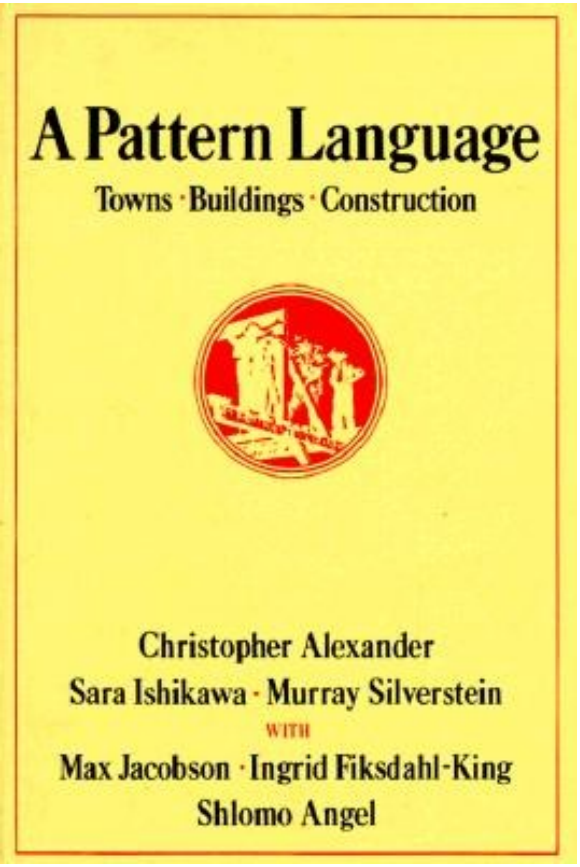
Design Patterns in OOP

Following the “gang of four” (GoF)
Gamma, Helm, Johnson, Vlissides, *Design Patterns*,
Addison-Wesley 1995

Why Design Patterns?

- Apply well known and proven solutions
  - many problems are not new → no need to invent wheels
  - code structure easier to understand → easier maintenance
  - great help for beginners to learn good practice
  - patterns are not static, guide to individual solutions
- Analogies
  - song styles, theater pieces, novels, (architecture), engineering, ...

History



C. Alexander (1936-2022), computer scientist and architect

Critical of traditional modern architecture, patterns as solution guides in architecture, incremental building, interaction with users, empower laypeople to create designs

Medieval cities built according to rules, not rigid masterplans

Pattern Classification

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

Not all patterns covered* here, many more exist

Patterns and OOP

- Design patterns help to translate OOP design principles
  - dependency management
  - components
  - code reuse
  - ease of planned (and unplanned) changes
  - maintenance
  - code quality

Structured pattern description

- Pattern name
  - one- or two-word descriptive title
- Intent
  - what happens? Why? Design issue or problem?
- Motivation
  - example pattern application scenario
- Applicability
  - when to use? What problems solved?
- Structure
  - UML graphical description

Structured pattern description

- Participants and Collaborations
  - classes, objects, their roles and collaborations
- Consequences and Implementation
  - results and trade-offs, implementation tricks
- Examples
  - code, projects
- Related patterns
  - relation to other patterns, combined uses

Creational Patterns

- Organise object creation
- Class creational patterns
  - Factory Method
  - defer (part of) object creation to subclasses
- Object creational patterns
  - Abstract Factory
  - Singleton
  - defer (part of) object creation to other objects

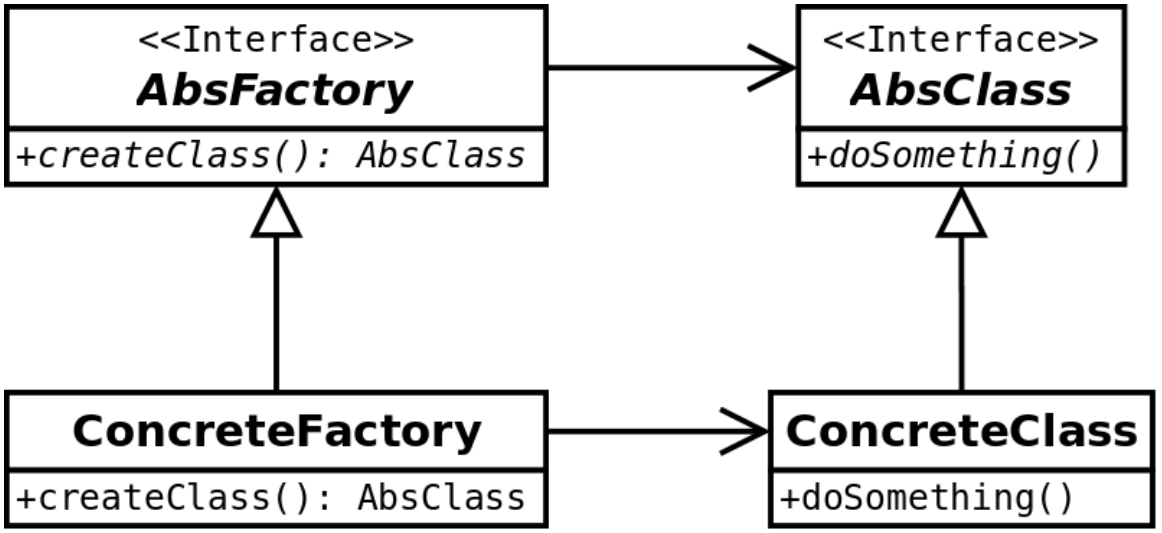
(Abstract) Factory Method

Create objects without dependence on concrete classes

Isolate concrete classes from higher levels, createClass() is Factory Method, AbsFactory is Abstract Factory

Easy to replace functionalities

Hard to change class structure



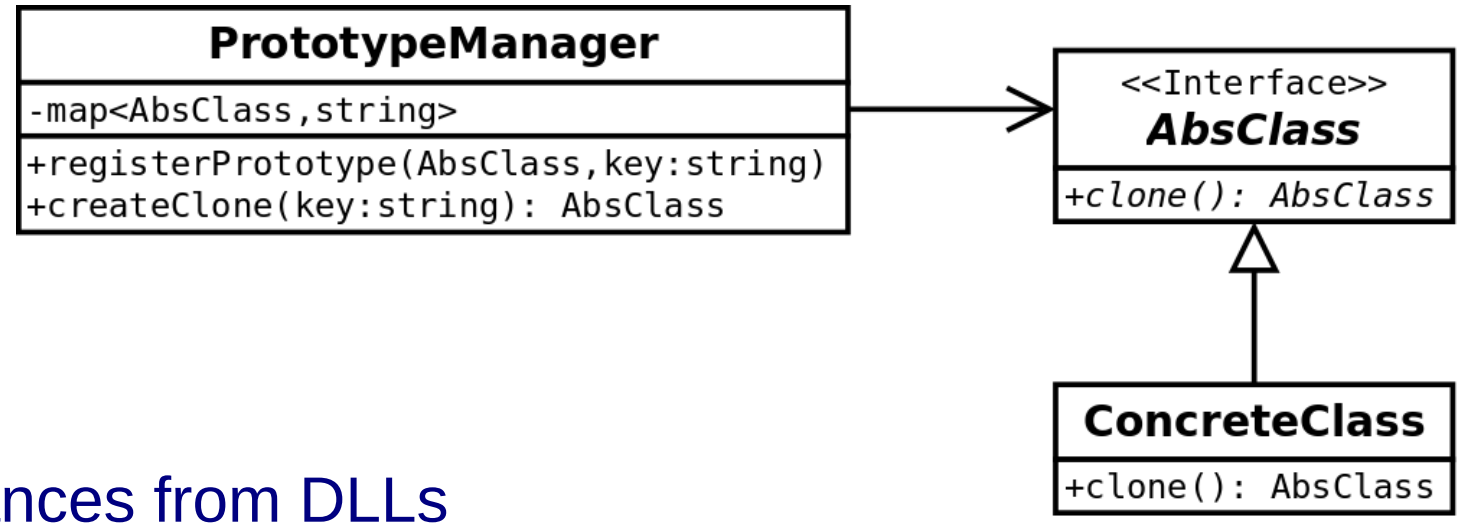
GUIs on different platforms, plug-ins

Alternative: Prototype

Prototype

Create new objects from a prototype through an interface to avoid dependency on concrete classes

Isolate concrete classes from higher level



Avoid hierarchy of factories

Easy to get instances from DLLs

Classes must support cloning, must decide shallow or deep copy, take care of initialisation

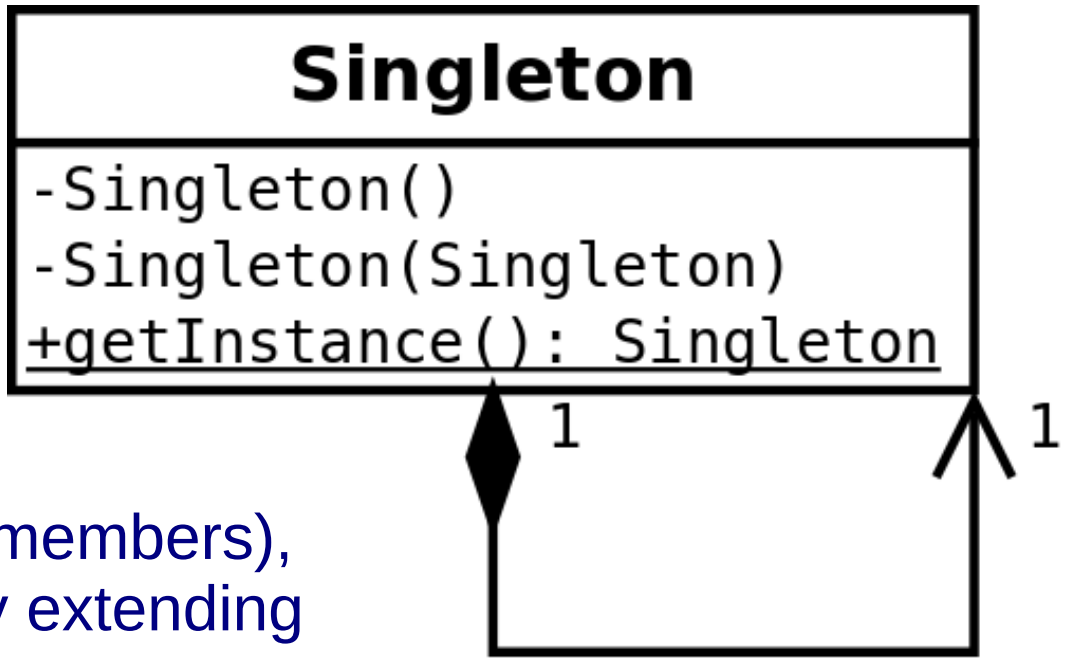
Alternative: (Abstract) Factory method

Singleton

Guarantee that there is only one instance of a class

Avoid confusion over central objects

Private constructors, static method to return handle to single static instance



Can be subclassed (vs. static members), control number of instances by extending getInstance

Used in more complex patterns

Structural Patterns

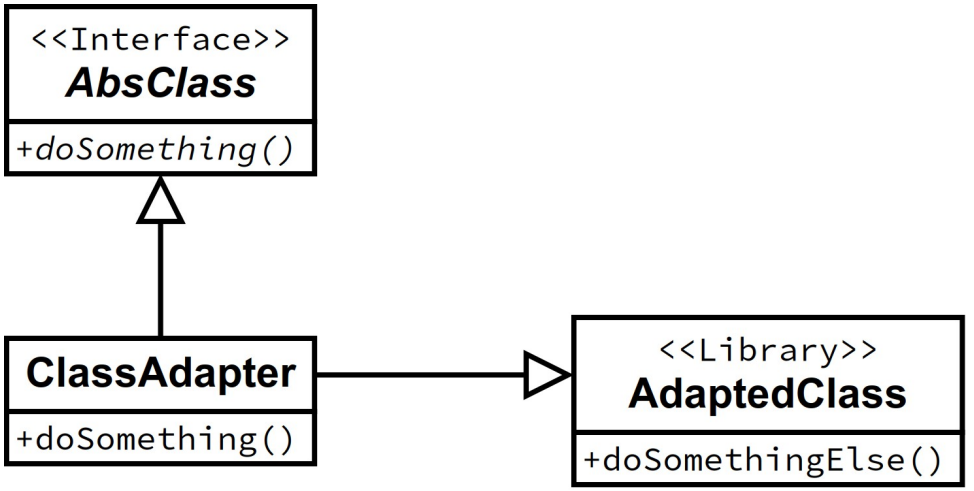
- Compose complex structures from small ones
- Class structural patterns
  - Compose interfaces or implementations using class inheritance
  - Adapter
- Object structural patterns
  - Compose objects to get new functionality, possibly at run-time
  - Adapter, Composite, Decorator, Proxy

Adapter

Convert (adapt) the interface of a class to interface expected by clients

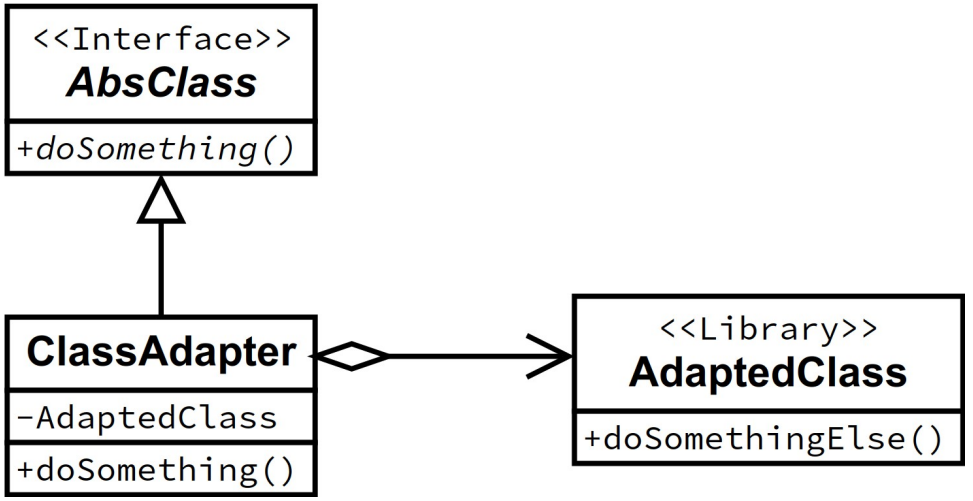
Use existing class (libraries)

Class adapter: mult. inheritance, implement request using AdaptedClass methods



Object adapter: hold reference, forward or translate requests

Decorator, Proxy (no interface changes)

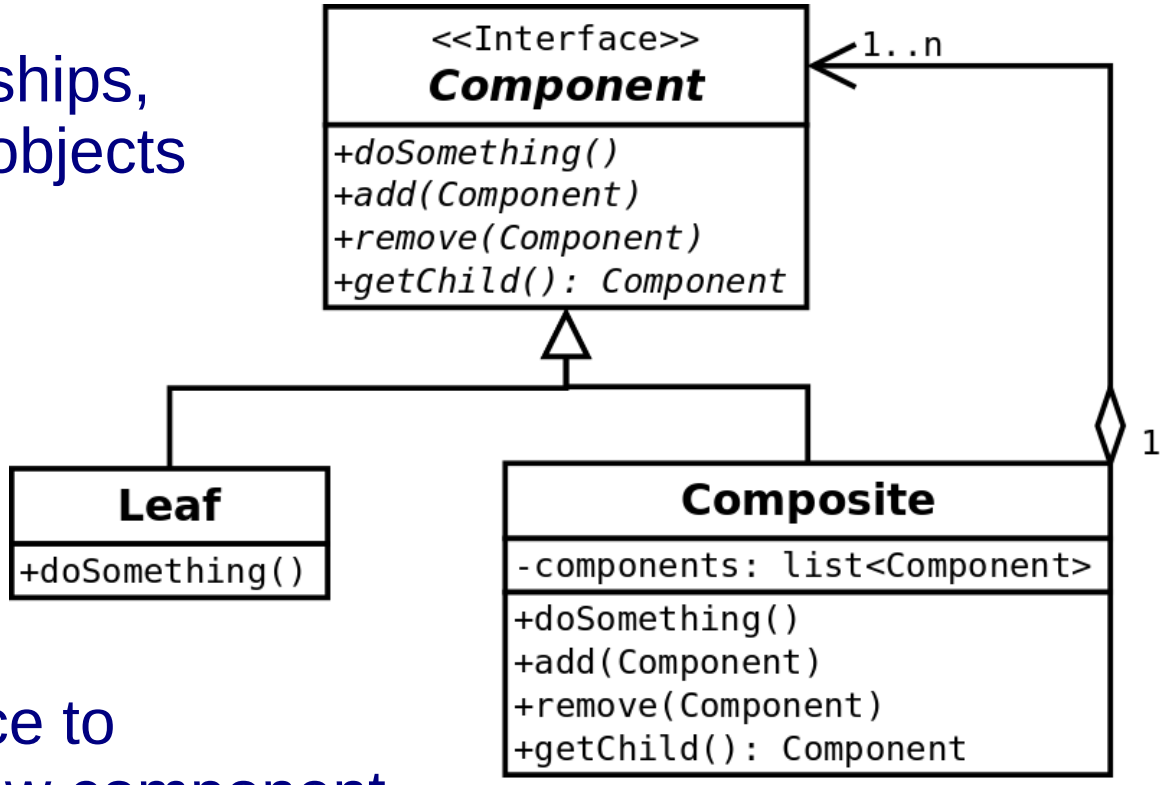


Composite

Compose object recursively into tree-like structures

Represent whole-part relationships, handle objects and groups of objects uniformly

Composite can contain simple objects (Leaf) or composites



Clients can compose complex objects, but don't see difference to simple objects, easy to add new component types

Decorator, CoR, Iterator, Visitor can collaborate

Decorator

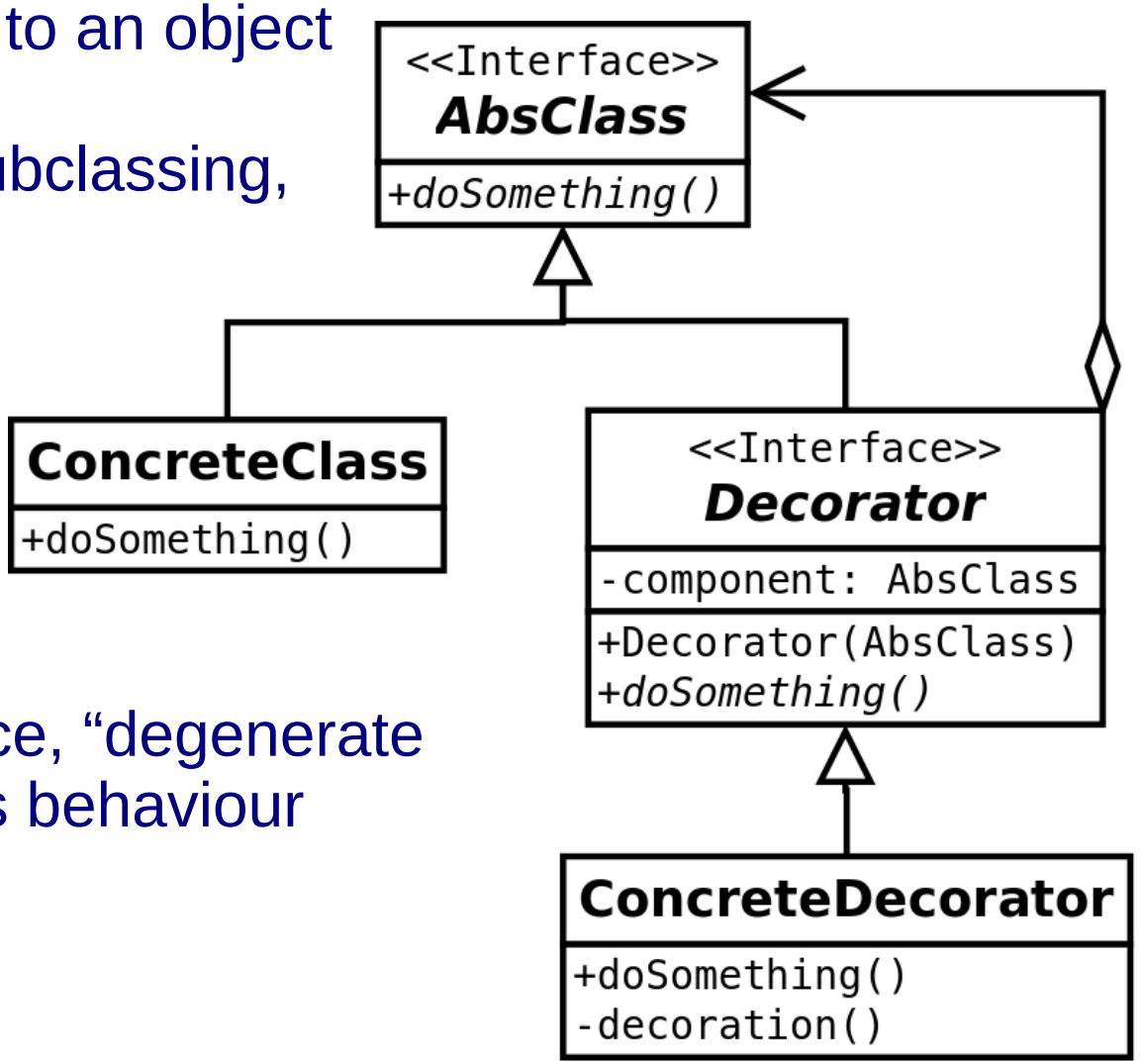
Add functionality dynamically to an object

Alternative to direct (static) subclassing, fight “combinatorics”

Decorator forwards requests to component

GUI toolkits, ...

Adapter also changes interface, “degenerate composite”, Strategy modifies behaviour

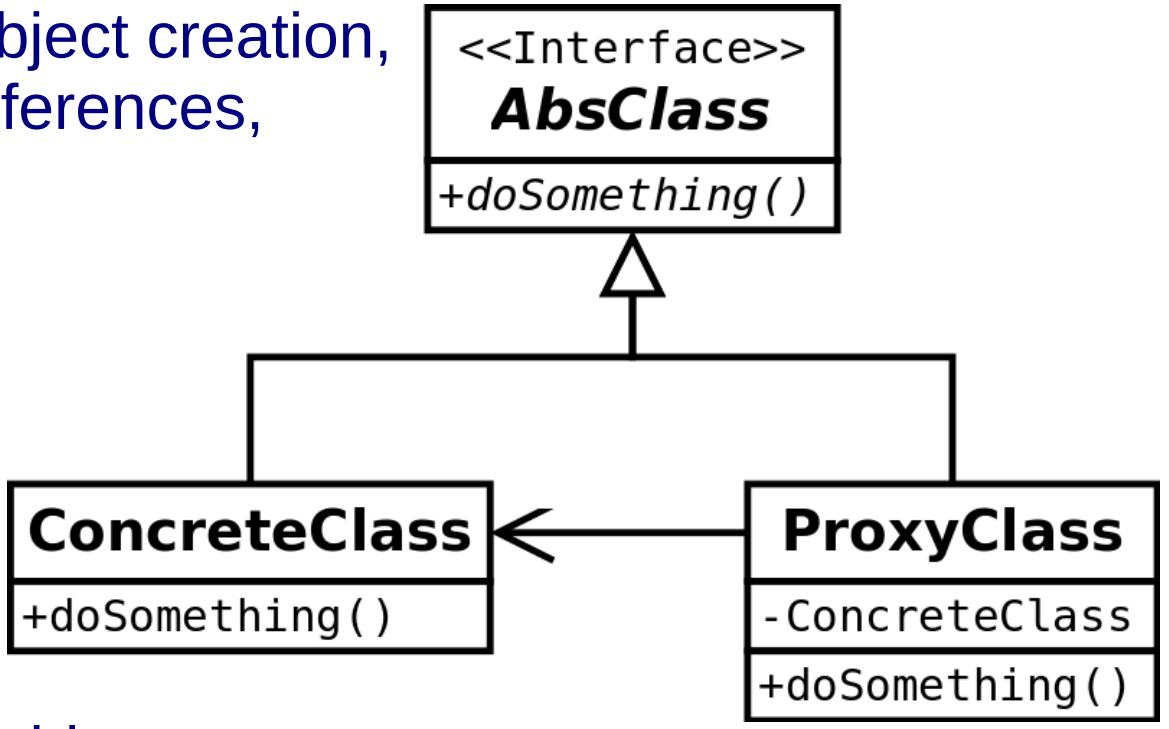


Proxy

Provide placeholder for another object to control access

Support “lazy” operations (object creation, IO) and/or caching, smart references, “copy-on-write”

Client sees only ProxyClass objects, requests forwarded to ConcreteClass objects



Helps handling “expensive” objects

Proxy provides access control, Decorator or Adapter modify behavior or interface

Behavioral Patterns

- Implement algorithms
- Class behavioral patterns
  - use inheritance to separate algorithm invariants from algorithm variants
  - Template Method
- Object behavioral patterns
  - use object composition to distribute algorithm parts (invariants, variants)
  - Chain of Responsibility, Iterator, State, Observer, Strategy

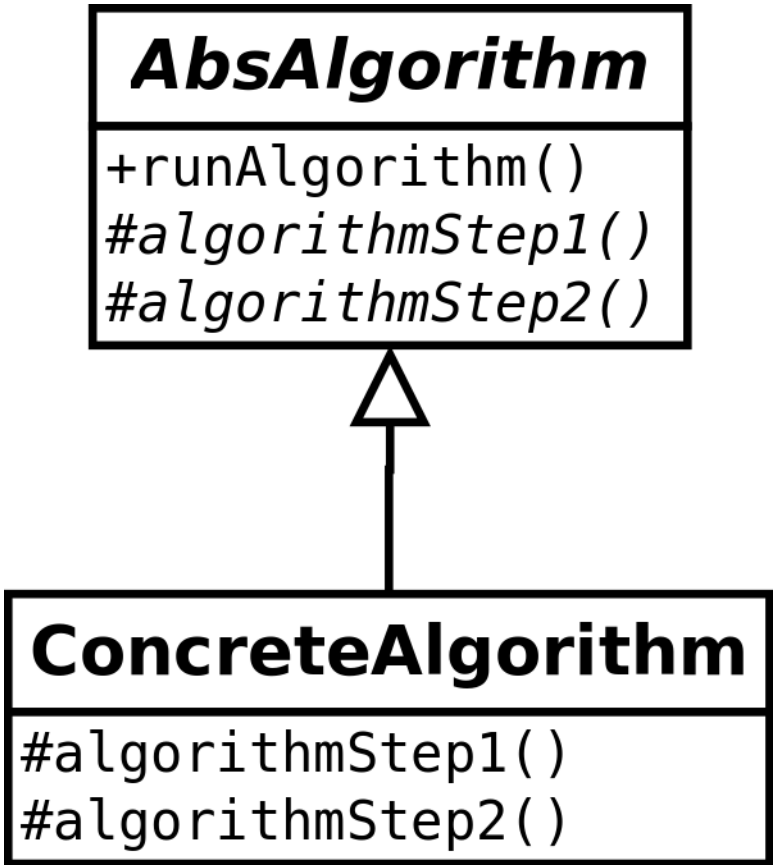
Template Method

Define invariant algorithm skeleton and defer variant steps to methods in subclasses

Algorithm family implementation, localise common behavior of classes

Dependency inversion from concrete to abstract → class libraries

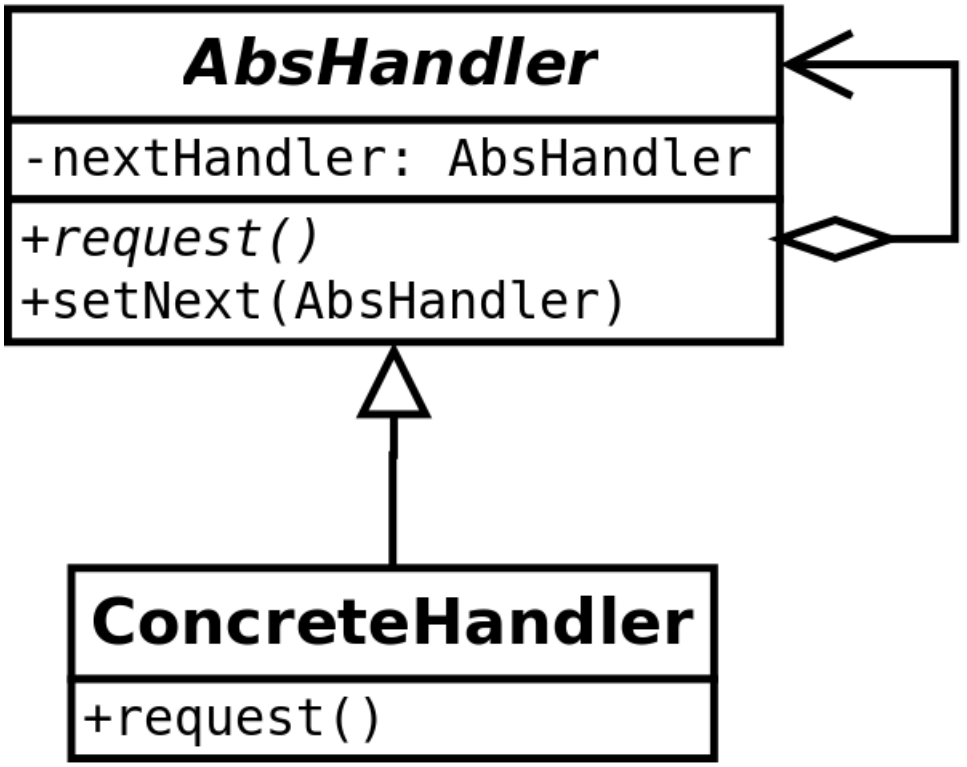
Factory Methods providing objects with algorithm steps often used in Template Method, Strategy gives algorithm variants at object level



Chain of Responsibility (CoR)

Allow several objects to handle a request by chaining them and passing the request along the chain, objects handle the request or pass it to the next object

In a dynamic system find correct object for a request

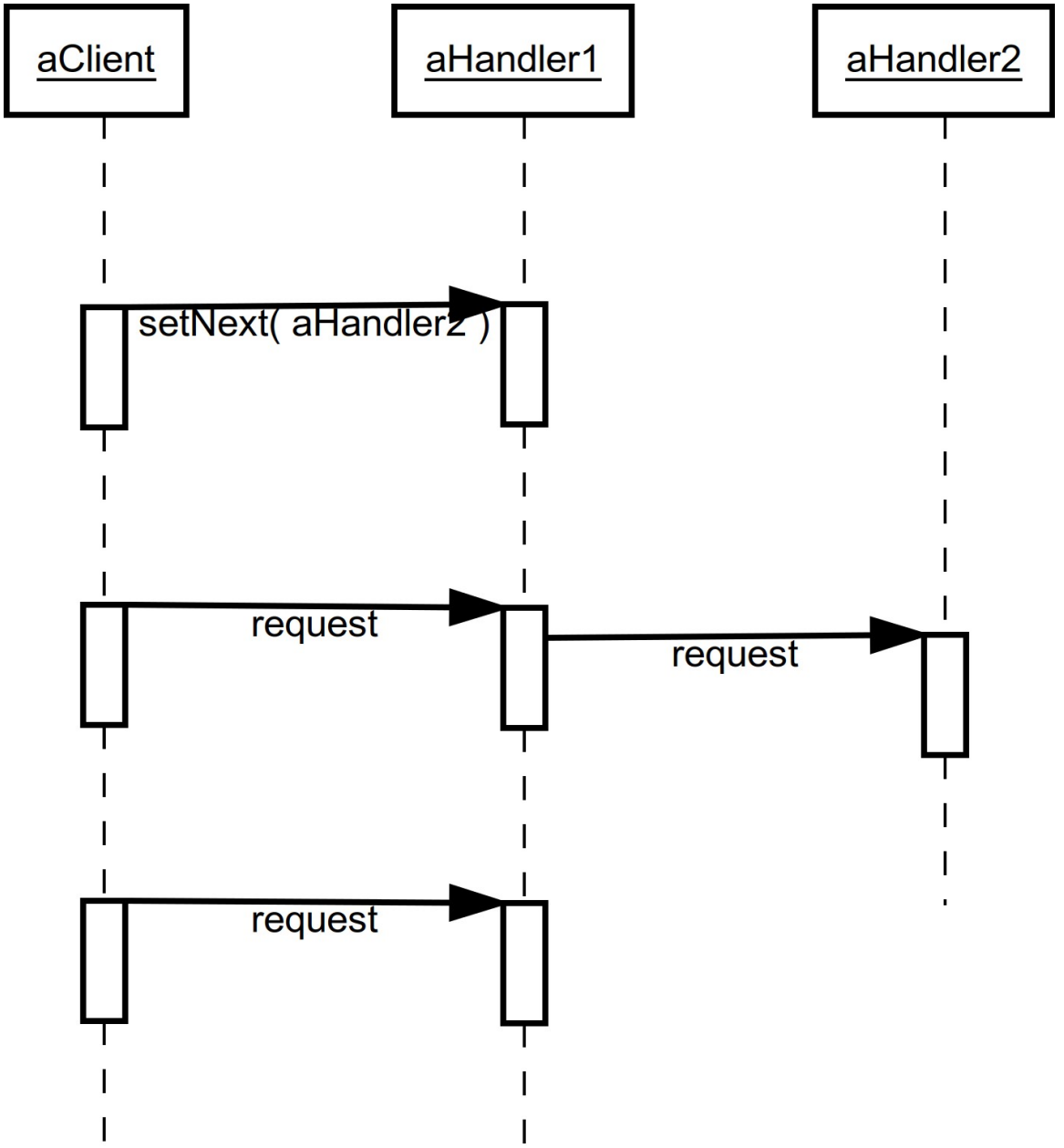


No direct connection between sender and receiver of request, can change request handling at run-time by reconfiguring the chain

Handle user events, collaboration with Composite where parent is next object, flexible procedures

Chain of Responsibility

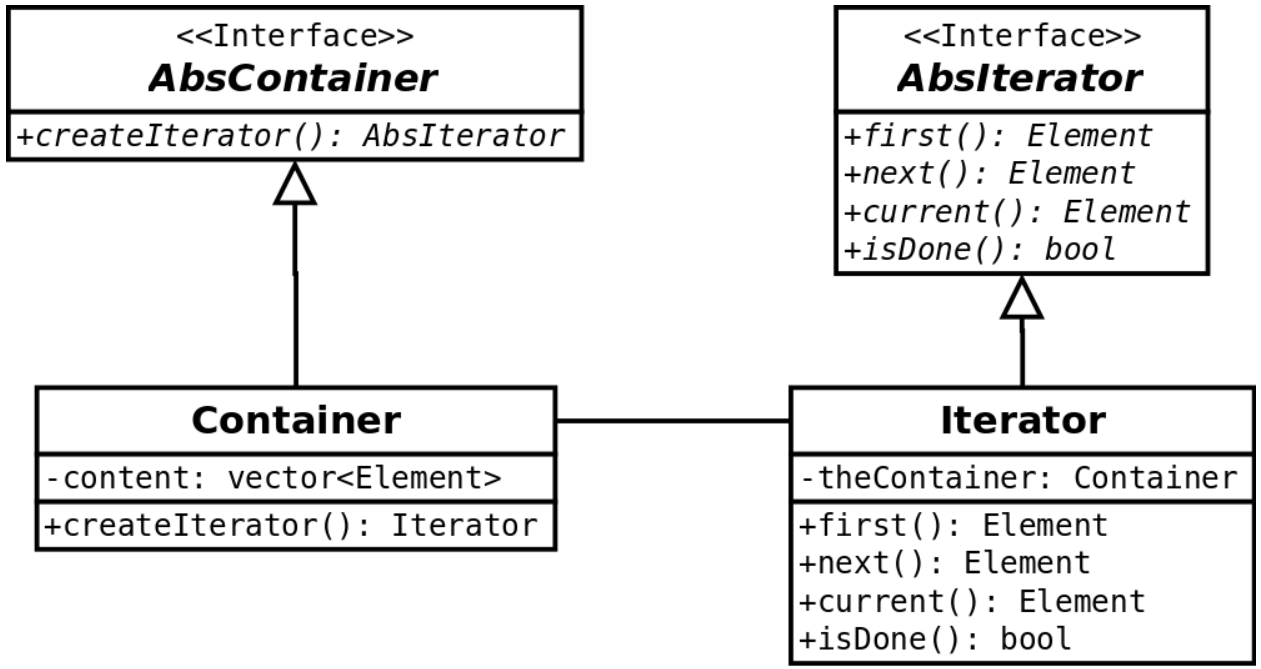
Object interaction diagram



Iterator

Access elements of a collection without exposing collection structure

Handle different collection structures, support heterogeneous collections, multiple traversals, different iteration algorithms



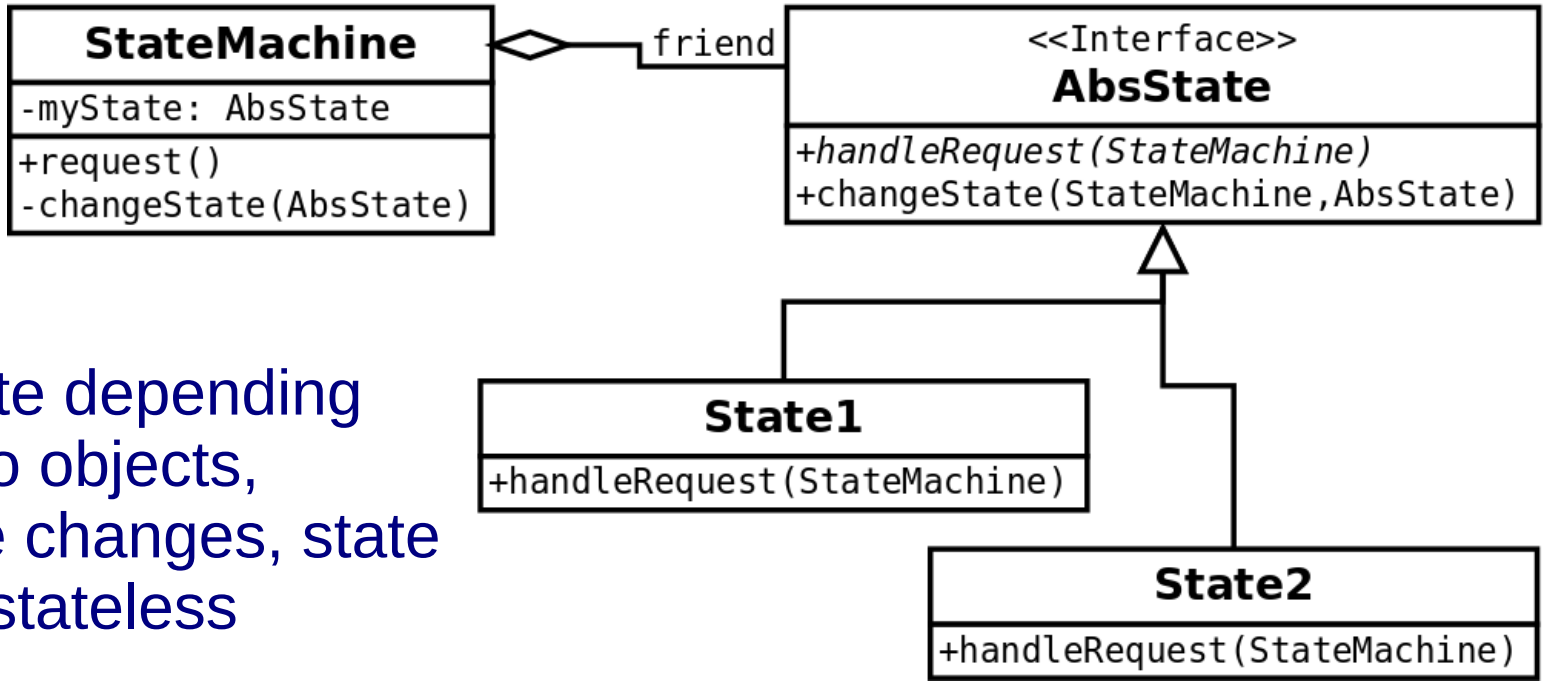
Container and Iterator tightly coupled, C++ with templates or interface+ RTTI for elements

Iterator over Composite structures, Factory Method to create Iterators

State

Allow object behavior change following state change

State machine modeling, refactoring of conditionals in methods depending on state

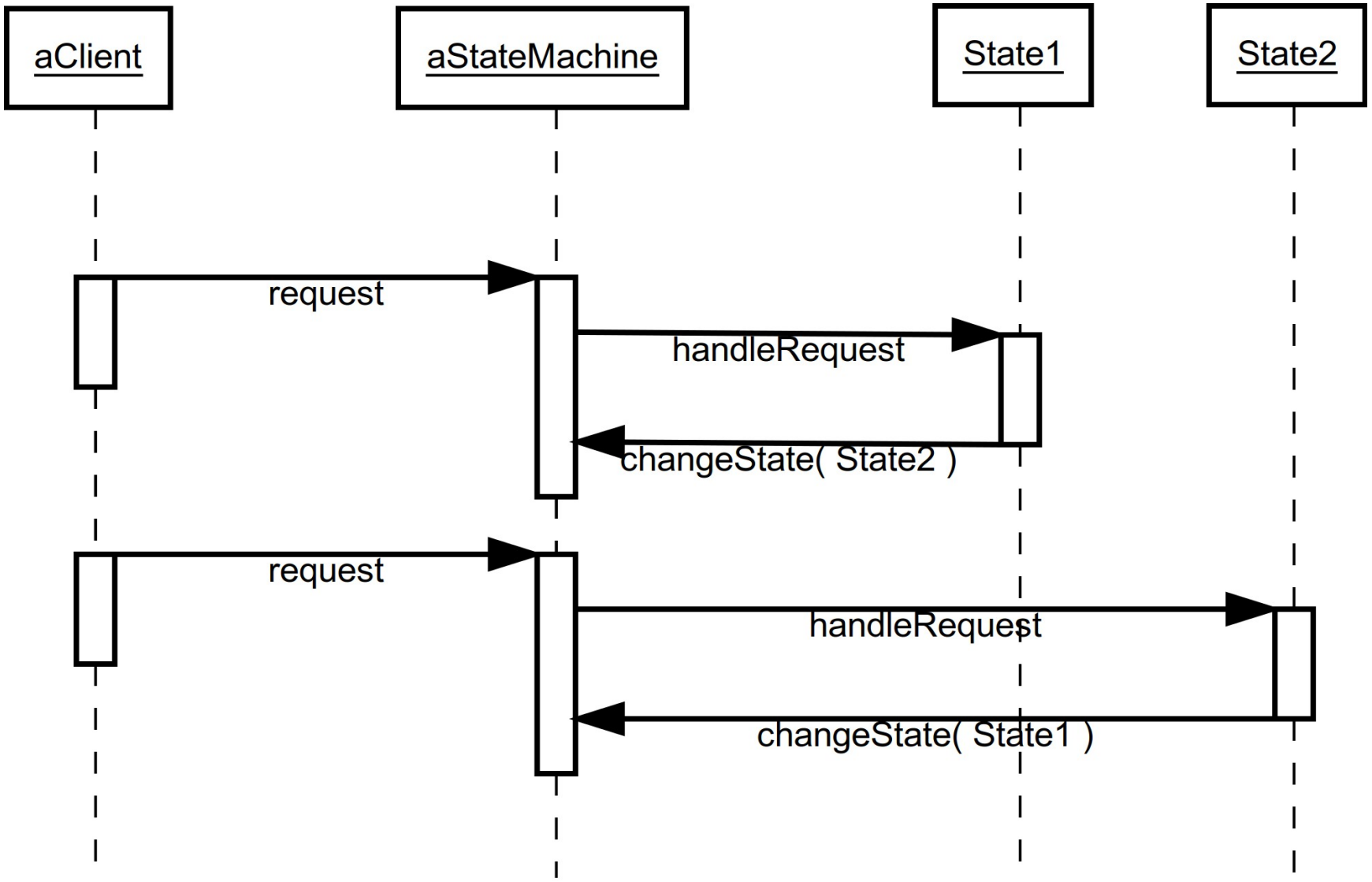


Localise state depending behavior into objects, explicit state changes, state objects are stateless

States can be Singletons

State

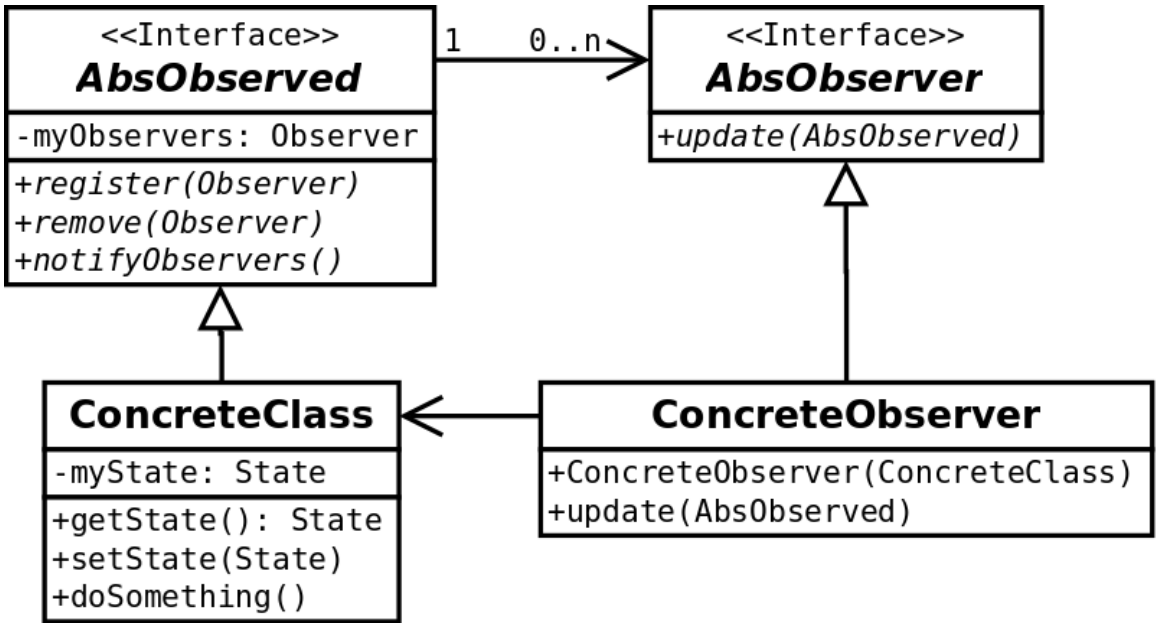
Object interaction diagram



Observer

Define one-to-many relation between objects to notify clients when target changes state

“Broadcast” messages avoiding tight coupling of objects



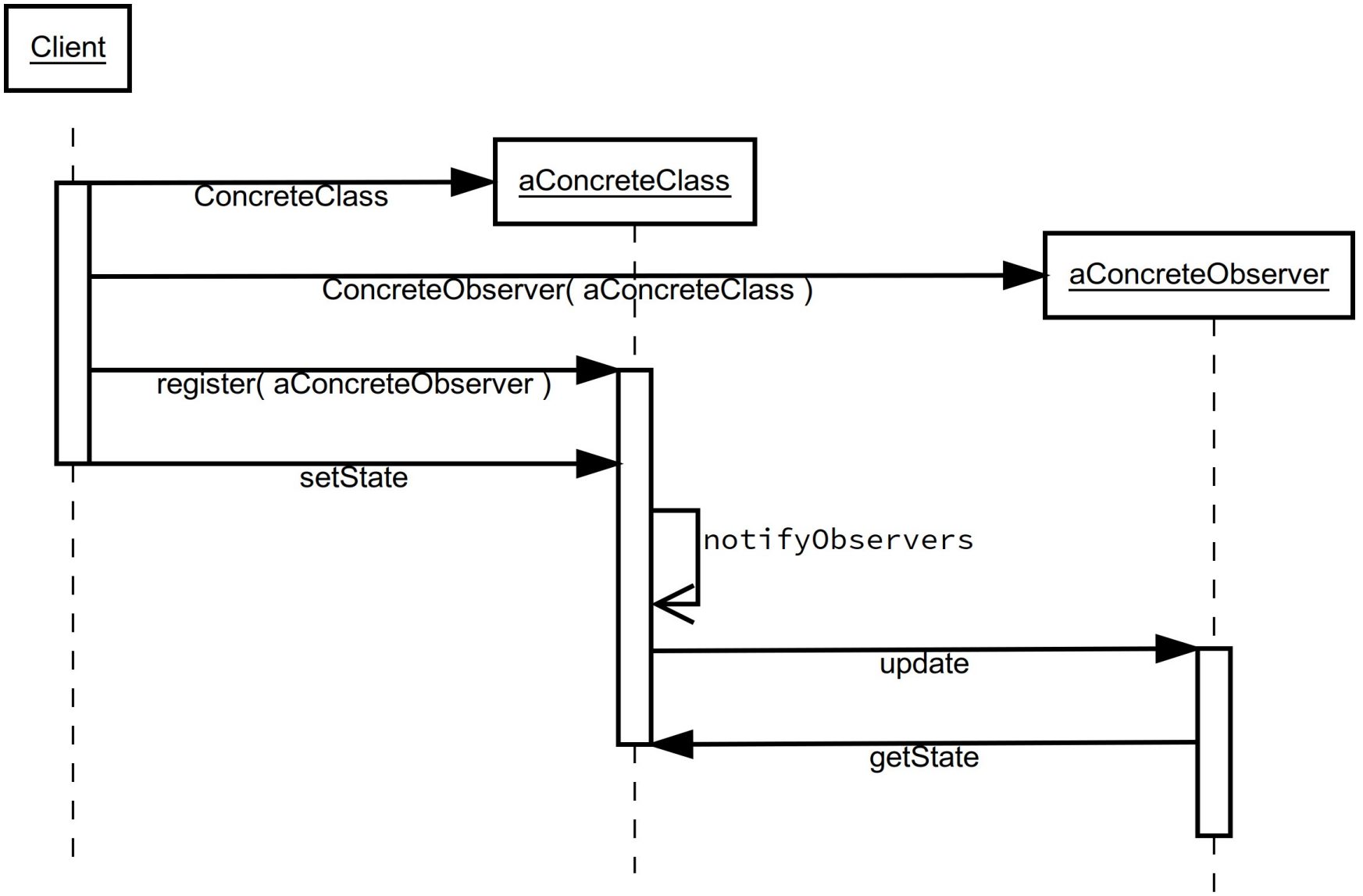
Updates to observers can be unexpected

Complex relation between observed and observer objects can be collected into a “ChangeManager” object

GUI objects observe drawable objects for redrawing

Observer

Object interaction diagram

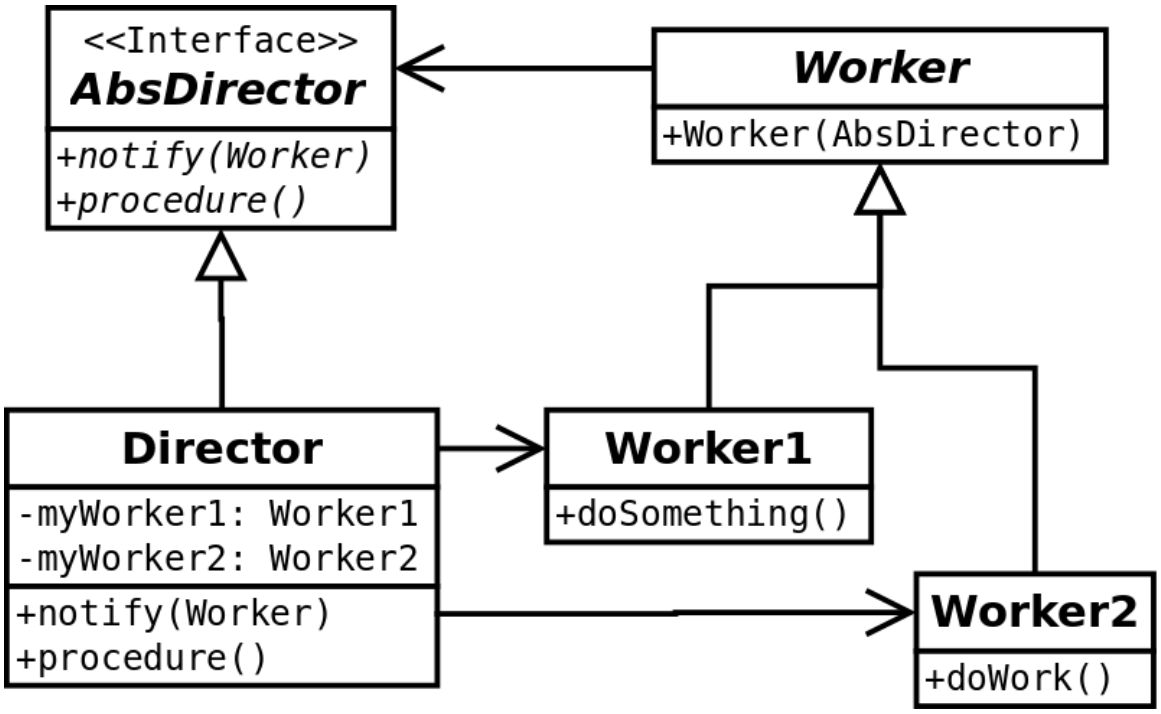


Mediator

Enclose object interactions in a central “controller” object

Complex but well defined communication between objects, use when objects have links to many other objects

Worker notifies Director with its address, Director identifies and decides next step

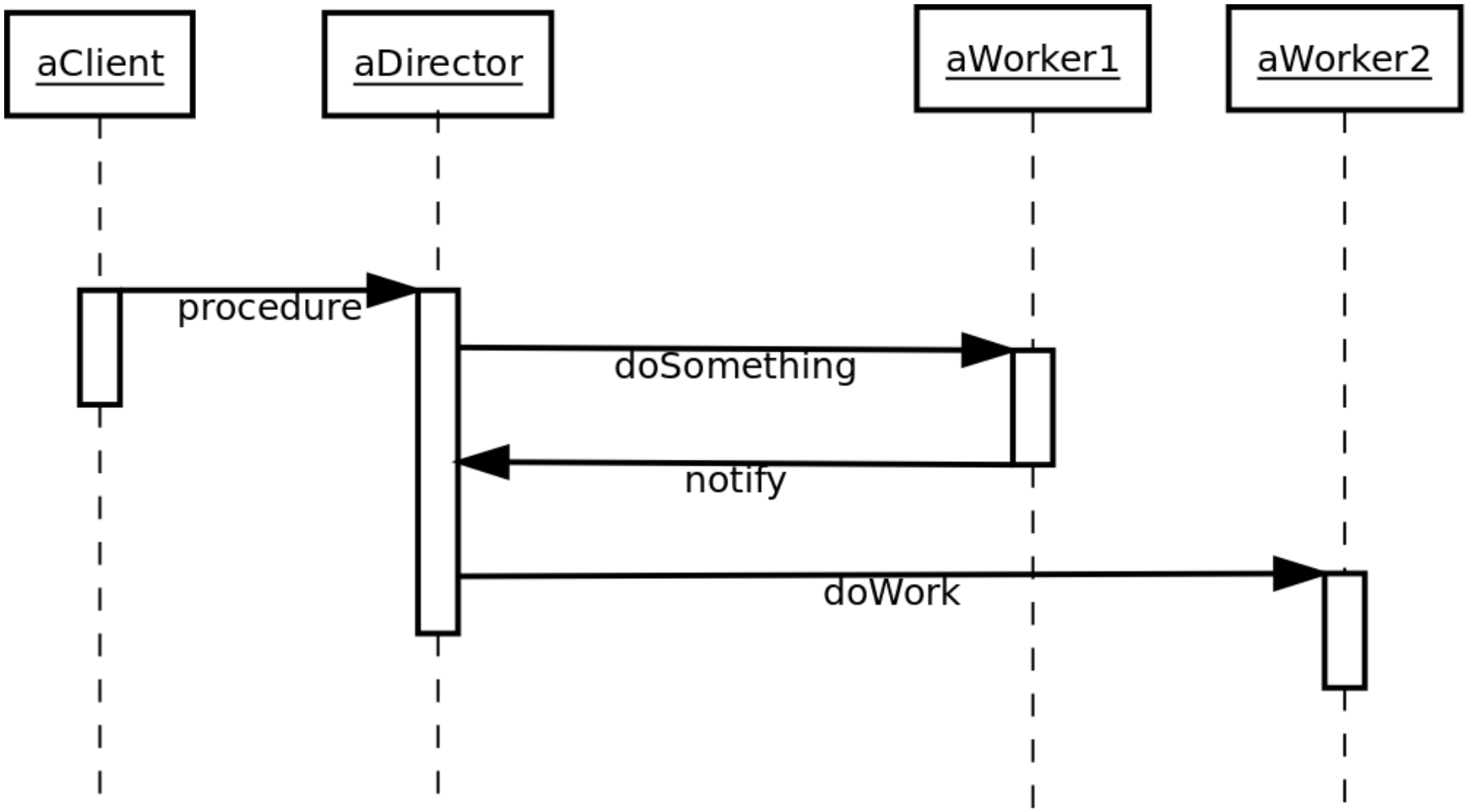


Decouple Workers, centralise control, can change protocol by subclassing Director

Director could be Observer of Workers

Mediator

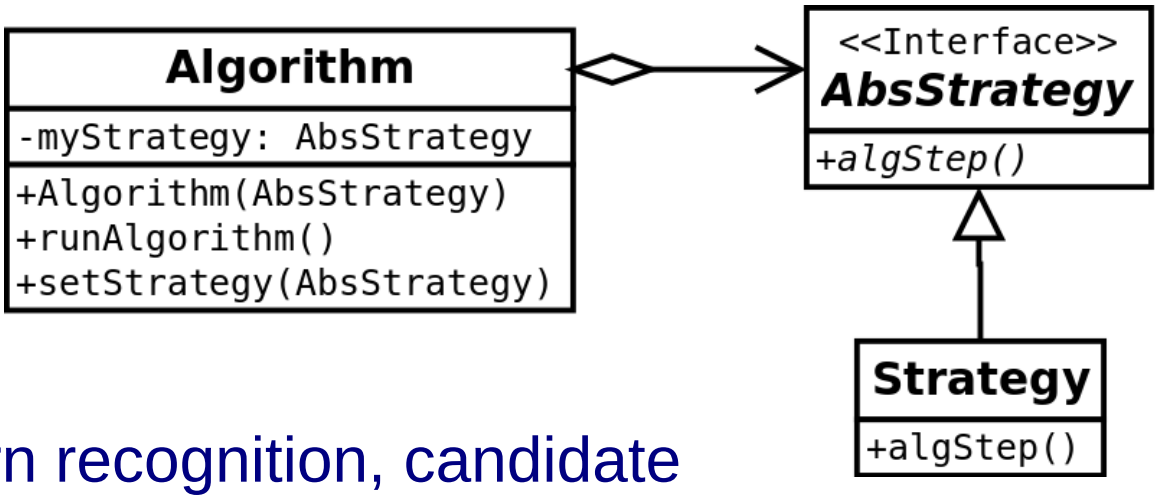
Object interaction diagram



Strategy

Define a family of algorithms interchangeable for clients

Make objects configurable for different behaviours, implement algorithm variants independent of invariants, hide details from clients
via Strategy class, remove conditionals from Algorithm, different implementations of same behaviour



Track finding algorithm (pattern recognition, candidate selection, track fit)

Summary and Discussion

- Creational
  - (Abstract) Factory Method vs Prototype
  - Only one object: Singleton
- Structural
  - Decorator: add behaviour
  - Composite: recursive object structures
  - Proxy: access control to other objects

Summary and Discussion

- Behavioral
  - Template and Strategy: algorithm (in-) variants
  - State: state-dependent behavior
  - Iterator: access to complex object collections
  - CoR: communication to varying number of objects
  - Observer vs Mediator: object communication (de-)centralised

Some HEP Patterns

- HEP offline programs have some special patterns
- Particular requirements
  - high throughput
  - variable algorithms
  - long lifetime of codes
  - programming interface for users

Transient/Persistent (Memento)

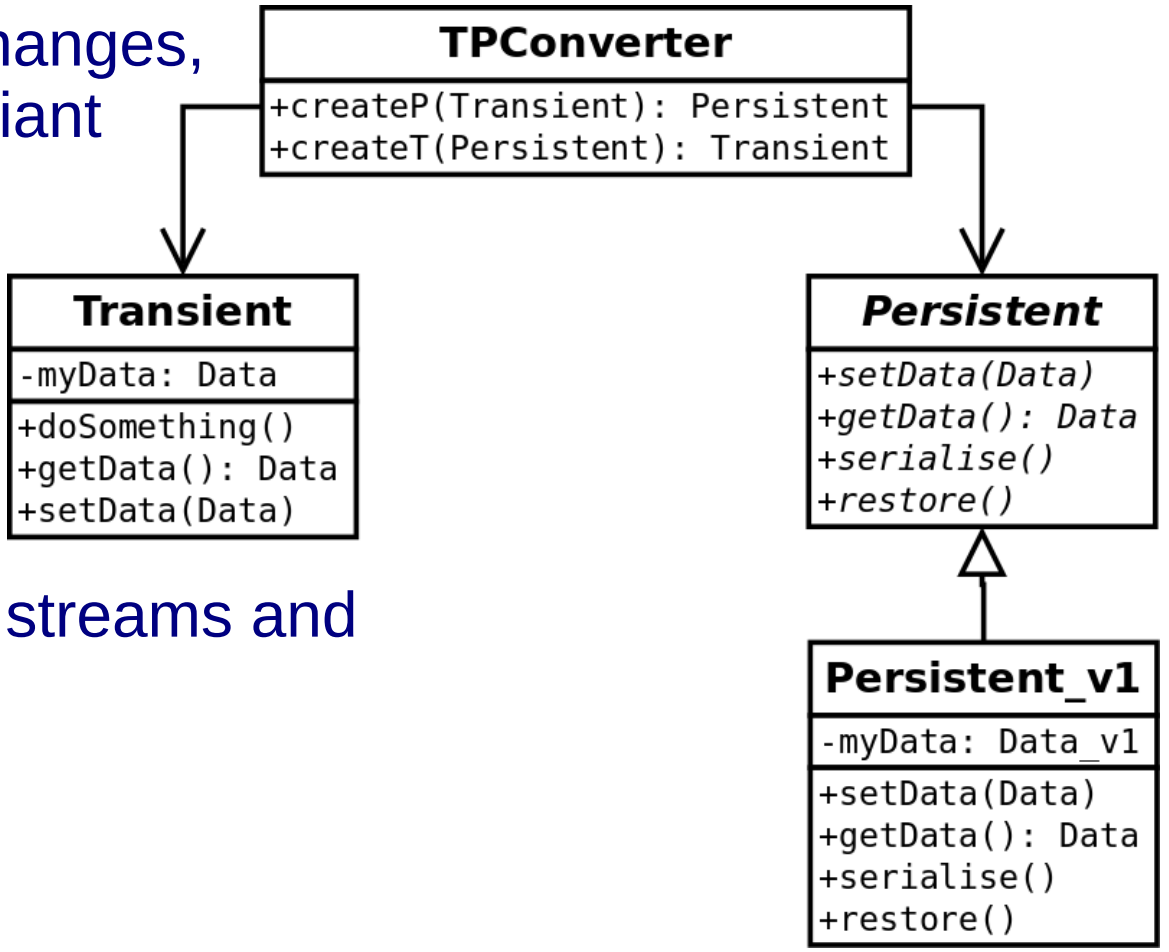
Decouple objects from the details of the storage system without violating data hiding

Storage systems subject to changes, keep other system parts invariant

Can replace storage system, Persistent and TPConverter

Memento w/o Converter

Use together with abstract IO streams and Blackboard

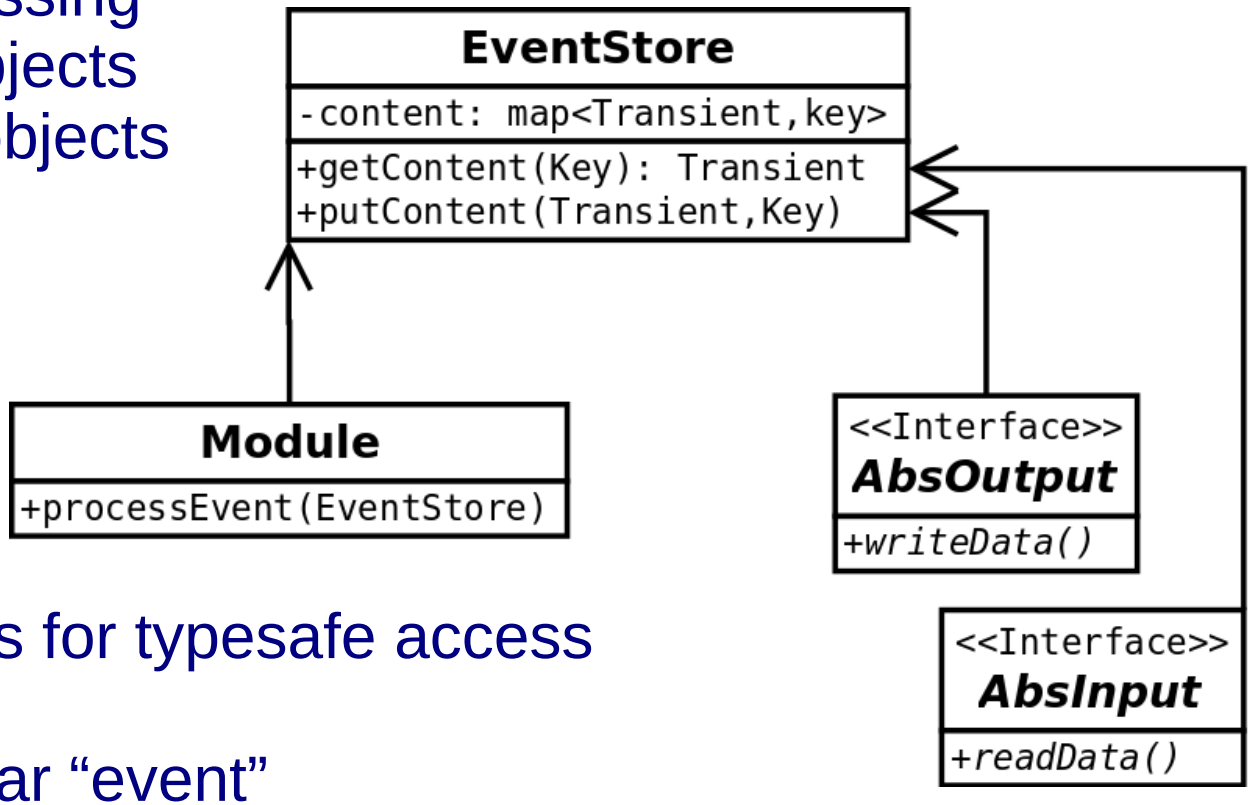


Blackboard

Model traditional HEP data processing with objects

EventStore is “COMMON BLOCK”
to hold event data, processing
Module gets Transient objects
and puts new Transient objects

AbsInput and AbsOutput
decouple the IO system
from the data processing



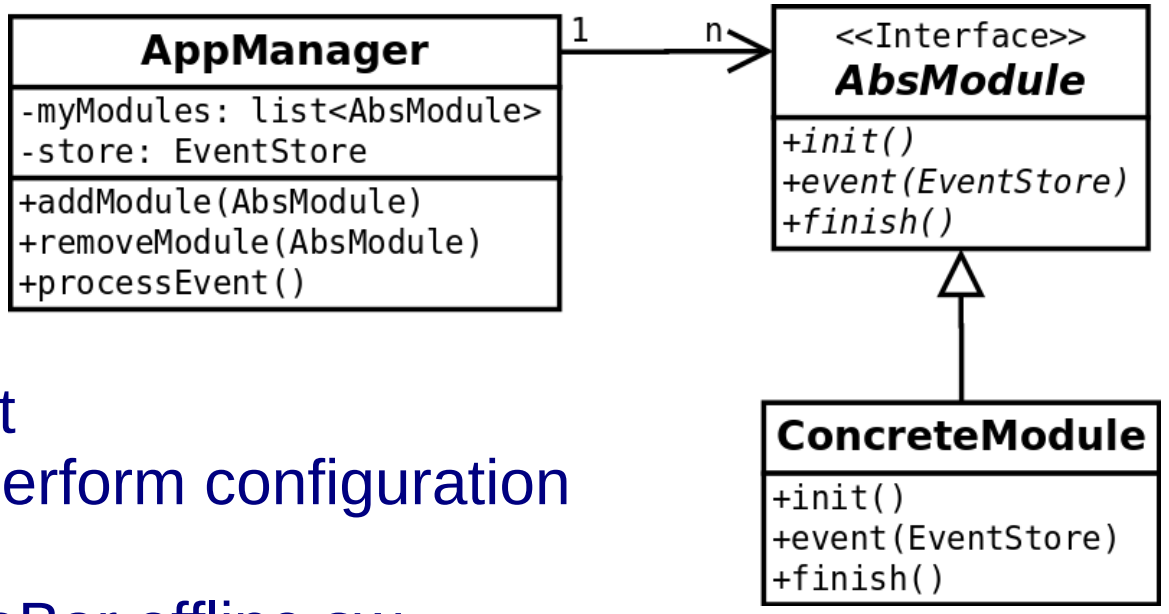
C++ use template classes for typesafe access

ATLAS “StoreGate”, BaBar “event”

Procedure

Setup for configurable procedures for event data processing

Establish framework for
Flexible data processing
procedures with stable
IO structure



Often combined with script
language (tcl, python) to perform configuration

ATLAS athena (Gaudi), BaBar offline sw, …

Mediator without callback to Director