Step 5. John sends a message in the chat room:

```
john.say("Hello, everyone!");
```

Step 6. Creation of a new person and joining them to the chat room:

```
const doe = new Person("Doe", chatRoom);

chatRoom.joinPerson(doe);
```

Step 7. Doe sends a message in the chat room:

```
doe.say("Hi, everyone!");
```

Whenever someone sends a message, the chat room takes care of distributing that message to all the other people in the room.

Final Code:

```
// Definition of the Person class, which represents people in the chat
room

class Person {

  constructor(name, chatRoom) {

    this.name = name;         // Person's name

    this.chatRoom = chatRoom; // Chat room they belong to

  }


  // Method to send a message in the chat room

  say(message) {

    this.chatRoom.sendMessage(this, message);

  }


  // Method to receive a message

  receive(message) {

    console.log(`${this.name} receives: ${message}`);

  }

}
```

```
// Definition of the ChatRoom class, which acts as a mediator
class ChatRoom {
  constructor() {
    this.people = []; // Stores people in the chat room
  }

  // Method to add a person to the chat room
  joinPerson(person) {
    this.people.push(person); // Add the person to the list
    person.chatRoom = this;  // Set the person's chat room as this room
  }

  // Method to send a message to all people in the room
  sendMessage(sender, message) {
    for (const person of this.people) {
      if (person !== sender) {
        person.receive(`${sender.name}: ${message}`);
      }
    }
  }
}

// Creation of a chat room
const chatRoom = new ChatRoom();

// Creation of people and joining them to the chat room
const john = new Person("John", chatRoom);
const tony = new Person("Tony", chatRoom);

chatRoom.joinPerson(john);
chatRoom.joinPerson(tony);

// John sends a message in the chat room
john.say("Hello, everyone!");

// Creation of a new person and joining them to the chat room
const doe = new Person("Doe", chatRoom);
chatRoom.joinPerson(doe);

// Doe sends a message in the chat room
doe.say("Hi, everyone!");
```

Memento

The Memento pattern allows restoring an object to its previous state.

The Memento pattern is a software design pattern that provides the capability to restore an object to a previous state. It is implemented using three objects: the originator, the caretaker, and the memento.

Example:

Let's take an example of a bank account in which we store our previous state and have the functionality to undo.

Step 1. Definition of the Memento class:

```
class Memento {
  constructor(balance) {
    this.balance = balance;
  }
}
```

In this step, we create the Memento class, which will be used to store the state of the bank account at a given moment.

Step 2. Definition of the CuentaBanco class:

```
class BankAccount {
  constructor(balance = 0) {
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
    return new Memento(this.balance);
  }

  restore(memento) {
    this.balance = memento.balance;
  }

  toString() {
    return `Balance: ${this.balance}`;
  }
}
```

Here, we create the BankAccount class, which represents a bank account with the ability to make deposits, save states in the form of Mementos, and restore the previous state.

Step 3. Definition of the Caretaker class:

```
class Caretaker {
  constructor() {
    this.mementos = [];
  }

  addMemento(memento) {
    this.mementos.push(memento);
  }

  getMemento(index) {
    return this.mementos[index];
  }
}
```

The CareTaker class is responsible for storing the Mementos in a list so that previous states can be restored when needed.

Step 4. Using the Memento pattern:

```
// Create an instance of BankAccount
const bankAccount = new BankAccount(100);
const caretaker = new Caretaker();
```

```
// Make deposits and save states in Mementos
caretaker.addMemento(bankAccount.deposit(50));
console.log(bankAccount.toString()); // Balance: 150

caretaker.addMemento(bankAccount.deposit(25));
console.log(bankAccount.toString()); // Balance: 175

// Restore the previous state
bankAccount.restore(caretaker.getMemento(0));
console.log(bankAccount.toString()); // Balance: 150

bankAccount.restore(caretaker.getMemento(1));
console.log(bankAccount.toString()); // Balance: 175
```

In this step, we create an instance of BankAcccount and a CareTaker. Then, we make deposits into the bank account and save states as Mementos using the CareTaker. Finally, we restore previous states of the bank account using the Mementos and verify the resulting balances.

Final Code:

```
class Memento {
  constructor(balance) {
    this.balance = balance;
  }
}

class BankAccount {
  constructor(balance = 0) {
    this.balance = balance;
  }
  deposit(amount) {
    this.balance += amount;
    return new Memento(this.balance);
  }

  restore(memento) {
    this.balance = memento.balance;
  }

  toString() {
    return `Balance: ${this.balance}`;
  }
}

class Caretaker {
  constructor() {
    this.mementos = [];
  }

  addMemento(memento) {
    this.mementos.push(memento);
  }

  getMemento(index) {
    return this.mementos[index];
  }
}

// Create an instance of BankAccount
const bankAccount = new BankAccount(100);
const caretaker = new Caretaker();

// Make deposits and save states in Mementos
caretaker.addMemento(bankAccount.deposit(50));
console.log(bankAccount.toString()); // Balance: 150

caretaker.addMemento(bankAccount.deposit(25));
console.log(bankAccount.toString()); // Balance: 175

// Restore the previous state
bankAccount.restore(caretaker.getMemento(0));
console.log(bankAccount.toString()); // Balance: 150

bankAccount.restore(caretaker.getMemento(1));
console.log(bankAccount.toString()); // Balance: 175
```

Observer

Allows multiple observer objects to watch an event.

The Observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and automatically notifies them of any state changes, usually by calling one of their methods.

Example:

Let's take an example of a person where if a person gets sick, it shows a notification.

Step 1. Definition of classes and the Observer Pattern: In this step, we define three key classes: Event, GetSick, and Person. The Event class represents an observable event and contains the logic for subscribing, unsubscribing, and notifying observers.

The GetSick class is a specific event that contains information about the location where it occurs. The Person class is the observable subject that can "get sick" and notify observers when this happens.

```
// Event class representing an observable event

class Event {

  constructor() {

    this.handlers = new Map();

    this.counter = 0;

  }


  // Method to subscribe a handler to the event and return a unique
identifier

  subscribe(handler) {

    this.handlers.set(++this.counter, handler);

    return this.counter;

  }
  // Method to unsubscribe from an event given an identifier
  unsubscribe(index) {
    this.handlers.delete(index);
  }

  // Method to notify all observers when the event occurs
  fire(sender, args) {
    this.handlers.forEach((handler, id) => handler(sender, args));
  }
}

// GetSick class representing a specific event
class GetSick {
  constructor(location) {
    this.location = location;
  }
}

// Person class that will be the observable subject
class Person {
  constructor(location) {
    this.location = location;
    this.getSick = new Event(); // Create a "get sick" event
  }

  // Method to simulate the person getting sick and notify observers
  catchCold() {
    this.getSick.fire(this, new GetSick(this.location));
  }
}
```

Step 2. Using the Observer Pattern: In this step, we create an instance of Person and then subscribe an observer to the "get sick" event using the subscribe method of the Event class. We then simulate the person getting sick twice by calling the catchCold method. After that, we unsubscribe the observer using the unsubscribe method. Finally, we call the catchCold method again, but the unsubscribed observer no longer receives notifications.

This pattern allows multiple observers to watch an event (in this case, getting sick) without the Person class needing to know the details of who the observers are or how they react to the event.

```
// Create an instance of the person
let person = new Person("Route ABC");

// Subscribe an observer to the "get sick" event
let sub = person.getSick.subscribe((subject, event) => {
  console.log(`A doctor has been called to ${event.location}`);
});

// The person gets sick twice
person.catchCold();
person.catchCold();

// Unsubscribe the observer
person.getSick.unsubscribe(sub);

// The person gets sick again, but the unsubscribed observer no longer
receives the notification
person.catchCold();
```

Final Code:

```
// Definition of classes and the Observer Pattern
// Event class representing an observable event
class Event {
  constructor() {
    this.handlers = new Map();
    this.counter = 0;
  }

  // Method to subscribe a handler to the event and return a unique
identifier
  subscribe(handler) {
    this.handlers.set(++this.counter, handler);
    return this.counter;
  }

  // Method to unsubscribe from an event given an identifier
  unsubscribe(index) {
    this.handlers.delete(index);
  }

  // Method to notify all observers when the event occurs
  fire(sender, args) {
    this.handlers.forEach((handler, id) => handler(sender, args));
  }
}

// GetSick class representing a specific event
class GetSick {
  constructor(location) {
    this.location = location;
  }
}

// Person class that will be the observable subject
class Person {
  constructor(location) {
    this.location = location;
    this.getSick = new Event(); // Create a "get sick" event
  }

  // Method to simulate the person getting sick and notify observers
  catchCold() {
    this.getSick.fire(this, new GetSick(this.location));
  }
}

// Using the Observer Pattern

// Create an instance of the person
let person = new Person("Route ABC");

// Subscribe an observer to the "get sick" event
let sub = person.getSick.subscribe((subject, event) => {
  console.log(`A doctor has been called to ${event.location}`);
});

// The person gets sick twice
person.catchCold();
person.catchCold();
```

```
// Unsubscribe the observer

person.getSick.unsubscribe(sub);


// The person gets sick again, but the unsubscribed observer no longer
receives the notification

person.catchCold();
```

Visitor

Add operations to objects without having to modify them.

The visitor design pattern is a way to separate an algorithm from an object structure it operates on. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

Example:

Let's take an example of the NumericExpression class that gives us the result of the given expression.

1. NumericExpression Class: This class represents a simple numeric expression with a value.

```
class NumericExpression {
  constructor(value) {
    this.value = value;
  }

  print(buffer) {
    buffer.push(this.value.toString());
  }
}
```

2. SumExpression Class: This class represents a sum expression that takes two numeric expressions (left and right) and adds them.

```
class SumExpression {
  constructor(left, right) {
    this.left = left;
    this.right = right;
  }
  print(buffer) {
    buffer.push('(');
    this.left.print(buffer);
    buffer.push('+');
    this.right.print(buffer);
    buffer.push(')');
  }
}
```

3. Using the Visitor Pattern: Next, you create a composite expression that represents "5 + (1 + 9)" using instances of the NumericExpression and SumExpression classes.

```
let e = new SumExpression(
    new NumericExpression(5),
    new SumExpression(
        new NumericExpression(1),
        new NumericExpression(9)
    )
);
```

Printing the Expression: Finally, you use the print() method to get a readable representation of the expression and store it in a buffer.

```
let buffer = [];
e.print(buffer);
console.log(buffer.join('')); // Output: (5+(1+9))
```

This example illustrates how the visitor design pattern allows you to add the printing operation to complex objects without having to modify their internal structure. This makes the code more flexible and extensible for future operations without affecting existing classes.

Final Code:

```
class NumericExpression {
  constructor(value) {
    this.value = value;
  }

  print(buffer) {
    buffer.push(this.value.toString());
  }
}

class SumExpression {
  constructor(left, right) {
    this.left = left;
    this.right = right;
  }

  print(buffer) {
    buffer.push('(');
    this.left.print(buffer);
    buffer.push('+');
    this.right.print(buffer);
    buffer.push(')');
  }
}

// Creating a composite expression: 5 + (1 + 9)
let e = new SumExpression(
    new NumericExpression(5),
    new SumExpression(
        new NumericExpression(1),
        new NumericExpression(9)
    )
```

```
);

let buffer = [];
e.print(buffer);

console.log(buffer.join('')); // Expected Output: (5+(1+9))
```

The expected output is (5+(1+9)), which is the representation of the "5 + (1 + 9)" expression after printing it using the visitor pattern. This code demonstrates how the visitor pattern allows adding the printing operation to complex objects without modifying their internal structure.

Strategy

Allows selecting one of the algorithms in certain situations.

The strategy pattern is a behavioral software design pattern that allows selecting an algorithm at runtime. Instead of implementing a single algorithm directly, the code receives instructions at runtime on which one to use from a family of algorithms.

Example:

Let's take an example where we have a text processor that will display data based on the chosen strategy (HTML or Markdown).

The strategy design pattern allows selecting an algorithm at runtime. **Here's the example step by step:**

1. Definition of Output Formats: An OutputFormat object is defined that lists the available output formats, in this case, Markdown and HTML.

2. Abstract Strategy Class ListStrategy: An abstract class called ListStrategy is created, which defines abstract methods start, end, and addListItem. These methods will be implemented by specific strategies.

```
class ListStrategy {
  start(buffer) {}
  end(buffer) {}
  addListItem(buffer, item) {}
}
```

3. Specific List Strategies: Two classes are created that inherit from ListStrategy to implement specific strategies: MarkdownListStrategy and HTMLListStrategy. Each one implements the abstract methods of the base strategy according to its specific format.

```
class MarkdownListStrategy extends ListStrategy {
  addListItem(buffer, item) {
    buffer.push(` * ${item}`);
  }
}
```

```
class HTMLListStrategy extends ListStrategy {
  start(buffer) {
    buffer.push("<ul>");
  }

  end(buffer) {
    buffer.push("</ul>");
  }

  addListItem(buffer, item) {
    buffer.push(` <li>${item}</li>`);
  }
}
```

4. TextProcessor Class: A TextProcessor class is created that accepts an output format and uses a specific list strategy based on the selected format.

```
class TextProcessor {
  constructor(outputFormat) {
    this.buffer = [];
    this.setFormat(outputFormat);
  }

  setFormat(format) {
    switch (format) {
      case OutputFormat.markdown:
        this.listStrategy = new MarkdownListStrategy();
        break;
      case OutputFormat.html:
        this.listStrategy = new HTMLListStrategy();
        break;
    }
  }

  appendList(items) {
    this.listStrategy.start(this.buffer);
    for (let item of items) {
      this.listStrategy.addListItem(this.buffer, item);
    }
    this.listStrategy.end(this.buffer);
  }

  clear() {
    this.buffer = [];
  }

  toString() {
    return this.buffer.join("\\n");
  }
}
```

5. Using the Text Processor: An instance of TextProcessor is created, the output format is set, and a list is added. Then, the result is displayed.

```
let textProcessor = new TextProcessor();

textProcessor.setFormat(OutputFormat.markdown);
textProcessor.appendList(["one", "two", "three"]);
console.log(textProcessor.toString());

textProcessor.clear();
textProcessor.setFormat(OutputFormat.html);
textProcessor.appendList(["one", "two", "three"]);
```

```
console.log(textProcessor.toString());
```

Final Code:

```
// Enumeration of output formats
let OutputFormat = Object.freeze({
  markdown: 0,
  html: 1,
});

// Abstract class ListStrategy
class ListStrategy {
  start(buffer) {}
  end(buffer) {}
  addListItem(buffer, item) {}
}

// Specific strategy for Markdown format
class MarkdownListStrategy extends ListStrategy {
  addListItem(buffer, item) {
    buffer.push(` * ${item}`);
  }
}

// Specific strategy for HTML format
class HTMLListStrategy extends ListStrategy {
  start(buffer) {
    buffer.push("<ul>");
  }

  end(buffer) {
    buffer.push("</ul>");
  }

  addListItem(buffer, item) {
    buffer.push(`  <li>${item}</li>`);
  }
}

// TextProcessor class
class TextProcessor {
  constructor(outputFormat) {
    this.buffer = [];
    this.setFormat(outputFormat);
  }

  setFormat(format) {
    switch (format) {
      case OutputFormat.markdown:
        this.listStrategy = new MarkdownListStrategy();
        break;
      case OutputFormat.html:
        this.listStrategy = new HTMLListStrategy();
        break;
    }
  }

  appendList(items) {
    this.listStrategy.start(this.buffer);
    for (let item of items) {
      this.listStrategy.addListItem(this.buffer, item);
    }
    this.listStrategy.end(this.buffer);
  }

  clear() {
    this.buffer = [];
  }

  toString() {
    return this.buffer.join("\\n");
  }
}

// Create an instance of the text processor
let textProcessor = new TextProcessor();

// Set the output format to Markdown
textProcessor.setFormat(OutputFormat.markdown);

// Append a list of items
textProcessor.appendList(["one", "two", "three"]);

// Display the output in Markdown format
console.log("Markdown Format Output:");
console.log(textProcessor.toString());

// Clear the text processor
textProcessor.clear();

// Set the output format to HTML
```

```
textProcessor.setFormat(OutputFormat.html);

// Append a list of items
textProcessor.appendList(["one", "two", "three"]);

// Display the output in HTML format
console.log("\\nHTML Format Output:");
console.log(textProcessor.toString());
```

Salida esperada en formato HTML:

```
<ul>
 <li>one</li>
 <li>two</li>
 <li>three</li>
</ul>
```

This code demonstrates how different output format strategies (Markdown and HTML) can be easily switched at runtime using the strategy design pattern.

State

Modifies the behavior of an object when its internal state changes.

The State pattern is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. This pattern is closely related to the concept of finite state machines.

Example:

Let's take an example of a light switch in which turning the switch on or off changes its state.

This is an example of the "State" design pattern, which allows an object to change its behavior when its internal state changes. Here is the code with comments explaining each step:

Step 1: We create two state classes, OnState and OffState, which inherit from the base State class. Each state class defines how the switch should behave when it is in that particular state.

```
class OnState extends State {
  constructor() {
    super();
    console.log("Lights are on");
  }

  turnOff(sw) {
    console.log("Turning off the lights...");
    sw.state = new OffState();
  }
}

class OffState extends State {
  constructor() {
    super();
    console.log("Lights are off...");
  }

  turnOn(sw) {
    console.log("Turning on the lights...");
    sw.state = new OnState();
  }
}
```

Step 2: We create the Switch class, which contains an internal state and turnOn() and turnOff() methods to change the state. We initialize the switch with the off state.

```
class Switch {
  constructor() {
    this.state = new OffState();
  }

  turnOn() {
    this.state.turnOn(this);
  }

  turnOff() {
    this.state.turnOff(this);
  }
}
```

Step 3: We create the abstract base class State, which defines the turnOn() and turnOff() methods. Concrete state classes must implement these methods.

```
class State {
  constructor() {
    if (this.constructor === State) throw new Error("Abstract!");
  }

  turnOn(sw) {
    console.log("The light is on.");
  }

  turnOff(sw) {
    console.log("The light is off.");
  }
}
```

Step 4: We create an instance of the Switch class and test its methods to change the state of the lights. First, we turn the lights on and then we turn them off.

```
let lightSwitch = new Switch();
lightSwitch.turnOn();
lightSwitch.turnOff();
```

This example demonstrates how the "State" design pattern allows an object, in this case, a light switch, to change its behavior based on its internal state (on or off). Each state has its own behaviors defined in the concrete state classes.

Final Code:

```
// Create two state classes

class OnState extends State {
  constructor() {
    super();
    console.log("Lights are on");
  }

  turnOff(sw) {
    console.log("Turning off the lights...");

    sw.state = new OffState();

  }

}


class OffState extends State {

  constructor() {

    super();

    console.log("Lights are off...");

  }


  turnOn(sw) {

    console.log("Turning on the lights...");

    sw.state = new OnState();

  }

}


// Create the Switch class


class Switch {

  constructor() {

    this.state = new OffState(); // Initialize with the off state

  }


  turnOn() {

    this.state.turnOn(this); // Call the turn-on method of the current
state

  }


  turnOff() {
    this.state.turnOff(this); // Call the turn-off method of the current
state

  }

}


// Create the abstract base class State


class State {

  constructor() {

    if (this.constructor === State) throw new Error("Abstract!"); //
Prevent direct instantiation

  }


  turnOn(sw) {

    console.log("The light is on.");

  }


  turnOff(sw) {

    console.log("The light is off.");

  }

}


// Create an instance of the Switch class and test its methods


let lightSwitch = new Switch();

lightSwitch.turnOn(); // Turn on the lights

lightSwitch.turnOff(); // Turn off the lights
```

Template Method

Define the skeleton of an algorithm as an abstract class, specifying how it should be performed.

Template Method is a method in a superclass, typically an abstract superclass, that defines the skeleton of an operation in terms of a series of high-level steps.

Example:

Let's take an example of a chess game. The Template Method design pattern is a design pattern that defines the skeleton of an algorithm in a superclass but allows subclasses to implement certain steps of the algorithm without changing its overall structure. **Here's the code step by step:**

Step 1: We create the base class Game, which will serve as the game's skeleton and defines the overall flow of the game.

```
class Game {
  constructor(numberOfPlayers) {
    this.numberOfPlayers = numberOfPlayers;
    this.currentPlayer = 0;
  }

  execute() {
    this.initialize();
    while (!this.haveWinner) {
      this.takeTurn();
    }
    console.log(`Player ${this.winningPlayer} wins.`);
  }

  initialize() {}
  get haveWinner() {}
  takeTurn() {}
```