Behavioral Patterns

Strategy

State

Motivating Example

- Users log in with password. We need to save these passwords in a DB in an encrypted form.
- Currently we use 3 encryption algorithms.
  - Generate hash using one of them and then save in DB
- Problem: How to write setPassword to support both encryption?
  - Later I may embed new encryptions

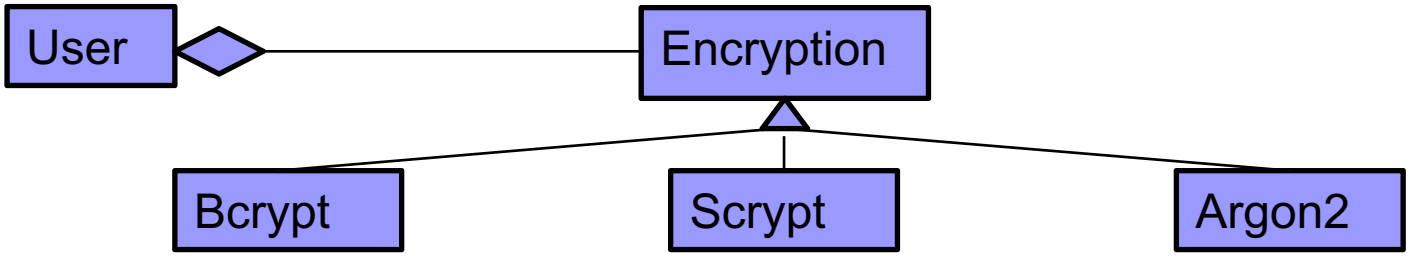
```
class User{
    public:  virtual bool setPassword(const String&
passwd);
                    virtual bool checkPassword(const String&
```

How to implement setPassword?

- Soln1: conditionals in setPassword and getPassword method
  - Switch and long ifs are not extensible
- Soln2: inheritance
  - Abstract User and Subclasses override setPassword and getPassword
    - Too many subclasses! One for each encryption
    - Should I use inheritance just to override 1 method?
- Soln3: encapsulate what varies
  - What is varying?

How to implement setPassword?

- What is varying? A function realization
  - Put it in a class -- a class for generateHash()
  - Choose the suitable function at runtime



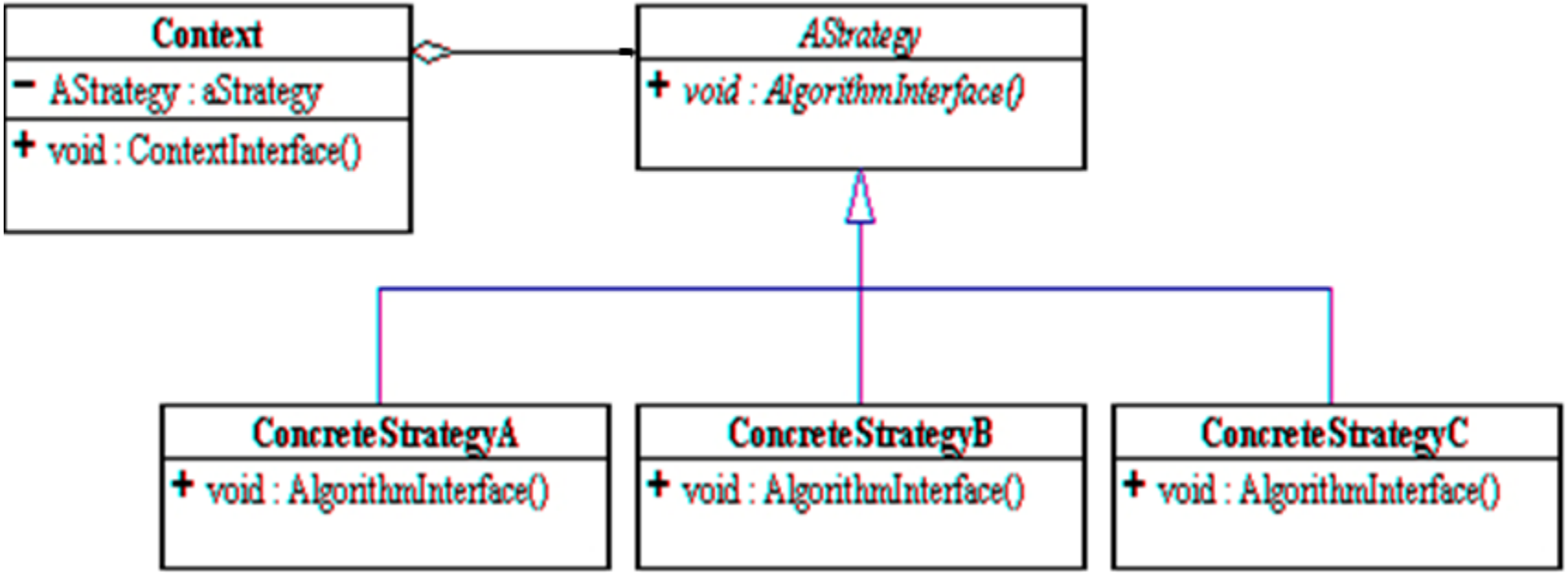
- Choose a suitable one from the **algorithm family** at runtime
  - Delegation instead of inheritance

Strategy

- **Intent**:
  - Define a family of algorithms, **encapsulate** each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from clients that use it.
- A.k.a. Policy

Strategy -Structure

- **Participants?**



setPassword with Strategy

Participants?

```
class User{
 public:
    virtual void setPassword(const String& passwd);
    virtual bool checkPassword(const String&
passwd);
     User(Encryption* e); ~User()=default;
private:     Encryption* strategy;
….}
void User::setPassword(const String& pass){
          checkStrength(pass);
          save(strategy ->generateHash(pass));
}
User::User(Encryption* e):strategy(e){…}
```

Strategy - Collaborations

- The context object receives requests from the client and delegates them to the strategy object.
  - Usually, the ConcreteStrategy is created by the client and passed to the context. From this point the clients interacts only with the context.
- The Context objects contains a reference to the ConcreteStrategy that should be used.
- When an operation is required then the algorithm is run from the strategy object.
- The Context is not aware of the strategy implementation.
- If necessary, additional objects can be defined to pass data from context object to strategy.

Recall the Car example 2nd week

- Car class and its 2 operations/behaviors:
  - Brake and accelerate
- These behaviors change frequently between models, so implement these behaviors in subclasses: overriding
  - For each new model, override
    - Beware: Code duplication across models
    - The work of managing these behaviors increases greatly as the number of models increases

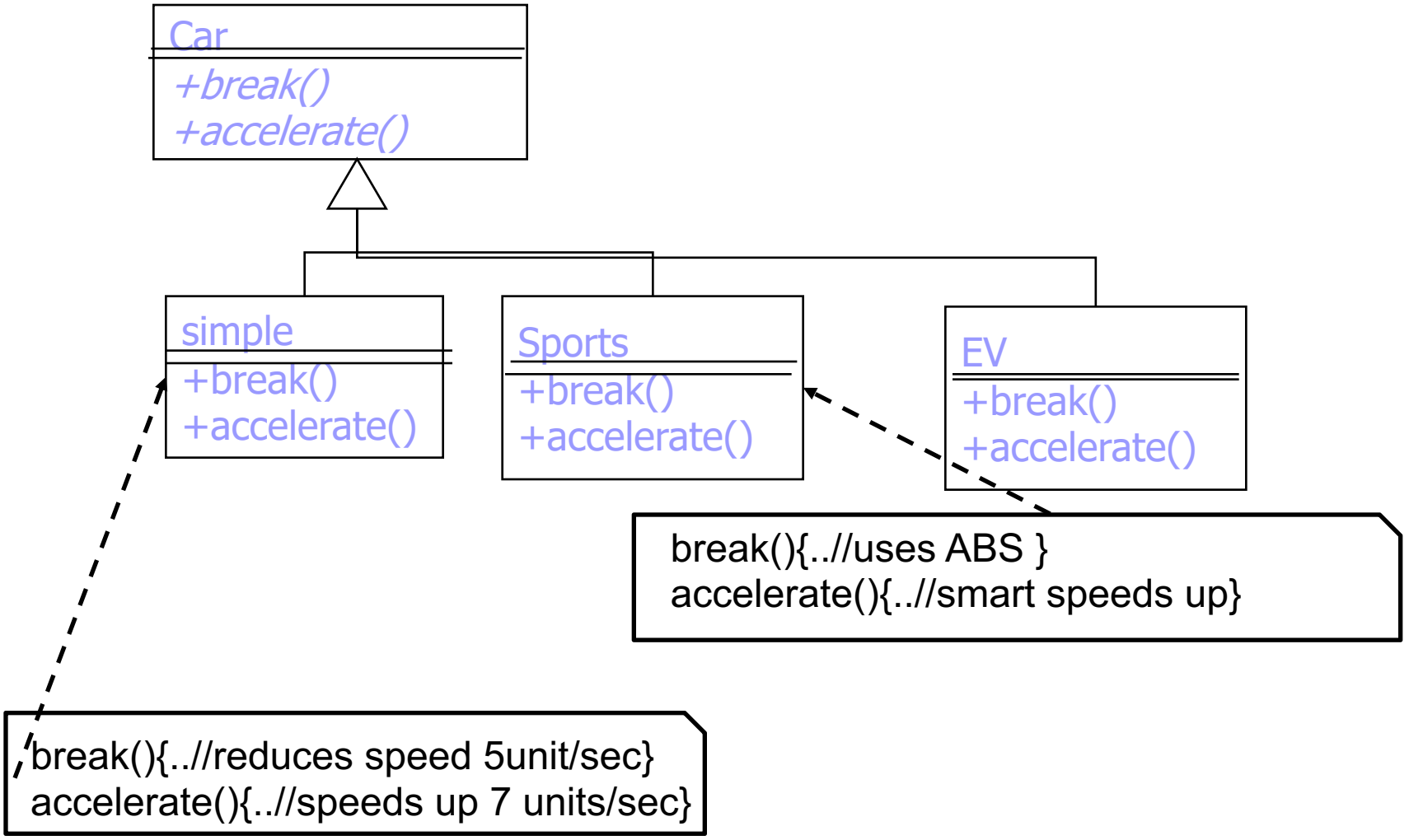
Alternative1

```
public class  Car{
  public void break(){
      switch(model){
      case Simple:
        //reduce speed 5 unit/sec
      case Sportcar:
       //….with ABS
      case EV:
        //regenerative breaking
   }
//other members
```

```
public void accelerate(){
      switch(model){
      case Simple:
              //speed up  7u/s
      case EV:
          //smart speed up
      case Sports: //….
}
```

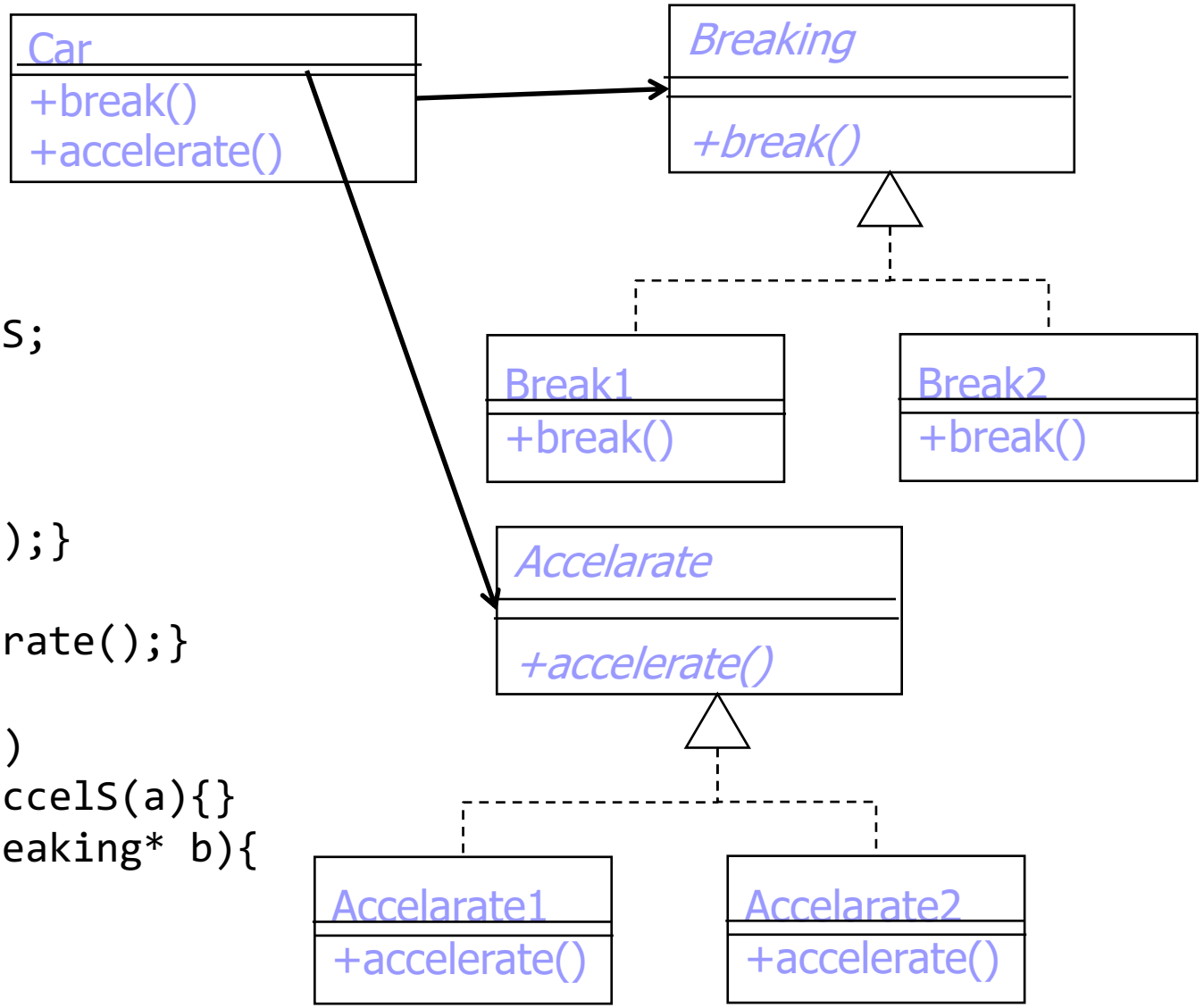
- as ugly as it gets,
- useless when it comes to extensibility
- throws out any hope of it being reusable.

Alternative 2:

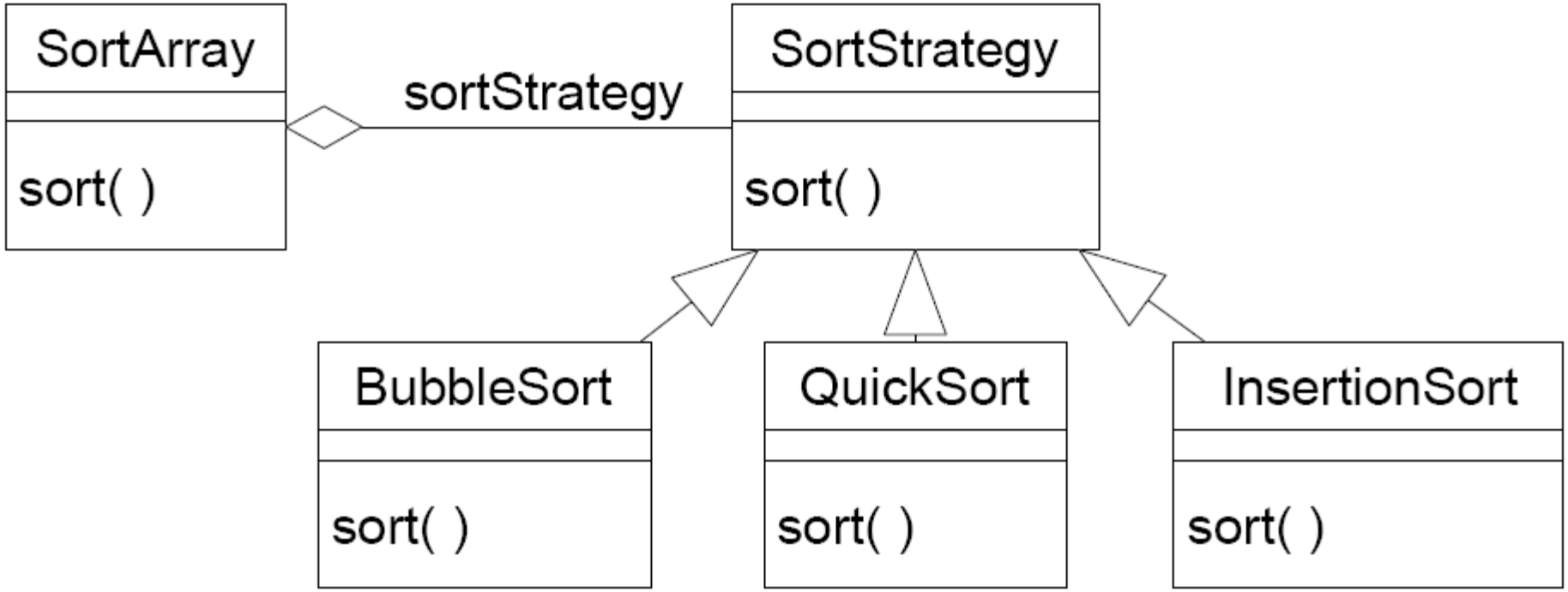


Strategy Solution

```
class Car{
private:
Breaking* breakS;
Accelerate( accelS;
…
public:
break(){
   breakS->break();}
accelerate(){
   accelS->accelerate();}
Car(Breaking* b,
    Accelerate* a)
    :breakS(b), accelS(a){}
void setBreakS(Breaking* b){
    breakS=b;}
…
}
```



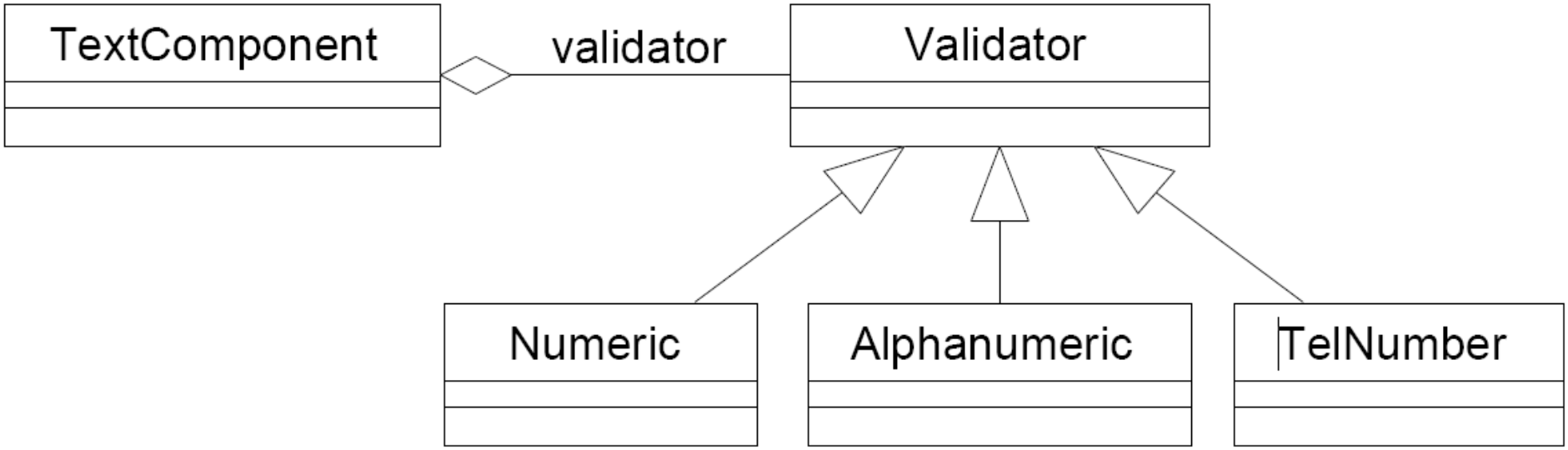
Example 3: Strategy



Application where the sorting algorithm is chosen at runtime

Example 4

- A GUI text component object wants to decide at runtime what strategy it should use to validate user input.
- Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.

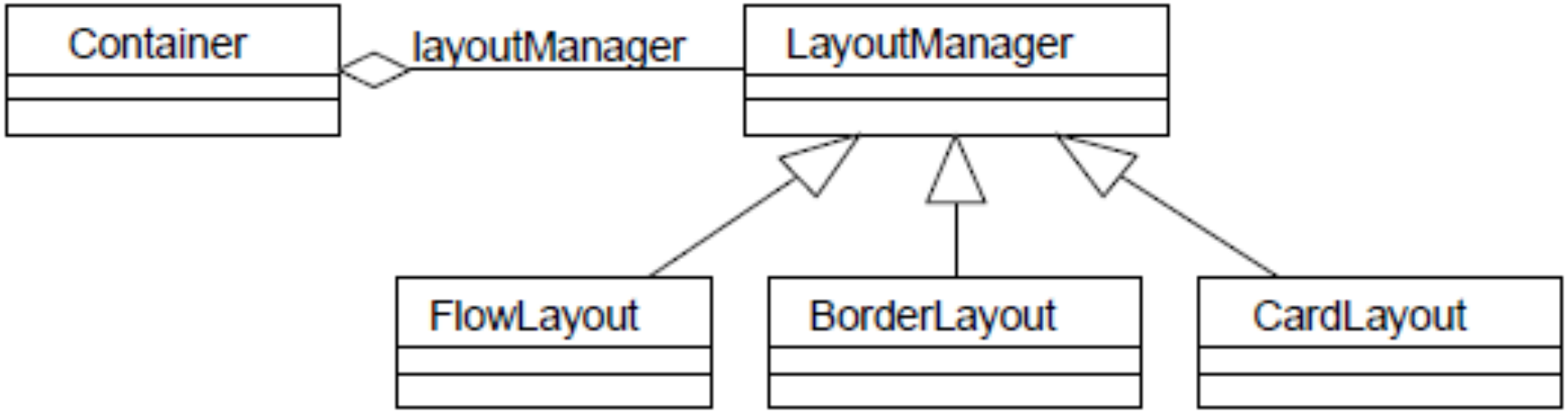


Applicability -Strategy

- Many related classes differ only in behavior
  - you can reduce these several objects to one class that uses several Strategies.
- Need different variants of an algorithm
  - switch from one algorithm to another during runtime
- Avoid exposing complex, algorithm specific data structures
  - E.g. intermediate data structure used for compression
- A class defines many behaviors and chooses one with conditionals

Example 5 – javax.swing

- A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available.




```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("Press"));
```

Example 6

- There are numerous border types
  - Line, titled, edged,…
- Each visual component draws itself with different bordering (assume they are not decorated)
- All they differ is border drawing algorithm
  - There is a family of border drawing algorithm

Example

```
class JComponent{….
 protected void paintBorder(Graphics g) {
  switch(getBorderType()) {
   case LINE_BORDER:    paintLineBorder(g); break;
   case ETCHED_BORDER: paintEtchedBorder(g); break;
   case TITLED_BORDER: paintTitledBorder(g); break;
   ...}
}
```

// The actual implementation of the JComponent.paintBorder() method

```
protected void paintBorder(Graphics g) {
  Border border = getBorder();
   if (border != null) {
     border.paintBorder(this,g,0,0,getWidth(),
  getHeight()); }
}
```

Border Object has the drawing algorithm not the JComponent

Strategy – Consequences-1

- Eliminates conditional statements
  - When you have several behaviors together in one class, you'll use conditionals
  - With Strategies you won't need to check for anything, since whatever the current strategy is just executes without asking questions
- Alternative to subclassing context object to achieve different behaviors
  - With Strategies all you need to do is switch the context's strategy and it will immediately change how it behaves.

Strategy – Consequences-2

- Change the algorithm on the fly
- We can introduce new strategies without having to change the context. (OCP)
- Strategies can be shared
- Increases number of objects and communication overhead btw strategy and context
  - See the implementation issues-2

Strategy: Key notes

- Family of algorithms
  - One strategy class per algorithm
- Make them interchangeable
  - Just change the current strategy object and the algorithm changes
- Client of the context can use different algorithms easily
- Strategy is about delegating **how** an action is performed.
- Alternative for subclassing

Implementation issues-1

- Lambda functions instead of explicit strategy classes
  - Cons: strategies will not be shared
  - Pro: less number of classes

```
class Car {
public:
    std::function<void()> strategy;
    Car(std::function<void()> strategy) : strategy(strategy) {}
    void applyBrakes() {  strategy(); }
};

int main() {
    Car sportsCar( { std::cout << "Applying ABS brakes...\n"; });
    sportsCar.applyBrakes();
}
```

Lambda

```
interface BrakingStrategy {      void applyBrakes();}   //JAVA
class Car {
   private BrakingStrategy strategy;
    public Car(BrakingStrategy strategy) {  this.strategy = strategy;   }
    public void applyBrakes(){    strategy.applyBrakes();    }
}
public static void main(String[] args) {
    Car sportsCar = new Car(() ->
                                System.out.println("Applying ABS brakes..."));
   sportsCar.applyBrakes();
}
//body of lambda implements a single abstract method
```

Using C++ templates

- Cons: Cannot change strategy

```
template <class AStrategy>
class Context {
        void operation() {

theStrategy.DoAlgorithm();
                //delegate to strategy
       }
    private:
        AStrategy theStrategy;
};
```

```
class MyStrategy {
    public: void DoAlgorithm();
 };


//The class is then configured
//with a Strategy class when it's
//instantiated:
  Context<MyStrategy>
aContext;
```

Implementation issues -2

- Data from Context to Strategy
  1. Pass data as parameter
     - `strategy->execute(data1, data2);`
     - Not all strategies may need the same data
  2. Pass the context reference
     - `strategy->execute(this);`
     - Strategy would query context what data it needs
     - Context's interface with getters
     - Strategy is tightly coupled with context
  - There is no best

Implementation issues-3

- Default Strategy in the Context
  - Benefit: Clients don't have to deal with Strategy objects at all *unless* they don't like the default behavior
- Make **stateless** Strategies
  - When Strategies are stateless, they can be shared
  - E.g. 3 users share Bcrypt for hashing their password
  - No need to clutter the memory with same objects
  - Strategies make good flyweights

Stateless strategies

If a Strategy object has no instance variables, it is **stateless.**

- All its data comes from parameters (or the Context).
- We can share a single instance of a stateless strategy across all Context objects.
- This is the **Flyweight** pattern.
  - public static final Strategy FAST_STRATEGY = new FastStrategy();
  - public static final Strategy ACCURATE_STRATEGY = new AccurateStrategy();
  - A good flyweight implementation would use a Flyweight factory.

Question: why not delete?

```
class User{
 public:
     virtual void setPassword(const String& passwd);
     virtual bool checkPassword(const String&
passwd);
      User(Encryption* e); ~User()=default;
private:      Encryption* strategy;
….}
void User::setPassword(const String& pass){
          checkStrength(pass);
          save(strategy ->generateHash(pass));
}
User::User(Encryption* e):strategy(e){…}
```

Strategies are shared: flyweight

Strategy -Related patterns -1

- Strategies can be **flyweights**
- **Template M** vs **Strategy**
  - **Template Method** fixes skeleton of algorithm, variation in steps
  - **Strategy** changes the algorithm completely
    - Change mergesort to bublesort
- **Decorator** vs Strategy
  - Both alternative to inheritance
  - Decorator adds functionality on top -wrapper
  - Strategy replaces functionality

Strategy -Related patterns -2

- **Bridge** and Strategy have the same Class diagram, but intent is different
  - strategy is related with the behavior and bridge is for structure.
  - the coupling between the context and strategies is tighter than the coupling between the abstraction and implementation in the bridge pattern.
- State pattern (next) vs Strategy
- Command (later)

Strategy –Known Use

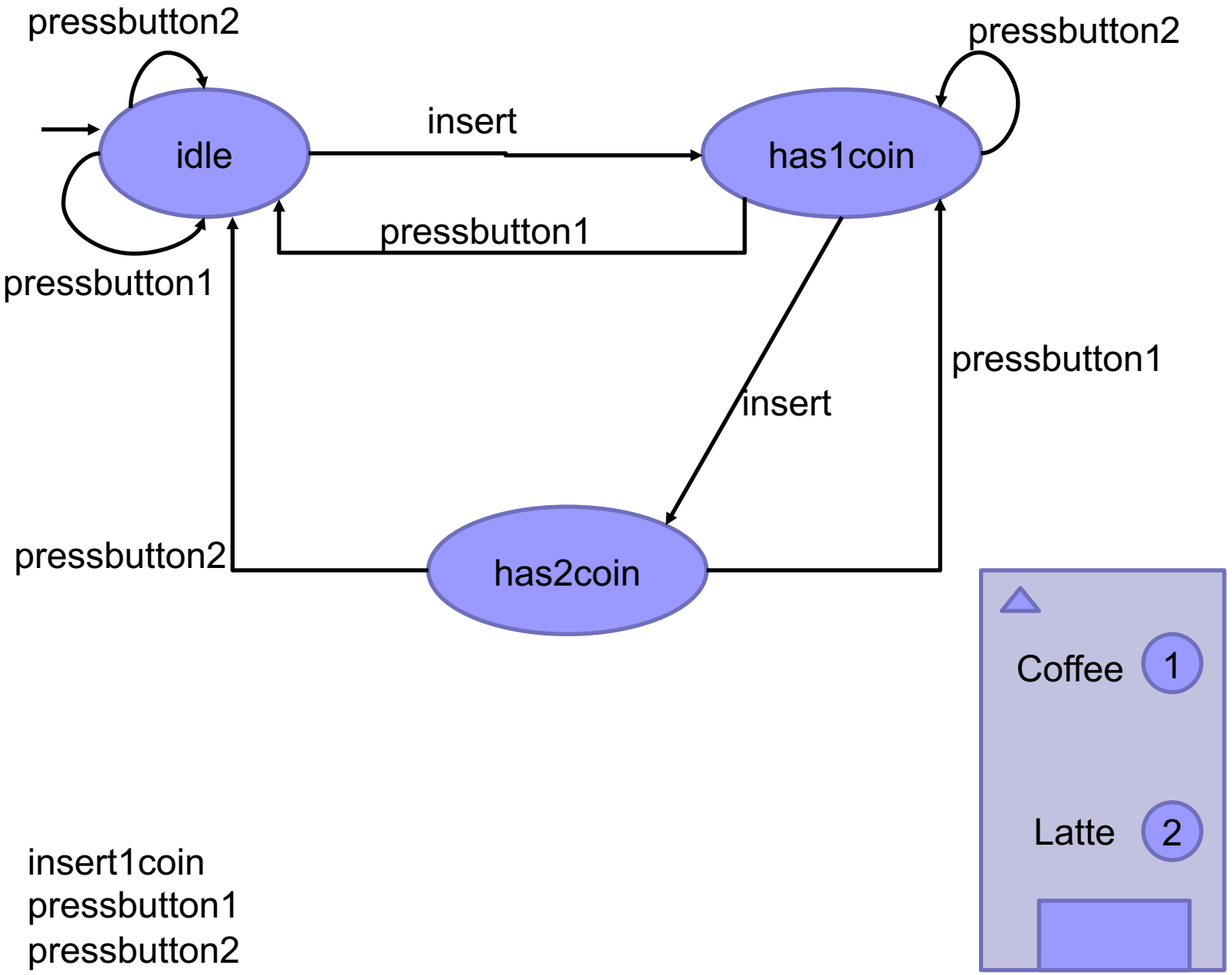
- java.util.Comparator with compare() method is a strategy used by many, e.g. sort method of Collections

Objects with modes

- Objects may respond differently to the same event in different modes
- Example: a Vending machine button press
  - If no coins inserted, ask for payment
  - If correct payment, dispense the product
  - If out of product, return coins

Let's draw a state diagram: nocoins, soldOut,…

How would you **implement** the dispense method?



insert1coin
pressbutton1
pressbutton2

Problem: a class whose behavior depends on some internal state.

- A FSM implemented with a long if/else or switch statement.

```
void handle(Event e) {`
    if (state == STATE_A) { ... if (e == X) state = STATE_B; }`
    else if (state == STATE_B) { ... if (e == Y) state = STATE_A; }
}
```

- This is hard to maintain and violates the Open-Closed Principle.

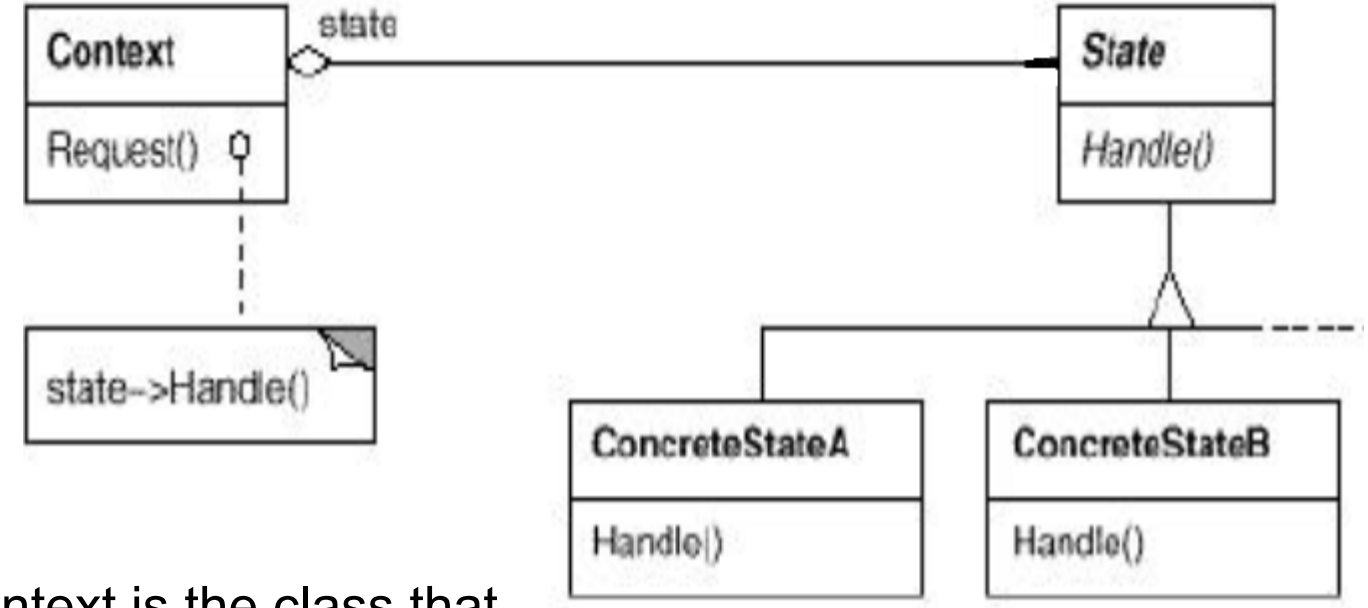
Change the behavior…

- Now, I need to change the behavior of the vending machine to ask for 3 coins as payment
- How will your code get effected?
- How many places need to change?

State Pattern

- **Intent**
  - Allow an object to alter its behavior when its internal state changes.
  - The object will appear to change its class.
    - As if a totally different implementation has taken place at runtime
- Soln: encapsulate each state in its own object.
- The Context (the FSM) will delegate all state-specific behavior to its **current state object.**

State -Structure



Context is the class that
has a number of internal states
It is delegating the behavior
To a state instance

Handle each request
based on the specific state

State Pattern

- **Intent**
  - Allow an object to alter its behavior when its internal state changes.
- **Applicability**
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state.
    - The State pattern puts each branch of the conditional in a separate class.
    - *Especially useful when state-specific code change frequently*

State-Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request.
  - This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects.
  - Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

```
public class VendingMachine{   //the Context
  public final State nocoins=new NoCoinState();
  //states as flyweights
   ….
  State currentState;
  public VendingMachine(){ …  currentState=noCoins; …}
  public  void insertCoin(){
      currentState.insertCoin(this); //delegate action
  }
  public void buttonpressed(){
     currentState.buttonpressed(this); //delegate
  }
   …
  //methods for state transition
   void setState(State s){currentState=s;}
   State getNoCoinState(){return nocoins;}
   …//other getstate methods
```

```
public interface State{ //issue-should this be a class?
    void insertCoin(VendingMachine machine);
    void buttonPressed(VendingMachine machine);
    //other actions
}

public class NoCoinState implements State{
     void insertCoin(VendingMachine machine){
        //do the state transition
         machine.setState( new HasCoinState()); //issue
      }
      void buttonpressed(VendingMachine machine){
          print(“sorry, no coins in the machine”);
     }
}
```

Who determines the next state?

1. Context
   - If the state transition is fixed yes, but it is not flexible
   - If the states will be used in different state machines with different transitions
   - Centralized and visible transitions

   Context:: changeState(){currentstate=…}
2. Concrete state
   - How to do in states
     - context.changeState(new XXXState()); //no
     - context.changeState(context. getXXXState ());
     - context.changeState(XXXState.getInstance());
   - Code change request is handled similar to adding a node into a linked list

Effect of change

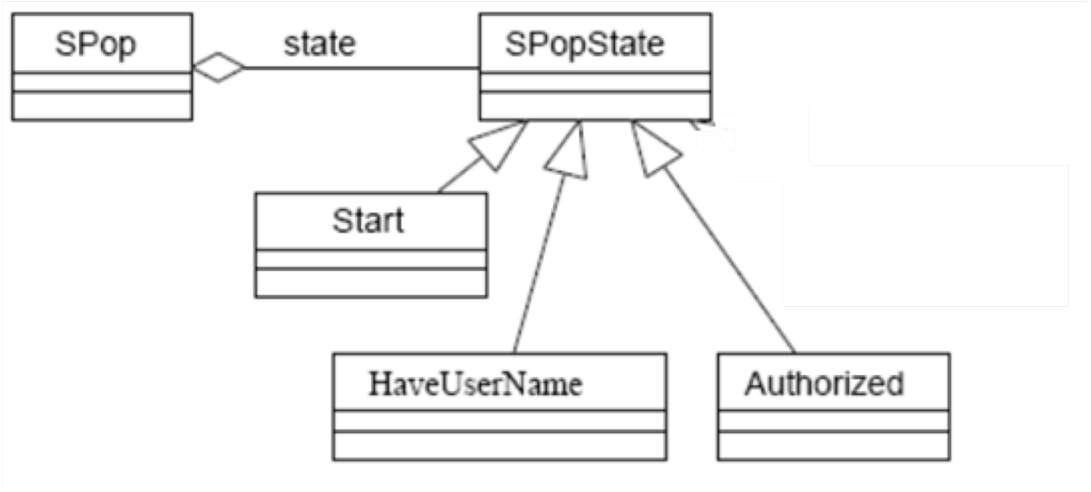
- Now, the change the payment to 4 coins.
- How many places in the code to change?

State pattern

- Localize the behavior for each state into its own class
  - so that if one state is modified, other states are not messed up
  - every state just implements its own actions
- Helps to remove “if”s that are difficult to maintain
- Closed each state for modification and left the machine open to extension by adding new states
  - Easier to add new states in the machine
- Have a structure that maps the state diagram

Example 2: Simple POP

- Simple POP (Post Office Protocol)
  - Get user name (USER)
  - Get password (PASS)
  - Get the size of all messages (LIST)
  - Retrieve a message (RETR)
  - Quit



Simple POP implementation

```
public class SPopState {
  public SPopState user(SPOP context,String userName) {
           //default action here}
  public SPopState pass(SPOP context,String password) {
          //default action here}
  public SPopState list(SPOP context){
         //default action here}
  public SPopState retr(SPOP context,int messageNumber){
          //default action here}
 public SPopState quit(SPOP context,) {
          //default action here}
}
```

Why default action?

Simple POP implementation

```
public class Authorized extends SPopState {
    public void quit (SPOP context) {
        //state transition
         context.setState(context.getStart());
    }
  public void list (SPOP context) {
        //list emails
        //no state transition: stay in this state.
  }
  public void retr (SPOP context,int messageNumber) {
        //list emails
        //no state transition: stay in this state.
  }
}
```

State classes determine the next state

Simple POP implementation

```
public class Start extends SPopState {
   public void user (SPOP context, String userName) {
       //state transition
        context.setState(new HaveUserName(userName));}
}

public class HaveUserName extends SPopState {
     private String userName;    // CANNOT SHARE!
    public HaveUserName(String u){this.userName=u;}

    public void pass(SPOP context. String password) {
     if (validateUser(userName, password)
           context.setState(context.getAutherized());
      else
           context.setState(context.getStart());}
}
```

State classes deter
the next state

and the context is…

```
public class SPop{
  private SPopState state=new Start();

  public void user(String name){
        state.user(this,name);}
  public void pass(String pwd){
        state.pass(this,pwd);}
  public void list(){ state.list();}
  ……
}
```

State vs. Strategy

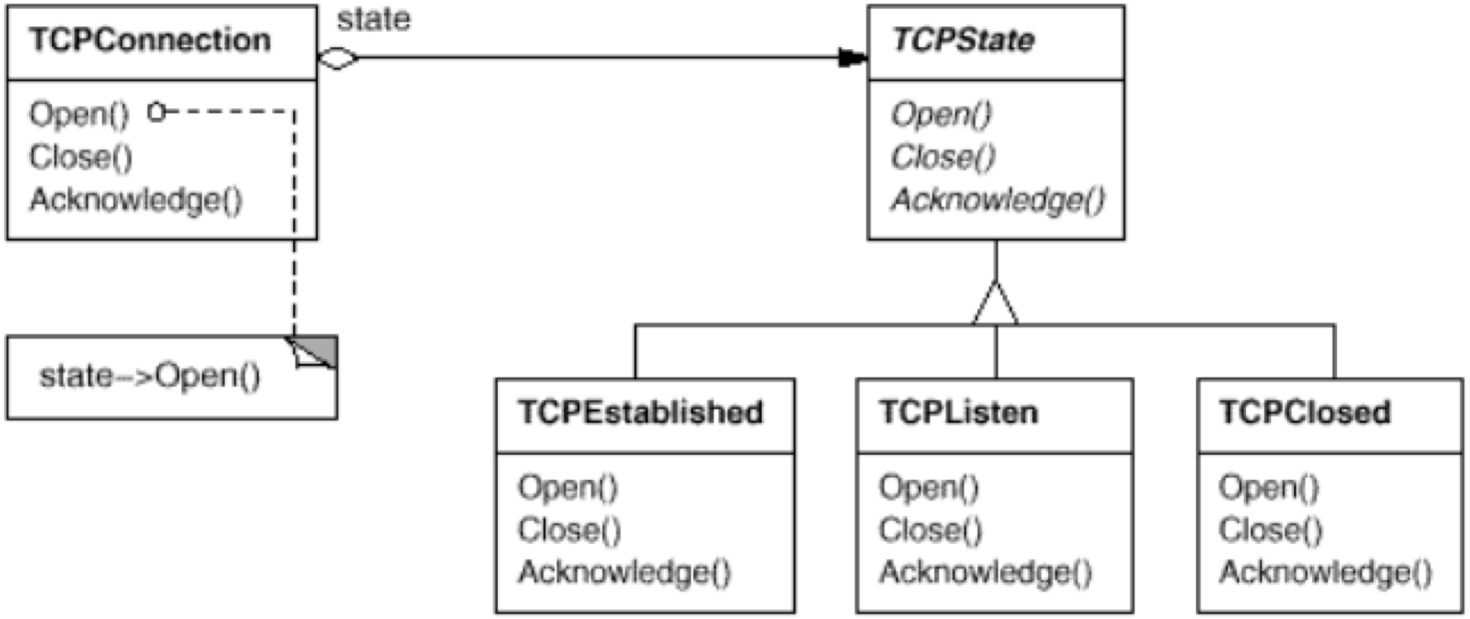
Strategy

- *Who Drives Change?* **Client** (externally).
  - context.setStrategy(new FastSort())
- **Intent:** Answers **"How"** to do an action.
  - How to sort? How to encrypt?
- The Context is **given** a behavior

State

- *Who Drives Change?* **Context or State** (internally).
  account.setState(OVERDRAWN_STATE)
- **Intent:** Answers **"What"** to do (or "When") based on the situation.
  - What do I do when `withdraw` is called **in this state**?
- The Context **is** a behavior.

GoF example – C++ illustration



```
class TCPOctetStream;
class TCPState;

class TCPConnection {
public:
    TCPConnection();

    void ActiveOpen();
    void PassiveOpen();
    void Close();

    void Send();
    void Acknowledge();
    void Synchronize();

    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
```

```
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}

void TCPConnection::ChangeState (TCPS
    _state = s;
}

void TCPConnection::ActiveOpen () {
    _state->ActiveOpen(this);
}

void TCPConnection::PassiveOpen () {
    _state->PassiveOpen(this);
}

void TCPConnection::Close () {
    _state->Close(this);
}

void TCPConnection::Acknowledge () {
    _state->Acknowledge(this);
}

void TCPConnection::Synchronize () {
    _state->Synchronize(this);
}
```

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};

void TCPState::Transmit (TCPConnection*, TCPOctetStream*) { }
void TCPState::ActiveOpen (TCPConnection*) { }
void TCPState::PassiveOpen (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::Synchronize (TCPConnection*) { }

void TCPState::ChangeState (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
```

A State Subclass implements its state specific behavior only

```
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();

    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};

void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN

    ChangeState(t, TCPListen::Instance());
}

void TCPEstablished::Transmit (
    TCPConnection* t, TCPOctetStream* o
) {
    t->ProcessOctet(o);
}
```

State transition

How is it implemented?

State-Consequences -1

- Puts all behavior associated with a state into one object
  - It localizes state-specific behavior and partitions behavior for different states.
  - Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.

State – Consequences-2

- Makes state transition explicit
  - Assume we are not using the state pattern
  - Recall that, object state is valuation of its attributes
    - Let numCoins, numProd be attributes of Machine
    - A machine object with numCoins=10, numProd=5
    - A machine object with numCoins=12, numProd=3
  - A state change occur as assignment to some of object's attributes
    - Machine::insert(){numCoins++;}
    - Machine::pushButton(){ if(numCoins>prevNum) numProd--;}
  - Introducing separate objects for different states makes the transitions more explicit.

I could achieve same effect with an integer variable denoting the state. Why not use that?

State – Consequences-3

- State transition logic is partitioned between the State subclasses rather than in a switch statement
  - Supports open-close principle
  - Removes look-alike conditionals or case statements scattered throughout Context's implementation.
  - Otherwise, adding a new state could require changing several operations, which complicates maintenance.
- Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes

State – Consequences-4

- Easy to forbit unallowed actions in a state
  - E.g. eject quarter in a noquater state
- States objects can be shared
  - Especially when they have no instance variables
- Can be overkill if a state machine has only a few states or rarely changes.
- More objects, rather than a single class
  - Since the pattern distributes behavior for different states across several State subclasses.
  - Such distribution is actually good if there are many states, which would otherwise require large conditional statements.

Thread safety

- State change need to be protected in multithreaded programs
- Consider locking Context methods before delegating to state object
  - Might be inefficient
- Make states immutable as well
- Beware: when locking before state delegation, we might end up with deadlock if there is locking inside the state method too

State-Known use

- javax.faces.lifecycle.LifeCycle,
  - controlled by FacesServlet,
  - the behavior of execute method is dependent on current phase (state) of JSF lifecycle

READ https://gameprogrammingpatterns.com/state.html

State vs. Strategy

Strategy

- *Who Drives Change?* **Client** (externally).
  - context.setStrategy(new FastSort())
- **Intent:** Answers **"How"** to do an action.
  - How to sort? How to encrypt?
- The Context is **given** a behavior

State

- *Who Drives Change?* **Context or State** (internally).
  account.setState(OVERDRAWN_STATE)
- **Intent:** Answers **"What"** to do (or "When") based on the situation.
  - What do I do when `withdraw` is called **in this state**?
- The Context **is** a behavior.

State vs Strategy

- Same structure but different intents
  - What are the intents?
  - State object encapsulates a state-dependent behavior (and possibly the transition)
  - Strategy object encapsulates algorithm
- State object in a context **will** be changed, Strategy **may** change