

Design Patterns

Applying design principles often leads to the same kind of design in similar situations.

A *design pattern* is a template for a design.

- saves time developing a design
- improves design by employing tried-and-true organizations
- provides a shared vocabulary

Matching the *intent* of a pattern to the situation at hand is essential for using patterns correctly.

Variations of Behavior

- Strategy – to parameterize variation of behavior
 - typically only varies one piece of the behavior, not the whole set of what the context can do
 - client picks the variation for the context to use
 - variants are peers – none of them know about the others
 - the strategy in effect can be swapped at runtime, even on an existing object
- State – to allow variation of behavior when internal state changes
 - each state is a full alternate version of the context's behavior, not an isolated piece
 - the context manages its own states, setting its own starting state
 - state objects often know about each other and drive their own transitions
- Template Method – to allow variations in steps of a process
 - defines a fixed algorithm skeleton in a base class
 - subclass override specific steps
 - the variant in effect is fixed by which subclass was instantiated

Matching the *intent* of a pattern to the situation at hand is essential for using patterns correctly.

Pattern: Strategy

Intent.

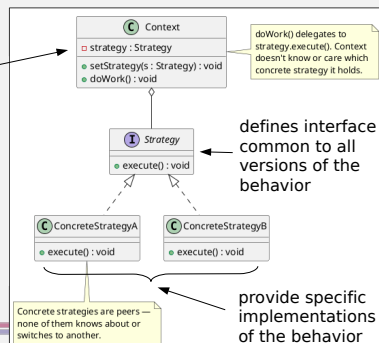
- parameterize variation of behavior

Code structure.

contains a reference to a strategy object (initialized through constructor and/or setter)
passing a strategy object as parameter to method(s) needing it isn't technically the Strategy pattern

delegates some behavior (the parts that vary) to strategy object

may define an interface that allows strategy object to access its data



Strategy Example

- multiple ways to pay for an online order

```

public interface PaymentStrategy {
    void pay(double amount);
}

public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    public CreditCardPayment(String cardNumber) { this.cardNumber = cardNumber; }
    public void pay(double amount) {
        System.out.println("Charged $" + amount + " to card " + cardNumber);
    }
}

public class PayPalPayment implements PaymentStrategy {
    private String email;
    public PayPalPayment(String email) { this.email = email; }
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " via PayPal account " + email);
    }
}

public class CashPayment implements PaymentStrategy {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " in cash");
    }
}

```

```

public class ShoppingCart {
    private PaymentStrategy paymentStrategy;
    private List<Item> items = new ArrayList<>();

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void addItem(Item item) { items.add(item); }

    public void checkout() {
        double total = getTotal(items);
        paymentStrategy.pay(total);
    }

    ...
}

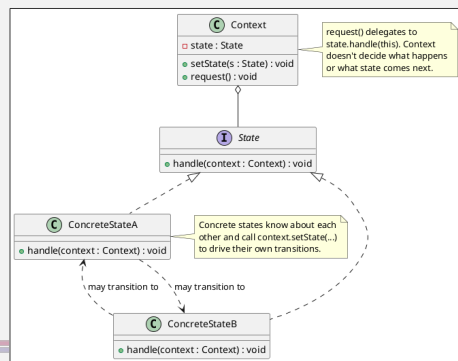
```

Pattern: State

Intent.

- allow variation of behavior when internal state changes

Code structure.



State Example

- vending machine with different responses to an action

```

public class VendingMachine {
    private static final int NO_COIN = 0;
    private static final int HAS_COIN = 1;
    private static final int SOLD = 2;

    private int state = NO_COIN;

    public void insertCoin() {
        if (state == NO_COIN) {
            System.out.println("Coin accepted");
            state = HAS_COIN;
        } else if (state == HAS_COIN) {
            System.out.println("Coin already inserted");
        } else if (state == SOLD) {
            System.out.println("Please wait, dispensing item");
        }
    }

    public void selectItem() { ... }
    public void dispense() { ... }
}

```

without using the State
pattern – a set of constants
and a big if statement in
each method

```

public interface VendingMachineState {
    void insertCoin(VendingMachine machine);
    void selectItem(VendingMachine machine);
    void dispense(VendingMachine machine);
}

public class NoCoinState implements VendingMachineState {
    public void insertCoin(VendingMachine machine) {
        System.out.println("Coin accepted");
        machine.setState(new HasCoinState());
    }
    public void selectItem(VendingMachine machine) {
        System.out.println("Insert a coin first");
    }
    public void dispense(VendingMachine machine) {
        System.out.println("Insert a coin first");
    }
}

```

using State pattern

```

public class HasCoinState implements VendingMachineState {
    public void insertCoin(VendingMachine machine) {
        System.out.println("Coin already inserted");
    }
    public void selectItem(VendingMachine machine) {
        System.out.println("Item selected");
        machine.setState(new SoldState());
    }
    public void dispense(VendingMachine machine) {
        System.out.println("Select an item first");
    }
}

public class SoldState implements VendingMachineState {
    public void insertCoin(VendingMachine machine) {
        System.out.println("Please wait, dispensing item");
    }
    public void selectItem(VendingMachine machine) {
        System.out.println("Already sold, please wait");
    }
    public void dispense(VendingMachine machine) {
        System.out.println("Item dispensed");
        machine.setState(new NoCoinState());
    }
}

```

```

public class VendingMachine {
    private VendingMachineState state = new NoCoinState();

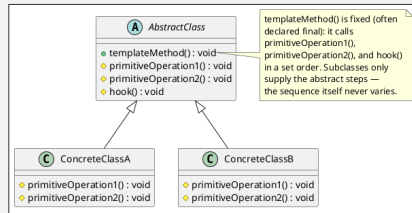
    void setState(VendingMachineState state) { this.state = state; }

    public void insertCoin() { state.insertCoin(this); }
    public void selectItem() { state.selectItem(this); }
    public void dispense() { state.dispense(this); }
}

```

Template Method

- intent
 - to allow variation in the steps of a process



primitive operations are expected to be defined by subclasses – things that definitely vary

hooks provide an opportunity for subclasses to override (not possible if the operation is instead hardcoded in the template method) but that is not the expectation

Example

- brewing a hot beverage

```
public abstract class CaffeineBeverage {
    public final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    private void boilWater() {
        System.out.println("Boiling water");
    }
    private void pourInCup() {
        System.out.println("Pouring into cup");
    }
    protected abstract void brew();
    protected abstract void addCondiments();
}
```

these parts of the process are unlikely to change, but encapsulating them in methods instead of hardcoding them directly in prepareRecipe() provides *hooks* that a subclass could override

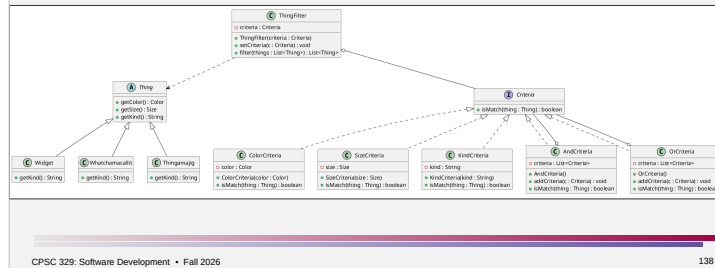
the primitive operations that are variant dependent – subclass must provide a definition

```
public class Coffee extends CaffeineBeverage {
    protected void brew() {
        System.out.println("Dripping coffee through filter");
    }
    protected void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

public class Tea extends CaffeineBeverage {
    protected void brew() {
        System.out.println("Steeping the tea");
    }
    protected void addCondiments() {
        System.out.println("Adding lemon");
    }
}
```

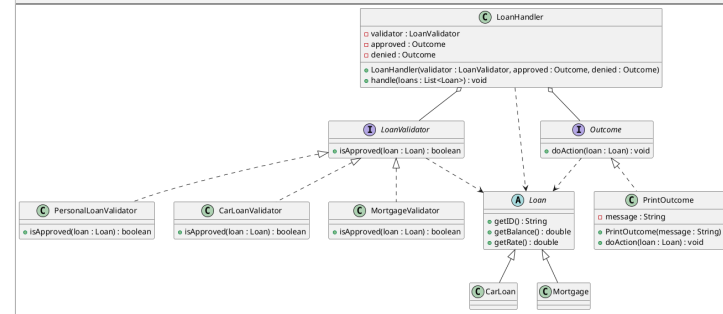
Which Pattern?

- ThingFilter – can filter things based on size, color, kind, ...
 - Strategy
 - filtering logic is a single well-defined piece of behavior that varies independently of everything else ThingFilter does



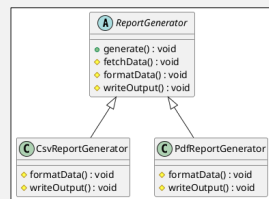
Which Pattern?

- LoanHandler – approve or disapprove loans in a collection
 - Strategy
 - the rules for approving a loan and the handling of the outcome are single, independent pieces of behavior



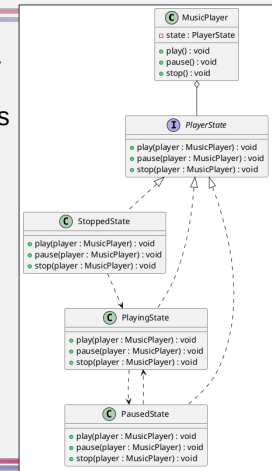
Which Pattern?

- A ReportGenerator base class defines generate(), which calls fetchData(), formatData(), and writeOutput() in that fixed order. CsvReportGenerator and PdfReportGenerator override formatData() and writeOutput() but never touch generate().
 - Template Method
 - a fixed skeleton containing steps in a certain order



Which Pattern?

- A MusicPlayer behaves differently depending on whether it's currently Stopped, Playing, or Paused. Pressing "play" while stopped starts playback and moves it to Playing; pressing "play" while paused resumes and also moves it to Playing; pressing "pause" while playing moves it to Paused — and each of these mode objects decides where to go next.
 - State
 - each mode determines both what a button press does and which mode comes next

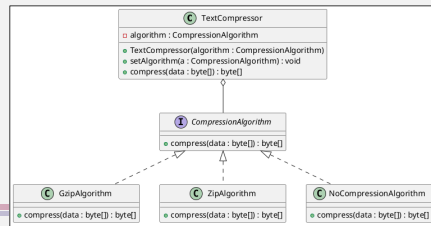


Which Pattern?

- A TextCompressor takes a CompressionAlgorithm object in its constructor (gzip, zip, or a no-op passthrough) and calls algorithm.compress(data). A setAlgorithm() method lets the same TextCompressor instance switch algorithms later.

→ Strategy

- compression algorithm is one interchangeable piece of behavior

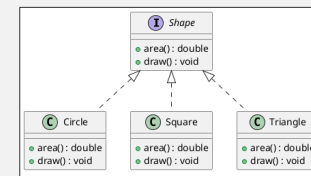


Which Pattern?

- A Shape interface is implemented by Circle, Square, and Triangle, each providing its own area() and draw().

→ none of the above

- just polymorphism – no parameterized behavior



Which Pattern?

- An Elevator changes what a floor-button press does depending on whether it's Idle, MovingUp, MovingDown, or DoorsOpen. After reaching a requested floor, the elevator itself decides whether to open its doors or continue toward the next request.

→ Strategy

- behavior changes according to state, elevator decides for itself when to change state

