Design Patterns

Dr. Matthias Hölzl

16. Dezember 2020

Entwurfsmuster
Teil 2

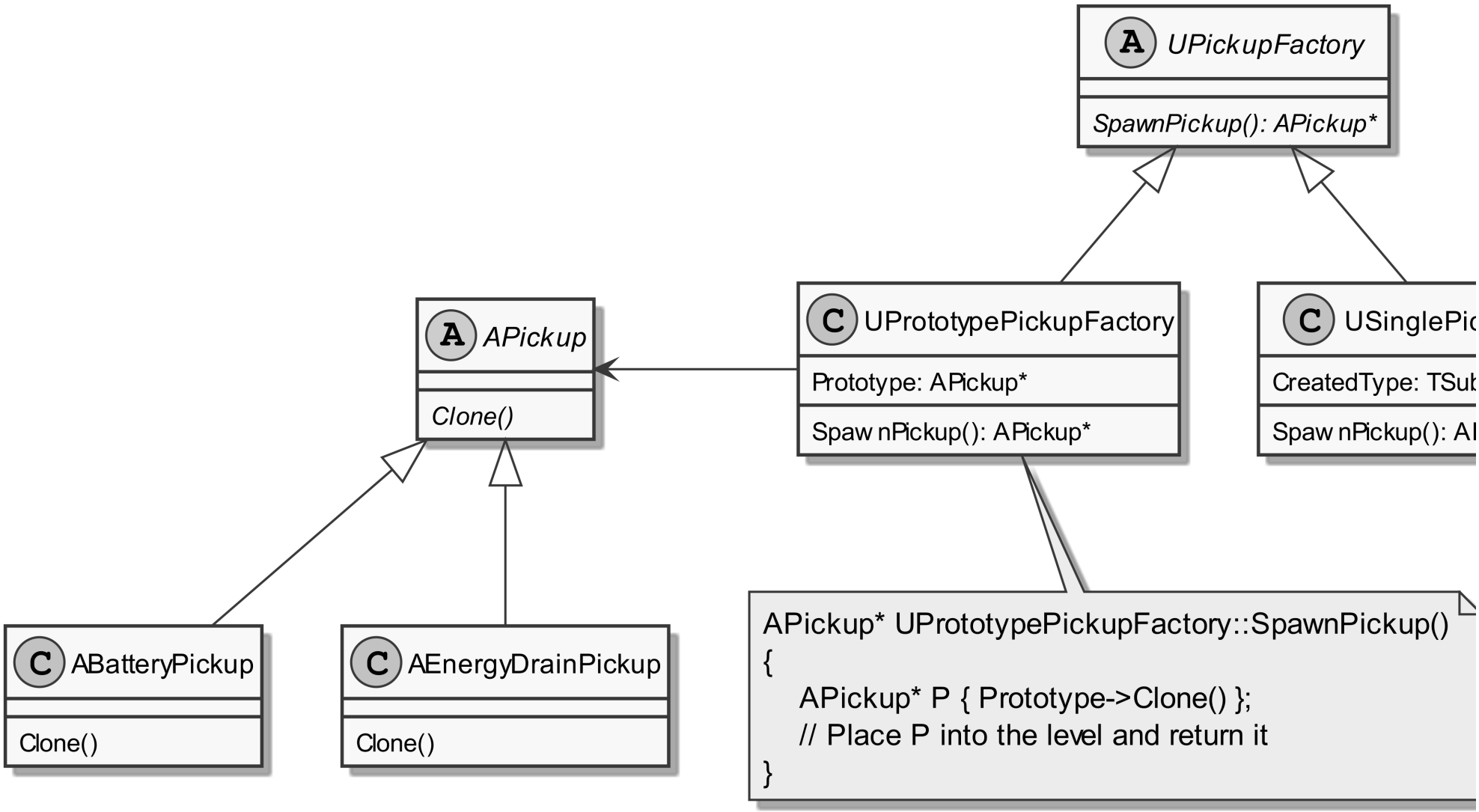
Designproblem: Konfigurierbare Bonusitems

Unser jetziges Design hat einige positive Eigenschaften:

- Der von einem Spawn-Volume erzeugte Pickup-Typ kann leicht geändert werden
- Das Hinzufügen von neuen Pickup-Typen geht ohne jede Änderung an der Spawn-Volume Implementierung
- Es können verschiedene Bonusitem-Typen pro Spawn-Volume erzeugt werden

Aber: Es ist nicht leicht möglich die in einem Spawn-Volume erzeugten Objekte zu konfigurieren.

Fünfter Entwurf: Prototyp-Basiertes Spawn-Volume



Prototype (Creational Pattern)

Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation

[Editor for music scores based on a general framework for graphical editors]

Prototype (Creational Pattern)

Applicability

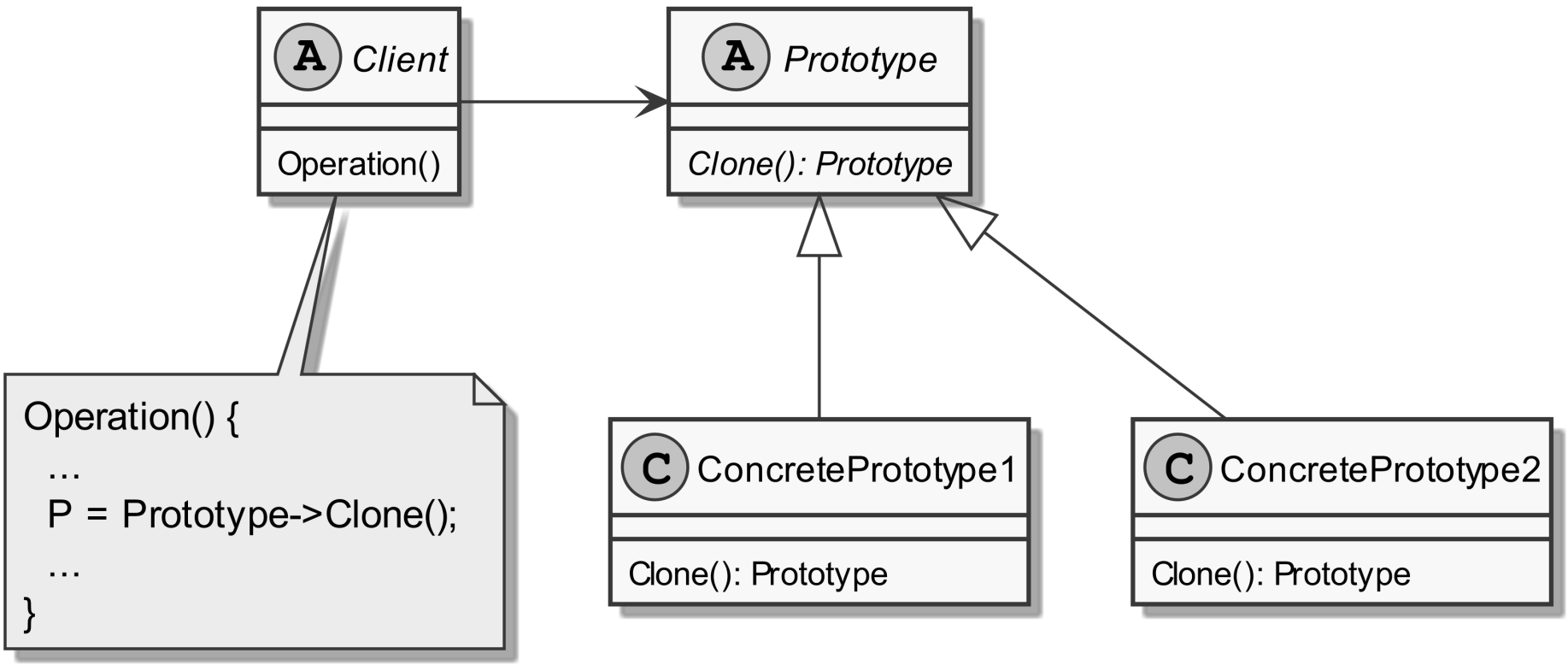
Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or
- when instances of a class can have one of only a few different combinations of state.

It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Prototype (Creational Pattern)

Structure



Prototype (Creational Pattern)

Participants

- Prototype
  declares an interface for cloning itself
- ConcretePrototype
  implements an operation for cloning itself
- Client
  creates a new object by asking a prototype to clone itself

Collaborations

A client asks a prototype to clone itself.

Prototype (Creational Pattern)

Consequences

Prototype has many of the same consequences that Abstract Factory (87) and Builder (97) have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reduced subclassing
- Configuring an application with classes dynamically

The main liability of the Prototype pattern is that each subclass of Prototype must implement the Clone operation, which may be difficult.

Prototype (Creational Pattern)

Implementation and Sample Code . . .

Related Patterns

Prototype and Abstract Factory (87) are competing patterns in some ways, as we discuss at the end of this chapter. They can also be used together, however. An Abstract Factory might store a set of prototypes from which to clone and return product objects.

Designs that make heavy use of the Composite (163) and Decorator (175) patterns often can benefit from Prototype as well.

Designproblem: Verschiedene Modi

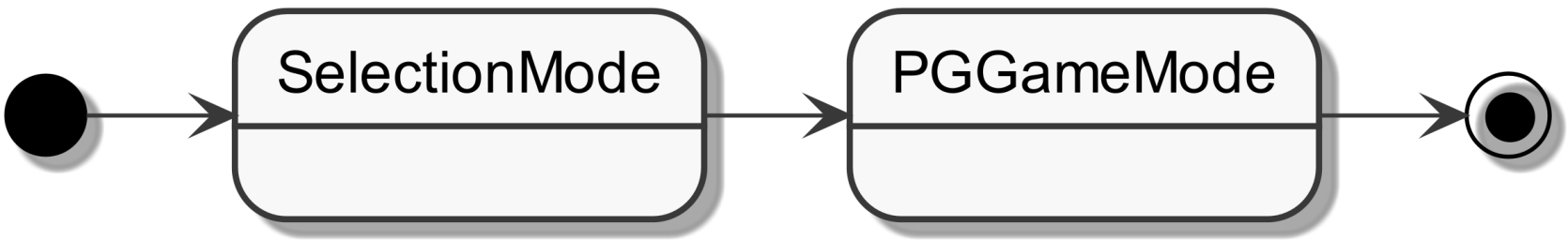
Der Designer hat sich entschieden, dass der Spieler eine Auswahl zwischen verschiedenen Figuren haben soll. Um das zu ermöglichen muss dass System zwischen zwei „Zuständen“ unterscheiden können:

- Auswahl einer Spielfigur
- Spielen des eigentlichen Levels

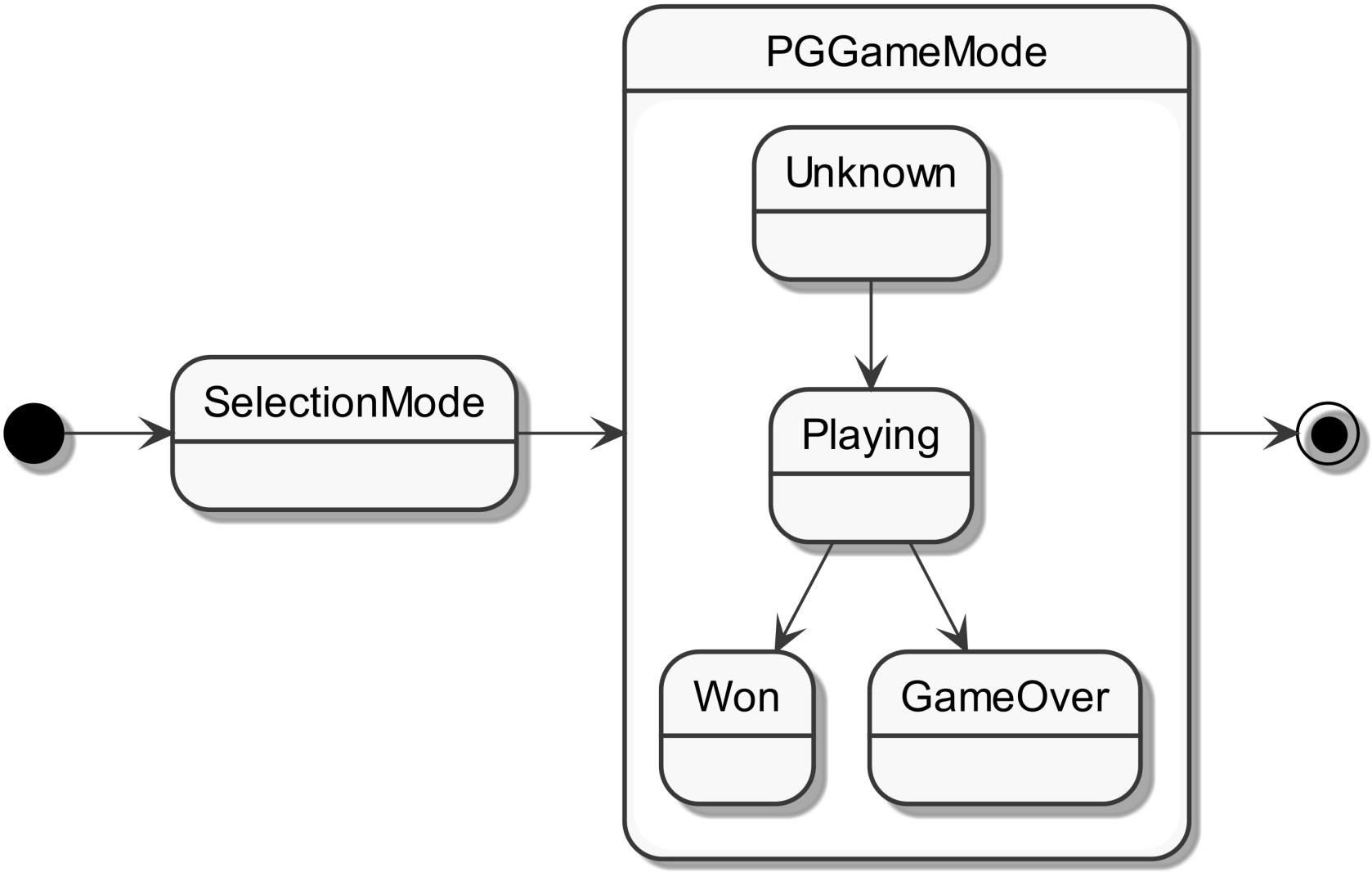
Während des Spielens treten drei (bzw. vier) Unterzustände auf:

- (Spiel ist in einem undefinierten Zustand)
- Spiel wird ausgeführt
- Spieler hat gewonnen
- Spieler hat verloren

Designproblem: Verschiedene Modi



Designproblem: Verschiedene Modi



State (Behavioral Pattern)

Intent

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

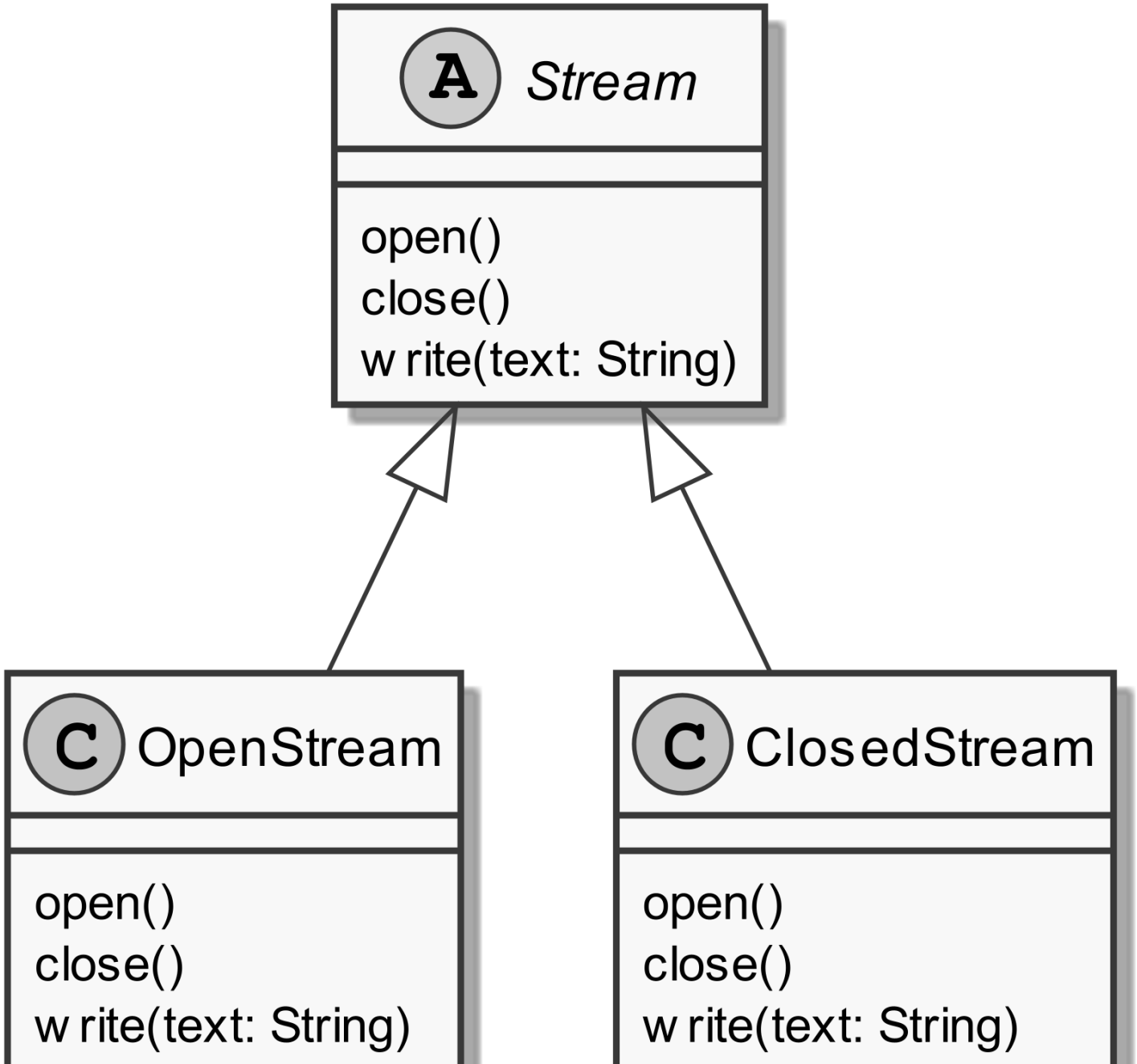
Also Known As

Objects for States

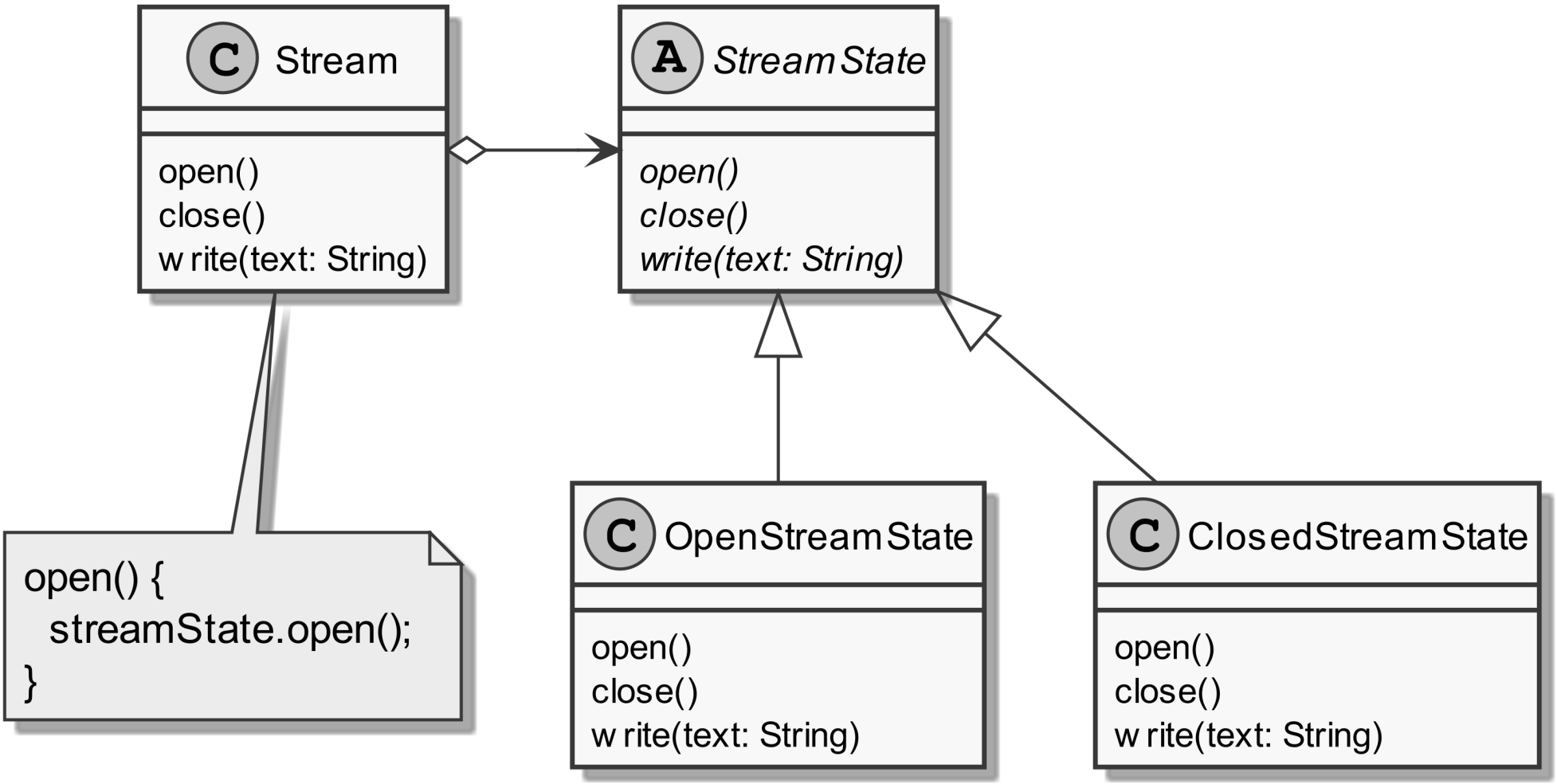
Movivation

[TCP-connection object. We use Streams as simpler example.]

State (Behavioral Pattern)



State (Behavioral Pattern)



State (Behavioral Pattern)

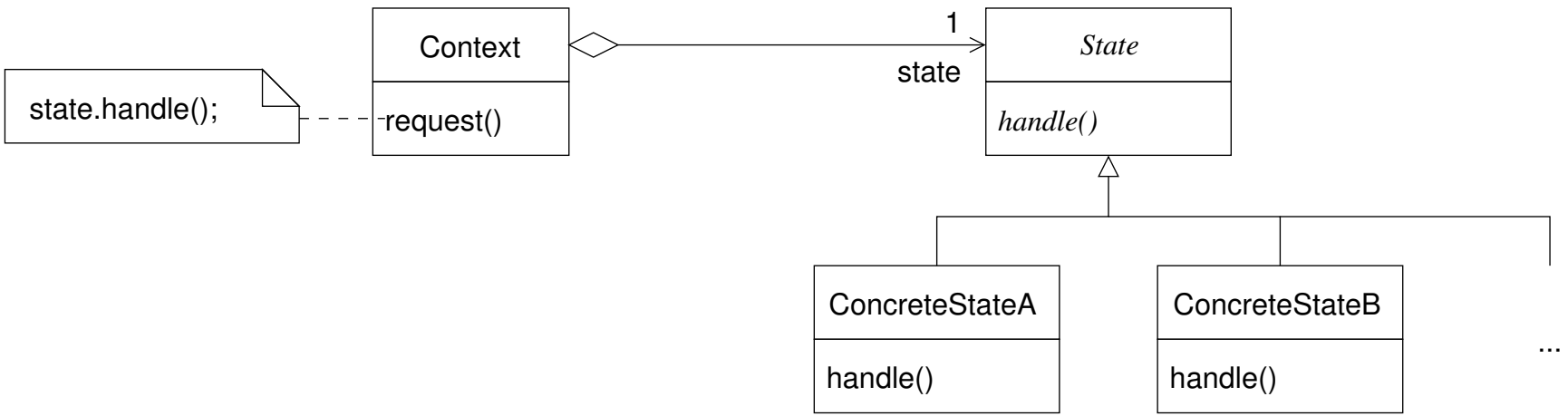
Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

State (Behavioral Pattern)

Structure



State (Behavioral Pattern)

Participants

- Context
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- State
  - defines an interface for encapsulating the behavior associated with a particular state of the Context
- ConcreteState subclasses
  - each subclass implements a behavior associated with a state of the Context

State (Behavioral Pattern)

Collaborations

- Context delegates state-specific requests to the current ConcreteState object
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

State (Behavioral Pattern)

Consequences

The State pattern has the following consequences:

- It localizes state-specific behavior and partitions behavior for different states.
- It makes state transitions explicit
- State objects can be shared

State (Behavioral Pattern)

Implementation and Sample Code ...

Known Uses

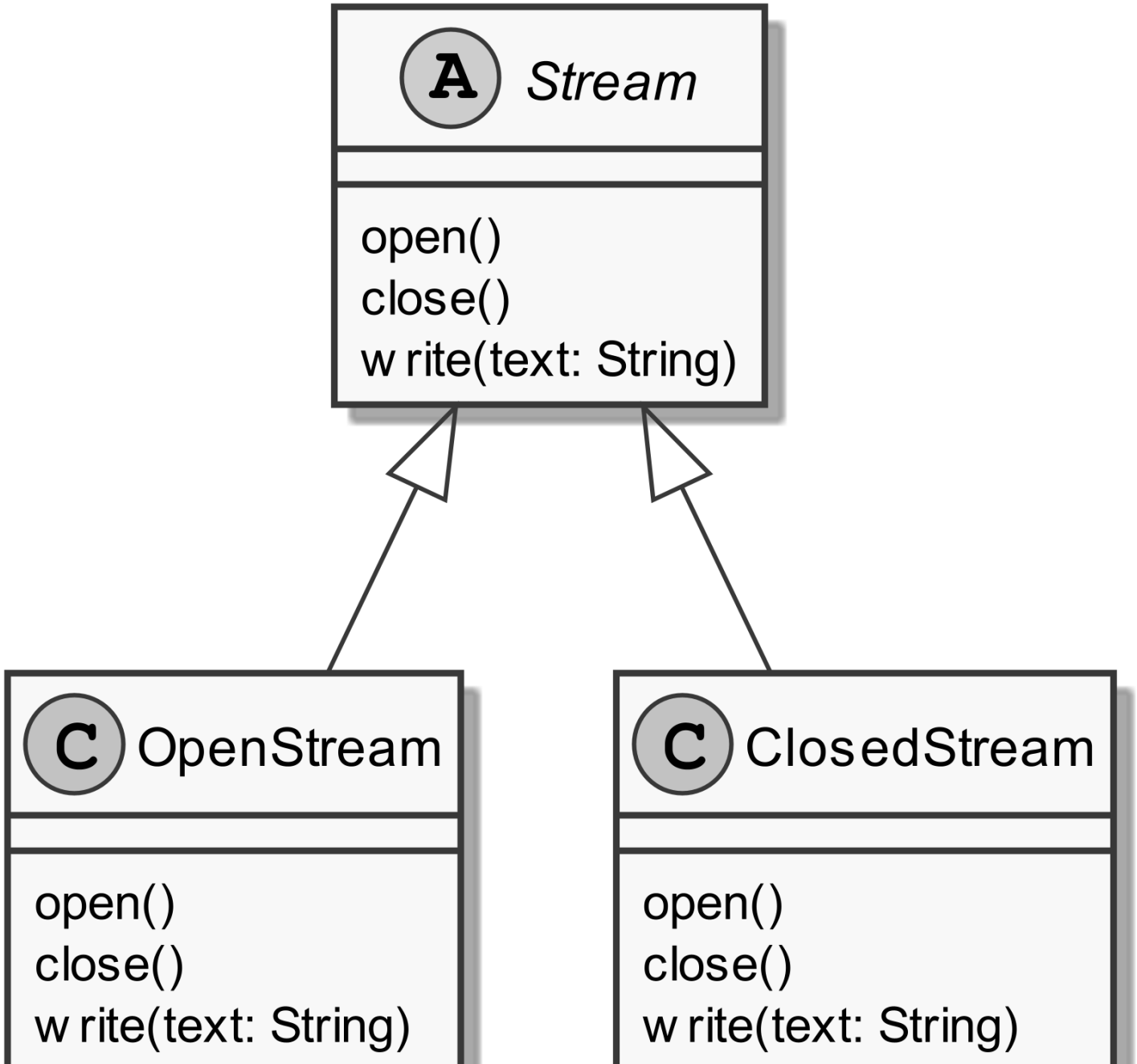
Realization of state diagrams by state objects.

Related Patterns

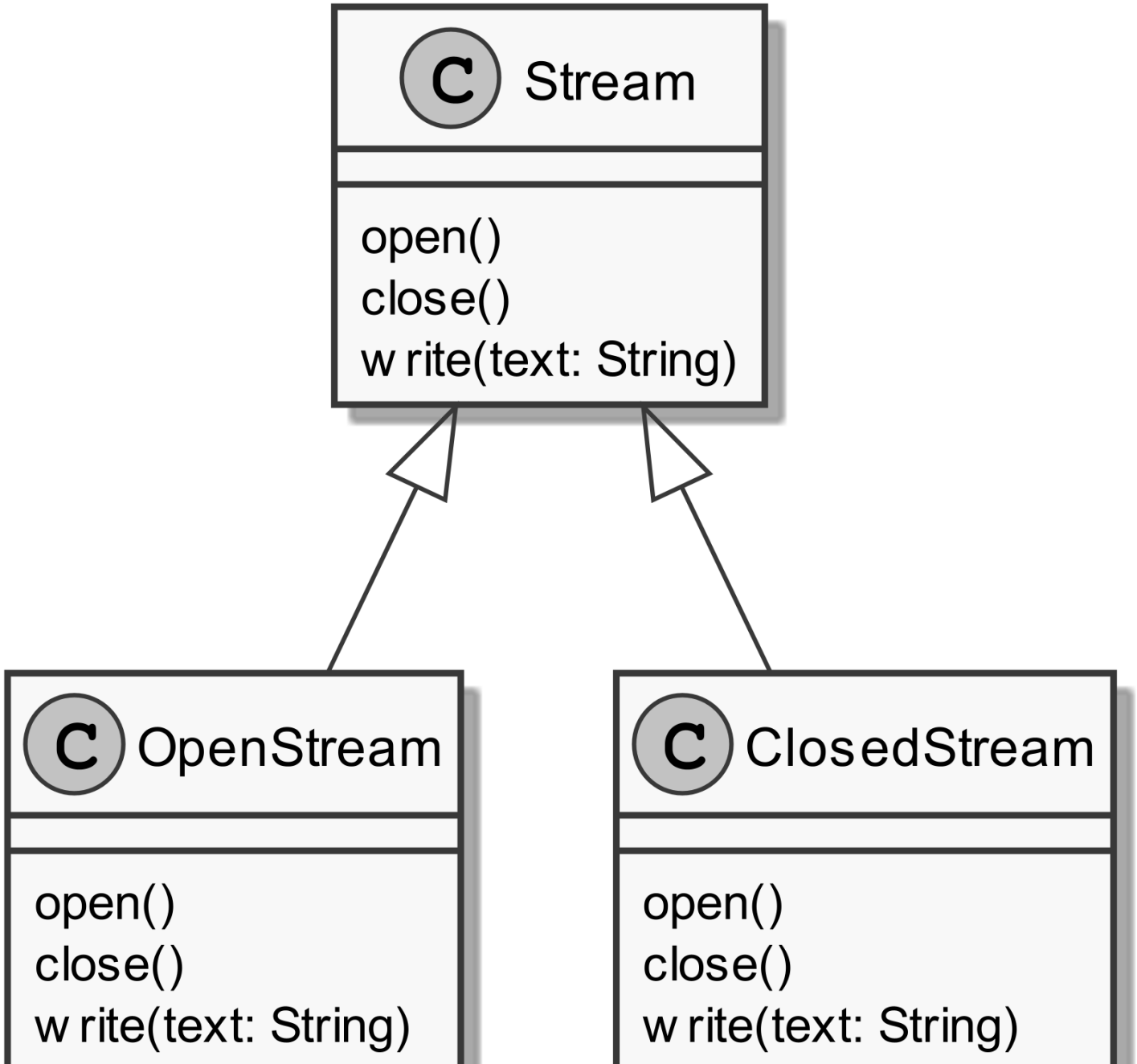
The Flyweight (195) pattern explains when and how State objects can be shared.

State objects are often Singletons (127).

State Pattern und Reflection



State Pattern und Reflection



State Pattern und Reflection

```
class MyStream
{
  MyStream(Pathname Pathname = Pathname::NoPathname,
           bool TryToOpen = false)
    : _Pathname{ Pathname }
  {
    Super();
    if (TryToOpen)
    {
      Format("Opening stream %s during construction.",
             stream);
      Open();
    }
  }
...
```

State Pattern und Reflection

```
...
  private Pathname _Pathname;

  virtual bool Open()
  {
    if (_Pathname.Exists())
    {
      ChangeClass(OpenStream::GetClass());
      return true;
    }
    else
    {
      return false;
    }
  }
...
```

State Pattern und Reflection

```
...
  virtual void Close()
  {
    ChangeClass(ClosedStream::GetClass());
  }
...
```

State Pattern und Reflection

```
...
  virtual void Write(String Text)
  {
    Format("Trying to open stream before writing.");
    if (Open())
    {
      // Note that this now a call to
      // OpenStream::Write()!
      Write(Text);
    }
    else
    {
      Warn("Cannot write to %s", this);
    }
  }
};
```

State Pattern und Reflection

```
class OpenStream : public Stream
{
  virtual bool Open() {}; // Do nothing.

  virtual void Write(String Text)
  {
    Format("Writing to stream: %s", Text);
  }
};
```

State Pattern und Reflection

```
class ClosedStream : public Stream
{
  virtual void Close() {}; // Do nothing.

  virtual void Write(String Text)
  {
    Format("Cannot write to closed stream.");
  }
};
```

State Pattern und Reflection

```
void TestStreamWriting(Stream Stream)
{
  Format("Stream: %s.", Stream);
  Stream->Write("Hello!");
  Format("Stream: %s.", Stream);
  Stream->Close();
  Format("Stream: %s.", Stream);
  Stream->Write("Goodbye!");
}
```

State Pattern und Reflection

```
int main()
{
  TestStreamWriting(new Stream());
  TestStreamWriting(new Stream("a-file-that-exists"));
  TestStreamWriting(new Stream("a-file-that-exists",
                               true));
...
```

State Pattern und Reflection

```
Stream: #<MY-STREAM {1002CDAC93}>.
Trying to open stream before writing.
WARNING: Cannot write to #<MY-STREAM {1002CDAC93}>.
Stream: #<MY-STREAM {1002CDAC93}>.
Stream: #<CLOSED-STREAM {1002CDAC93}>.
Cannot write to closed stream.
```

State Pattern und Reflection

```
Stream: #<MY-STREAM {1002E758A3}>.
Trying to open stream before writing.
Writing to stream: Hello!
Stream: #<OPEN-STREAM {1002E758A3}>.
Stream: #<CLOSED-STREAM {1002E758A3}>.
Cannot write to closed stream.
```

State Pattern und Reflection

```
Opening stream #<MY-STREAM {1002EE3213}>
                                        during construction.
Stream: #<OPEN-STREAM {1002EE3213}>.
Writing to stream: Hello!
Stream: #<OPEN-STREAM {1002EE3213}>.
Stream: #<CLOSED-STREAM {1002EE3213}>.
Cannot write to closed stream.
```

State Pattern und Reflection

```
  ...
  Stream::MyStream(Pathname Pathname = Pathname::NoPathname
                   bool TryToOpen = true)
    : _Pathname{ Pathname }
  {
    Super();
    if (TryToOpen)
    {
      Format("Opening stream %s during construction.",
             stream);
      Open();
    }
  }
  TestStreamWriting(new Stream("a-file-that-exists",
                               true));
}
```

State Pattern und Reflection

```
Opening stream #<MY-STREAM {1002F4DB43}>
                                        during construction.
Stream: #<OPEN-STREAM {1002F4DB43}>.
Writing to stream: Hello!
Stream: #<OPEN-STREAM {1002F4DB43}>.
Stream: #<CLOSED-STREAM {1002F4DB43}>.
Cannot write to closed stream.
```

State Pattern und Reflection

```
(defclass my-stream ()
  ((stream-pathname :accessor stream-pathname
                    :initarg :pathname :initform nil)))

(defgeneric open-my-stream (stream))

(defmethod open-my-stream ((stream my-stream))
  (if (and (pathnamep (stream-pathname stream))
           (probe-file (stream-pathname stream)))
      (progn
        (change-class stream 'open-stream)
        t)
      nil))
```

State Pattern und Reflection

```
(defmethod initialize-instance :after
      ((stream my-stream) &key try-to-open)
  (when try-to-open
    (format *standard-output*
            "Opening stream ~A during construction."
            stream)
    (open-my-stream stream)))
```

State Pattern und Reflection

```
(defgeneric close-my-stream (stream))

(defmethod close-my-stream ((stream my-stream))
  (change-class stream 'closed-stream))
```

State Pattern und Reflection

```
(defgeneric write-to-my-stream (stream text))

(defmethod write-to-my-stream ((stream my-stream) text)
  (format *standard-output*
          "~&Trying to open stream before writing.")
  (if (open-my-stream stream)
      (write-to-my-stream stream text)
      (warn "Cannot write to ~A." stream)))
```

State Pattern und Reflection

```
(defclass open-stream (my-stream)
  ())

(defmethod open-my-stream ((stream open-stream))
  ;; do nothing
  nil)

(defmethod write-to-my-stream ((stream open-stream) text)
  (format *standard-output*
          "~&Writing to stream: ~A" text))
```

State Pattern und Reflection

```
(defclass closed-stream (my-stream)
  ())

(defmethod close-my-stream ((stream closed-stream))
  ;; do nothing
  nil)

(defmethod write-to-my-stream ((stream closed-stream) text)
  (format *standard-output*
          "~&Cannot write to closed stream."))
```

State Pattern und Reflection

```
(defun test-stream-writing (stream)
  (format *standard-output* "~&Stream: ~A." stream)
  (write-to-my-stream stream "Hello!")
  (format *standard-output* "~&Stream: ~A." stream)
  (close-my-stream stream)
  (format *standard-output* "~&Stream: ~A." stream)
  (write-to-my-stream stream "Goodbye!")
  (format *standard-output* "~2%"))
```

State Pattern und Reflection

```
(test-stream-writing
  (make-instance 'my-stream))
(test-stream-writing
  (make-instance 'my-stream
                 :pathname #P"change-class.lisp"))
(test-stream-writing
  (make-instance 'my-stream
                 :pathname #P"change-class.lisp"
                 :try-to-open t))
```

State Pattern und Reflection

```
(defmethod initialize-instance :after
      ((stream my-stream) &key (try-to-open t))
  (when try-to-open
    (format *standard-output*
            "Opening stream ~A during construction."
            stream)
    (open-my-stream stream)))

(test-stream-writing
  (make-instance 'my-stream
                 :pathname #P"change-class.lisp"))
```

Reflection: Visualisierung von Objekten

Problem

Wie kann ich einen generischen Mechanismus schaffen, der es Entwicklern oder Anwendern erlaubt Anwendungsobjekte zu erzeugen oder zu konfigurieren?

Mögliche Lösung

Zugriff mittels Reflection.

Reflection: Visualisierung von Objekten

```
(defgeneric print-info (object))

(defmethod print-info :before (object)
  (format t "~&Object has type ~A."
          (class-name (class-of object)))
  (format t "~&Its superclasses are ~A."
          (mapcar 'class-name
                  (class-superclasses (class-of object)))))

(defmethod print-info (object)
  (format t "~&Its value is ~A.~2%" object))
```

Reflection: Visualisierung von Objekten

```
(defmethod print-info ((object standard-object))
  (dolist (slot (mop::class-slots (class-of object)))
    (let ((name (mop::slot-definition-name slot)))
      (format t "~&Slot ~A has value ~A."
              name
              (slot-value object name))))
  (format t "~2%"))
```

Reflection: Visualisierung von Objekten

```
(defclass my-class ()
  ((slot-a :initform 10 :initarg :a)
   (slot-b :initform 20 :initarg :b)))

(defclass your-class (my-class)
  ((slot-c :initform 30 :initarg :c)))

(defclass another-class ()
  ((just-one-slot :initform "Foo")))

(defclass a-fancy-class (your-class another-class)
  ((and-yet-another-slot :initform "Bar")))
```

Reflection: Visualisierung von Objekten

```
(print-info (make-instance 'my-class))
(print-info (make-instance 'my-class :a 1 :b 99))
(print-info (make-instance 'your-class))
(print-info (make-instance 'another-class))
(print-info (make-instance 'a-fancy-class))
(print-info 123)
(print-info "Foo")
```

Reflection: Visualisierung von Objekten

```
Object has type MY-CLASS.
Its superclasses are (MY-CLASS STANDARD-OBJECT).
Slot SLOT-A has value 10.
Slot SLOT-B has value 20.

Object has type MY-CLASS.
Its superclasses are (MY-CLASS STANDARD-OBJECT).
Slot SLOT-A has value 1.
Slot SLOT-B has value 99.

Object has type YOUR-CLASS.
Its superclasses are (YOUR-CLASS MY-CLASS STANDARD-OBJECT)
Slot SLOT-A has value 10.
Slot SLOT-B has value 20.
Slot SLOT-C has value 30.
```

Reflection: Visualisierung von Objekten

```
Object has type ANOTHER-CLASS.
Its superclasses are (ANOTHER-CLASS STANDARD-OBJECT).
Slot JUST-ONE-SLOT has value Foo.

Object has type A-FANCY-CLASS.
Its superclasses are (A-FANCY-CLASS YOUR-CLASS
                      ANOTHER-CLASS MY-CLASS
                      STANDARD-OBJECT).
Slot JUST-ONE-SLOT has value Foo.
Slot SLOT-A has value 10.
Slot SLOT-B has value 20.
Slot SLOT-C has value 30.
Slot AND-YET-ANOTHER-SLOT has value Bar.
```

Reflection: Visualisierung von Objekten

```
Object has type FIXNUM.
Its superclasses are (FIXNUM INTEGER RATIONAL
                      REAL NUMBER T).
Its value is 123.

Object has type SIMPLE-CHARACTER-STRING.
Its superclasses are (SIMPLE-CHARACTER-STRING
                      CHARACTER-STRING SIMPLE-STRING
                      STRING VECTOR SEQUENCE
                      SIMPLE-ARRAY ARRAY T).
Its value is Foo.
```

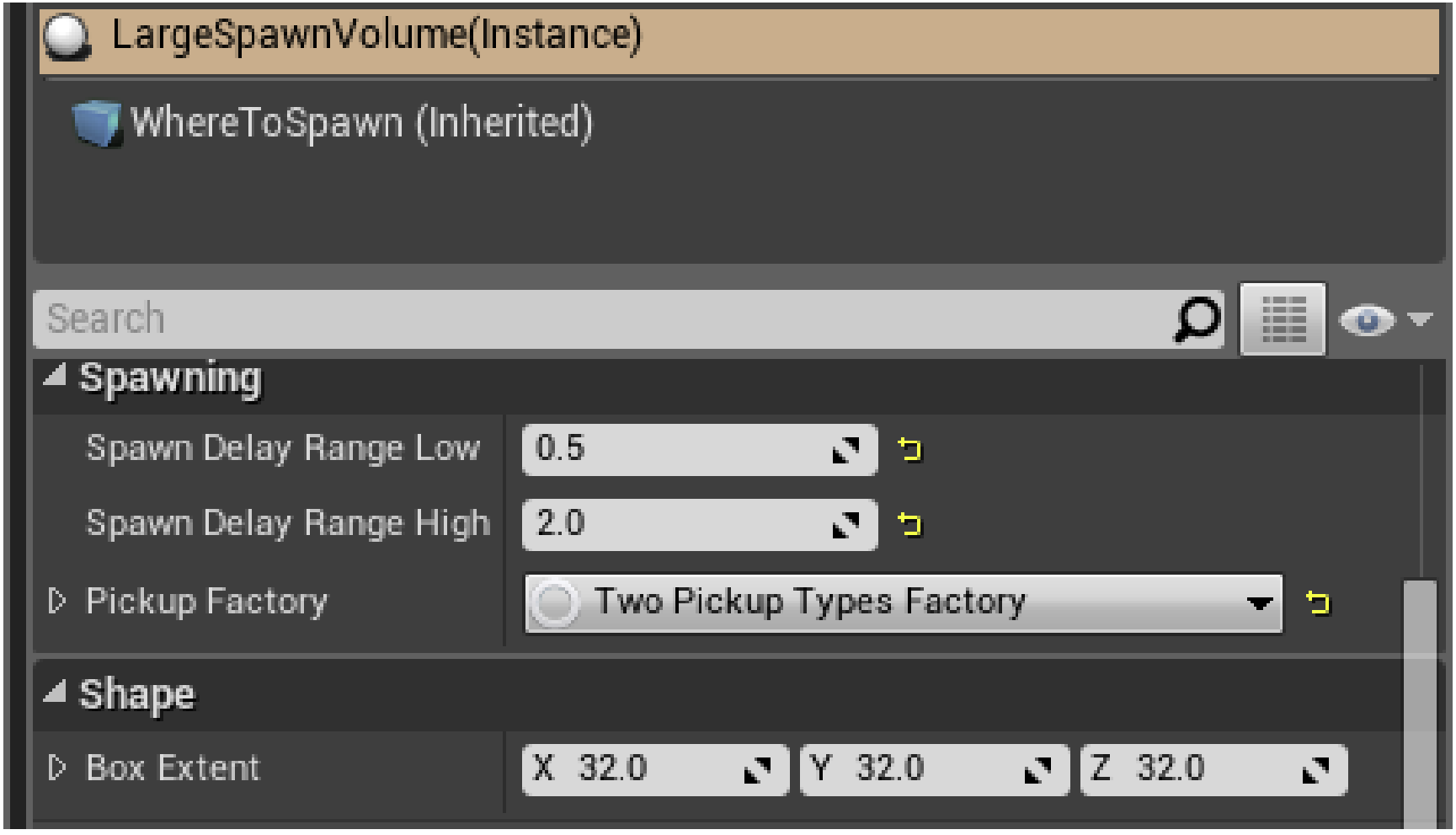
Reflection: Visualisierung von Objekten

```
UPROPERTY(EditAnywhere, BlueprintReadWrite,
          Category = "Spawning")
float SpawnDelayRangeLow;

UPROPERTY(EditAnywhere, BlueprintReadWrite,
          Category = "Spawning")
float SpawnDelayRangeHigh;

UPROPERTY(EditAnywhere, BlueprintReadWrite,
          Instanced, Category = "Spawning")
UPickupFactory* PickupFactory;
```

Reflection: Visualisierung von Objekten

```
UCLASS(Abstract, Blueprintable, EditInlineNew)
class PG_API UPickupFactory : public UObject
{ ... }

UCLASS(AutoExpandCategories = "Spawning")
class PG_API UTwoPickupTypesFactory
  : public UPickupFactory
{ ... }
```

Reflection: Visualisierung von Objekten



Designproblem: Verschiedene Spielerfiguren

- Wir haben jetzt die Möglichkeit geschaffen, dass Spieler verschiedene Spielerfiguren auswählen können
- Diese Figuren sollen auch unterschiedlich auf die einzelnen Bonusitems reagieren
- Im Moment wird der Effekt, den ein Bonusitem auf die Spielerfigur hat vom Bonusitem selber bestimmt. Das ist nicht flexibel genug
- Zusätzlich soll es möglich sein, während des Spiels die Reaktion auf Bonusitems zu verändern (Power-Ups, Malus-Items)

Designproblem: Verschiedene Spielerfiguren

Mögliche Lösung

- Lagere die Reaktion auf Pickups in eine Klassenhierarchie aus, die *Handler* definiert, die nur für die Interaktion zwischen Figuren und Bonusitems zuständig sind
- Jeder Handler implementiert nur einen Teil aller möglichen Kombinationen
- Wenn ein Handler nicht auf eine Pickup/Character-Kombination reagieren kann, reicht sie die Arbeit an eine andere Klasse weiter

Chain of Responsibility (Behavioral Pattern)

Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation

Consider a context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. [. . .]

Chain of Responsibility (Behavioral Pattern)

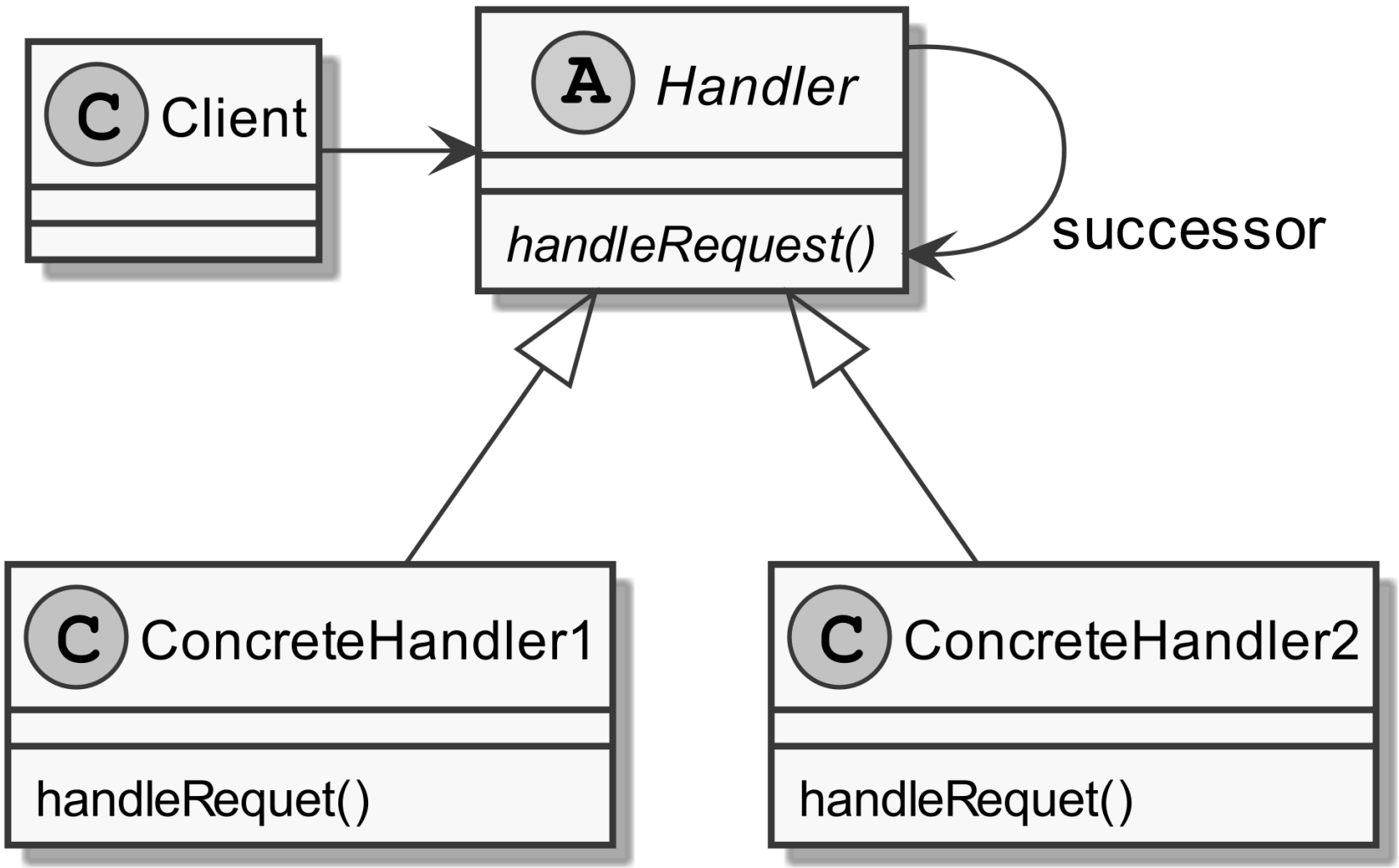
Applicability

Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically
- You want to issue a request to one of several objects without specifying the receiver explicitly
- The set of objects that can handle a request should be specified dynamically

Chain of Responsibility (Behavioral Pattern)

Structure



Chain of Responsibility (Behavioral Pattern)

Participants

- Handler
  - defines an interface for handling requests
  - (optional) implements the successor link
- ConcreteHandler
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- Client
  - initiates the request to a ConcreteHandler object on the chain.

Collaborations

When a client issues a request, the request propagates along the chain until a ConcreteHandler object takes responsibility for handling it.

Chain of Responsibility (Behavioral Pattern)

Consequences

Chain of Responsibility has the following benefits and liabilities:

- Reduced coupling. The pattern frees an object from knowing which other object handles a request. [. . .] As a result, Chain of Responsibility can simplify object interconnections. [. . .]
- Added flexibility in assigning responsibilities to objects. [. . .] You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialize handlers statically.
- Receipt isn't guaranteed. Since a request has no explicit receiver, there's no guarantee it'll be handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

Chain of Responsibility (Behavioral Pattern)

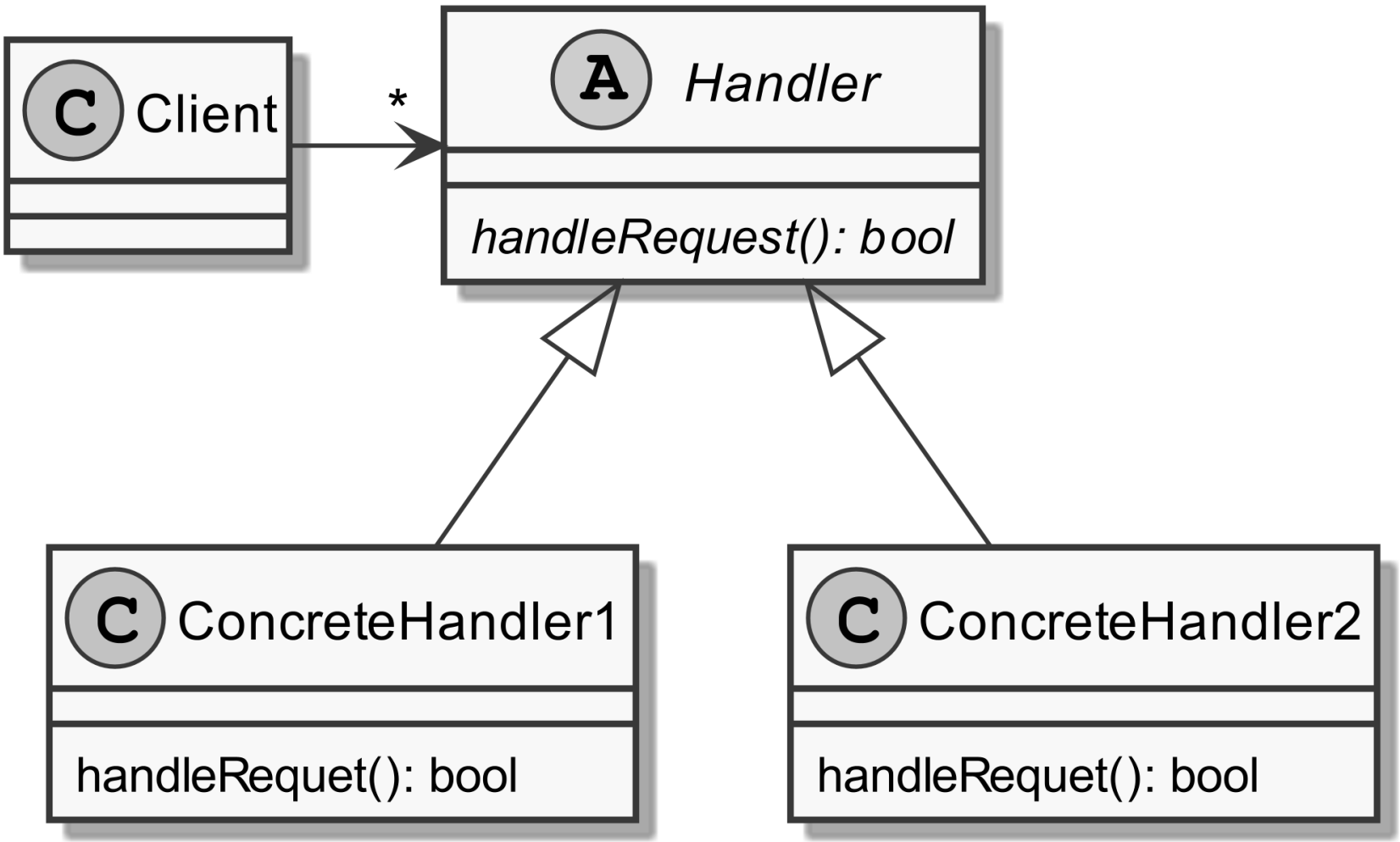
Related Patterns

Chain of Responsibility is often applied in conjunction with Composite (163). There, a component's parent can act as its successor.

Designproblem: Verschiedene Spielerfiguren

- Chain of Responsibility löst die Anforderungen, die am Beginn der Aufgabenstellung genannt wurden.
- Die Kette an Nachfolger-Handlern ermöglicht es, leicht neue Verhalten einzuführen
- *Aber:* Die verkettete Struktur der Nachfolger-Handler erschwert die Konfiguration der Handler für die Spiele-Designer
- *Daher:* Verwendung von Handlern ohne Verkettung; der Client ruft die Handler explizit der Reihe nach auf
- Handler geben Boole'schen Wert zurück, der angibt ob sie die die Aufgabe bearbeitet haben

Designproblem: Verschiedene Spielerfiguren



Designproblem: Verschiedene Spielerfiguren

Bemerkung

Wir hatten ursprünglich die Reaktion der Spielfigur auf Pickups direkt in der Klasse `APGCharacter` implementiert. Wäre das nicht ausreichend gewesen?

Das war aus mehreren Gründen kein besonders gutes Design:

- Jede Art von Spielfigur muss die Reaktion auf alle Pickup-Typen selber implementieren. Wir wollen die Möglichkeit haben, gemeinsame Verhalten wieder zu verwenden.
- Die Änderung des Verhaltens zur Laufzeit ist schwierig und muss für jede Art von Figur gesondert implementiert werden.

Designproblem: Benutzeroberfläche

- Komponenten in der Benutzeroberfläche haben eine hierarchische Struktur:
  - Bildschirm
  - Fenster
  - Eingabemaske
  - Eingabefeld, Button, etc.
- Das Interface der Komponenten soll einheitlich sein

Composite (Structural Pattern)

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation

Graphical editors let users build complex diagrams from simpler components. Components can be grouped to form larger components, which can themselves be grouped to form even larger components. Code that uses these components should not have to distinguish between primitives and complex components. The *Composite* pattern describes how to use recursive composition so that clients don't have to make this distinction.

Composite (Structural Pattern)

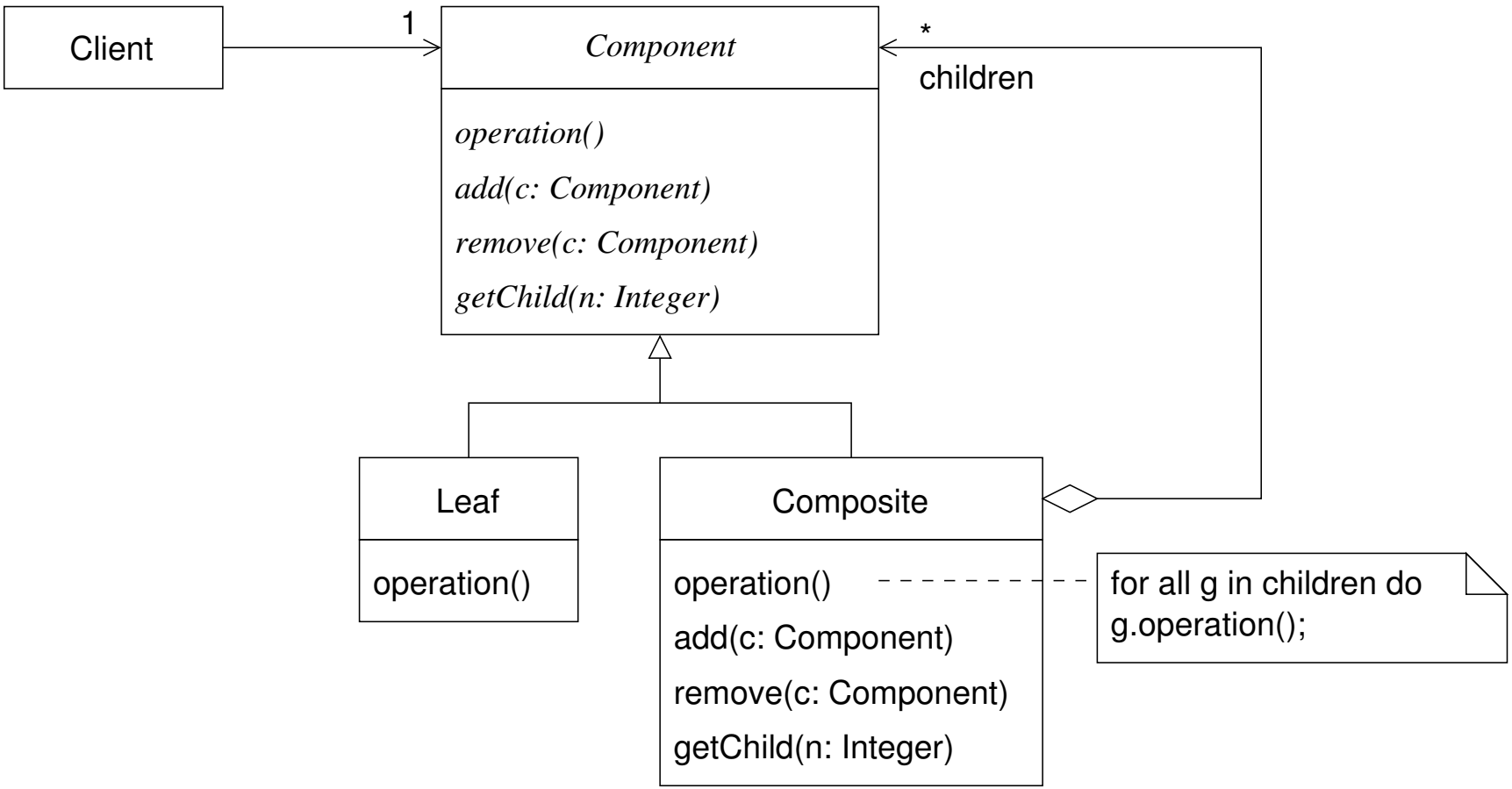
Applicability

Use the Composite pattern when

- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite (Structural Pattern)

Structure



Composite (Structural Pattern)

Consequences

The *Composite* pattern

- defines class hierarchies consisting of primitive and composite objects
- makes the clients simple
- makes it easier to add new kinds of components
- can make your design overly general

Composite (Structural Pattern)

Known uses `Composite` and `Control` in SWT, abstract syntax trees in compilers, ...

Related Patterns

- Chain of Command
- Decorator
- Iterator
- Visitor
- ...

Example ...

Designproblem: Unabhängige Item-Erzeuger

- Bevor das Spiel startet und nach dem Ende des Spiels sollen alle Item-Erzeuger angehalten werden.
- Der Stand des Spiels wird von einer Instanz der Klasse UGameMode verwaltet
- UGameMode soll nichts über Item-Erzeuger wissen, sondern nur die Transitionen zwischen Spiel aktiv/gewonnen/verloren verwalten
- APickup soll nichts über die Funktionalität von UGameMode wissen

Observer (Behavioral Pattern)

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Motivation

When partitioning a system into collaborating classes it is often necessary to maintain consistency between related objects. We don't want to have tight coupling between these classes. The Observer pattern describes how a *subject* may have any number of *observers* that are kept update about changes to the subject.

Observer (Behavioral Pattern)

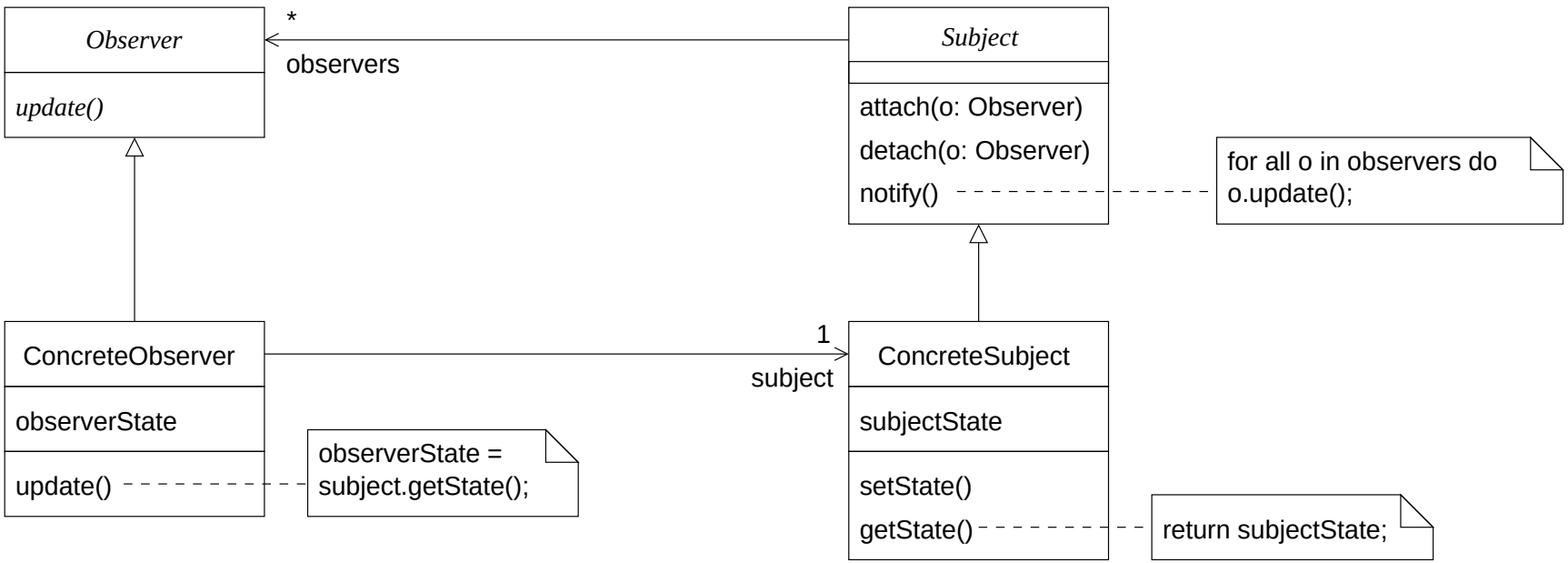
Applicability

Use the Observer pattern when

- An abstraction has two aspects and one of them depends on the other.
- When a change in one object causes change in several others and you don't know exactly how many or which ones.
- When an Object needs to notify other objects without making assumptions what these objects are.

Observer (Behavioral Pattern)

Structure



Observer (Behavioral Pattern)

Participants

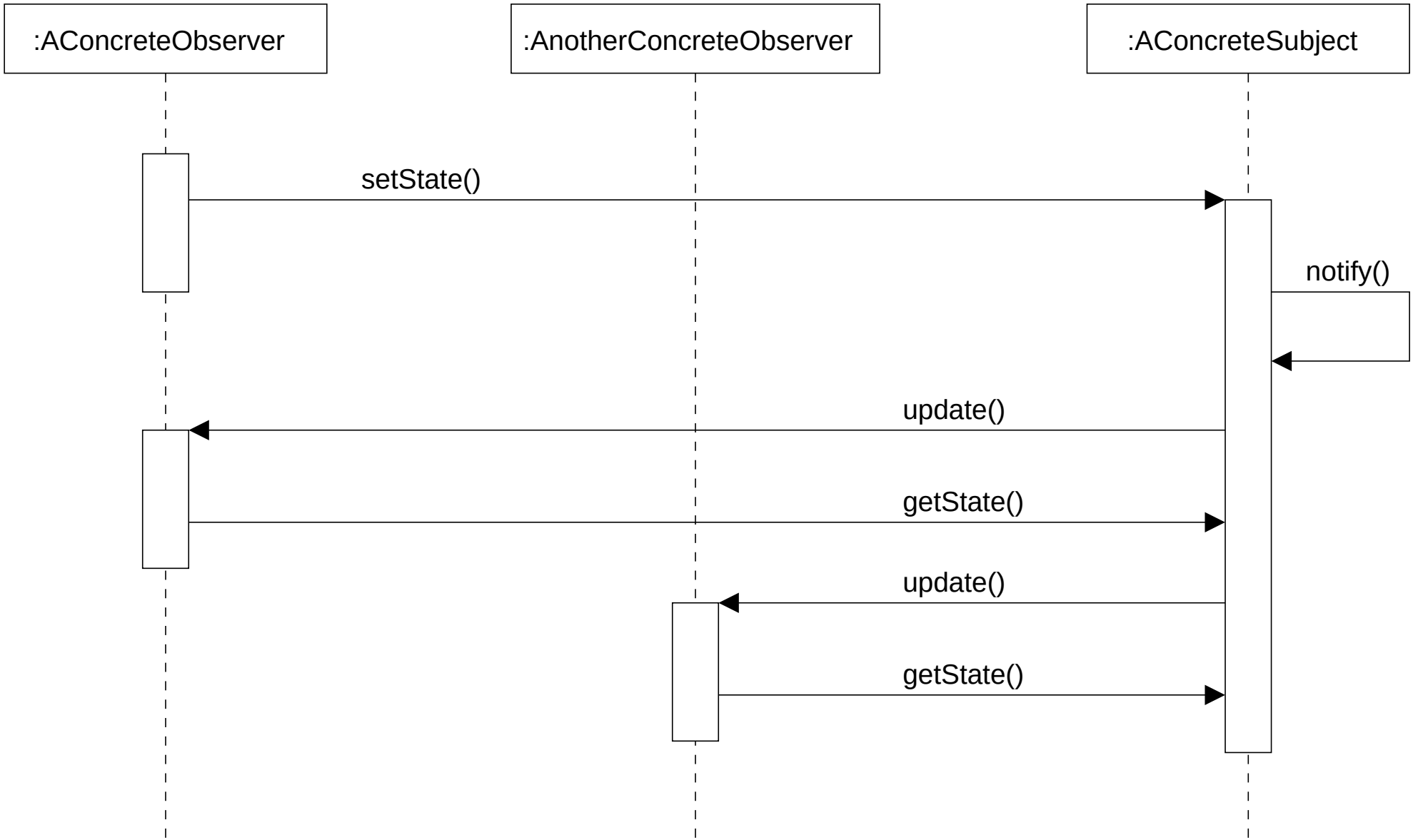
- Subject
  - knows its observers. Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers when its state changes
- ConcreteObserver
  - maintains a reference to a ConcreteSubject object
  - stores state that should stay consistent with the subject's
  - implements the Observer updating interface to keep its state consistent with the subject's

Observer (Behavioral Pattern)

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Observer (Behavioral Pattern)



Observer (Behavioral Pattern)

Consequences

The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers. Further benefits and liabilities of the Observer pattern include the following:

- Abstract coupling between Subject and Observer [...]
- Support for broadcast communication [...]
- Unexpected updates [...]

Observer (Behavioral Pattern)

Known uses

Event listeners in user interfaces (SWT)

Related Patterns

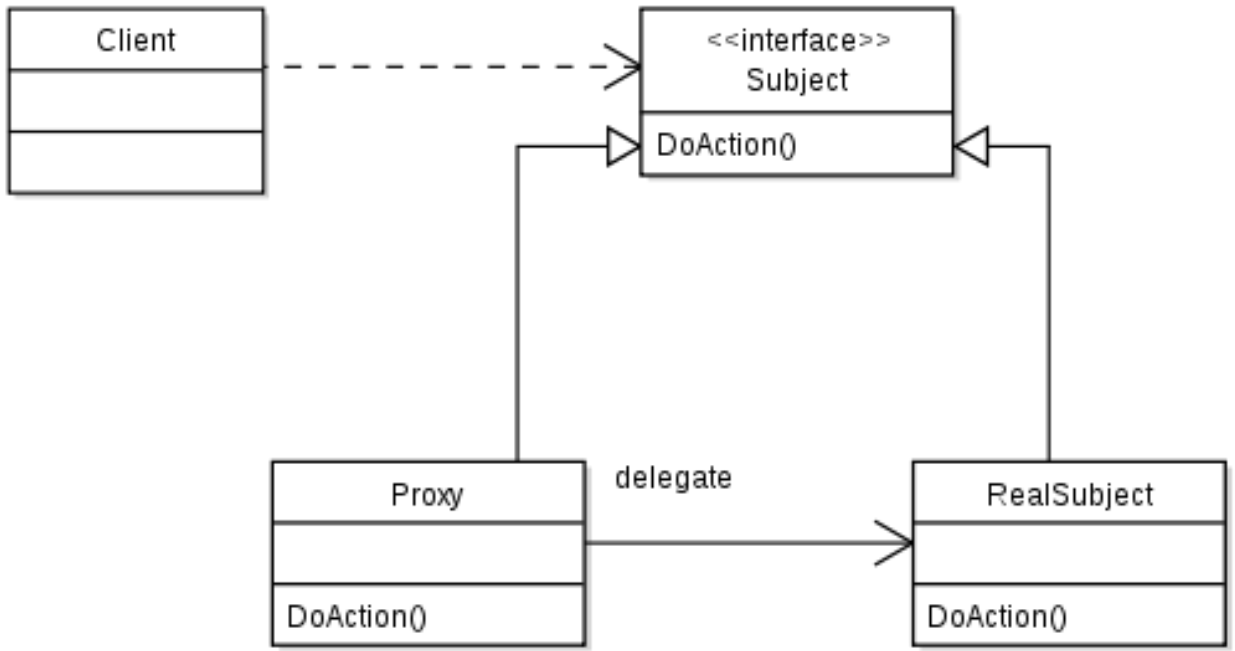
- Mediator (273): By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers
- Singleton (127): The ChangeManager [in a variant implementation of the pattern] may use the Singleton pattern to make it unique and globally accessible

Proxy (Structural Pattern)

Intent

Provide a surrogate or placeholder for another object to control access to it.

Structure

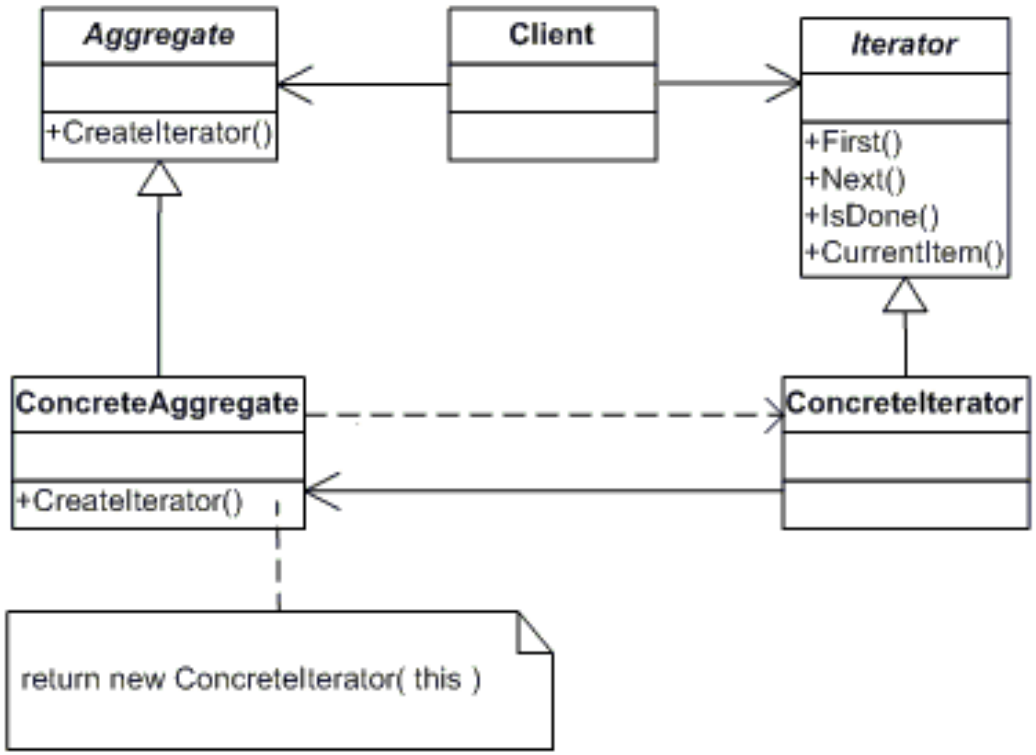


Iterator (Behaviorual Pattern)

Intent

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

Structure



Zusammenfassung

- Ein Entwurfsmuster liefert eine generische Beschreibung einer bewährten Lösung für eine wiederkehrende Problemklasse.
- Die Beschreibung eines Entwurfsmuster erfolgt in einer strukturierten Form, die aus verschiedenen Elementen besteht (Name des Musters, Zweck, Anwendbarkeit, Lösungsstruktur, ...)
- Im Design-Pattern Katalog von Gamma et al. werden 23 objektorientierte Pattern beschrieben.
- Diese Pattern sind nach den Gesichtspunkten „Creational“ (z.B. Abstract Factory), „Structural“ (z.B. Composite) und „Behavioral“ (z.B. State, Observer) klassifiziert.