Software Design Patterns: Session Five

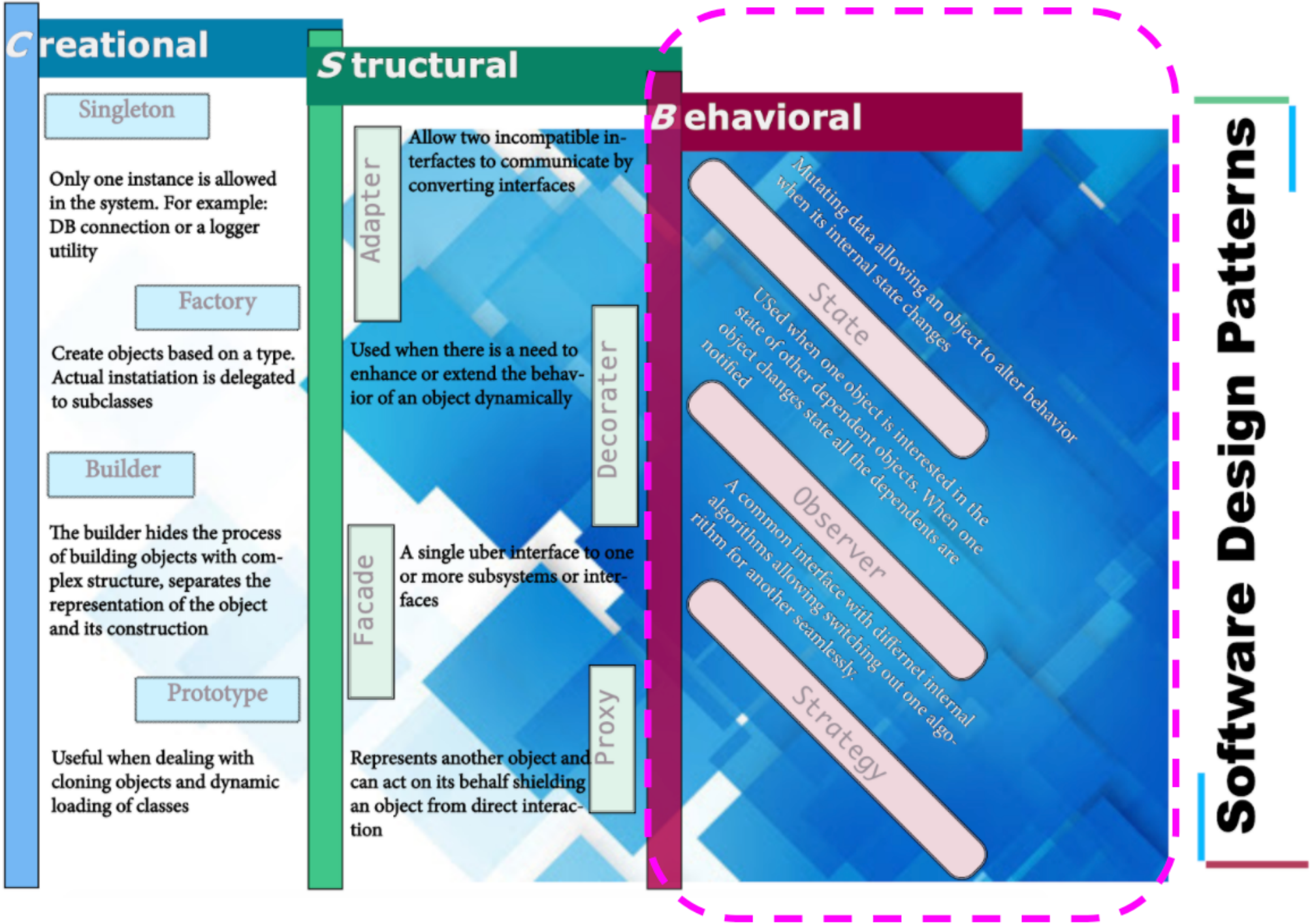
Sahar Mostafa
UC Berkeley Extension
Summer 2020

Half moon bay

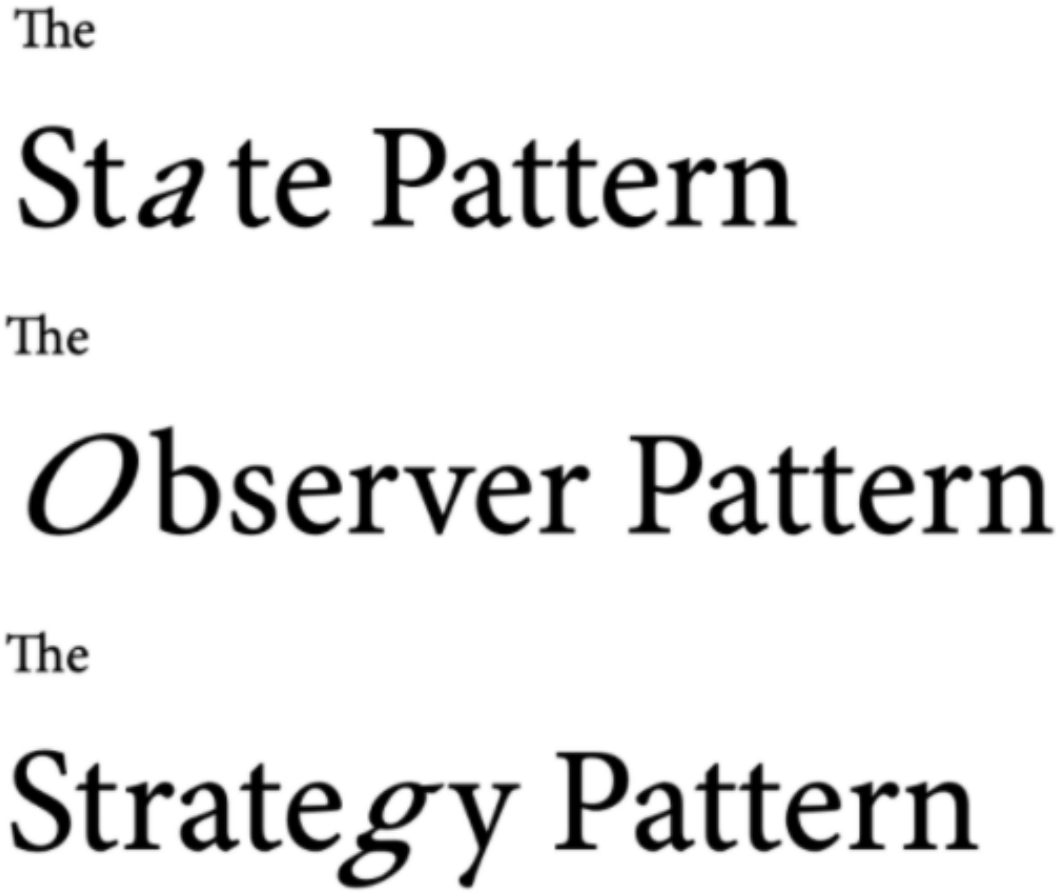
Week In Review

1. Structural Design Patterns
2. Adaptor Pattern
3. Decorator Pattern
4. Facade Pattern
5. Proxy Pattern
6. Exercises

Software Design Patterns



Behavioral Patterns



Description

Intent - Algorithms and Responsibilities

Patterns of communications between objects or classes

Focus is shifted from the overall flow to how objects are interconnected

Control over how a system behaves and cooperates internally

Behavior

State Pattern

Description

Intent

Allow an object to alter behavior when its internal state changes

Motivation

Object to respond differently based on the current state

Consequences

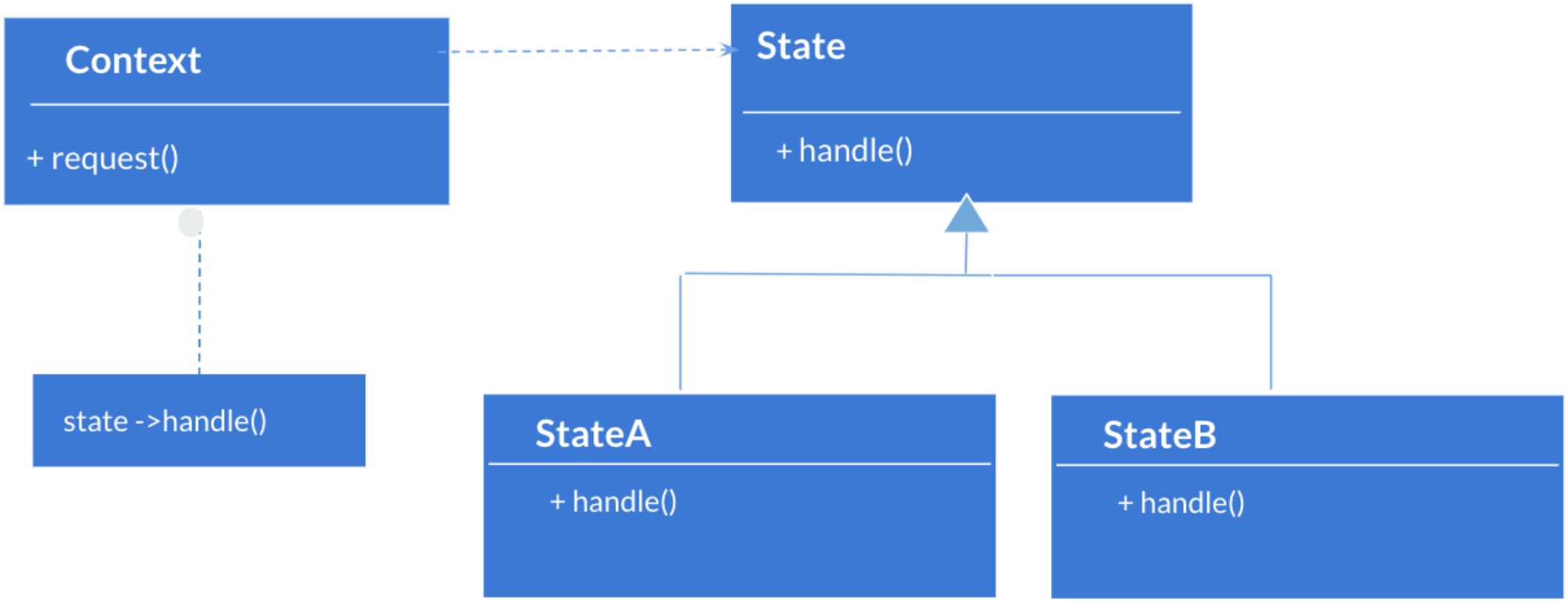
Encapsulate all the behavior associated with one state into one object

State transitions are explicit

Use cases

TCP Connection changing state from Established to Listening to Closed

UML Diagram


Context
+ request()
state ->handle()
State
+ handle()
StateA
+ handle()
StateB
+ handle()

Implementation

```
class TCPConnection {
public:
  void activeOpen();
  void send();
private:
  TCPState* _tcpState;
};

class TCPState {
public:
  virtual void
activeOpen(TCPConnection* connection);
  virtual void send(TCPConnection*
connection);
  void changeState(TCPState* state);
};
```

```
class TCPListen : public
TCPState {
public:
  Static TCPState*
Instance();
  void send(TCPConnection*
connection);
private:
  TCPState* _tcpState;
};

void
TCPListen::send(TCPConnecti
on* conn) {
changeState(conn,
TCPClosed::Instance());
```

```
void TCPConnection:TCPCon
nection() {
  _tcpState =
TCPClosed::Instance();
}
void TCPConnection:activeOpen() {
  _tcpState->activeOpen(this);
}
void TCPConnection:send() {
  _tcpState->send(this);
}
```

Behavior

Observer Pattern

Description

Intent

Define a dependency between objects when one object changes state, all dependents are notified to take action

Motivation

Maintain consistency in between external cooperating classes

Changing an object requires changing others

Consequences

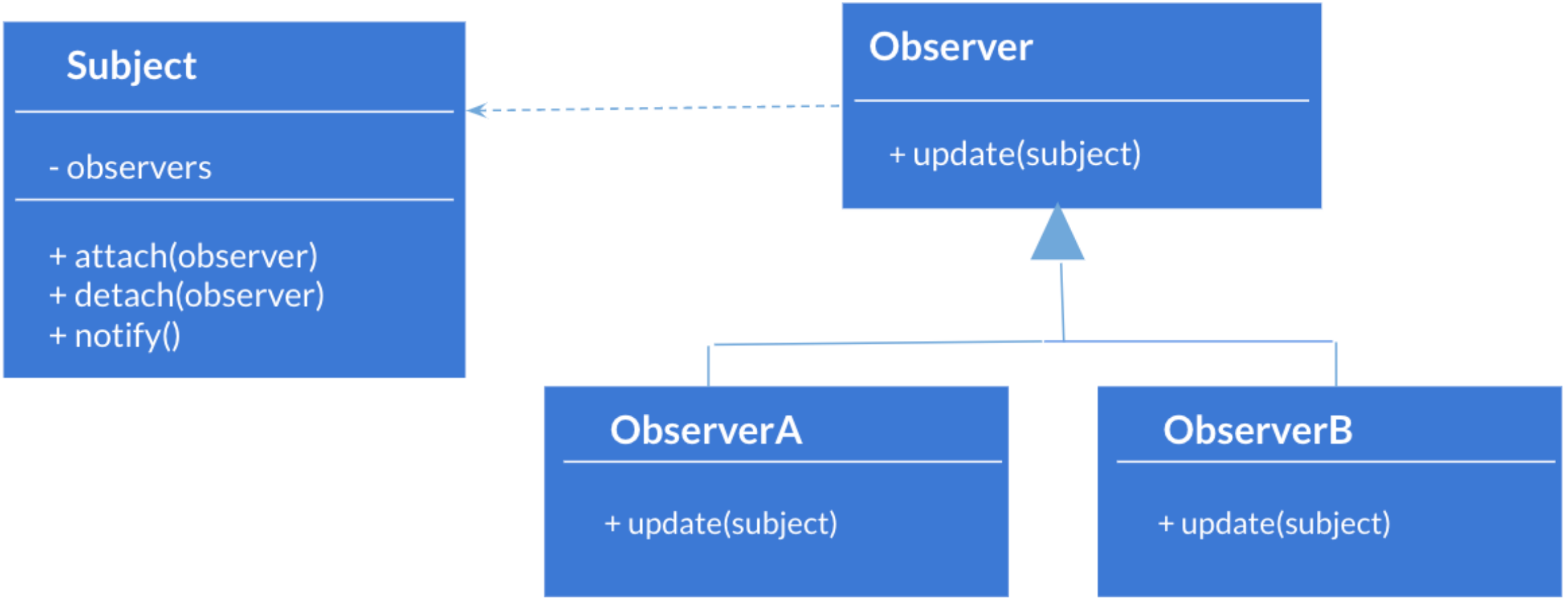
Abstract coupling between observers and subjects

Supports broadcast notifications

Use cases

Push vs pull subscription model

UML Diagram


Subject
- observers
+ attach(observer)
+ detach(observer)
+ notify()
Observer
+ update(subject)
ObserverA
+ update(subject)
ObserverB
+ update(subject)

Implementation

```
public class Subject {
private List<Observer> observers = new ArrayList<Observer>();
   private int state;
   public int getState() {
      return state;
   }
   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }
   public void attach(Observer observer) {
     observers.add(observer);
  }
  public void notifyAllObservers() {
     for (auto observer : observers)
         observer.update();
}
```

```
public abstract class Observer {
protected Subject subject;
public abstract void update();
}

public class EventObserver extends Observer{

   public EventObserver(Subject subject) {
      this.subject = subject;
      this.subject.attach(this);
   }
   @Override
   public void update(Subject subject) {
      if (subject == _subject) {}
    }
}  }
```


15 mins Break

Structure

Strategy Pattern

Description

Intent

Define algorithms make them interchangeable

Vary implementation independently from clients

Motivation

Different algorithms are appropriate at different times

Difficult to add new algorithms while decoupling from the client

Consequences

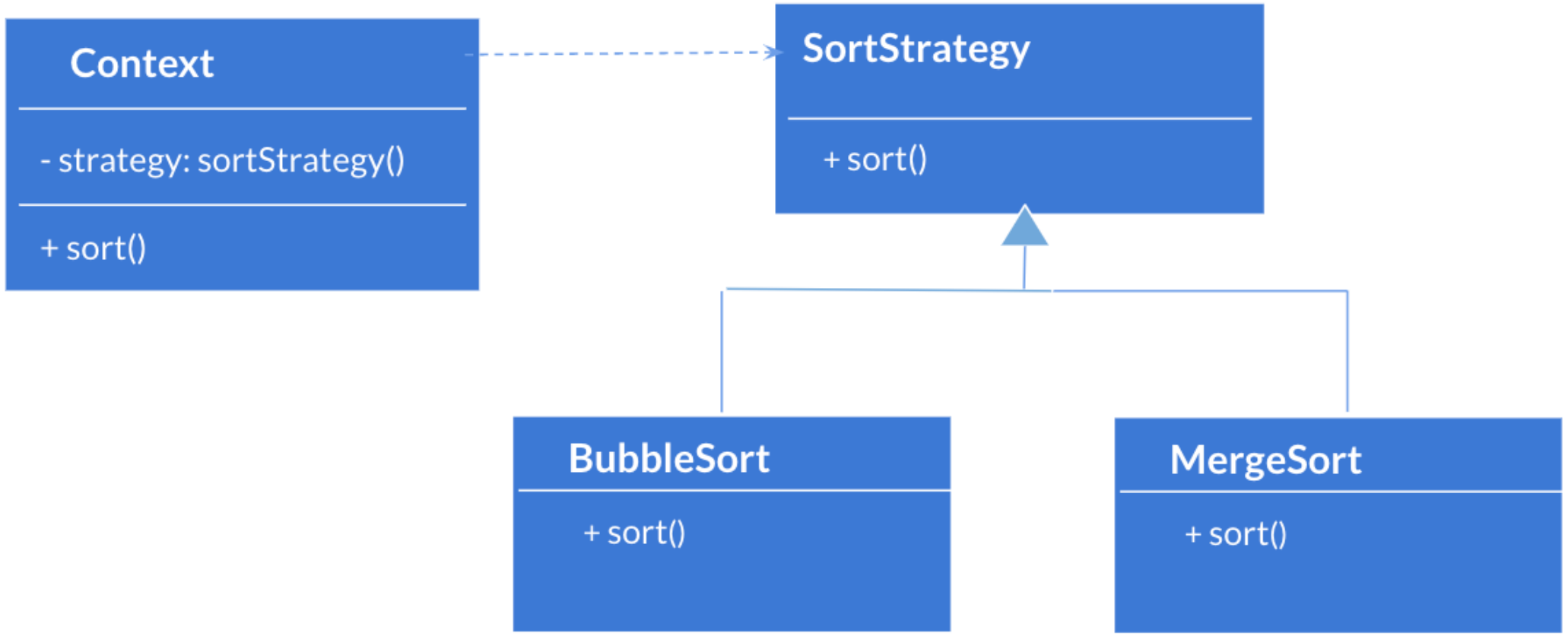
Alternative to subclassing by encapsulating algorithm under a separate strategy

Eliminate conditional statements to select desired behavior

Use cases

Sorting algorithms

UML Diagram


Context
- strategy: sortStrategy()
+ sort()
SortStrategy
+ sort()
BubbleSort
+ sort()
MergeSort
+ sort()

Implementation

```
interface SortStrategy {
 public void sort(int[] values);
}

class BubbleSort implements SortStrategy {
 public void sort(int[] values) {}
}

class MergeSort implements SortStrategy {
 public void sort(int[] values) {}
}
```

```
class Context {
 private final SortStrategy sortStrategy;

 public Context(SortStrategy sortStrategy) {
  this.sortStrategy = sortStrategy;
 }

 public void printSorted(int[] values) {
     sortStrategy.sort(values);
 }
}
```

Structure

Chain of Responsibility Pattern

Description

Intent

One received object multiple handlers

Motivation

Decouple sender from receiver by allowing more than one object to handle the request

Consequences

Same object is passed to successors till one object processes the request

Added flexibility in assigning responsibility to objects - No guaranteed receipt

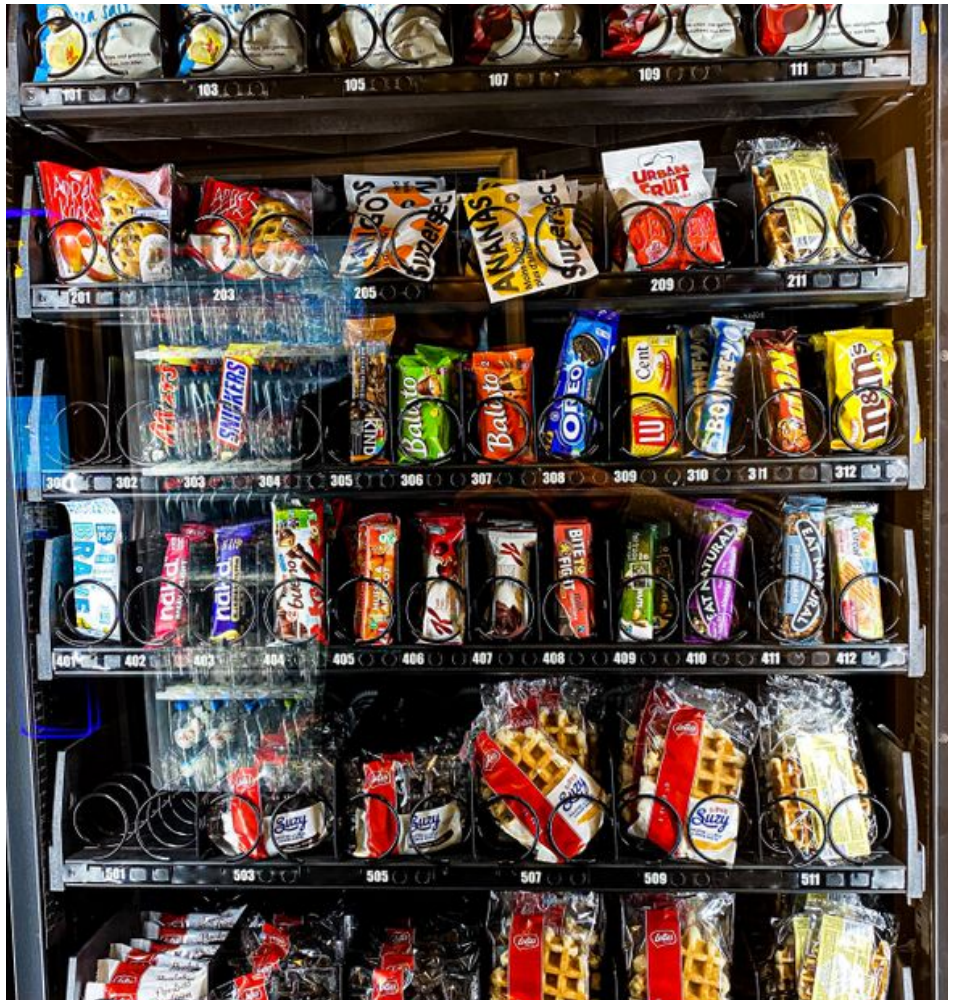
Use cases

Cocoa and Cocoa iOS and OSX Touch frameworks


15 mins Break

Exercises

Vending Machine



- **Design a vending machine with the following actions:**

  User selection: vending machine to display selections

  User paying: vending machine to accept form of payments

  Deliver product: vending machine to dispense the item selected by user

- **Which behavioral design pattern to use?**

- **Handle error cases**

- **Provide UML class diagram and test cases highlighting how the classes can be called**


15 mins Break

Event Notification

- Design an event notification system that alerts users when a new event is added to the system. Users should be able to register to notifications and act accordingly to reserve a spot
- Which structural design pattern to use?
- Provide UML class diagram and test cases highlighting how the classes can be called

Code Review

```
class Rental...
  double getCharge() {
      double result = 0;
      switch (getMovie().getPriceCode()) {
          case Movie.REGULAR:
              result += 2;
              if (getDaysRented() > 2)
                  result += (getDaysRented() - 2) * 1.5;
              break;
          case Movie.NEW_RELEASE:
              result += getDaysRented() * 3;
              break;
          case Movie.CHILDRENS:
              result += 1.5;
              if (getDaysRented() > 3)
                  result += (getDaysRented() - 3) * 1.5;
              break;
      }
      return result;
  }
```

source: Martin Fowler, Refactoring: Improving the design of Existing Code

Next Week

Session Six

1. Review Design Patterns Session
2. Mid term Assignment
3. Group Exercises
4. Introduction to Cloud Services