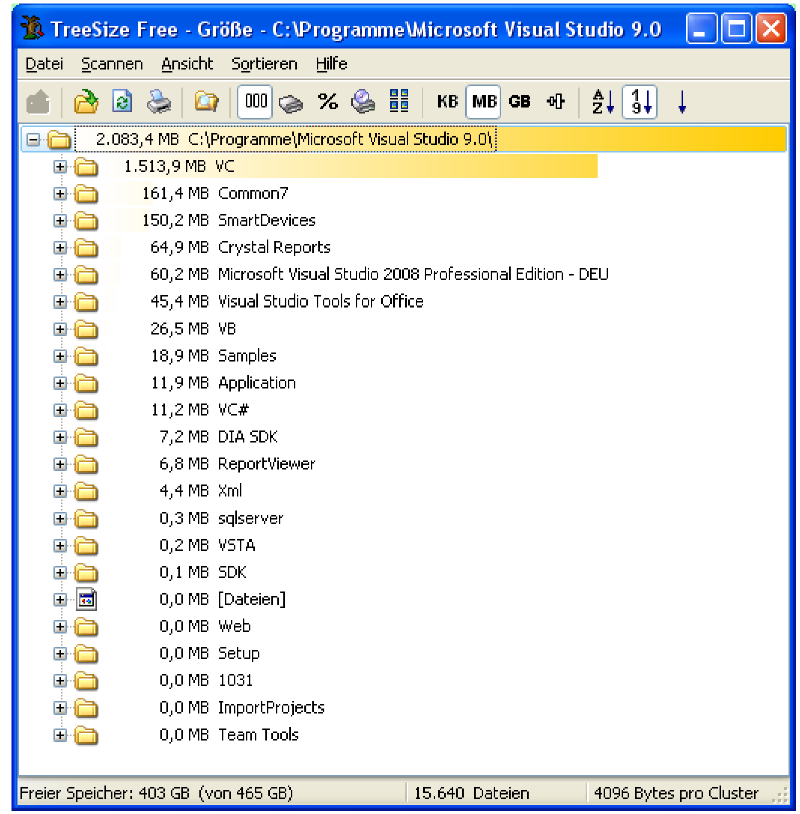
Iterator: Example

- Separate sequence (“list”) implementation from sequence traversal

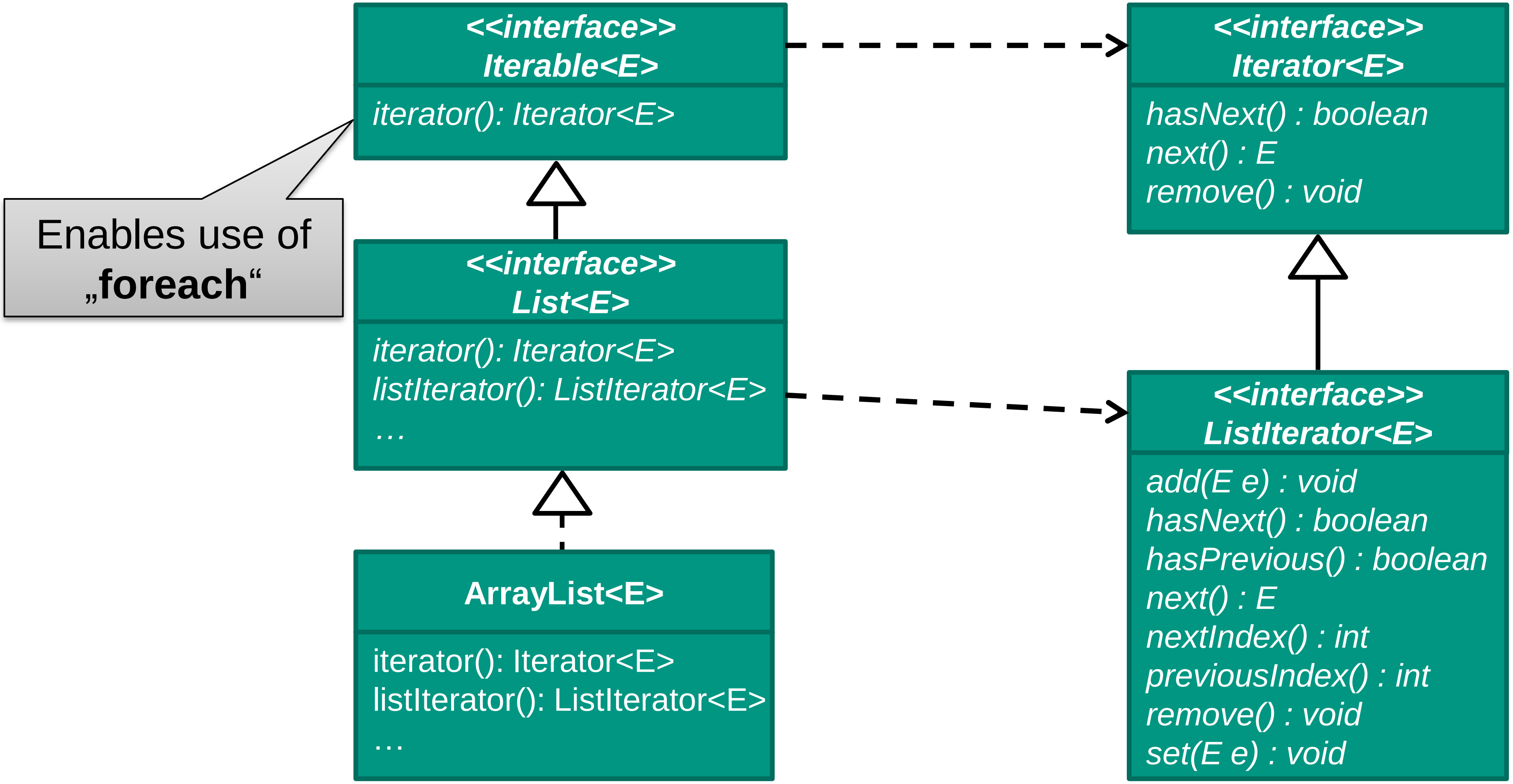


Iterator: Application example

- Use an iterator to access all directories and contained files
- Access all directories and files, determine size and sizes.



Iterator: Java Implementation (1)



Iterator: Java Implementation (2)

- **Variant 1:**

```
ArrayList<String>
stringArrayList =
  new ArrayList<String>();

Iterator<String> iter =
  stringArrayList.iterator();

while(iter.hasNext()) {
  System.out.println(
    iter.next()
  );
}
```

Variant 2:

```
ArrayList<String>
stringArrayList =
  new ArrayList<String>();

for(String str: stringArrayList)
{
  System.out.println(str);
```

Possible because `ArrayList` implements interface `Iterable`

Iterator: Enumerator vs. Iterator in Java

- Interface `Enumerator` included from beginning
- Interface `Iterator` first with version 1.2
- Iterator allows removal of elements with method `remove`()
  - No `remove`() method in Enumerator
- In Iterator, when `remove()` method is not possible/sensible, implement remove() so that it throws `java.lang.UnsupportedOperationException` exception

Read more: Head First Design Patterns, Chapter 9

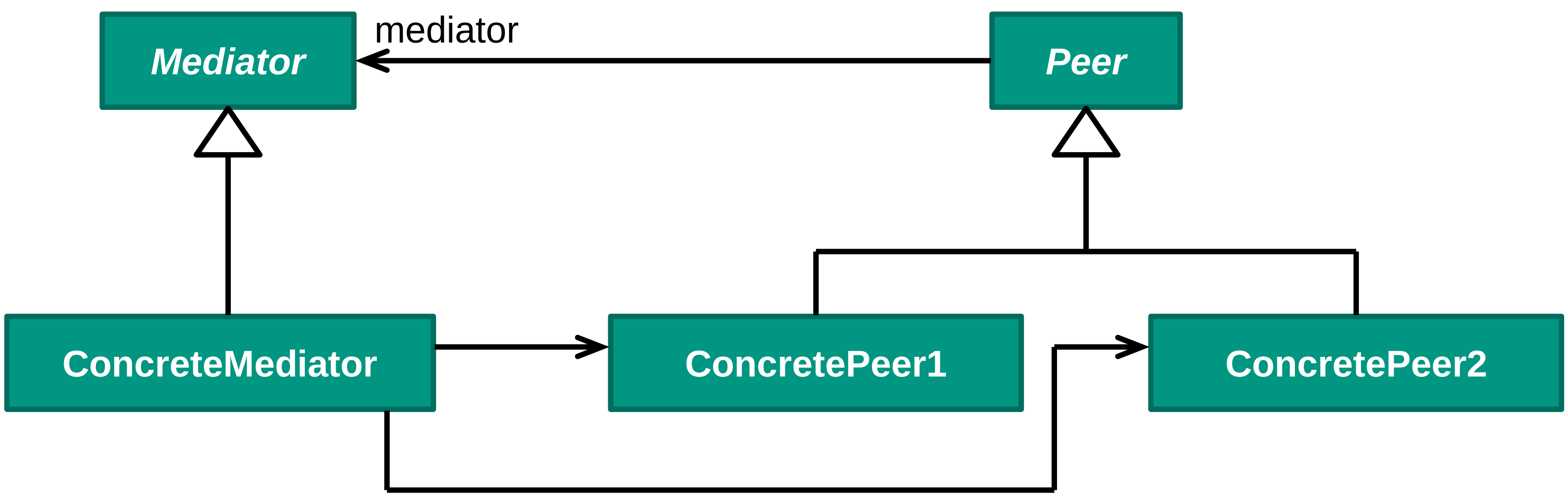
Iterator: Applicability

- Enable access to composite object without exposing internal structure (representation).
- Enables polymorphic Iteration; uniform interface for traversal of structures with diverse compositions.
- Robust: several iterators can be active simultaneously, each maintaining its own “counter”, i.e. index into to the traversed structure

Mediator

- Purpose
  - Define object that encapsulates the cooperation of a set of objects
  - Mediators enable loose coupling:
    - Avoid direct references of objects to each other
    - Localizes the cooperation aspects

Mediator: Structure



Mediator: Example 1)

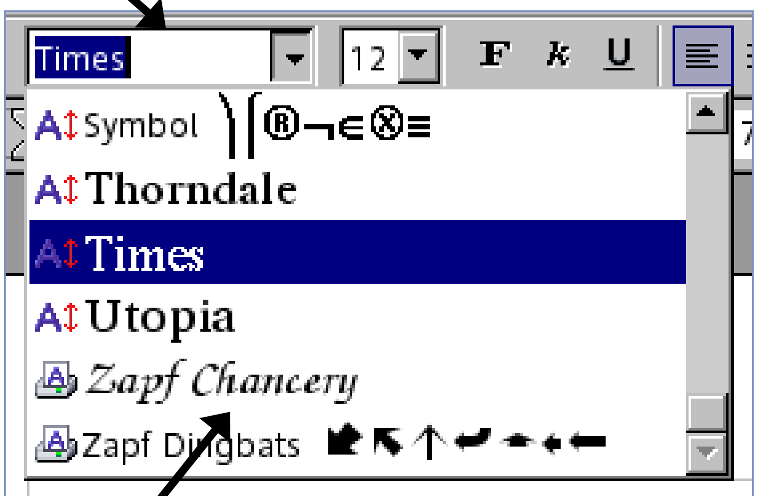
- Dependencies between elements of dialog box:
  - Buttons, menus, input fields, etc.
  - E.g. button deactivated while text input field is empty.
- Different dialog boxes require different interaction patterns
- Distributing different interaction patterns over cooperating objects leads to proliferation of subclasses

- Solution: encapsulates specific interaction regime in mediator

- (Analogon:
  - Components are words;
  - Mediator corresponds to sentence structure)

Mediator: Example (2)

InputField

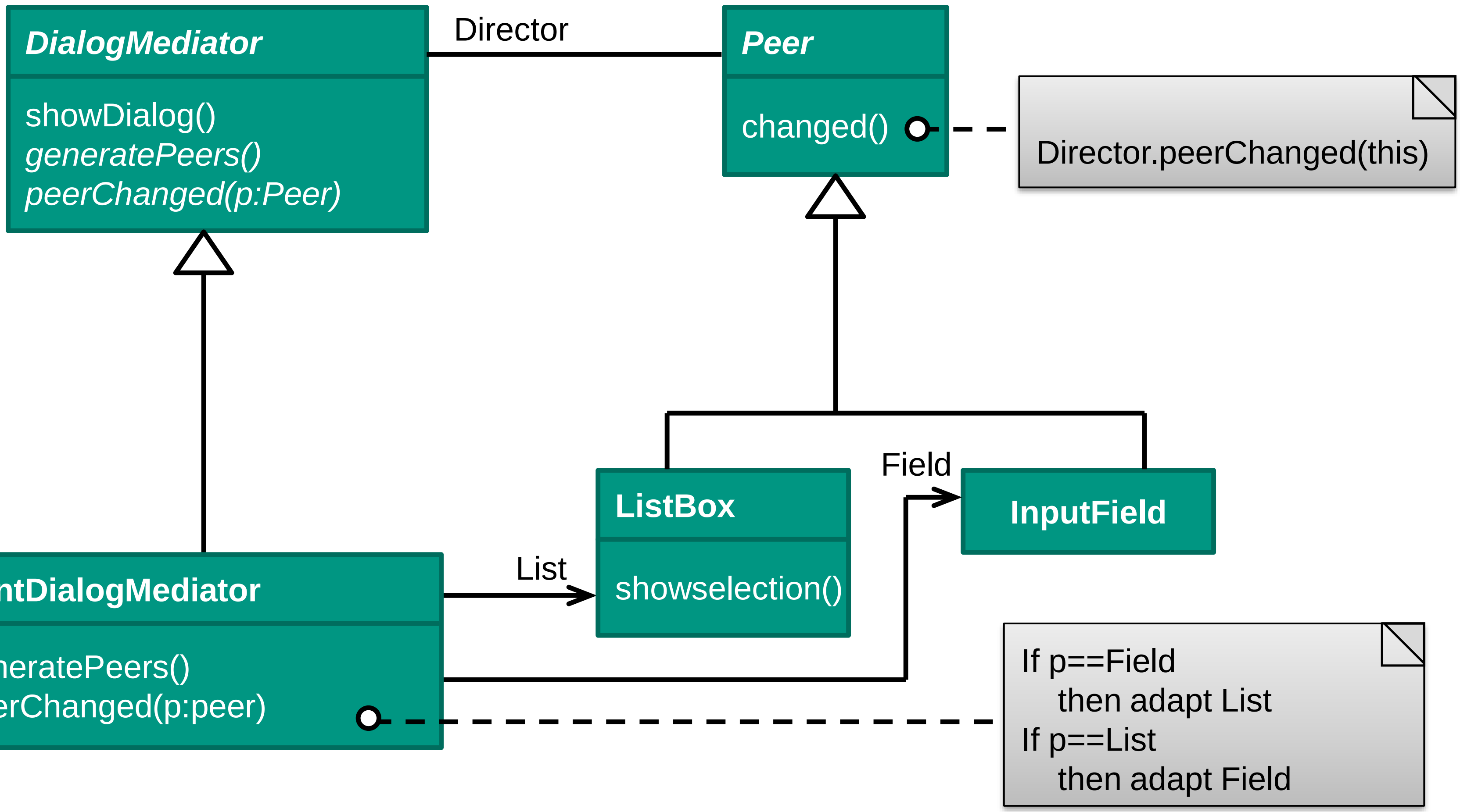


Listbox

- Typing prompts opening of dropdown menu
- Input field show first menu item consistent with text input
- Chosen menu item appears in input field

Mediator: Example (3)

- Window with Font dialog



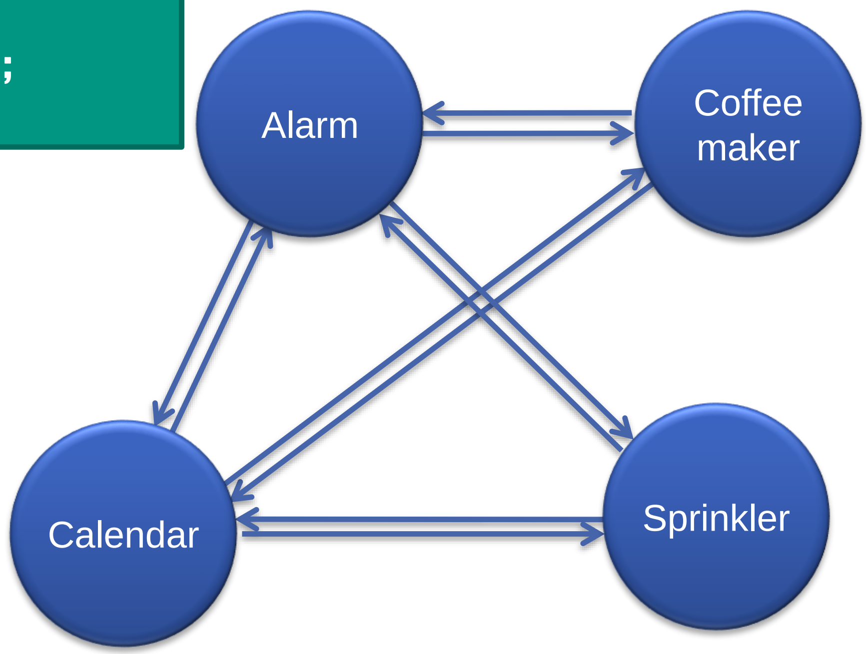
Mediator: Example 2 (1)

Alarm

```
onEvent() {
  checkCalendar();
  checkSprinkler();
  turnonCoffeemaker();
}
```

Coffeemaker

```
onEvent() {
  checkCalendar();
  checkAlarm();
}
```



Calendar

```
onEvent() {
  checkDayOfWeek();
  irrigateLawn();
  makeCoffee();
  setoffAlarm();
}
```

Sprinkler

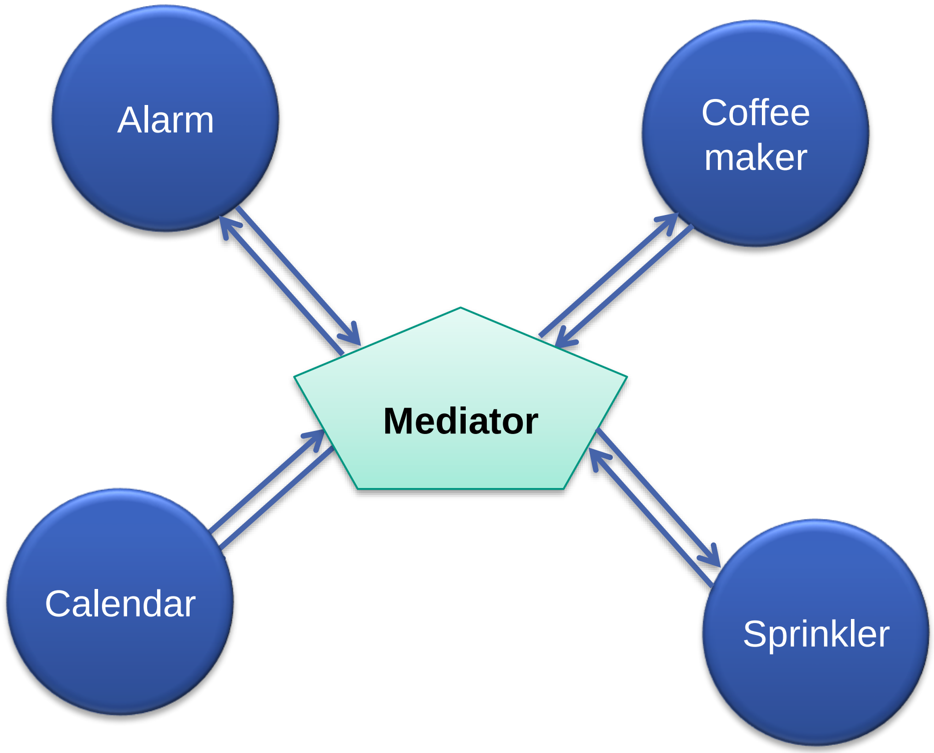
```
onEvent() {
  checkCalendar();
  checkShower();
  checkTemperature()
  checkWeather();
}
```

Each class needs to know all the others (almost).

Mediator: Example 2 (2)

Mediator

```
onEvent() {
  if (alarmtime) {
    checkCalendar();
    checkShower();
    checkTemperature();
  }
  if (weekend) {
    checkWeather();
  }
  if (garbageCollection) {
    setAlarmEarlier();
  }
}
```



- Mediator coordinates
- Other classes only coupled with mediator

Mediator: Applicability

- Set of objects need to interact in well-defined but complex manner
  - Mutual dependence is complex
  - Interactions and consequences are hard to understand
- Reuse of objects is difficult
  - Because situation-specific dependencies are built into objects
- Distributed behavior needs to be customized
  - With proliferating subclasses

Variant Patterns

- Abstract factory
- Visitor
- Template
- Factory
- Compositum
- Strategy
- Decorator

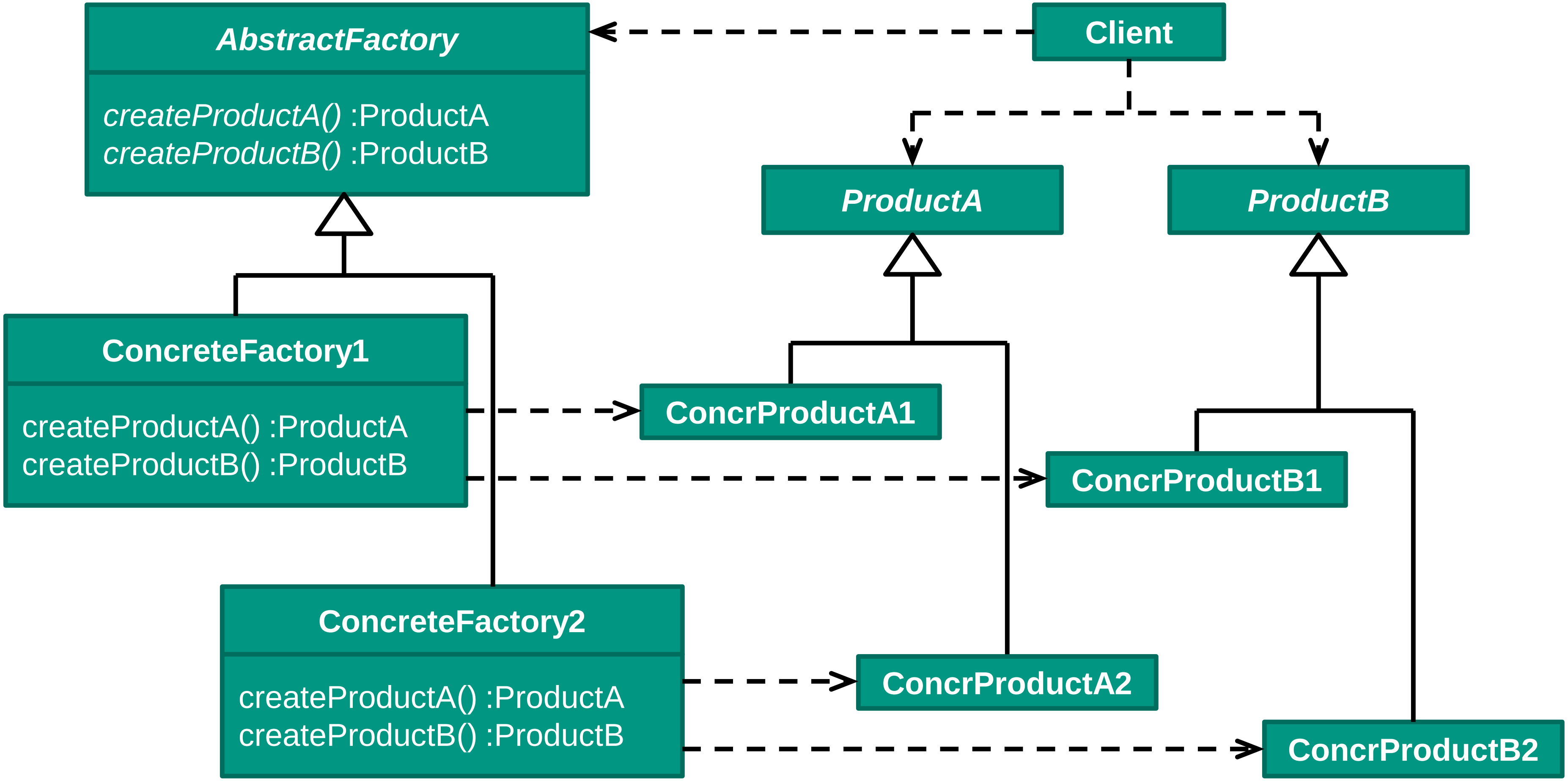
Abstract Factory

- Purpose
  - Interface for creating families of related objects or interdependent objects
    - Without referring to their concrete classes

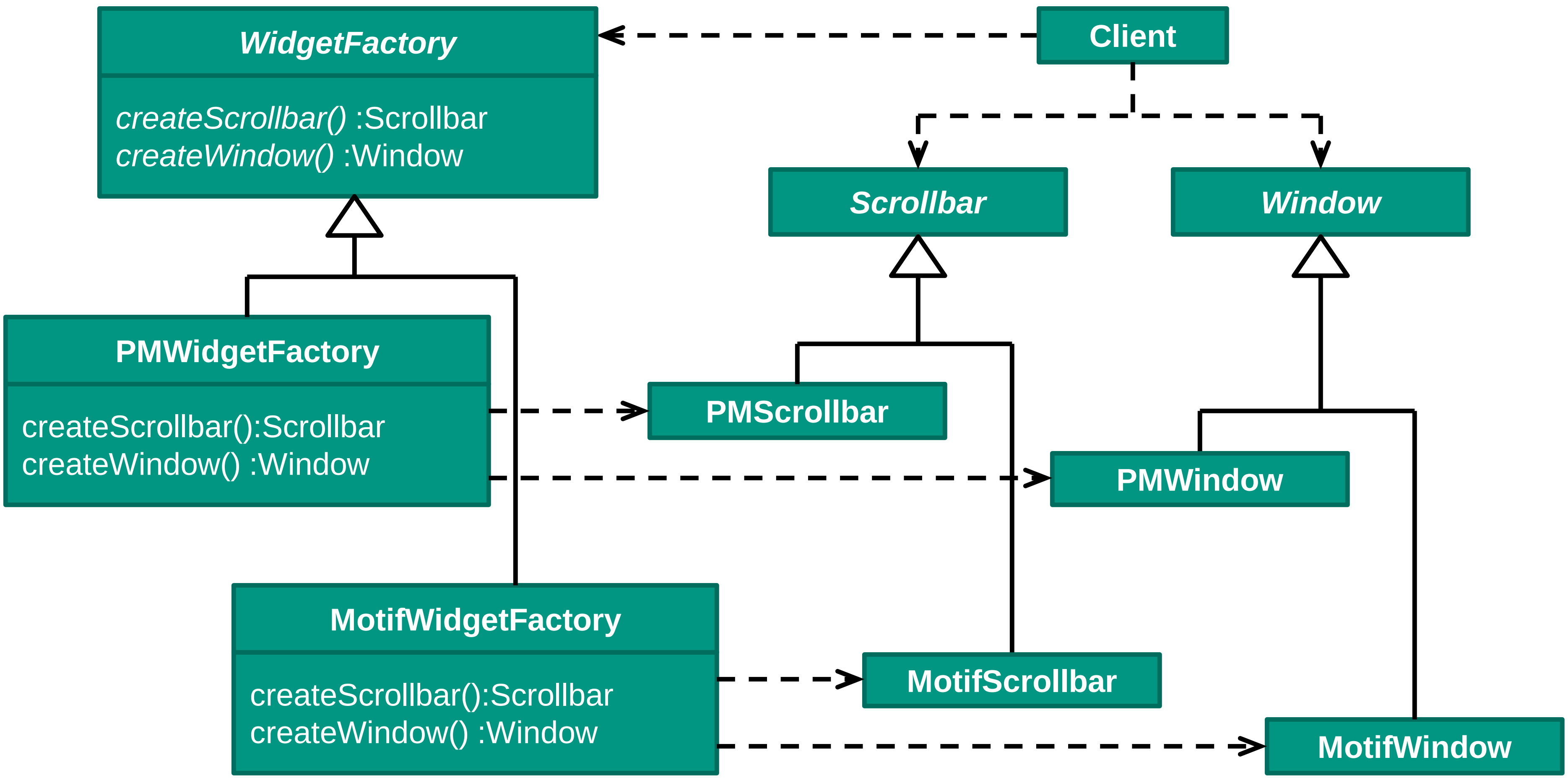
- Synonyms: Kit

Read more: Head First Design Patterns, Chapter 4

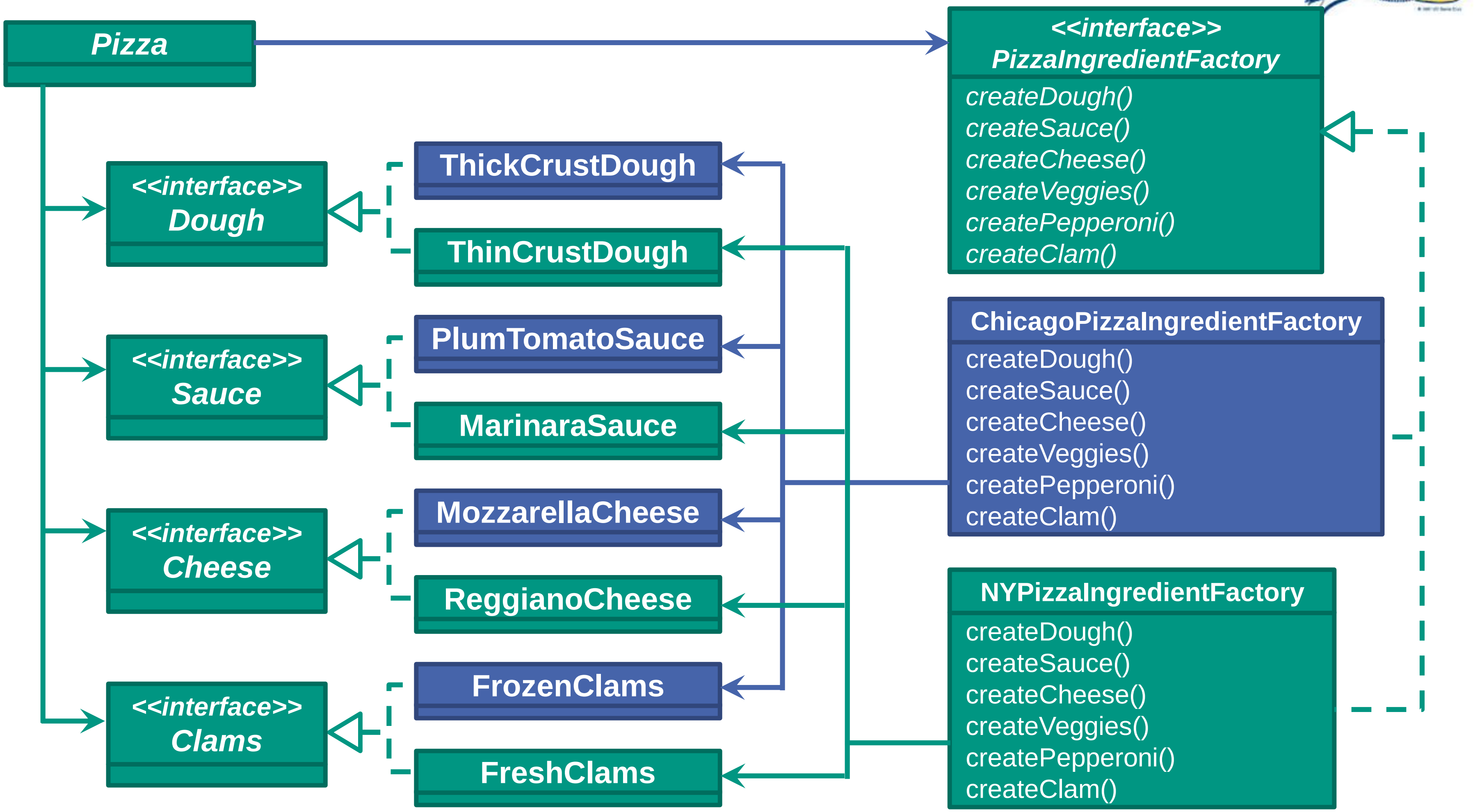
Abstract Factory: Structure



Abstract Factory: Example (1)



Abstract Factory: Example (2)



Example: Head First Design Patterns, page 157 ff.

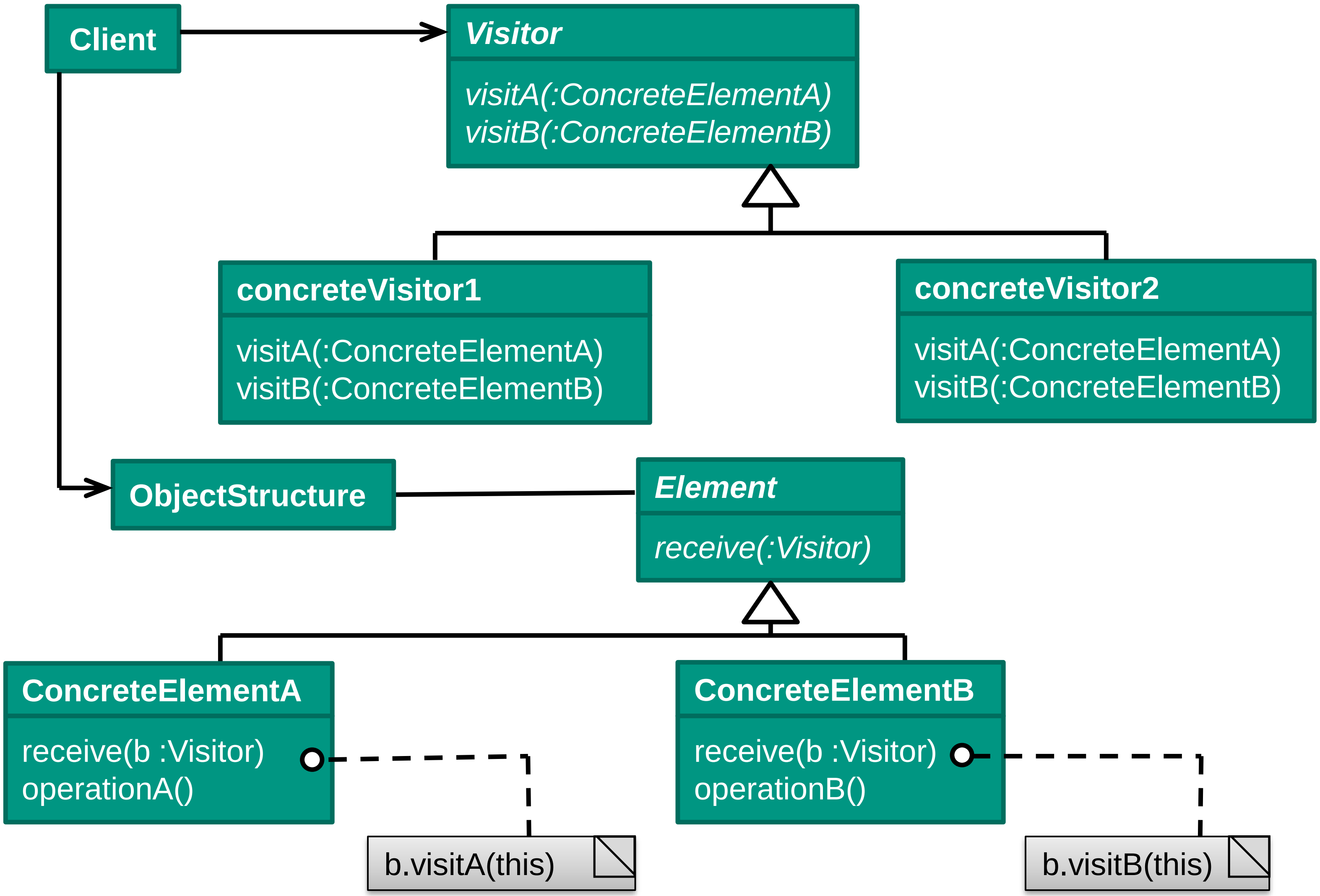
Abstract Factory: Applicability

- System should be independent of how its components are created, composed, and represented
- System should be configurable within one ore more product families
- When a coherent family of system objects needs to be used together
  - Mutual component fit needs to be ensured
- Class library that exposes only interfaces but not implementations

Visitor

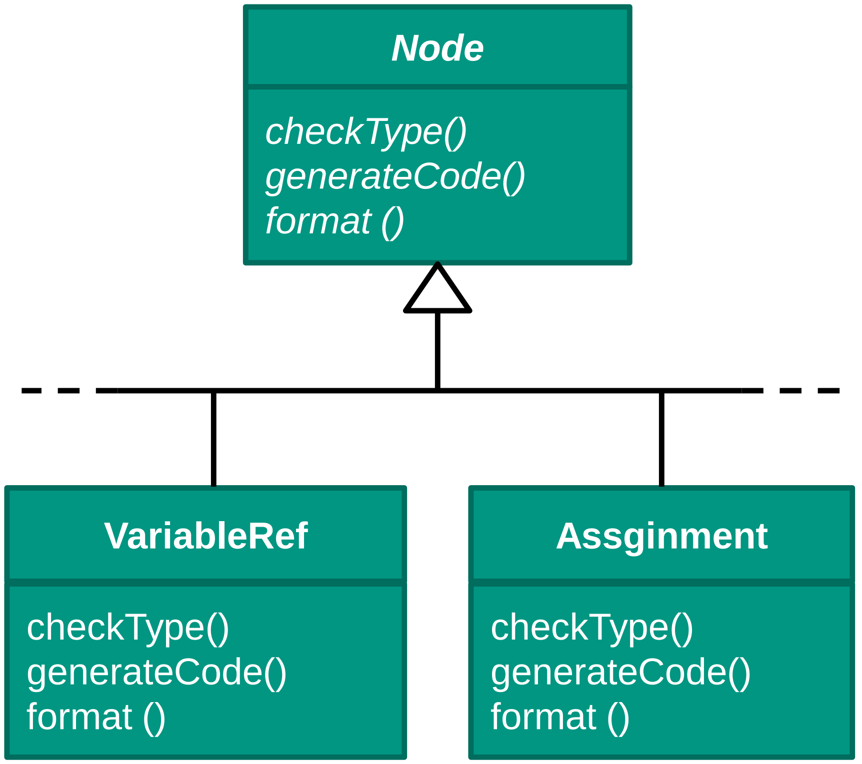
- Purpose
  - Encapsulate an operation as an object
    - Operation on elements of an object structure
  - Enables definition of new operation on object structure without changing classes of objects to be operated on

Visitor: Structure



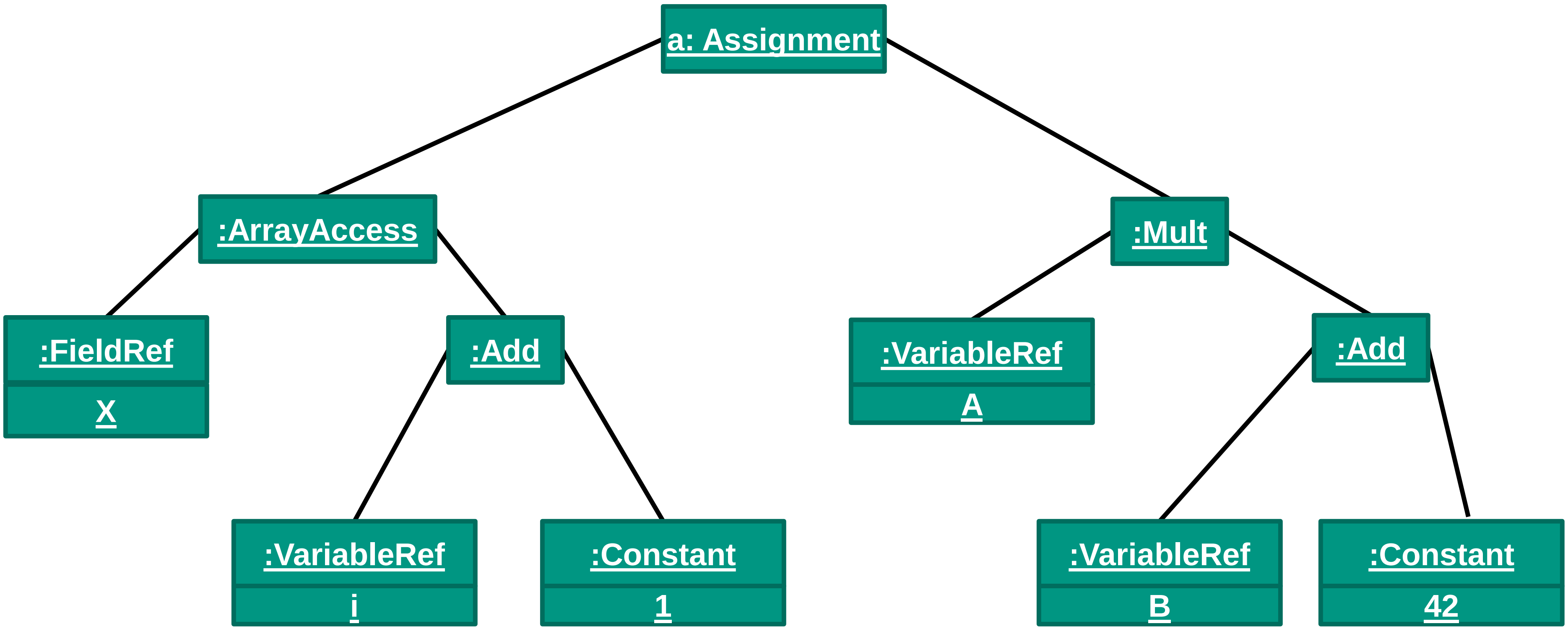
Visitor: Example (1) w/o visitor

- Abstract syntax trees in compiler
- Operations distributed over many classes
- Introduction of new operation requires modification of all classes



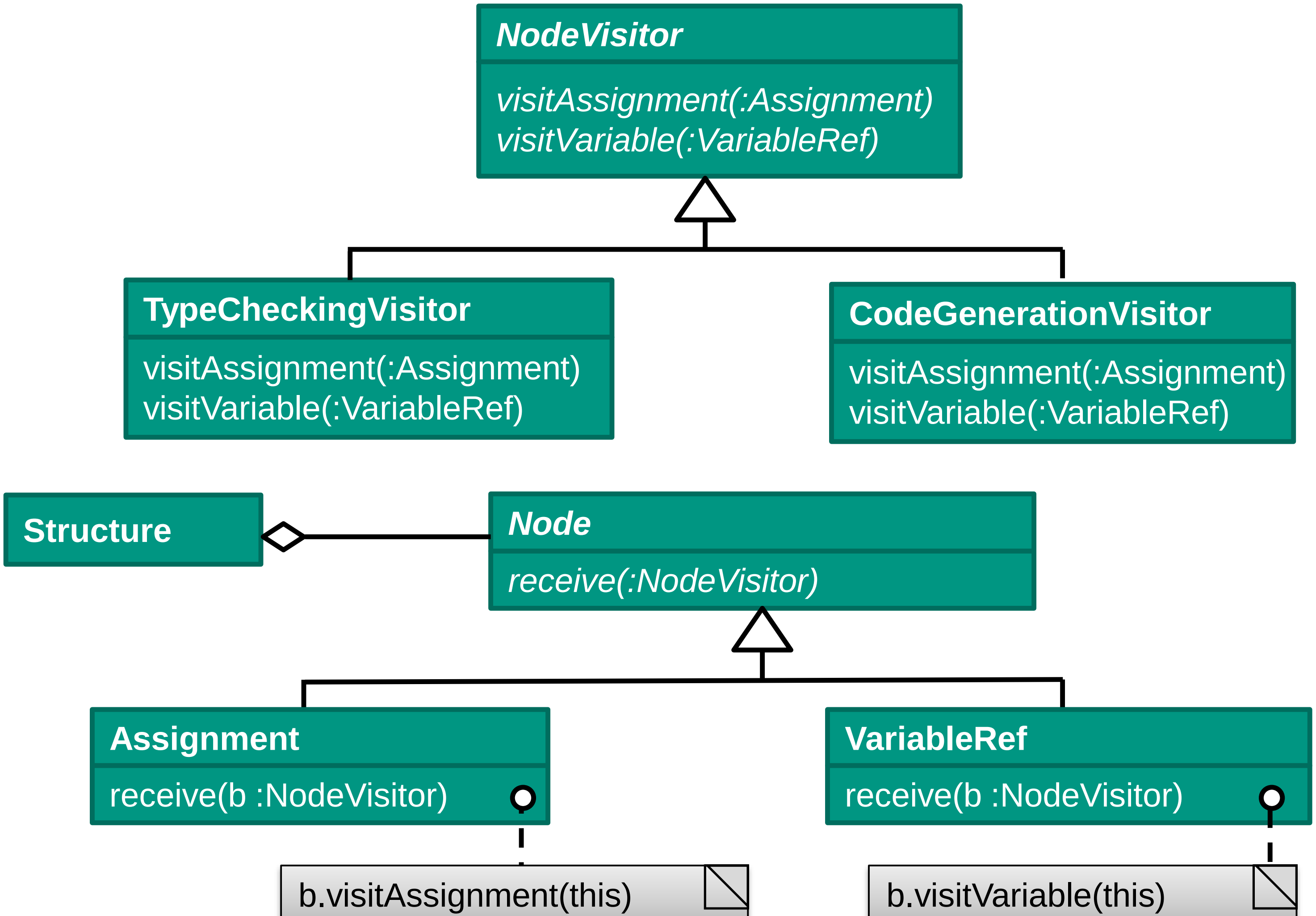
Example Abstract Syntax Tree

```
X[i+1] = A * (B + 42)
```



What does `a.receive(new CodeGenVisitor())` do?

Visitor: Example (2) with Visitor



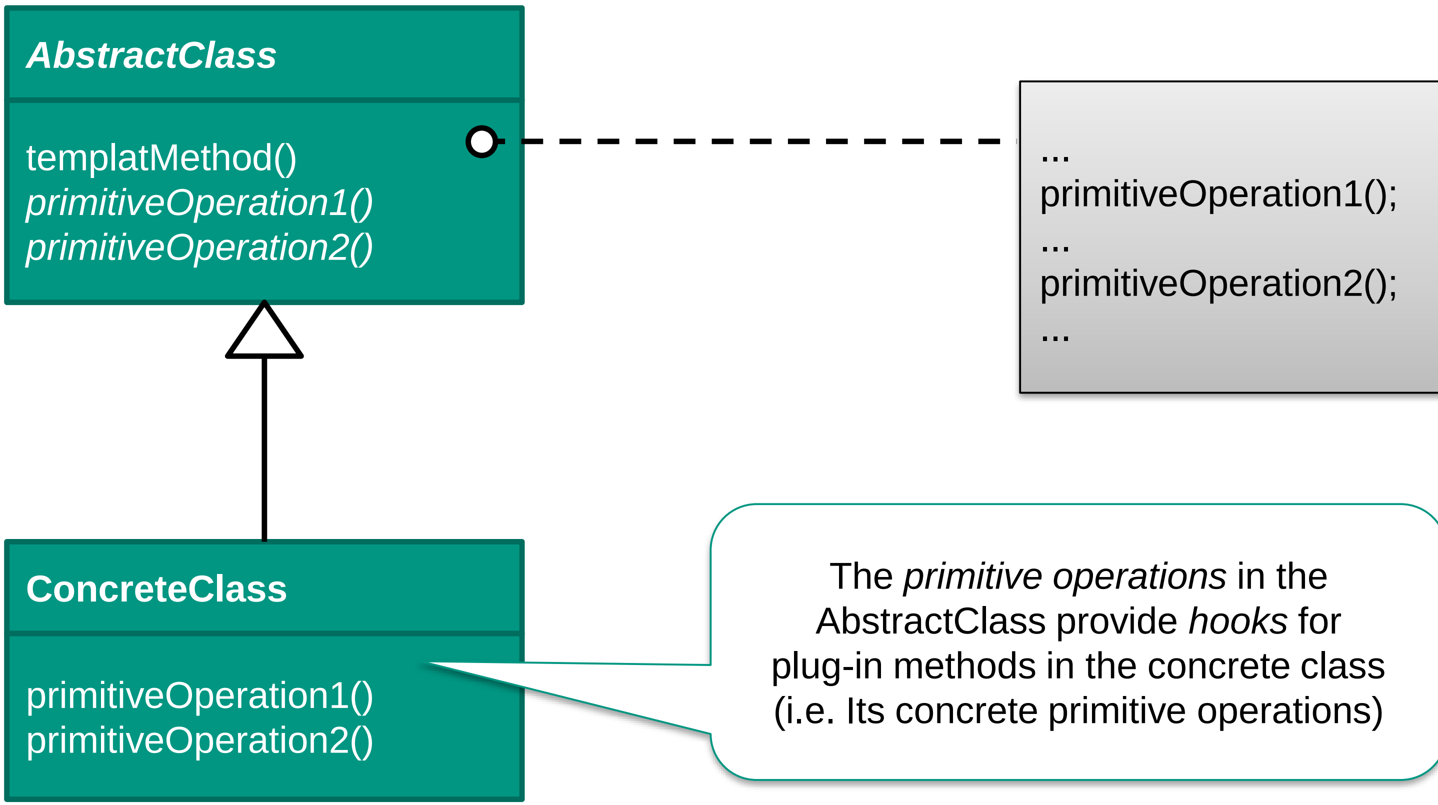
Visitor: Applicability

- An object structure contains object of many different classes with different interfaces
  - A number of different operations specific to the object classes need to defined
- A number of unrelated operations need to be executed on the objects of an object structure without contaminating the structure
- Frequent new operations need to be executed on a stable object structure
- **Note**: In functional programming, the visitor pattern is known as a catamorphism (a general form of homomorphism)
  - E.g. length is a homomorphism from strings to the natural numbers
    - length(“”) = 0
    - length(“a”) = 1   for each character a
    - length(s ++ t) = length(s) + length(t)

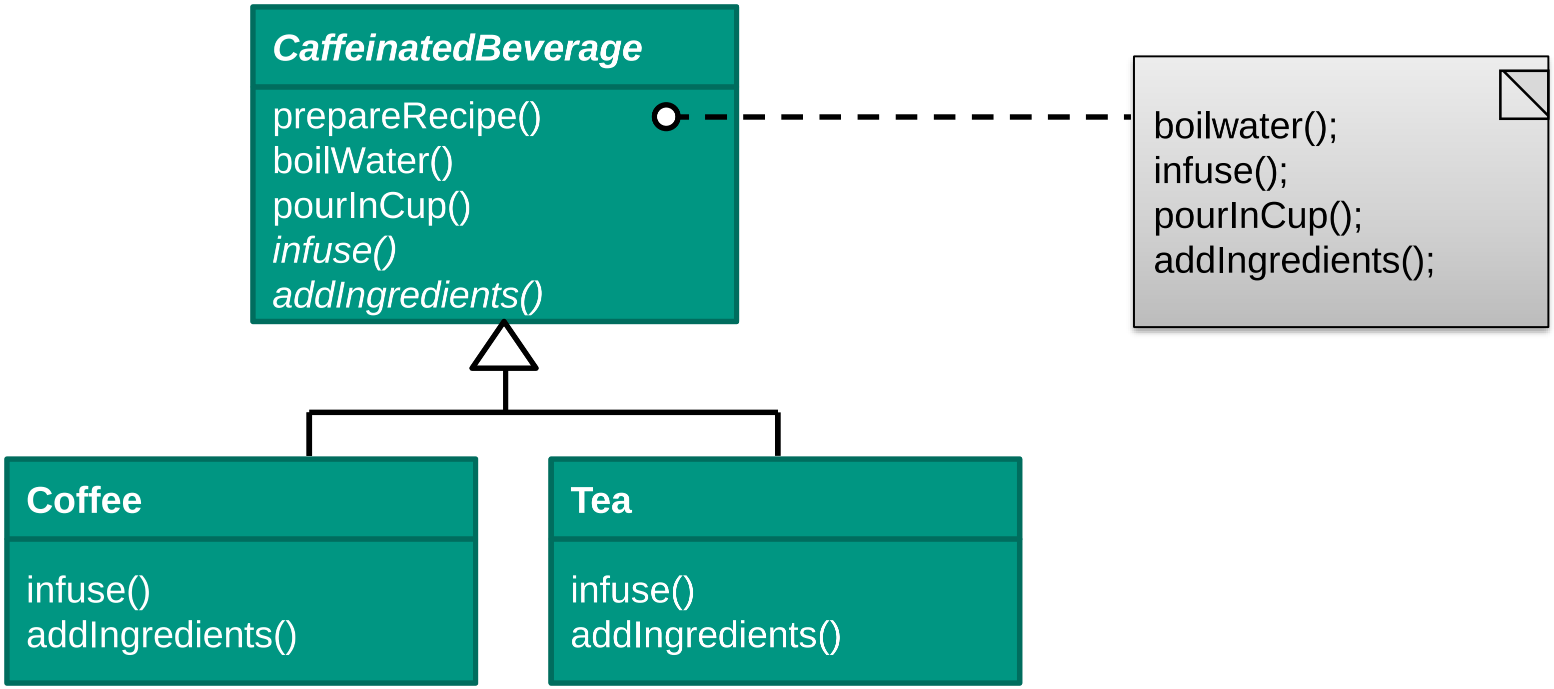
Template Method

- Purpose
  - Define skeleton of an algorithm in an operation/method and delegate individual steps to subclasses
  - Use of template method enables varying the steps of an algorithm without changing its overall structure
    - In functional languages achieved by passing functions/methods as parameters

Template Method: Structure



Template Method: Example



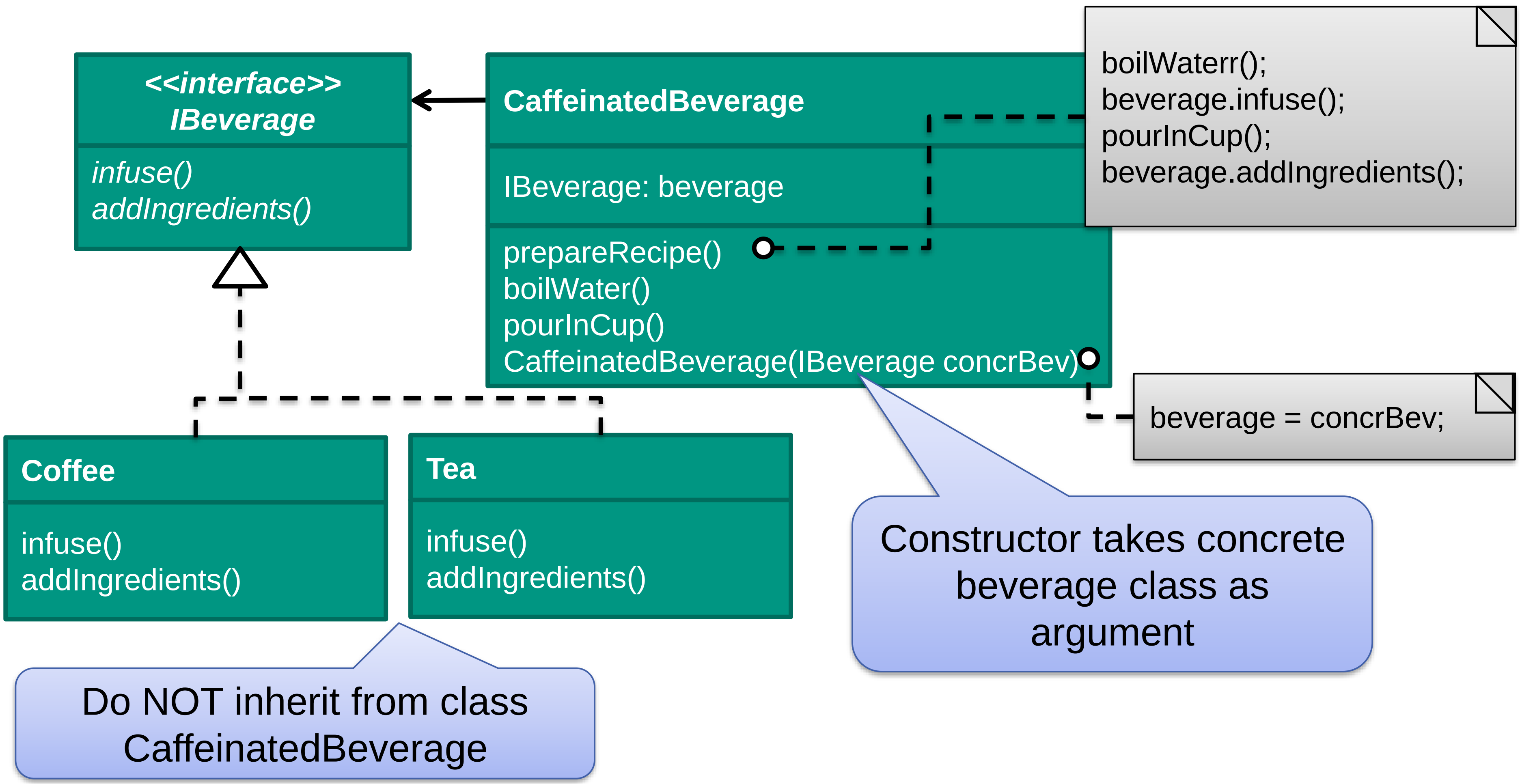
Example: Head First Design Patterns, page 280 ff.

Template Method: Applicability

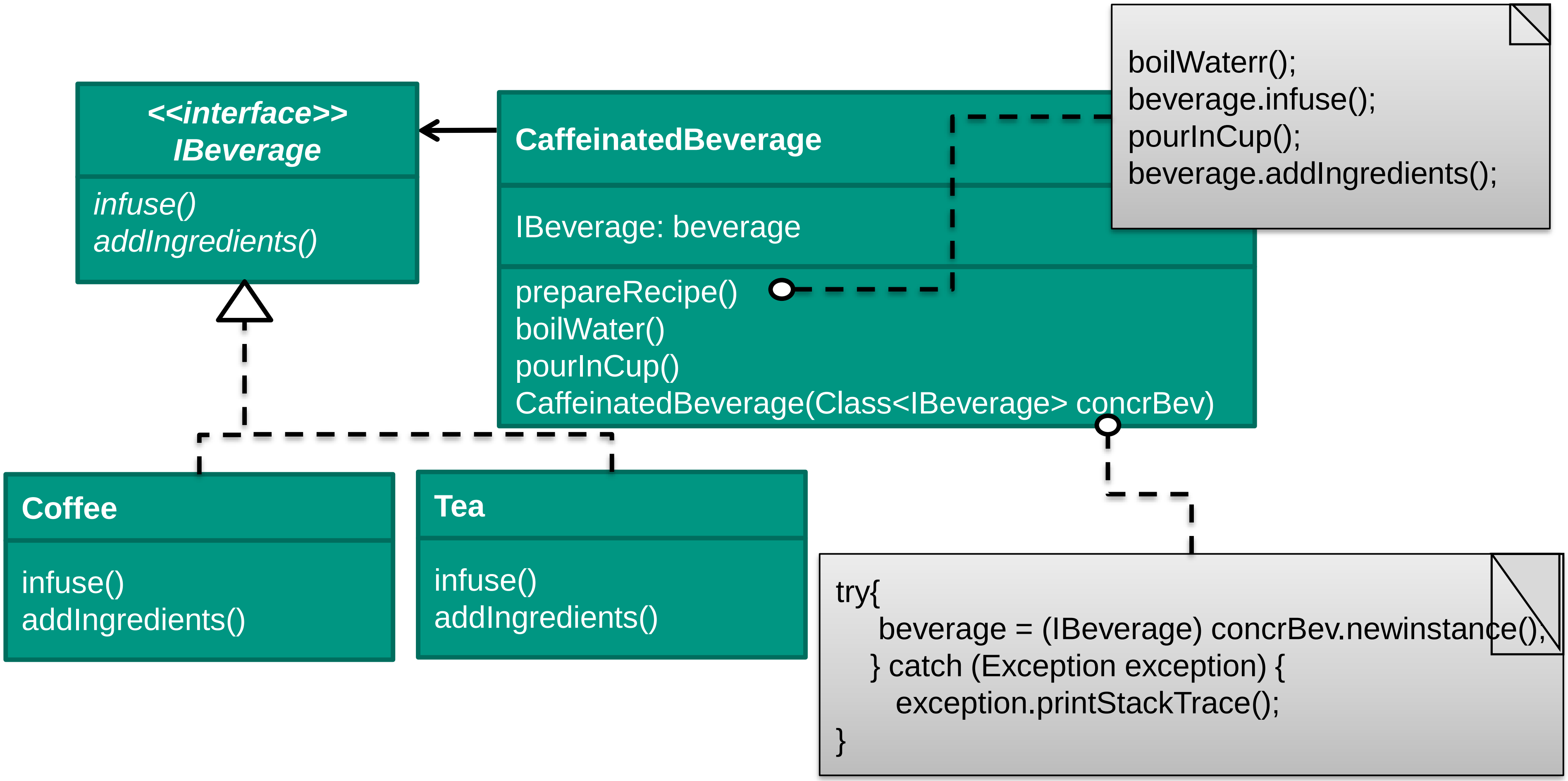
- Specify invariant part (scheme) of algorithm only once; leave “holes” for specializations to fill in various behaviors.
- Allows factorization of common algorithm structure; avoids duplication of code.
- Controlled extension of subclasses: *hooks* specify where specific methods can be plugged in.
- Algorithm level equivalent of Frame Architectures

Read more: Head First Design Patterns, Chapter 8

Template Method: Alternative 1



Template Method: Alternative 2: Generic Classes

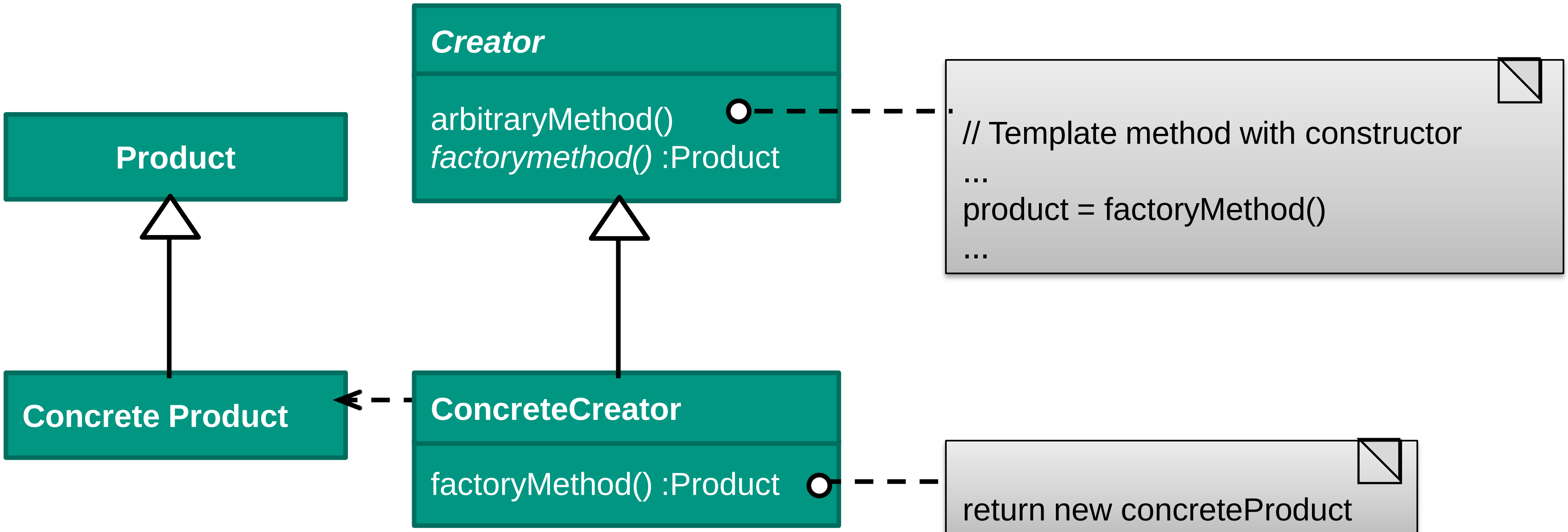



```
Class<IBeverage> t = (Class<IBeverage>)Class.forName("kit.ipd.Tea");
CaffeinatedBeverage tea = new CaffeinatedBeverage(t);
tea.prepareRecipe();
```

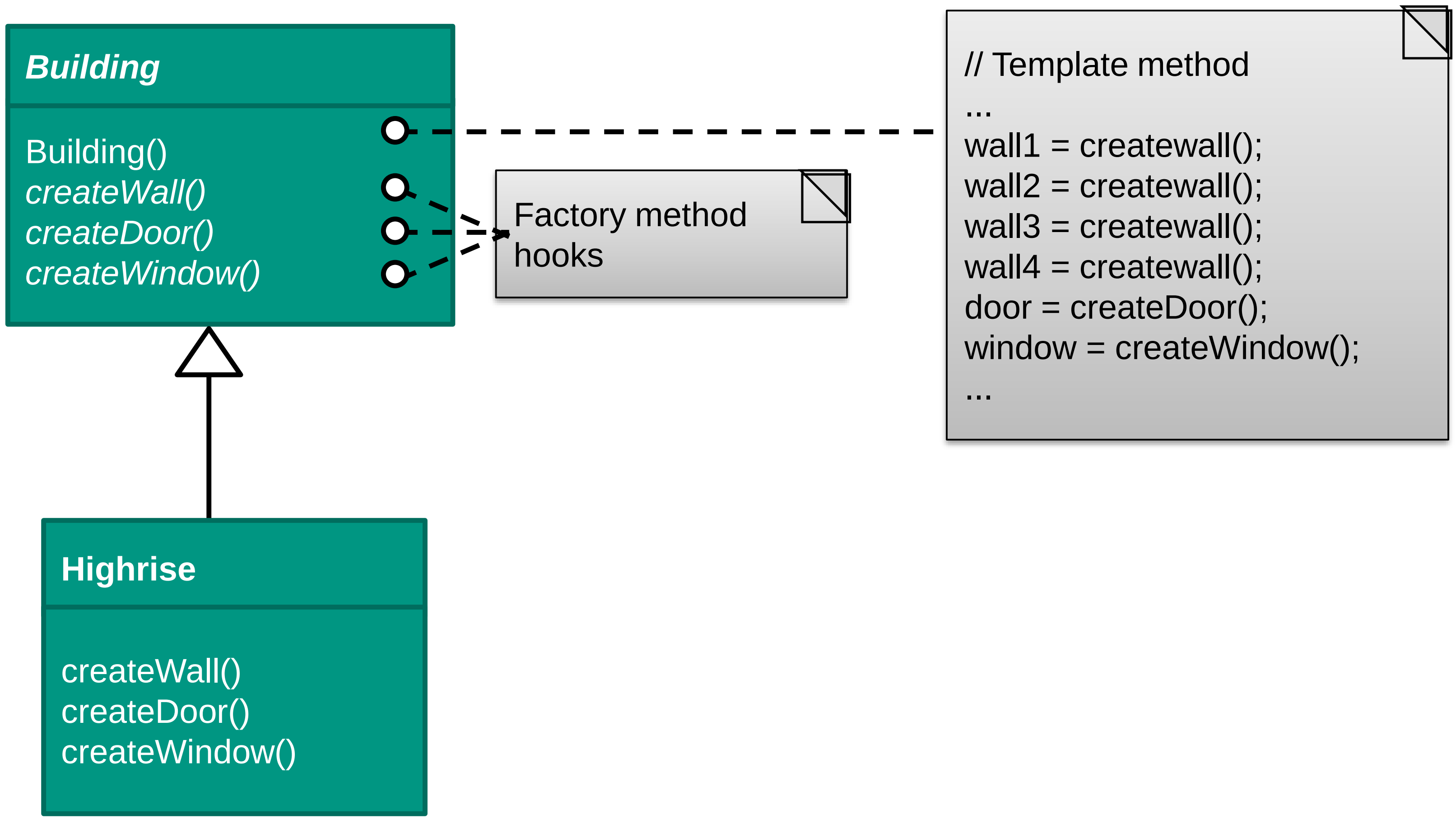
Factory Method

- Purpose
  - Define class interface with operations for creating an object but let subclasses decide what the class of the object to be created is.
  - Factory methods enable delegation of object creation to subclasses.

Factory Method: Structure



Factory Method: Example



Template Method: Applicability

- When a class needs to construct objects whose class it doesn’t know
- When a class wants its subclasses to determine the class of objects to be created
- When classes want to delegate responsibilities to several helper subclasses but the knowledge to whom responsibilities are delegated should be localized

- A factory method is a plug-in to a template method for object creation.

Composite

- Purpose
  - Combine objects into tree structures to generates parts hierarchies.
  - The pattern enables clients to treat elements and subcomponents uniformly

- Synonyms: Whole-Part

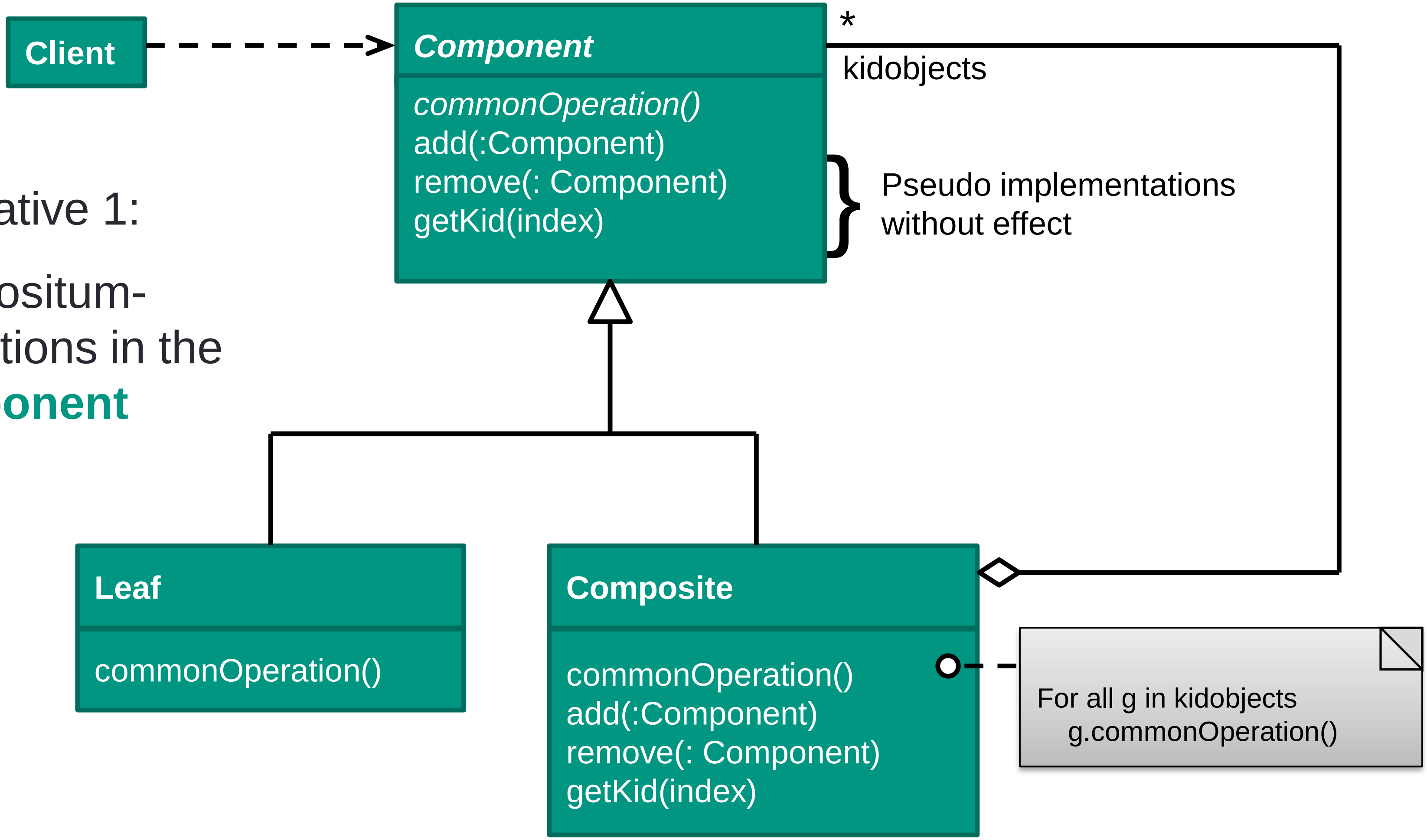
Motivation

- Part hierarchies occur whenever complex objects are modeled, e.g. data base systems, graphics applications, text processing, CAD, CIM (computer-integrated manufacturing), …
- In these applications elementary objects are grouped into larger units, i.e. (sub)components or aggregates
- Frequently, a program wants to treat elementary objects and aggregates uniformly.
- Composite factors out the common properties of elementary objects and aggregates and forms a superclass.

Composite: Structure

Alternative 1:

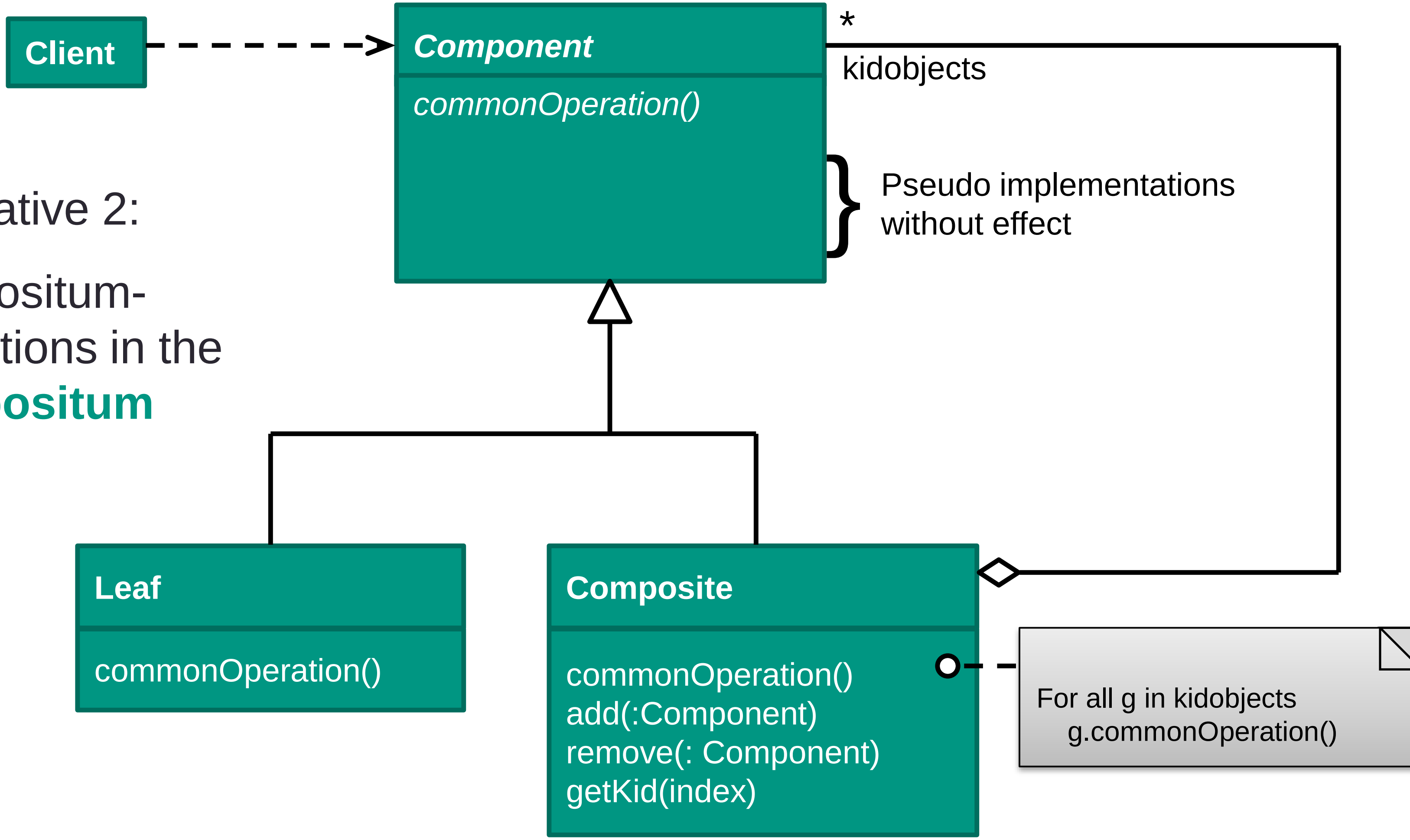
Compositum-Operations in the **Component**



Composite: Structure (2)

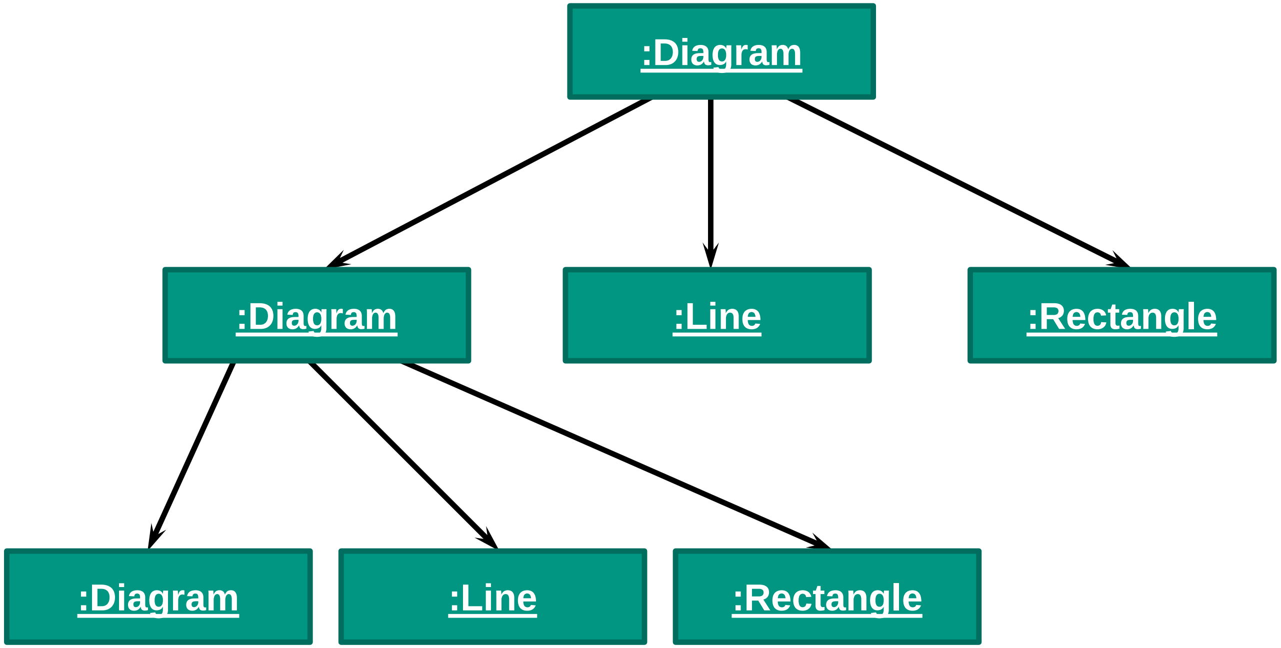
Alternative 2:

Compositum-Operations in the **Compositum**



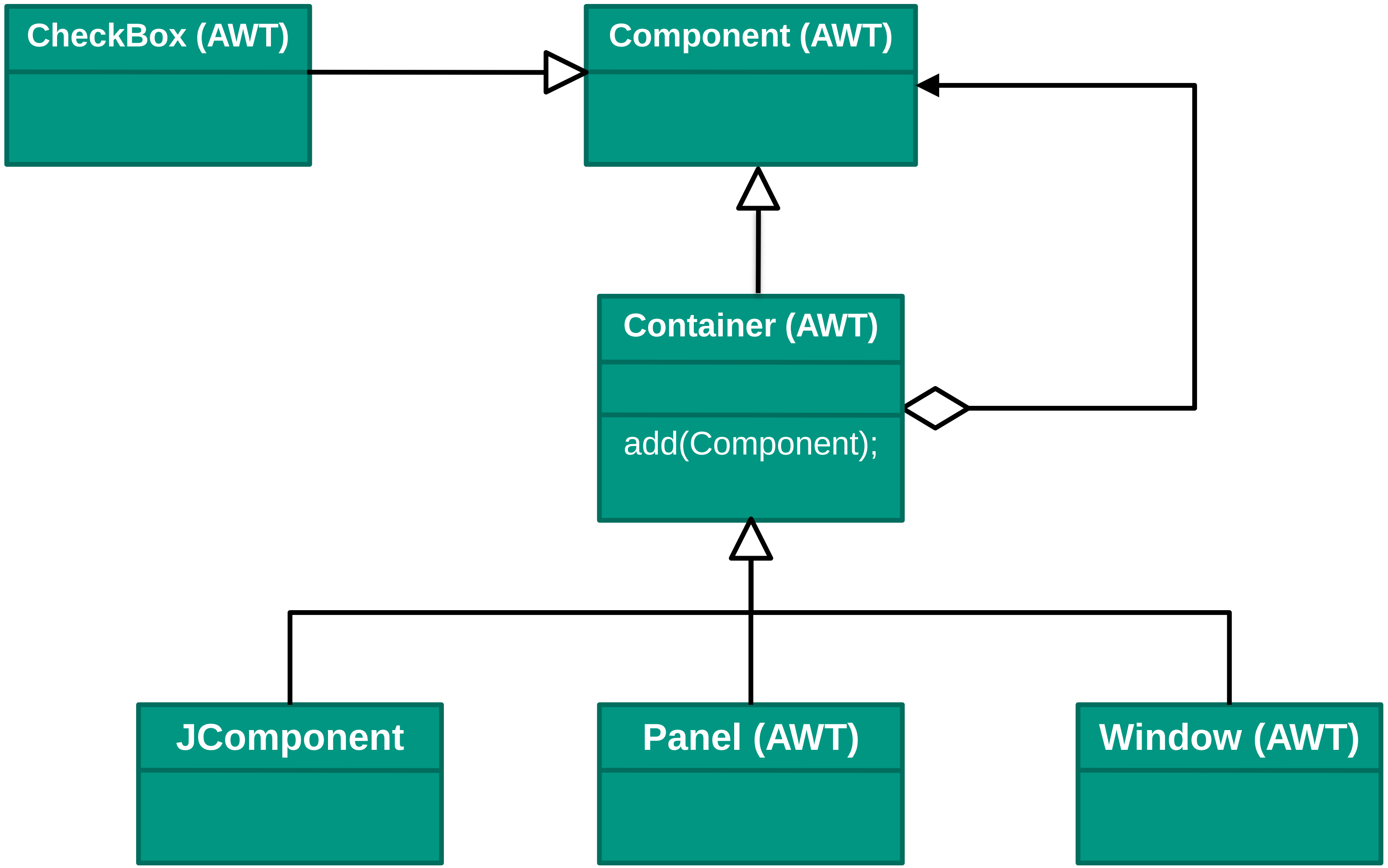
Composite: Example

- Composite graphics objects

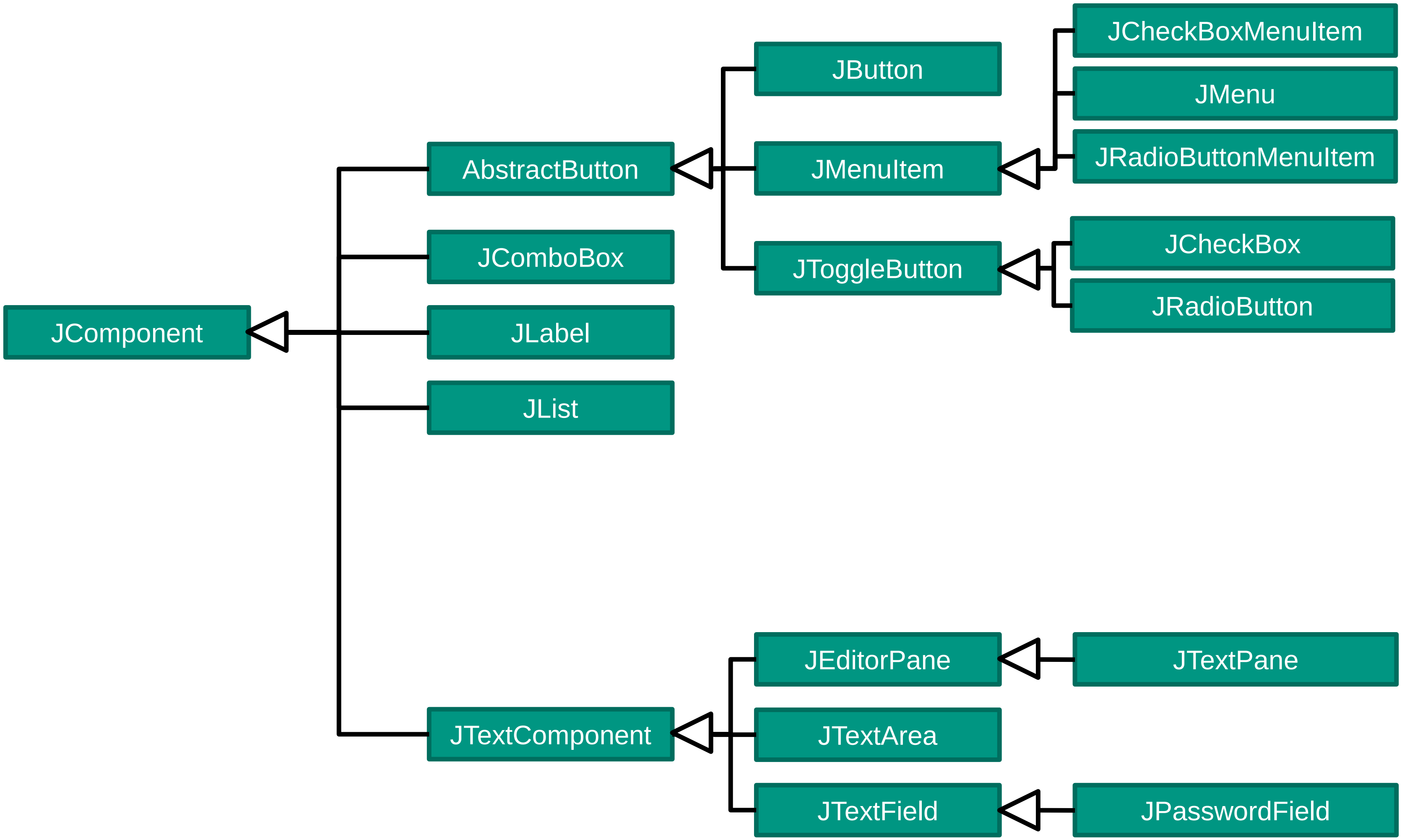


- Common operations: **draw(), translate(), delete(), scale()**

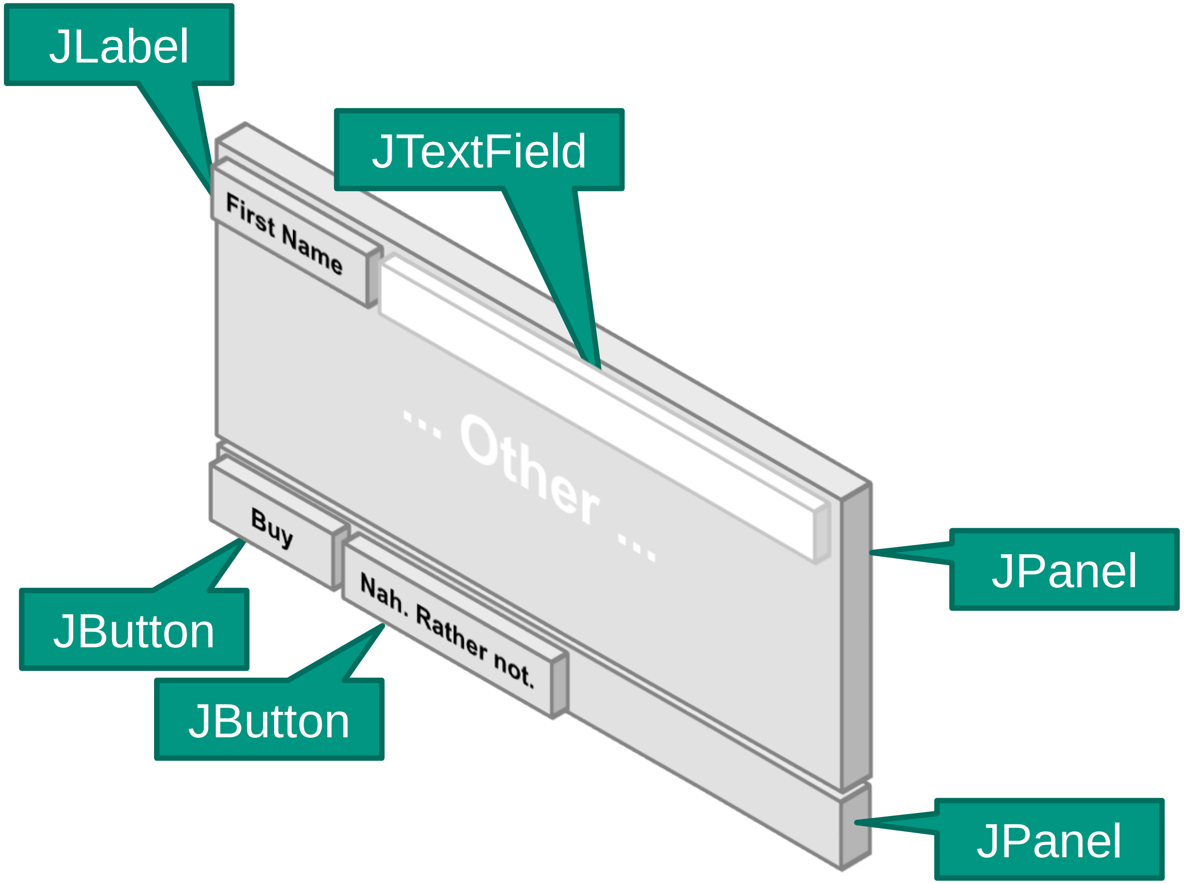
Composite: Example from Java (1)



Composite: Example from Java (2)



Composite: Java (3)



Composite: Applicability

- Class Composite manipulates the container structure that stores the components

- Applicability
  - Need to represent parts hierarchies
  - When clients need to manipulate elements and aggregates in uniform way.

Composite: Implementation (1)

- Often useful to keep a parent reference in each component
  - Parent reference can be updated by add() and remove() methods in the compositum
  - Dividing components can transform tree into a DAG (multiple parents)

Composite: Implementation (2)

- Maximize component interface
  - To guarantee transparency (to clients), the component interface should provide as many methods common to elements and composite as possible
  - When compositum methods are defined in the component, then getKidObject() should return nothing; can be achieved through implementation in component
  - add() and remove() should fail for leaves and return an error or raise an exception

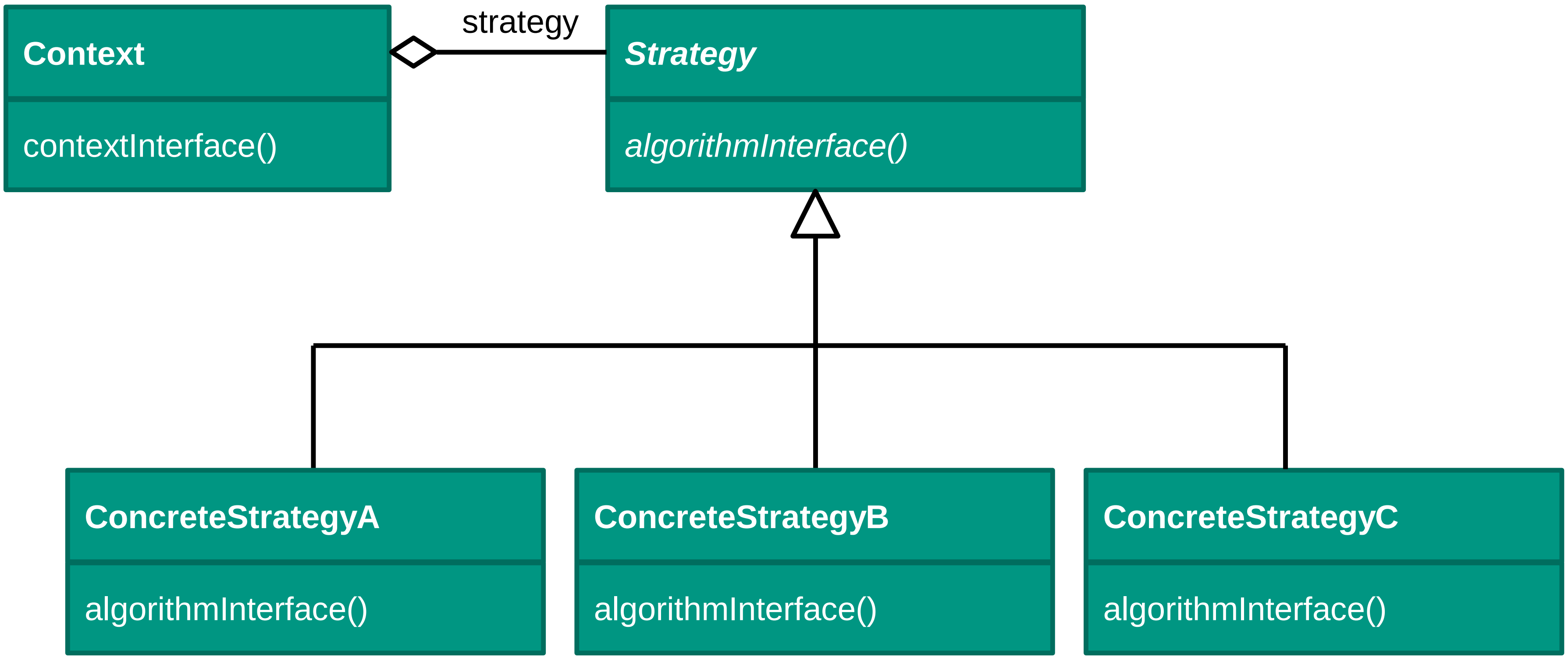
Composite: Implementation (3)

- Storing kid elements
  - Arrays, lists, sets, hash tables – depends on application and needed efficiency
  - For a fixed number of kids (e.g. binary tree), provide specialized access function for each kid (e.g. left, right)
  - Use iterator to provide access to sequence of kids
    - Order of kids crucial in some applications (e.g layout managers); not in others

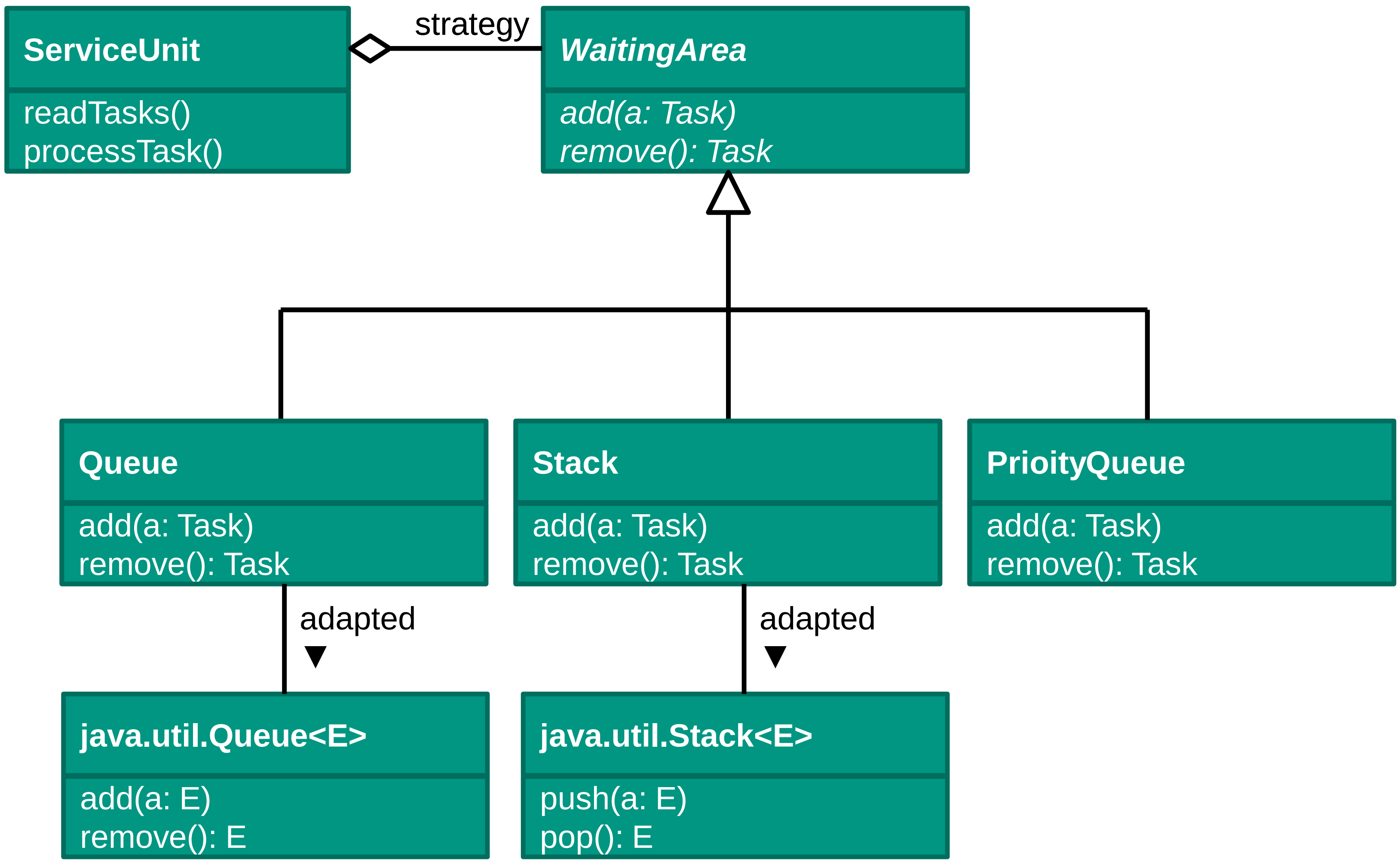
Strategy

- Purpose
  - Define a family of algorithms, encapsulate (abstract) them and make them interchangeable.
  - This pattern supports algorithm variation independent from clients
- Synonyms
  - Policy
- Motivation
  - Sometimes there is a need to vary algorithm depending on performance requirements, load or throughput, or type of data.

Strategy: Structure

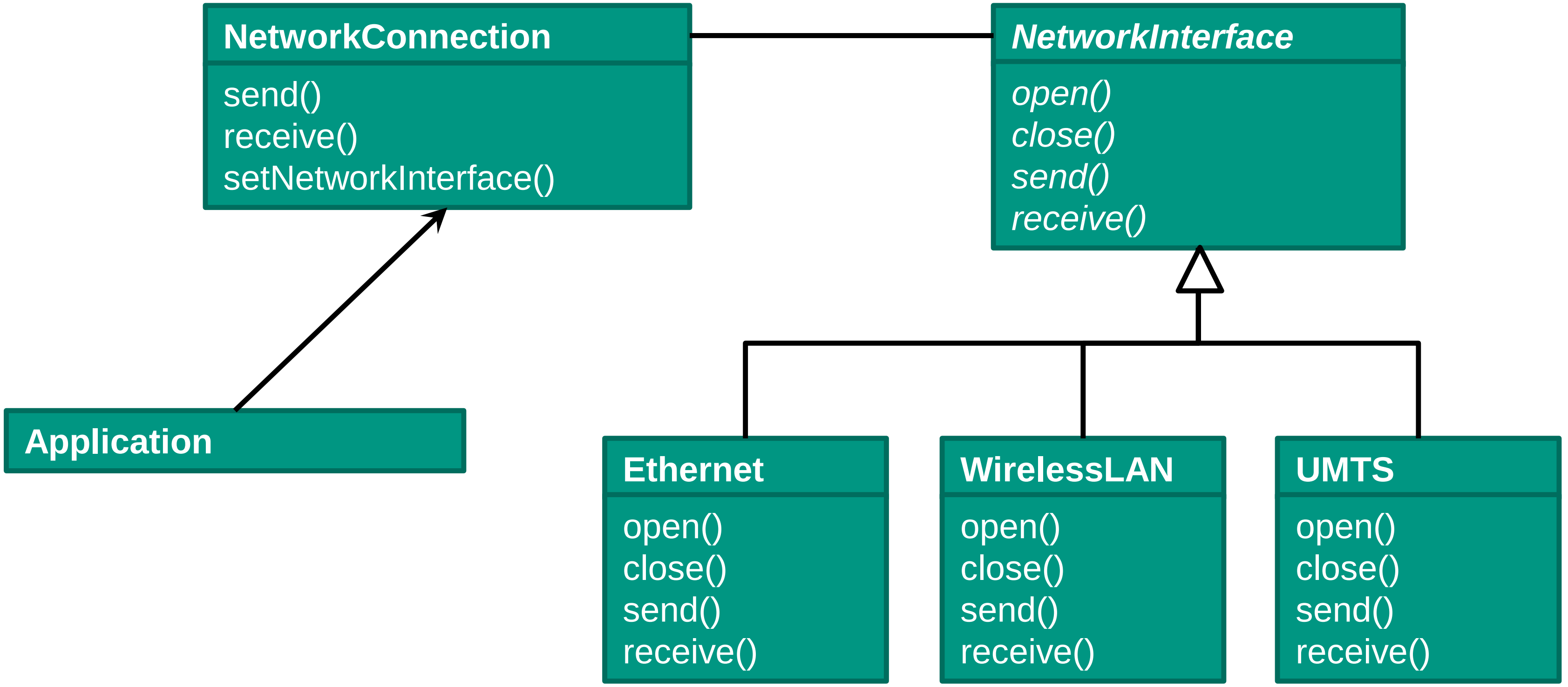


Strategy: Example 1



Strategy: Example 2

- Encapsulation of network interface



Strategy: Applicability

- Need to configure system with one of several possible realizations of the same logical behavior (as seen from the client).
- Need to adjust implementation depending on performance requirements of specific context
- Shield data structure details form clients
- Enables “switchless programming” when case distinctions would have to be made in several operations

Read more: Head First Design Patterns, Chapter 1

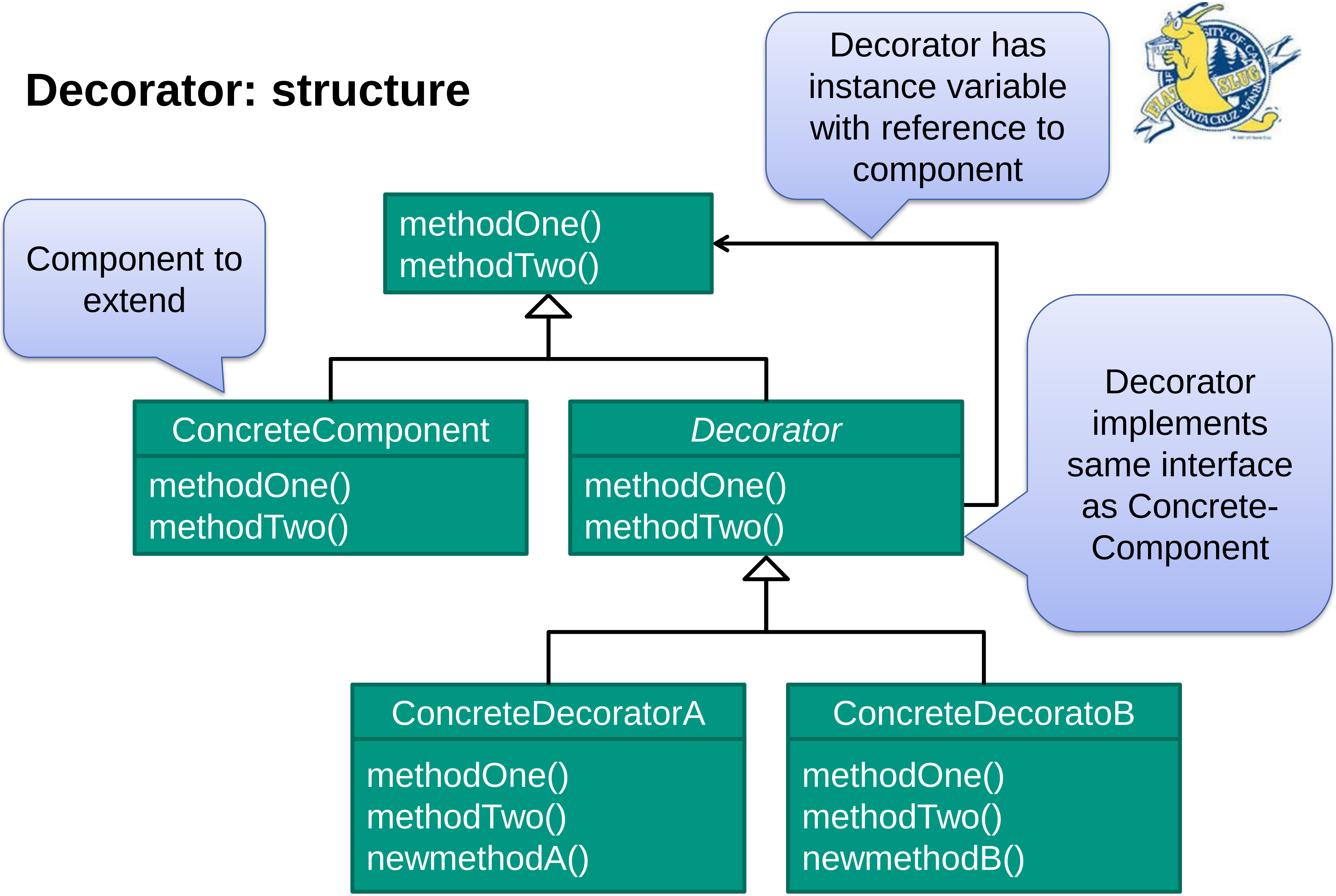
Applicability (2)

- Alternative is to specialize context for the needed cases
  - Subclasses would only differ in specific behavior
  - Fails to factor the common behavior
- Strategy supports dynamic change of behavior

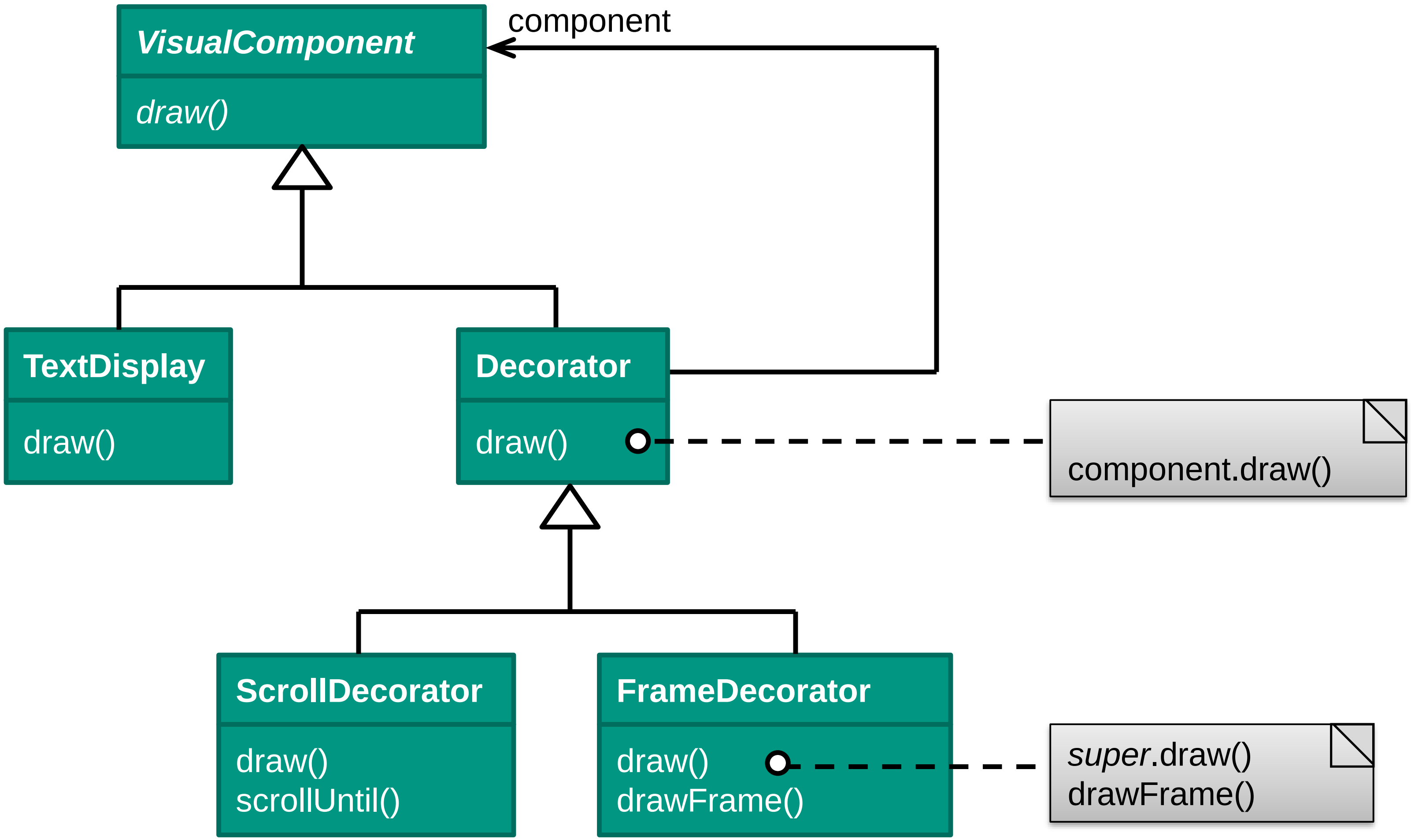
Decorator

- Purpose
  - Add dynamically functionality to an object
  - Alternative to inheritance
- Note: don't confuse with Proxy pattern

Decorator: structure



Decorator: Example



Decorator vs. Proxy

- Decorator
  - Adds functionality to object without changing it
  - Can extend the interface of object
- Proxy
  - Access control
  - Can be used just like the actual object
  - Can hide latency
  - Can hide methods
  - Represents specific actual object

Read more: Head First Design Patterns, page 472 ff.