# Padrões de Projeto

## Unidade 1
## Aula 2

*Edison Silva e*
*Valter Camargo*


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# Agenda para Hoje

- **Período da Manhã** → Padrões : Factory e Template Method
- **Período da Tarde** → Padrões: State, Observer …
- Teremos ao longo do dia as seguintes atividades:
  - Atividade com padrão Factory
  - Atividade com padrão Template Method
  - Atividade com padrões Observer e State
- Astah para UML
  - https://drive.google.com/drive/u/1/folders/1IGblPbLjml6gEQRGxDVdnTCh9blVeqo2
- Slides
  - https://drive.google.com/drive/u/1/folders/1fSQ7Ec8G354PjGoPr05dFj1QWjsU7ivQ

**Grupos**

→ Grupos de 4 alunos

→ Grupos trabalham em atividades relacionadas com padrões

→ Cada professor escolhe um grupo para apresentar o resultado para a turma toda para cada atividade. Assim, sempre dois grupos apresentam a solução.

# Factory

(Creational pattern)


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# FACTORY METHOD

Hoje você sabe quais são os tipos que o seu código cliente deve trabalhar, mas você também sabe que novos tipos irão aparecer no futuro ….

## When to use?

Two situations:

1. Use Factory Method when you don't know in advance the exact types and dependencies of the objects your code should work with.
2. Use Factory Method when you want to make the client code independent (unconsciousness) of the concrete objects it uses

# FACTORY METHOD

## Example

Consider the case of a class for managing employees… one of the responsibilities of this class is **to calculate the salaries of each employee type** (secretaries, sales person, managers, technical leaders, etc).

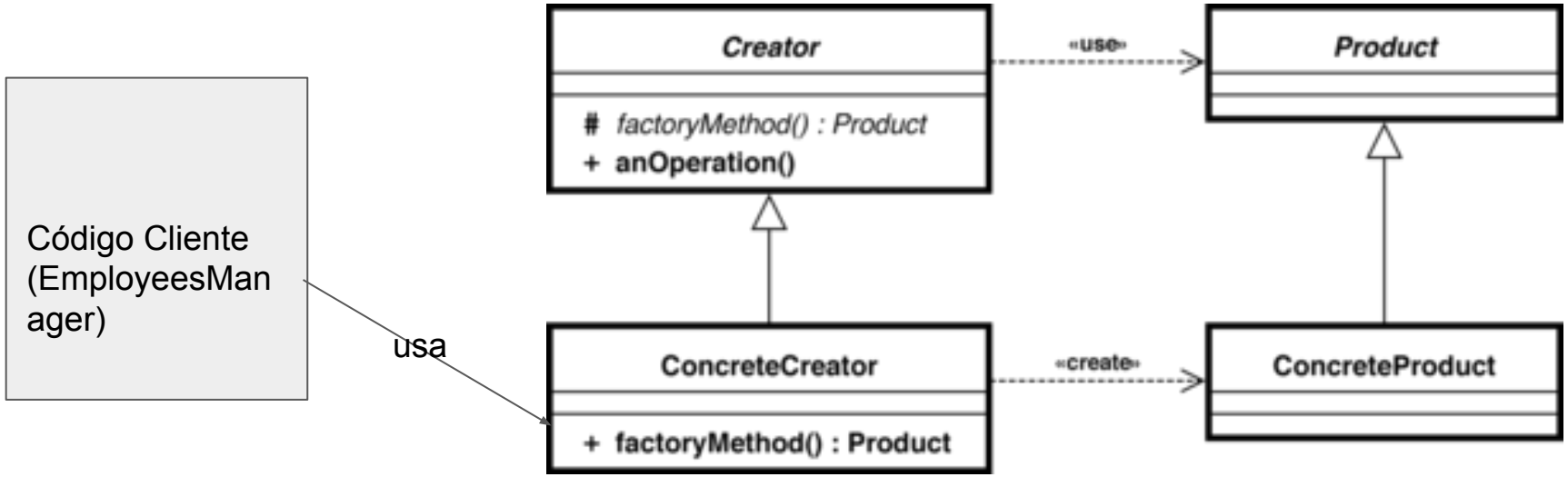
It would be good if this class (EmployeeManager (client code)) did not know the specific types the employees it manipulates…

```
class EmployeesManager {

    public calculateSalaries() {

        ...
        Employee e1 = new Secretary();
        ...
        Employee e2 = new SalesPerson();
        ...
```



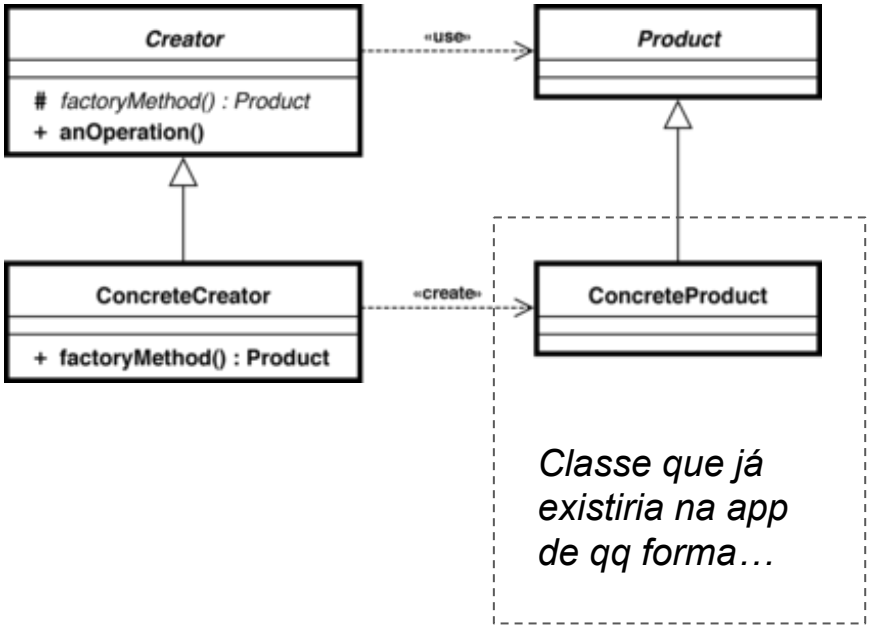
This is not good ……
My **client code** will be very dependent on the employee types..

# FACTORY METHOD




O código cliente usaria (se não usar o padrão) diretamente o produto concreto
Note que, a intenção do padrão Factory fica mais clara e evidente quando pensamos em um sistema distribuído, em que classes precisam se comunicar sem saberem de antemão quem usará os seus métodos e quais métodos serão chamados.

# FACTORY METHOD

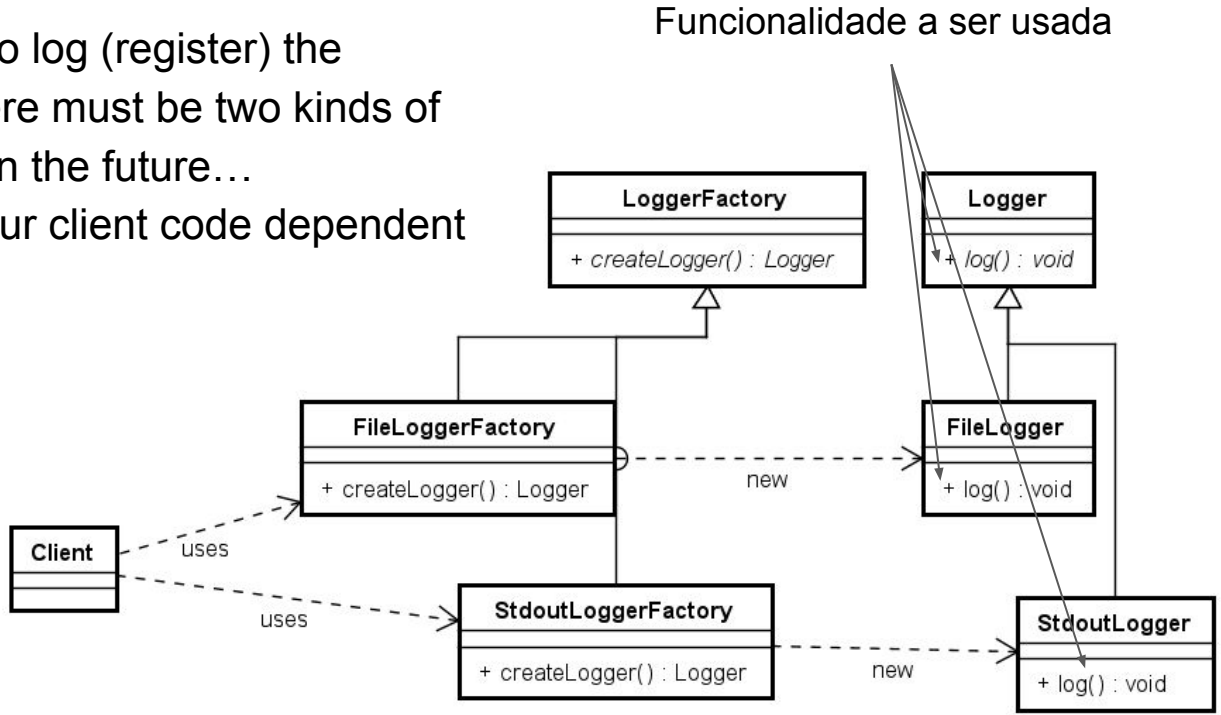



```python
class Creator:
    def factoryMethod(self) -> Product:
        raise NotImplementedError

class Product:
    def doStuff(self) -> None:
        raise NotImplementedError

class ConcreteCreator(Creator):
    def factoryMethod(self) -> Product:
        return ConcreteProduct()

class ConcreteProduct(Product):
    def doStuff(self) -> None:
        pass
```

# FACTORY METHOD

Context:

- You system must have way to log (register) the execution of the system. There must be two kinds of logs, but others can appear in the future…
- You wouldn´t like to make your client code dependent on the types of loggers...



# FACTORY METHOD IN PYTHON

```
import logging
import sys

class Logger:
    def log(self) -> None:
        raise NotImplementedError

class FileLogger(Logger):

    _filePath: None

    def __init__(self, filePath: str) -> None:
        self._filePath = filePath

    def log(self) -> None:
        logging.basicConfig()
        logging.info('This message will get logged on to a file')

class StdoutLogger(Logger):

    def log(self) -> None:
        handler = logging.getLogger()
        handler.setLevel(logging.INFO)
        ch = logging.StreamHandler(sys.stdout)
        ...
        handler.info('This message will get logged on to a stdout')
```

```
class LoggerFactory:

    def createLogger(self) -> Logger:
        raise NotImplementedError

class FileLoggerFactory(LoggerFactory):
    def createLogger(self) -> Logger:
        return FileLogger('file.log')

class StdoutLoggerFactory(LoggerFactory):
    def createLogger(self) -> Logger:
        return StdoutLogger()
```

new()
Veja que isto é uma delegação
...

```
if __name__ == "__main__":
    fileLogger = FileLoggerFactory().createLogger()
    fileLogger.log()

    stdoutLogger =
StdoutLoggerFactory().createLogger()
    stdoutLogger.log()
```

Client code !

# FACTORY METHOD IN JAVA

```
abstract class Logger:

    public void abstract log(Logger);

class FileLogger extends Logger {

    private String filePath

    public FileLogger(String filePath){
       this.filePath = filePath;
    }

    public void log (Logger) {
        logging.basicConfig()
        logging.info('This message will get logged on to a file')
    }
class StdoutLogger extends Logger {

    public void log(Logger) {
       ….
       Sustem.out.println(“This message will be logged to a stdout”)

}
```

Visibilidade de pacote (se possível)

```
public abstract class LoggerFactory:

       public abstract Logger createLogger() ;

public class FileLoggerFactory extends LoggerFactory {
       public Logger createLogger() {
              return new FileLogger();
}


public class StdoutLoggerFactory extends LoggerFactory {
       public Logger createLogger() {
              Return new StdoutLogger();

}

public class main {
       …
       Logger logger = FileLoggerFactory.createLogger();
       logger.log();
       …
       logger = StdoutLoggerFactory.createLogger();
       logger.log();
```

Client code !

# Exercício para os grupos

- Imagine a existência de um sistema de gerenciamento de clientes que possui dois tipos de clientes: Físico e Jurídico.
- Existe um código cliente que não deve conhecer os tipos de clientes, mas que deve ser capaz de criar objetos deles
- **Forma de resolução**
  - Criar um diagrama de classes UML
  - Acrescentar trechos de código nas partes importantes do padrão.

# Template Method


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# TEMPLATE METHOD

**When to use?**

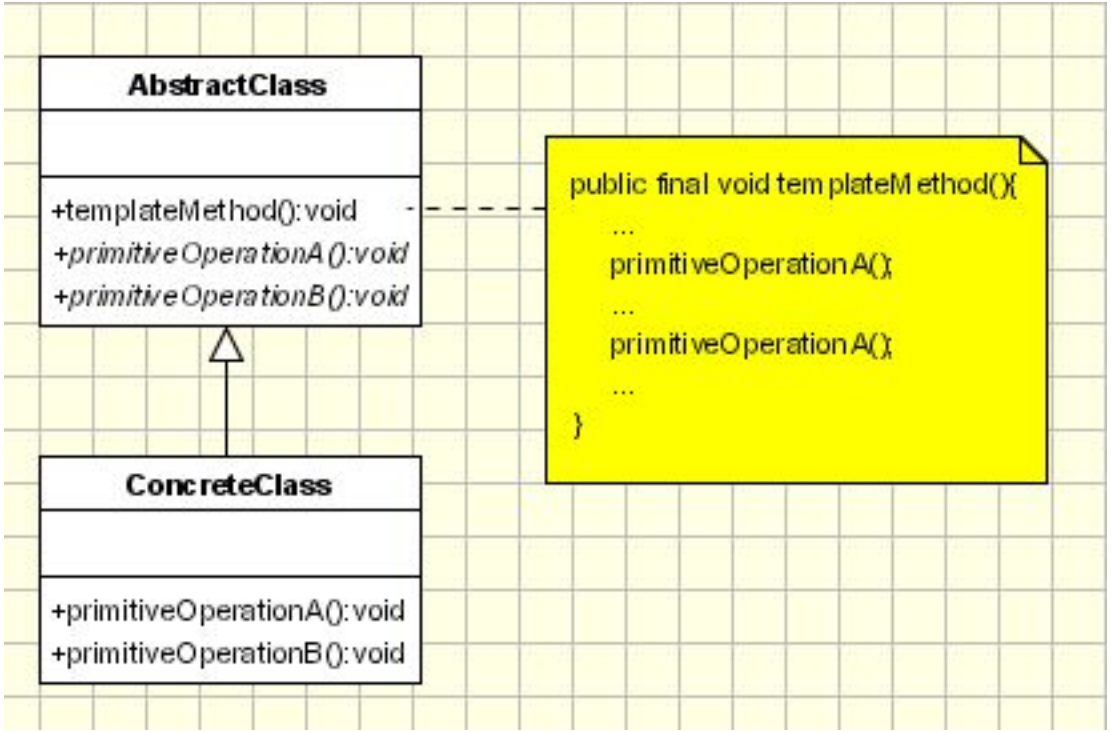
This pattern must be used when you are creating a solution that will e used for others. Besides, this solution (program) requires details that only the final user knows.

# TEMPLATE METHOD

**Example**

This pattern is used in frameworks, where the user can define application-specific details.

# GENERAL STRUCTURE



# TEMPLATE METHOD - EXAMPLE

```
public abstract class OdbcConnection extends ConnectionManager {

    private String currentJdbc;

    public OdbcConnection(){

        this.currentJdbc = "jdbc:odbc:" + setDSN();
        this.driver = "sun.jdbc.odbc.JdbcOdbcDriver";

    }

    public abstract String setDSN();

    public void connect(){
    try
    {
```

Classes pertencente do Framework

Quem desenvolve é o engenheiro do framework

# TEMPLATE METHOD - EXAMPLE

```
package persistence.instantiation;
import persistence.connection.*;

public class myConnectionVariabilities extends OdbcConnection
{

        ...

        public String setDSN()
        {
                return "Workshop";
        }
}
```

Classe pertencente à aplicação

Quem desenvolve é o engenheiro da aplicação

# Atividade para os Grupos

Você é responsável pela implementação de um software de mineração de dados. A funcionalidade do seu código é que ele seja capaz de **analisar dados** e **enviar um relatório dessa análise**;

Em princípio, não se sabe o formato dos dados a serem processados… inclusive, podem existir n tipos de dados - *até mesmo formatos que sejam específicos de determinados clientes*.

Independentemente do formato, o seu código deve estar preparado para **fazer a análise** e **enviar o relatório**.

Neste contexto, o que pode variar e que não tem como você prever é (específico de cada cliente):

- Como abrir o arquivo dos dados a serem analisados
- Como extrair os dados do arquivo
- Como fazer o parser dos dados
- Como fechar o arquivo

O seu código deve ser capaz de trabalhar levando em conta essas variações possíveis

# Padrões de Projeto

## Unidade 2
## Aula 3

*Edison Silva e*
*Valter Camargo*


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# State

(behavior pattern)


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# STATE PATTERN

- O que são “estados” ?
- O que significa dizer que um “objeto está em um determinado estado” ?
- Faz sentido dizer que o Sistema tem “estado” ?
- Este padrão (em seu formato original) só é aplicável para situações em que o objeto possui estados mutuamente exclusivos.
- Se um objeto pode estar em vários estados ao mesmo tempo, este padrão precisa ser estendido/modificado

# STATE PATTERN

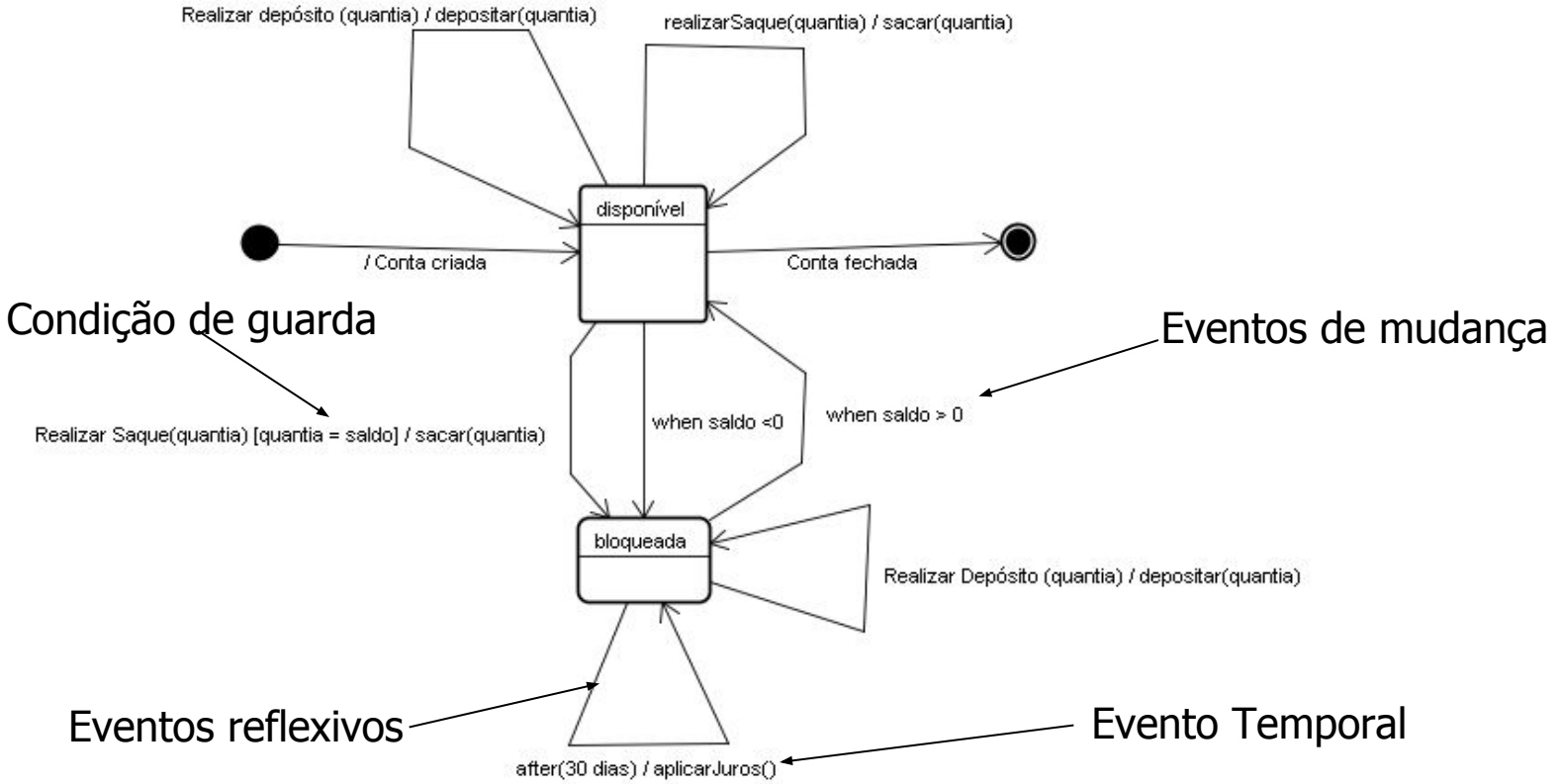
## When to use?

When you wish to represent clearly the states of an object/class in the system, so that you can handle them explicitly

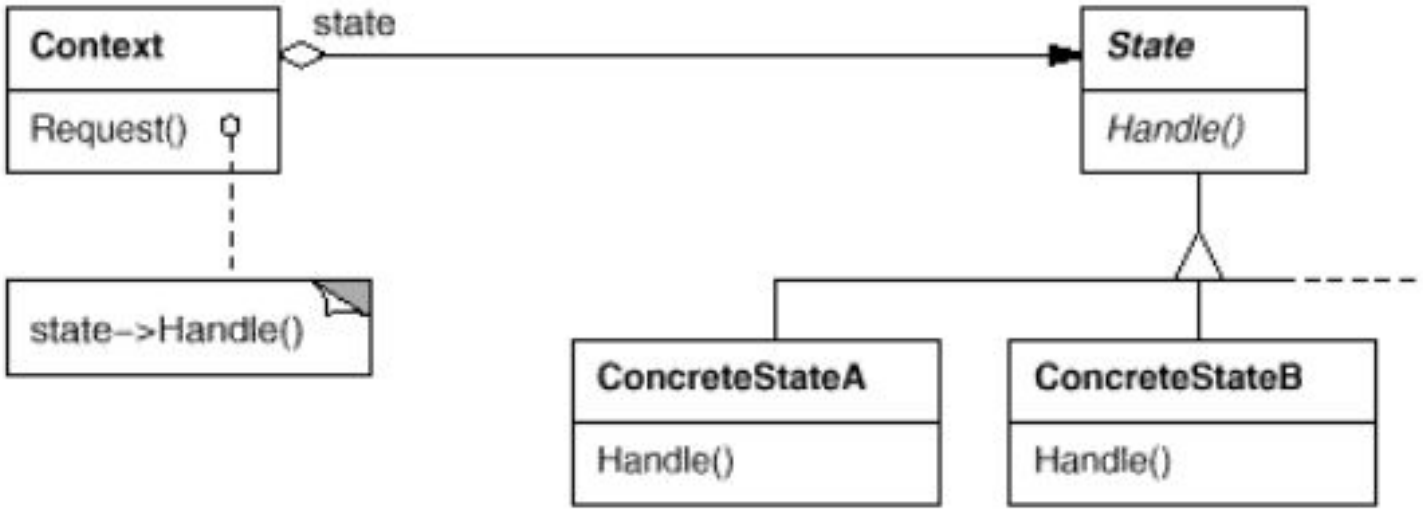
## Intention

Making the behavior of an object be dependent on the state it is in a specific moment, i.e., the behavior will be different according to the state it is

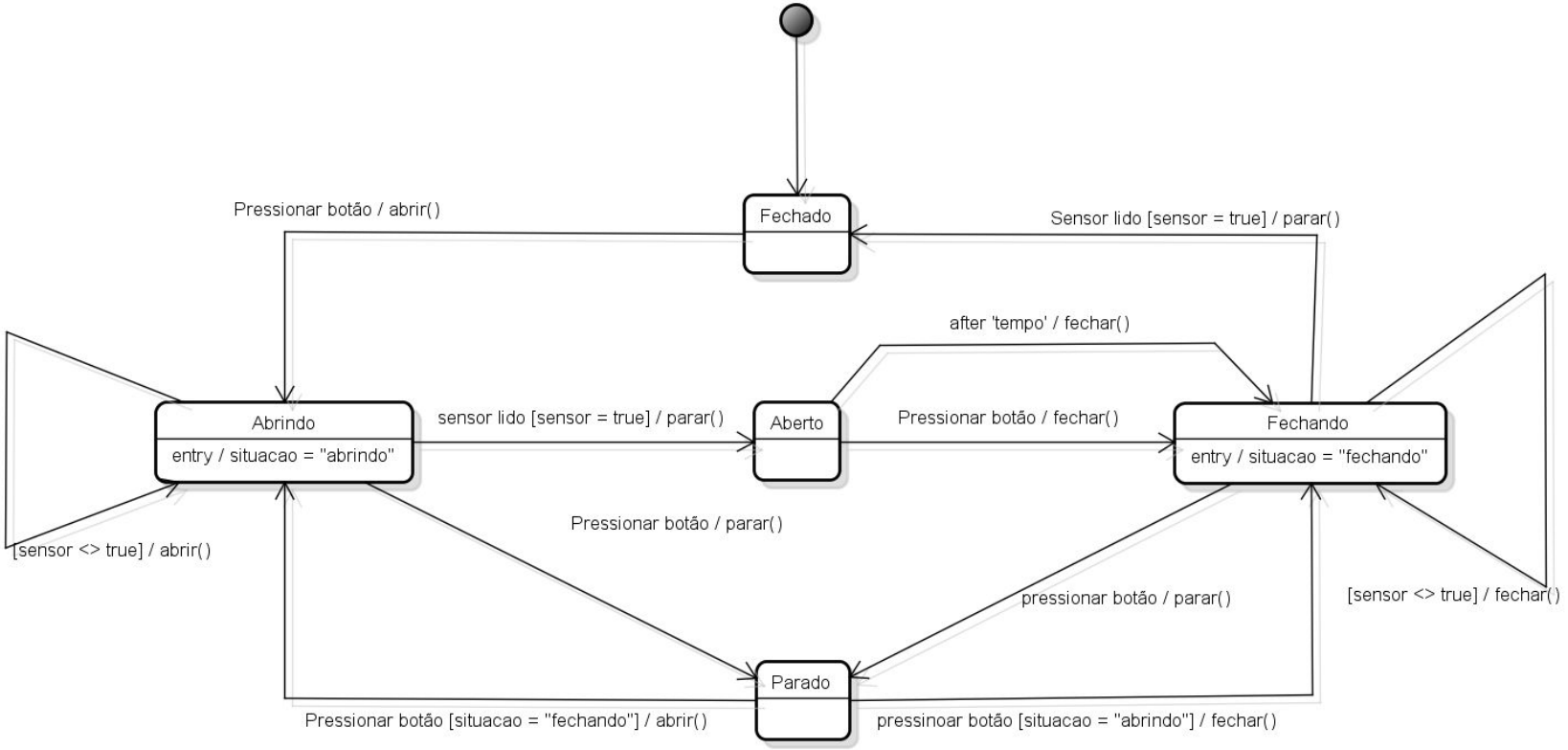
# STATE PATTERN - EXAMPLE



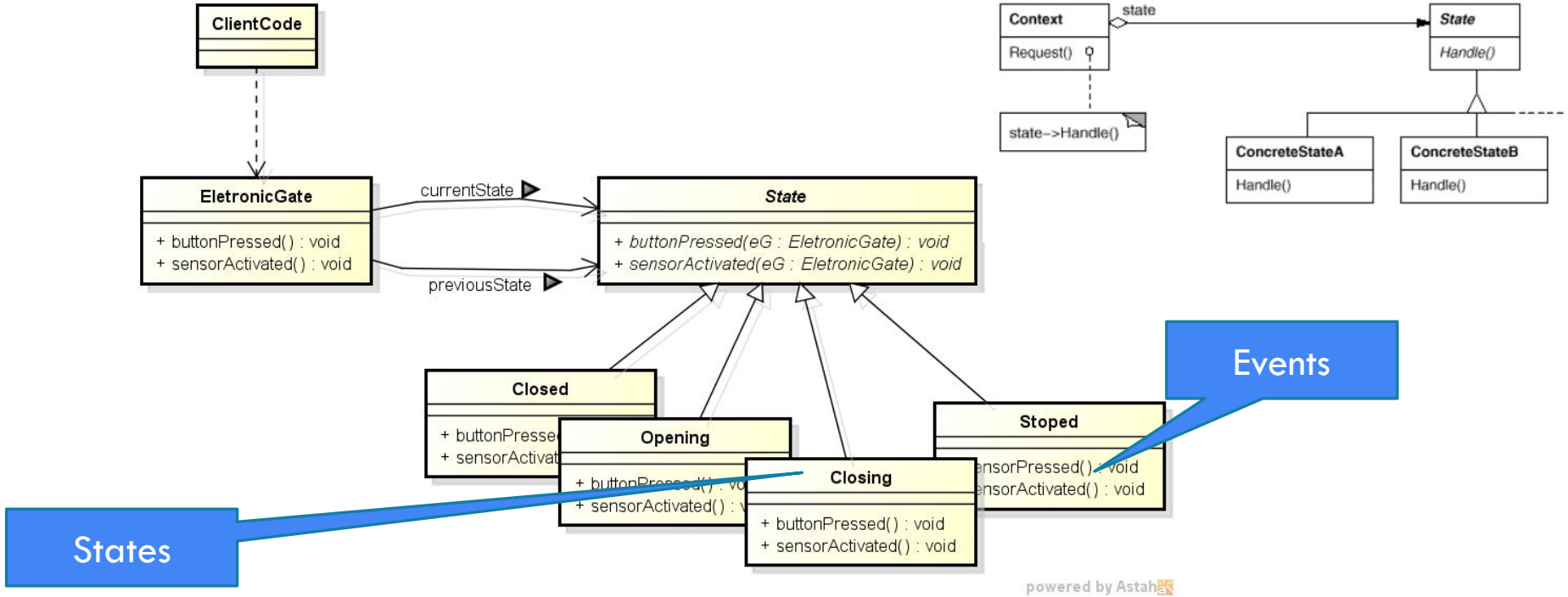
# STATE PATTERN - Generic

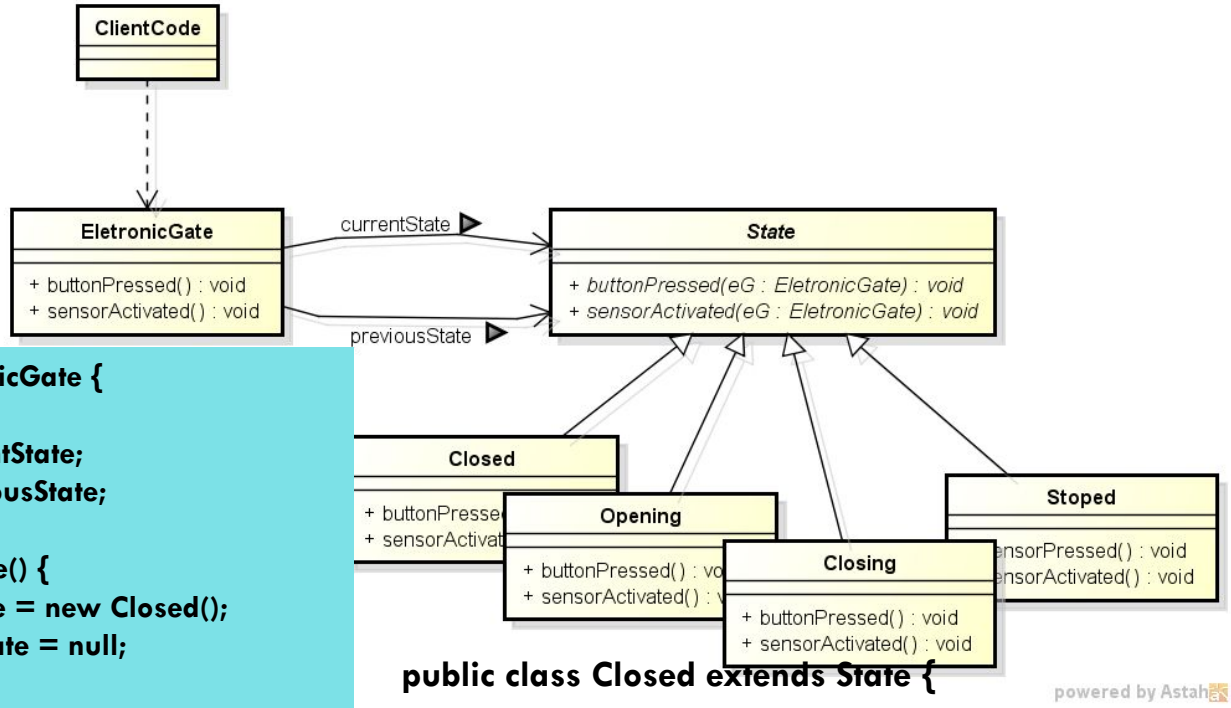


# STATE PATTERN - Electronic Gate



# STATE PATTERN - Eletronic Gate



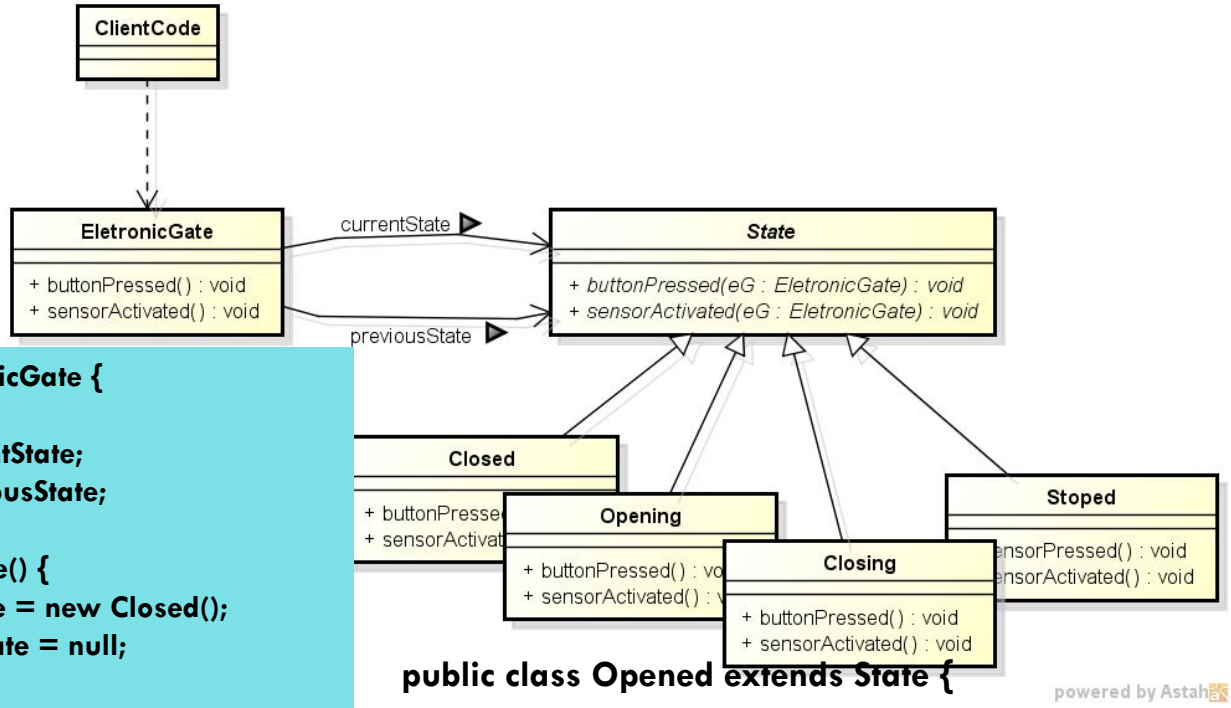



```
public class EletronicGate {

private State currentState;
private State previousState;

public EletronicGate() {
     this.currentState = new Closed();
     this.previousState = null;
}
public void buttonPressed() {
   this.currentState.buttonPressed(this);
}
public void sensorActivated() {
this.currentState.sensorActivated (this);
}
}
```

Delegation

```
public class Closed extends State {

public void buttonPressed(EletronicGate eletronicGate) {
     eletronicGate.setCurrentState(new Opening());
     eletronicGate.setPreviousState(this);
}
public void sensorActivated(EletronicGate eletronicGate) {
     //nothing....
}
```

State Transition

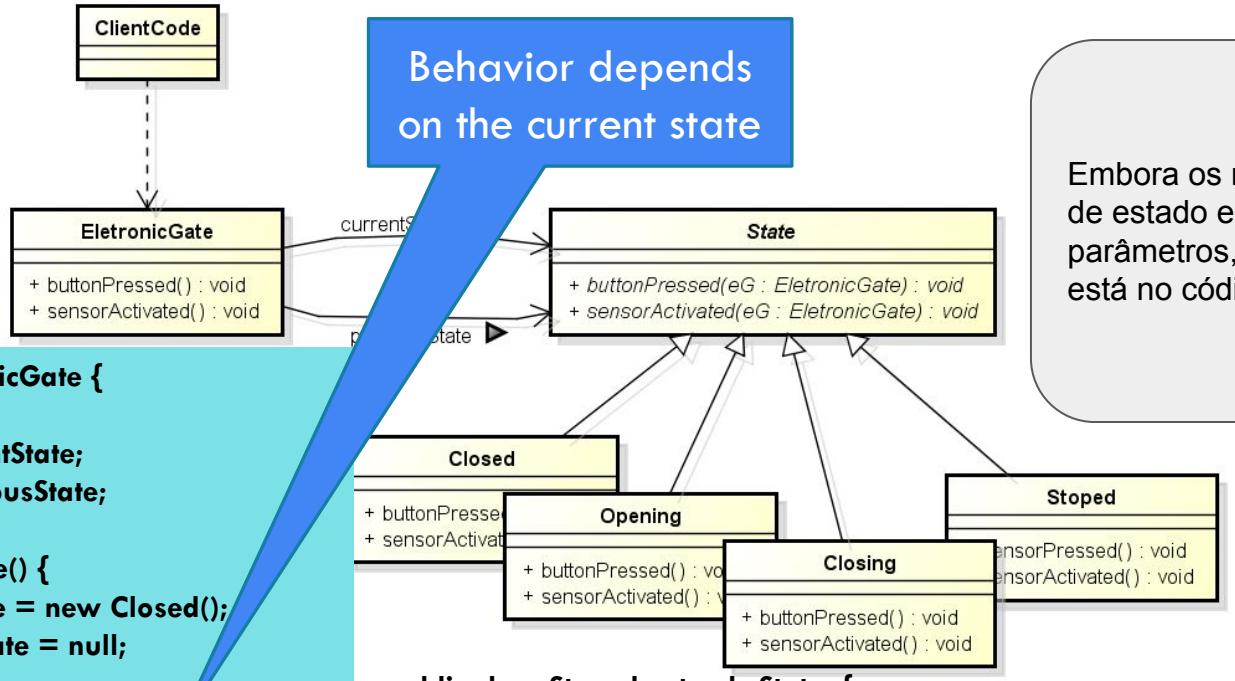



```
public class EletronicGate {

private State currentState;
private State previousState;

public EletronicGate() {
     this.currentState = new Closed();
     this.previousState = null;
}
public void buttonPressed() {
   this.currentState.buttonPressed(this);
}
public void sensorActivated() {
this.currentState.sensorActivated (this);
}
}
```

Delegation

```
public class Opened extends State {

public void buttonPressed(EletronicGate eletronicGate) {
     eletronicGate.setCurrentState(new Closing());
     eletronicGate.setPreviousState(this);
}
public void sensorActivated(EletronicGate eletronicGate) {
     //nothing....
}
```

State Transition




Embora os métodos nas classe de estado estejam sem parâmetros, elas possuem, como está no código-fonte !

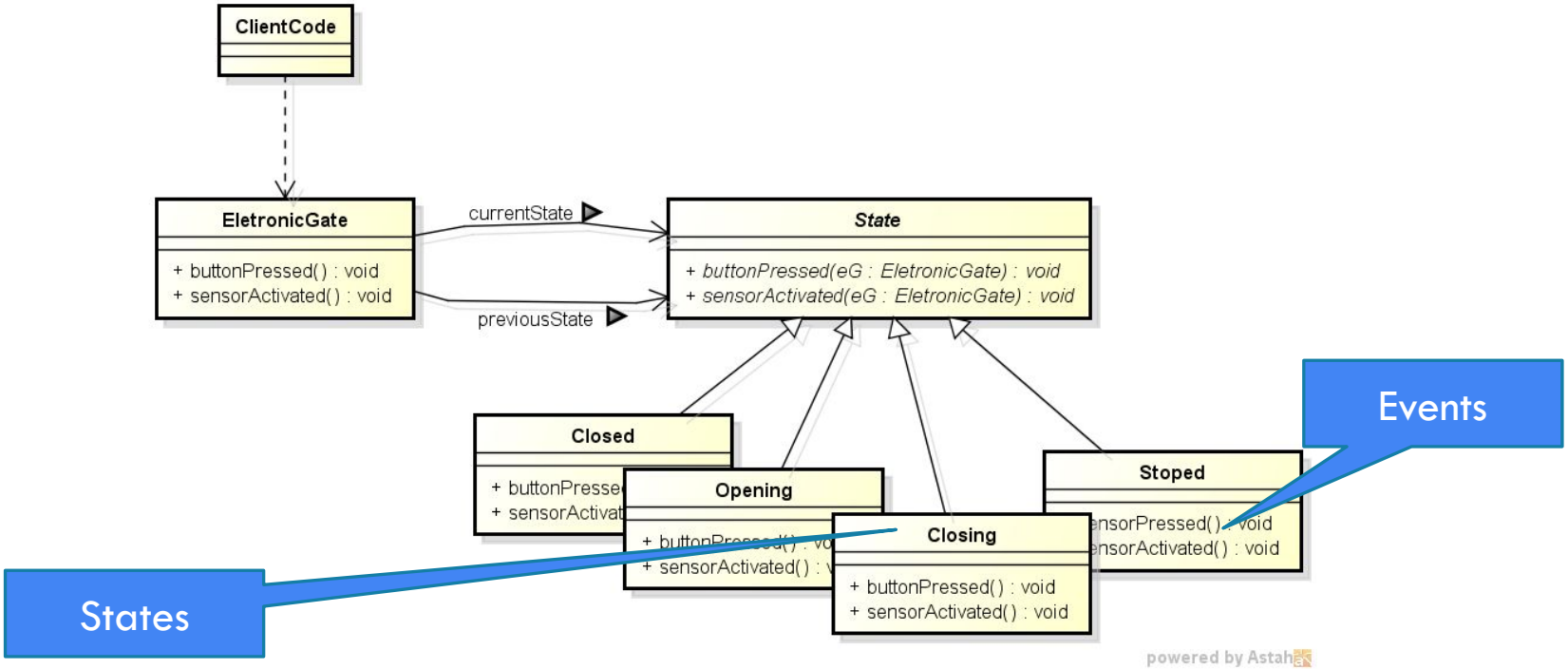
```
public class EletronicGate {

private State currentState;
private State previousState;

public EletronicGate() {
     this.currentState = new Closed();
     this.previousState = null;
}
public void buttonPressed() {
   this.currentState.buttonPressed(this);
}
public void sensorActivated() {
this.currentState.sensorActivated (this);
}
}
```

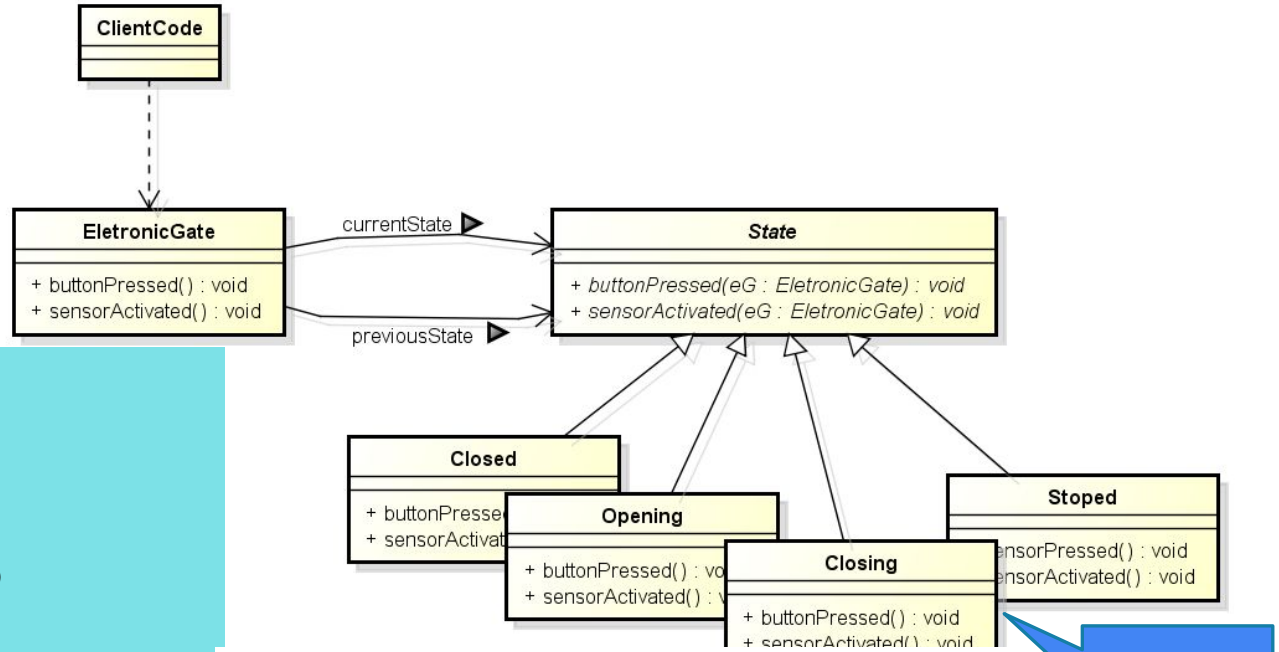
```
public class Stoped extends State  {
public void buttonPressed(EletronicGate eletronicGate) {
    if (eletronicGate.getPreviousState().indexOf("Opening") >= 1) {
            eletronicGate.setCurrentState(new Closing());
     } else {
             eletronicGate.setCurrentState(new Opening());
     }
     eletronicGate.setPreviousState(this);
}
}
```

Guards

Transitions

# STATE PATTERN - Eletronic Gate Python






```
class EletronicGate():

     currentState: State
    _previousState: State

    def __init__(self) -> None:
        self._currentState = Closed()
        self._previousState = None
        pass

    def buttonPressed(self) -> None:
        self._currentState.buttonPressed(self)

    def sensorActivated(self) -> None:
        self._currentState.sendorActivated(self)
```

Delegation

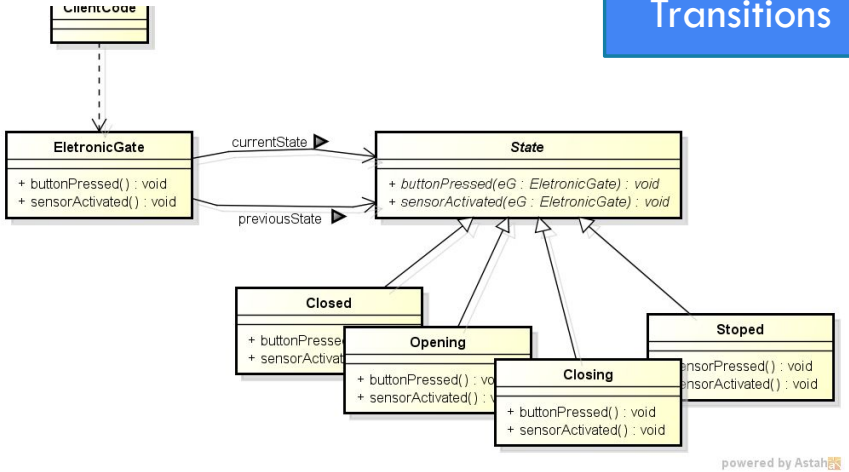
```
class Closed(State):
    def   init  (self) -> None:
          super().__init__()

    def buttonPressed(self, eletronicGate: EletronicGate) ->
None:
        eletronicGate.setCurrentState(Opening())
        eletronicGate.setPreviousState(self)

    def sensorActivated(self, eletronicGate: EletornicGate) ->
None:
         pass
```

Behavior depends on the current state

```
class EletronicGate():

     currentState: State
    _previousState: State

    def __init__(self) -> None:
        self._currentState = Close( )
        self._previousState = None
        pass

    def buttonPressed(self) -> None:
        self._currentState.buttonPressed(self)

    def sensorActivated(self) -> None:
        self._currentState.sendorActivated(self)
```

```
class Stoped(State):
    def   init  (self) -> None:
        super().__init__()

    def buttonPressed(self, eletronicGate: EletornicGate) -> None:
        if (eletronicGate.getPreviousState().indexOf("Opening") >= 1):
            eletronicGate.setCurrentState(Closing())
        else:
            eletronicGate.setCurrentState(Opening())

        eletronicGate.setPreviousState(self)

    def sensorActivated(self, eletronicGate: Eletorn    e) -> None:
        pass
```

Transitions



# ACTIVITY

Você recebeu a incumbência de desenvolver um sistema para gerenciamento de elevadores. São elevadores básicos, que sobem, descem, páram no andar desejado, emperram e ficam ociosos em manutenção de vez em quando…

Desenvolva uma solução para isso por meio de :

- Diagrama de classes
- Trechos de código em Java/Python para este sistema.

# Observer

(behavior pattern)


INFORMAÇÃO
TECNOLOGIA
& INOVAÇÃO

# OBSERVER

**When to use?**

Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically.

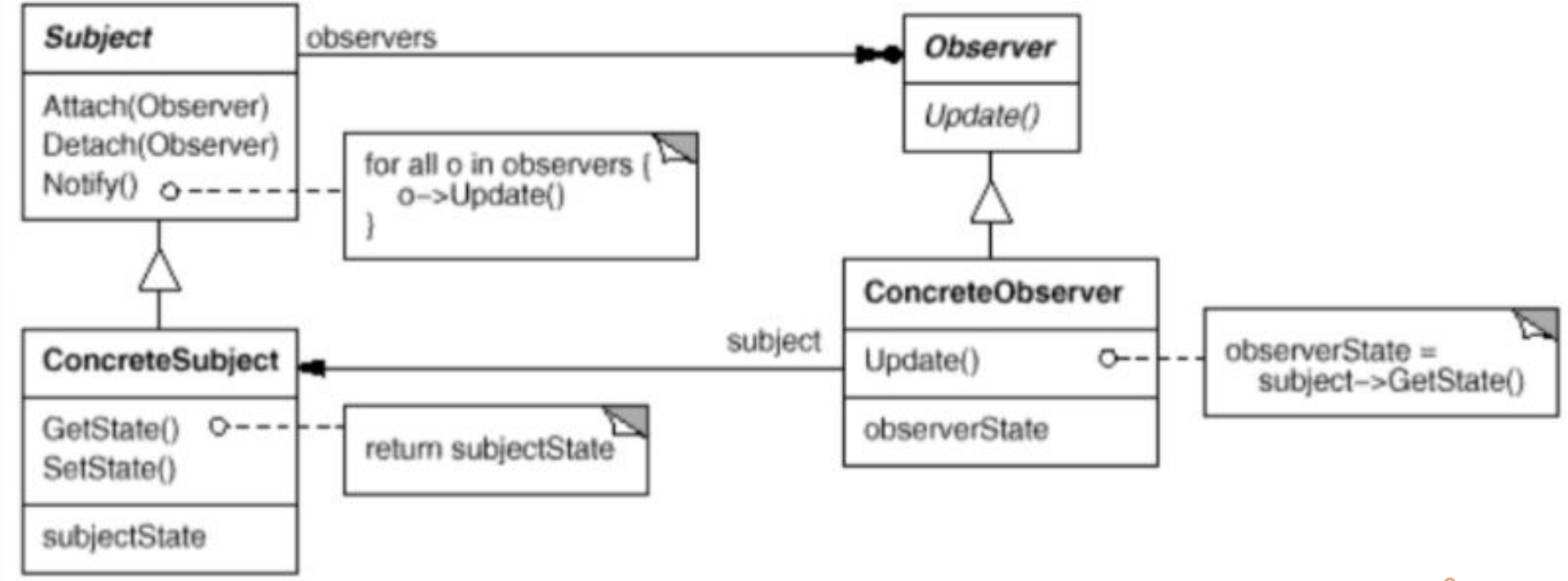
# OBSERVER

## Example:

If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

The **publisher** maintains a list of subscribers and knows which magazines they're interested in. **Subscribers** can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

# OBSERVER



# OBSERVER

```python
class Subject():

    def attach(self, observer: Observer) -> None:
        raise NotImplementedError

    def detach(self, observer: Observer) -> None:
        raise NotImplementedError

    def notify(self) -> None:
        raise NotImplementedError


class ConcreteSubject(Subject):

    _state: int = None

    _observers: List[Observer] = []

    def attach(self, observer: Observer) -> None:
        print("Subject: Attached an observer.")
        self._observers.append(observer)

    def detach(self, observer: Observer) -> None:
        self._observers.remove(observer)

    def notify(self) -> None:

        print("Subject: Notifying observers...")
        for observer in self._observers:
            observer.update(self)

    def some_business_logic(self) -> None:
        print(
         "\nSubject: I'm doing something important."
        )
        self._state = randrange(0, 10)

        print(f"Subject: My state has just changed
              to: {self._state}")
        self.notify()
```

# OBSERVER

```
class Observer():

    def update(self, subject: Subject) -> None:
        raise NotImplementedError

—--------------------

class ConcreteObserverA(Observer):
    def update(self, subject: Subject) -> None:
        if subject._state < 3:
            print("ConcreteObserverA: Reacted to the event")

—--------------------

class ConcreteObserverB(Observer):
    def update(self, subject: Subject) -> None:
        if subject._state == 0 or subject._state >= 2:
            print("ConcreteObserverB: Reacted to the event")
```

# EXEMPLO DE OBSERVER/LISTENERS...

```
public class TestProgram {

   public static void main(String[] args){
      TestProgramm a = new TestProgramm ();
      a.go();
   }

   public void go (){
      MyLightListener lightListener = new MyLightListener (); //concrete observer
      MyTouchListener touchListener = new MyTouchListener (); //concrete observer

      LightSensor light = new LightSensor (SensorPort.S1);
      TouchSensor touch = new TouchSensor (SensorPort.S2);

      SensorPort.S1.addSensorPortListener(lightListener);  //ATTACH
      SensorPort.S2.addSensorPortListener(touchListener);  //ATTACH

      LCD.drawString("LightListener:", 0, 1);
      LCD.drawString("TouchListener:", 0, 4);

      Button.waitForPress();
  }
```



When the state of the subjects (sensors/ports) change, the listeners will be automatically notified...

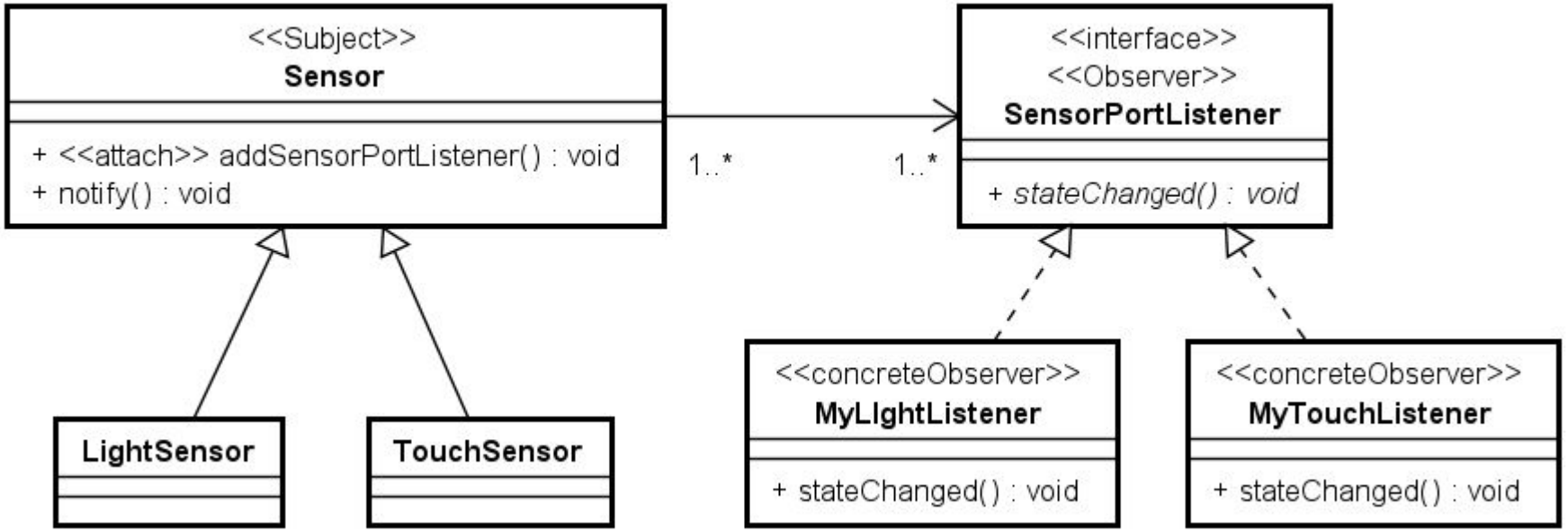
# EXEMPLO DE OBSERVER/LISTENERS...

These are the “update” methods !

```
class MyLightListener implements SensorPortListener {
    @overrides

    public void stateChanged(SensorPort source, int oldValue, int newValue) {
         LCD.drawString("alt: "+ oldValue + "   neu: " + newValue, 0, 2);
      }
   }

class MyTouchListener implements SensorPortListener {

     @overrides
     public void stateChanged (SensorPort source, int oldValue, int newValue){
           LCD.drawString("alt: " + oldValue + "   neu: " + newValue, 0, 5);
      }

}
```

Notice that there is no need to implement the **notify()** operation ….

# EXEMPLO DE OBSERVER/LISTENERS...



# Atividade

**Requisito Funcional:**

*Um novo empreendimento imobiliário está com intenção de desenvolver um sistema inteligente de* ***monitoramento*** *de elevadores. Você foi incumbido de projetar uma solução computacional que gerencia o controle de movimentação dos vários elevadores do edifício. Os elevadores são convencionais, podendo* ***subir****,* ***descer*** *e* ***parar*** *nos andares solicitados.*

*A inteligência do sistema está no fato de que o edifício deve* ***monitorar*** *o funcionamento dos elevadores para proporcionar mais segurança e também uma experiência de uso mais agradável para os moradores. As ações que o sistema pode tomar variam desde ações de segurança até mesmo a ações de entretenimento, como reprodução de músicas dentro do elevador.*

*Exemplos:*

*- Se um determinado elevador* ***emperrar*** *durante seu funcionamento, o sistema deve perceber e automaticamente invocar a equipe de manutenção e segurança. Enquanto isso uma música agradável pode começar a ser reproduzida naquele elevador específico para acalmar o usuário.*

*- Se um determinado elevador* ***estiver em manutenção****, o sistema pode, entre outras ações, interferir na velocidade de deslocamento dos outros elevadores.*

*- Se um determinado elevador estiver em movimentação (subindo ou descendo), dependendo do peso interno a velocidade do elevador pode ser alterada*

**Requisitos Não-Funcionais:**

*- O código cliente que manipula os elevadores deve ser o mais inconsciente possível dos objetos concretos existentes*

*- Nesta primeira versão do sistema, um elevador nunca estará em dois estados ao mesmo tempo.*

*- Embora não haja limitação de memória na infraestrutura computacional usada, deve-se prezar para que não haja* ***criação desnecessária de objetos em memória****.*