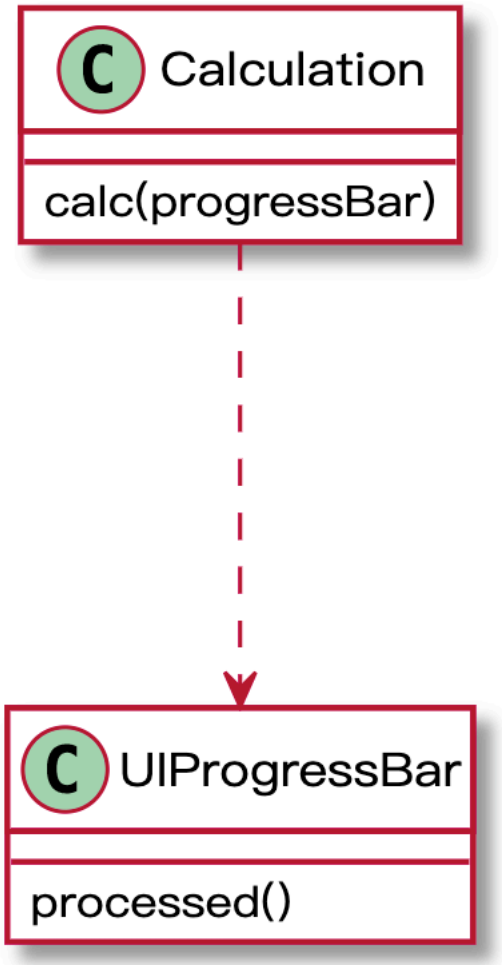
Observer Pattern

场景1：进度条

- Calculation.calc中有比较复杂的计算，可以分解为n步，每次执行一步需要更新进度条。



场景1：进度条

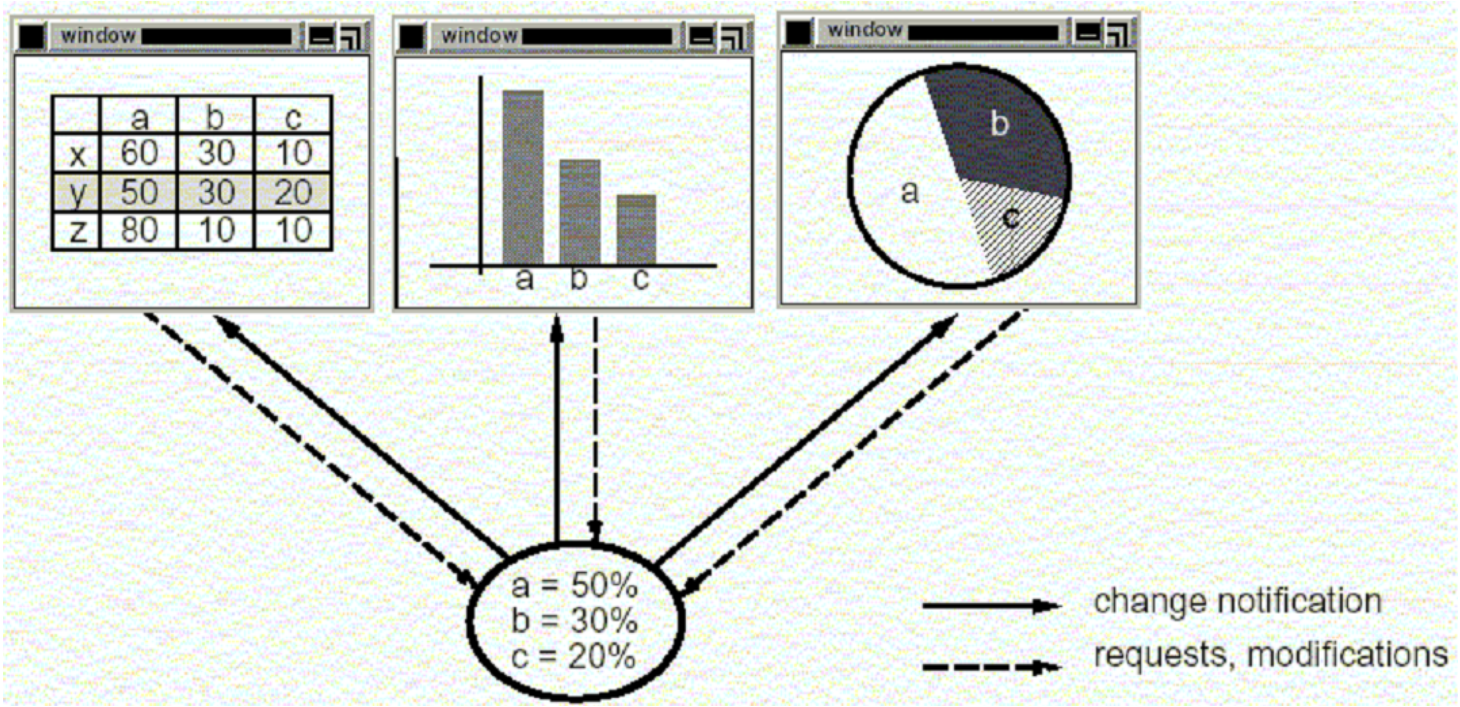
- 简单的方法如图所示。
- 但如果需要对Calculation.calc写自动测试脚本会比较困难

```
class UIProgressBar{
    processed(present);
}

class Calculation{
    calc(bar:UIProgressBar){
        for(i=0; i<n; i++){
            bar.processed(i)
        }
    }
}
```

```
class CalculationTest{
    @Test
    public void normalCalcTest(){
        new Calculation().calc(???)
    }
}
```

场景2：多视图

- 一组数据有多种呈现方式，并且数据发生变化后，所有相关的视图需要同步变话。



场景2：多视图

- 简单实现方法如图所示。
- 问题：
  - 自动化测试的问题
  - 新增加一种呈现方式需要修改Data的代码

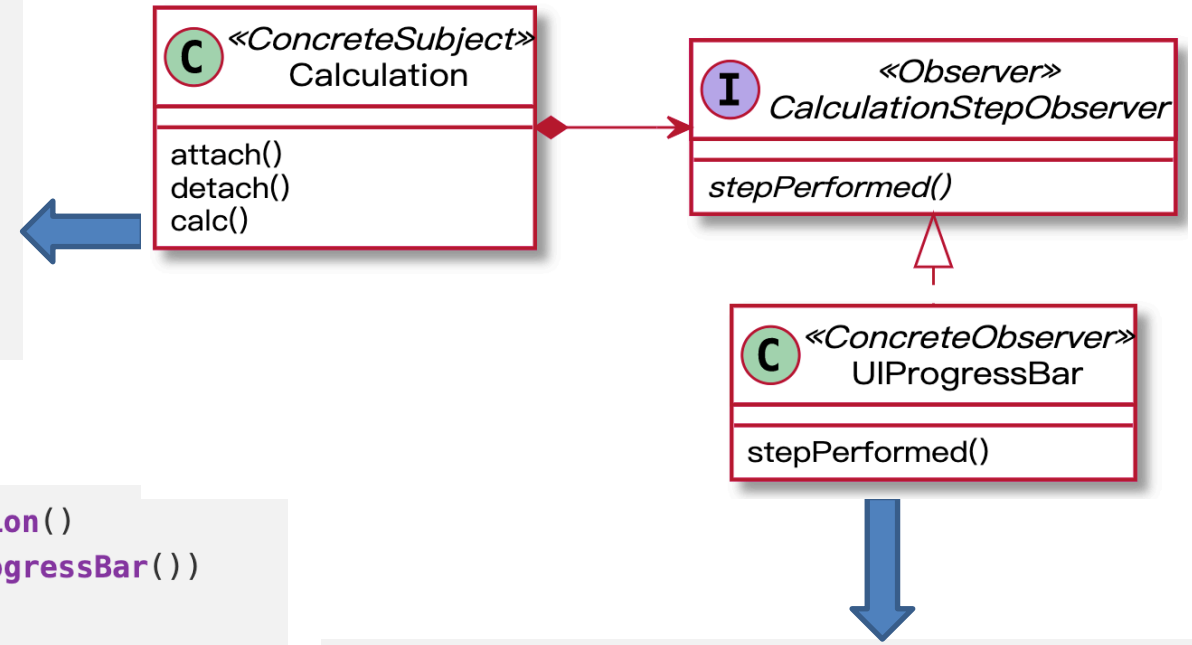
```
class Data{
    private spreadSheet:SpreadSheet;
    private barChart:BarChart;
    private pieChart:pieChart;

    public updateData(a,b,c){
        ...
        spreadSheet.setData(a,b,c);
        barChart.redraw(a,b,c);
        pieChart.update(a,b,c);
        ...
    }
}
```

Observer

- Motivation
  - A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
  - You don’t want to achieve consistency by making the class tightly coupled, because that reduces their reusability.

解决方案：进度条

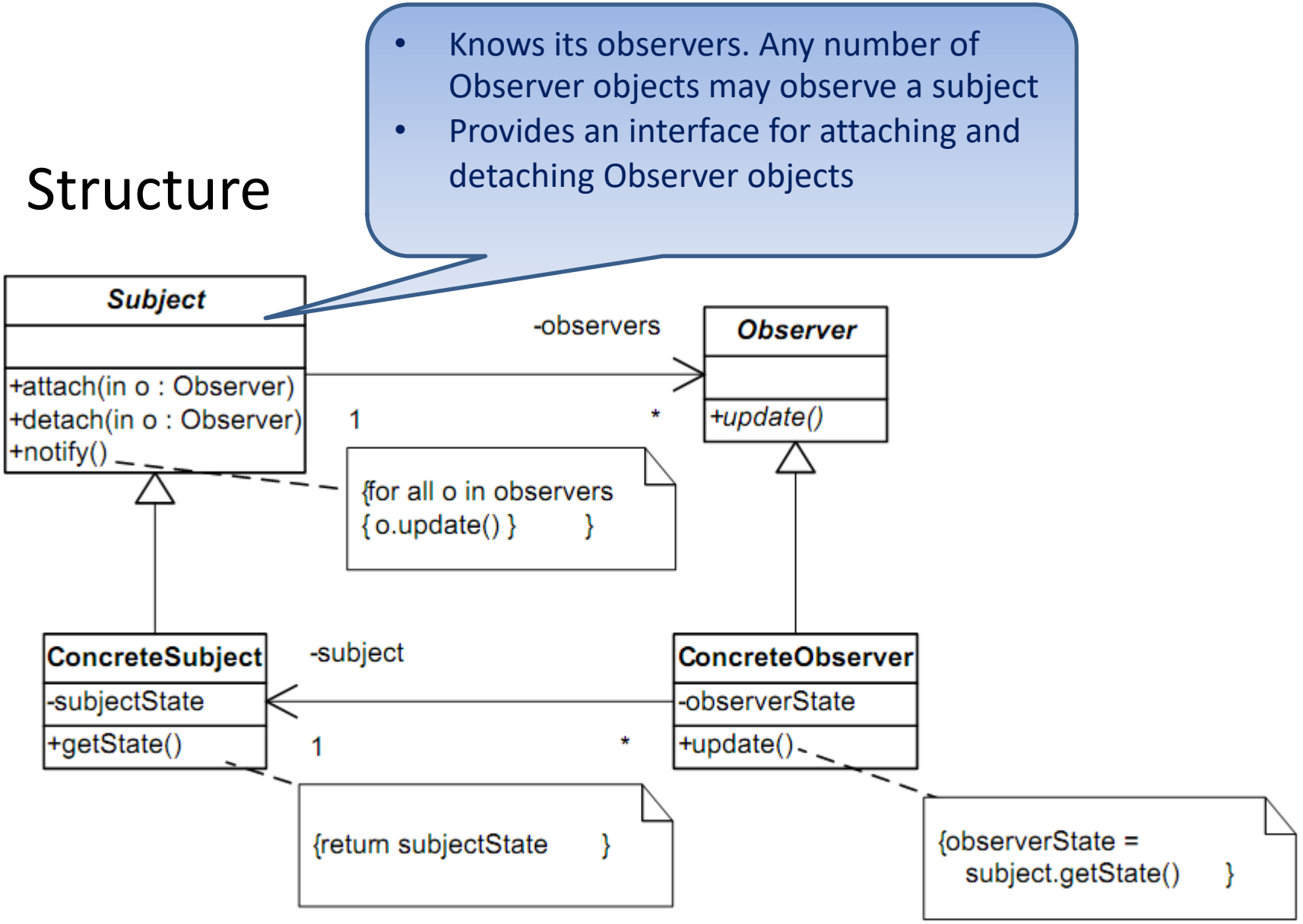
```
calc(){
    for(let i=0;i<10;i++){
        //...perform calculation
        this.listeners.forEach(
            x=>x.stepPerformed()
        )
    }
}
```



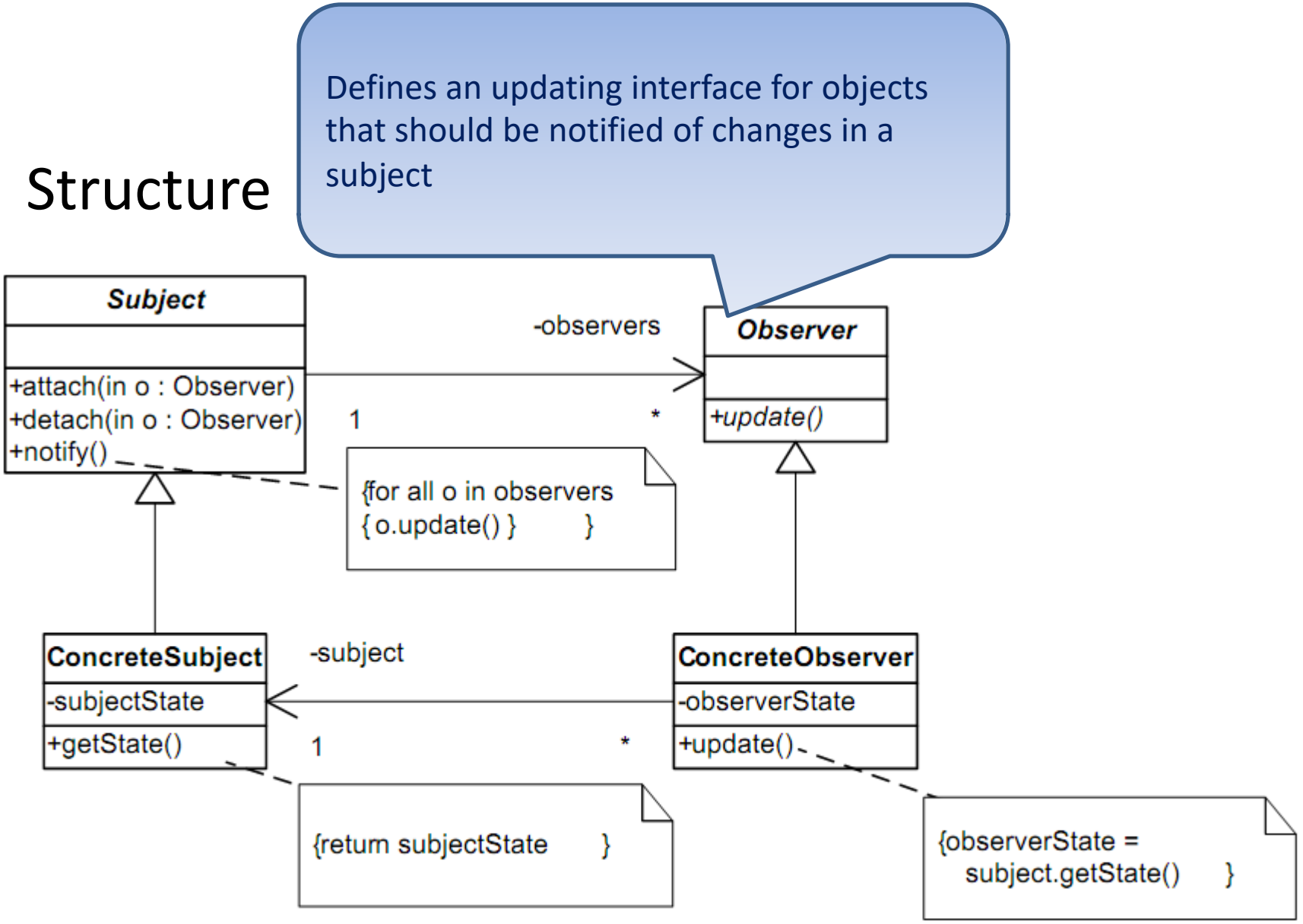

```
const calculation = new Calculation()
calculation.addListener(new UIProgressBar())
calculation.calc()
/**
 * output:
    0
    00
    000
    0000
    00000
    000000
    0000000
    00000000
    000000000
 */
```

```
stepPerformed(): void {
    this.step = this.step + 1
    console.log(Array(this.step).join("0"))
}
```

- Structure

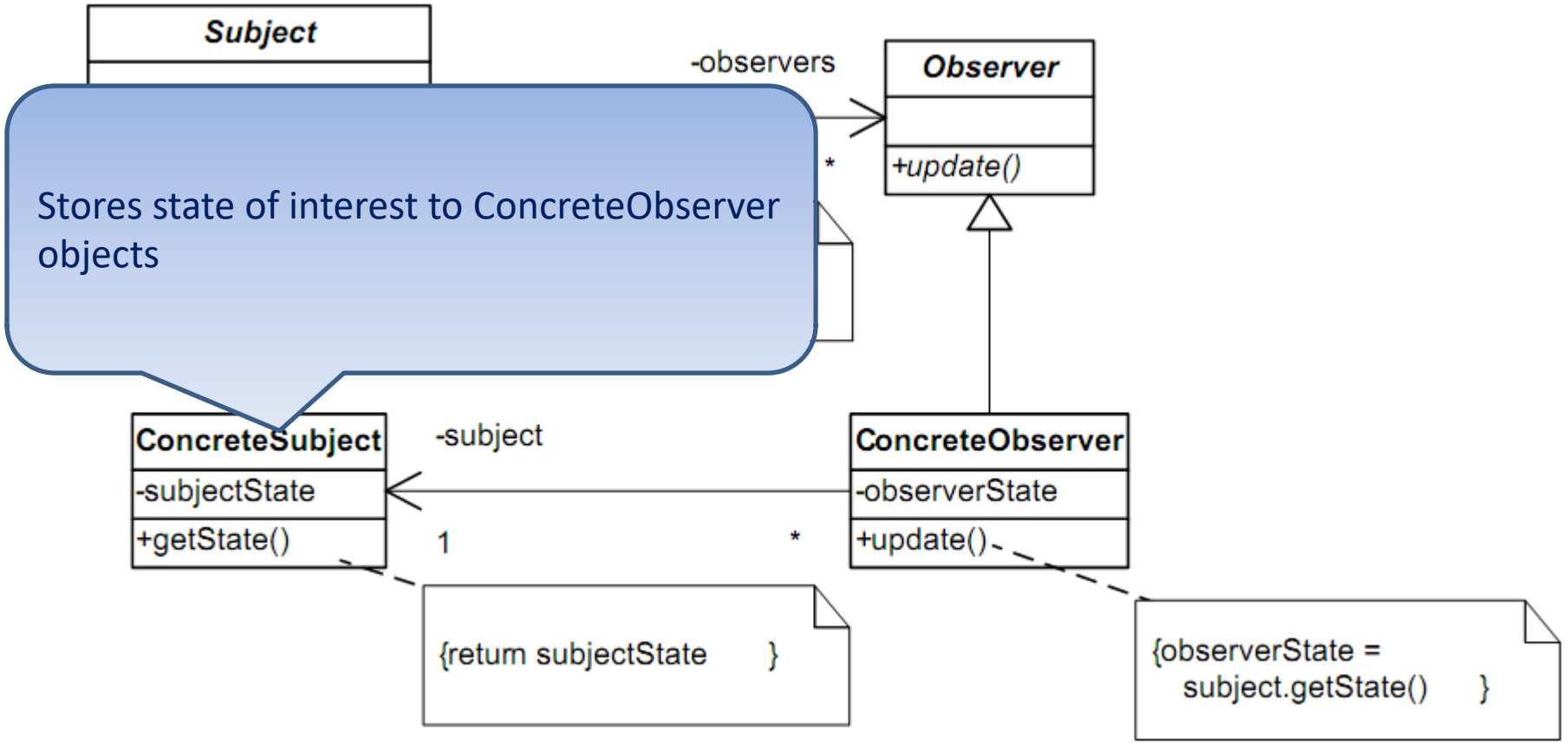

Knows its observers. Any number of Observer objects may observe a subject
Provides an interface for attaching and detaching Observer objects
Subject
+attach(in o : Observer)
+detach(in o : Observer)
+notify()
-observers
1
*
Observer
+update()
{for all o in observers
{ o.update() } }
ConcreteSubject
-subjectState
+getState()
-subject
1
*
ConcreteObserver
-observerState
+update()
{return subjectState }
{observerState =
subject.getState() }

- Structure


Defines an updating interface for objects that should be notified of changes in a subject
Subject
+attach(in o : Observer)
+detach(in o : Observer)
+notify()
-observers
1
*
Observer
+update()
{for all o in observers
{ o.update() } }
ConcreteSubject
-subjectState
+getState()
-subject
1
*
ConcreteObserver
-observerState
+update()
{return subjectState }
{observerState =
subject.getState() }

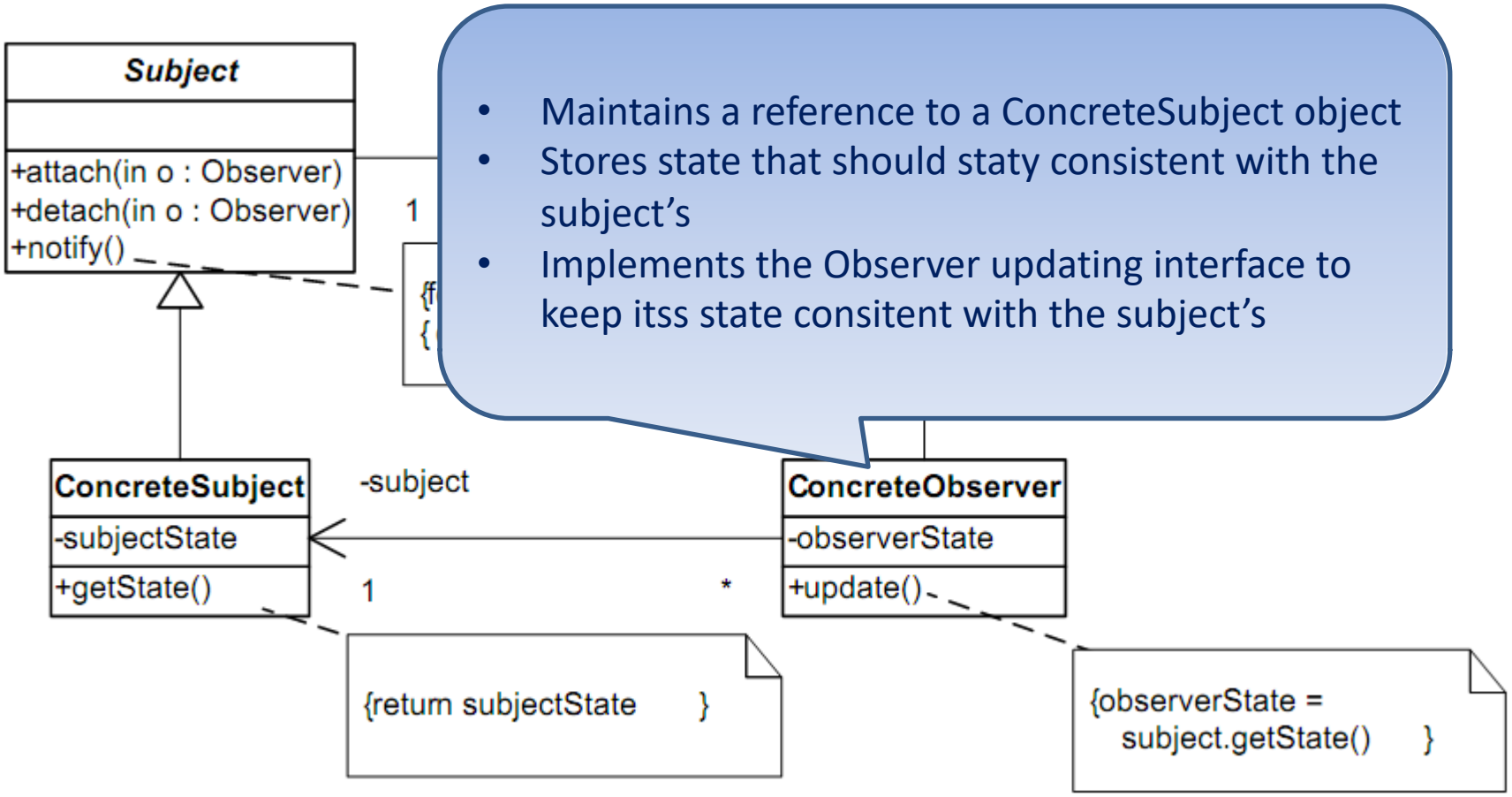
Observer

- Structure

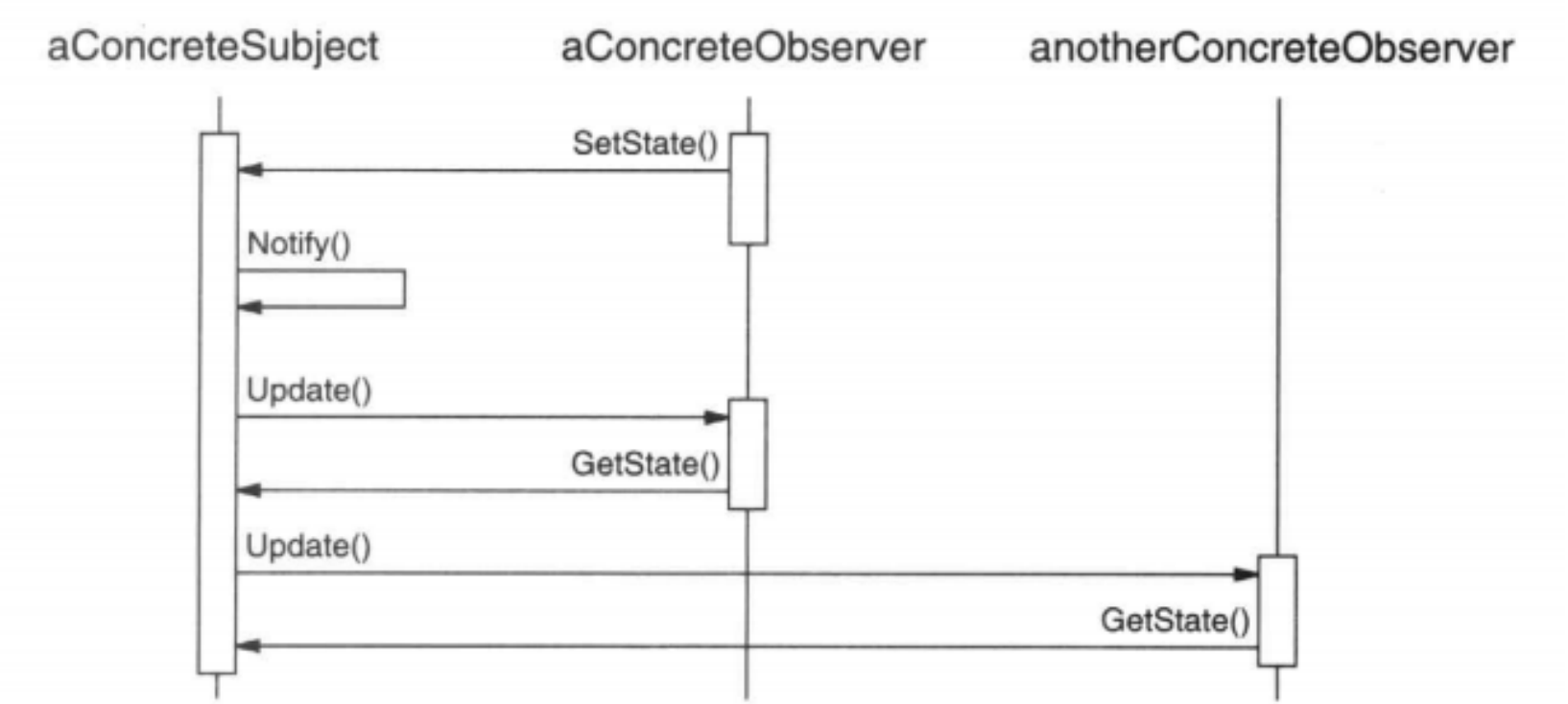


Observer

- Structure



Observer: Collaborations



Observer -Example

- Consider ClockTimer a concrete subject that stores and maintains the time of day. It notifies its observers each second and provides the interface to retrieve time units:

```
class ClockTimer: public Subject {
public:
    ClockTimer();
    virtual intGetHour();
    virtual intGetMinute();
    virtual intGetSecond();
    void Tick();
};
```

```
void ClockTimer::Tick() {
// update internal time-keeping state
// ...
Notify();
}
```

the Tick operation is called by some timer and it updates the ClockTimer's internal state and calls Notify to inform observers.

- consider a DigitialClock concrete Observer:

```
class DigitalClock: public Widget, public Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void Update(Subject*);
    // overrides Observer operation
    virtual void Draw();
   // overrides Widget operation;
   // defines how to draw the digital  clock
private:
    ClockTimer* _subject;
};
```

```
DigitalClock::DigitalClock(ClockTimer* s) {
_subject = s;
_subject->Attach(this);
}
DigitalClock:: ~ DigitalClock() {
_subject->Detach(this);
}
```

- before drawing the clock face, the Update function checks if the notifying subject is the clock's subject:

```
void DigitalClock::Update(Subject* theChangedSubject) {
      Draw();
}
void DigitalClock::Draw() {
    // get the new values from the subject
    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
         // etc.
        // draw the digital clock
}
```

- as well, an AnalogClock class can be defined as:

```
class  AnalogClock: public Widget, public Observer {
public:
     AnalogClock(ClockTimer*);
     virtual void Update(Subject*);
      virtual void Draw();
        // ...
};
```

- can also create an AnalogClock and a DigitalClock that show the same time:

```
ClockTimer*  timer = new  ClockTimer;
AnalogClock*  analogClock= new  AnalogClock(timer);
DigitalClock*  digitalClock= new  DigitalClock(timer);
```

- on each clock tick, both clocks will be updated and will redisplay themselves

Observer

- Consequences
  - +Modularity: subject and observers may vary independently
  - +Extensibility: can define and add any number of observers
  - +Customizability: different observers provide different views of subject

思考题

- 思考1：观察者在处理相关内容时可能会抛出异常。假如S有A，B，C三个观察者，会被依次执行。如果执行B的时候发生异常：
  - 1. A的执行结果是否有效？
  - 2. C是否需要继续执行？

思考题2：

新建一个Employee时，需要同时新建一个User。图示的方法有什么不好的地方？可以如何修改？