DESIGN PATTERNS

software
development
academy

Author: Rafał Roppel

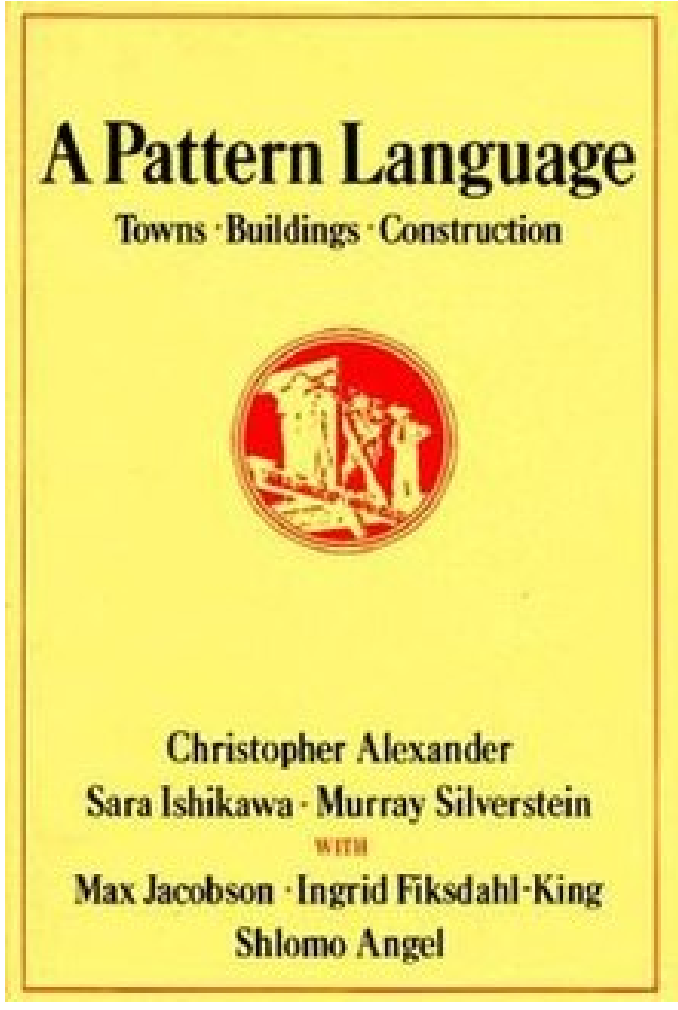
ORIGIN OF PATTERNS

Author: Rafał Roppel

CHRISTOPHER ALEXANDER

Author: Rafał Roppel

A PATTERN LANGUAGE



Author: Rafał Roppel

THE PATTERN DEFINITION

The elements of this language are entities called patterns. Each pattern describes a problem that 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.

Author: Rafał Roppel

GOF: GANG OF FOUR

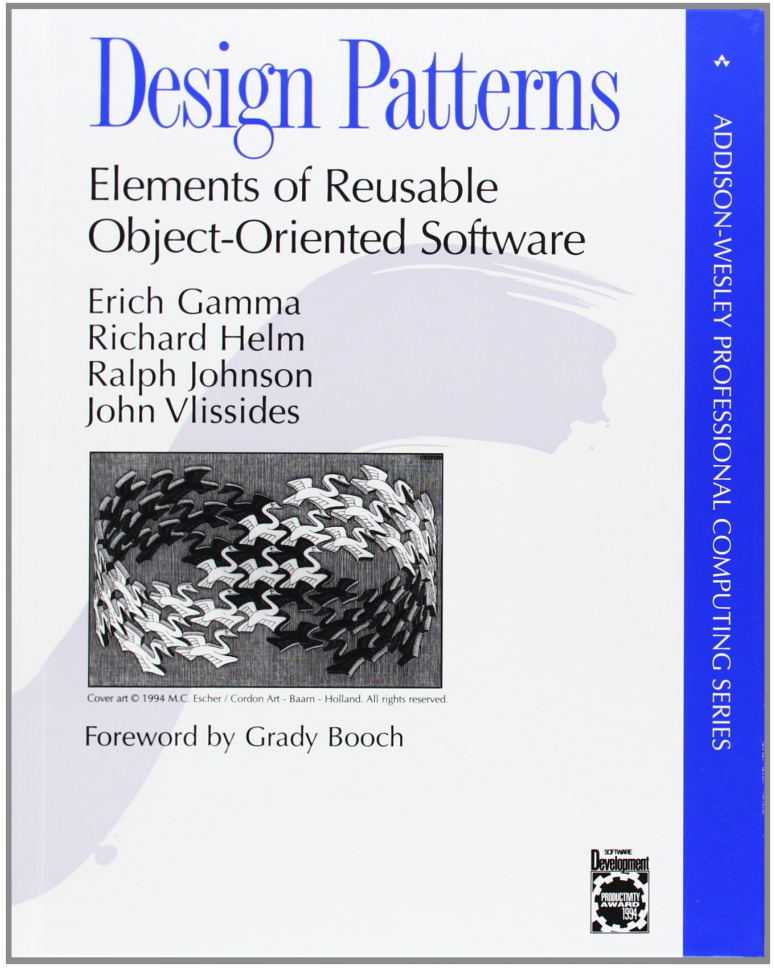
Author: Rafał Roppel

THE "GANG OF FOUR"

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Author: Rafał Roppel

THE DESIGN PATTERNS BOOK



Author: Rafał Roppel

THE PATTERN DEFINITION

The design patterns in this book are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

BASIC INGREDIENTS OF A PATTERN

- **context** - a design situation giving rise to a design problem
- **problem** - a set of forces occuring in that context
- **solution** - a form or rule that can be applied to resolve these forces

Author: Rafał Roppel

TYPES OF DESIGN PATTERNS

- **architectural** - high-level strategies that are concerned with large-scale components and global properties and mechanisms of a system
- **design patterns** - medium-level strategies that are concerned with the structure and behavior of entities, and their relationships
- **idioms** - paradigm-specific and language-specific programming techniques that fill in low-level internal or external details of a component's structure or behavior

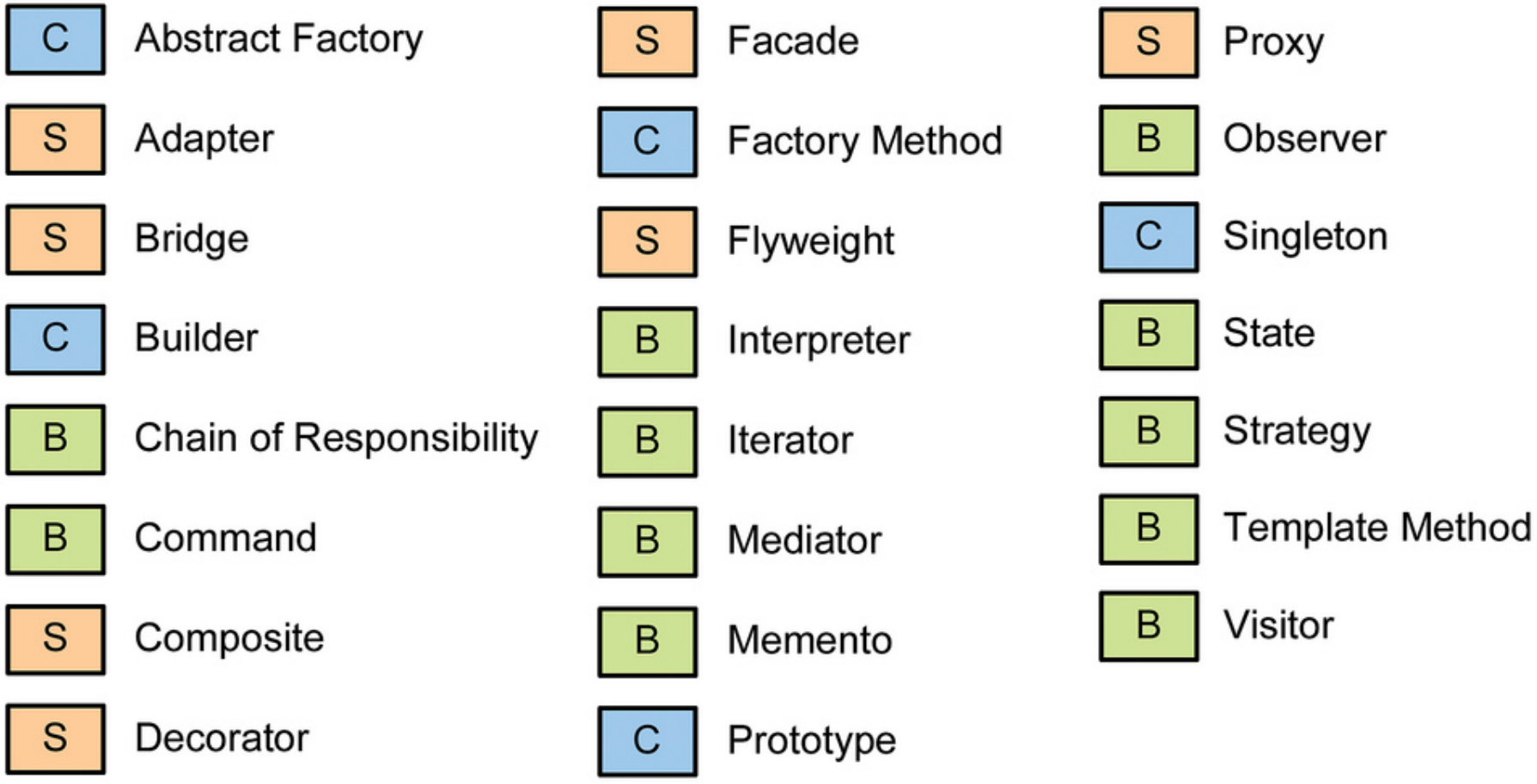
Author: Rafał Roppel

CATEGORIES OF DESIGN PATTERNS

- **creational** - used to construct objects such that they can be decoupled from their implementing system
- **structural** - used to form large object structures between many disparate objects
- **behavioral** - used to manage algorithms, relationships, and responsibilities between objects

Author: Rafał Roppel

THE 23 GANG OF FOUR DESIGN PATTERNS



Author: Rafał Roppel

CREATIONAL PATTERNS

Author: Rafał Roppel

CREATIONAL PATTERNS

- **Abstract Factory** (aka Kit)
- **Builder**
- **Factory Method**
- Prototype
- **Singleton**

Author: Rafał Roppel

ABSTRACT FACTORY

Author: Rafał Roppel

ABSTRACT FACTORY

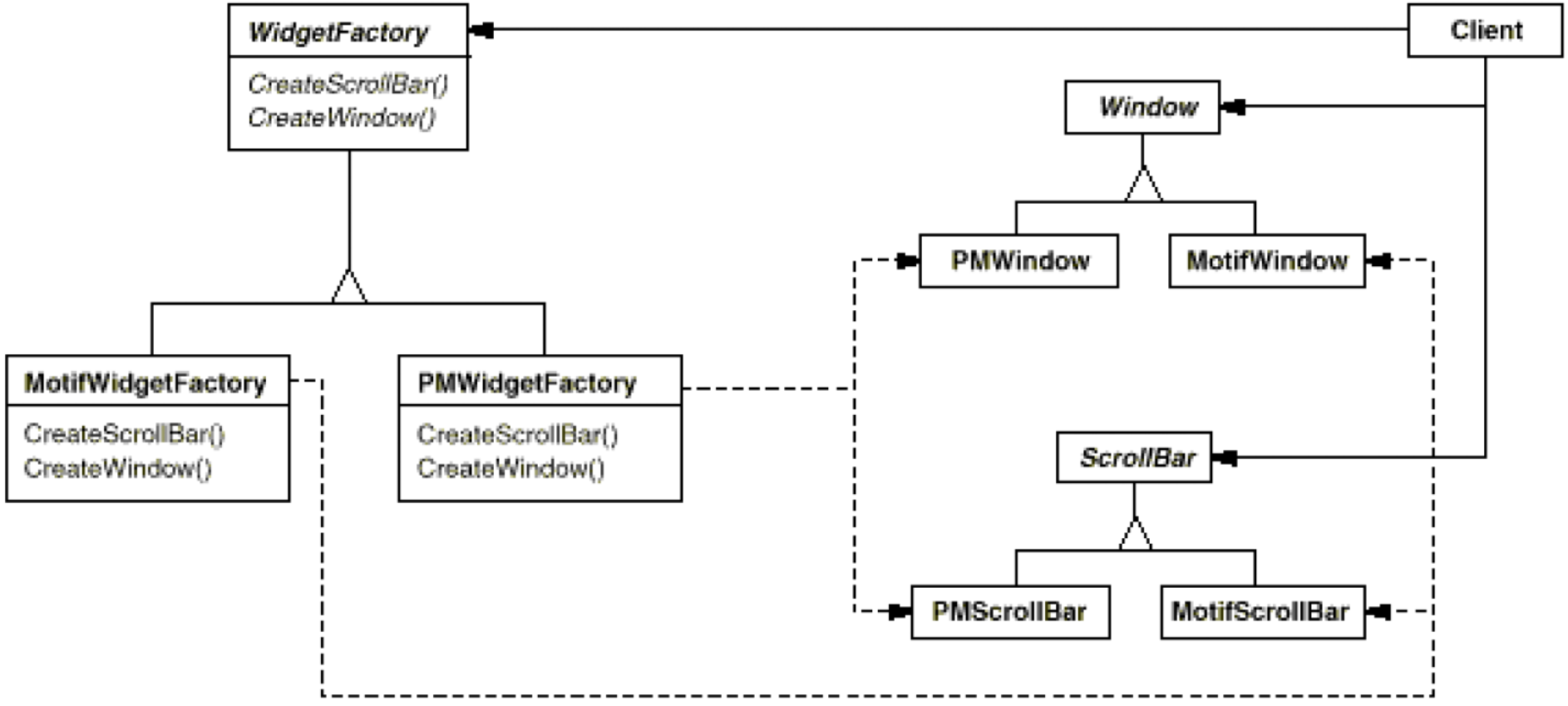
Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

- the creation of objects should be independent of the system utilizing them
- systems should be capable of using multiple families of objects
- families of objects must be used together
- libraries must be published without exposing implementation details
- concrete classes should be decoupled from clients

Author: Rafał Roppel

ABSTRACT FACTORY

diagram



Author: Rafał Roppel

ABSTRACT FACTORY

examples in JDK

- `java.util.Calendar#getInstance()`
- `java.util.Arrays#asList()`
- `java.util.ResourceBundle#getBundle()`
- `java.sql.DriverManager#getConnection(`
- `java.sql.Connection#createStatement()`
- `java.sql.Statement#executeQuery()`
- `java.text.NumberFormat#getInstance()`

Author: Rafał Roppel

ABSTRACT FACTORY

DEMO

Author: Rafał Roppel

BUILDER

Author: Rafał Roppel

BUILDER

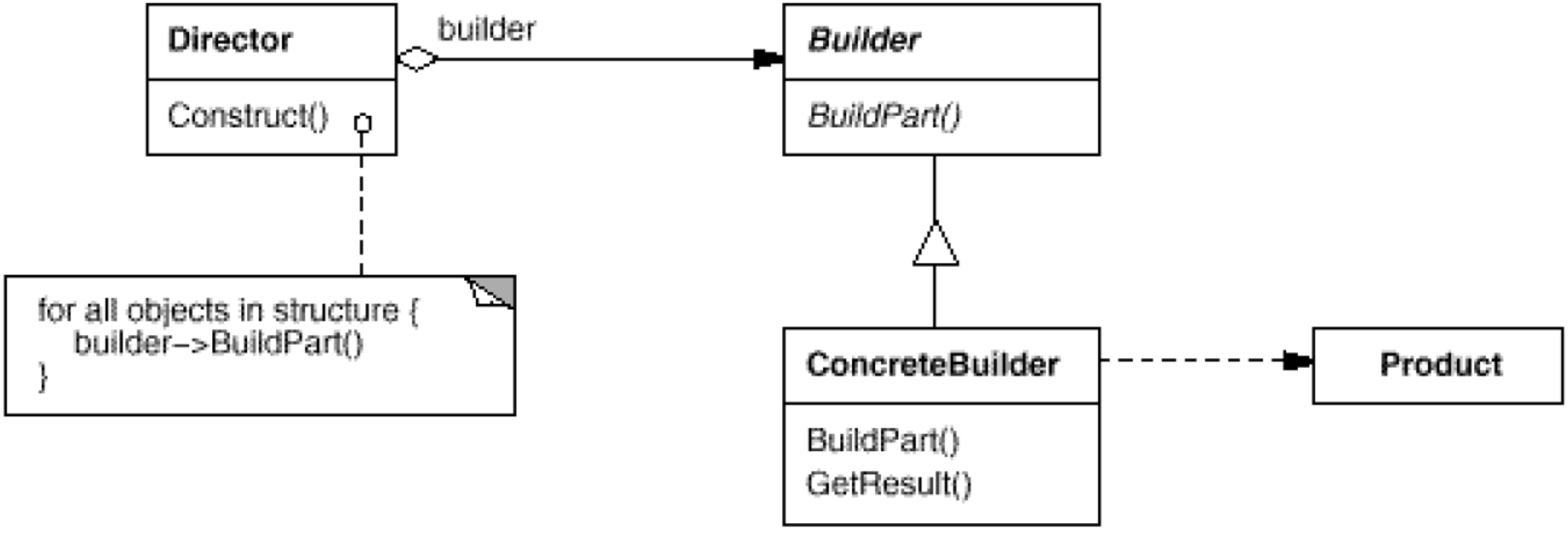
Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

- object creation algorithms should be decoupled from the system
- multiple representations of creation algorithms are required
- the addition of new creation functionality without changing the core code is necessary
- runtime control over the creation process is required

Author: Rafał Roppel

BUILDER

diagram



Author: Rafał Roppel

BUILDER

examples in JDK

- `java.lang.StringBuilder#append()`
- `java.lang.StringBuffer#append()`
- `java.nio.ByteBuffer#put()` (also on `CharBu`
  `ShortBuffer`, `IntBuffer`, `LongBuffer`, `Float`
  and `DoubleBuffer`)
- `javax.swing.GroupLayout.Group#addComp`
- all implementations of `java.lang.Appendable`

Author: Rafał Roppel

BUILDER

DEMO

Author: Rafał Roppel

FACTORY METHOD

Author: Rafał Roppel

FACTORY METHOD

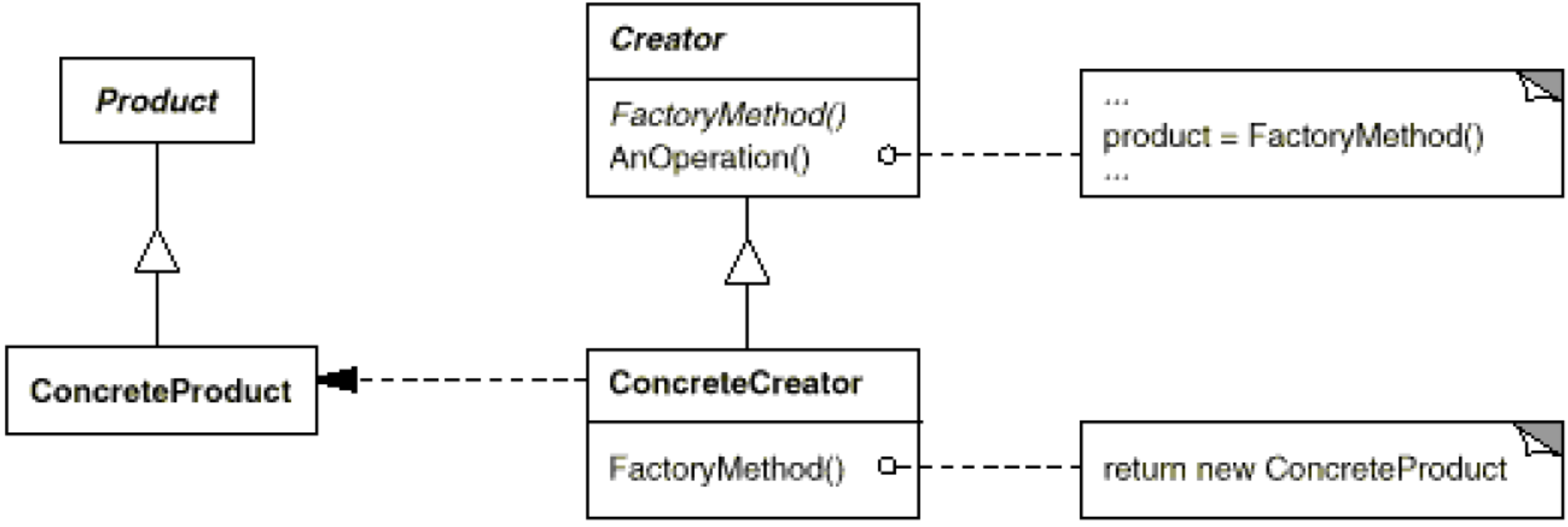
Exposes a method for creating objects, allowing subclasses to control the actual creation process.

- a class will not know what classes it will be required to create
- subclasses may specify what objects should be created
- parent classes wish to defer creation to their subclasses

Author: Rafał Roppel

FACTORY METHOD

diagram



Author: Rafał Roppel

FACTORY METHOD

examples in JDK

- `java.util.Calendar#getInstance()`
- `java.util.ResourceBundle#getBundle()`
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`

Author: Rafał Roppel

FACTORY METHOD

DEMO

Author: Rafał Roppel

PROTOTYPE

Author: Rafał Roppel

PROTOTYPE

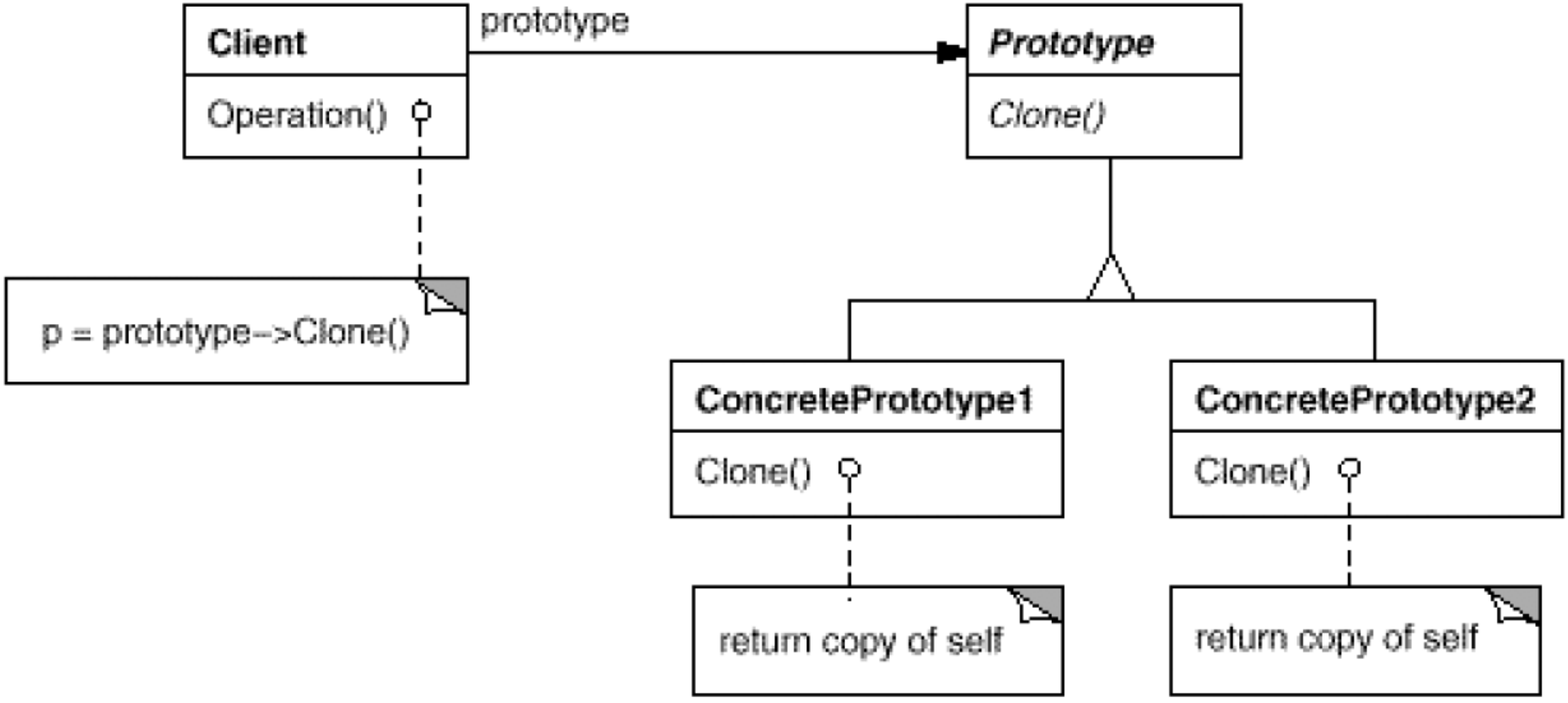
Exposes a method for creating objects, allowing subclasses to control the actual creation process.

- a class will not know what classes it will be required to create
- subclasses may specify what objects should be created
- parent classes wish to defer creation to their subclasses

Author: Rafał Roppel

PROTOTYPE

diagram



Author: Rafał Roppel

PROTOTYPE

examples in JDK

- `java.lang.Object#clone()`
- `java.lang.Cloneable`

Author: Rafał Roppel

SINGLETON

Author: Rafał Roppel

SINGLETON

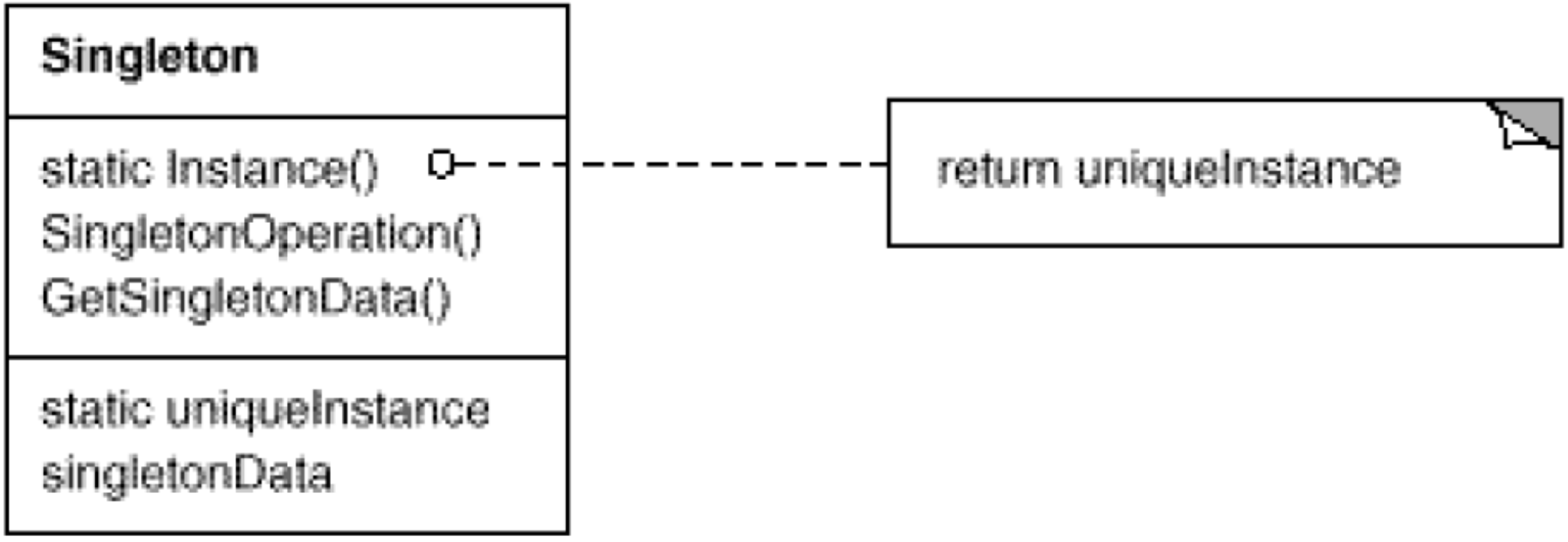
Ensures that only one instance of a class is allowed within a system.

- exactly one instance of a class is required
- controlled access to a single object is necessary

Author: Rafał Roppel

SINGLETON

diagram



Author: Rafał Roppel

SINGLETON

examples in JDK

- `java.lang.Runtime#getRuntime()`
- `java.awt.Desktop#getDesktop()`
- `java.lang.System#getSecurityManager()`

Author: Rafał Roppel

SINGLETON

DEMO

Author: Rafał Roppel

STRUCTURAL PATTERNS

Author: Rafał Roppel

STRUCTURAL PATTERNS

- Adapter (aka Wrapper)
- Bridge
- Composite
- **Decorator** (aka Wrapper)
- Facade
- Flyweight
- Proxy (aka Surrogate)

Author: Rafał Roppel

ADAPTER

Author: Rafał Roppel

ADAPTER

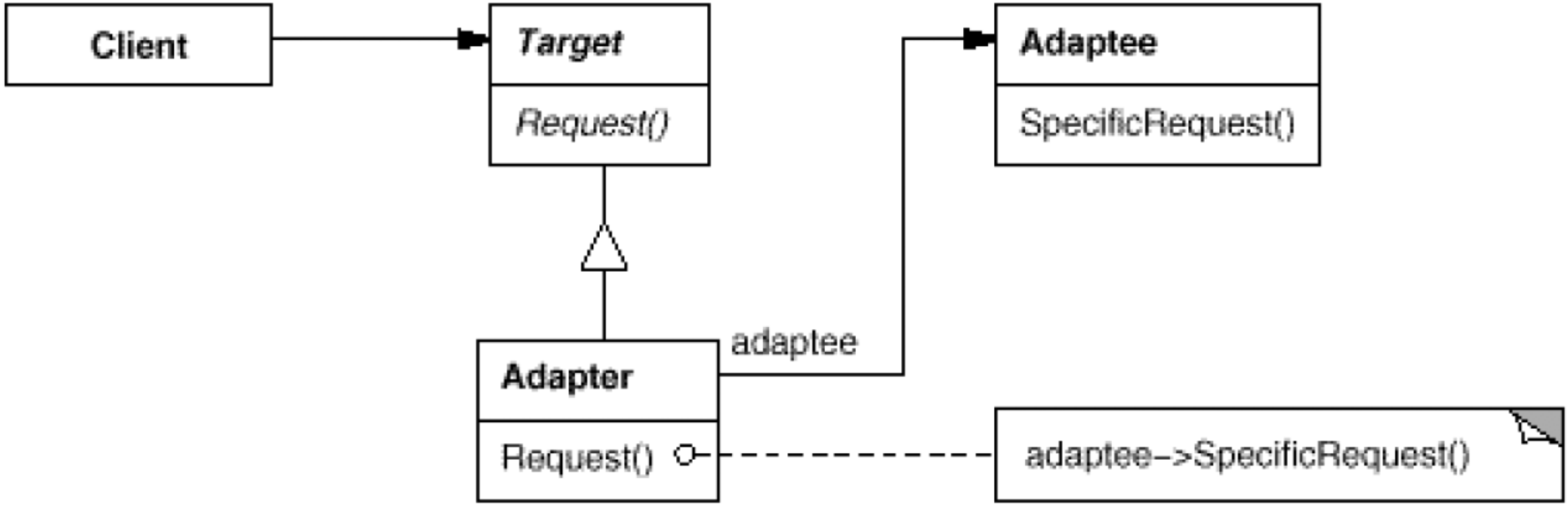
Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

- a class to be used doesn’t meet interface requirements
- complex conditions tie object behavior to its state
- transitions between states need to be explicit

Author: Rafał Roppel

ADAPTER

diagram



Author: Rafał Roppel

ADAPTER

examples in JDK

- `java.util.Arrays#asList()`
- `java.util.Collections#list()`
- `java.util.Collections#enumeration()`
- `java.io.InputStreamReader(InputStream` (returns `Reader`)
- `java.io.OutputStreamWriter(OutputStre` (returns `Writer`)

Author: Rafał Roppel

BRIDGE

Author: Rafał Roppel

BRIDGE

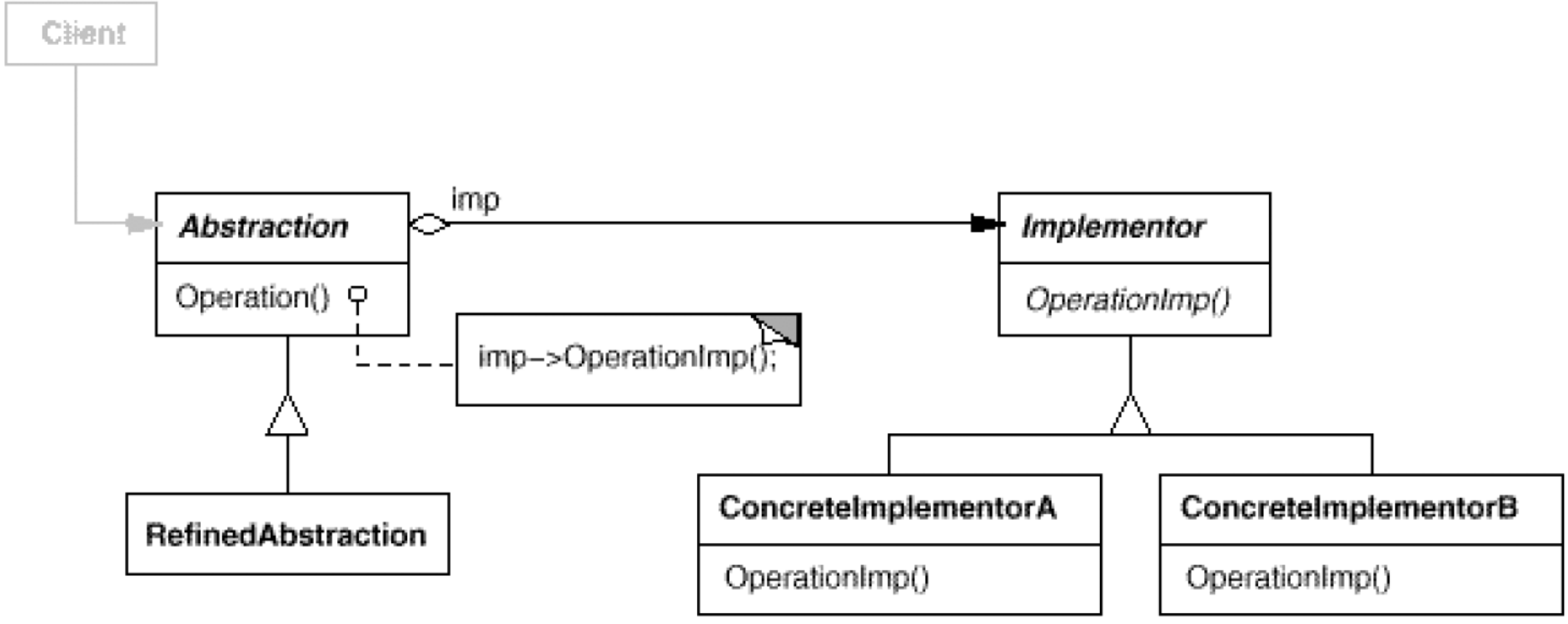
Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

- abstractions and implementations should not be bound at compile time
- abstractions and implementations should be independently extensible
- changes in the implementation of an abstraction should have no impact on clients
- implementation details should be hidden from the client

Author: Rafał Roppel

BRIDGE

diagram



Author: Rafał Roppel

BRIDGE

examples in JDK

- `new LinkedHashMap(LinkedHashSet, List)` which returns an unmodifiable linked map which doesn't clone the items, but uses them
- `java.util.Collections#newSetFromMap()` and `singletonXXX()` methods however comes close

Author: Rafał Roppel

COMPOSITE

Author: Rafał Roppel

COMPOSITE

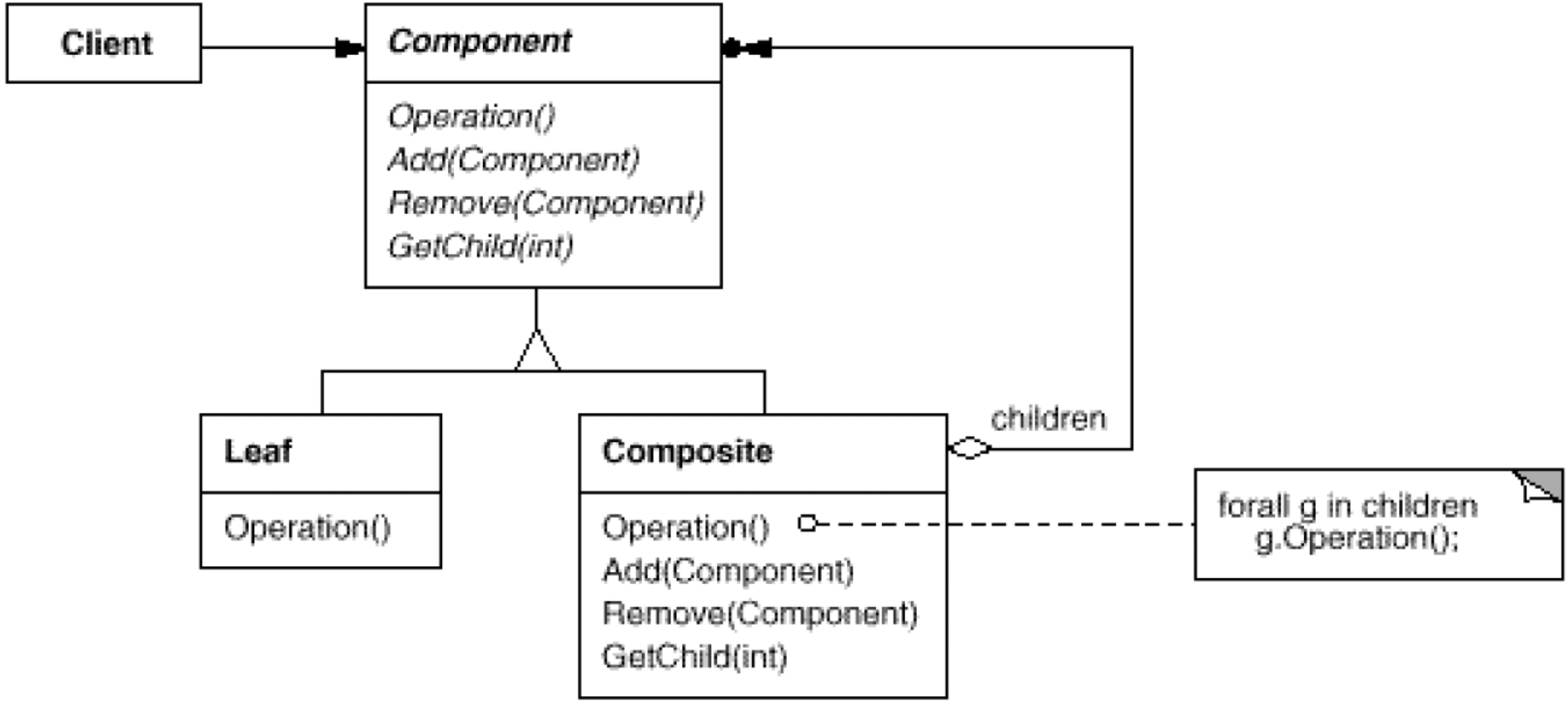
Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

- hierarchical representations of objects are needed
- objects and compositions of objects should be treated uniformly

Author: Rafał Roppel

COMPOSITE

diagram



Author: Rafał Roppel

COMPOSITE

examples in JDK

- `java.awt.Container#add(Component)` (practically all over Swing thus)
- `javax.faces.component.UIComponent#getChildren()` (practically all over JSF UI thus)

DECORATOR

Author: Rafał Roppel

DECORATOR

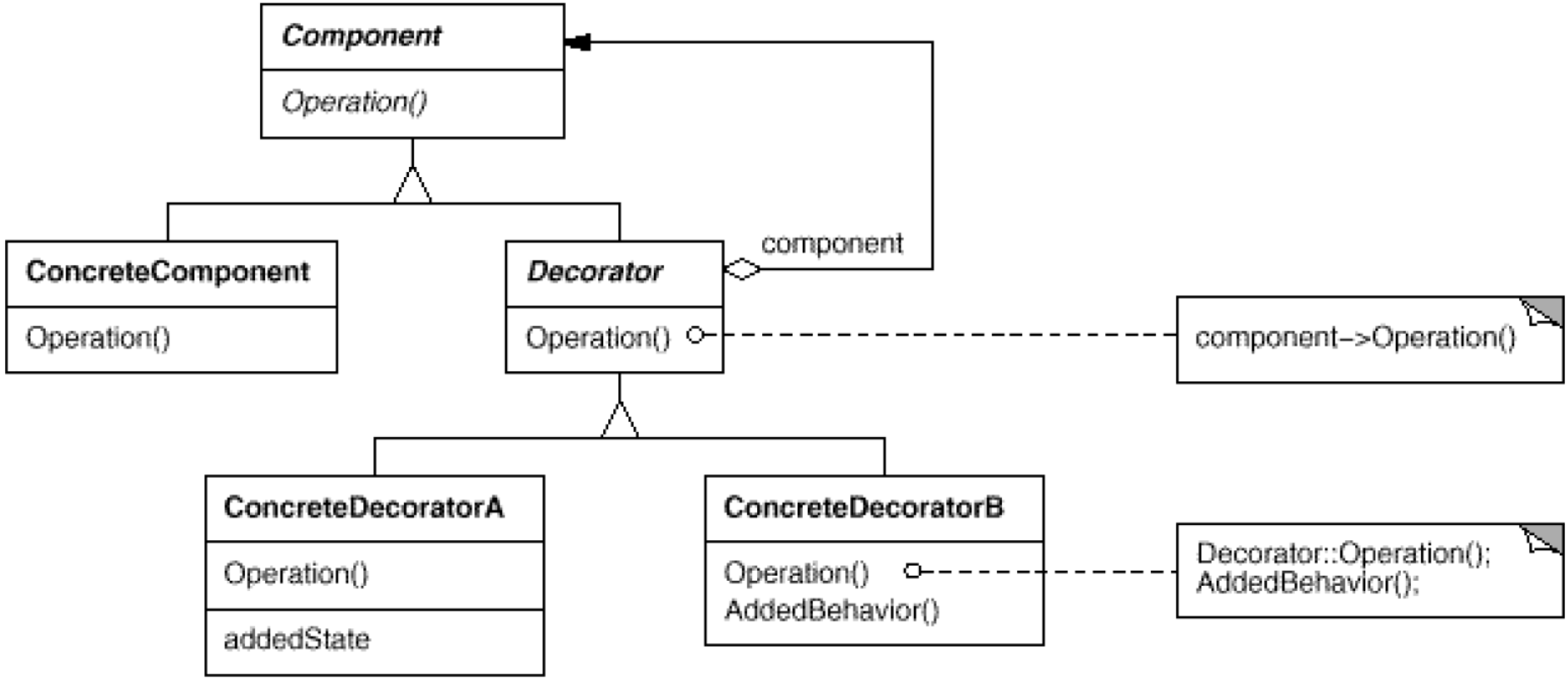
Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

- object responsibilities and behaviors should be dynamically modifiable
- concrete implementations should be decoupled from responsibilities and behaviors
- subclassing to achieve modification is impractical or impossible
- specific functionality should not reside high in the object hierarchy
- a lot of little objects surrounding a concrete implementation is acceptable

Author: Rafał Roppel

DECORATOR

diagram



Author: Rafał Roppel

DECORATOR

examples in JDK

- `java.io.BufferedInputStream(InputStre`
- `java.io.DataInputStream(InputStream)`
- `java.io.BufferedOutputStream(OutputSt`
- `java.util.zip.ZipOutputStream(OutputS`
- `java.util.Collections#checked[List|Ma`
  `SortedSet|SortedMap]()`

Author: Rafał Roppel

DECORATOR

DEMO

Author: Rafał Roppel

FACADE

Author: Rafał Roppel

FACADE

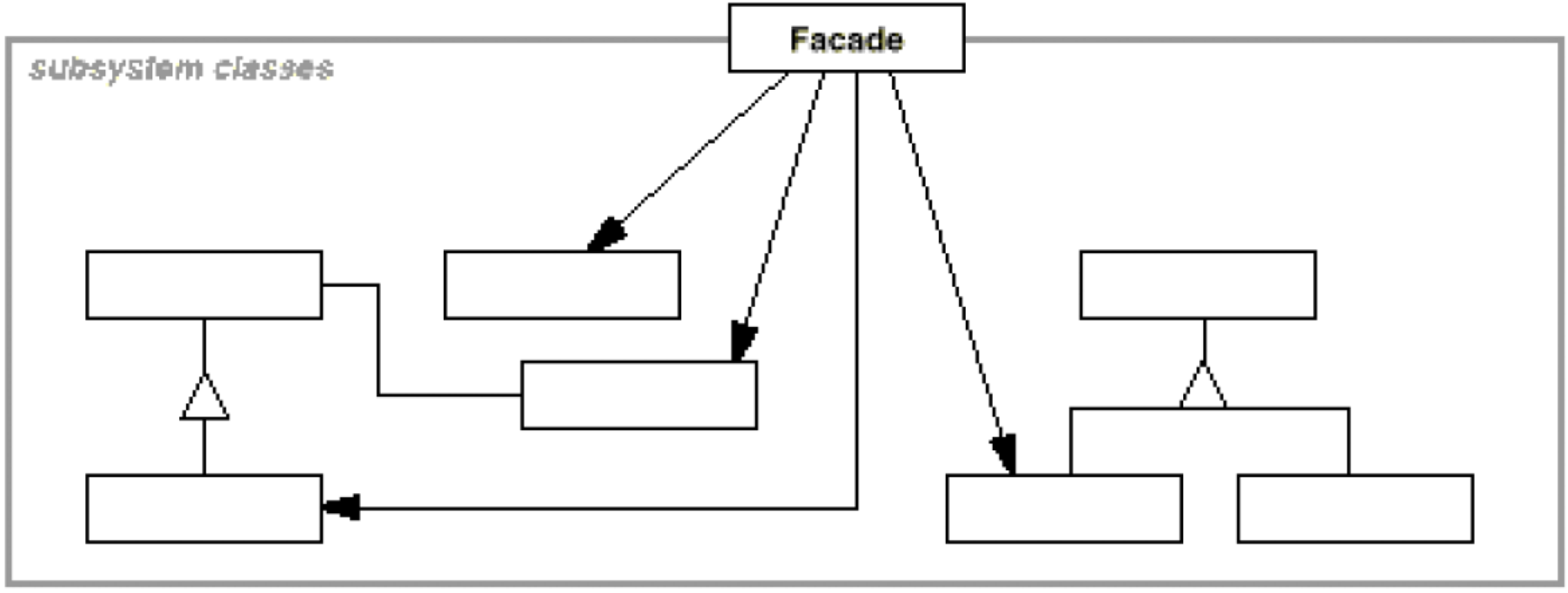
Supplies a single interface to a set of interfaces within a system.

- a simple interface is needed to provide access to a complex system
- there are many dependencies between system implementations and clients
- systems and subsystems should be layered

Author: Rafał Roppel

FACADE

diagram



Author: Rafał Roppel

FACADE

examples in JDK

- `javax.faces.context.FacesContext`it internally uses among others the abstract/interface types `LifeCycle`, `ViewHandler`, `NavigationHandler` and many more without that the enduser has to worry about it (which are however overrideable by injection)
- `javax.faces.context.ExternalContext`, whichinternally uses `ServletContext`, `HttpSession`, `HttpServletRequest`, `HttpServletResponse`, etc.

Author: Rafał Roppel

FLYWEIGHT

Author: Rafał Roppel

FLYWEIGHT

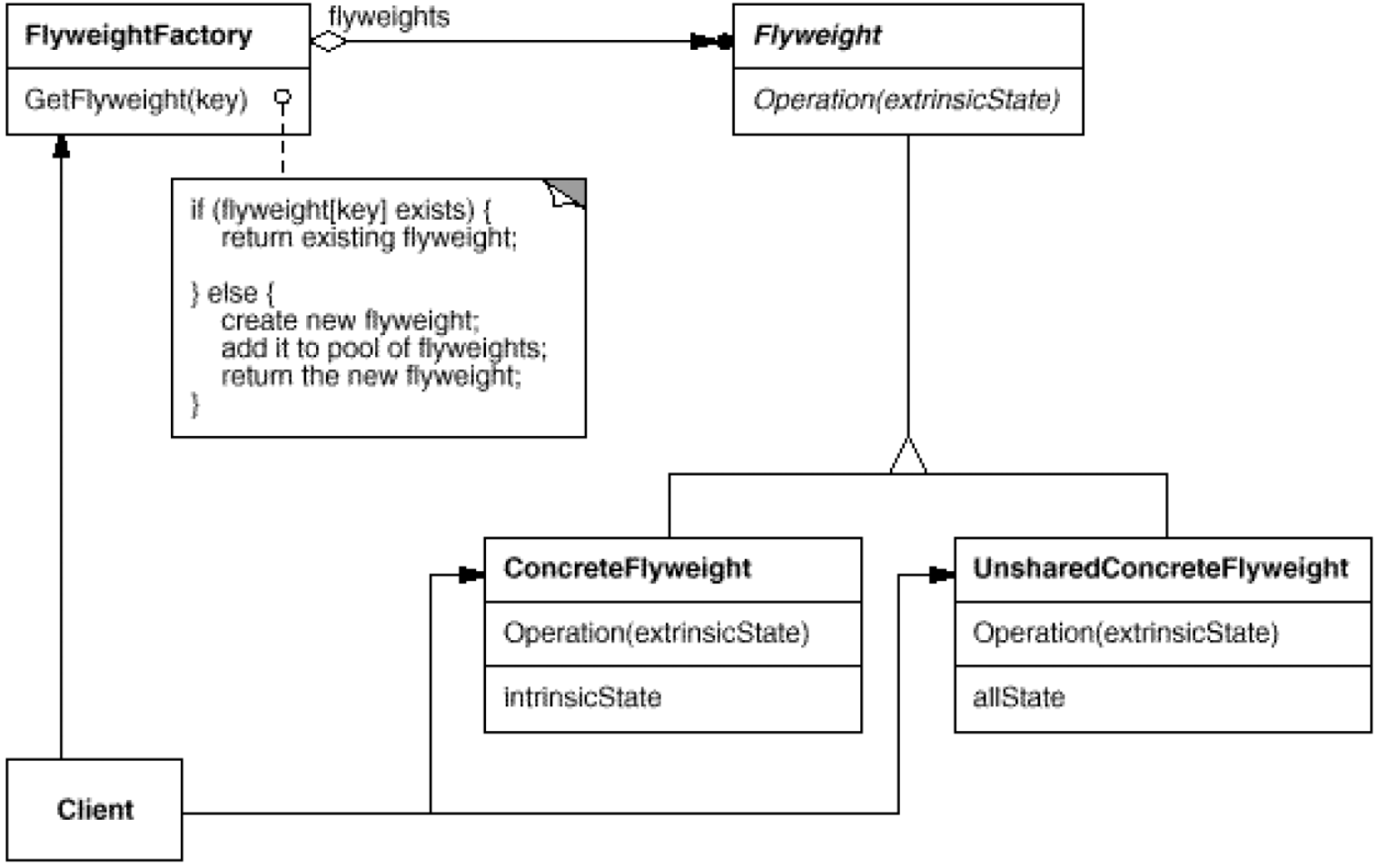
Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

- many like objects are used and storage cost is high
- the majority of each object's state can be made extrinsic
- a few shared objects can replace many unshared ones
- the identity of each object does not matter

Author: Rafał Roppel

FLYWEIGHT

diagram



Author: Rafał Roppel

FLYWEIGHT

examples in JDK

- `java.lang.Integer#valueOf(int)` (also on `Boolean`, `Byte`, `Character`, `Short` and `Long`)

Author: Rafał Roppel

PROXY

Author: Rafał Roppel

PROXY

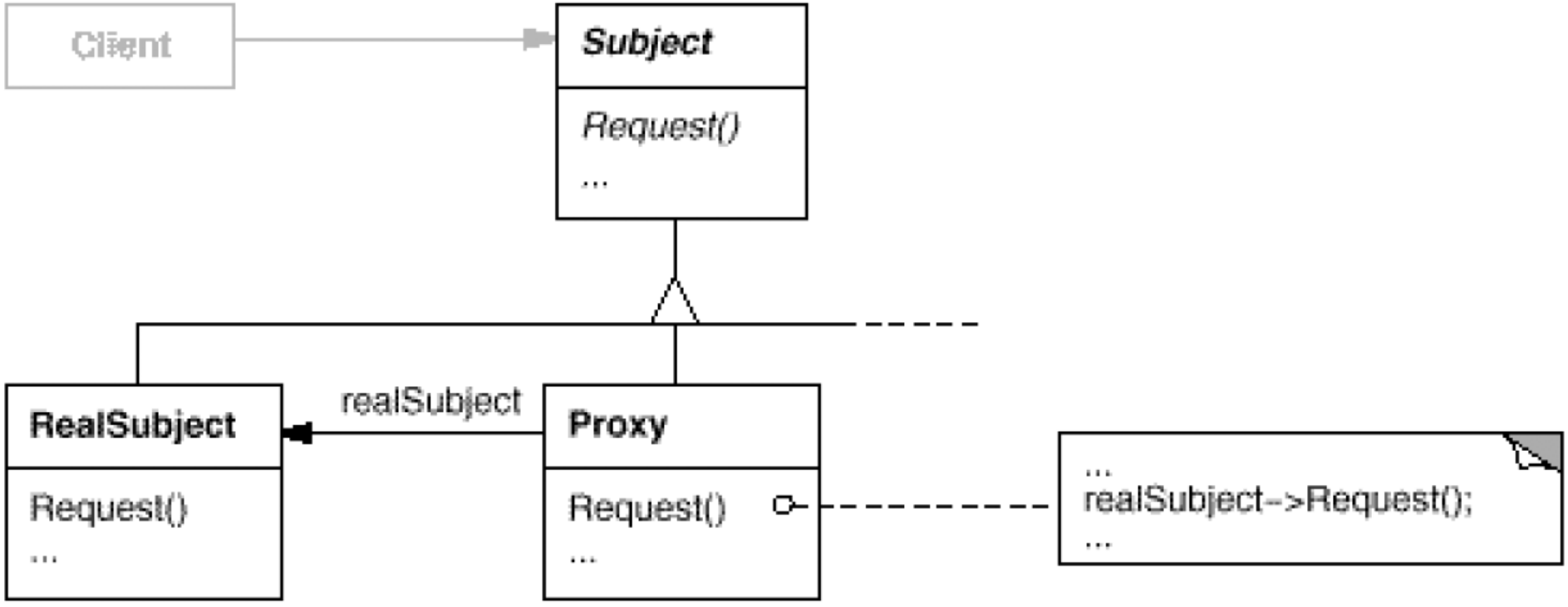
Allows for object level access control by acting as a pass through entity or a placeholder object.

- the object being represented is external to the system
- objects need to be created on demand
- access control for the original object is required
- added functionality is required when an object is accessed

Author: Rafał Roppel

PROXY

diagram



Author: Rafał Roppel

PROXY

examples in JDK

- `java.lang.reflect.Proxy`
- `java.rmi.*` (whole package)

Author: Rafał Roppel

BEHAVIORAL PATTERNS

Author: Rafał Roppel

BEHAVIORAL PATTERNS

- Chain Of Responsibility
- **Command** (aka Action, Transaction)
- Interpreter
- Iterator (aka Cursor)
- Mediator
- Memento (aka Token)
- Observer
- State
- **Strategy** (aka Policy)
- **Template Method**
- **Visitor**

Author: Rafał Roppel

CHAIN OF RESPONSIBILITY

Author: Rafał Roppel

CHAIN OF RESPONSIBILITY

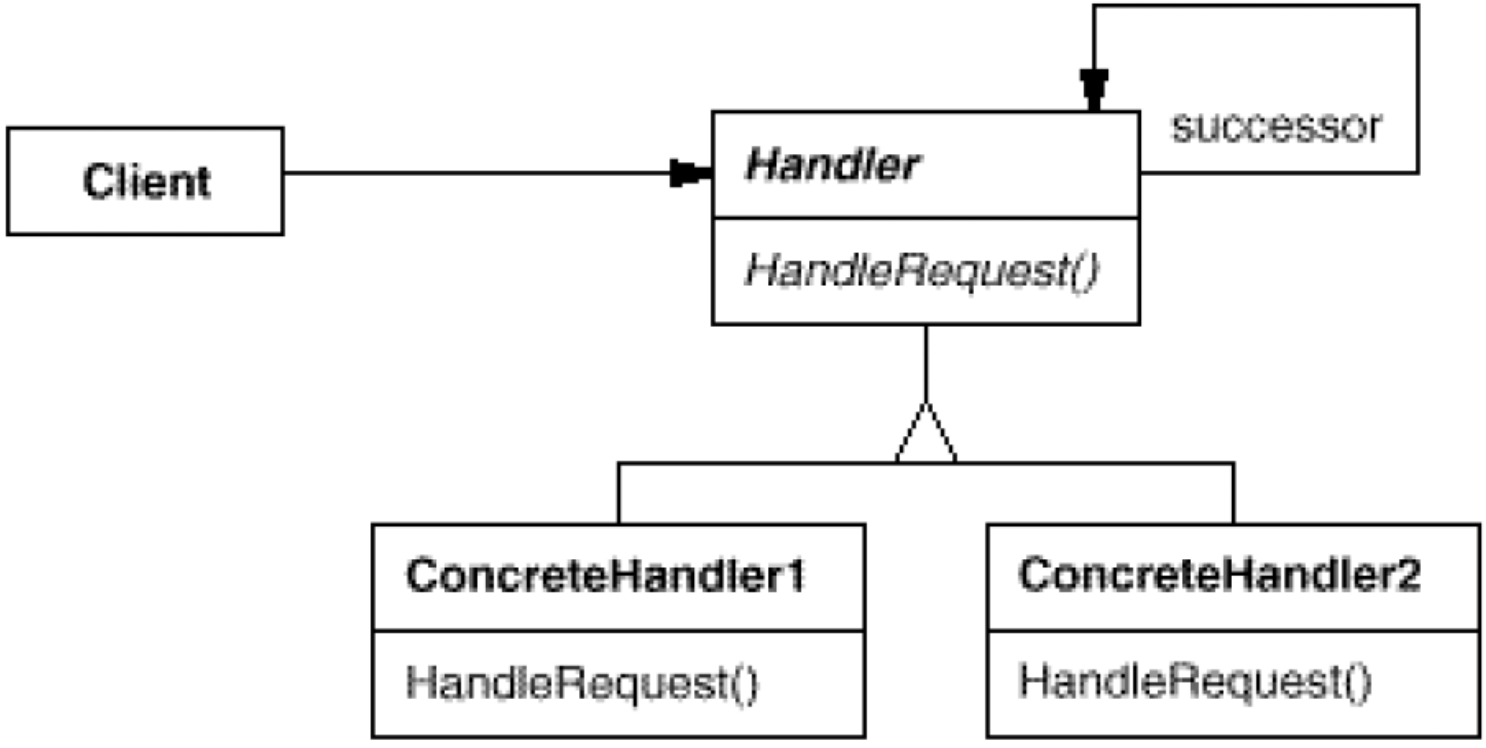
Gives more than one object an opportunity to handle a request by linking receiving objects together.

- multiple objects may handle a request and the handler doesn't have to be a specific object
- a set of objects should be able to handle a request with the handler determined at runtime
- a request not being handled is an acceptable potential outcome

Author: Rafał Roppel

CHAIN OF RESPONSIBILITY

diagram



Author: Rafał Roppel

CHAIN OF RESPONSIBILITY

examples in JDK

- `java.util.logging.Logger#log()`
- `javax.servlet.Filter#doFilter()`

Author: Rafał Roppel

COMMAND

Author: Rafał Roppel

COMMAND

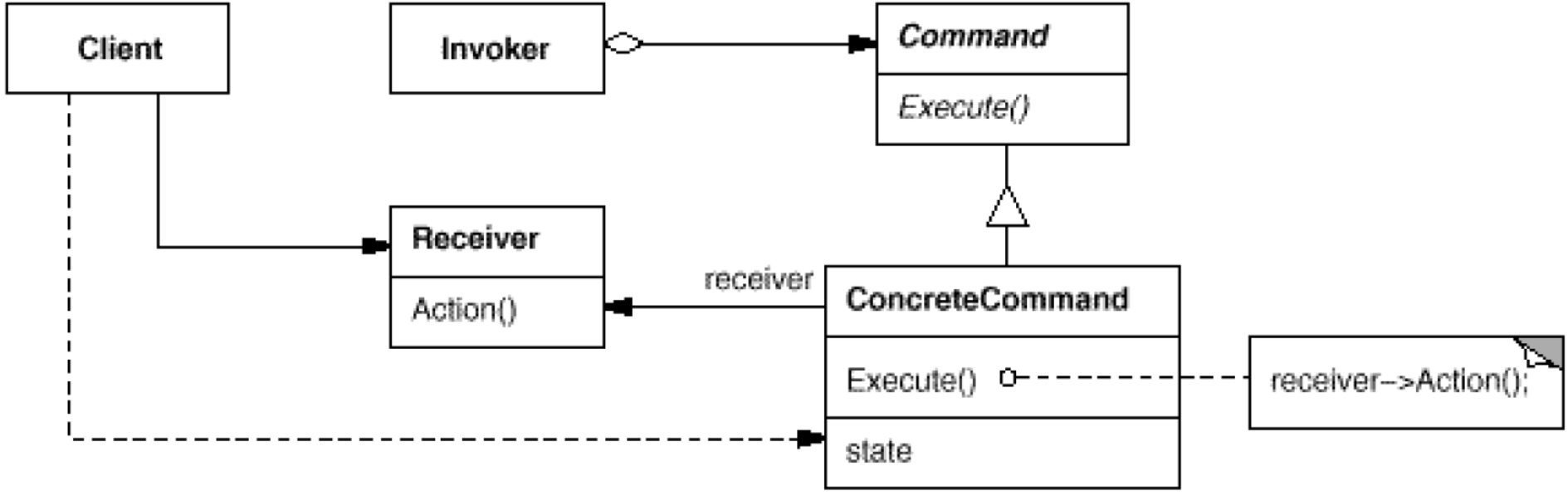
Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks

- you need callback functionality
- requests need to be handled at variant times or in variant orders
- a history of requests is needed
- the invoker should be decoupled from the object handling the invocation

Author: Rafał Roppel

COMMAND

diagram



Author: Rafał Roppel

COMMAND

examples in JDK

- `java.lang.Runnable`
- `javax.swing.Action`

Author: Rafał Roppel

COMMAND

DEMO

Author: Rafał Roppel

INTERPRETER

Author: Rafał Roppel

INTERPRETER

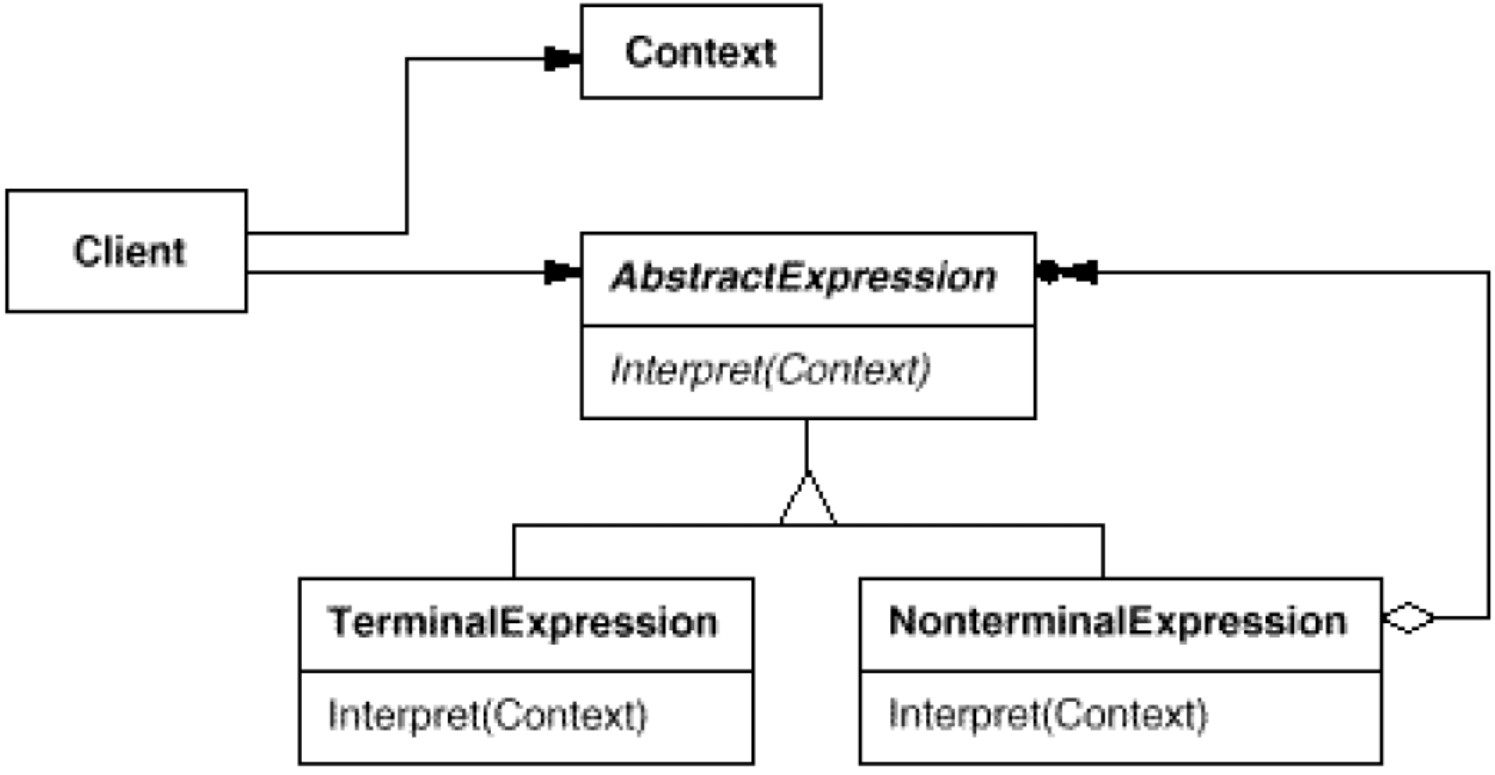
Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

- there is grammar to interpret that can be represened as large syntax trees
- the grammar is simple
- efficiency is not important
- decoupling grammar from underlying expressions is desired

Author: Rafał Roppel

INTERPRETER

diagram



Author: Rafał Roppel

INTERPRETER

examples in JDK

- `java.util.Pattern`
- `java.text.Normalizer`
- `java.text.Format`

Author: Rafał Roppel

ITERATOR

Author: Rafał Roppel

ITERATOR

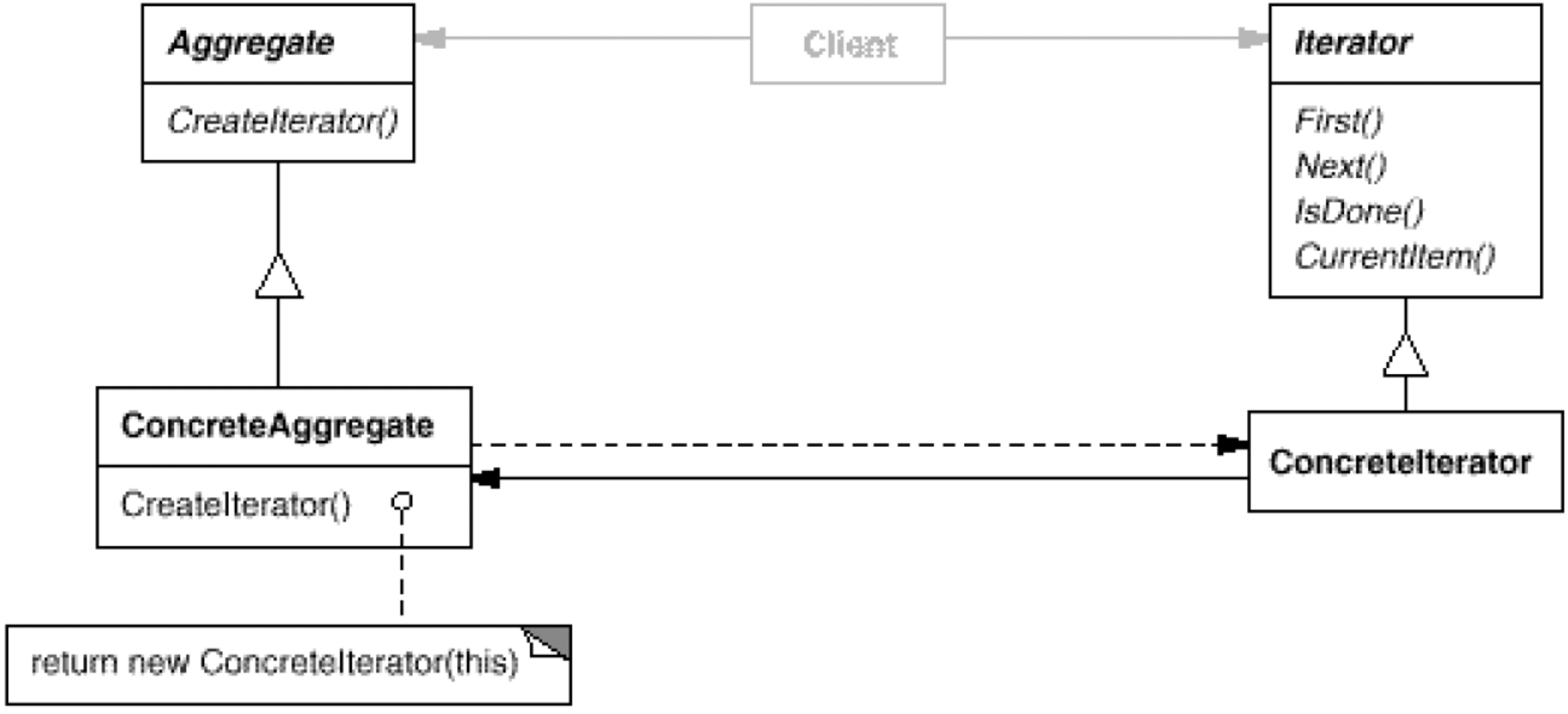
Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

- access to elements is needed without access to the entire representation
- multiple or concurrent traversals of the elements are needed
- a uniform interface for traversal is needed
- subtle differences exist between the implementation details of various iterators

Author: Rafał Roppel

ITERATOR

diagram



Author: Rafał Roppel

ITERATOR

examples in JDK

- `java.util.Iterator`
- `java.util.Enumeration`

Author: Rafał Roppel

MEDIATOR

Author: Rafał Roppel

MEDIATOR

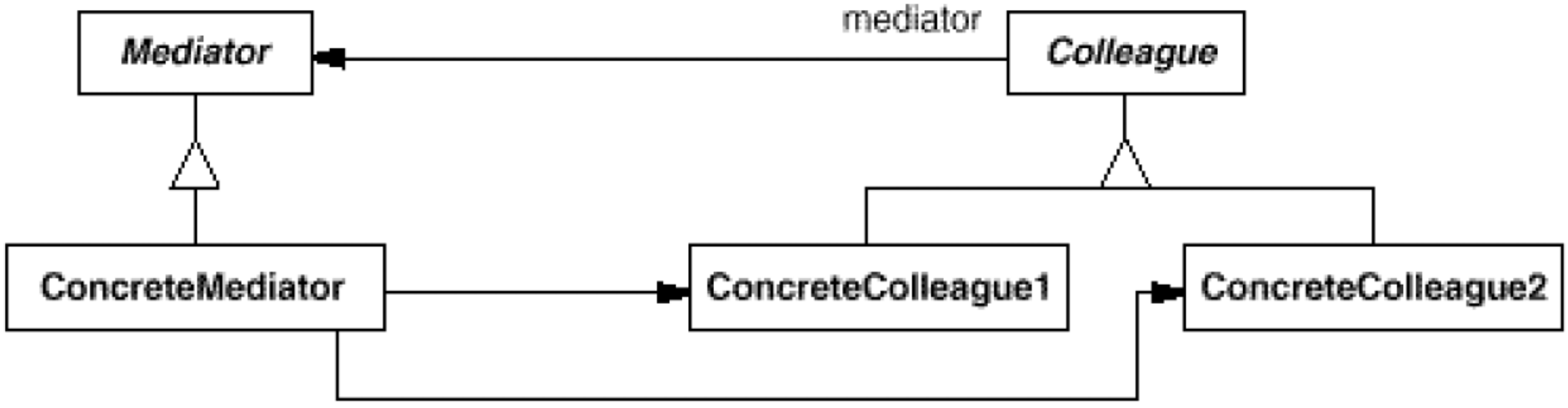
Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

- communication between sets of objects is well defined and complex
- too many relationships exist and common point of control or communication is needed

Author: Rafał Roppel

MEDIATOR

diagram



Author: Rafał Roppel

MEDIATOR

examples in JDK

- `java.util.concurrent.ScheduledExecutorServ` (all `scheduleXXX()` methods)
- `java.util.concurrent.ExecutorService` (the `invokeXXX()` and `submit()` methods)
- `java.util.concurrent.Executor#execute()`
- `java.util.Timer` (all `scheduleXXX()` methods)
- `java.lang.reflect.Method#invoke()`

Author: Rafał Roppel

MEMENTO

Author: Rafał Roppel

MEMENTO

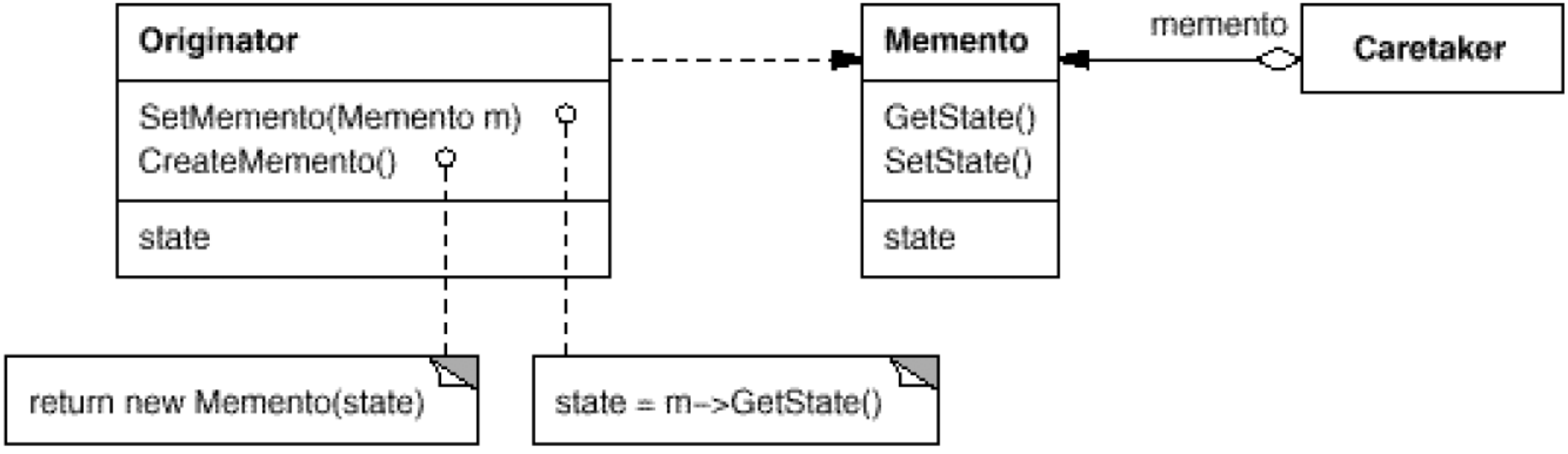
Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

- the internal state of an object must be saved and restored at a later time
- internal state cannot be exposed by interfaces without exposing implementation
- encapsulation boundaries must be preserved

Author: Rafał Roppel

MEMENTO

diagram



Author: Rafał Roppel

MEMENTO

examples in JDK

- `java.util.Date`
- `java.io.Serializable`

Author: Rafał Roppel

OBSERVER

Author: Rafał Roppel

OBSERVER

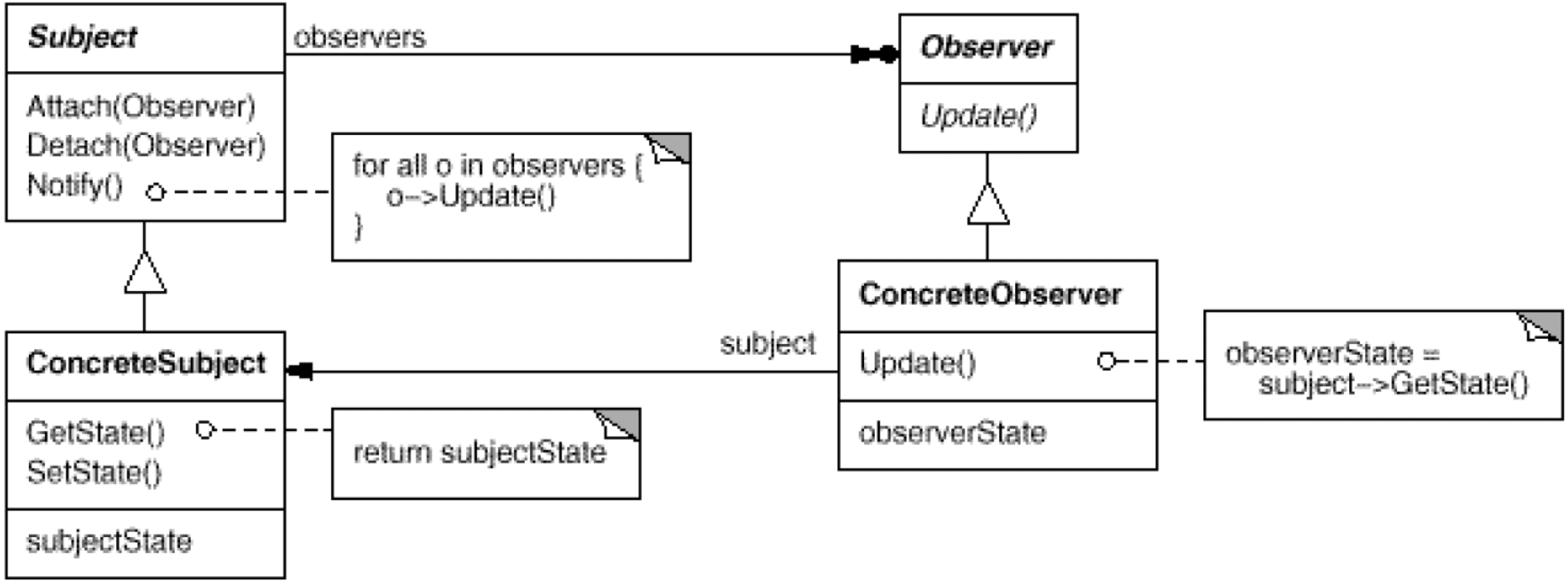
Lets one or more objects be notified of state changes in other objects within the system.

- state changes in one or more objects should trigger behavior in other objects
- broadcasting capabilities are required
- an understanding exists that objects will be blind to the expense of notification

Author: Rafał Roppel

OBSERVER

diagram



Author: Rafał Roppel

OBSERVER

examples in JDK

- `java.util.Observer` / `java.util.Observable`
- All implementations of `java.util.EventListener`
- `javax.servlet.http.HttpSessionBindingListe`
- `javax.servlet.http.HttpSessionAttributeLis`
- `javax.faces.event.PhaseListener`

Author: Rafał Roppel

STATE

Author: Rafał Roppel

STATE

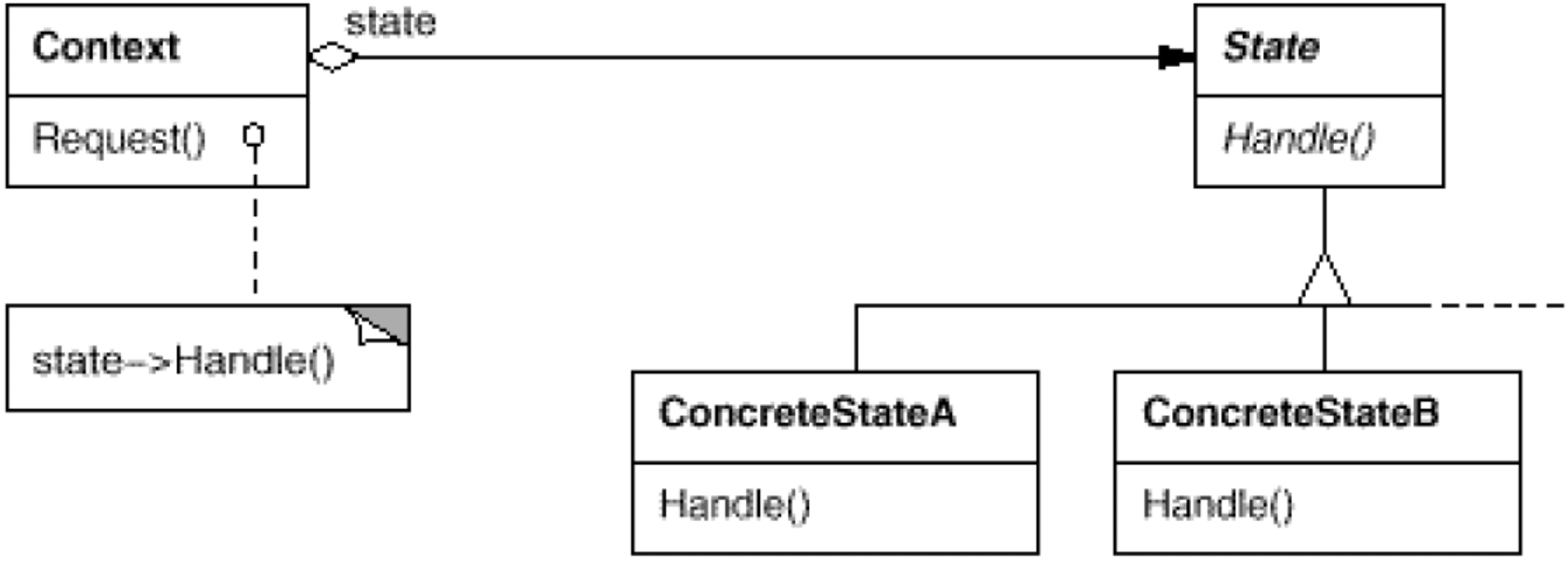
Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

- the behavior of an object should be influenced by its state
- complex conditions tie object behavior to its state
- transitions between states need to be explicit

Author: Rafał Roppel

STATE

diagram



Author: Rafał Roppel

STATE

examples in JDK

- `javax.faces.lifecycle.LifeCycle#execu`

STRATEGY

Author: Rafał Roppel

STRATEGY

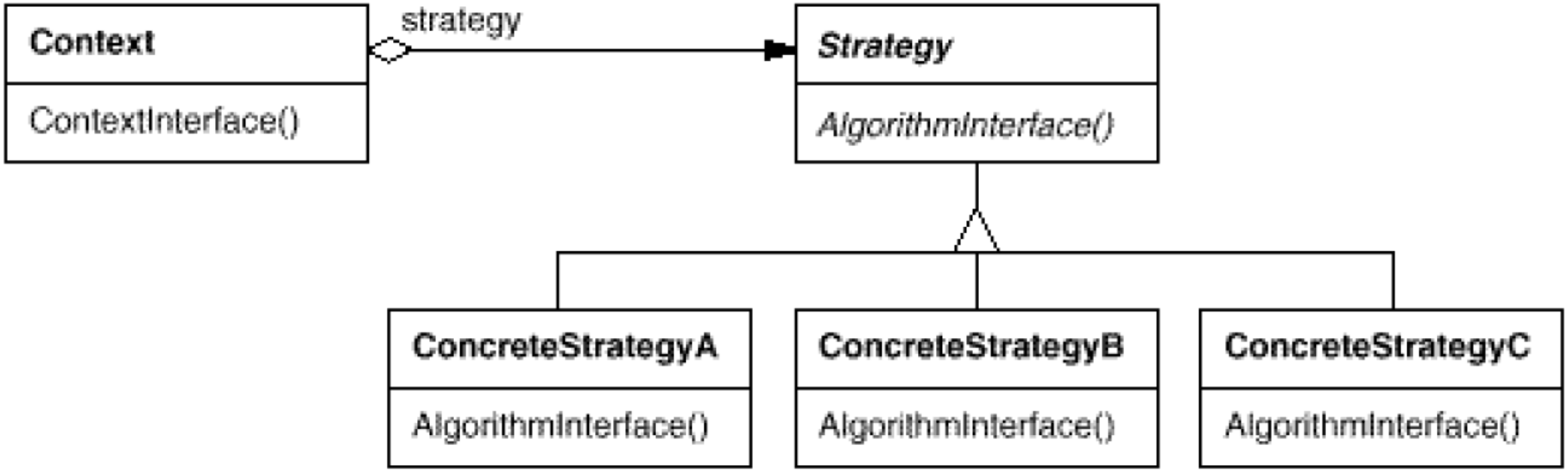
Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

- the only difference between many related classes is their behavior
- multiple versions or variations of an algorithm are required
- algorithms access or utilize data that calling code shouldn’t be exposed to
- the behavior of a class should be defined at runtime
- conditional statements are complex and hard to maintain

Author: Rafał Roppel

STRATEGY

diagram



Author: Rafał Roppel

STRATEGY

examples in JDK

- `java.util.Comparator#compare()`
- `javax.servlet.http.HttpServlet`
- `javax.servlet.Filter#doFilter()`

STRATEGY

DEMO

Author: Rafał Roppel

TEMPLATE METHOD

Author: Rafał Roppel

TEMPLATE METHOD

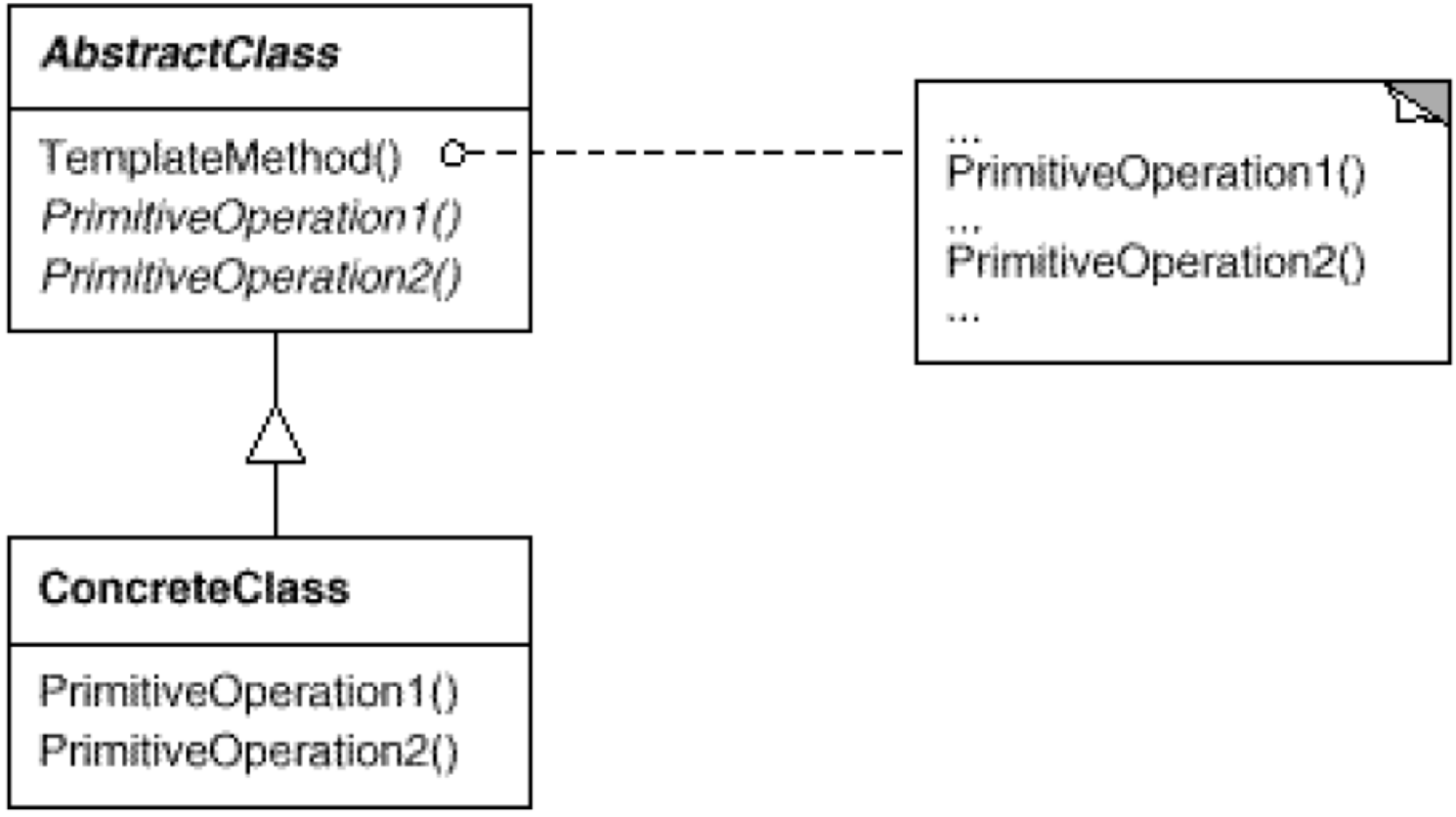
Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

- a single abstract implementation of an algorithm is needed
- common behavior among subclasses should be localized to a common class
- parent classes should be able to uniformly invoke behavior in their subclasses
- most or all subclasses need to implement the behavior

Author: Rafał Roppel

TEMPLATE METHOD

diagram



Author: Rafał Roppel

TEMPLATE METHOD

examples in JDK

- `java.util.Collections#sort()`
- `java.io.InputStream#skip()`
- `java.io.InputStream#read()`
- `java.util.AbstractList#indexOf()`

Author: Rafał Roppel

TEMPLATE METHOD

DEMO

Author: Rafał Roppel

VISITOR

Author: Rafał Roppel

VISITOR

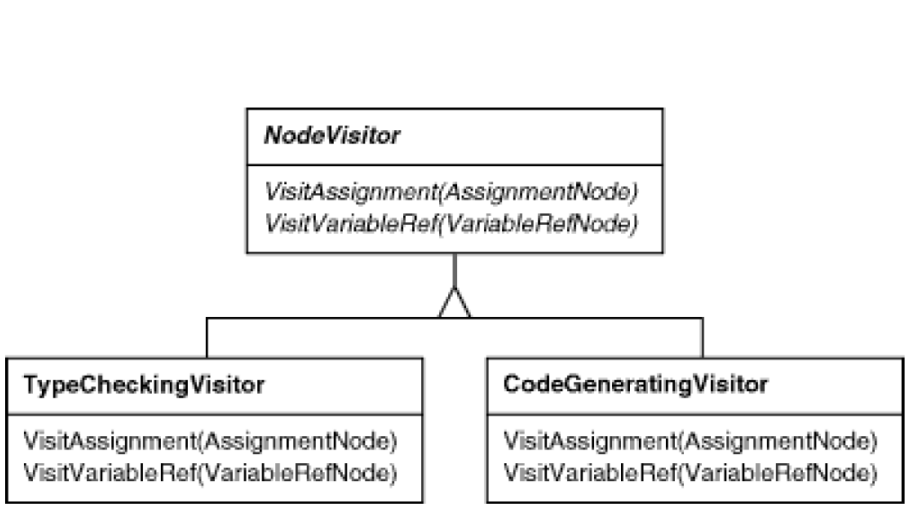
Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

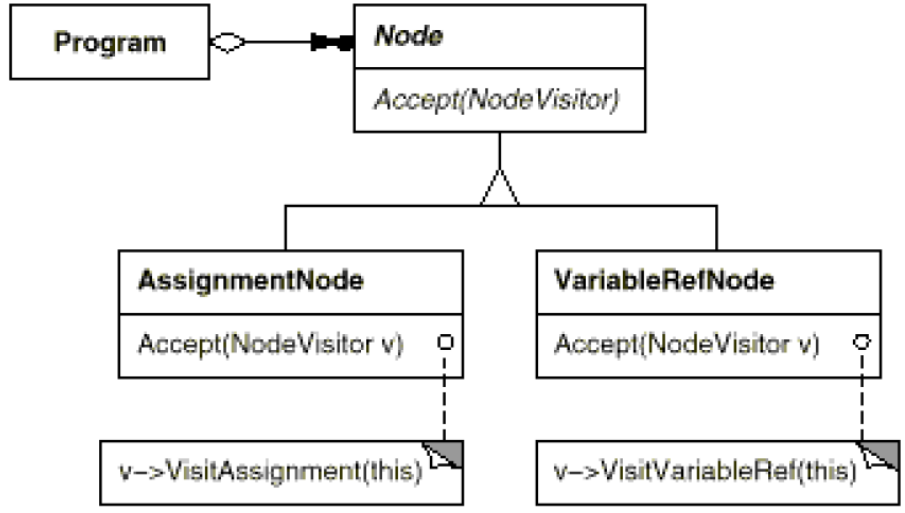
- an object structure must have many unrelated operations performedupon it
- the object structure can’t change but operations performed on it can
- operations must be performed on the concrete classes of an object structure
- exposing internal state or operations of the object structure is acceptable
- operations should be able to operate on multiple object structures that implement the same interface sets

Author: Rafał Roppel

VISITOR

diagram





Author: Rafał Roppel

VISITOR

examples in JDK

- `javax.lang.model.element.Element` and `javax.lang.model.element.ElementVisit`
- `javax.lang.model.type.TypeMirror` and `javax.lang.model.type.TypeVisitor`

Author: Rafał Roppel

VISITOR

DEMO

Author: Rafał Roppel

FURTHER READING

- Head First Design Patterns
- Design patterns: elements of reusable object-oriented software
- Refactoring: Improving the Design of Existing Code
- Refactoring to Patterns
- Domain-Driven Design Distilled

Author: Rafał Roppel