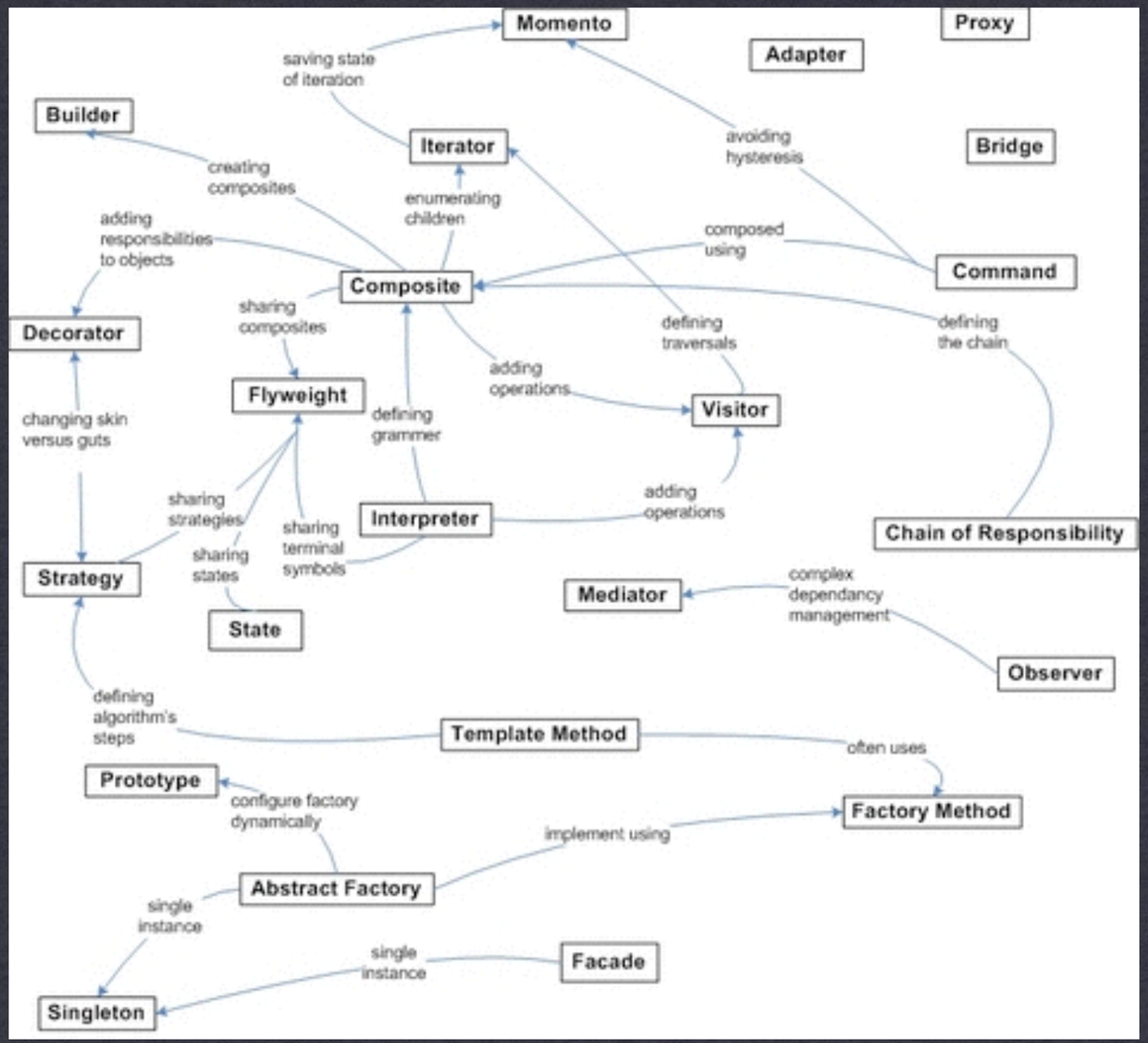
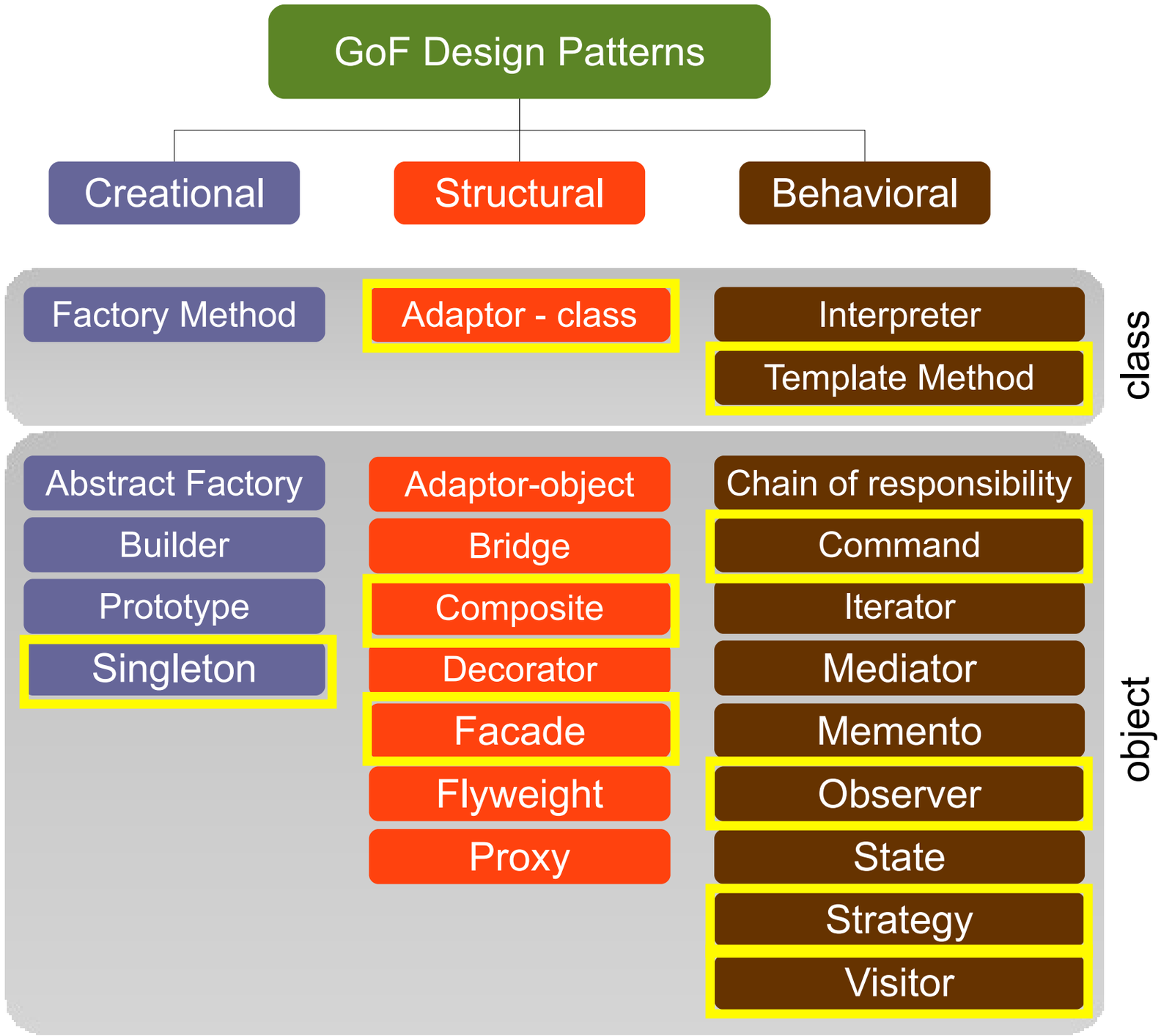


Material and some slide content from:
- Head First Design Patterns Book
- GoF Design Patterns Book

Design Patterns

Reid Holmes

GoF design patterns



Pattern vocabulary

- Shared vocabulary
  - communicate qualities
  - reduce verbosity
  - focus on design
  - increase understanding

Observer example

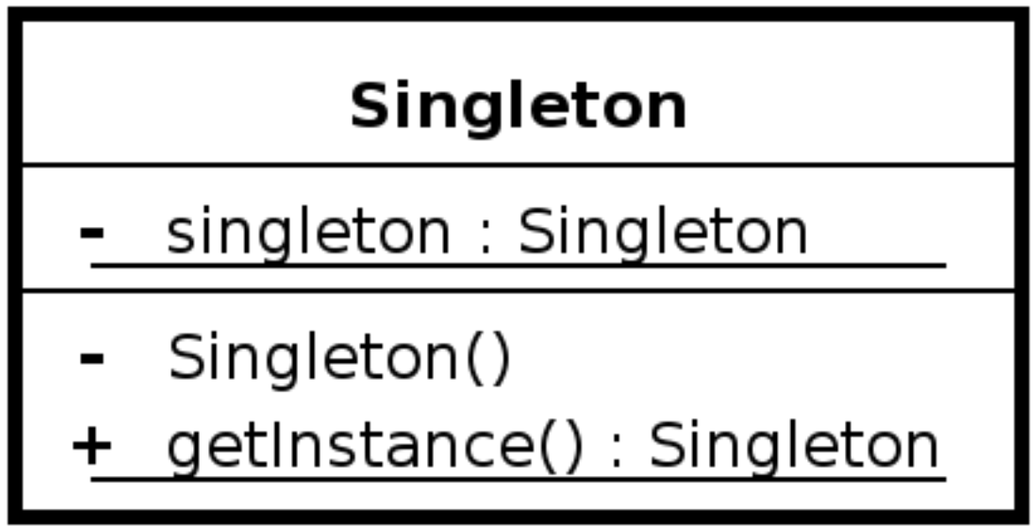
- Weather data example (similar to Eclipse example)
  - WeatherData
    - temp, humidity, pressure
    - calls newData() whenever something changes
      - bad: update views directly from here
  - WeatherViews
    - Current View
    - Forecast View
    - Stats View

Singleton

- Intent: “Ensure a class has only one instance”
- Motivation: For situations when having multiple copies of an object is either unnecessary or incorrect.
- Applicability:
  - Situations when there must be only one copy of a class.

Singleton

- Structure:



- Participants:
  - an instance operation that retrieves the instance.
  - may be responsible for creating instance.

Singleton

- Collaborations
  - All collaboration via instance operation.
- Consequences:
  - Controlled access to instance.
  - Reduced name space.
  - Permits variable number of instances.
  - More flexible than class operation

Singleton

- Implementation:

  1. Ensure a unique instance.
  2. Provide an easy access point.

- Related to:

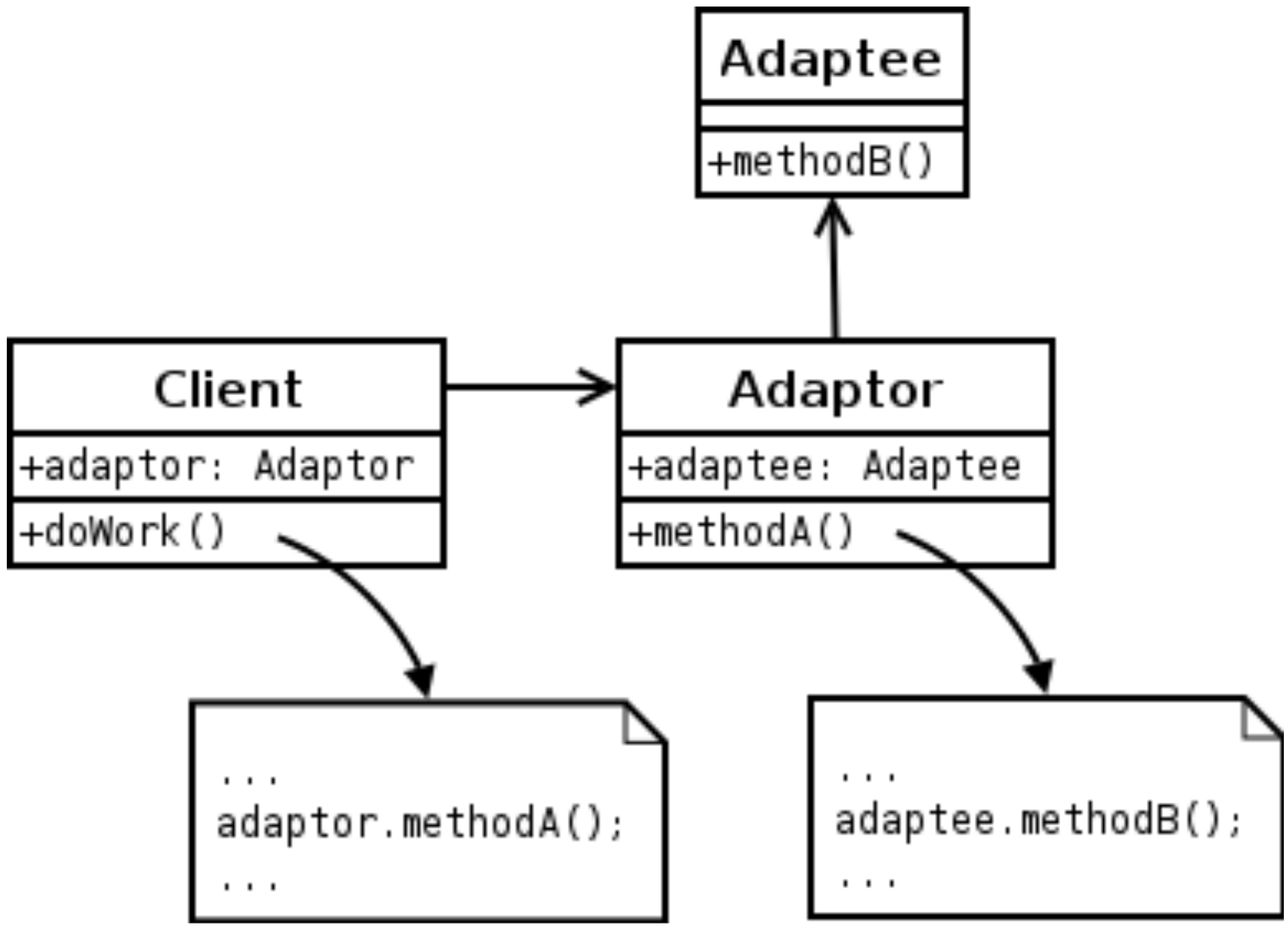
  - Can be used to create Abstract Factory, Builder, and Prototype.

Adapter

- Intent: “Convert the interface of one class to make it compatible with another class.”
- Motivation: Components often have incompatible interfaces that cannot be used by a client but also cannot be modified.

- Applicability:
  - When an intermediate representation between one component and another is required.
  - When you want to isolate a client from changes in an external component’s interface.

Adapter


Adaptee
+methodB()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+adaptee: Adaptee
+methodA()
...
adaptor.methodA();
...
...
adaptee.methodB();
...

Adapter

- Participants:
  - Client: Calls the adapter; is completely isolated from the adaptee.
  - Adapter: Forwards calls between client and adapter. The adapter may have to interact with multiple classes.
  - Adaptee: Component being adapted.
- Alternatives:
  - Both class and object adapters exist. Class adapters use multiple inheritance to be an instance of both objects simultaneously while object adaptors rely on composition.

Adapter

- Implementation:
  - 1) Client calls adapter.
  - 2) Adapter marshals calls and interacts with adaptee.

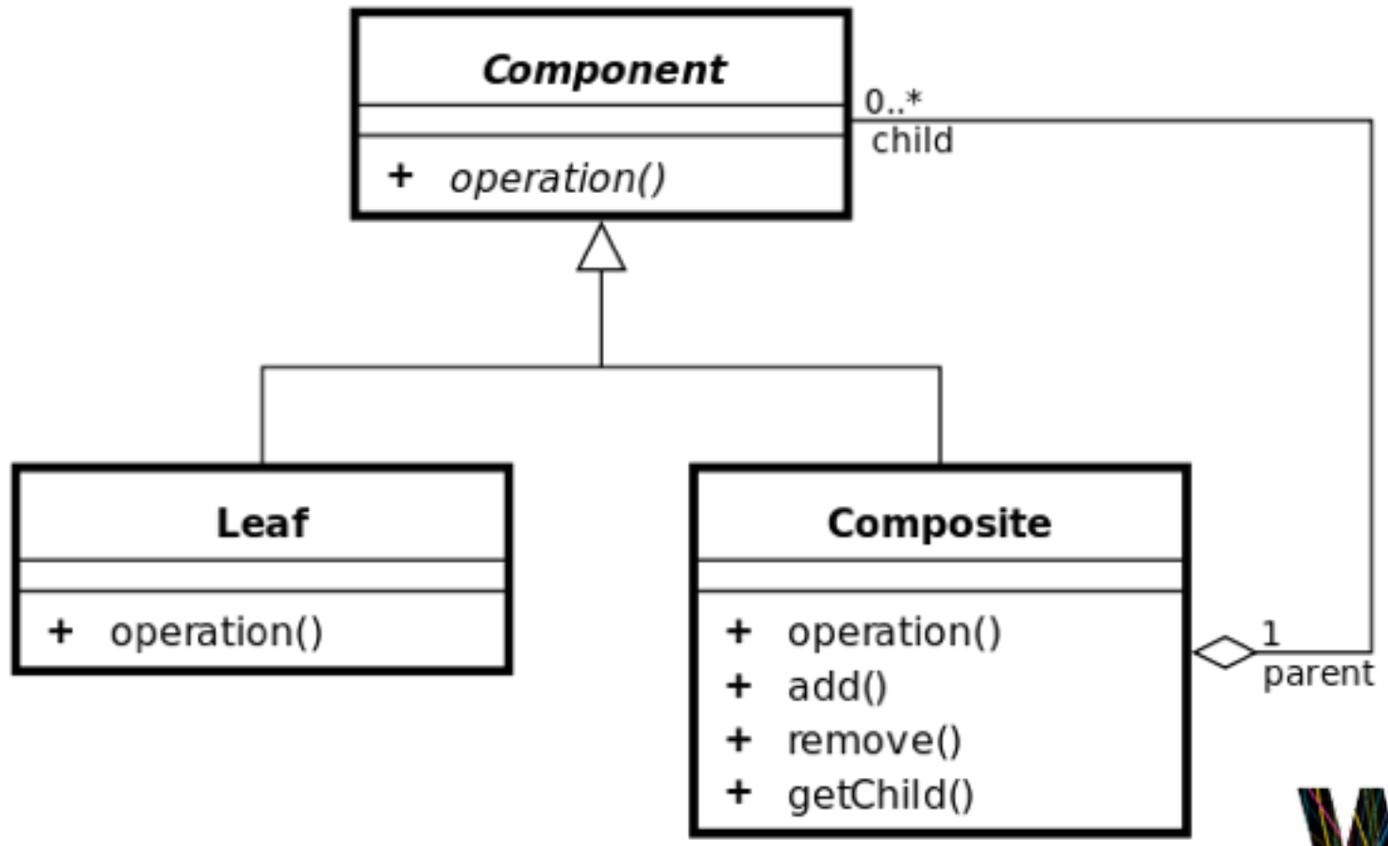
- Related to: Bridge is an *a priori* version of the adapter pattern. The object-based Adapter heavily leverages delegation. Facades define new interfaces simplifying existing modules.

Composite

- Intent: “Enable a group of objects to be treated as single object”
- Motivation: Differentiating between interior and leaf nodes in tree-structured data data increases system complexity.
- Applicability:
  - If you notice you are treating groups and individual of objects the same way
  - Can also be used when primitives and objects need to be treated identically

Composite

- Participants:
  - Component: base class
  - Leaf: individual leaf node
  - Composite: node that maintains a list of children nodes



Composite

- Implementation:
  - 1) Composite maintains a list of child elements and methods to maintain the children.
  - 2) Composite object applies overridden methods from the component across all child methods.
- Known uses:
- Related to: Decorators are often used along with the Composite pattern to augment objects while grouping them.

Facade

- Intent: “Provide a unified, higher-level, interface to a whole module making it easier to use.”
- Motivation: Composing classes into subsystems reduces complexity. Using a Facade minimizes the communication dependencies between subsystems.
- Applicability:
  - When you want a simple interface to a complex subsystem.
  - There are many dependencies between clients and a subsystem.
  - You want to layer your subsystems.

Facade

- Participants:
  - Facade
  - Subsystem classes
- Collaborations:
  - Clients interact subsystem via Facade.
- Consequences:
  - Shields clients from subsystem components.
  - Promotes weak coupling. (strong within subsystem, weak between them)
  - Doesn’t prevent access to subsystem classes.

Facade

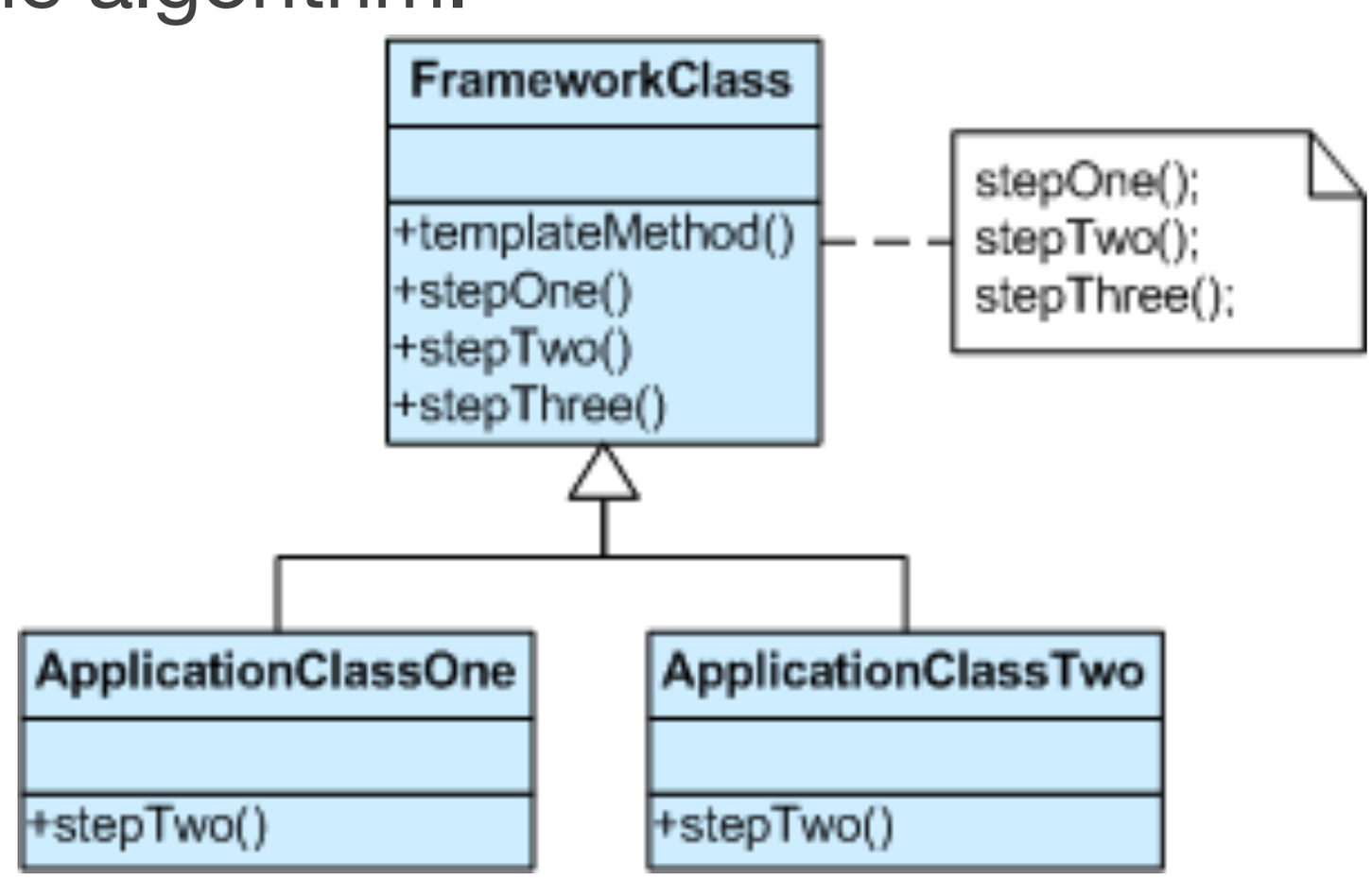
- Implementation:
  - 1) Analyze client / subsystem tangling.
  - 2) Create interface. Abstract factories can also be used to add further decoupling.
- Known uses: Varied.
- Related to: Abstract Factory can be used with Facade to create subsystem objects. Facades are frequently Singletons. Abstracts functionality similar to Mediator but does not concentrate on communication.

Template Method

- Intent: “Define the skeleton of an algorithm deferring some steps to client subclasses”
- Motivation: When two algorithms are largely the same but differ in only a few small details the algorithm can be encapsulated in a base class and defer specific functionality to child classes.
- Applicability:
  - Template method implements invariant parts of the algorithm.
  - Base class contains the majority of functionality, reducing code duplication.

Template Method

- Participants:
  - Base class, defining common functionality in concrete method making calls to abstract ‘hook’ methods.
  - Sub classes, overriding subclass-specific parts of the algorithm.



Template Method

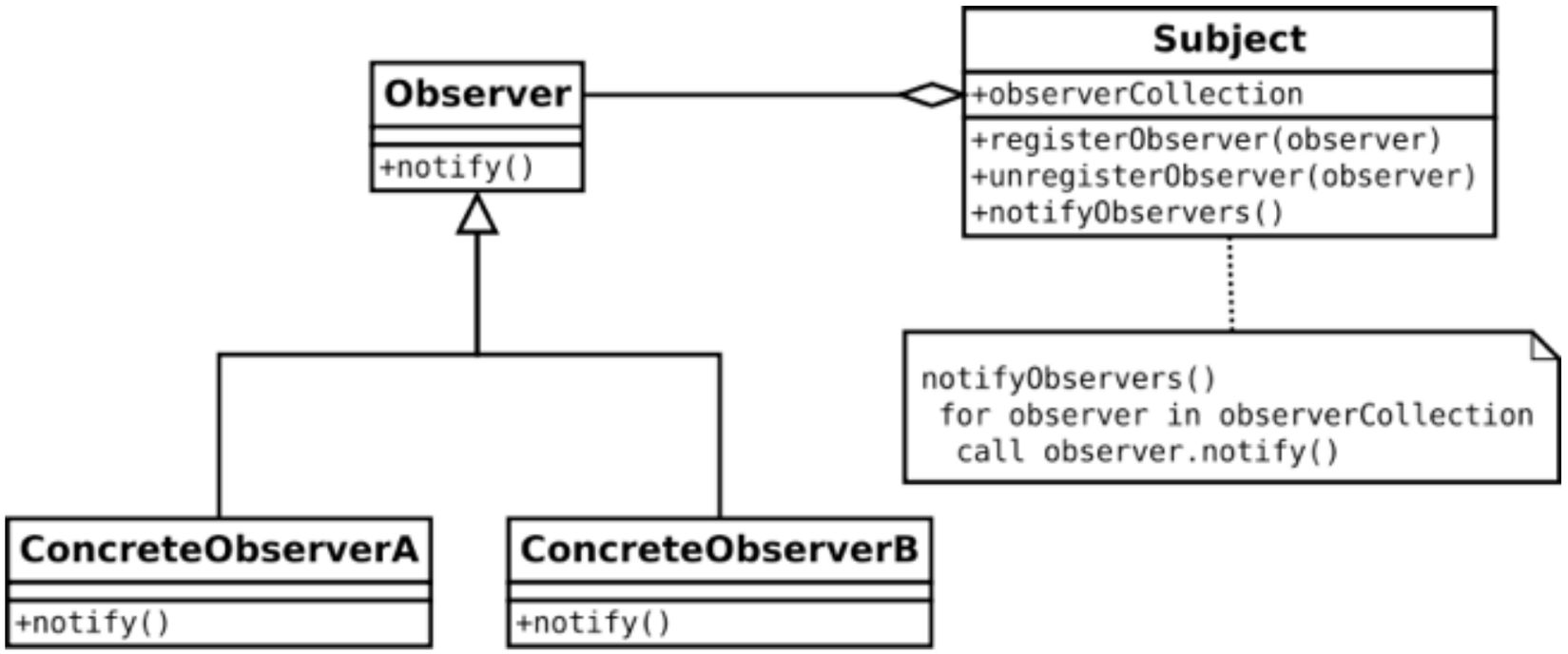
- Implementation:
  - 1) Define template method in base class implementing skeleton of algorithm that calls hook methods (which can be either abstract or provide default implementations).
  - 2) Subclasses provide the hook methods that are relevant to them.
  - 3) Base class prevents template method from being overridden (e.g., declare it final).
- Related to: Strategy is a composition-based version of the template method (which is inheritance-based).

Observer

- Intent: Define a one-to-many relationship between objects so that when an object changes state its dependents are updated automatically
- Motivation: To maintain consistency between multiple different objects without tightly coupling them
- Applicability:
  - When you want to compartmentalize modifications to two dependent objects
  - When you want to publish updates but not couple classes

Observer

- Structure:



- Participants:
  - Subject: tracks observers and fires updates
  - Observer: subscribes/unsubscribes to subjects, receives updates

Observer

- Collaborations
  - Subjects call observer's update method when they change
  - Subjects can forward data (push) or just send blank update notifications (pull)
- Consequences:
  - Reduce coupling between subject & observer
  - Support broadcast communication
  - Can result in expensive updates

Observer

- Implementation:
  1. Subjects track observers (abstract class helpful)
  2. Caching updates
  3. Push vs. pull
- Related to:
  - Employed by MVC & MVP.

GWT example

```
Window.addResizeHandler(new ResizeHandler() {
        @Override
        public void onResize(ResizeEvent event) {
           if (event.getWidth() > event.getHeight()) {
              setPortrait(false);
           } else {
              setPortrait(true);
           }
        }
     });
```

Command

- Intent: “Encapsulate requests enabling clients to log / undo them as required.”
- Motivation: In situations where you need to be able to make requests to objects without knowing anything about the request itself or the receiver of the request, the command pattern enables you to pass requests as objects.
- Applicability:
  - Parameterize requests.
  - Specify, queue, and log actions.
  - Support undo.
  - Model high-level operations on primitive operations.

Command

- Structure

- Participants:
  - Command / ConcreteCommand
  - Client
  - Invoker
  - Receiver

Command

- Collaborations:
  - Client creates ConcreteCommand and specifies receiver.
  - Invoker stores ConcreteCommand object.
  - Invoker requests execute on Command; stores state for undoing prior to execute (if undoable).
  - Concrete invokes operations on its receiver to perform request.
- Consequences:
  - Decouples the invoker from the object that knows how to perform an action.
  - Commands are first-class objects.
  - Commands can be assembled into composite.
  - Adding new commands is easy.

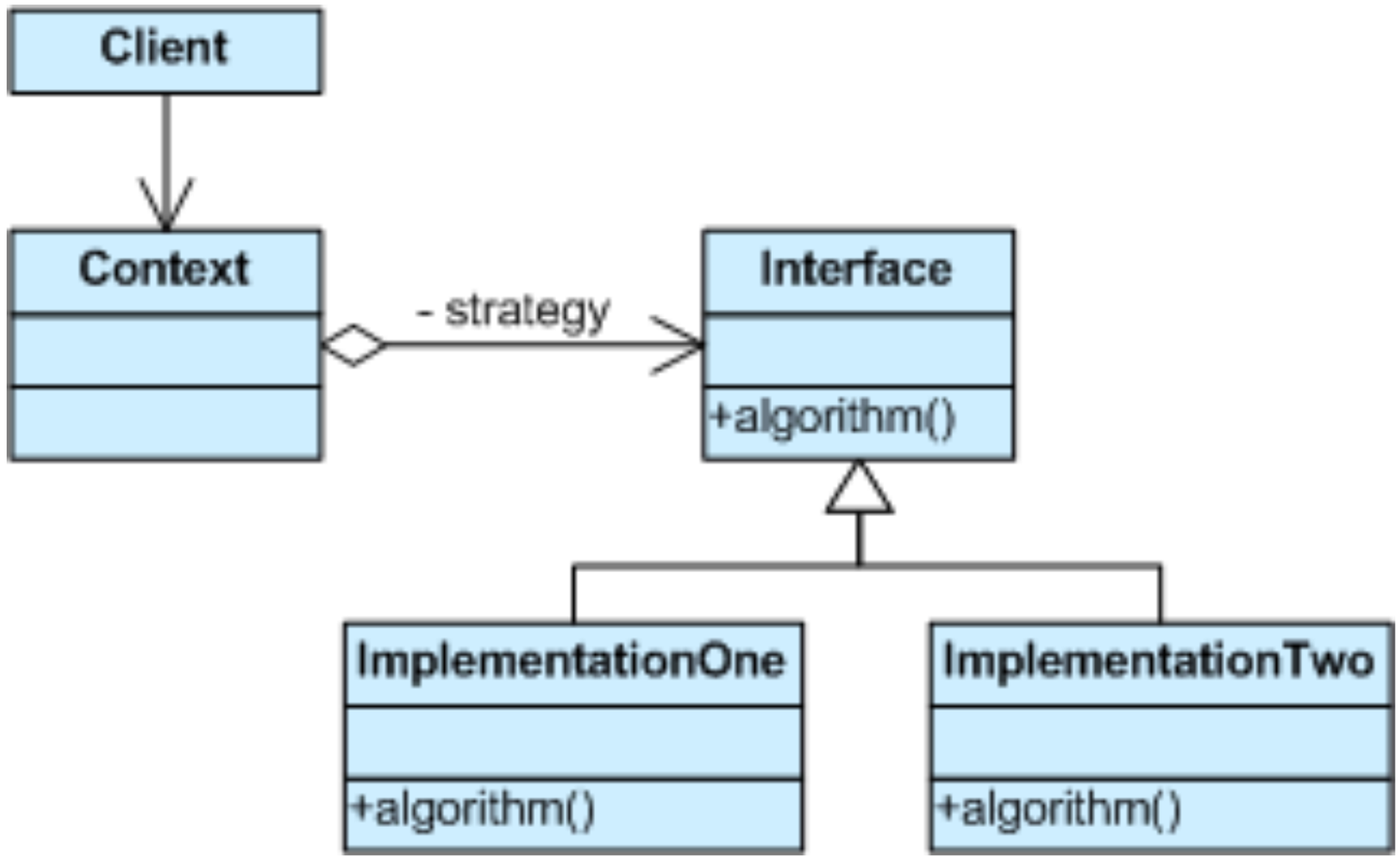
Command

- Implementation:
  - 1) How smart should a command be?
  - 2) Support undo/redo.
  - 3) Avoiding error accumulation in the undo process.
- Related to: Composite commands can be created; the Memento pattern can store undo state. Commands often use Prototype when they need to be stored for undo/redo.

Strategy

- Intent: “Define a family of algorithms that can be easily interchanged with each other”
- Motivation: Support the open/closed principle by abstracting algorithms behind an interface; clients use the interface while subclasses provide the functionality and can be easily interchanged.
- Applicability:
  - When you want to be able to replace a behaviour at runtime (strategy reference can be dynamically altered).
  - When you want to have a family of behaviours that might not be applicable for the client class.

Strategy


Client
Context
- strategy
Interface
+algorithm()
ImplementationOne
+algorithm()
ImplementationTwo
+algorithm()

Strategy

- Participants:
  - Context contains reference to chosen strategy and invokes algorithm.
  - Strategy interface declares algorithm structure.
    - ConcreteStrategy implements algorithm. Context does not use any methods not defined in the Strategy interface.
- Consequences:
  - Context uses the interface, not the concrete class.
  - Concrete classes can be easily exchanged.

Strategy

- Implementation:
  - 1) Create strategy interface and reference in client.
  - 2) Interact only through interface.

- Related to: Abstract Factory uses inheritance to achieve a subset of what Strategy does though a composition-based behaviour. Decorators are similar but focus on extending the functionality of a base class.

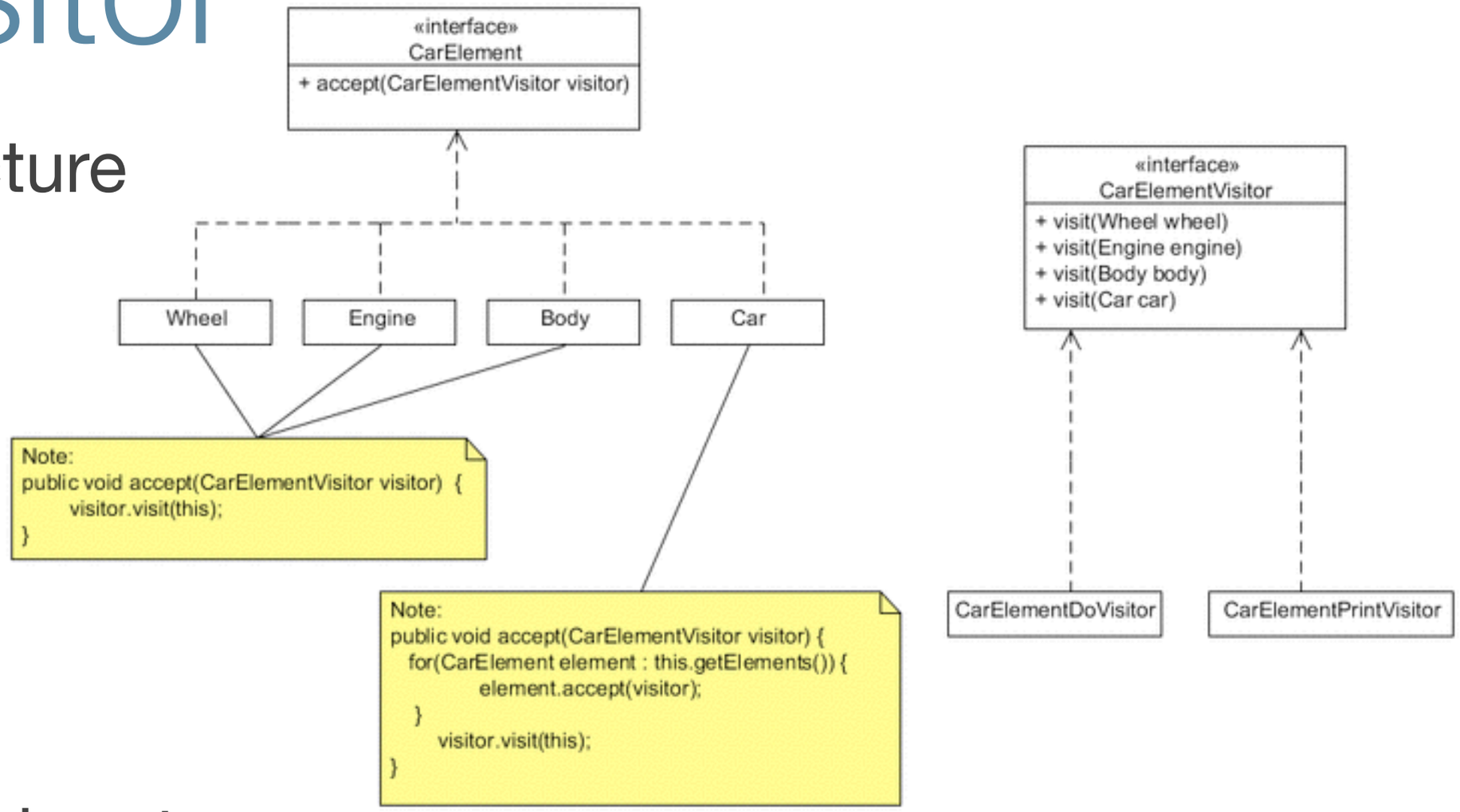
Visitor

- Intent: “Represent operations to be performed on classes of elements.”
- Motivation: Consider a large tree of objects that you want to perform an analysis on; this could require changing many objects. Visitors enable these

- Applicability:
  - When you have a large object structure you want to traverse.
  - When you have many different operations you want to perform but don’t want to pollute the objects.
  - The objects rarely change but the operations may.

Visitor

- Structure



- Participants:
  - Visitor / ConcreteVisitor
  - Element / ConcreteElement
  - ObjectStructure

Visitor

- Collaborations:
  - Client creates the ConcreteVisitor that traverses the object structure.
  - The visited object calls its corresponding visitor method on the ConcreteVisitor.
- Consequences:
  - Adding new operations is easy.
  - Visitor gathers related operations (and separates unrelated ones).
  - Adding ConcreteElement classes is hard.
  - Accumulating state.
  - Negative: Breaking encapsulation. (visitor may need access to internal state)

Visitor

- Implementation:
  - 1) Double dispatch.
  - 2) Who traverses structure?
- Related to: Good at visiting Composite structures.