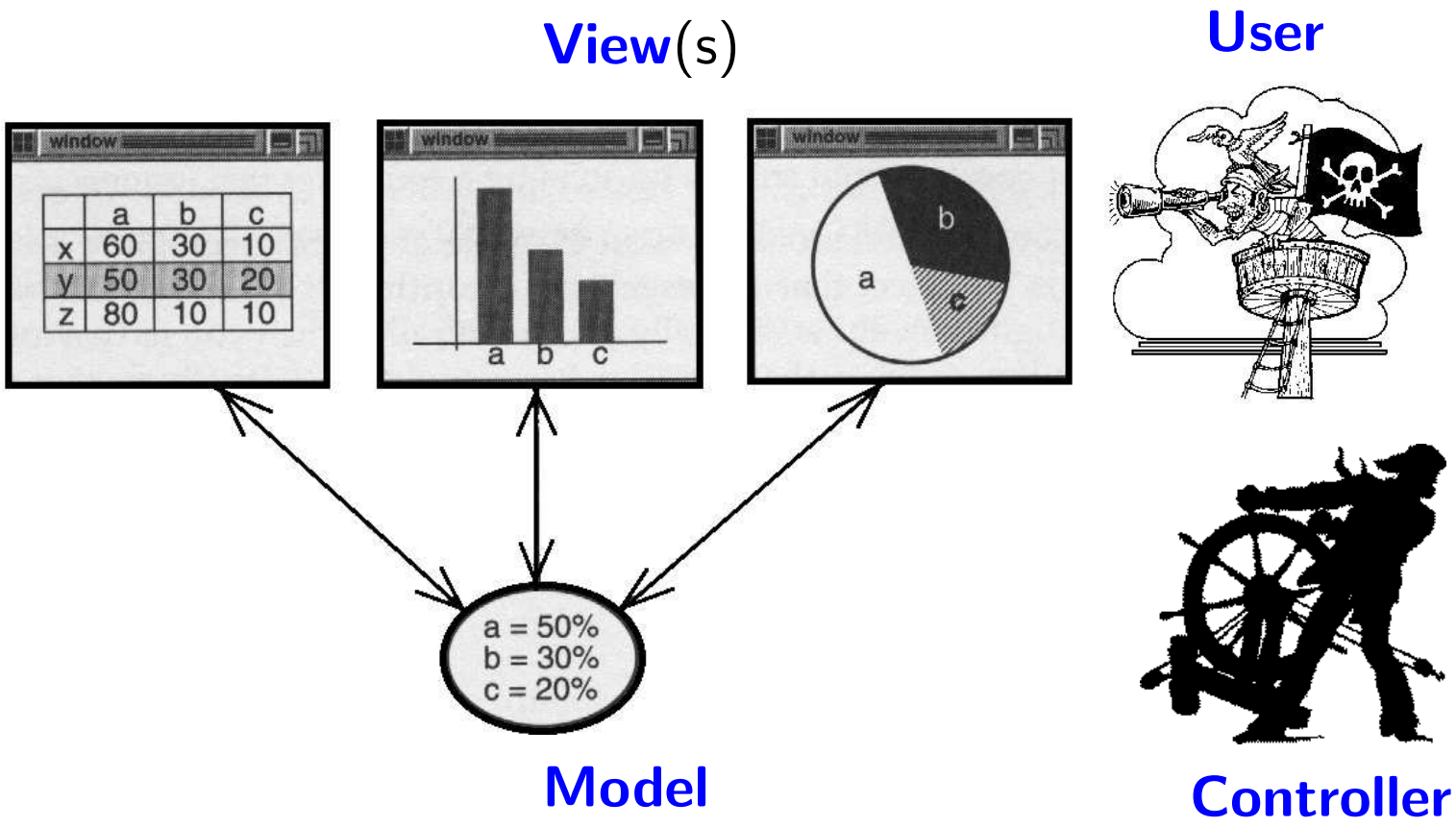
Model-View-Controller (MVC)
Gamma et al. (1995); Buschmann et al. (1996)

- Architectural style/design patternhybrid
- Aggregate design pattern out of
  - Composite
  - Observer
  - Strategy
  - Factory Method
- Clear distinction of data (model), data representation on a screen (view) and control of data manipulation or views (controller)

How MVC works – an overview



Description of MVC, Trygve M. H. Reenskaug

Intent Interactive applications with a flexible human-computer interface.

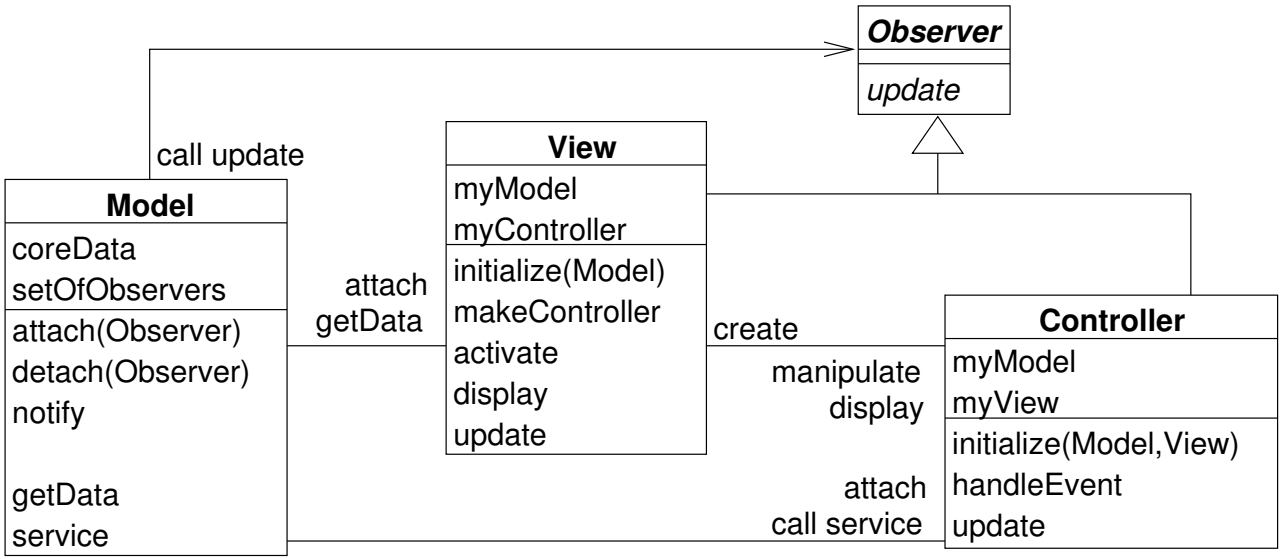
Motivation Adaptability and reuse

Participants MVC separates the application into three (independent) components

- *Model* offers core functionality and data
- *View* provides information to the user
- *Controller* handles user input

- All three components are related by a *change-propagation mechanism*.
- *View* and *Controller* constitute the user interface.

MVC class diagram



- **Model**: does not know View or Controller beforehand, announces change by calling *update*, related components request model state by *getData*, if needed keep data in a data base
- **View**: connected to Model, displays data (visually, acoustically, or similar) normally on a screen
- **Controller**: administrates Views, manipulates data on behalf of the user, "brain" of the application

Design Patterns of MVC in detail

Composite I

Classification object/structural

Intent

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

Also Known As —.

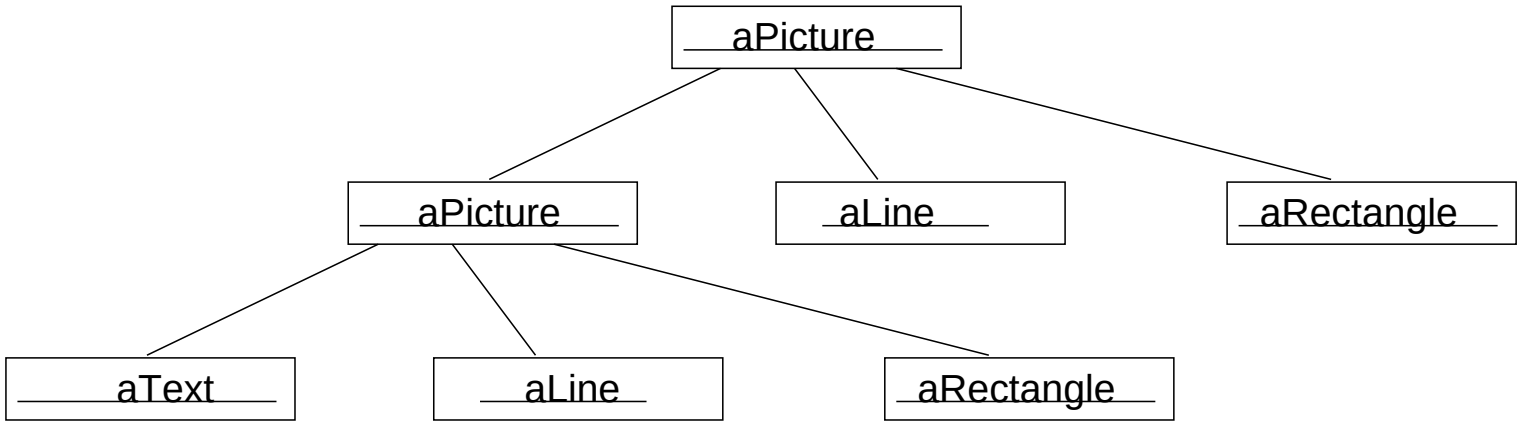
Motivation

Users can build complex diagrams out of simple components by using graphics applications.

Problem: Code that uses the corresponding classes must treat primitive and container objects differently, even if most of the time the user treats them identically. The Composite pattern describes how a recursive composition can be designed so that the client does not have to distinguish between primitive objects and containers.

Composite II

Example:



common operations: *draw*(), *move*(), *delete*(), *scale*()

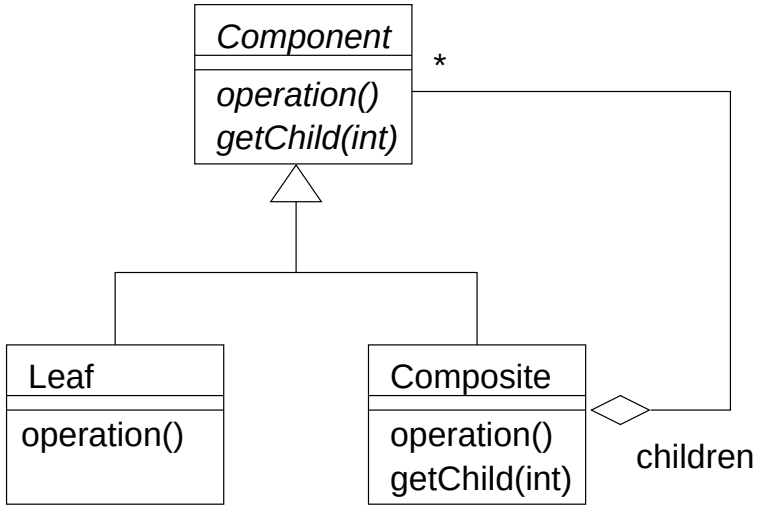
Composite III

Applicability

Use the Composite pattern when

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

Structure (abstract classes and operations are noted in *italics*)



Composite IV

Participants

- *Component* (graphic)
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes, as appropriate
  - declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- *Leaf* (rectangle, line, text etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition

Composite V

- *Composite* (picture)
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in the *Component* interface
- *Client* (not contained in the class diagram)
  - manipulates objects in the composition through *Component* interface

Collaborations

Clients use the *Component* class interface to interact with objects in the composite structure. If the recipient is a leaf, then the request is handled directly. If the recipient is a composite, then it usually forwards the request to its child components, possibly performing additional operations before and/or after forwarding.

Composite VI

Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects.
  Whenever client code expects a primitive object, then it can also take a composite object.
- makes the client simple
  Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite object. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

Composite VII

- makes it easier to add new kinds of components.
  Newly defined subclasses of *Composite* or *Leaf* work automatically with existing structures and client code. Clients don't have to be changed for new component classes.
- can make your design overly general
  The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Composite VIII

Implementation Gamma et al. (1995) considers the following aspects:

1. Explicit parents references
   Should be defined in the *Component* class.
2. Sharing components
   Can be useful to reduce storage requirements, but destroys tree structure.
3. Maximizing the *Component* interface
   Necessary to make clients unaware of the specific *Leaf* or *Composite* classes they are using. Default implementation in *Component* can be overwritten in subclasses.

Composite IX

4. Declaring the child management operations
   Declaration of *add*- and *remove*-operations in the class *Component* results in transparency; all components can be treated uniformly. Costs safety, because meaningless operations can be called, e.g. adding to objects to leafs.
   Defining child management in the *Composites* class gives safety, but is at the expense of transparency (leaves and composites have different interfaces).
5. Should *Component* implement a list of components?
   Incurs a space penalty for every leaf.
6. Child ordering
   When child ordering is an issue, applying the Iterator pattern is recommended.

Composite X

7. Caching to improve performance
   Useful, if the compositions have to be traversed or searched frequently.
8. Who should delete components?
   In languages without garbage collection, it's usually best to make a composite responsible for deleting its children when it's destroyed.
9. What's the best data structure for storing components?
   Depends on aspects of efficiency.

Composite XI

Sample Code

Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies.

```
1  class Equipment {
2      public:
3          virtual Equipment();
4          const char* Name() { return _name; }
5          virtual Watt Power();
6          virtual Currency NetPrice();
7          virtual Currency DiscountPrice();
8          virtual void Add(Equipment*);
9          virtual void Remove(Equipment*);
10         virtual Iterator<Equipment*>* CreateIterator();
11     protected:
12         Equipment(const char*);
13     private:
14         const char* _name;
15     };
```

Composite XII

Equipment declares operations that return the attributes of a piece of equipment, like its power consumption and cost. A *CreateIterator*-operation returns an iterator for accessing its parts.

Further classes such as *FloppyDisk* as class for leaves and *CompositeEquipment* for composite equipment are defined in the Gamma et al. (1995).

Composite XIII

Known Uses

- *View*-class in Model/View/Controller
- Composite structure for parse trees
- Portfolio containing assets

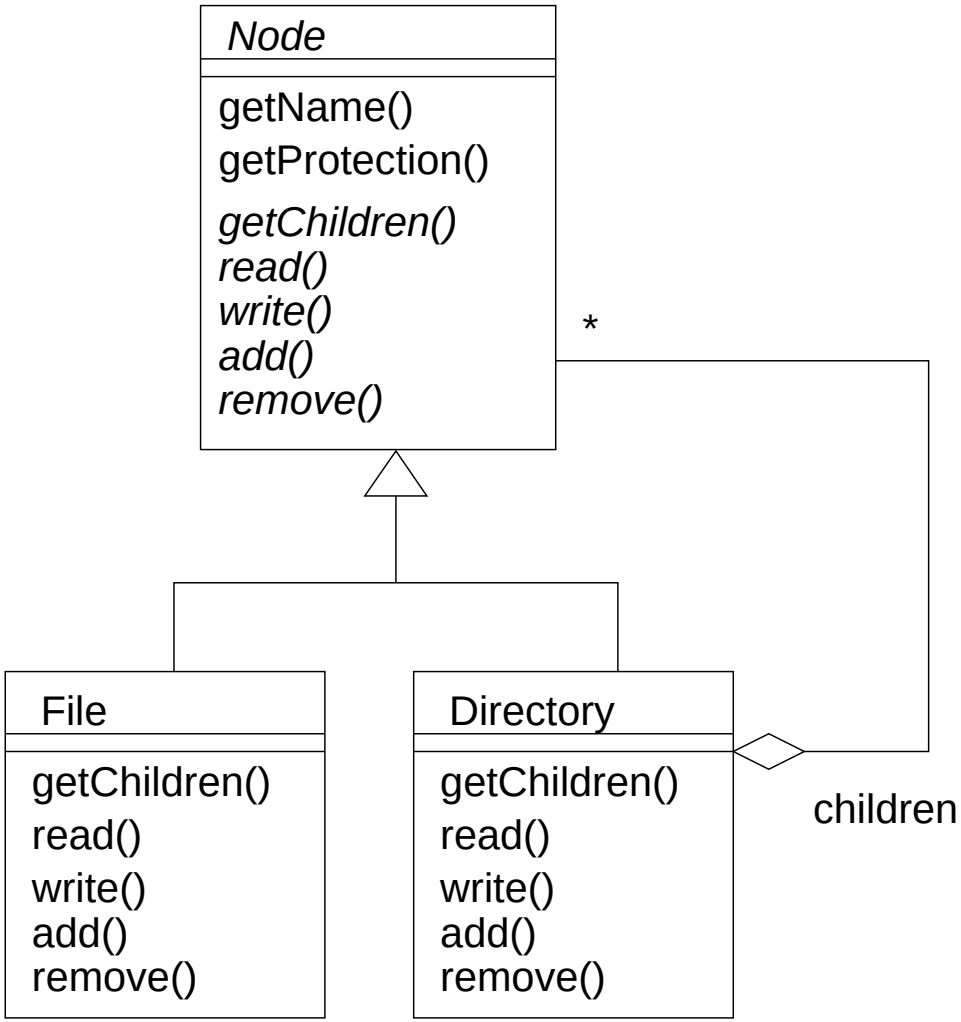
Related Patterns

- Often the component-parent link is used for a **Chain of Responsibility**.
- **Decorator** is often used with composites. When decorators and composites are used together, they will usually have a common parent class.
- **Flyweight** lets you share components, but they can no longer refer to their parents.
- **Iterator** can be used to traverse composites.
- **Visitor** localizes operations and behavior that would otherwise be distributed across *Composite* and *Leaf* classes.

Example: file system

- File system should be able to handle file structures of any size and complexity.
- *Directories* and (basic) *files* should be distinguished.
- The code, e.g. for selecting the name of a directory should be the same as for files. The same holds for size, access rights, etc.
- It should be easy to add new types of files (e.g. symbolic links).

Application of the composite pattern



Observer (or: Publisher/subscriber) I

Classification object/behavioral

Intent

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

Also Known As Dependents, Publish-Subscribe

Observer (or: Publisher/subscriber) II

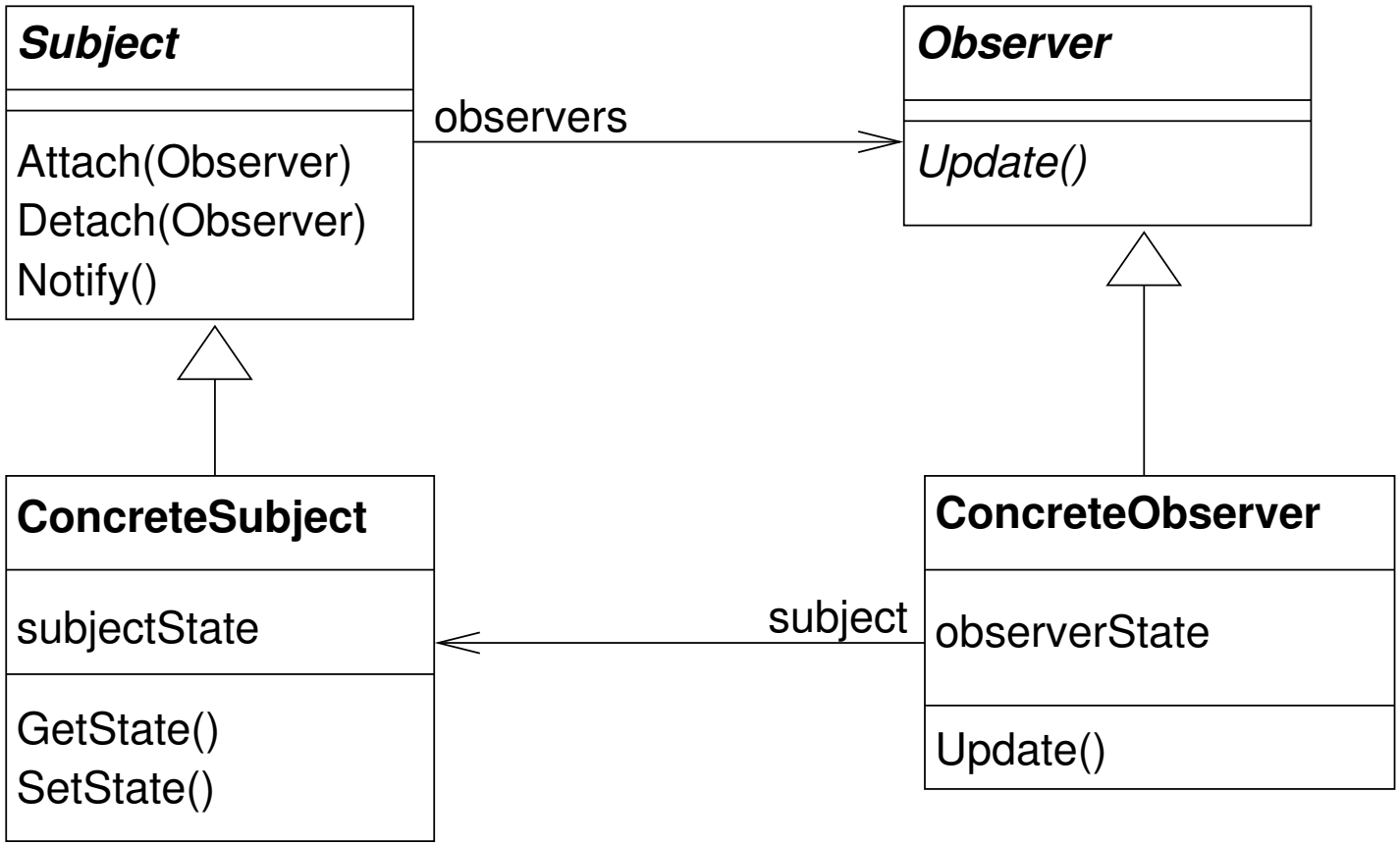
Applicability

Use the Observer pattern when

- change to one object requires changing others, and you do not know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who these objects are.
- data changes a one place, but many other components depend on this data
- the number and identity of dependent components is not known a priori or may change over timer
- polling is not feasible.

Observer (or: Publisher/subscriber) III

Structure



Observer (or: Publisher/subscriber) IV

Participants

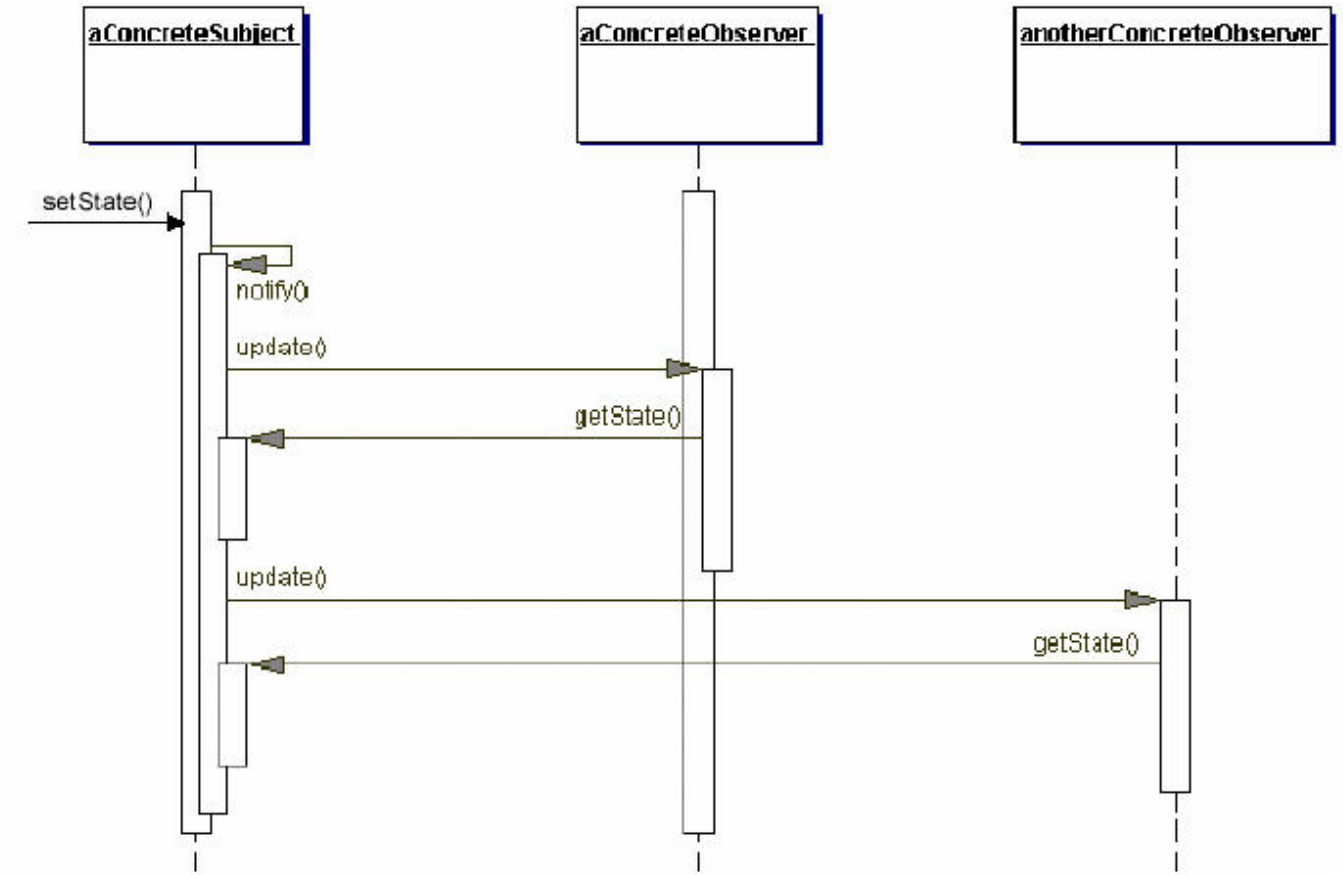
- *Subject*
  - knows its observers
  - provides an interface for attaching and detaching Observer objects
- *Observer*
  - defines an update interface for objects that should be notified of changes in a subject
- *ConcreteSubject*
  - stores state of interest to *ConcreteObserver* object
  - sends a notification to its observers when its state changes
  - Also called **publisher**.

Observer (or: Publisher/subscriber) V

- *ConcreteObserver*
  - maintains a reference to a *ConcreteSubject* object
  - stores state that should stay consistent with the *ConcreteSubject*'s
  - implements the `update`-interface of *Observer*
  - components/objects depend on changes
  - Also called **subscriber**.

Observer (or: Publisher/subscriber) VI

Dynamics:

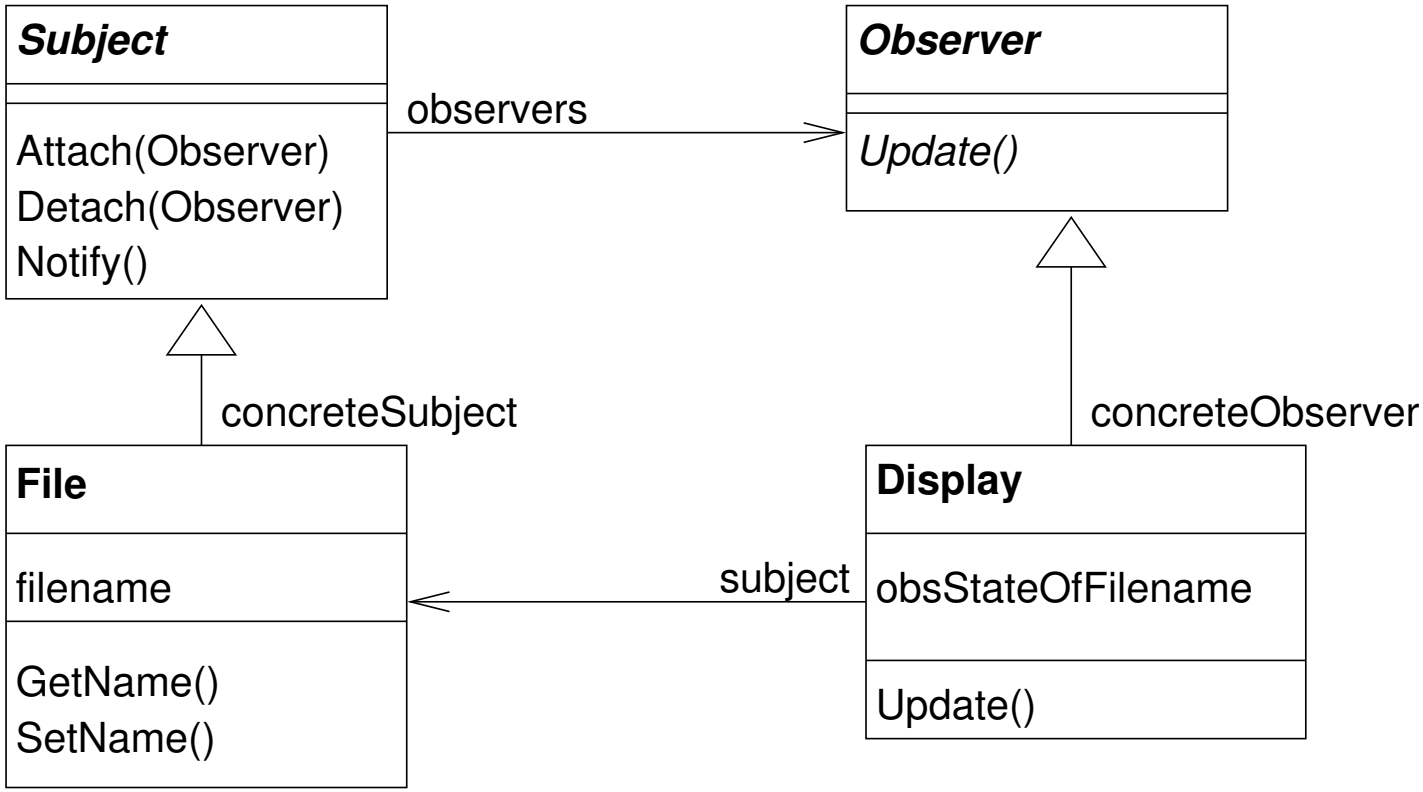


Related Patterns Mediator, Singleton

Example: file system

- When the name of a file or directory is changed (`setName`), the representation of the name on the display will be updated, too.

Application of the Observer pattern



Strategy I

Classification object/behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

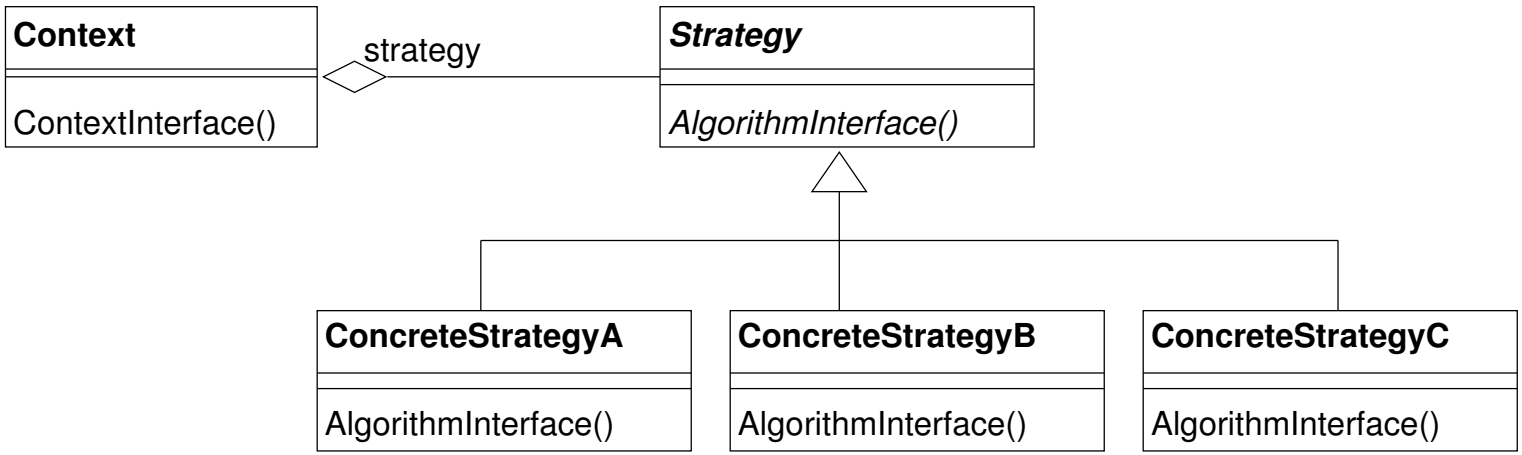
Also Known as Policy

Applicability Use the Strategy pattern when

- many related classes differ only in their behavior. Strategy provides a way to configure a class with one of many behaviors.
- you need different variants of an algorithm.
- an algorithm uses data that clients should not know about.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

Strategy II

Structure

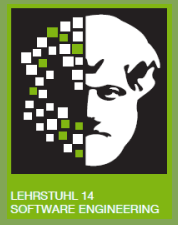


Strategy III

Participants

- *Strategy*
  - declares an interface common to all supported algorithms. Context uses this interface to call algorithm defined by a *ConcreteStrategy*.
- *ConcreteStrategy*
  - implements the algorithm using the *Strategy* interface.
- *Context*
  - is configured with a *ConcreteStrategy* object
  - may define an interface that lets *Strategy* access its data

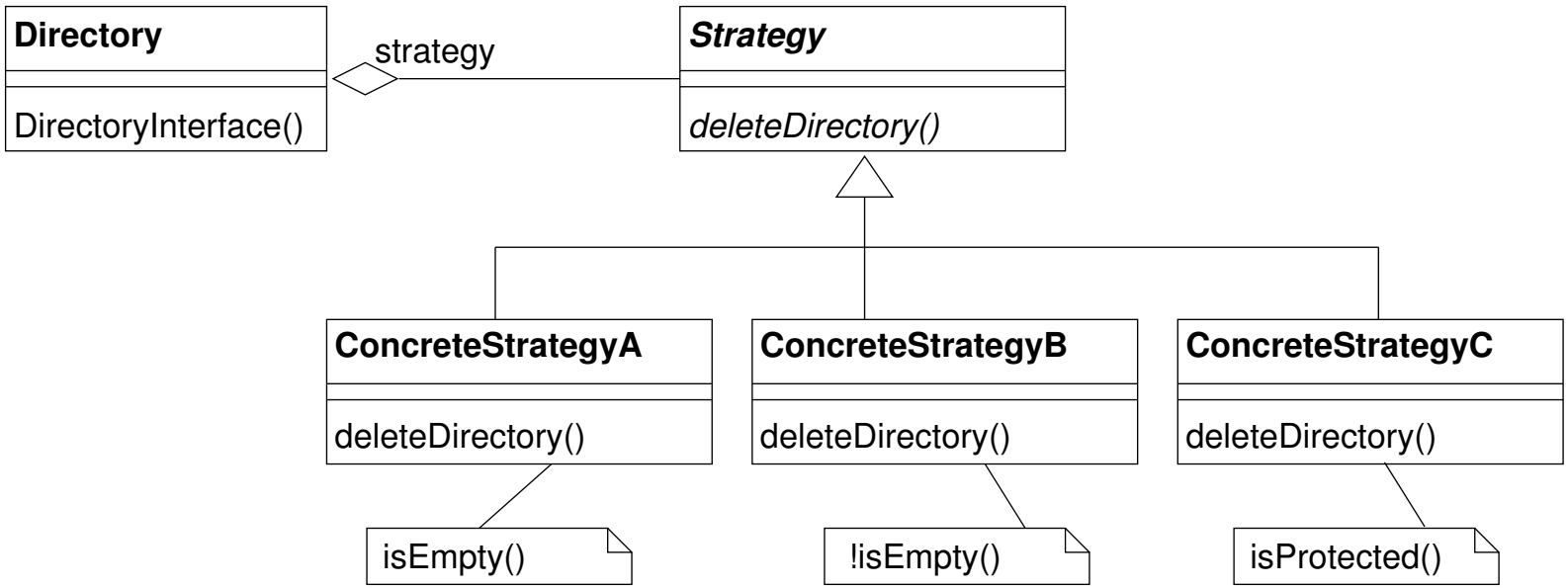
Related Patterns Flyweight



Example: file system

- It is not allowed to delete directories which contain files or are write protected.

Application of the Strategy pattern



Factory Method I

Classification creational

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate.

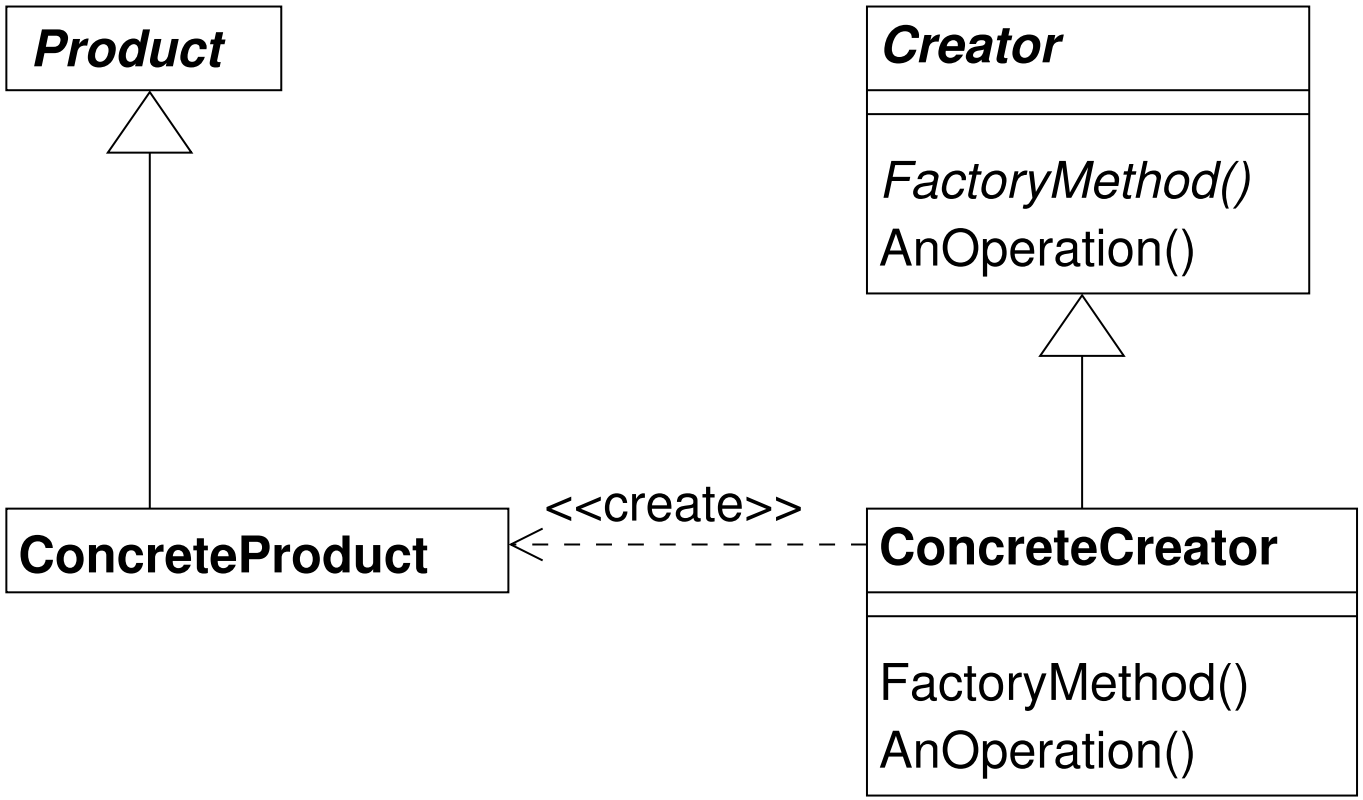
Also Known As Virtual Constructor

Applicability Use the Factory Method pattern when

- a class cannot anticipate the class of objects it must create.
- a class wants its subclass to specify the object it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method II

Structure



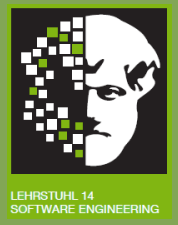
Factory Method III

Participants

- *Product* (Files)
  - defines the interface of objects the *FactoryMethod*() creates
- *ConcreteProduct* (Text-File)
  - implements the *Product* interface
- *Creator* (Application)
  - declares the *FactoryMethod*(), which returns an object of type *Product*
- *ConcreteCreator* (Open Office)
  - overrides the *FactoryMethod*() to return an instance of a *ConcreteProduct*

Related Patterns

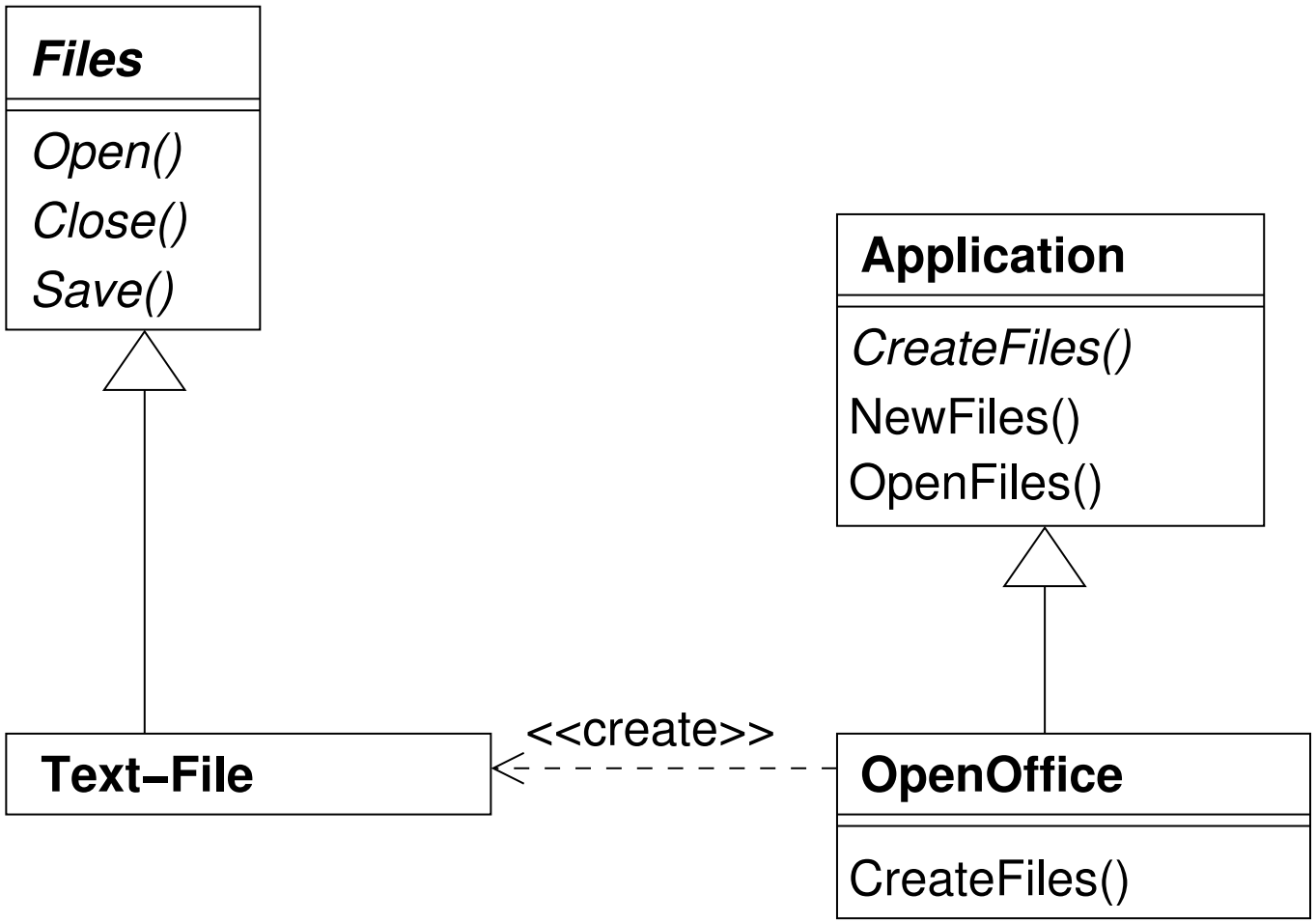
Abstract Factory, Template Method, Prototypes



Example: file system

- The file system offers the creation of files, where the kind of file to be created (.txt, .ods, .xls) depends on the particular application.

Application of the Factory Method pattern



Singleton I

Classification object/creational

Intent Ensure a class only has one instance and provide a global point of access to it.

Motivation

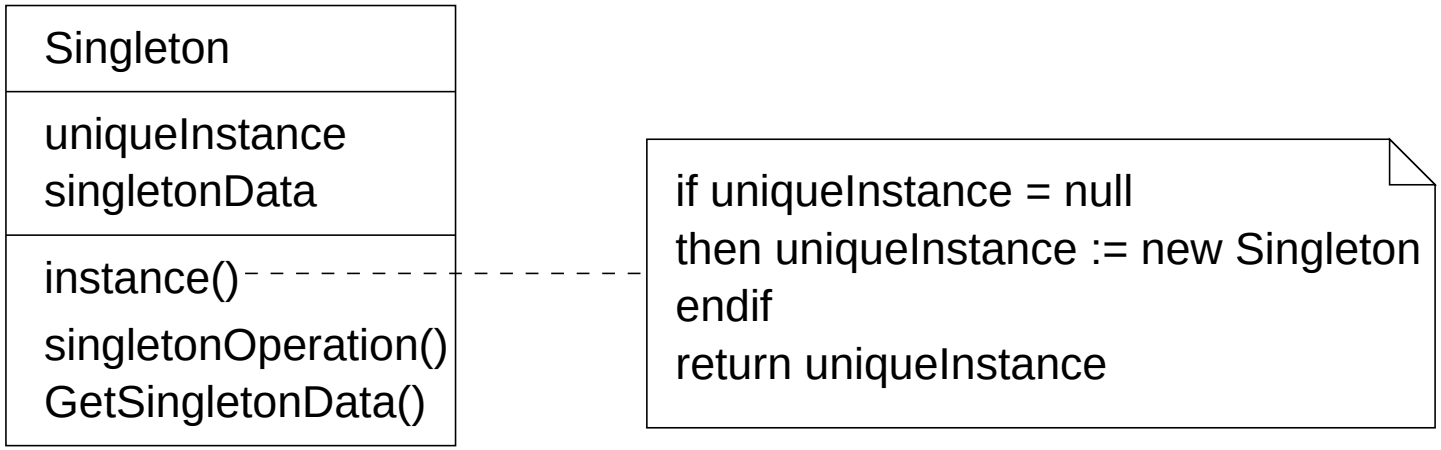
It's important for some classes to have exactly one instance (e.g., printer spooler). That class should be responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance.

Applicability Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton II

Structure



Participants

- *Singleton*
  - defines an *instance* operation that lets clients access its unique instance.
  - may be responsible for creating its own unique instance.

Singleton III

Collaborations

- Clients access a singleton instance solely through the singleton's *instance*-operation.

Consequences

- Controlled access to the sole instance.
- Improvement over global variables.
- Permits refinement of operations and representation through subclassing
- Permits a variable (!) number of instances.

Related Patterns Abstract Factory, Builder, Prototype

Example: file system

Consider users in a multi-user system:

- User logs in to the system.
  - generates an object of the class *UserSession*
- We want to ensure that
  - a only a maximum number of user sessions exist per user.
  - user sessions are only generated if authentication was successful.
- Basic concept:
  - singleton-pattern
  - variation necessary