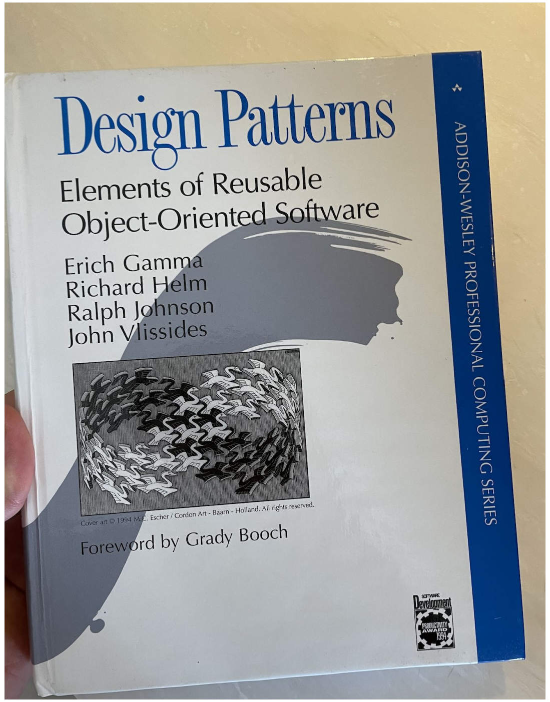
Refresher on OOP

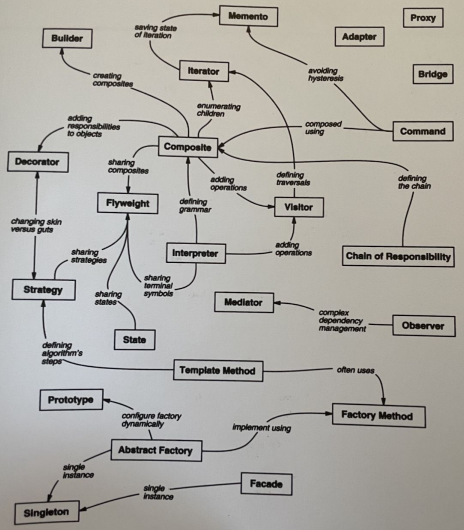
2021-08-31 @nikolay

Classic work on OOP

- Published in 1995
- Collection of tips from SWEs of 1980~1995
- Linux is 4y old
- Python is 4y old
- Web and first bowser is 1y old
- … 10y before git
- … 4 years before Google


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Patterns



Design Pattern Relationships

Creational

Some piece of construction varies

- Abstract Factory - construct families of objects (e.g. Windows windows, MacOS windows)
- Builder - construct single complex object
- Factory Method - subclasses decide how to implement single function, function is called in parent class internals
- Prototype - create instance first, and then copy it and modify when you need it. Prototype manager constructor called with concrete implementations, has abstract ones internally.
- Singleton - make single instance *without* shared global variable, rely on static class and private-public mechanisms

Structural

Some piece of structural relations between objects varies

These four are enclosures of single object

- Adapter - transform interface of adoptee into form acceptable by external clients
- Bridge - different implementations of same interface. You can do single abstract class and inheritance, but inheritance would binds interface to implementation forever. Bridge breaks it into `..Implementation` interface, so you have interface (Window), Implementation Interface (WindowImplementation), Concrete Implementation (MacOSWindowImplementation).
- Decorator - transparent enclosure, adds responsibilities to object dynamically. This is skin of object (as opposed to Strategy that is body). One decorator wraps other decorator wraps other decorator.
- Proxy - placeholder for some other object, has same interface. Can be resolved to original object later. (e.g. image that is being downloaded)

These structure multiple objects

- Composite - recursive structure, object that contains children has same interface as children
- Facade - single point of exit for system with many pieces. Make external clients to route requests through single point to all internally interviened systems.
- Flyweight - reuse objects transparently, allow efficient usage of resources. Important to differentiate *intrinsic* (stored in flyweight) and *extrinsic* state (dictated by context of where flyweight is called).

Behavioral

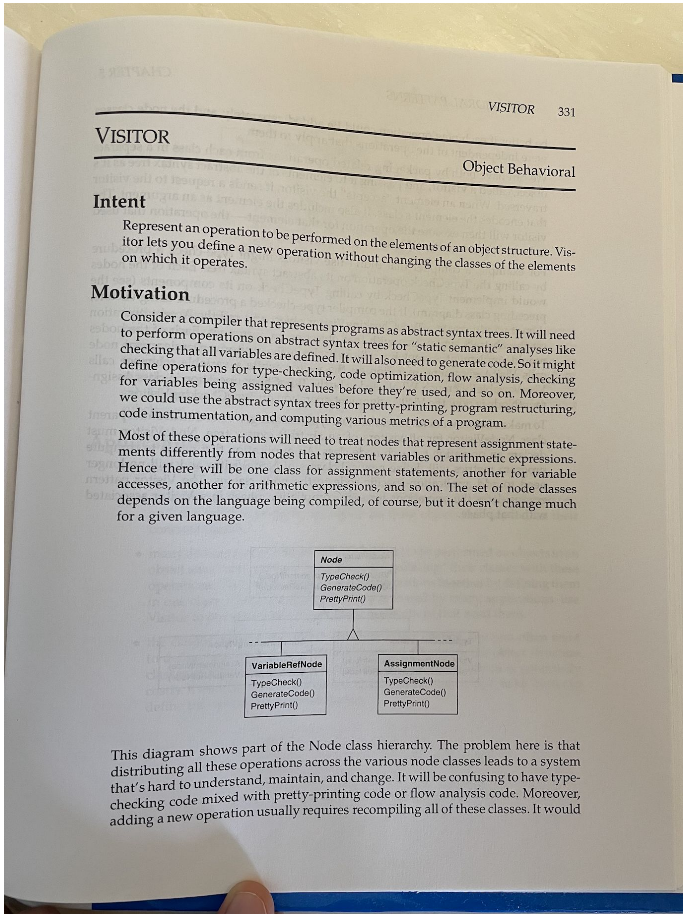
Some piece of behavior varies.

- Chain of responsibility - pass request from one handler to other. All handlers `Handle` with same argument. Handlers can pass handler without action. Chain of handlers need to be constructed by someone. Handlers can keep next handler inside of them. Only single handler is outgoing from each handler. No guarantee that any handler will run request.
- Command - single function call as an object. This is how `functional closure` could be implemented in C++. Binds arguments and who is going to execute it.
- Interpreter - duh, interpret language
- Iterator - traverse items one-by-one in different styles. `implicit iterator` = increments itself, you run it like `map(lambda x: x ** 2, [1, 2, 3])`. `explicit iterator` = need to be incremented, you run it as a for loop `for x := list.Start(); !x.Done(); x.Next()`.
- Mediator - when multiple objects talk to each other, to avoid `O(N^2)` complexity of iteration between `N` components, use single node to coordinate their interactions.
- Memento - extract private state from object. Make object possible to restore itself from private state. Note, client that extracts private state does not need to know what is inside. (e.g. binary of OS. I don't know what is inside, but OS can run it and restore *application* into its expected state. Binary is memento).
- Observer - publish-subscribe model, but with objects
- State - same entity, all states share common interface, but their logic differs in each state, can transition from one state to other. (e.g. TCP Connection).
- Strategy - encapsulate whole algorithm into object. You can do this in Go. Can call specific methods of algorithm.
- Template Method - child classes define how method has to be implemented. Parent class uses method. (Note how similar to Factory Method!).
- Visitor - make abstract visitor (visitor-A, visitor-B) that can be visited by *any* visitor, call each visitor function inside. Make abstract visitor (visitor-X, visitor-Y) that can visit any visitor, define function of visit for each item. Can add operations on visitors without need to change them much. You can use it with iterator. I too come up with this when working on my project, so it is very natural.

Covered Today

- Visitor
- Strategy
- Observer

Visitor



VISITOR

Object Behavioral

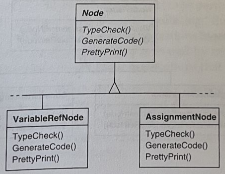
Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

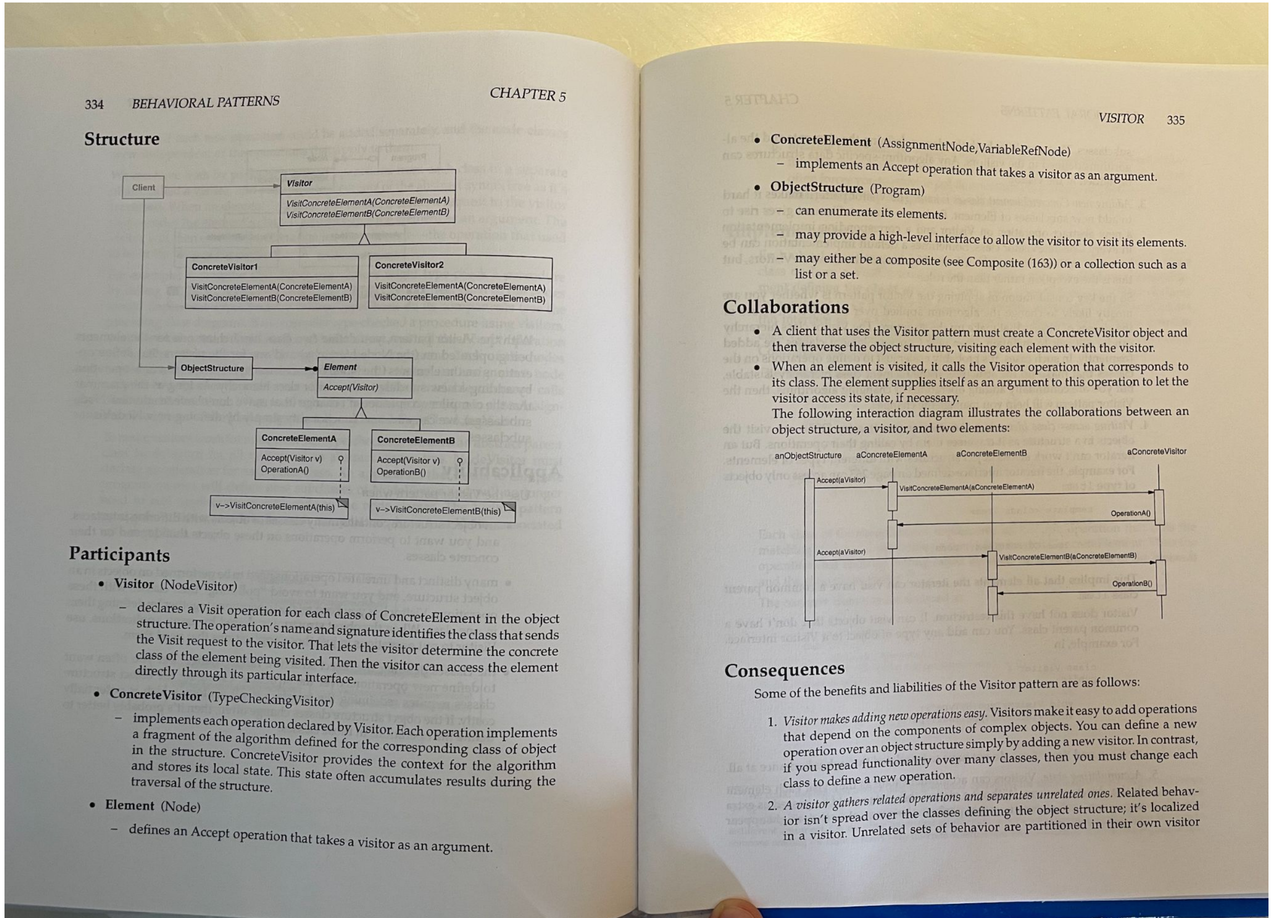
Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used, and so on. Moreover, we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics of a program.

Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions. Hence there will be one class for assignment statements, another for variable accesses, another for arithmetic expressions, and so on. The set of node classes depends on the language being compiled, of course, but it doesn't change much for a given language.

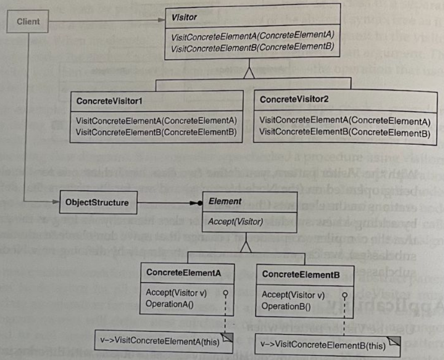


This diagram shows part of the Node class hierarchy. The problem here is that distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It will be confusing to have type-checking code mixed with pretty-printing code or flow analysis code. Moreover, adding a new operation usually requires recompiling all of these classes. It would

Visitor



Structure



Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Participants

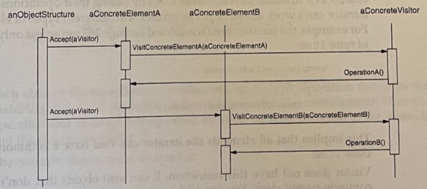
- **Visitor** (NodeVisitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element** (Node)
  - defines an Accept operation that takes a visitor as an argument.

- **ConcreteElement** (AssignmentNode,VariableRefNode)
  - implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure** (Program)
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite (see Composite (163)) or a collection such as a list or a set.

Collaborations

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

  The following interaction diagram illustrates the collaborations between an object structure, a visitor, and two elements:



Consequences

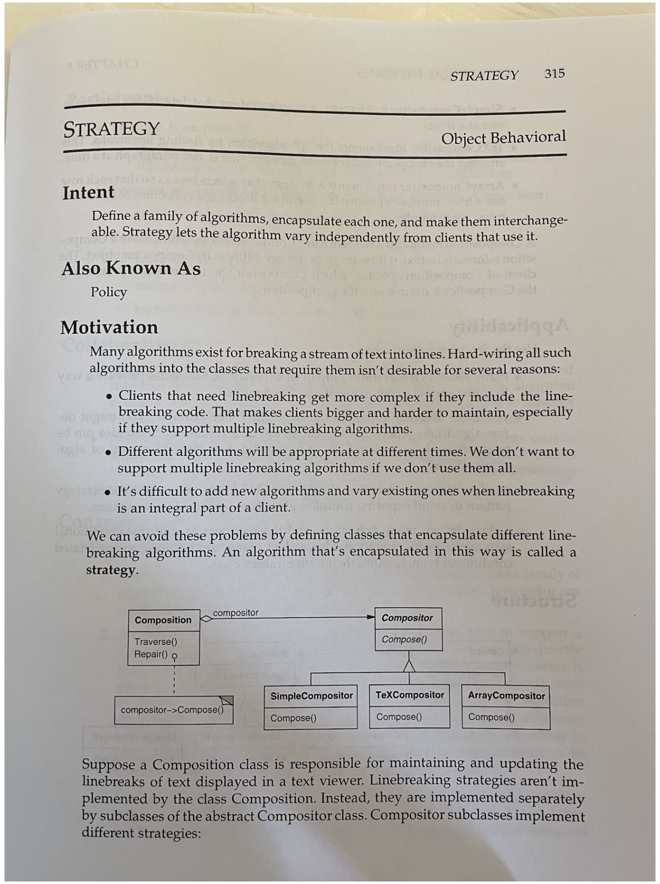
Some of the benefits and liabilities of the Visitor pattern are as follows:

1. *Visitor makes adding new operations easy.* Visitors make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor. In contrast, if you spread functionality over many classes, then you must change each class to define a new operation.
2. *A visitor gathers related operations and separates unrelated ones.* Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor

Visitor: Example

calendarheatmap.io

Strategy



STRATEGY

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

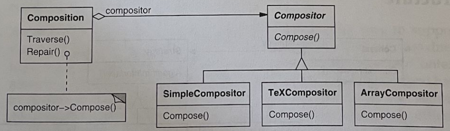
Policy

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

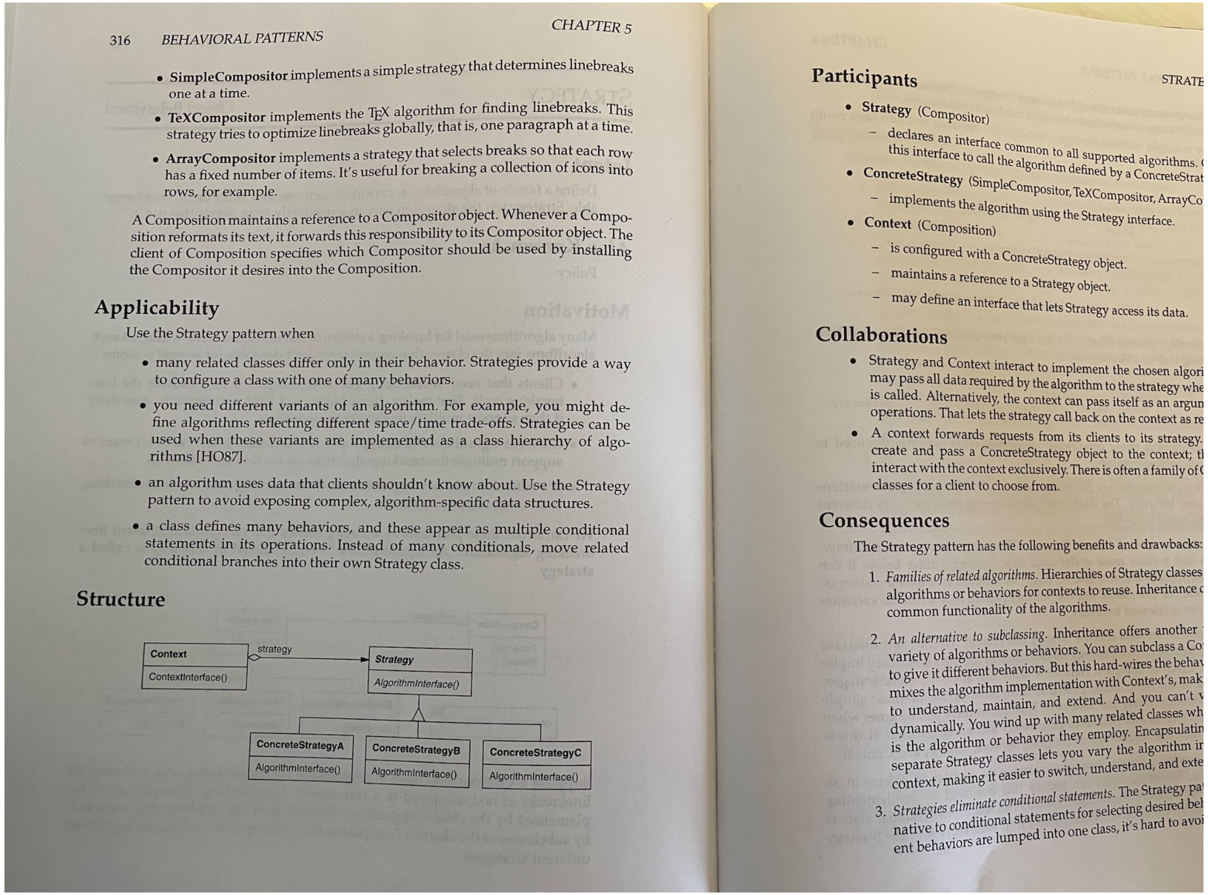
- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that's encapsulated in this way is called a **strategy**.



Suppose a Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class Composition. Instead, they are implemented separately by subclasses of the abstract Compositor class. Compositor subclasses implement different strategies:

Strategy



- **SimpleCompositor** implements a simple strategy that determines linebreaks one at a time.
- **TeXCompositor** implements the TEX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

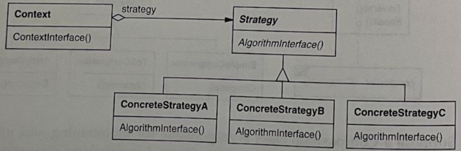
A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms [HO87].
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. (this interface to call the algorithm defined by a ConcreteStrat
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCo
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

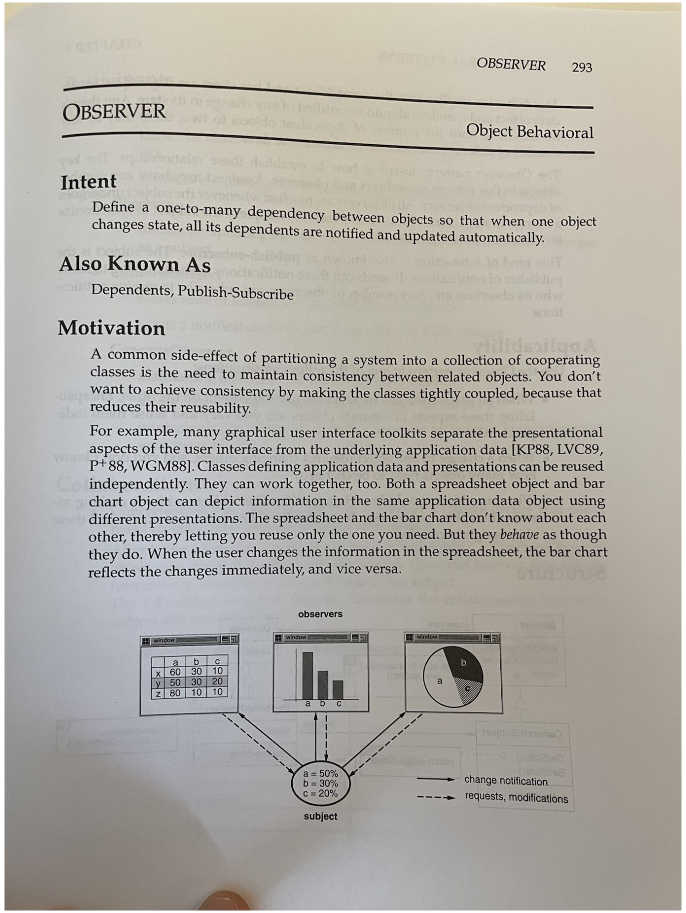
- Strategy and Context interact to implement the chosen algori may pass all data required by the algorithm to the strategy whe is called. Alternatively, the context can pass itself as an argum operations. That lets the strategy call back on the context as re
- A context forwards requests from its clients to its strategy. create and pass a ConcreteStrategy object to the context; tl interact with the context exclusively. There is often a family of (classes for a client to choose from.

Consequences

The Strategy pattern has the following benefits and drawbacks:

1. *Families of related algorithms.* Hierarchies of Strategy classes algorithms or behaviors for contexts to reuse. Inheritance c common functionality of the algorithms.
2. *An alternative to subclassing.* Inheritance offers another variety of algorithms or behaviors. You can subclass a Co to give it different behaviors. But this hard-wires the beha mixes the algorithm implementation with Context's, mak to understand, maintain, and extend. And you can't v dynamically. You wind up with many related classes wh is the algorithm or behavior they employ. Encapsulatin separate Strategy classes lets you vary the algorithm ir context, making it easier to switch, understand, and exte
3. *Strategies eliminate conditional statements.* The Strategy pa native to conditional statements for selecting desired bel ent behaviors are lumped into one class, it's hard to avoi

Observer



OBSERVER

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.

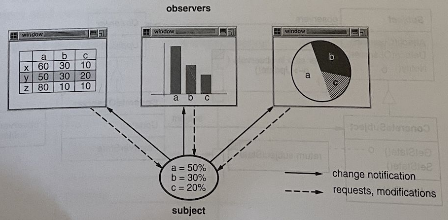
Also Known As

Dependents, Publish-Subscribe

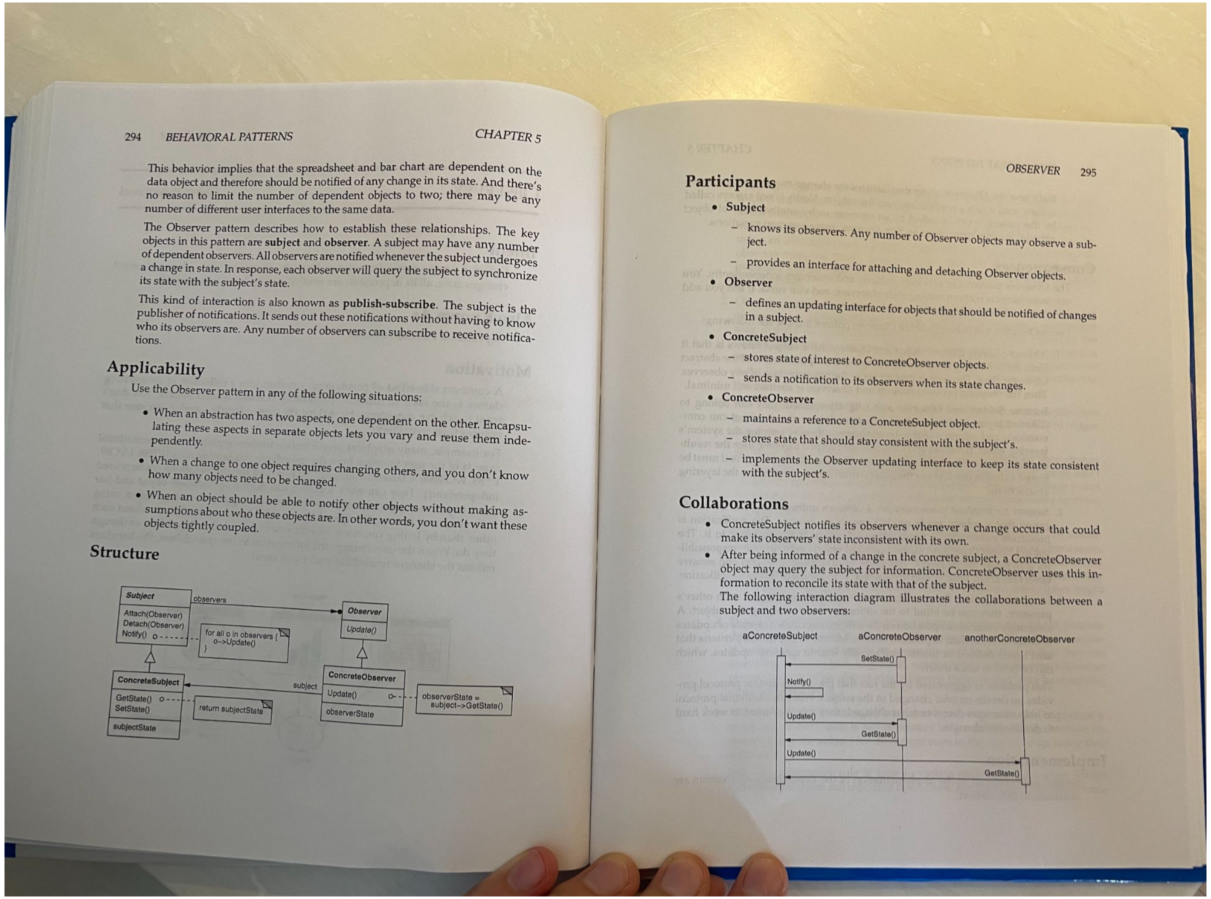
Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data [KP88, LVC89, P+88, WGM88]. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



Observer



This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.

The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

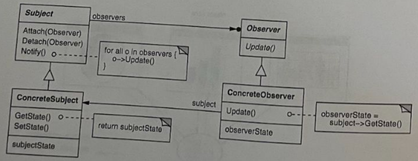
This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure



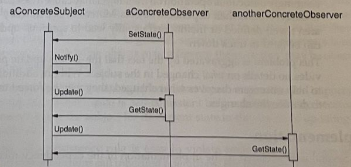
Participants

- **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.

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.

The following interaction diagram illustrates the collaborations between a subject and two observers:


aConcreteSubject
aConcreteObserver
anotherConcreteObserver
SetState()
Notify()
Update()
GetState()
Update()
GetState()