Topic 7 - Behavioural Design Patterns

References:

Design Patterns Elements of Reusable Object-Oriented Software

Java Design Patterns A Tutorial

Head First Design Patterns

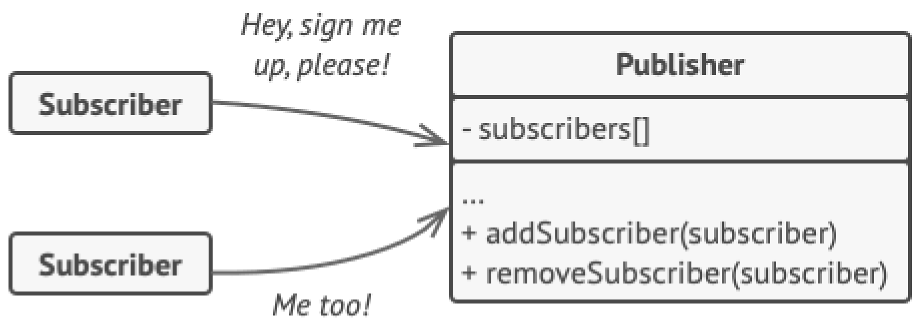
Mira Balaban Design Patterns

Design Patterns Explained

Observer: Motivation

- A spreadsheet view and bar chart are different presentations of the same application data object.
- The data objects are independent of the presentation layer.
- The different presentations do not know about each other.
- The presentations should be **notified** about changes in the data object.

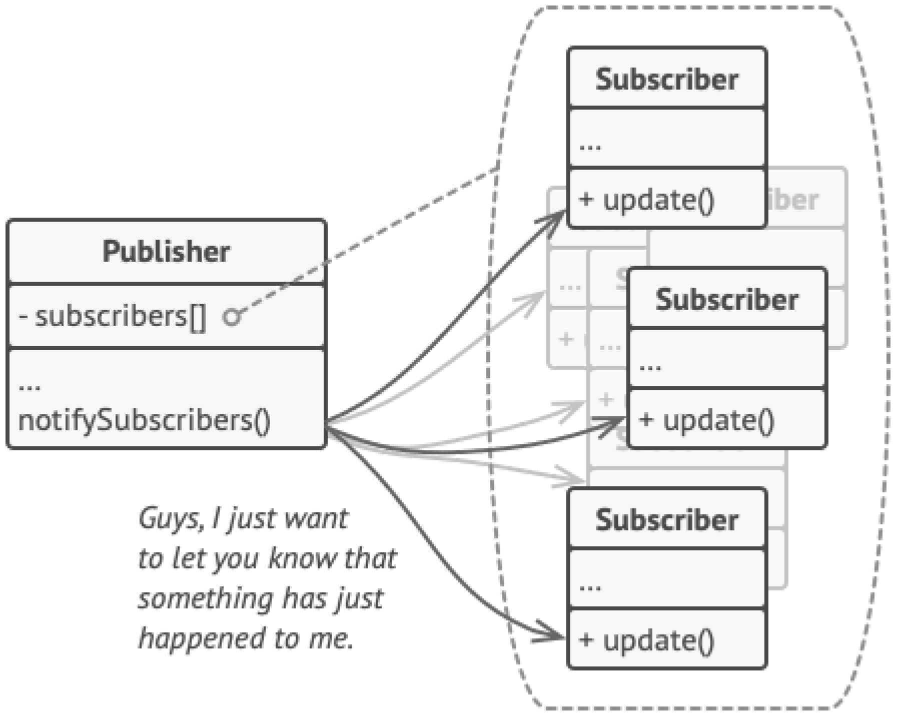
- *See Eric Gamma et al.:* **Design patterns : elements of reusable object-oriented software** Erich Gamma c1995 (avail. in library)



https://refactoring.guru/design-patterns/observer

Observer: Solution

- Key objects: *subject* (Observable) and *observer*.
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject changes its state.
- Each observer can query the subject to synchronize their states.



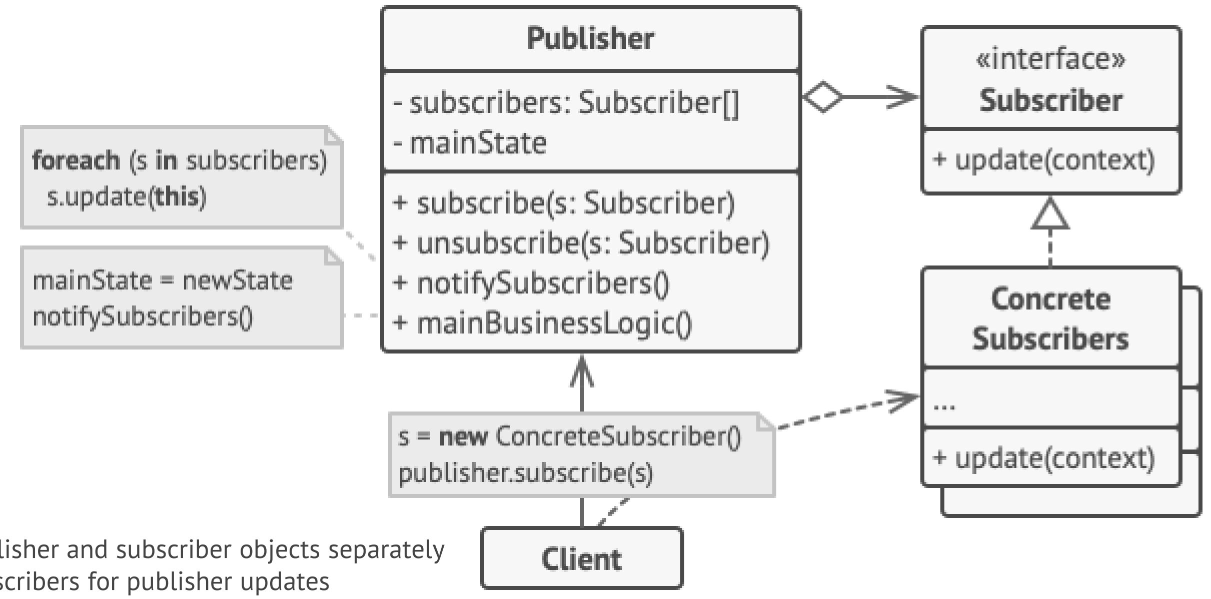
Observer: Participants

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

Observer: Class Diagram

1.The **Publisher** issues events of interest to other objects. These events occur when the publisher changes its state or executes some behaviors. Publishers contain a subscription infrastructure that lets new subscribers join and current subscribers leave the list.

➔ When a new event happens, the publisher goes over the subscription list and calls the notification method declared in the subscriber interface on each subscriber object.



```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

The Subscriber interface declares the notification interface. In most cases, it consists of a single update method. The method may have several parameters that let the publisher pass some event details along with the update.

Concrete Subscribers perform some actions in response to notifications issued by the publisher. All of these classes must implement the same interface so the publisher isn't coupled to concrete classes.

The **Client** creates publisher and subscriber objects separately and then registers subscribers for publisher updates

Model View Controller

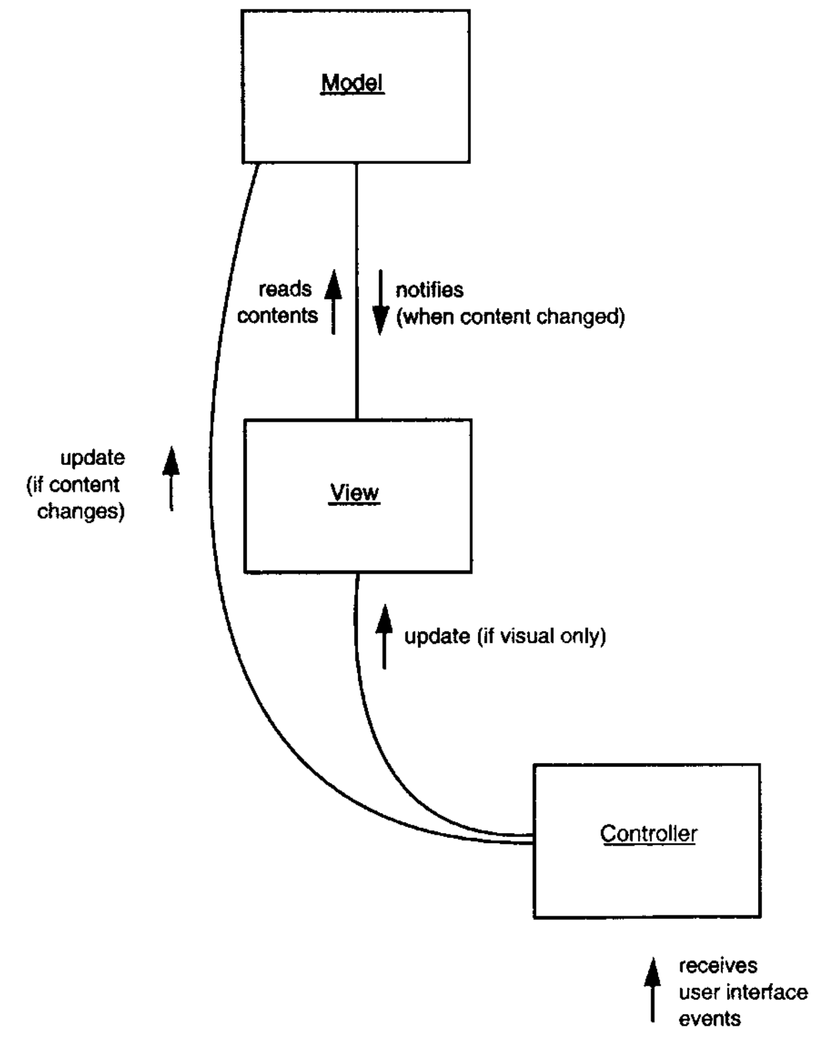
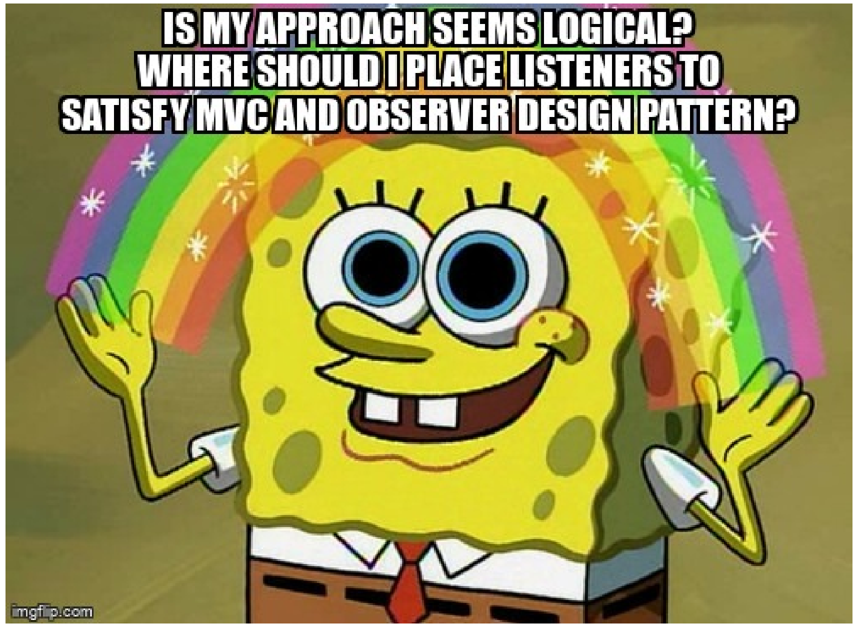

Model
reads contents
notifies (when content changed)
update (if content changes)
View
update (if visual only)
Controller
receives user interface events

Figure 9–3: Interactions between model, view, and controller objects

Observer: intent and context

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- apply *Observer* when
  - an abstraction has two aspects, one dependent on the other.
  - a change to one object requires changing others
  - an object should be able to notify other objects without making assumptions about the identity of these objects.


IS MY APPROACH SEEMS LOGICAL?
WHERE SHOULD I PLACE LISTENERS TO
SATISFY MVC AND OBSERVER DESIGN PATTERN?
imgflip.com

Observer: Considerations (1)

- Sometimes *observers* need to observe more than one *subject.*
- who triggers the update?
  - state-changing *subject* methods call *notify()* method (preferred), or
  - make clients responsible for calling *notify().*
- avoid dangling references when deleting *subjects*

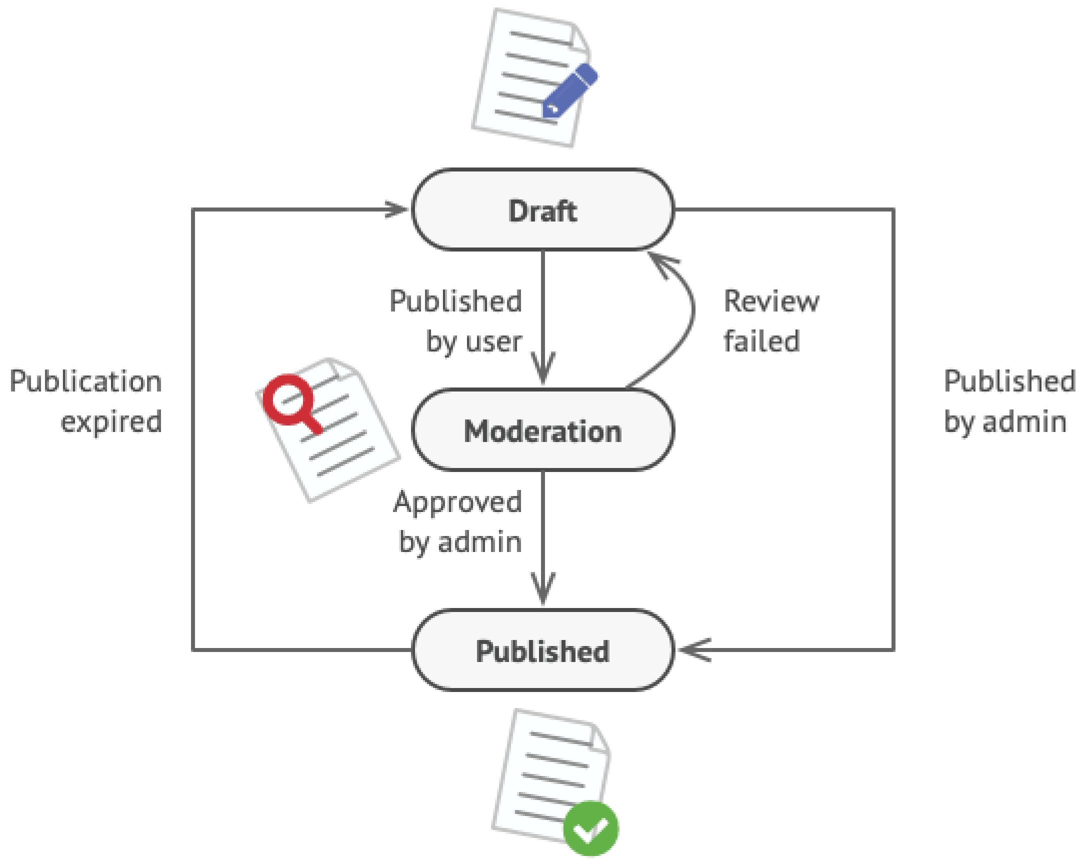
  -- *Subject* notifies its observers about its deletion.
- make sure *Subject*'s state is self-consistent before calling *notify* (the *observers* will query the state).

Observer: Considerations (2)

- avoiding observer-specific update protocols
  - push model: subject sends its observers detailed information about the changes
  - pull model: subject only informs observers that state has changed; observers need to query subject to find out what has changed/get changes
- specifying modifications of interest explicitly (publish/subscribe approach)
  - of interest when *observer* are interested in only some of the state-changing events:
    - *Subject.attach(Observer, interest)*
    - *Observer.update(Subject, interest)*
- encapsulating complex update semantics
  - when there is a highly complex relationship between subject and observer, introduce a ChangeManager class to reduce the amount of work.

State

- **State** is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.
- The State pattern is closely related to the concept of a **Finite-State Machine**.
- However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called *transitions*, are also finite and predetermined.



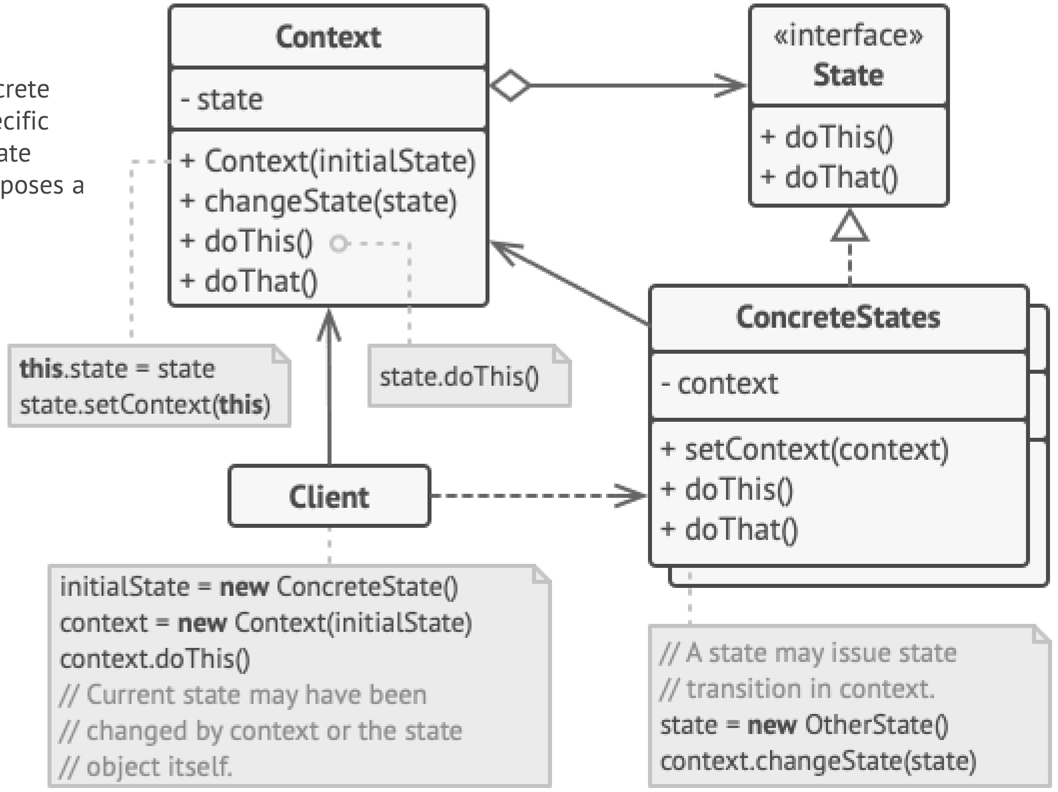
https://refactoring.guru/design-patterns/state https://refactoring.guru/design-patterns/state

State: Structure

The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.

Context stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.

Concrete States provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.
State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.



```
this.state = state
state.setContext(this)
```

```
state.doThis()
```

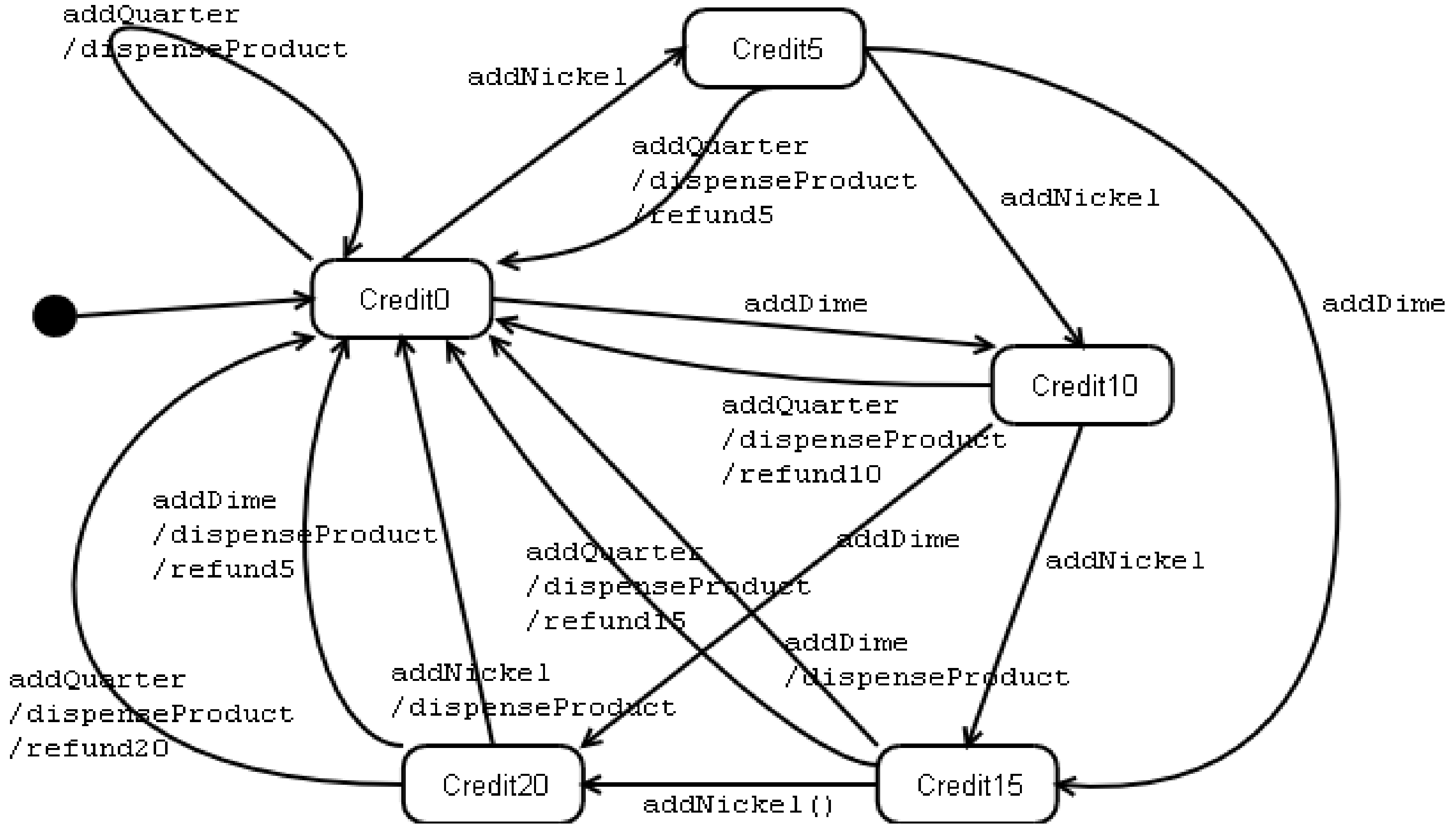
```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

Both context and concrete states can set the next state of the context and perform the actual state transition by replacing the state object linked to the context

State: Example

- example of an American vending machine:
  - product price is $0.25
  - machine accepts any combination of nickels (5 cents), dimes (10c), and quarters (25c)
  - customer enters coins; when credit reaches $0.25 product is dispensed, and refund is given for the remaining credit.
  - machine has display that shows the current balance


addQuarter /dispenseProduct
addNickel
Credit5
addQuarter /dispenseProduct /refund5
addNickel
addDime
Credit0
addDime
Credit10
addQuarter /dispenseProduct /refund10
addDime /dispenseProduct /refund5
addQuarter /dispenseProduct /refund15
addDime
addNickel
addDime /dispenseProduct
addQuarter /dispenseProduct /refund20
addNickel /dispenseProduct
Credit20
addNickel()
Credit15

“Traditional” implementation

- use integers to represent the states
  - more complex states may require enumerated types or object representation
- methods *addNickel(), addDime(),* and *etc.* to model user actions
- methods *refund(), displayBalance(),* and *dispenseProduct()* to model system’s actions
- conditional logic (with *if/switch* statements) depending on current state

Observations

- state-specific behavior scattered over different conditionals
  - changing one state's behavior requires visiting each of these
- inflexible: adding a state requires invasive change (editing each conditional)
- approach tends to lead to large monolithic classes
  - not clear how to partition functionality to decompose for greater cohesion

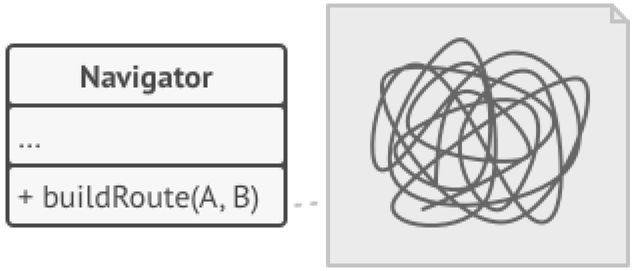
- See `state.vending.TraditionalVendingMachine` for naïve implementation
- See `state.vending.VendingMachine` for State pattern implementation

State: Benefits

- localizes state-specific behavior, and partitions behavior for different states
  - leads to small cohesive classes instead of one large uncohesive class
  - provides a natural structure for easier maintenance and comprehensibility
- avoids (long) if/switch statements with state-specific control flow
  - more extensible/maintainable i.e. you don't have to edit your switch statements after adding a new state
- makes state transitions explicit
  - create a new *ConcreteState* object, and assign it to the state field in *Context* upon each transition (including *start* and *stop* states)
- *state*-objects can be shared
  - and common functionality can be placed in abstract class *State*

Strategy

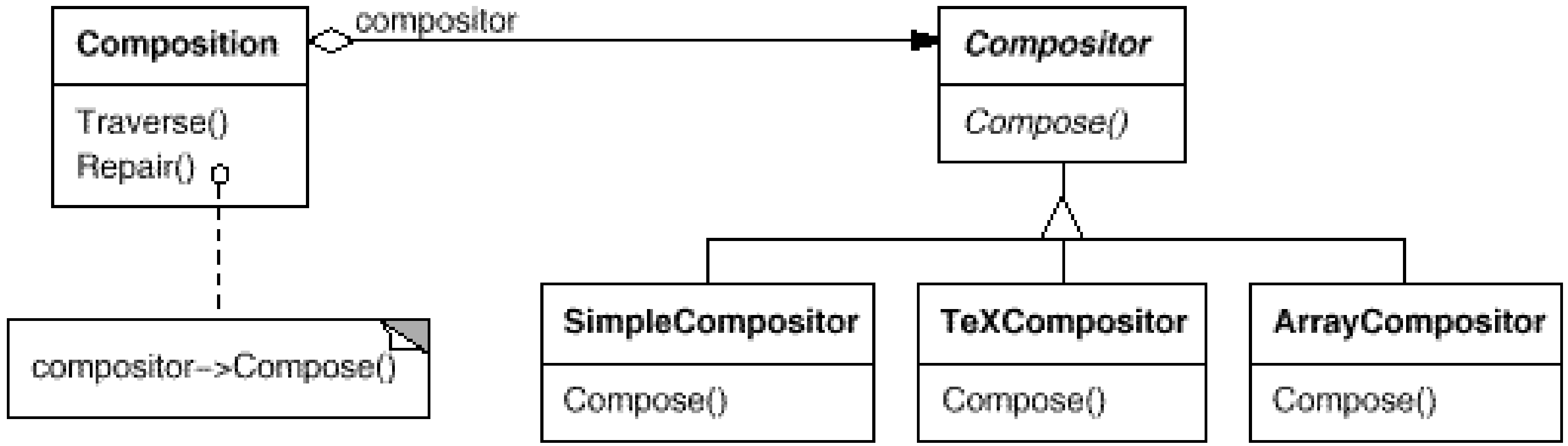
- Breaking a stream of text into lines.
- Many algorithms (plain text, html, java etc.)
- Hard wiring into the supplier (Composition) means the client cannot choose the algorithm
- Hard-wiring all such algorithms into the client classes isn't desirable:
  - Clients get more complex, harder to maintain.
  - Different algorithms are spread across different clients.
  - Difficult to add algorithms and vary existing ones when they are an integral part of a client.



https://refactoring.guru/design-patterns/strategy

Strategy: Solution

- Define classes that encapsulate different *linebreaking* algorithms -- a ***strategy***.



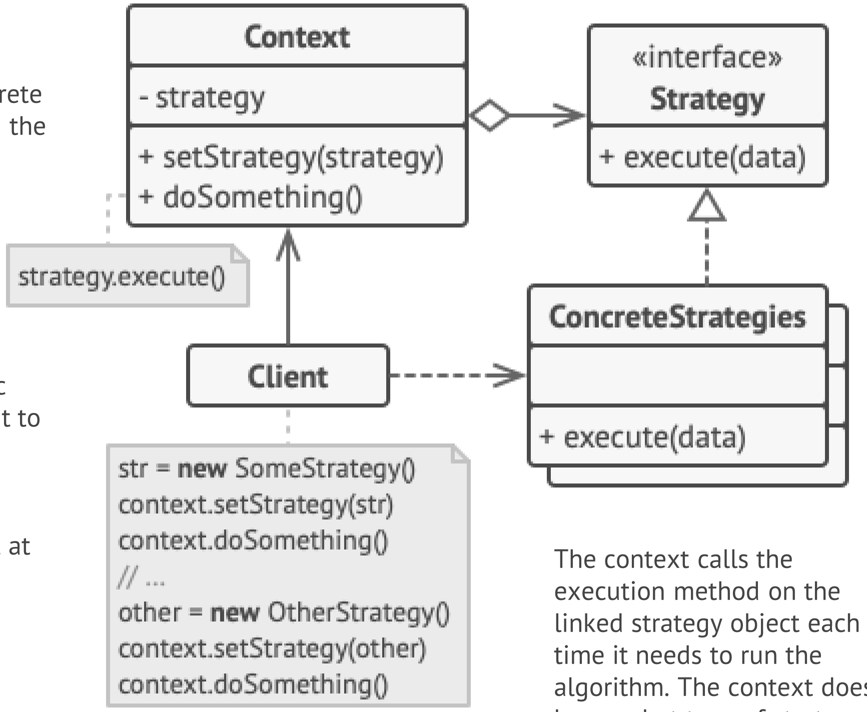
- Composition class is responsible for maintaining and updating the *linebreaks* of text displayed in a text viewer.

Strategy: Class diagram

https://refactoring.guru/design-patterns/strategy

The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.

The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.



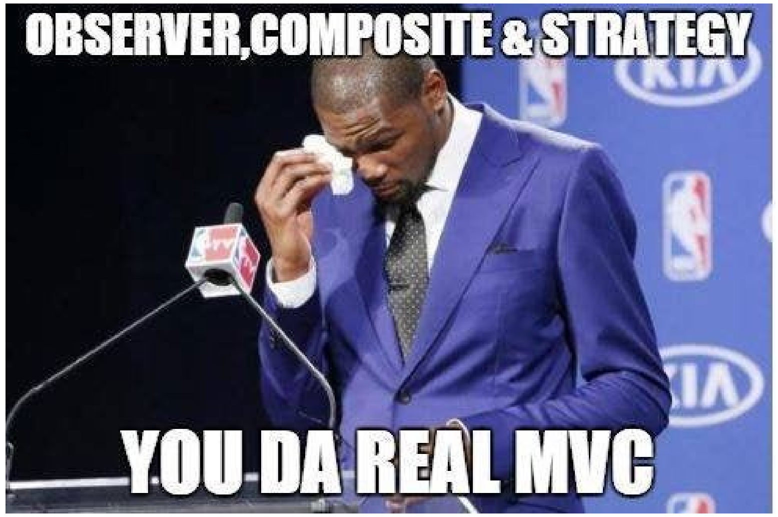
```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

The **Client** creates a specific strategy object and passes it to the context. The context exposes a setter which lets clients replace the strategy associated with the context at runtime.

Concrete Strategies implement different variations of an algorithm the context uses.

The context calls the execution method on the linked strategy object each time it needs to run the algorithm. The context doesn't know what type of strategy it works with or how the algorithm is executed.

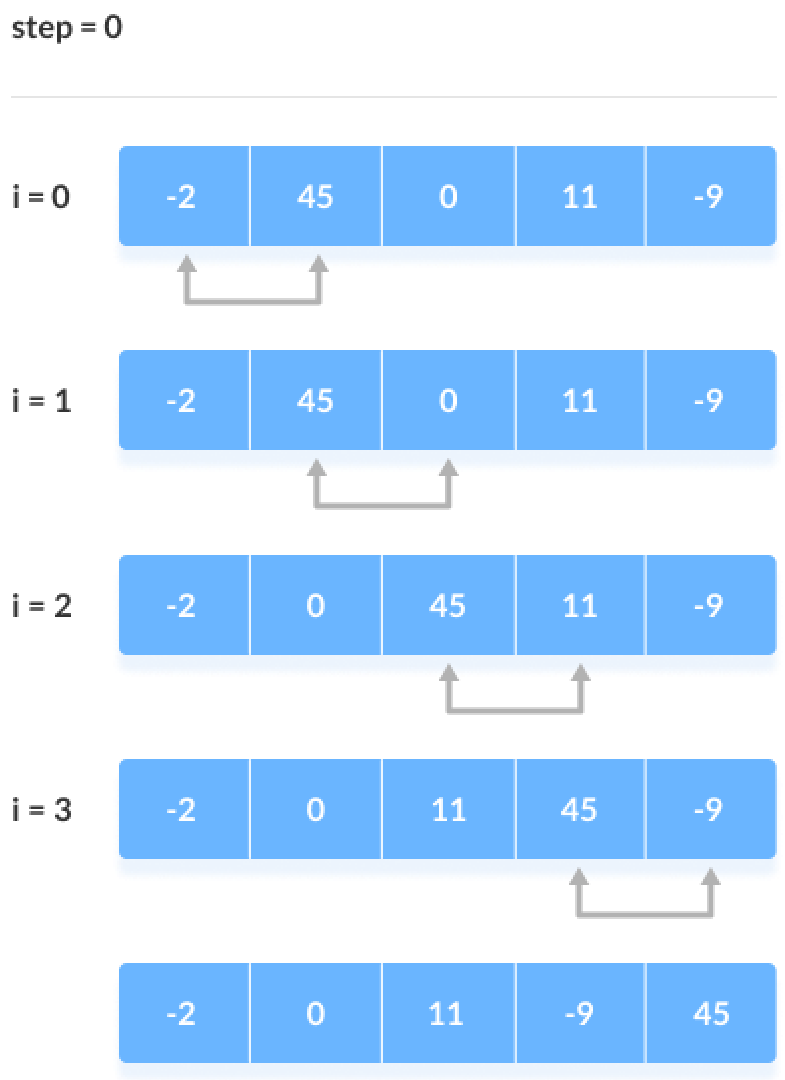
Strategy: Intent and context



- Define a family of algorithms, encapsulate each one, and make them interchangeable. *Strategy* lets the algorithm vary independently from the clients that use it.
- Use *Strategy* when:
  - –you need different variants of an algorithm (e.g. with different time/space tradeoffs)
  - –you want to avoid exposing details/data structures of an algorithm that clients shouldn’t know about

Strategy: Example

- method *searchByAuthorSorted()* from an implementation of a book-selling system
  - computes a *List* of *Books*
  - sorts this List by calling *BubbleSorter.sort()*, which implements bubble-sort and returns a sorted *List*
- This design hard-codes the choice of a specific sorting algorithm into the supplier
- see `strategy.search.TestSearchBasic`

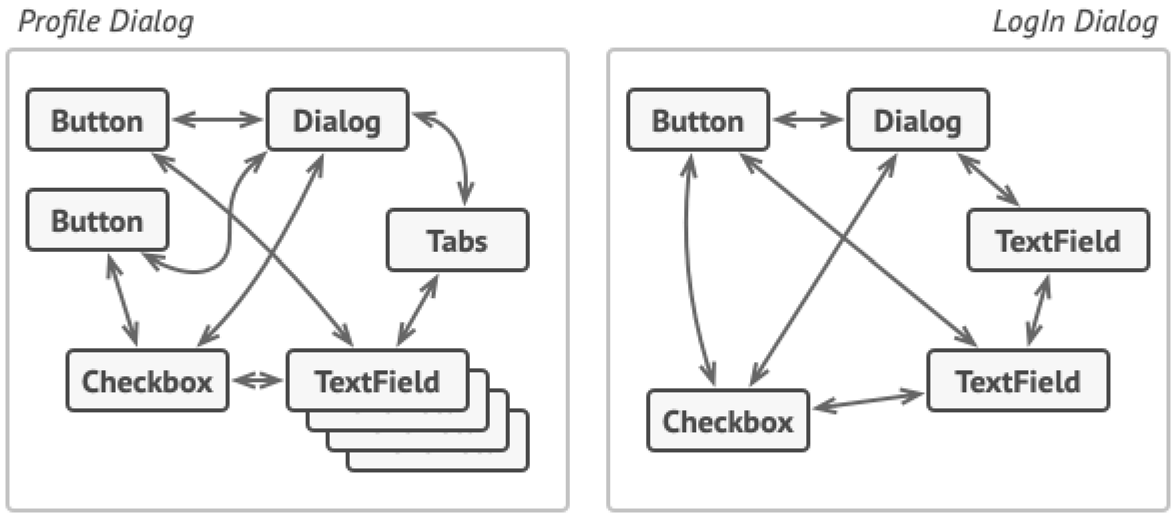


Strategy: Considerations

- suitable for families of algorithms with similar interfaces
- avoids conditional statements in the Context classes (client classes)
- avoids subclassing to provide a different implementation e.g. Template Method pattern
- clients must be aware of different strategies and select one of them
- performance penalty:
  - additional overhead of communication between Strategy and Context
  - increased number of objects
- Can use generics to select the strategy if it is fixed and known at compile time
- Can use a factory to select the strategy

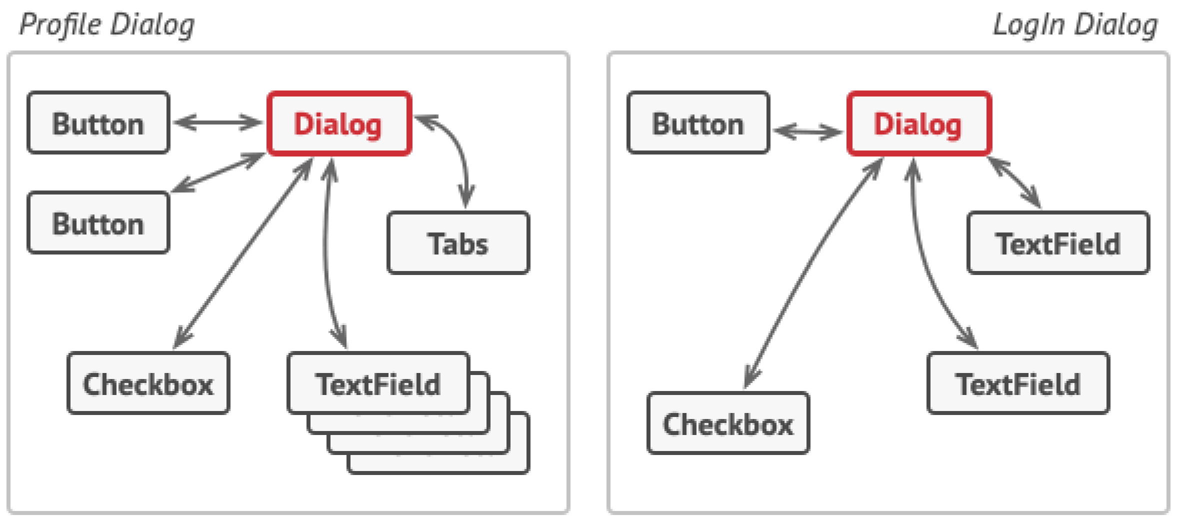
Mediator: Motivation

- When the program starts, the Copy and Clear buttons are disabled.
- When you select one of the items in the left box, it is copied into the textbox for editing, and the copy button in enabled.
- When you click on the Copy button, the selected name is added to the right box, and the Clear button is enabled.
- When Clear is clicked the right box and textfield are cleared and buttons are disabled again.
- In general, user interfaces involving list boxes and buttons have complex interactions (e.g. Config Dialog Boxes)



Mediator: Solution

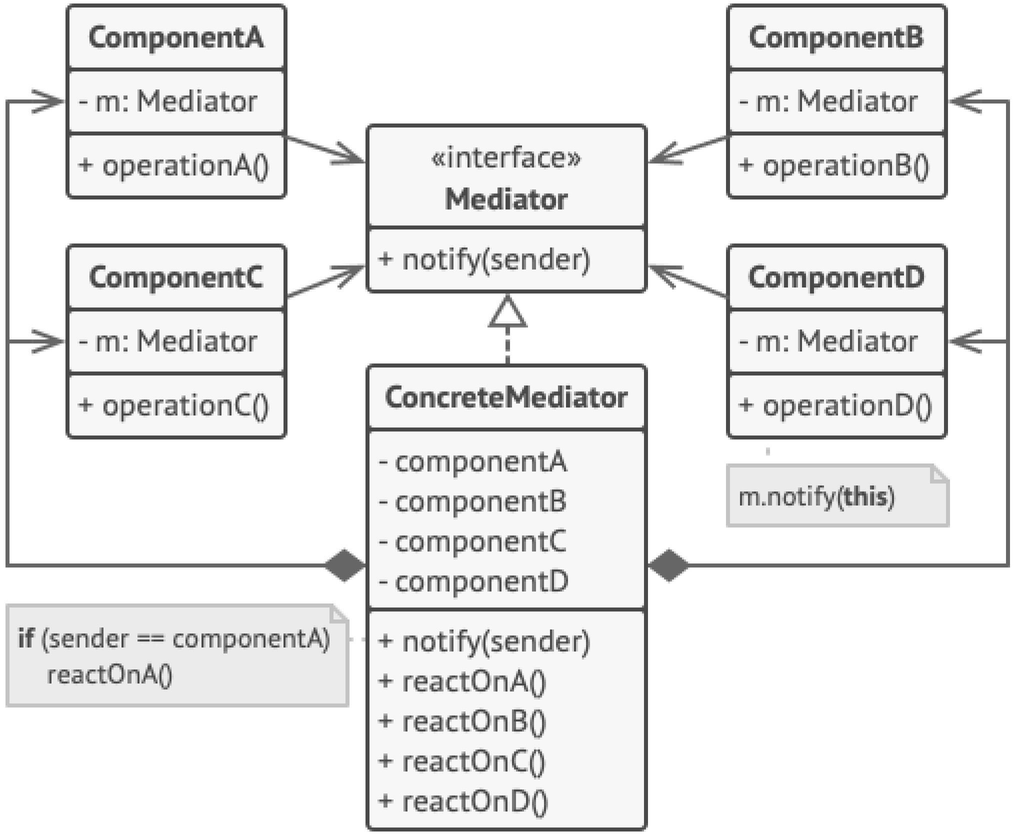
- The Mediator pattern simplifies the interaction by being the central class that is aware of the other classes in the system
- Each control that Mediator communicates with is called a Colleague.
- Each Colleague informs the Mediator when it receives a user event* and Mediator decides which classes should be informed.
- * usually via a UI controller/listener



Mediator:Participants & Structure

The **Mediator** interface declares methods of communication with components, which usually include just a single notification method. Components may pass any context as arguments of this method, including their own objects, but only in such a way that no coupling occurs between a receiving component and the sender's class.

Components are various classes that contain some business logic. Each component has a reference to a mediator, declared with the type of the mediator interface. The component isn't aware of the actual class of the mediator, so you can reuse the component in other programs by linking it to a different mediator.

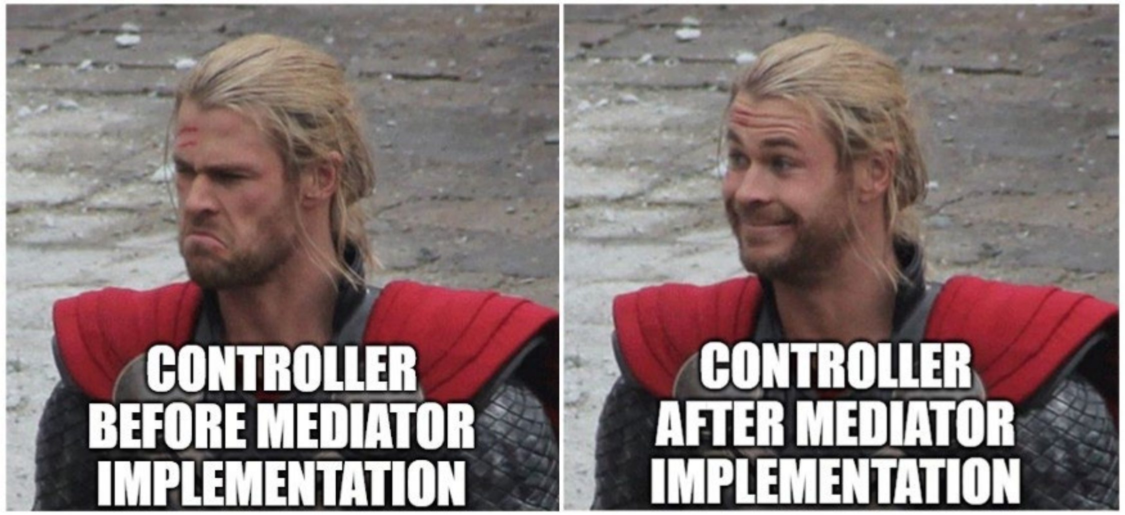


Concrete Mediators encapsulate relations between various components. Concrete mediators often keep references to all components they manage and sometimes even manage their lifecycle.

Mediator: Consequences

- Reduces coupling since:
  - Only the mediator has high coupling
  - Clients/colleagues are only coupled to the one mediator
  - Otherwise many UI components (colleagues) are coupled to many other UI components!

- Shares similarities with facade however communication is bi-directional
  - in the provided example the colleagues reference to the mediator is delegated to a separate controller (one per colleague/view)


CONTROLLER BEFORE MEDIATOR IMPLEMENTATION
CONTROLLER AFTER MEDIATOR IMPLEMENTATION

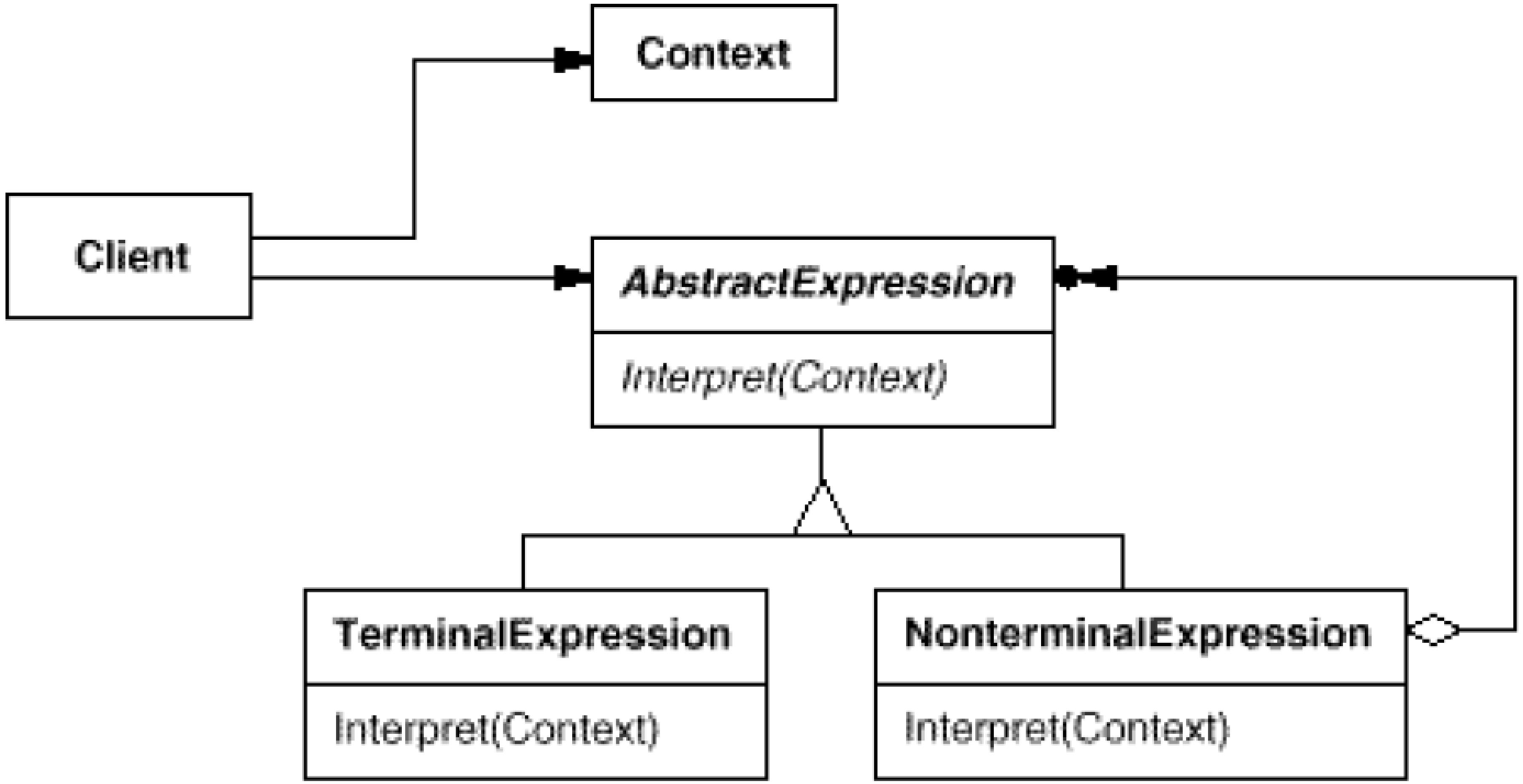
Mediator: Implementation Details

- Attention needs to be given to the construction process since this cannot take place in a single phase
- i.e. the components/colleagues need a reference to the mediator
- BUT the mediator needs references to the colleagues as well
- therefore a two stage construction is required
- stage 1: construct the mediator so it can be passed to the components (or Singleton as in the example code described below)
- stage 2: register the colleagues with the mediator via set method(s)
- the provided solution uses a singleton for static access from the colleagues (since many colleagues are coupled to it)
- but still requires the separate registration process since this cannot be handled via the getInstance() method (without introducing extensive coupling!)

Interpreter : Motivation

- If a problem has many variations then it can be useful to express the problem as a sentence in a simple language that can be interpreted. For example consider commands that can be issued to a robot (move – up – 2, turn-left, pickup …)
- The interpreter pattern describes how to define a grammar for a language, represent it and interpret it.
- e.g. regular expression syntax for pattern matching https://en.wikipedia.org/wiki/Regular_expression

Interpreter: General Structure



From: Erich Gamma et al., *Design patterns : elements of reusable object-oriented software* , Addison-Wesley, 1995

Interpreter: Participants

AbstractExpression
Declares an abstract interpret operation common to all nodes in the abstract syntax tree

TerminalExpression
Implements an interpret operation associated with terminal symbol

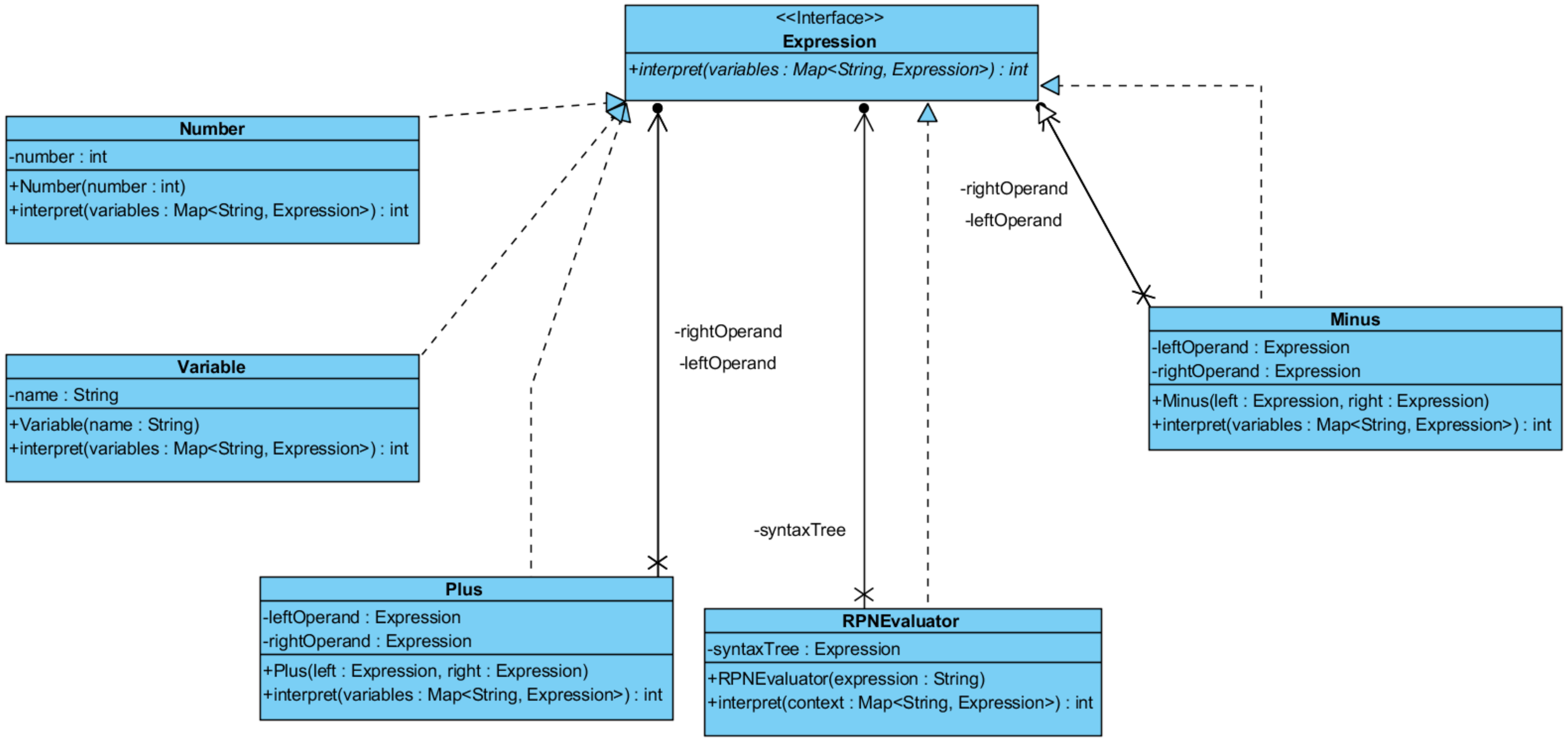
NonterminalExpression (+,-,alternation, repetition , …)
One class required for every rule in the grammar
The interpret typically calls itself recursively

Context
Information global to the interpreter such as the list of variables to be evaluated

Client
Builds an abstract syntax tree and invokes the interpret operation.

NOTE: there is a trade off between good structure and comprehensibility, and reduced performance due to recursive solution used by this pattern.

Interpreter: Class Diagram of Code Example



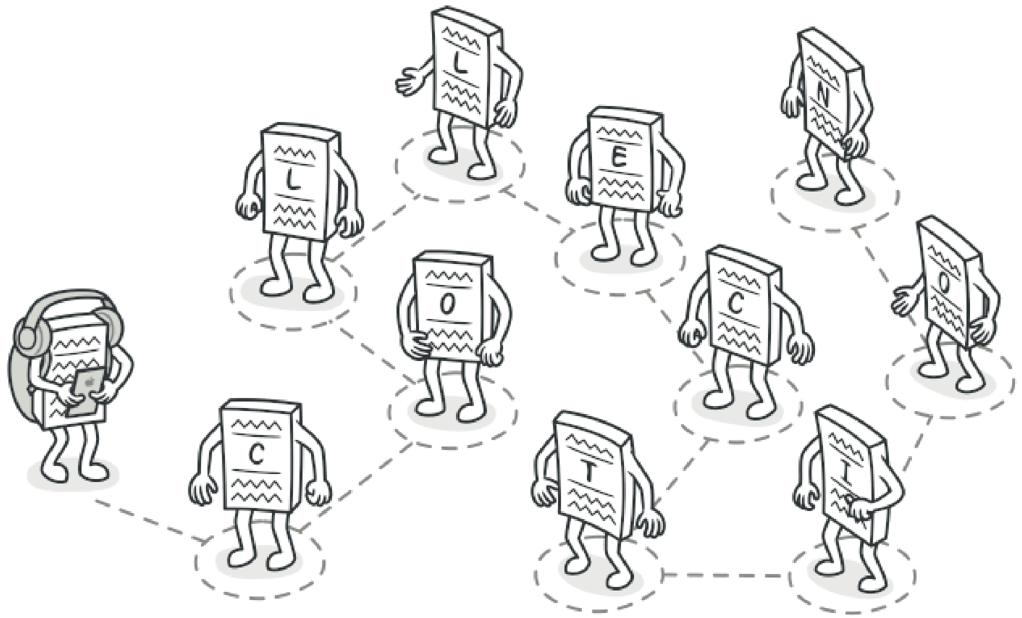
<<Interface>>
Expression
+interpret(variables : Map<String, Expression>) : int
Number
-number : int
+Number(number : int)
+interpret(variables : Map<String, Expression>) : int
Variable
-name : String
+Variable(name : String)
+interpret(variables : Map<String, Expression>) : int
Plus
-leftOperand : Expression
-rightOperand : Expression
+Plus(left : Expression, right : Expression)
+interpret(variables : Map<String, Expression>) : int
RPNEvaluator
-syntaxTree : Expression
+RPNEvaluator(expression : String)
+interpret(context : Map<String, Expression>) : int
Minus
-leftOperand : Expression
-rightOperand : Expression
+Minus(left : Expression, right : Expression)
+interpret(variables : Map<String, Expression>) : int
-rightOperand
-leftOperand
-syntaxTree
-rightOperand
-leftOperand

Interpreter : Consequences

- *Easy to change and extend the grammar.*
  - add or refactor classes
- *Implementing the grammar is easy*
  - since inheritance can share a lot of functionality
- *Complex grammars are harder to maintain*
  - explosion in number of classes and their relationships
  - parser or compiler generators may be more appropriate
- *New ways to interpret expressions (advanced!)*
  - by adding operations to the Expression classes
  - consider visitor if many or uncohesive operations

Iterator: Motivation

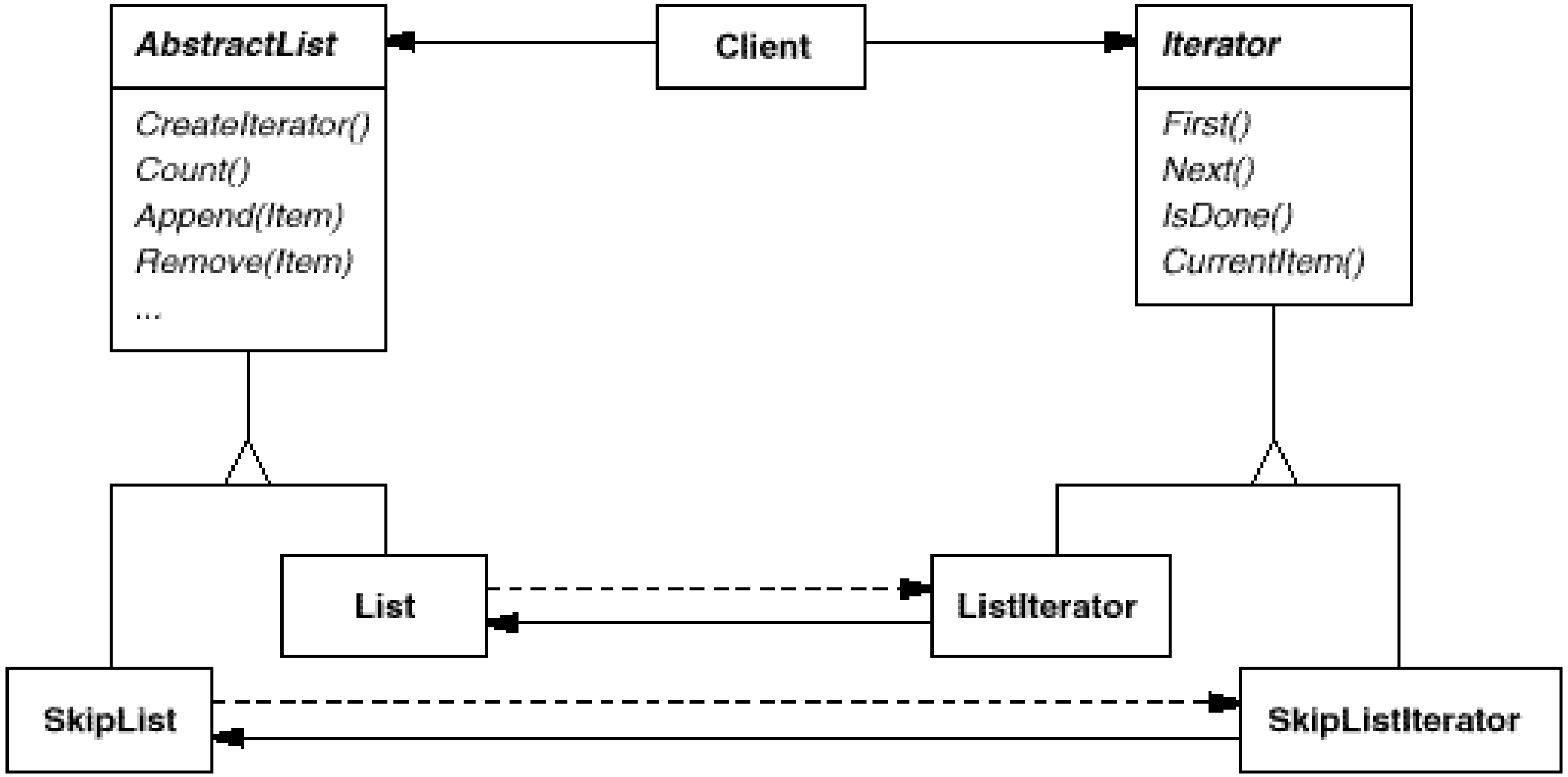
- Accessing the elements of an aggregate (Container) object without exposing its internal structure.
- Traverse the aggregate in different ways, depending on needs.
- Do not want to bloat the aggregate interface with operations for different traversals, even if they can be anticipated.
- Need to have more than one traversal pending on the same aggregate.
- Iterator pattern is utilised in the `java.util.Collection` interface via the `java.util.Iterator<E>` and `ListIterator<E>` interfaces



https://refactoring.guru/design-patterns/iterator

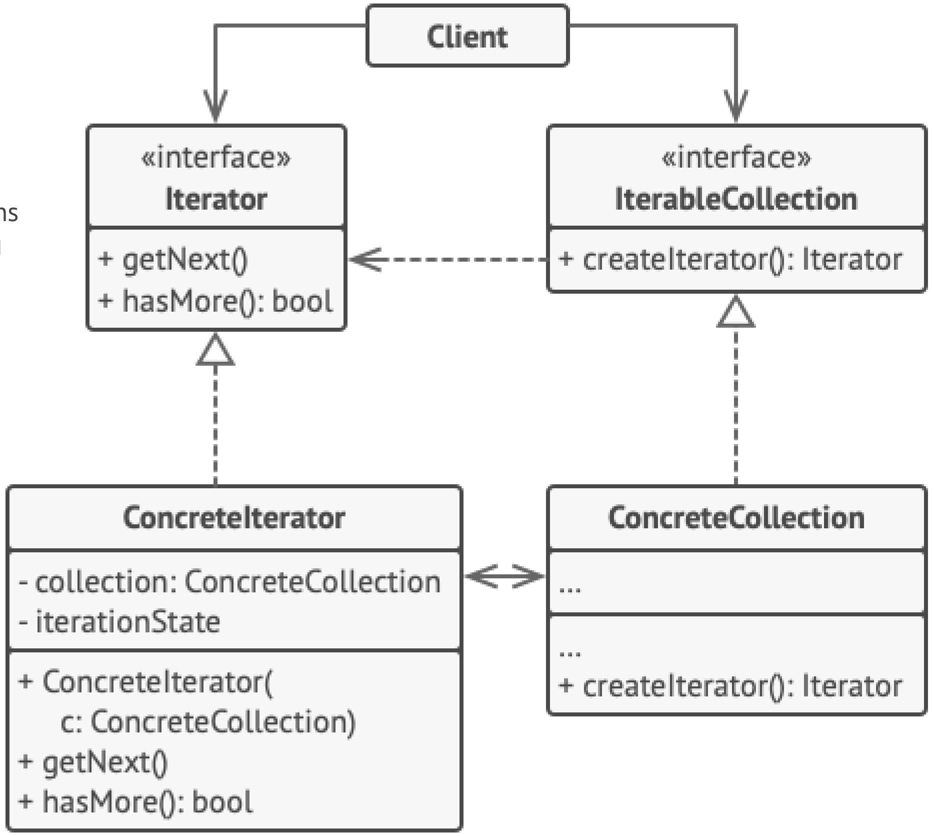
Iterator: Solution

- **Key idea:** Take the responsibility for access and traversal out of the aggregate object and put it into an *Iterator* object.
- The list objects are responsible for creating their corresponding iterator.



Iterator: Class Diagram

The **Iterator** interface declares the operations required for traversing a collection: fetching the next element, retrieving the current position, restarting iteration, etc.



Client
«interface» Iterator
+ getNext()
+ hasMore(): bool
«interface» IterableCollection
+ createIterator(): Iterator
ConcreteIterator
- collection: ConcreteCollection
- iterationState
+ ConcreteIterator(c: ConcreteCollection)
+ getNext()
+ hasMore(): bool
ConcreteCollection
...
...
+ createIterator(): Iterator

The **Collection** interface declares one or multiple methods for getting iterators compatible with the collection. Note that the return type of the methods must be declared as the iterator interface so that the concrete collections can return various kinds of iterators.

Concrete Iterators implement specific algorithms for traversing a collection. The iterator object should track the traversal progress on its own. This allows several iterators to traverse the same collection independently of each other

Concrete Collections return new instances of a particular concrete iterator class each time the client requests one. You might be wondering, where's the rest of the collection's code? Don't worry, it should be in the same class. It's just that these details aren't crucial to the actual pattern, so we're omitting them.

The **Client** works with both collections and iterators via their interfaces. This way the client isn't coupled to concrete classes, allowing you to use various collections and iterators with the same client code. Typically, clients don't create iterators on their own, but instead get them from collections. Yet, in certain cases, the client can create one directly; for example, when the client defines its own special iterator.

Iterator: Participants

- *Iterator*
  - defines an interface for accessing and traversing elements
- *ConcreteIterator*
  - implements the Iterator interface
  - keeps track of the current position in the traversal of the aggregate
- *Aggregate*
  - defines an interface for creating an Iterator object
- *ConcreteAggregate*
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Iterator: intent and context

- provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- apply *Iterator* for the following purposes:
  - to access an aggregate object's contents without exposing its internal representation
  - to support multiple traversals of aggregate objects
  - to provide a uniform interface for traversing <u>different</u> aggregate structures (support polymorphic iteration)
- see `iterator.books.*;` example