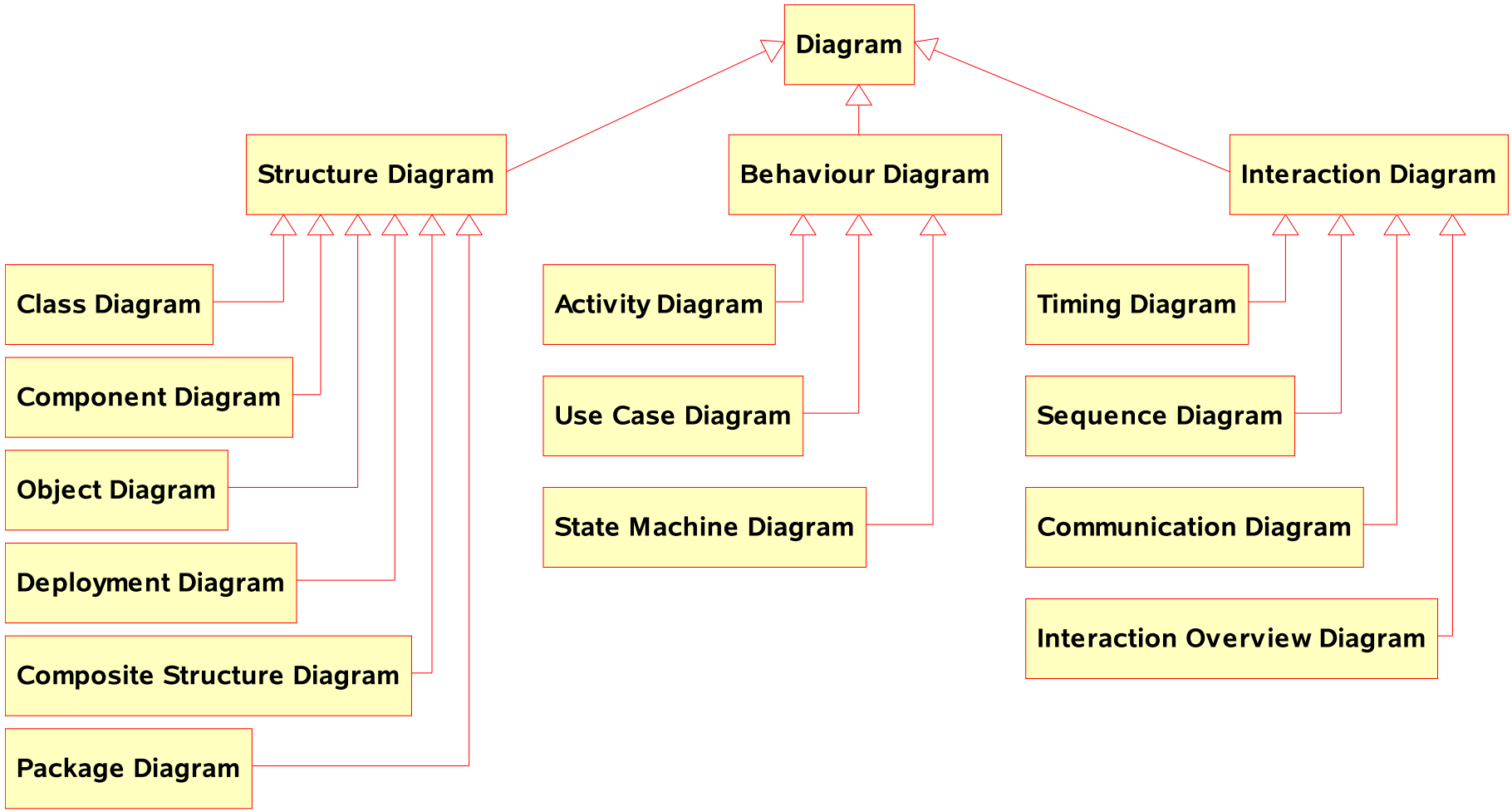
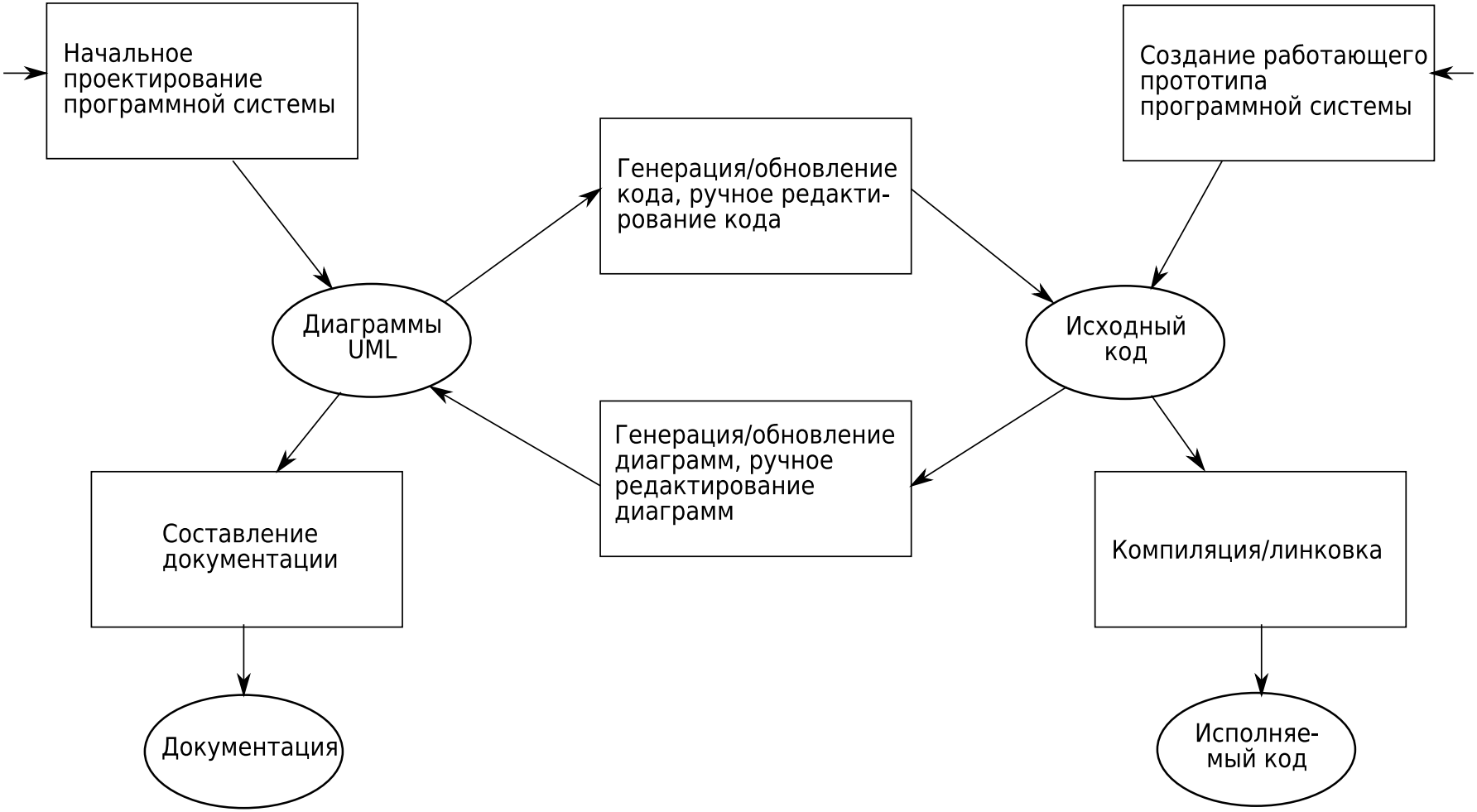
§1. Разработка программной системы с использованием UML

Unified Modeling Language (UML) – это стандартизованный язык моделирования общего назначения, используемый при разработке программного обеспечения.



Цикл разработки программной системы с использованием UML:



§2. Диаграммы классов

Определение. *Диаграмма классов* (class diagram) – диаграмма языка UML, на которой представлены классы с атрибутами и операциями, а также связывающие их отношения.

На диаграмме классов не указывается информация о временных аспектах функционирования системы.

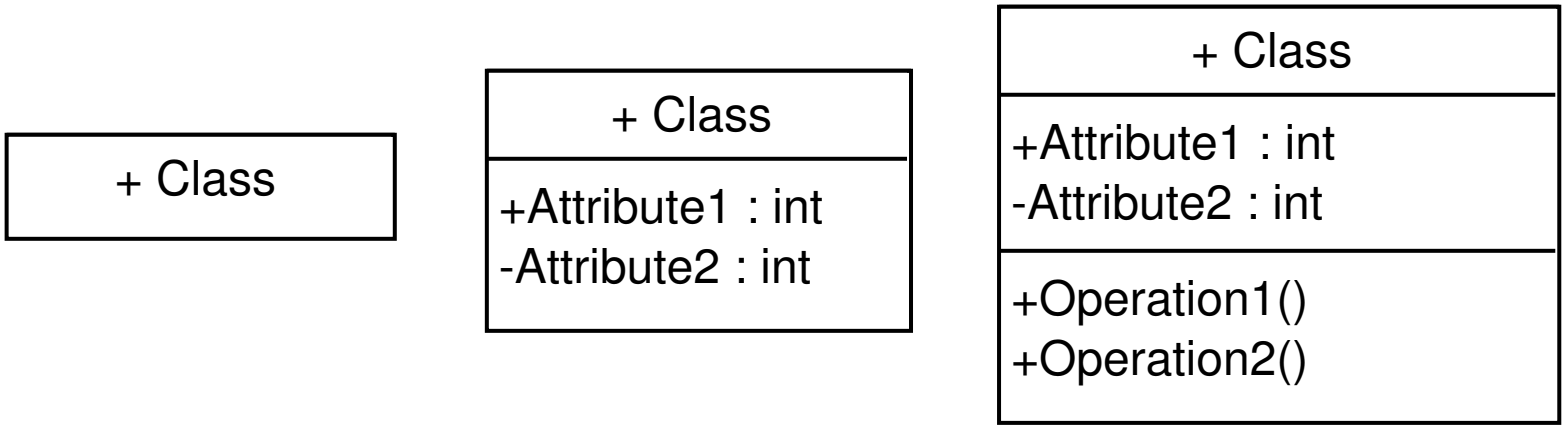
Определение. *Атрибут* (attribute) – это некий присущий объекту класса признак, позволяющий отличать один объект класса от другого.

При переводе модели на язык программирования атрибуты становятся свойствами или полями.

Определение. *Операция* (operation) – это сервис, предоставляемый каждым объектом класса по требованию своих клиентов, в качестве которых могут выступать другие объекты, в том числе и экземпляры данного класса.

Операциям класса соответствуют методы в программе.

Изображение класса в нотации UML:



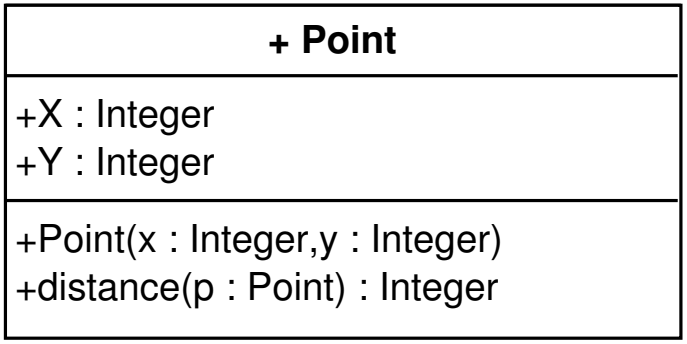
Имя абстрактного класса записывается курсивом.

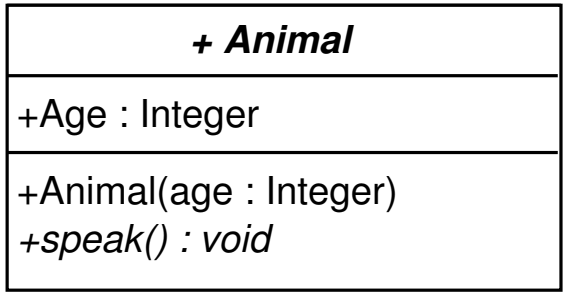
Перед именем интерфейса ставится так называемый *стереотип* «interface».

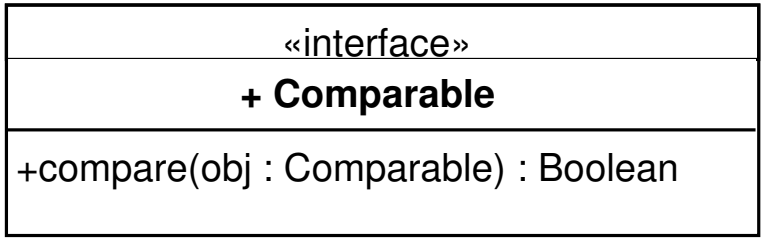
Кванторы видимости атрибутов и операций класса:

+	общедоступный (public)
#	защищённый (protected)
−	закрытый (private)
~	пакетный (package)

Примеры.







§3. Отношения между классами и объектами

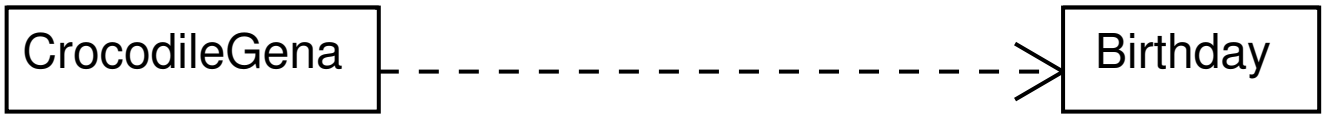
Отношение зависимости (dependency relationship).

Отношения на уровне объектов:
– отношение ассоциации (association relationship);
– отношение направленной ассоциации (directed association relationship);
– отношение агрегации (aggregation relationship);
– отношение композиции (composition relationship).

Отношения на уровне классов:
– отношение обобщения (generalization relationship);
– отношение реализации (realization relationship).

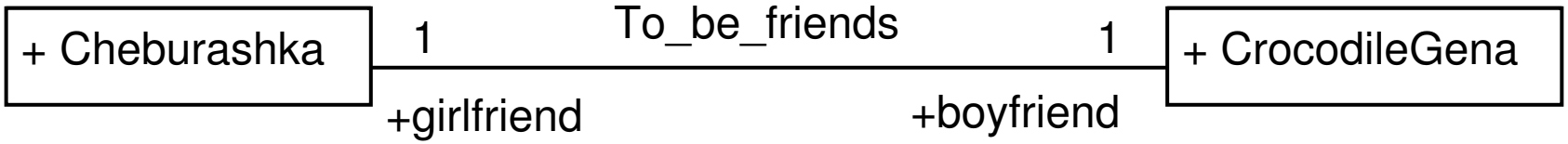
Определение. *Зависимость* (dependency) класса A от класса B – это самое слабое отношение, означающее, что реализация класса A тем или иным способом использует класс B.

Пример.



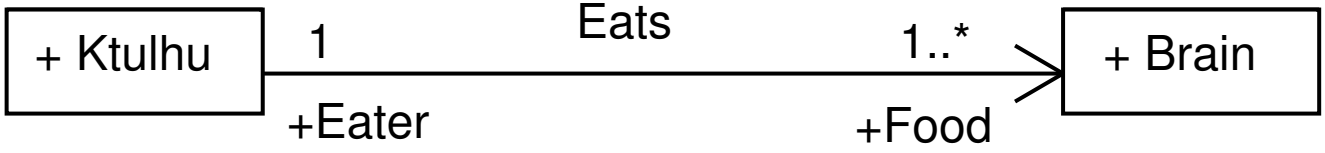
Определение. *Ассоциация* (association) между классами A и B – это отношение, означающее, что внутреннее состояние объекта класса A содержит ссылку на объект класса B (или массив ссылок на объекты класса B), и наоборот.

Пример.



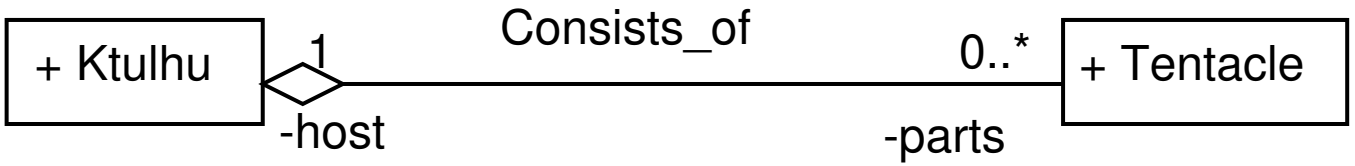
Определение. *Направленная ассоциация* (directed association) – это ассоциация между классами, выражающая несимметричную связь сущностей предметной области, соответствующих этим классам.

Пример.



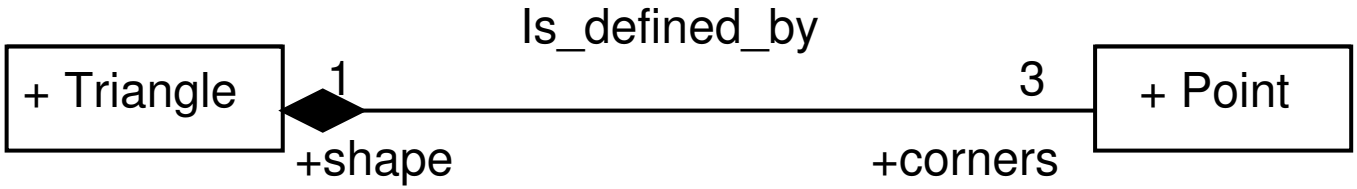
Определение. *Агрегация* (aggregation) объекта класса B объектом класса A – это направленная ассоциация, проведённая от класса A к классу B и выражающая тот факт, что сущность предметной области, соответствующая классу B, является составной частью сущности, соответствующей классу A.

Пример.



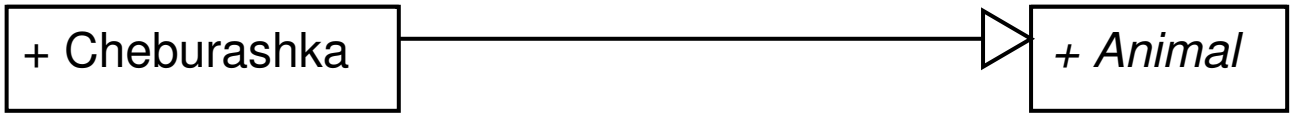
Определение. *Композиция* (composition) – это такая агрегация, при которой агрегируемый объект является неотъемлемой частью агрегирующего объекта.

Пример.



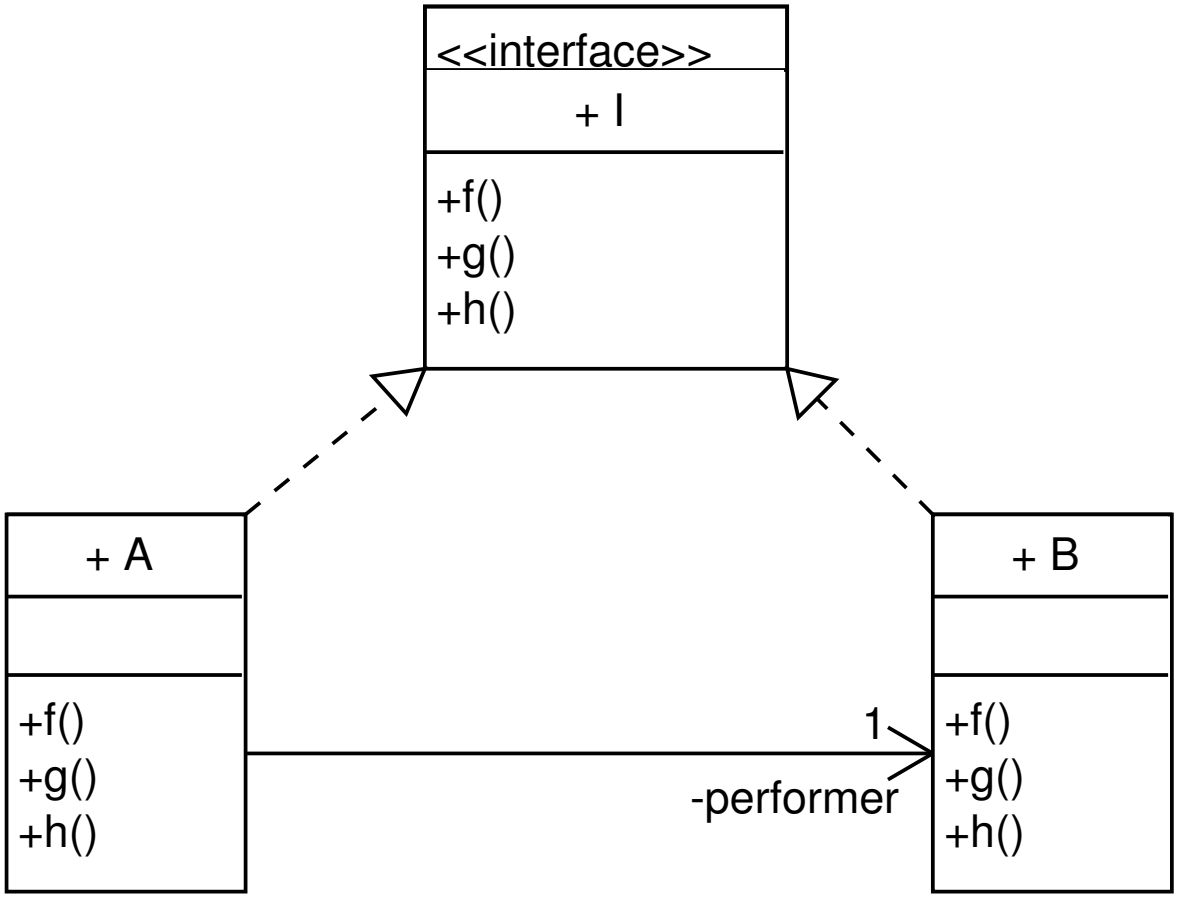
Определение. *Обобщение* (generalization) – это отношение классификации между более общим элементом модели (родителем или предком) и более частным или специальным элементом (дочерним или потомком).

Пример.

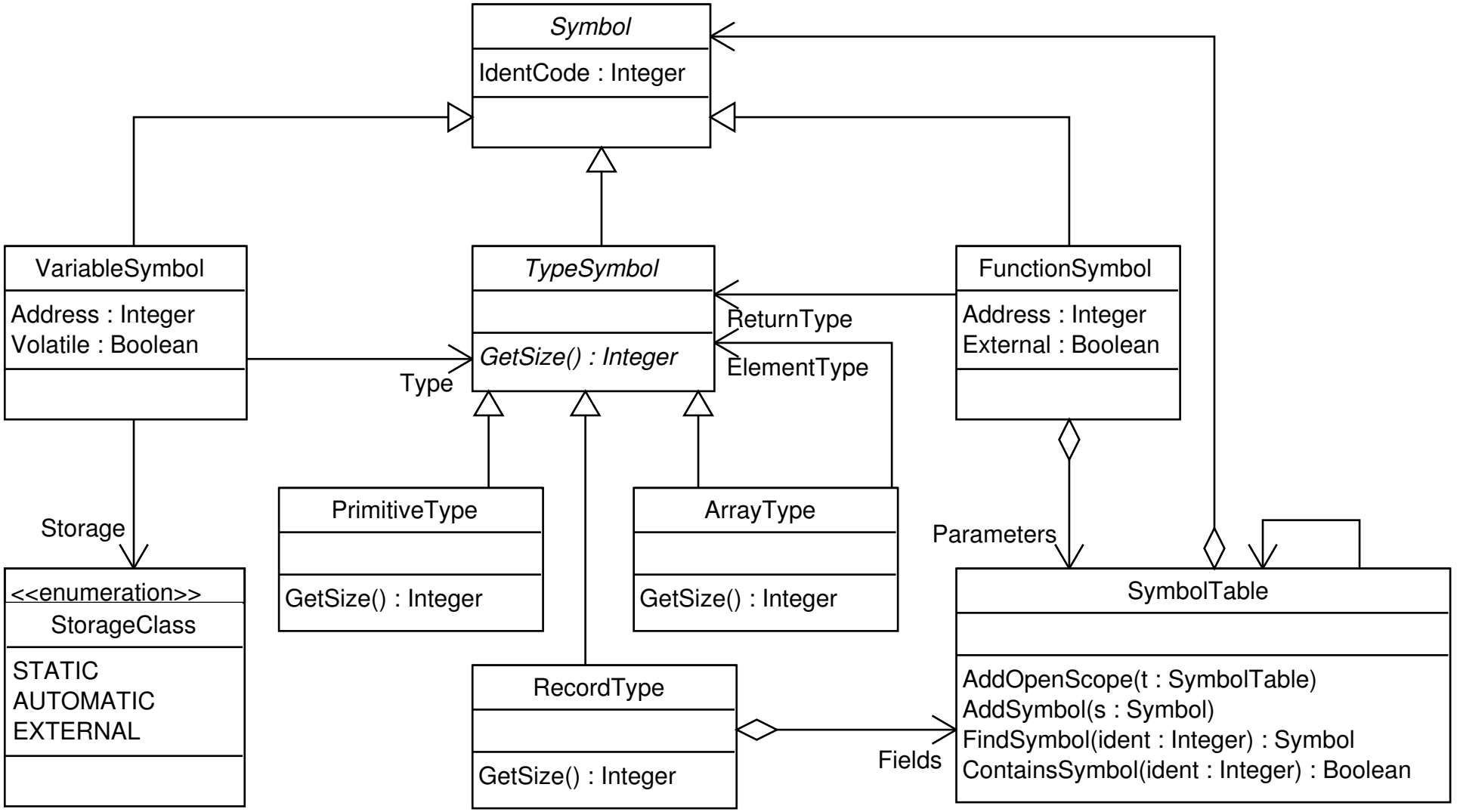


Определение. *Реализация* (realization) – это отношение между двумя элементами модели, при котором один элемент реализует поведение, определяемое другим элементом.

Пример.



Пример. (UML-диаграмма таблицы символов в компиляторе)



§4. Понятие образца проектирования. Образец Immutable

GoF (Gang of Four) – Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес.
Design Patterns. Elements of Reusable Object-Oriented Software.

Определение. *Образец проектирования* (design pattern) – это описание взаимодействующих объектов и классов, предназначенное для решения некоторой задачи проектирования общего характера, которая может возникать в различных частных ситуациях.

Образец. Если внутреннее состояние объекта некоторого класса защищено от изменений, то этот класс спроектирован в соответствии с образцом Immutable.

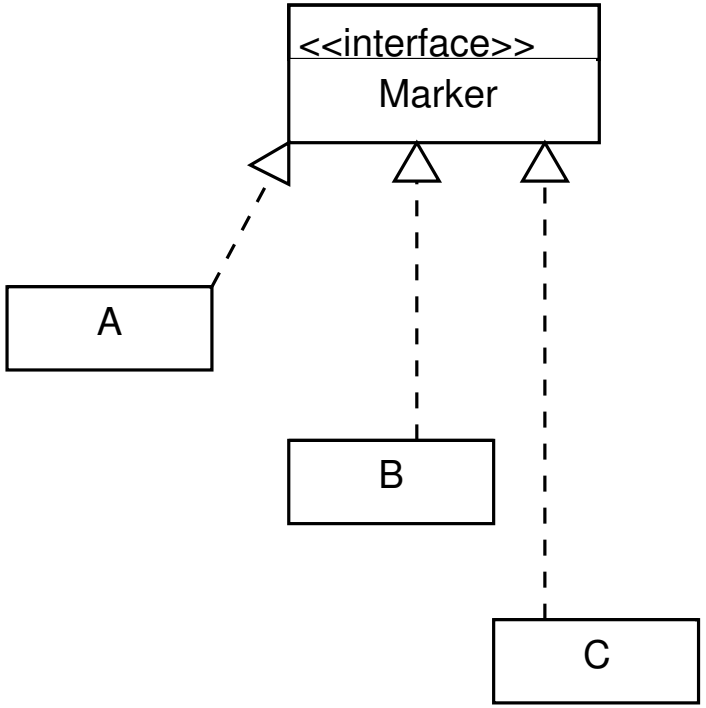
Образец Immutable подразумевает, что вместо изменения состояния объекта порождается новый объект с изменённым состоянием.

Пример.

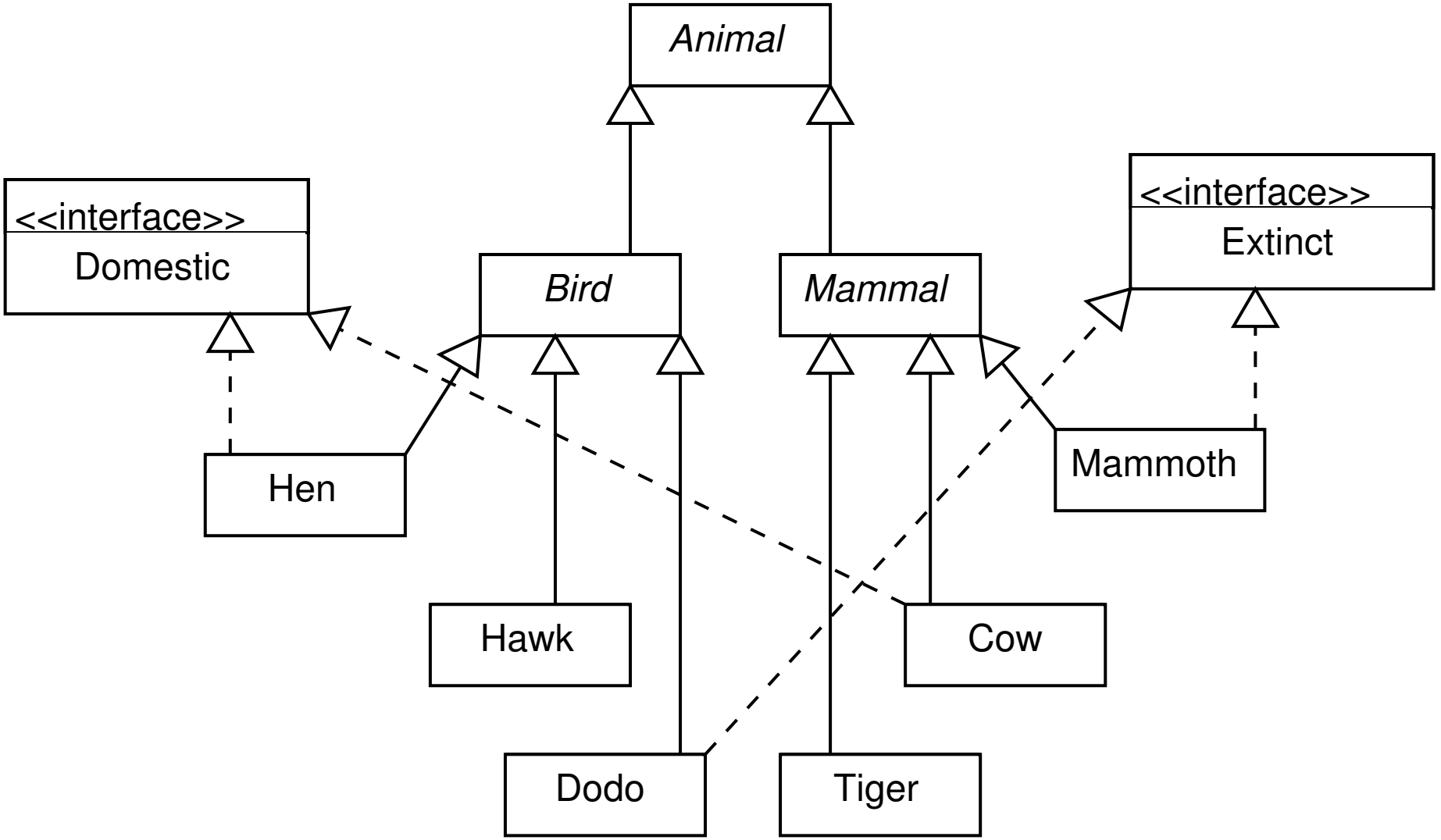
```
1  public class ImmutablePoint {
2          private int x, y;

4          public int getX() { return x; }
5          public int getY() { return y; }

7          public ImmutablePoint (int x, int y) {
8                  this.x = x;   this.y = y;
9          }

11         public ImmutablePoint changeX (int a_x) {
12                 return new ImmutablePoint(a_x, y);
13         }

15         public ImmutablePoint change_y (int a_y) {
16                 return new ImmutablePoint(x, a_y);
17         }
18 }
```

§5. Образец Marker Interface

Образец. Если все объекты некоторого набора классов имеют важную общую особенность, отличающую их от объектов остальных классов, то имеет смысл «пометить» каждый класс из этого набора с помощью специального вырожденного (т.е. пустого, не имеющего методов) интерфейса, реализовав тем самым образец Marker Interface.



Пример.


Animal
<<interface>>
Domestic
Bird
Mammal
<<interface>>
Extinct
Hen
Mammoth
Hawk
Cow
Dodo
Tiger

Пример.

```
1  public class Joke
2  {
3      static void talkAboutAnimal(Animal a)
4      {
5          if (a instanceof Extinct)
6              System.out.println("Всех уже съели :-(");
7          else
8              System.out.println
9                  ("Бери ложку, бери хлеб... :-)");
10     }
11 }
```

§6. Образцы Delegation и Proxy

Образец. Суть образца Delegation заключается в том, что объект класса A внешне выражает некоторое поведение, но в реальности передаёт ответственность за выполнение этого поведения объекту класса B.

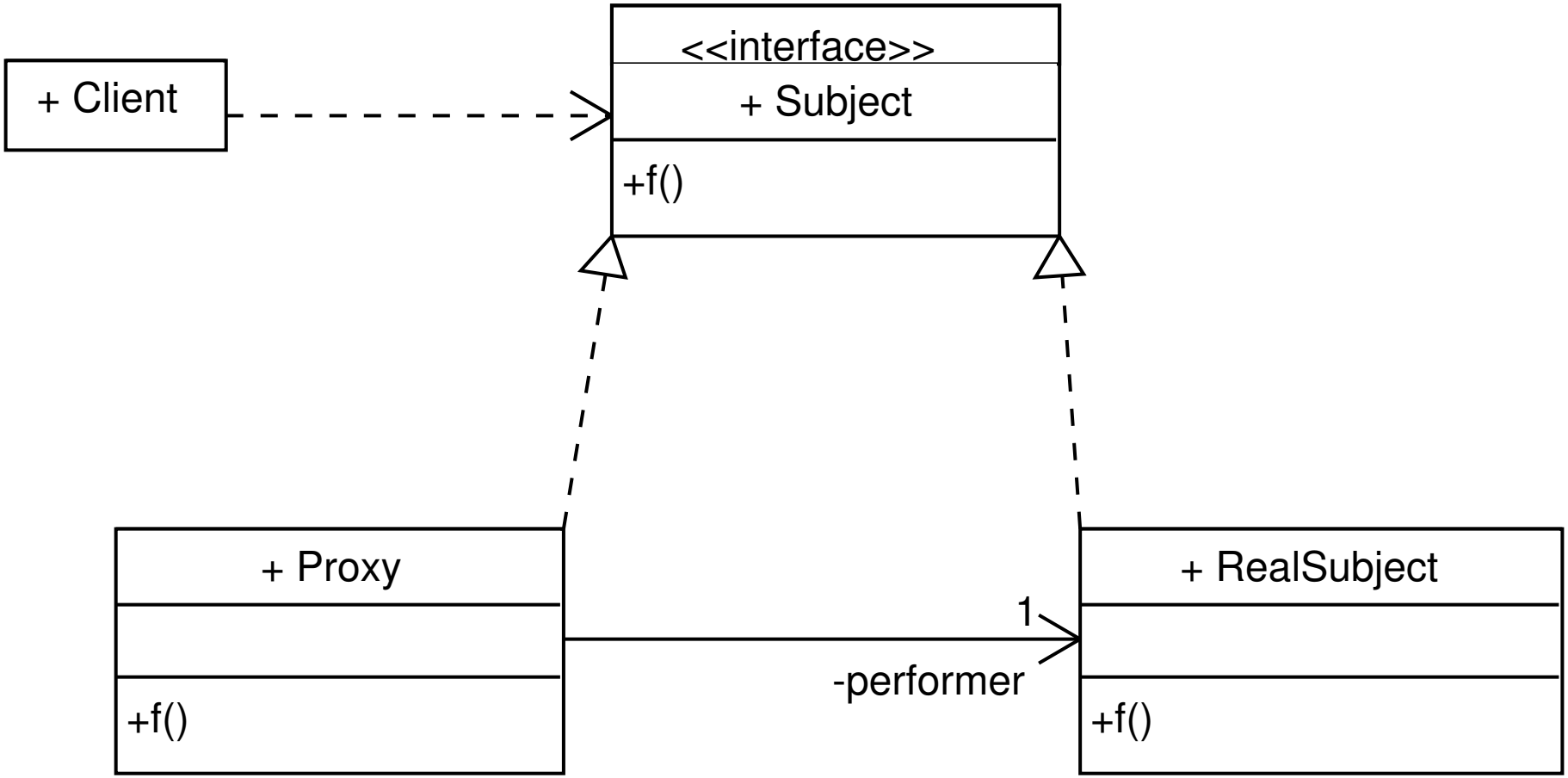
Пример.

```
1  public class CleverString {
2      private String s;
3      private int[] pi;

5      public CleverString(String s) {
6          this.s = s;
7          pi = new int[length()];
8          for (int i = 0, t = 0; i < length(); i++) {
9              while (t > 0 && charAt(t) != charAt(i)) {
10                 t = pi[t-1];
11             }
12             if (charAt(t) == charAt(i)) t++;
13             pi[i] = t;
14         }
15     }

17     ...
```

Пример. (продолжение)

```
15      ...

17      public int length() { return s.length(); }

19      public char charAt(int i) { return s.charAt(i); }

21      public CleverString substring(int i, int j) {
22          return new CleverString(s.substring(i, j));
23      }

25      public int borderAt(int i) { return pi[i]; }
26  }
```

Образец. Если целесообразно ограничить или как-то регламентировать доступ к некоторому объекту, используют образец проектирования Proxy, суть которого заключается в том, что доступ к объекту осуществляется не напрямую, а через объект-посредник.



Пример.

```
1  public class ListProxy implements List {
2      private List list;

4      public ListProxy(List list) { this.list = list; }

6      public Object get(int i) {
7          Object x = list.get(i);
8          System.out.println("get "+x+" at "+i);
9          return x;
10     }

12     public Object set(int i, Object elem) {
13         System.out.println("set "+elem+" to "+i);
14         return list.set(i, elem);
15     }

17     ...
18 }
```

Пример. (продолжение)

```
1  public class MainClass {
2      public static void main(String args[]) {
3          List a = Arrays.asList(7, 4, 5, 2, 1, 8);
4          List pa = new ListProxy(a);
5          for (int i = 0; i < pa.size()-1; i++) {
6              int k = i;
7              for (int j = i+1; j < pa.size(); j++) {
8                  Integer x = (Integer)pa.get(j),
9                          y = (Integer)pa.get(k);
10                 if (x.compareTo(y) < 0) k = j;
11             }
12             Object tmp = pa.get(i);
13             pa.set(i, pa.get(k));
14             pa.set(k, tmp);
15         }
16         for (Object x: pa) System.out.print(" "+x);
17         System.out.println();
18     }
19 }
```

§7. Образец Dependency Injection

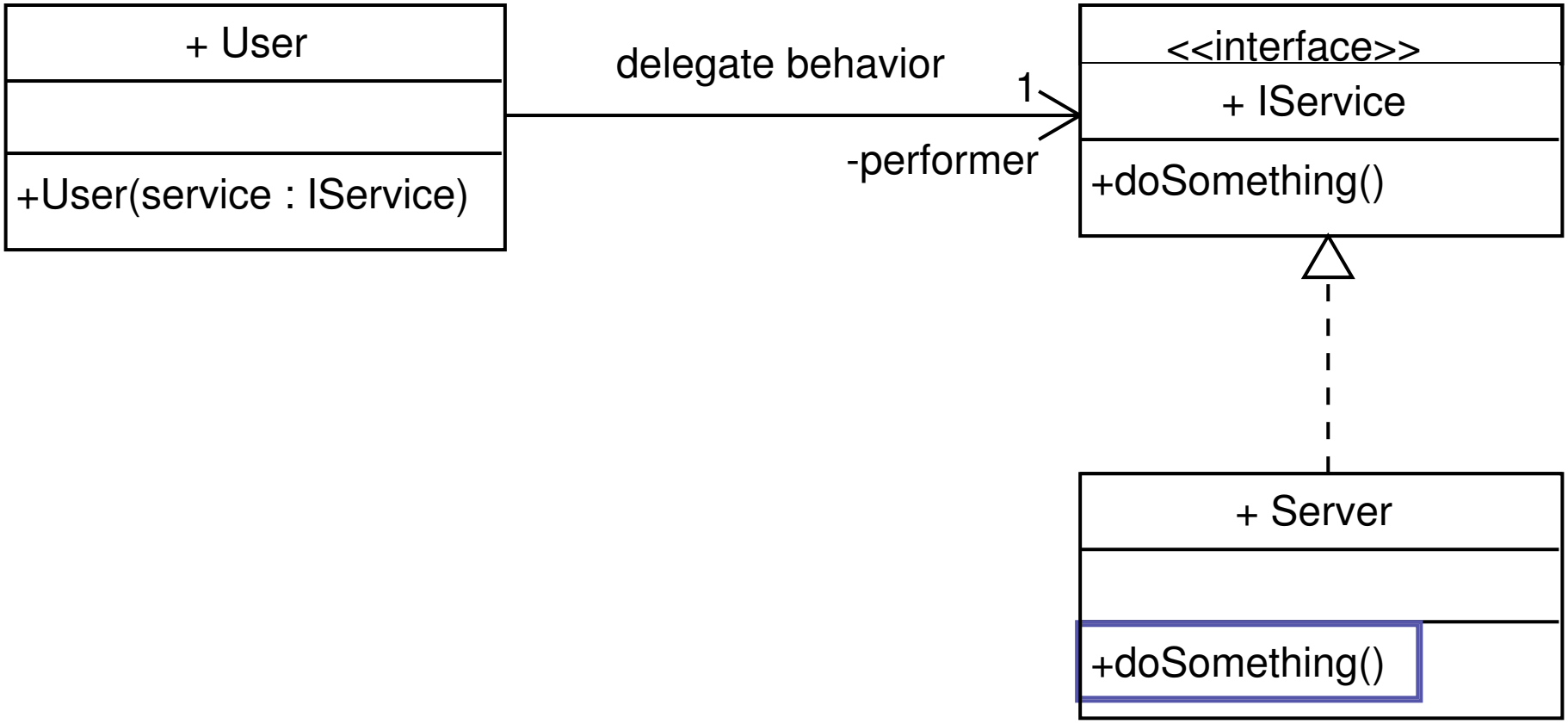
Образец. Чтобы обеспечить делегирование некоторого поведения заранее неизвестному объекту, используют образец Dependency Injection, который имеет три формы:

1. Constructor Injection;

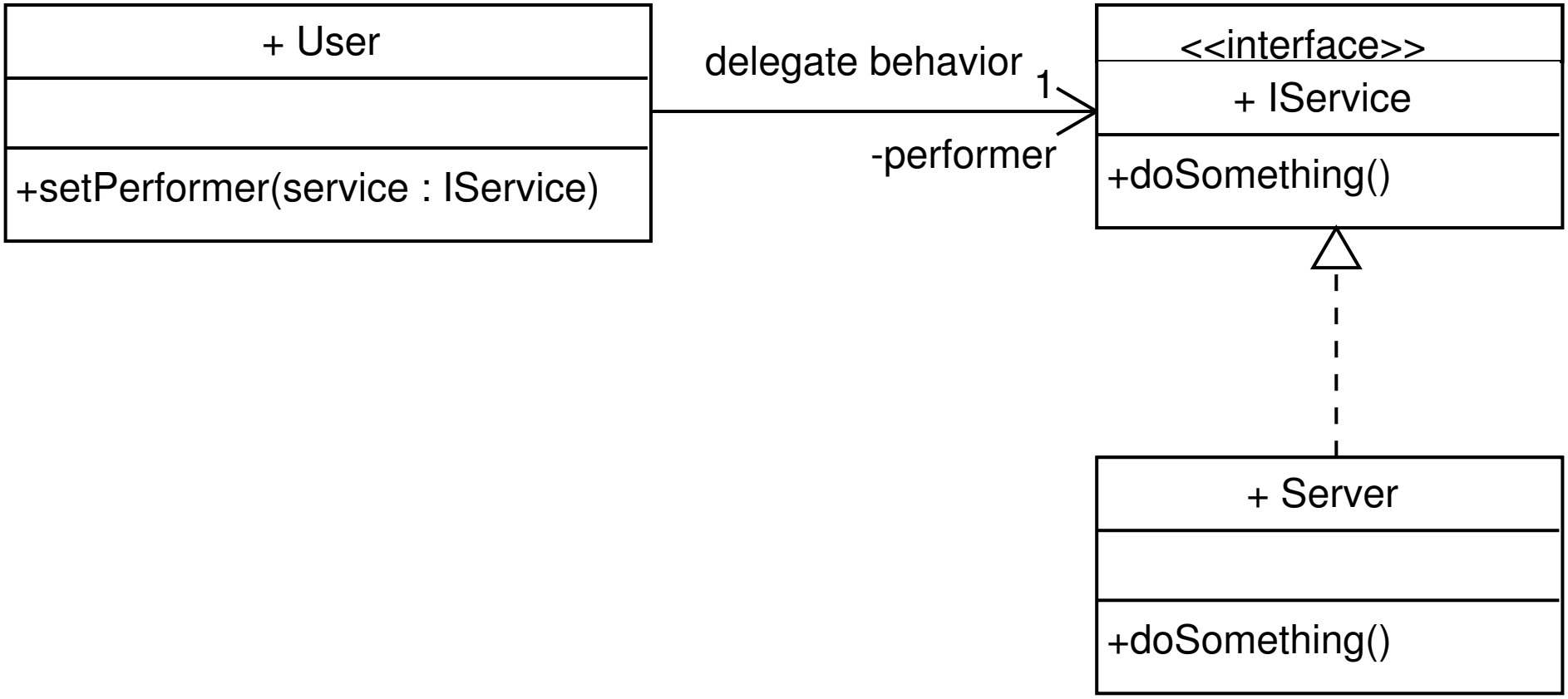
2. Setter Injection;

3. Interface Injection.

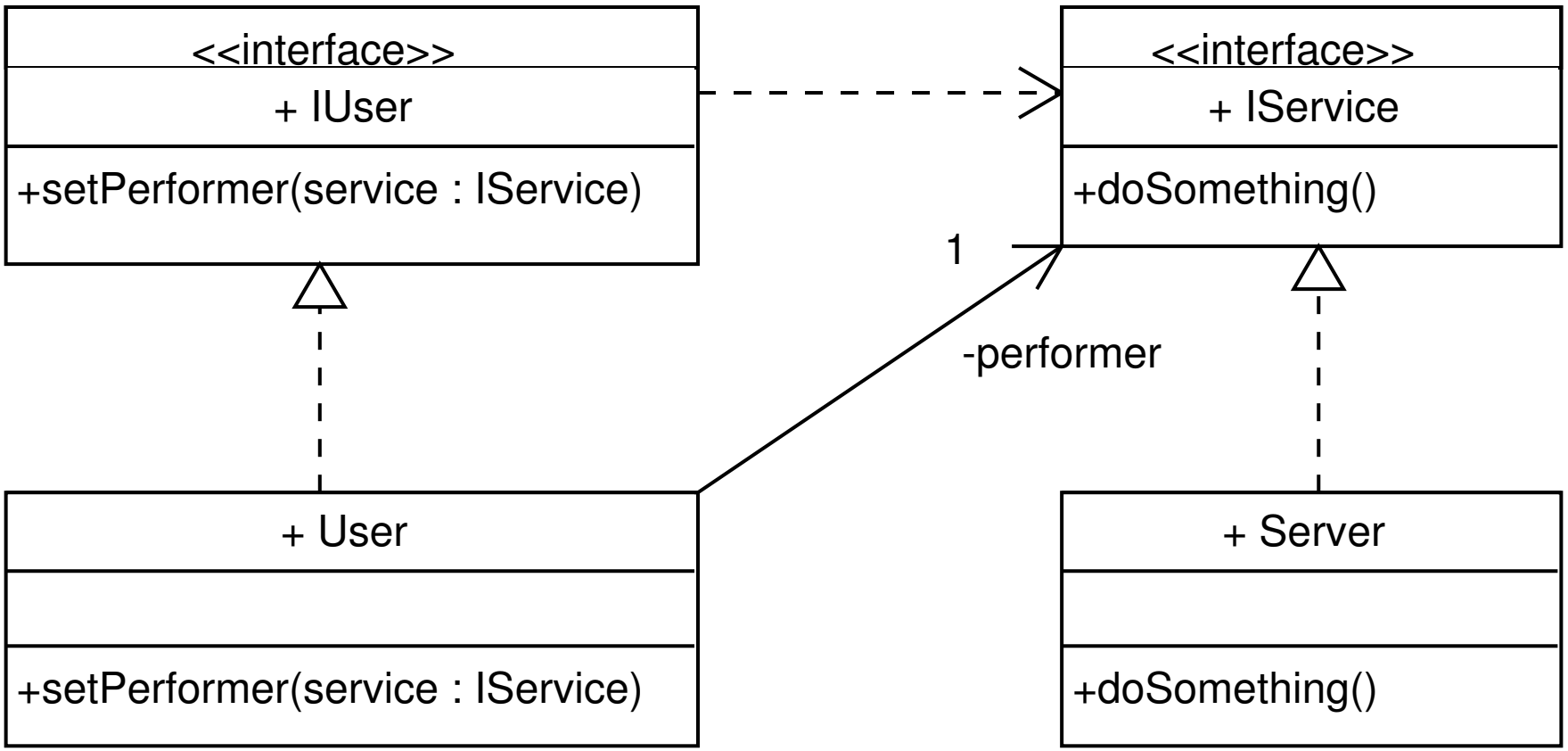
Constructor Injection



Setter Injection



Interface Injection

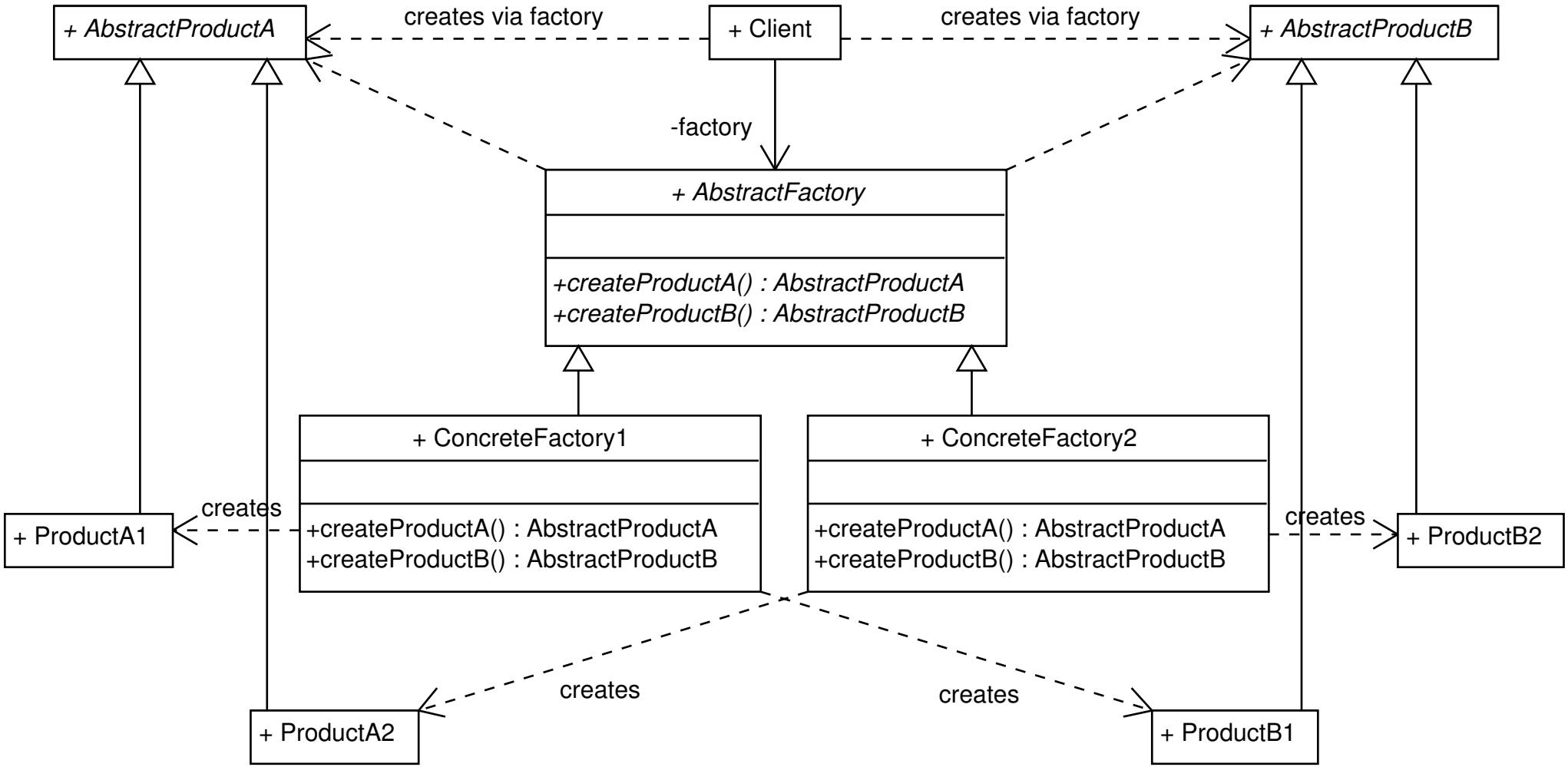


§8. Понятие порождающего образца. Образец Abstract Factory

Определение. *Порождающий образец* – это образец проектирования, решающий следующую задачу: как сделать некоторый фрагмент кода, в котором создаются объекты и устанавливаются связи между объектами, независимым от классов этих объектов.

Мы рассмотрим три порождающих образца:
– Abstract Factory;
– Builder;
– Singleton.

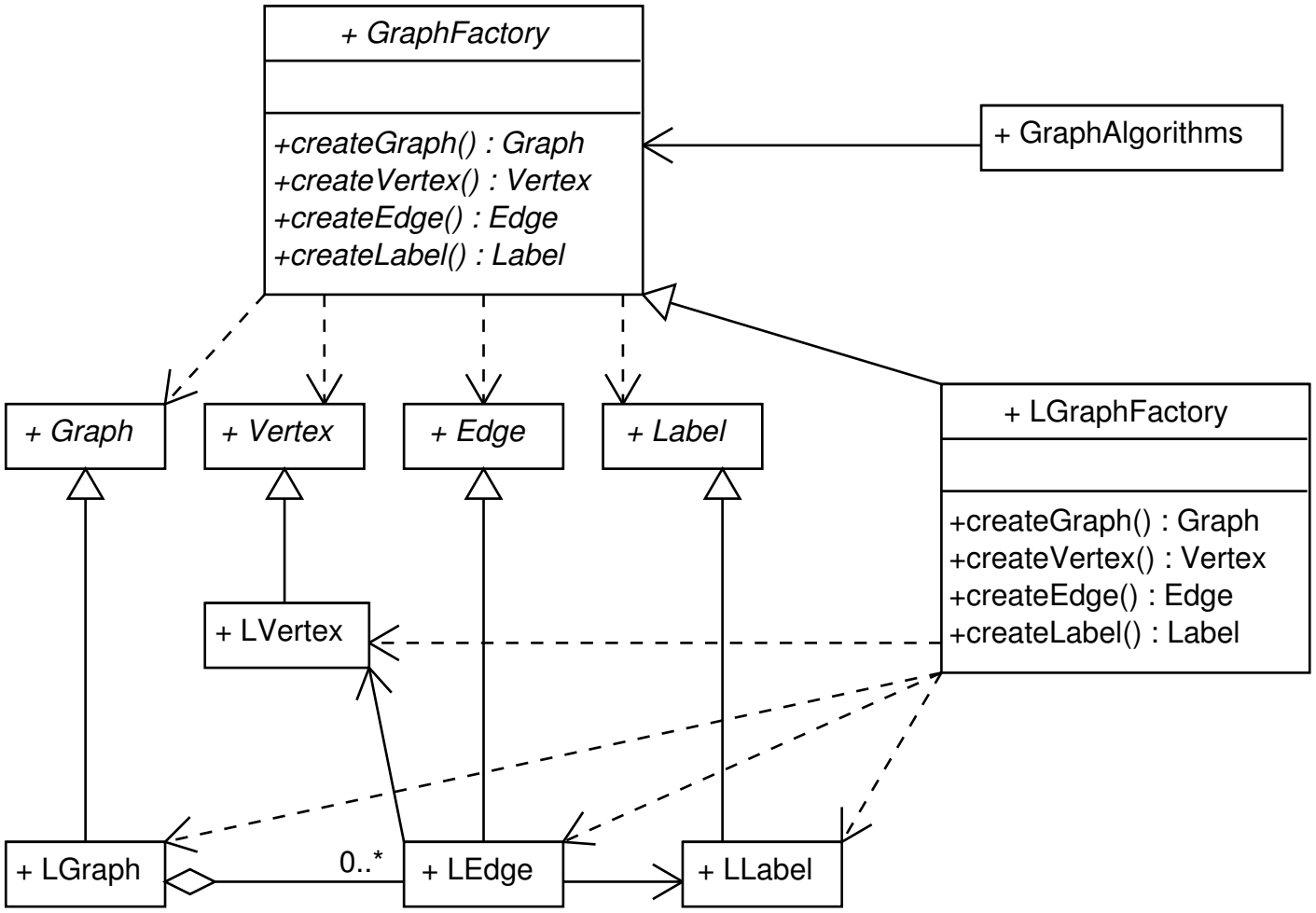
Образец. Abstract Factory предоставляет интерфейс для создания объектов семейства заранее неизвестных классов.



Образец Abstract Factory применяется в случае, если:

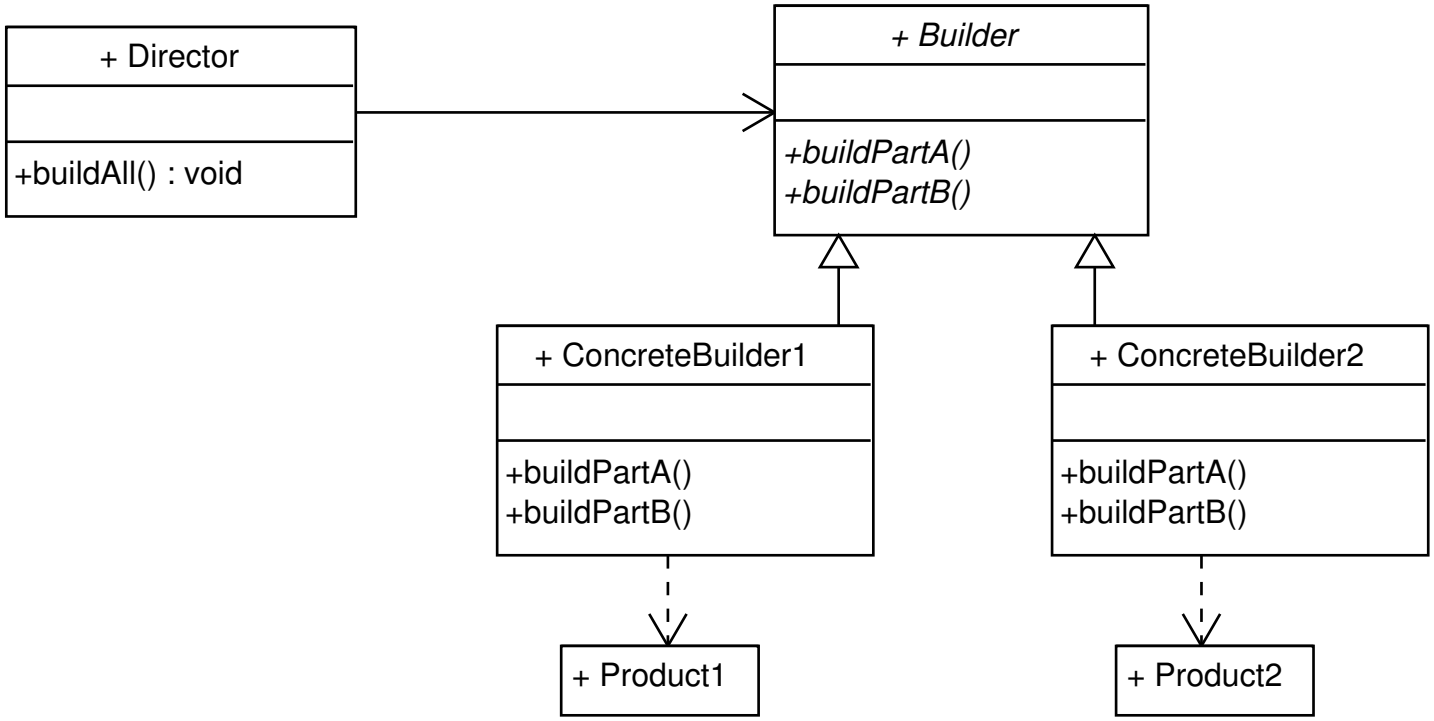
- система не должна зависеть от способа создания и внутреннего представления объектов, которые она создаёт;
- существует несколько семейств классов объектов, которые может создавать система; система должна допускать настройку на работу с одним из этих семейств;
- семейство взаимосвязанных классов спроектировано с расчётом на то, что эти классы будут использоваться вместе, и необходимо обеспечить выполнение этого ограничения;
- вы планируете оформить классы, объекты которых создаются в системе, в виде отдельной библиотеки, раскрывая только их интерфейсы, но не реализацию.

Пример. Необходимо разработать библиотеку классов для работы с размеченными графами, независимую от способа представления графов.



§9. Образец Builder

Образец. Builder отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления.

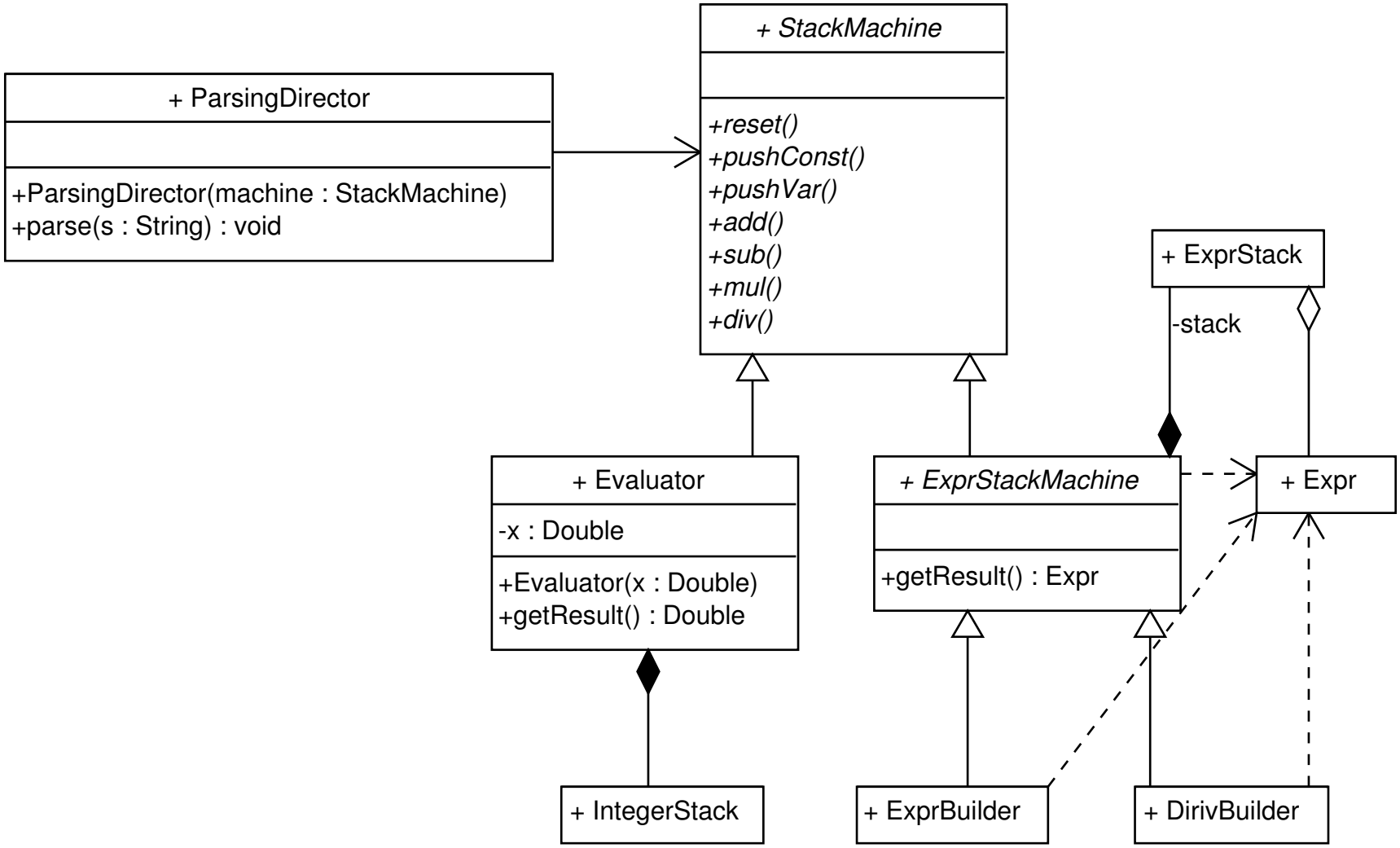


Образец Builder применяется в случае, если:

- алгоритм создания сложного объекта должен быть независим от частей, из которых состоит объект, и от того, как они собраны;

- процесс создания должен допускать получение различных представлений создаваемого объекта.

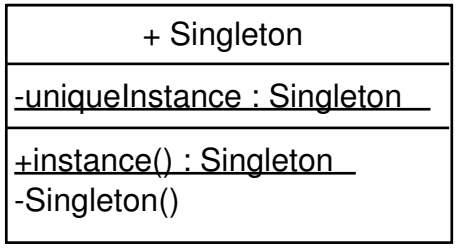
Пример. Необходимо разработать библиотеку классов, осуществляющих работу с арифметическими выражениями.



§10. Образец Singleton

Большие трудности при отладке приложения вызывает ошибка, связанная с существованием нескольких объектов, контролирующих один и тот же ресурс (конфигурационный файл, базу данных, сетевое соединение и т.п.).

Образец. Singleton гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.



Примечание. Статические атрибуты и операции обозначаются на диаграммах классов с помощью подчёркивания.

Пример.

```
1  public final class Singleton {
2      private static Singleton uniqueInstance;

4      public static Singleton Instance() {
5          if (uniqueInstance == null)
6              uniqueInstance = new Singleton();

8          return uniqueInstance;
9      }

11     private Singleton() { ... }
12     ...
13 }
```

§11. Понятие структурного образца. Образец Adapter

Определение. *Структурный образец* – это образец проектирования, показывающий, как из классов и объектов получаются более сложные структуры.

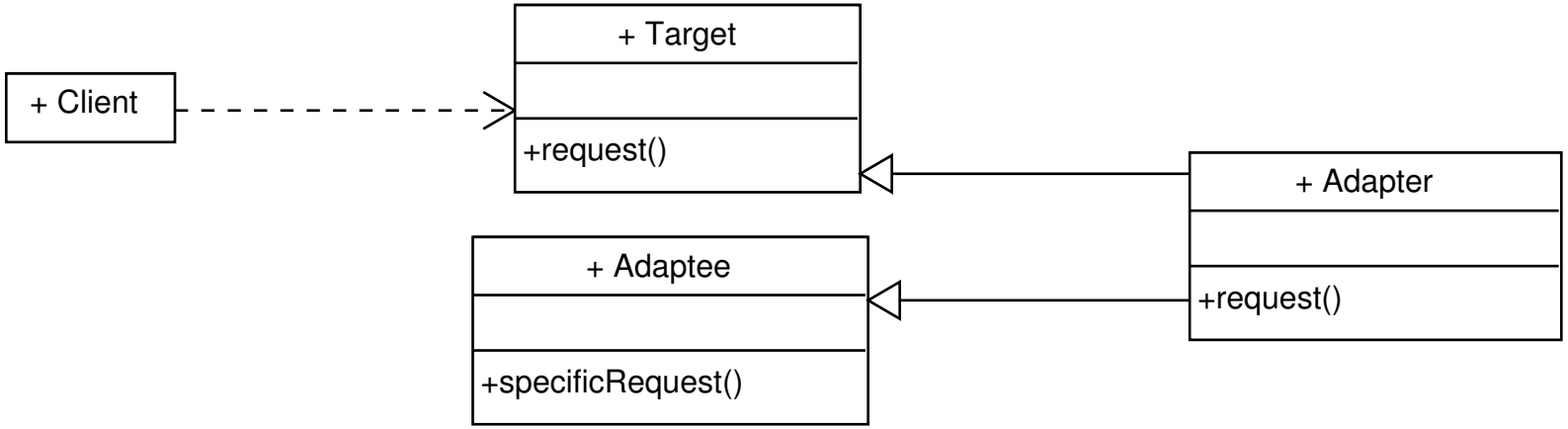
Мы рассмотрим четыре структурных образца:
– Adapter;
– Composite;
– Façade;
– Decorator.

Образец. Adapter (Wrapper) преобразует интерфейс некоторого класса в интерфейс, который ожидают клиенты. Тем самым Adapter обеспечивает совместную работу классов с несовместимыми интерфейсами и имеет две формы:

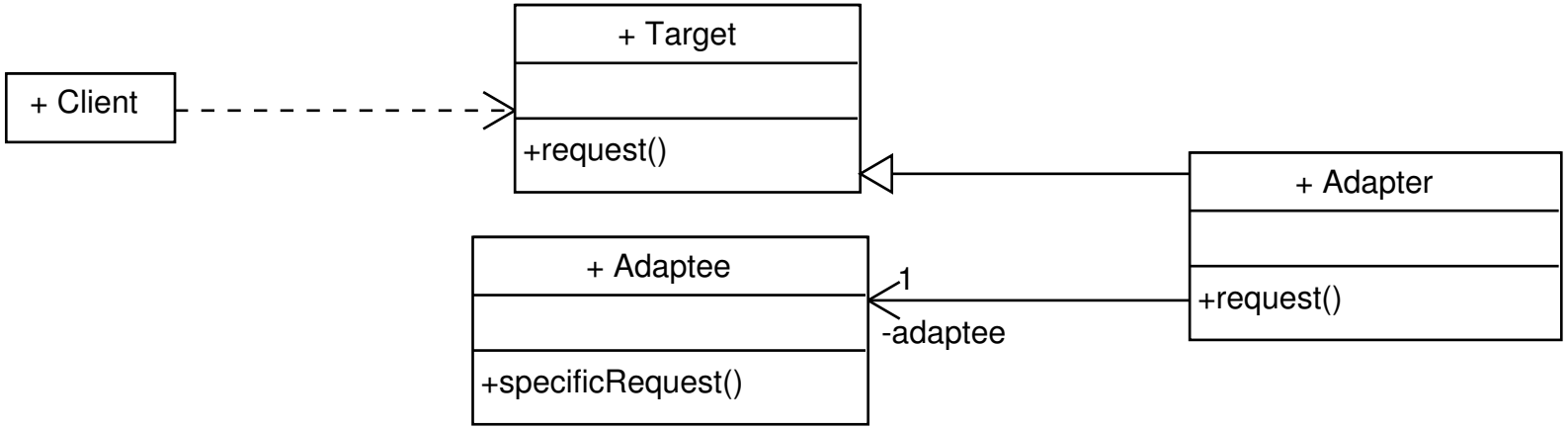
1. Class Adapter;

2. Object Adapter.

Class Adapter

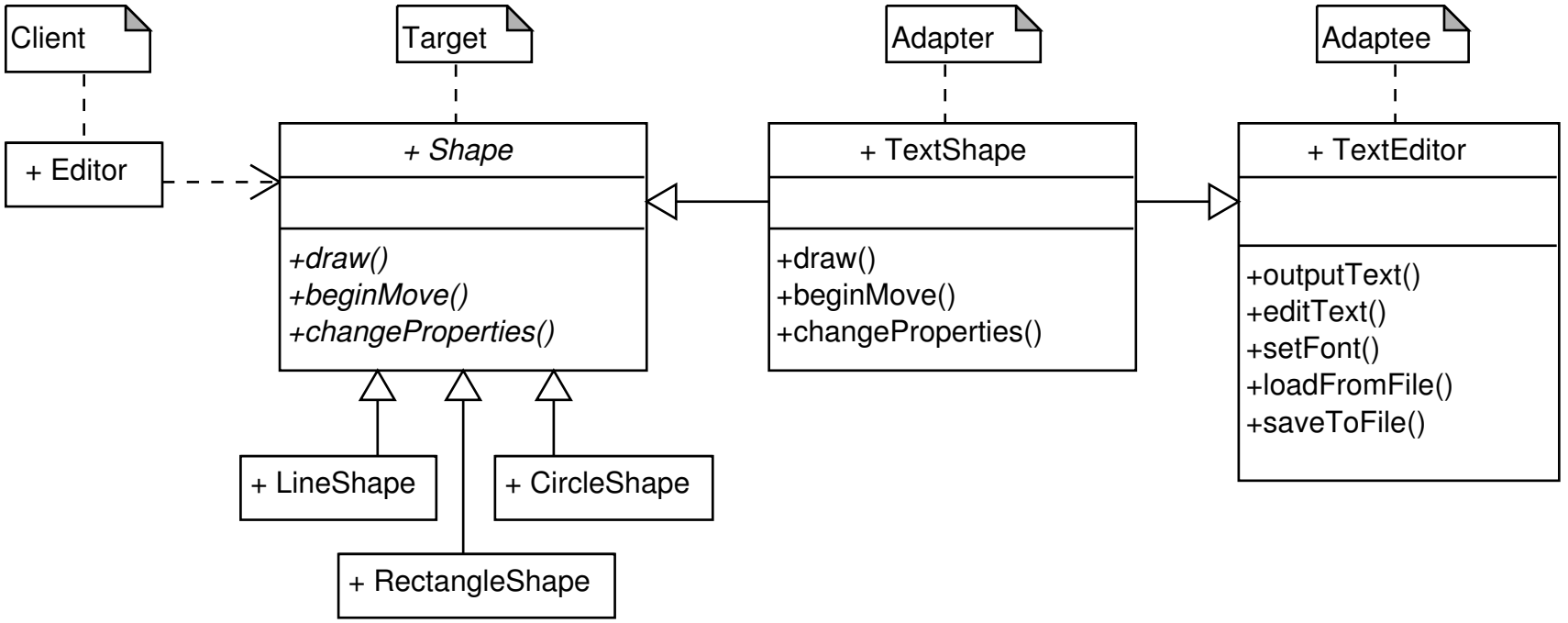


Object Adapter



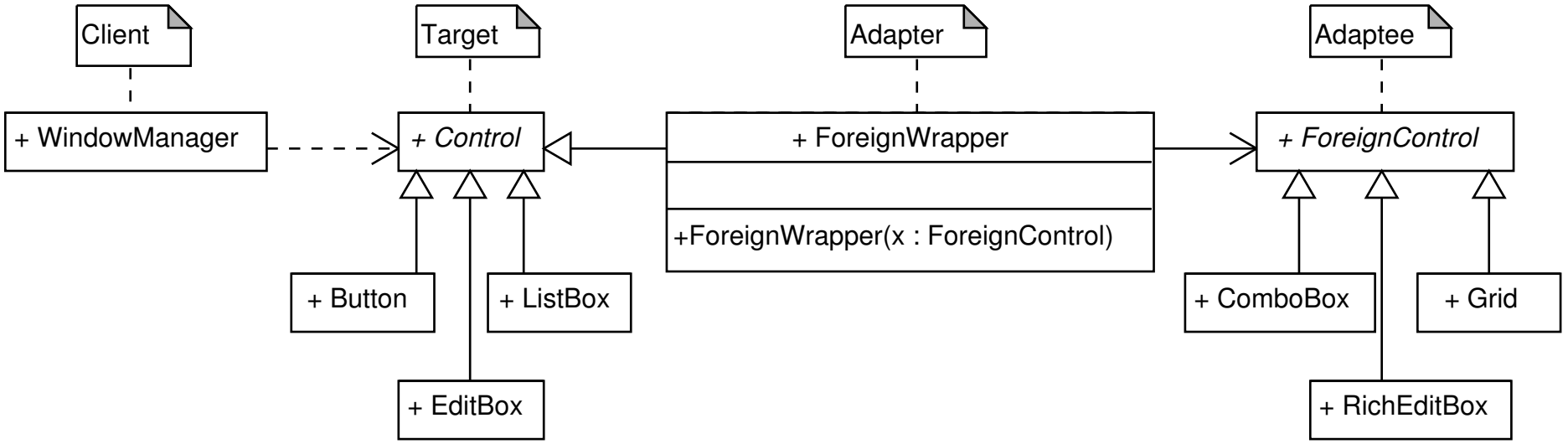
Образец Adapter применяется в случае, если необходимо использовать существующий класс, но его интерфейс не вписывается в архитектуру проектируемой системы.

Пример. (Class Adapter) Адаптирование библиотечного класса TextEditor в иерархию фигур, с которыми работает графический редактор.



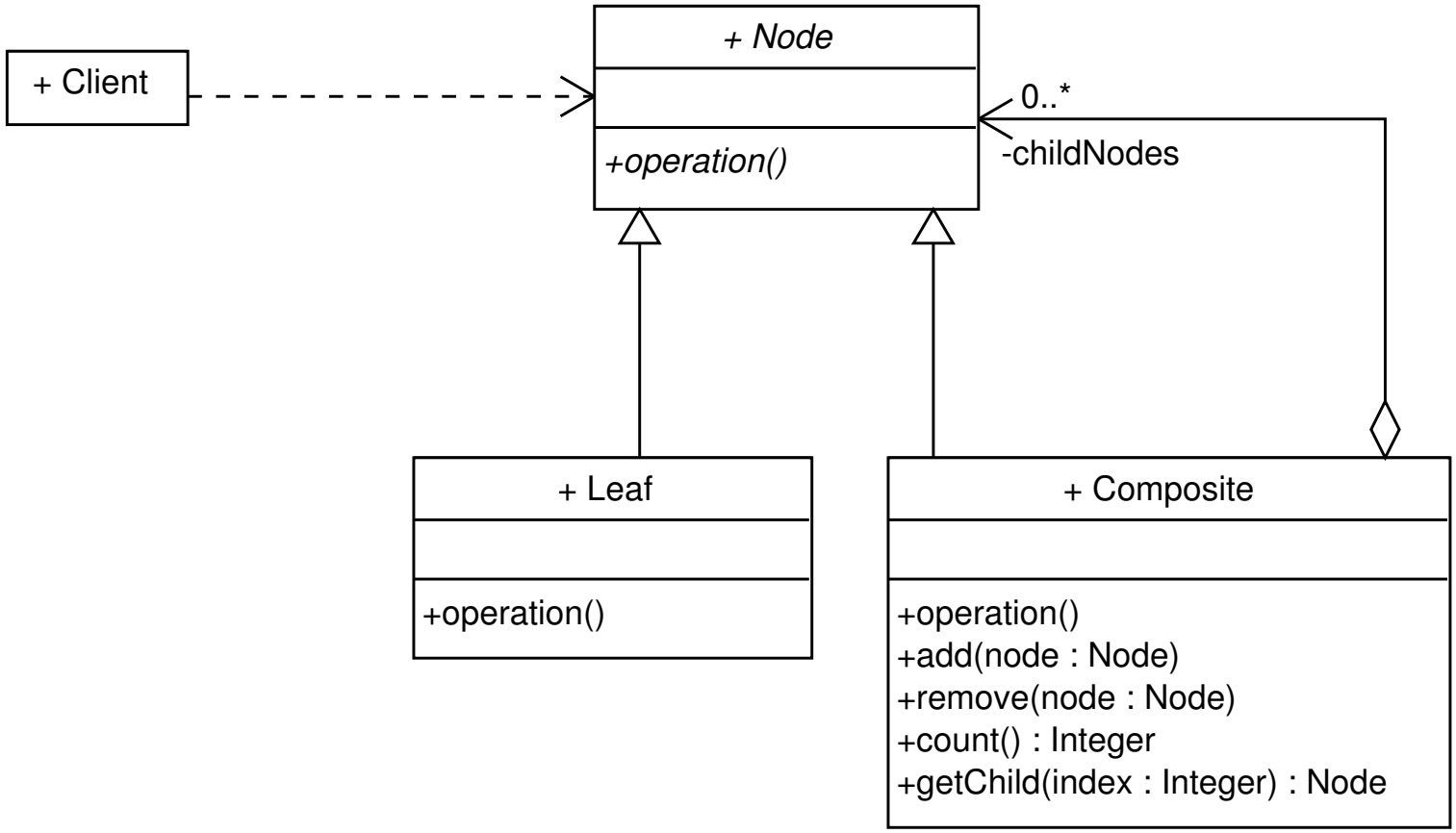
Следует использовать Object Adapter, а не Class Adapter, в следующей ситуации: имеется базовый класс X и несколько его подклассов X_1, X_2, ... , X_n, и нужно адаптировать сразу все эти подклассы. В этом случае непрактично порождать от каждого X_i новый класс Y_i, имеющий нужный интерфейс. Вместо этого лучше адаптировать интерфейс класса X по образцу Object Adapter.

Пример. (Object Adapter) Графический пользовательский интерфейс: адаптирование элементов управления из другой оконной библиотеки.



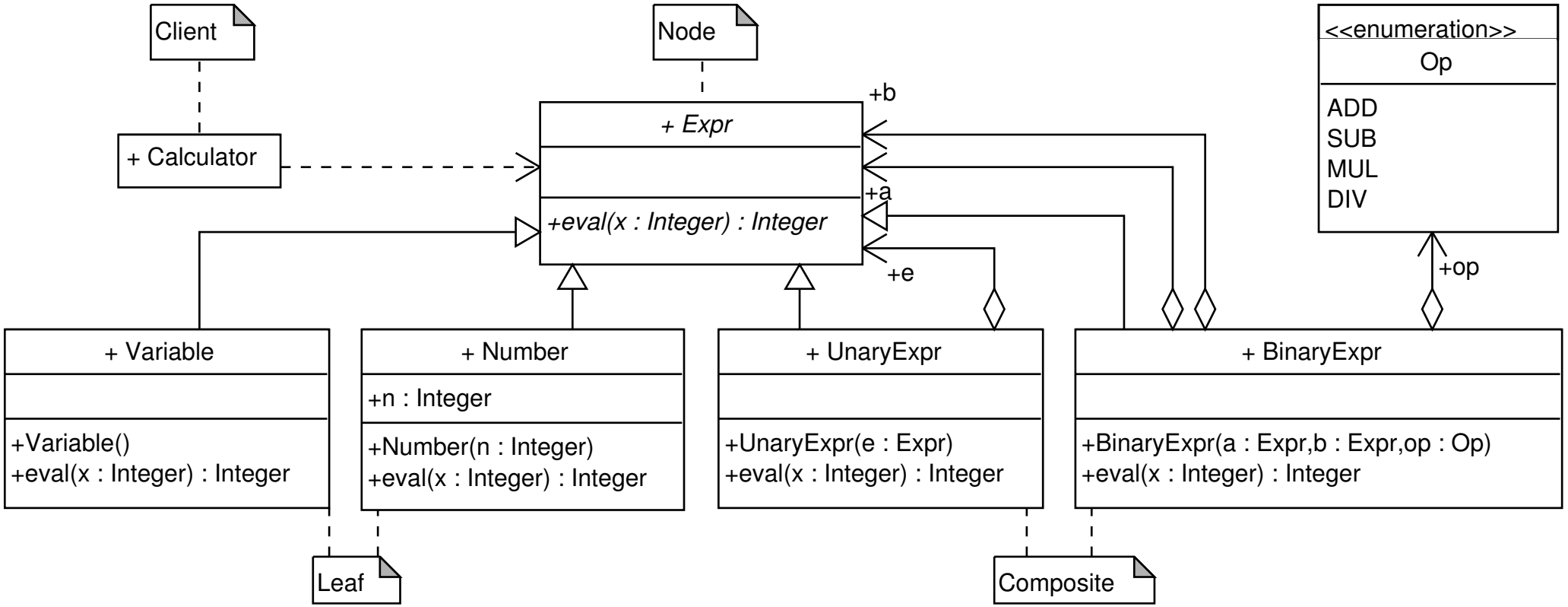
§12. Образец Composite

Образец. Composite компонует объекты в древовидные структуры для представления иерархий *часть-целое*.



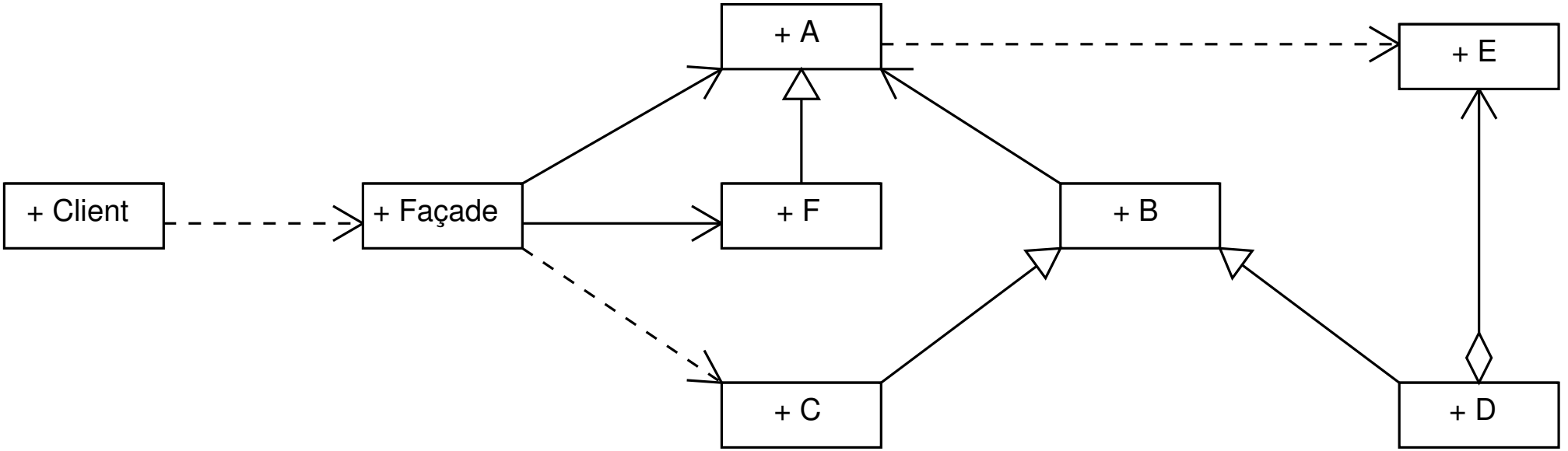
Образец Composite применяется в случае, если:
– нужно представить иерархию объектов вида *часть-целое*;
– клиенты должны единообразно обрабатывать составные и индивидуальные объекты.

Пример. Арифметические выражения.



§13. Образец Façade

Образец. Façade упрощает использование некоторой подсистемы, определяя единый интерфейс, через который клиент может работать с подсистемой.



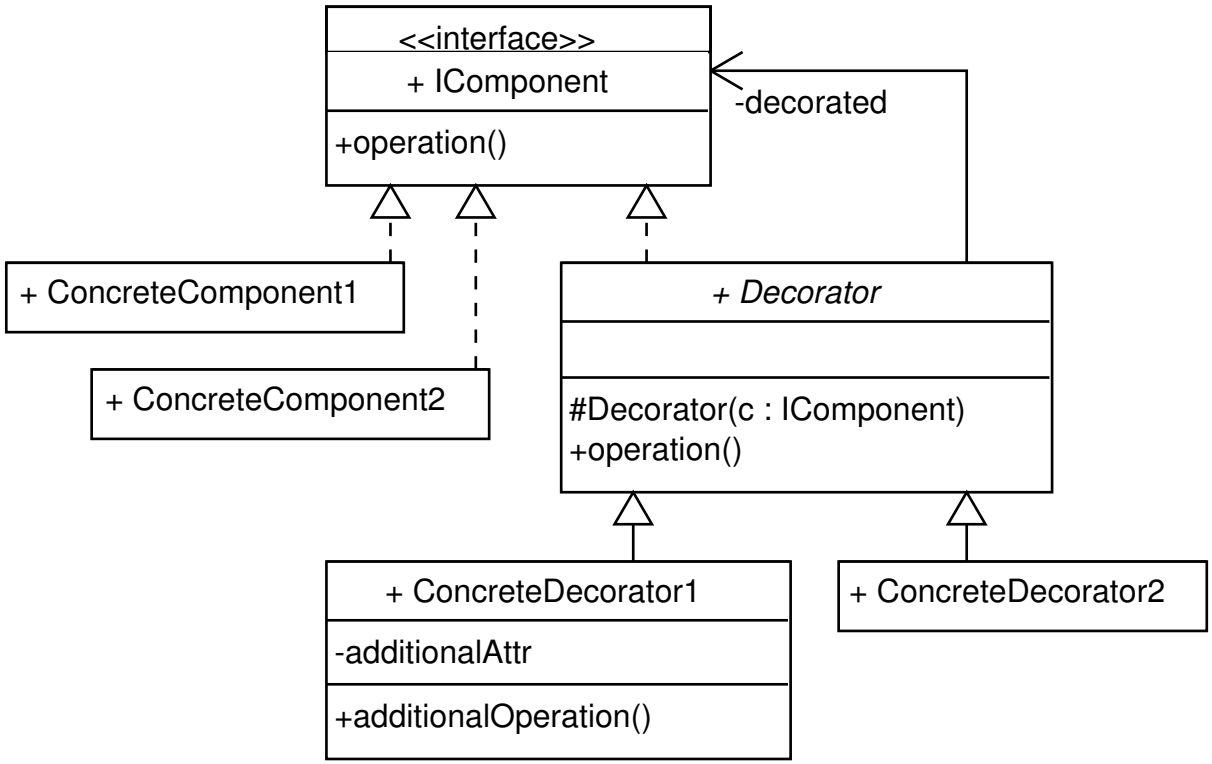
Образец Façade применяется в случае, если:

- желательно предоставить простой интерфейс к сложной системе, не раскрывающий всех её возможностей, но устраивающий большинство клиентов;

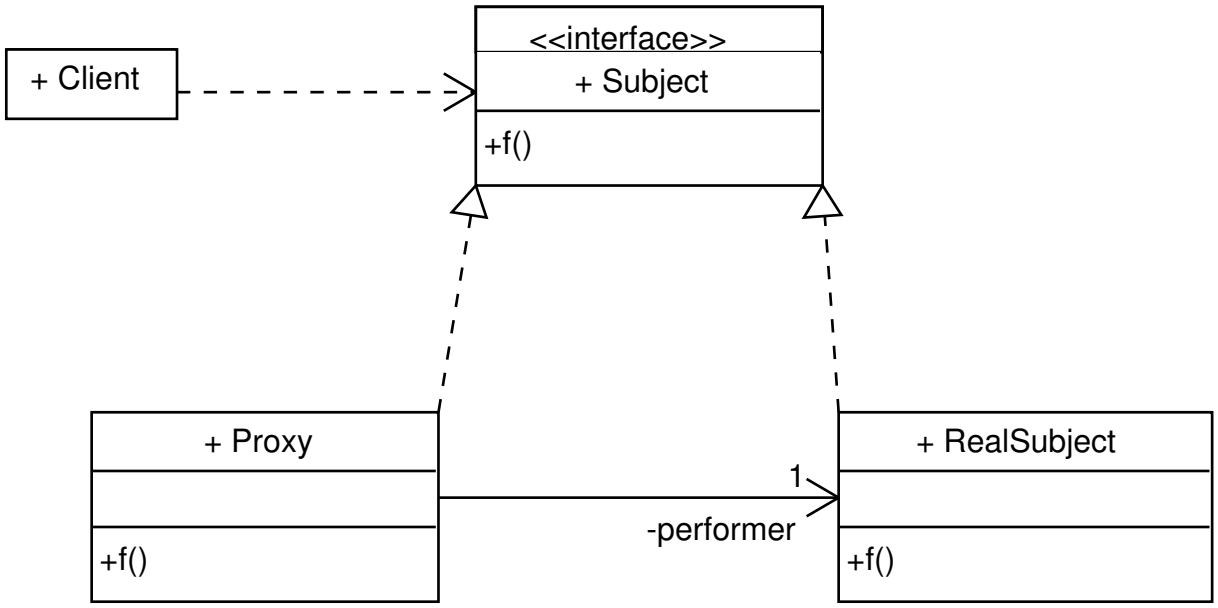
- необходимо изолировать подсистему в рамках большой системы для уменьшения числа зависимостей в системе.

§14. Образец Decorator

Образец. Decorator позволяет расширять функциональность некоторого объекта во время выполнения программы, не затрагивая другие экземпляры того же класса.



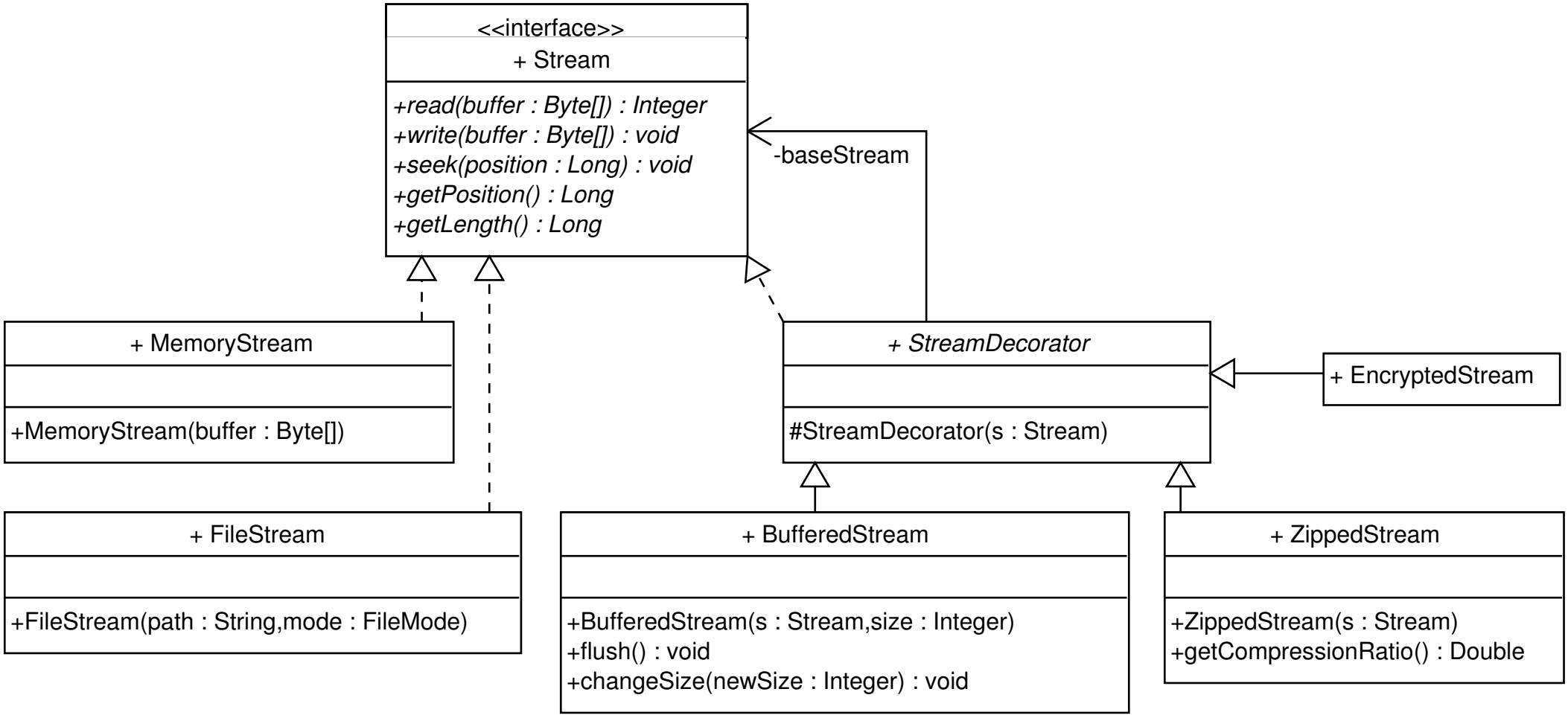
Отличия от образца Proxy:
– конкретные декораторы добавляют к интерфейсу IComponent новые методы, в то время как для класса Proxy это необязательно и нетипично;
– декорируемый объект внедряется в объект-декоратор через конструктор во время выполнения (Constructor Injection), тогда как создание объекта, доступ к которому контролирует прокси-объект, прописано внутри класса Proxy (известно во время компиляции).



Образец Decorator применяется в случае, если:
– расширение классов с помощью наследования непрактично (при большом количестве независимых расширений порождение отдельного подкласса на каждую комбинацию расширений приводит к взрывному увеличению количества классов);
– возможен отзыв расширений.

Недостаток образца Decorator:
– его использование может наводнить память огромным количеством мелких объектов, что отрицательно сказывается на производительности сборщика мусора.

Пример. Потоки ввода/вывода

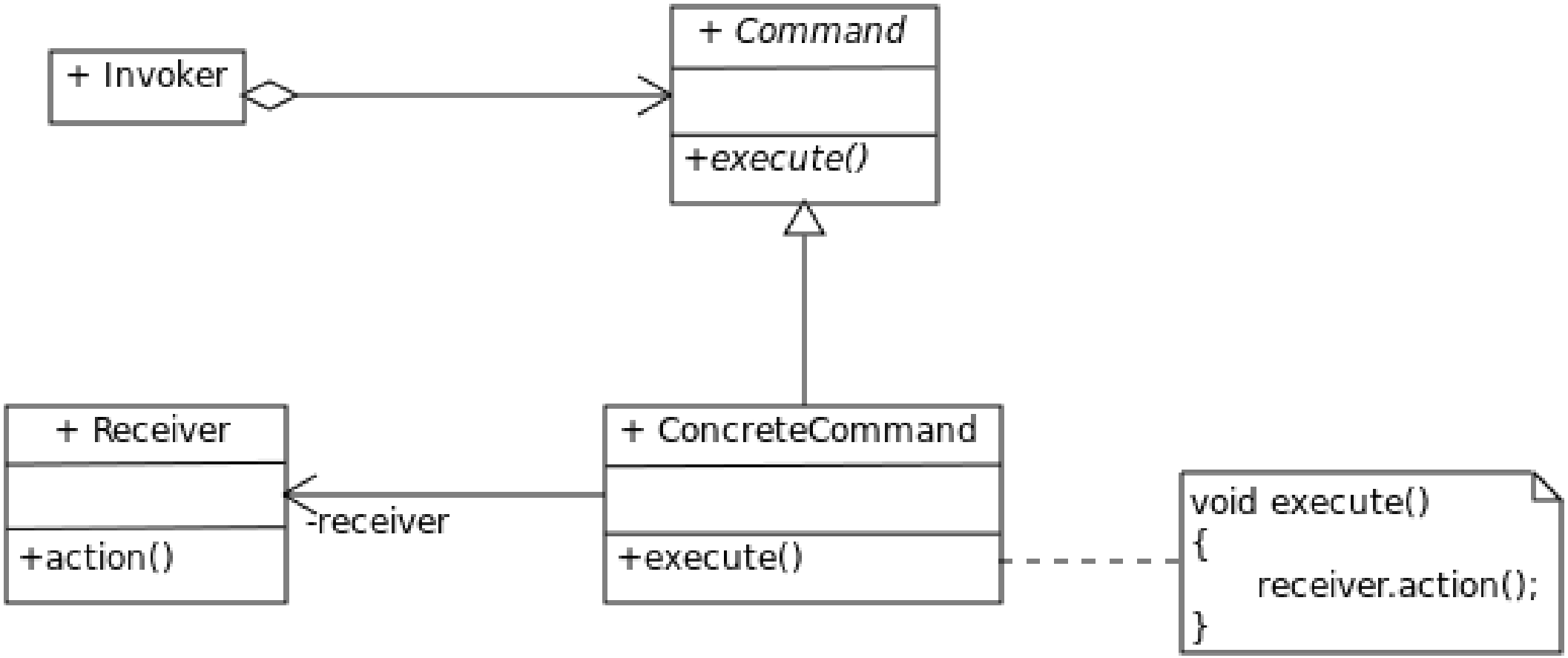


§15. Понятие поведенческого образца. Образец Command

Определение. *Поведенческий образец* – это образец проектирования, показывающий, как распределяются обязанности между объектами и осуществляется их взаимодействие.

Мы рассмотрим пять поведенческих образцов:
– Command;
– Iterator;
– Visitor;
– Chain of Responsibility;
– Observer.

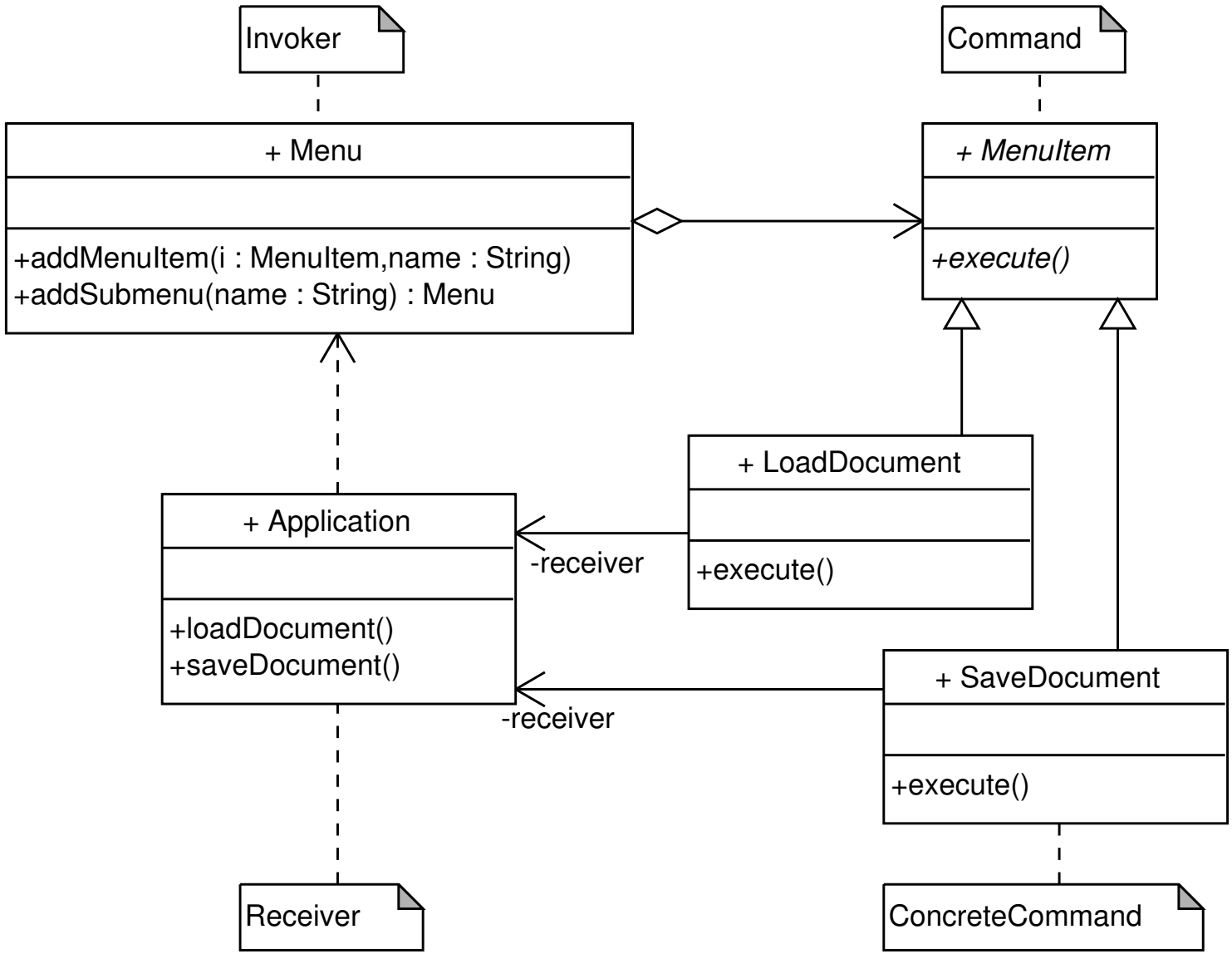
Образец. Command (Action, Transaction) инкапсулирует команды (или запросы) в объектах, что позволяет полиморфно обрабатывать эти команды, а именно: выполнять команды, организовывать очереди команд, обеспечивать отмену выполненных команд.



Образец Command применяется в случае, если:

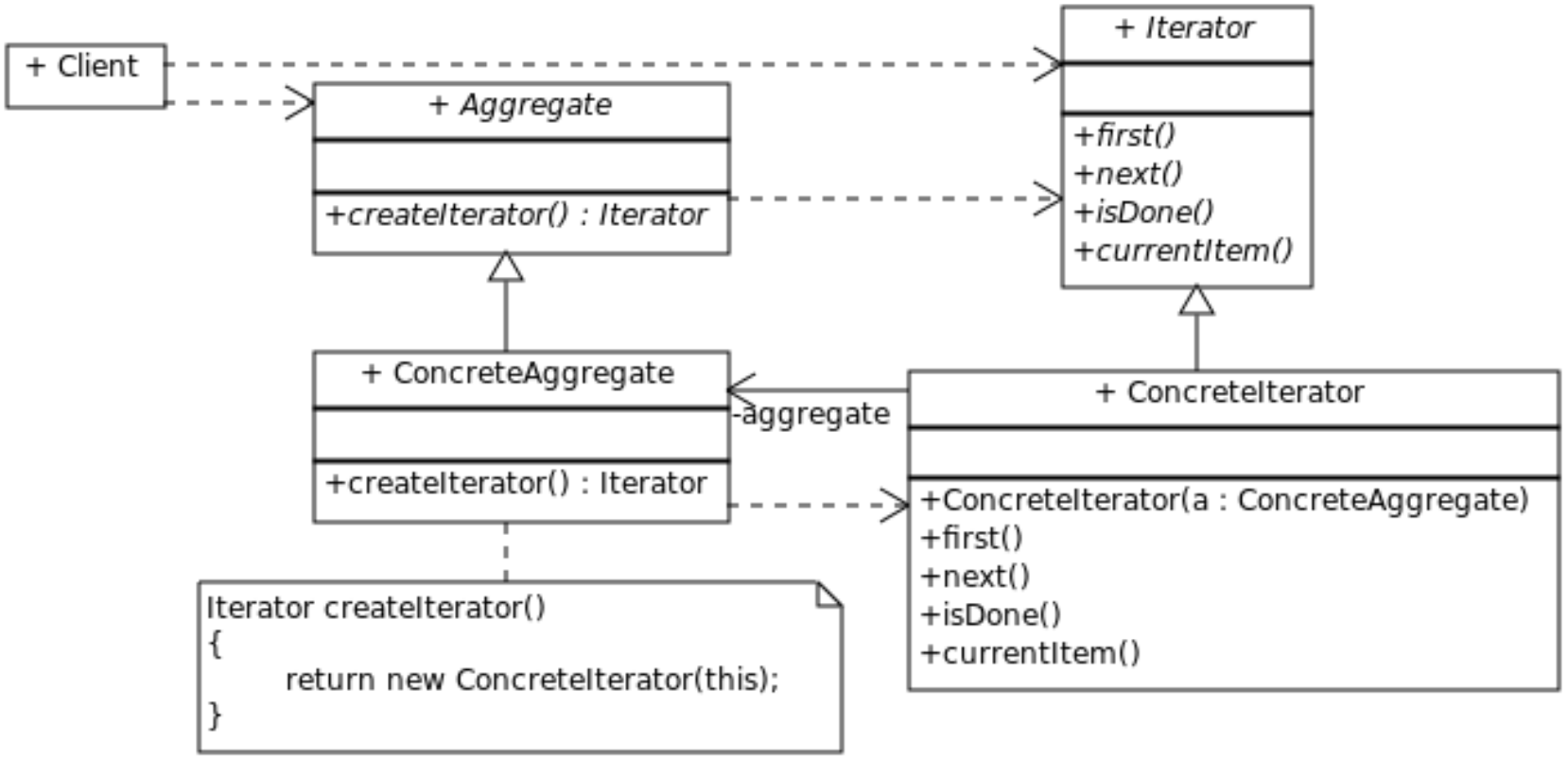
- нужно параметризовать некоторые объекты действиями (в объекты внедряются команды, и выполнение действий делегируется этим командам);
- определение и выполнение запросов разнесены во времени и/или пространстве;
- может потребоваться отмена команд (в этом случае команды должны также предоставлять метод Unexecute);
- требуется вести лог изменений с тем, чтобы можно было перевыполнить команды в случае сбоя.

Пример. Графический пользовательский интерфейс



§16. Образец Iterator

Образец. Iterator (Cursor) предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.



Образец Iterator применяется в случае, если:

- нужно скрыть внутреннее представление составного объекта;
- нужно обеспечить одновременное выполнение нескольких обходов составного объекта;
- требуется предоставить единообразный интерфейс для обхода различных составных объектов (то есть для поддержки полиморфной итерации).

Пример. (Java – кольцевой буфер)

```
1  import java.util.Iterator;

3  class Queue implements Iterable {
4      private Object[] buf;
5      private int head, tail, count;

7      public Queue(int size) { buf = new Object[size]; }

9      public void enqueue(Object x) {
10         buf[tail++] = x;
11         if (tail == buf.length) tail = 0;
12         count++;
13     }

15     public Object dequeue() {
16         Object x = buf[head++];
17         if (head == buf.length) head = 0;
18         count--;
19         return x;
20     }
21     ...
```

Пример. (продолжение)

```
21      ...
22      public Iterator iterator() { return new QueueIterator(); }

24      private class QueueIterator implements Iterator {
25          private int cur, left;

27          public QueueIterator() { cur = head;  left = count; }

29          public boolean hasNext() { return left != 0; }

31          public Object next() {
32              Object x = buf[cur++];
33              if (cur == buf.length) cur = 0;
34              left--;
35              return x;
36          }

38          public void remove() {
39              throw new UnsupportedOperationException();
40          }
41      }
42  }
```

Пример. (продолжение)

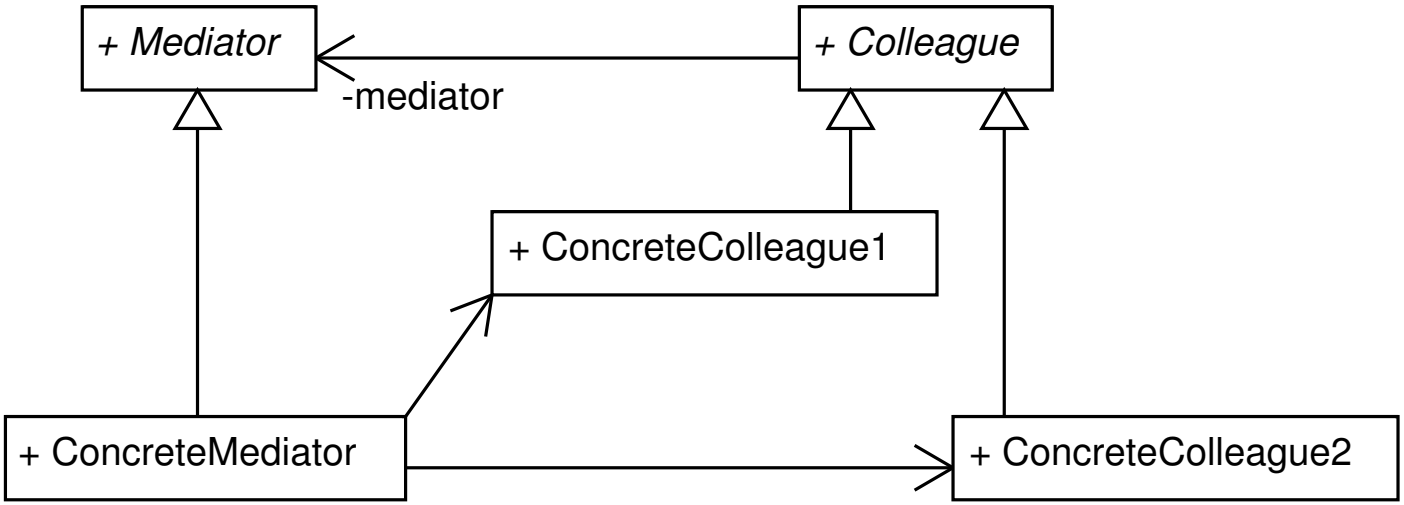
```
44 public class test {
45     public static void main(String[] args) {
46         Queue q = new Queue(10);
47         q.enqueue(45);
48         q.enqueue(20);
49         q.enqueue(30);

51         for (Iterator i = q.iterator(); i.hasNext(); ) {
52             Object x = i.next();
53             for (Iterator j = q.iterator(); j.hasNext(); ) {
54                 Object y = j.next();
55                 System.out.println("("+x+", "+y+")");
56             }
57         }
58     }
59 }
```

Пример. (продолжение – альтернативный вариант)

```
44 public class test {
45     public static void main(String[] args) {
46         Queue q = new Queue(10);
47         q.enqueue(45);
48         q.enqueue(20);
49         q.enqueue(30);
50         for (Object x : q) {
51             for (Object y : q) {
52                 System.out.println("("+x+", "+y+")");
53             }
54         }
55     }
56 }
```

§17. Образец Mediator

Образец. Mediator инкапсулирует информацию о том, как осуществляется взаимодействие в группе объектов.



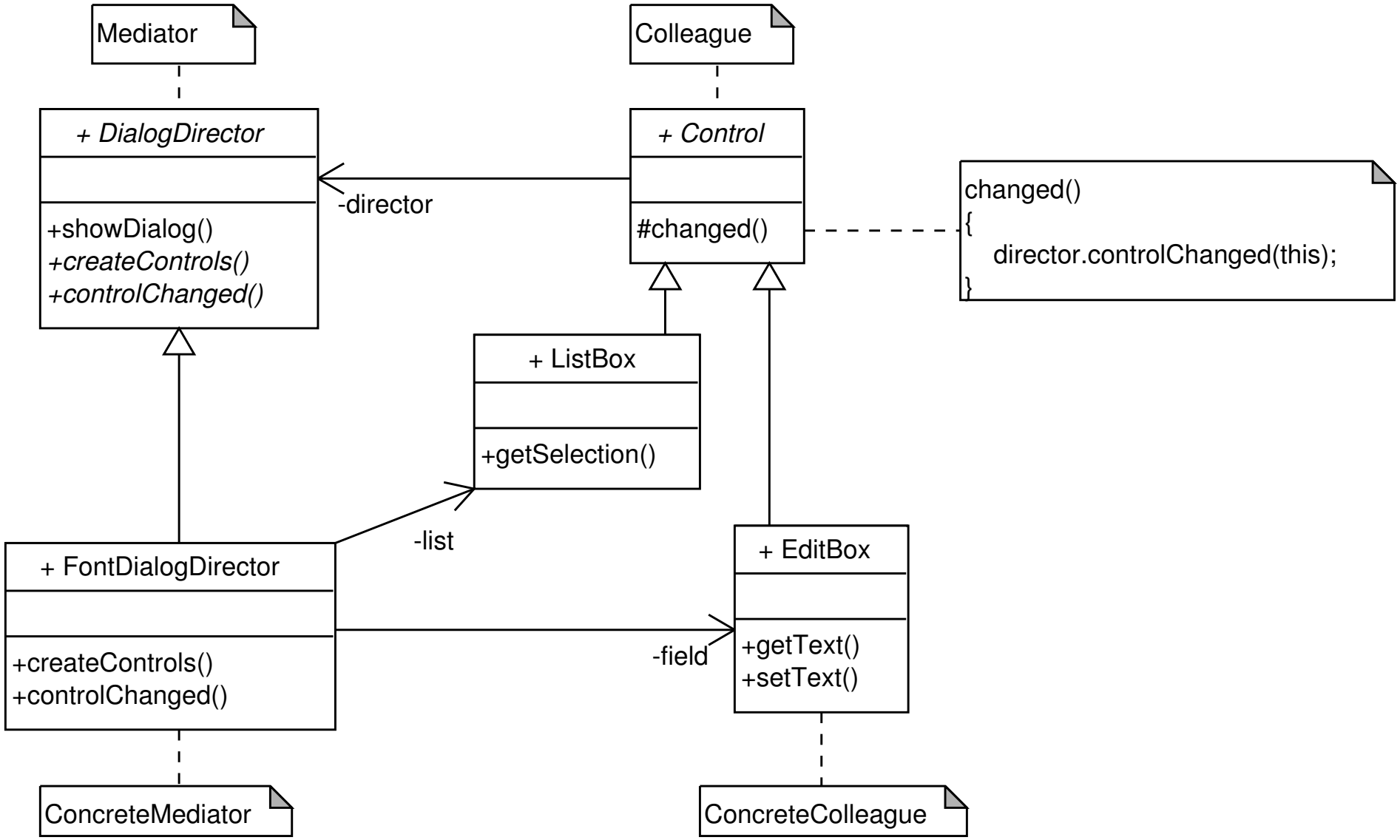
Образец Mediator применяется в случае, если:

- для группы объектов определён сложный сценарий взаимодействия, в результате чего взаимозависимости между объектами оказываются плохо структурированы и трудны для понимания;

- повторное использование класса затруднено, потому что объект класса ссылается и взаимодействует с большим количеством других объектов;

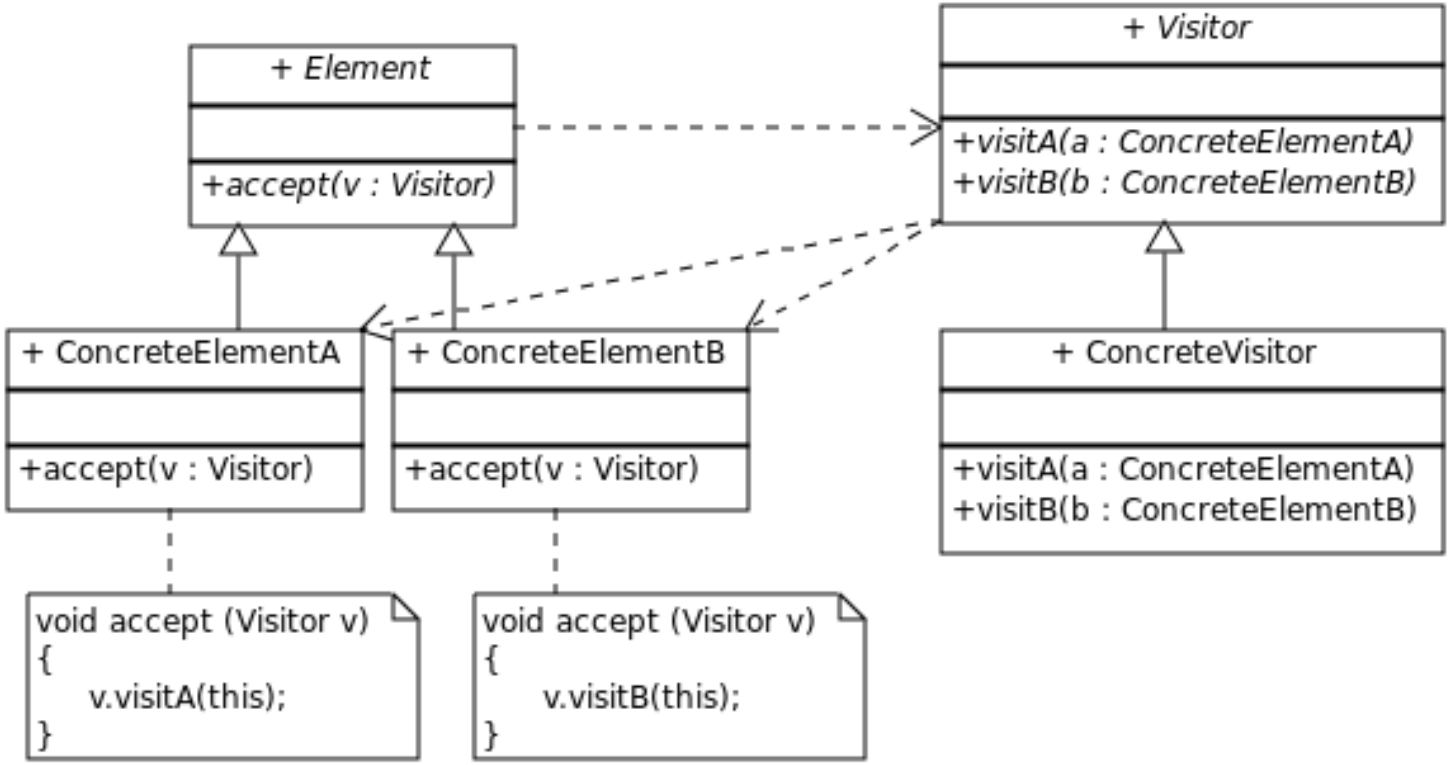
- нужно определять различные сценарии взаимодействия объектов разных классов, и при этом желательно обойтись без широкого применения наследования.

Пример. Графический пользовательский интерфейс



§18. Образец Visitor

Образец. Visitor инкапсулирует поведение, которое в противном случае пришлось бы распределять между классами.



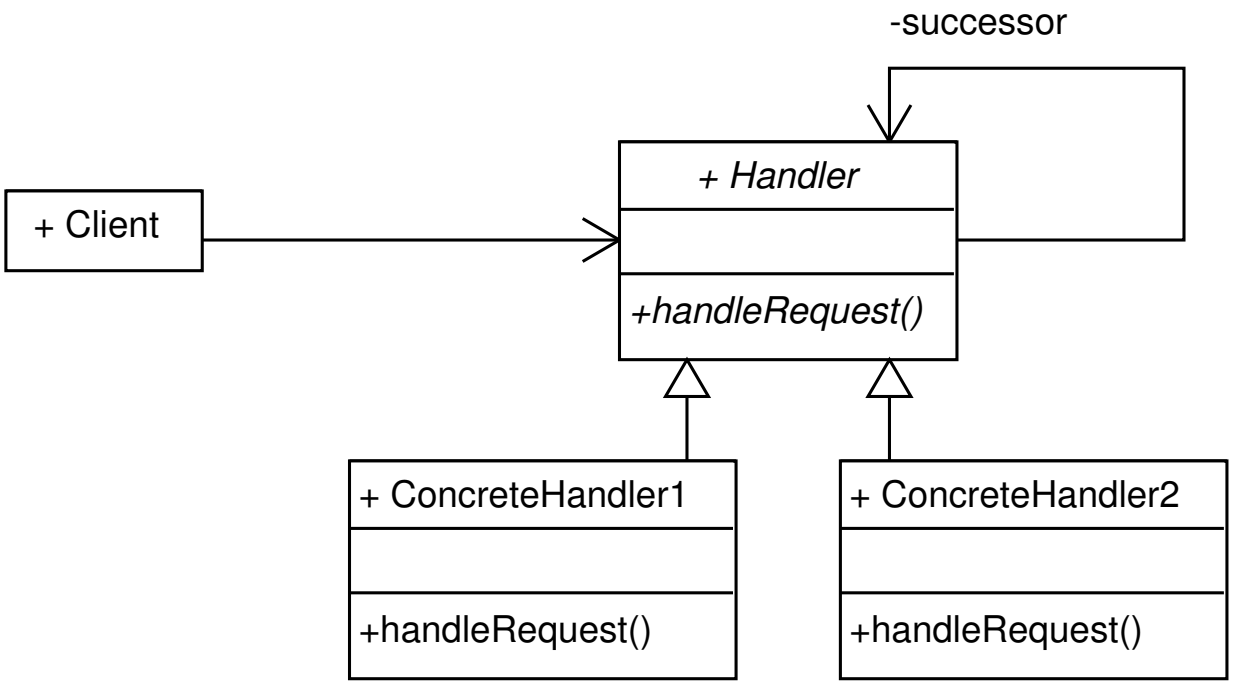
Образец Visitor применяется в случае, если:

- необходимо выполнить над объектами разных классов операцию, зависящую от класса обрабатываемого объекта (switch по типу объекта);

- над объектами надо выполнять разнообразные, не связанные между собой операции; при этом нежелательно «засорять» классы такими операциями.

§19. Образец Chain of Responsibility

Образец. Chain of Responsibility позволяет не привязывать отправителя запроса к получателю путём предоставления нескольким объектам возможности обработать запрос.

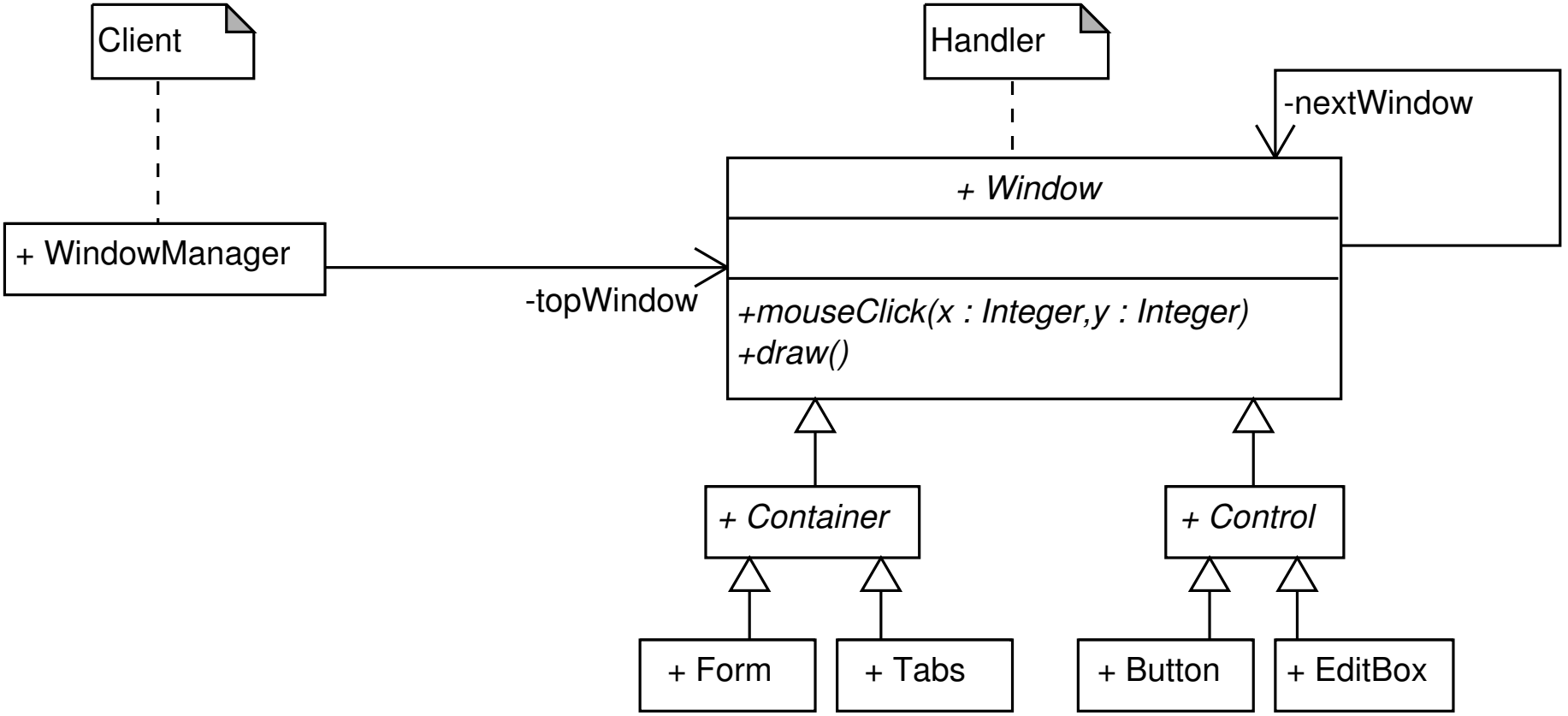


Запрос передаётся вдоль цепочки объектов-получателей до тех пор, пока один из объектов не «захочет» его выполнить.

Образец Chain of Responsibility применяется в случае, если:

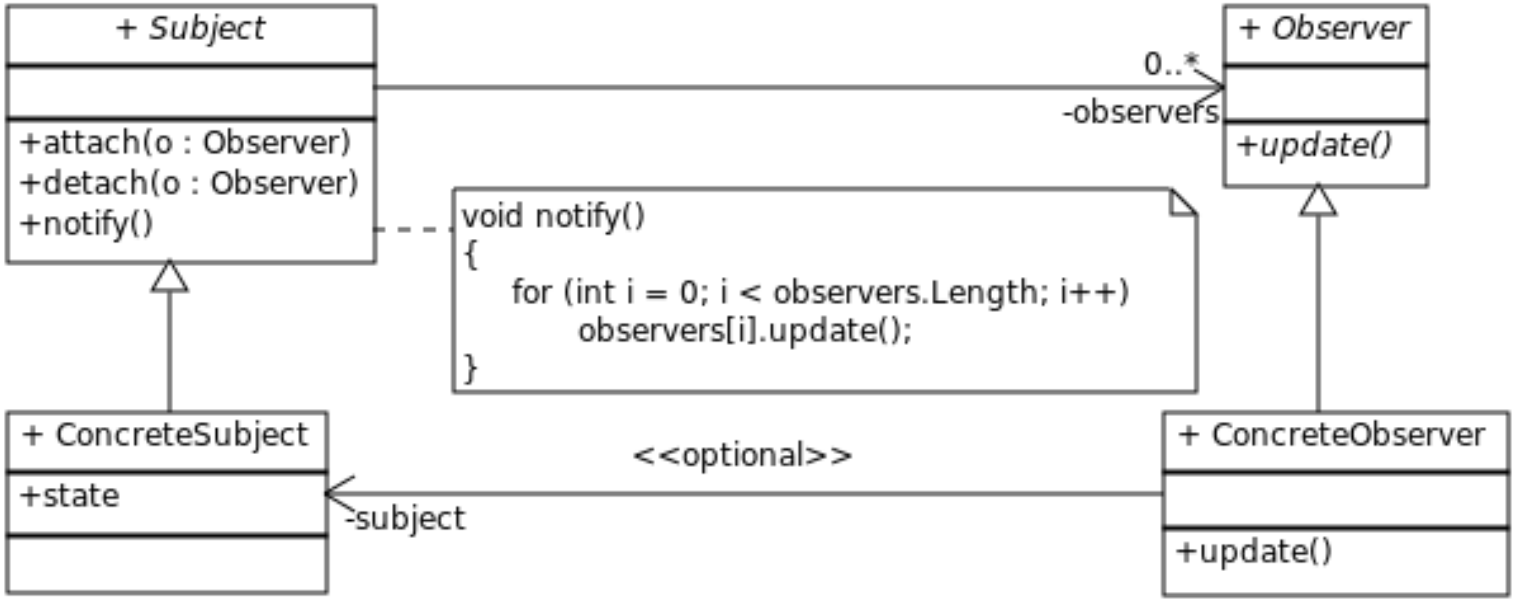
- более одного объекта могут обработать запрос, причём обработчик заранее неизвестен;
- набор объектов, способных обработать запрос, определяется во время выполнения программы.

Пример. Перекрывающиеся окна в графическом пользовательском интерфейсе.



§20. Образец Observer

Образец. Observer (Publish–Subscribe) определяет отношение «один ко многим» между объектами таким образом, что если один объект меняет своё состояние, то все зависимые от него объекты получают уведомления.



Образец Observer применяется в случае, если:

- абстракция имеет два аспекта, причём один зависит от другого (например, хранение/отображение данных); инкапсулирование этих аспектов в отдельных объектах позволяет их независимо менять и повторно использовать;

- изменение одного объекта требует изменения других объектов, и мы заранее не знаем, сколько этих объектов;

- объект должен иметь возможность уведомлять о чём-то другие объекты, но не должен знать, что это за объекты.

Пример. Элементы языка UML в редакторе UML-диаграмм