The Observer Design Pattern

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

April 10, 2025

Outline

Outline

Software Design

- Design starts mostly from/with requirements – evolving mostly from functionalities and other non-functional characteristics
  - In the waterfall model Design generally occurs after Requirements
  - In agile, design is performed during in each iteration
- To answer: How is the software solution going to be structured?
  - What are the main components – (functional composition) often directly from requirements' functionalities (e.g., use cases, user stories, scenarios)
  - How are these components related? – Possibly re-organize the components (composition/decomposition)
- Two main levels of design:
  - Architectural (high level) design
  - Detailed design
  - Different design concerns at different abstraction levels (e.g. classes vs. modules vs. entire system)
- How should we depict design – what notation/language?

Review: High-level and Low-level Designs

Architectural design (high-level design) patterns and styles

- MVC, Layered, Pipeline, Client-Server, SOA, ...

Detailed design (low-level design)

- Functional decomposition, database design, Object-Oriented design, user-interface design, ...
- Object-Oriented Design and UML – focused on modeling
- To discuss more about Object-Oriented design

Outline

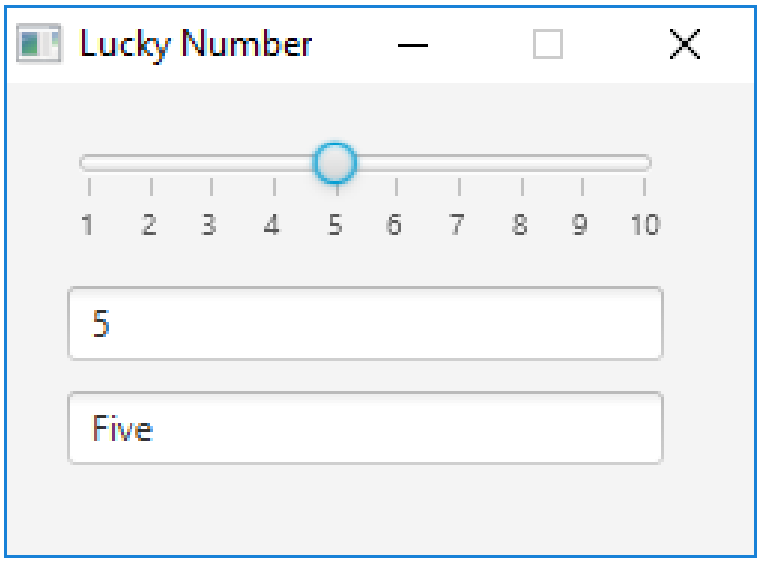
Inverse of Control

Source: Module 6 by Martin Robillard

- Inverse of control
- The observer pattern

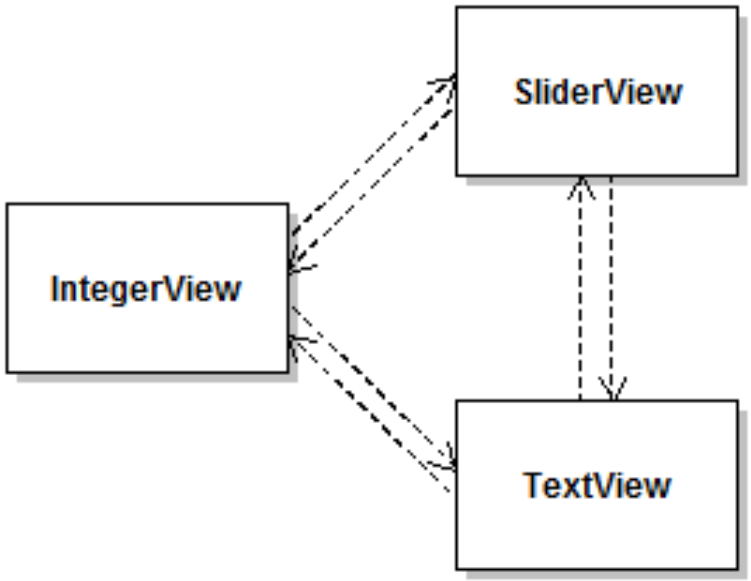
Motivation: Requirement of an Application

Consider this application which a number is selected/entered/showed in 3 different ways



Motivation: Design of the Application

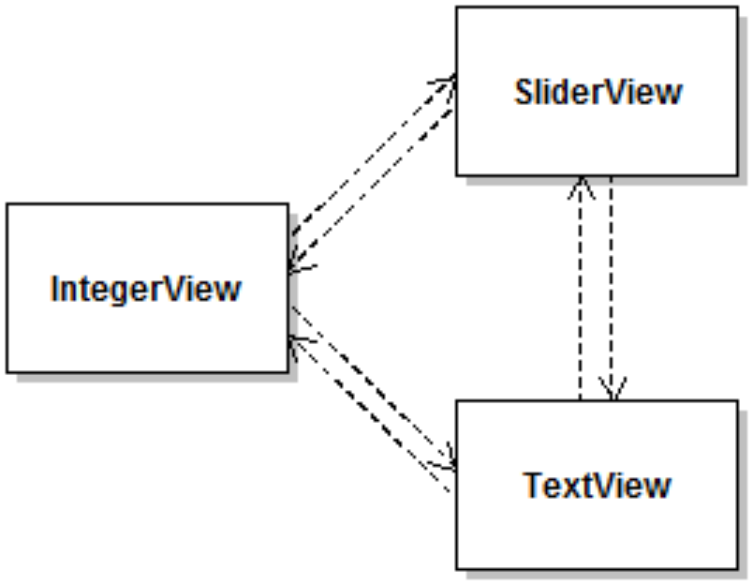
A way to implement this application has complete pairwise dependencies



Any critiques?

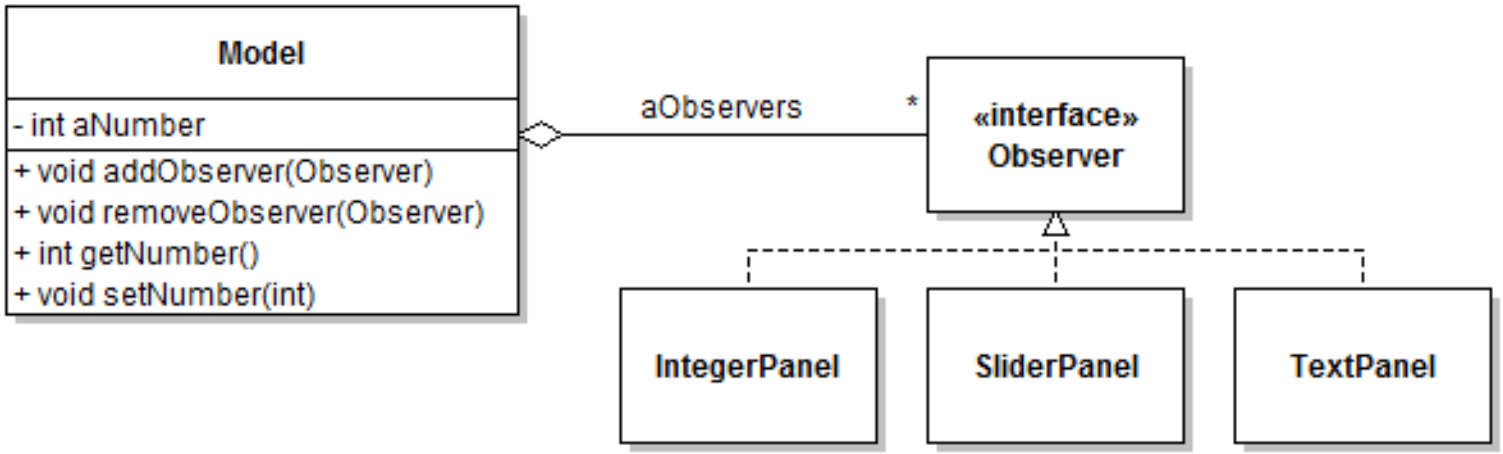
Critiques

- High coupling: Each panel explicitly depends on many other panels.
- Complexity: Complex idiosyncratic program logic is required to keep the different panels consistent.
- Low Extensibility: To add or remove a panel, it is necessary to modify all other panels.



Redesign of the Application

Let's consider this improved design using the Observer Design Pattern,

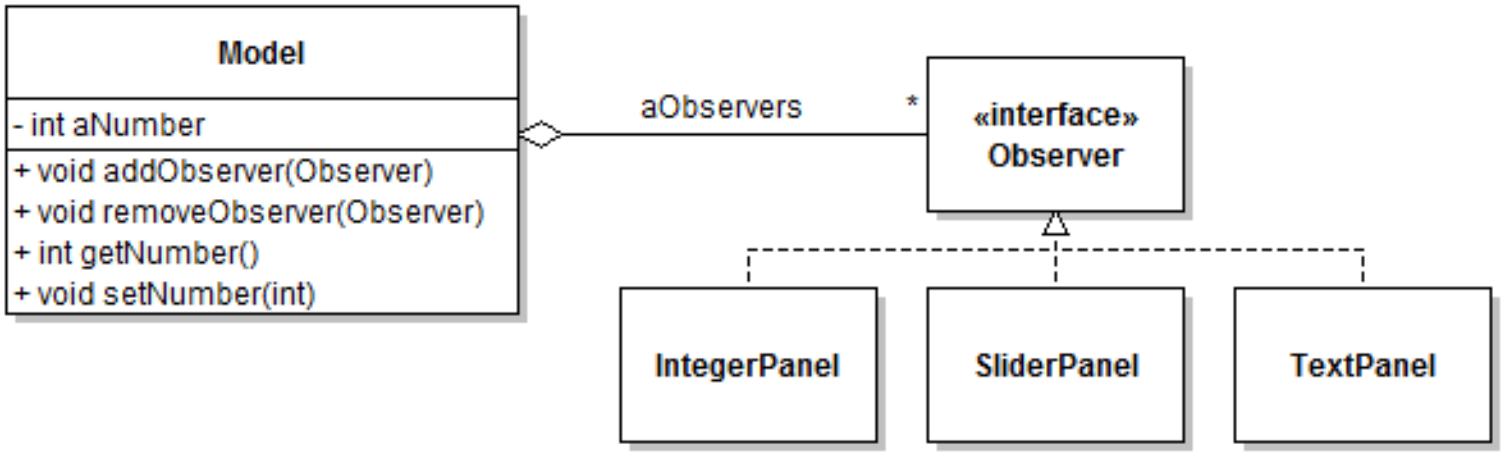


The Observer Design Pattern: store state of interest in specialized objects, and to allow other objects to observe this state.

Why is it better?

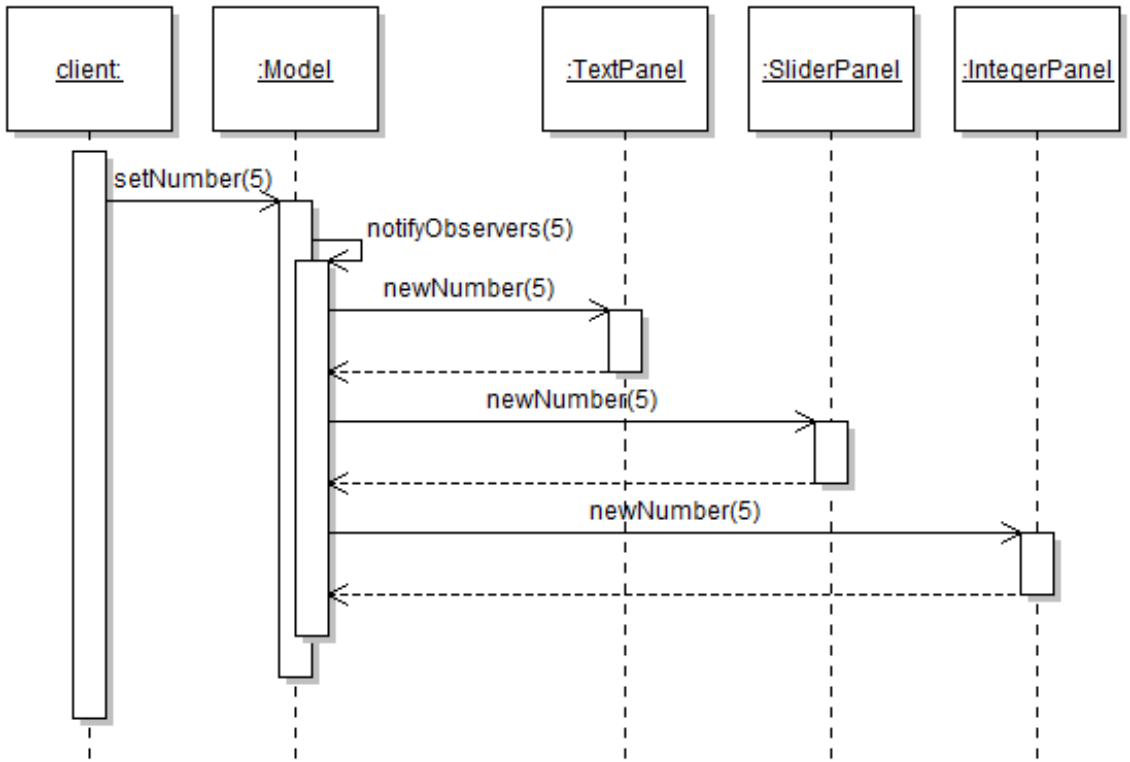
Redesign of the Application

- The model can be used without any observer;
- The model is aware that it can be observed, but its implementation does not depend on any concrete observer class.
- It is possible to register and de-register observers at run-time.



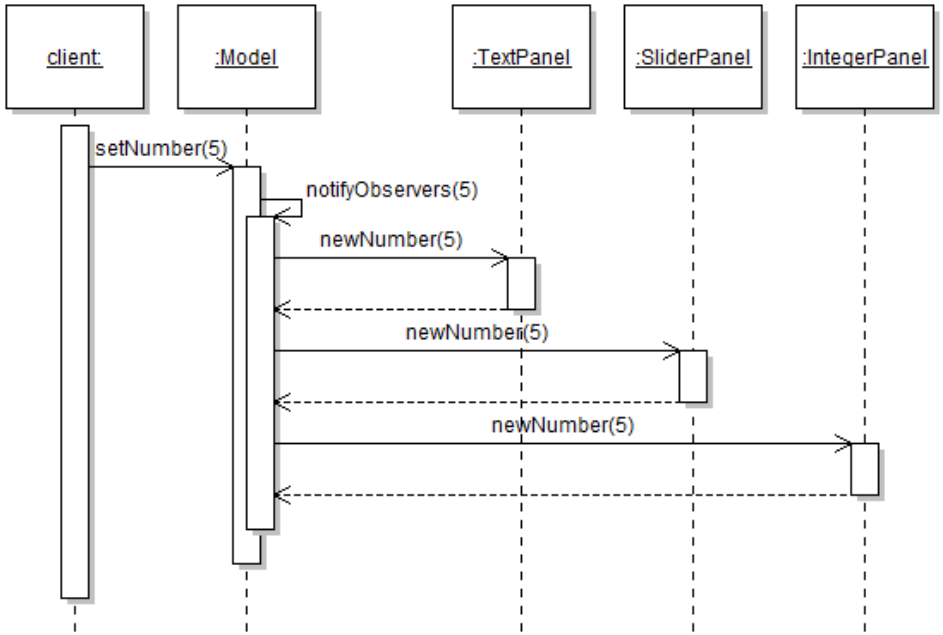
Inverse of Control and the Observer

How do the observers learn that there is new information in the model that they need to know about?



Inverse of Control and the Observer

- The model cycles through the observers and calls a "callback" method (defined on the Observer interface) on them.
- Inversion of control: the observer do not call a method on the model even though it is the observer that "makes" the change.
  - Also called the "Hollywood Principle" ("don't call us, we'll call you").



Cycling Through Callbacks

Essentially, the idea is as follows,

```
 1 class Model {
 2   List<Observer> observerList;
 3   // ...
 4   void setNumber() {
 5     for(Observer observer: observerList) {
 6       if (observer == null) continue;
 7       observer.event1();
 8       observer.event2(data);
 9       observer.event3(this);
10     }
11   }
12 }
```

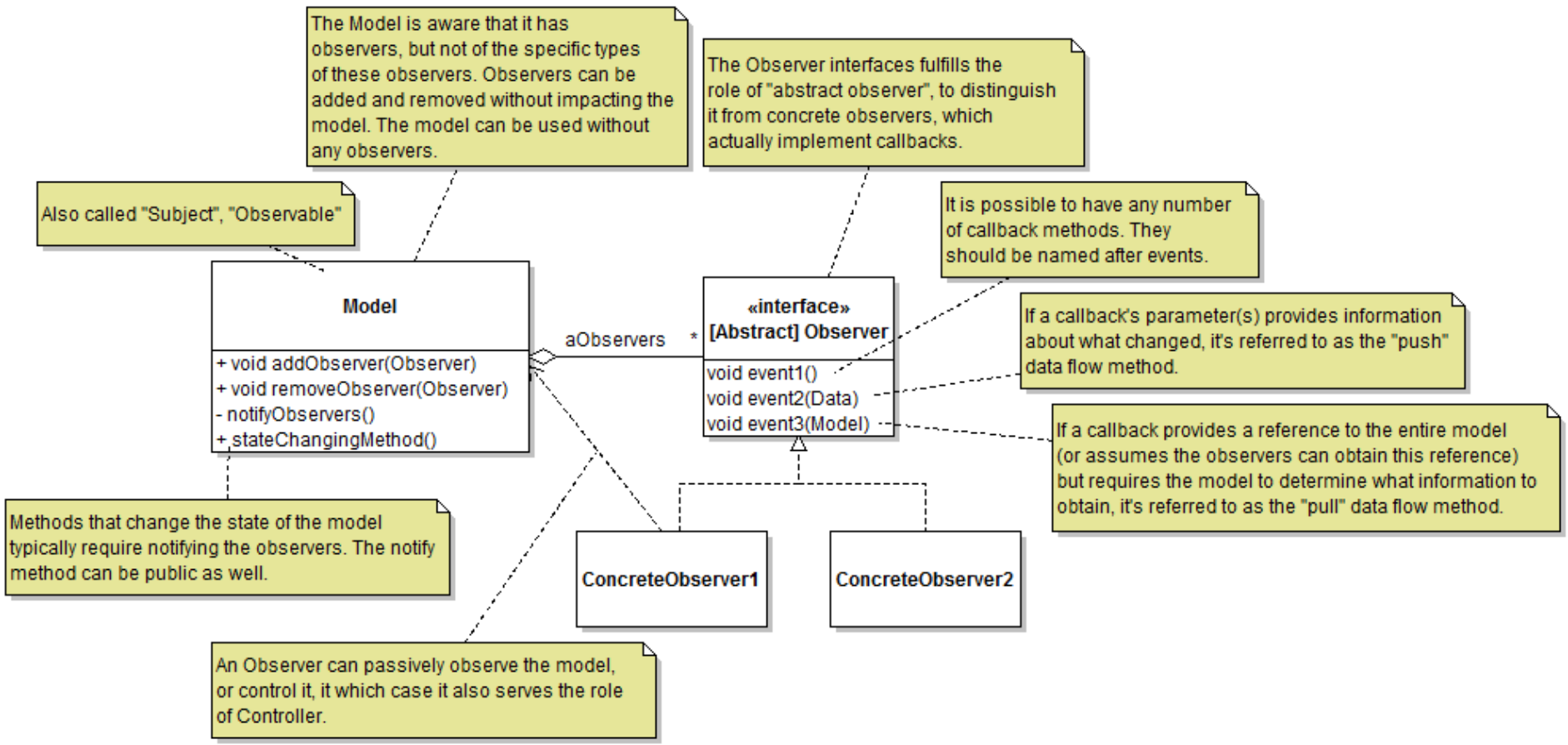
Alternative, use the event-driven framework – next slide

Cycling Through Callbacks

Alternative, use the event-driven framework,

- the model acts as the "event source".
- the model generates a series of "events" that correspond to different state changes, and
- other objects are in charge of reacting to these events – calling some methods (callbacks) by the framework.

The Observer Design Pattern: Summary



Outline

Example Application

Let's consider to develop an inventory system capable of keeping track of electronic equipment.

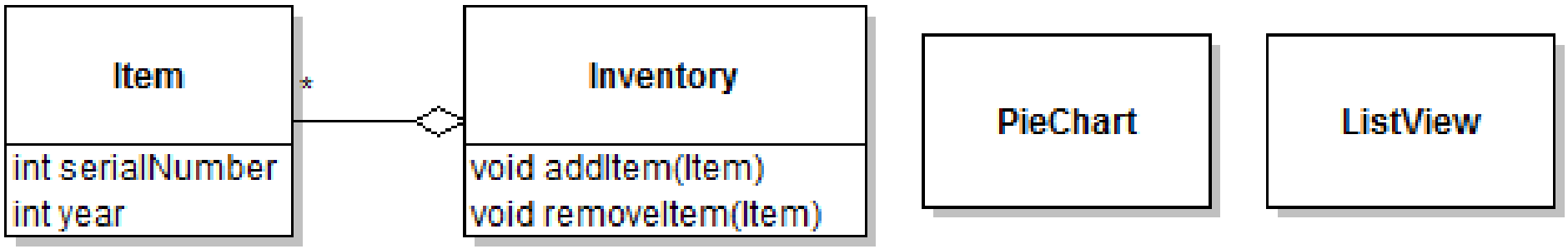
- An Item of equipment records a serial number (int) and production year (int).
- An Inventory object aggregates a bunch of Items. Various entities are interested in changes to the state of the Inventory.
- Clients can add or remove Items from the Inventory at any time.

For example,

- it should be possible to show the items in the Inventory in a ListView.
- It should also be possible to view a PieChart representing the proportion of Items in the Inventory for each production year (e.g., 2004=25%; 2005=30%, etc.).
- Views should be updated whenever items are added or removed from the Inventory.

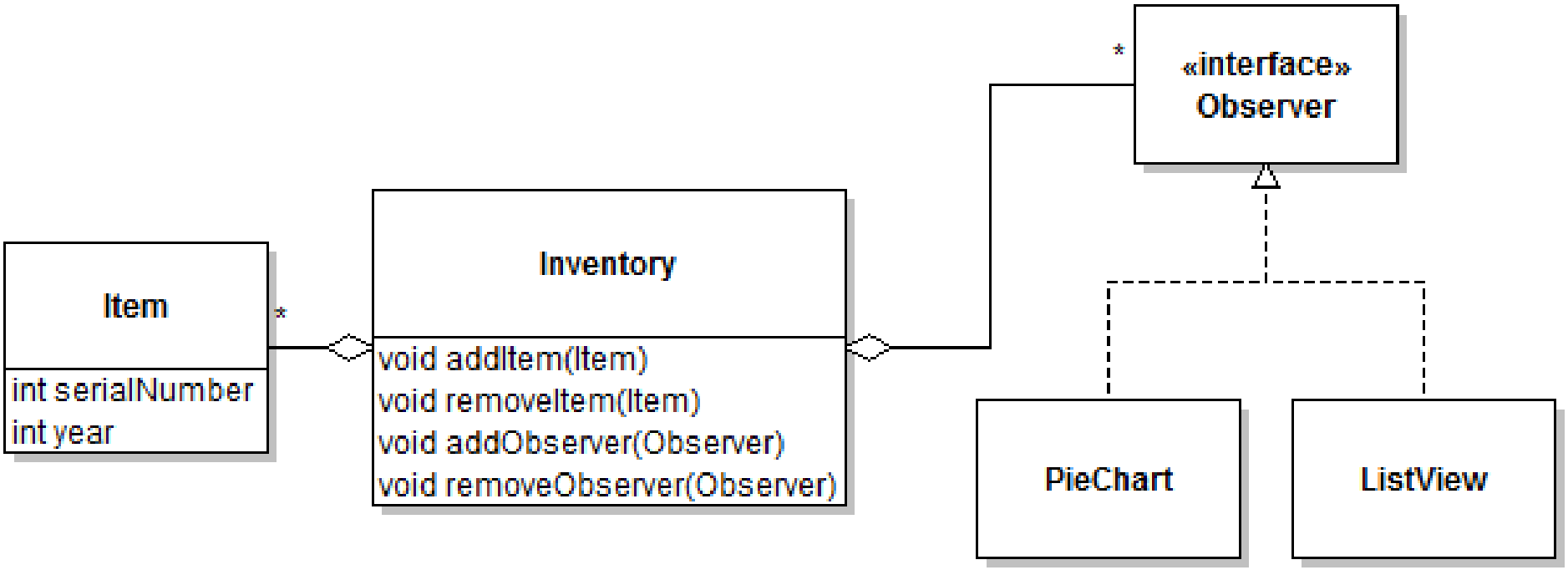
Basic Set of Classes

At a first glance, we need the following classes



Instantiating a Basic Observer Pattern

Design decision: the object containing the observable state would be instances of Inventory, and the objects observing this state would be PieChart and ListView.



Designing the Callbacks

There are only two possible events in this simple problem – adding and removing items

This leads to two design choices for the callbacks:

1. A single callback that indicates that an item was added or removed;
2. One callback for items added and one for items removed.

which one should we choose?

Designing the Callbacks

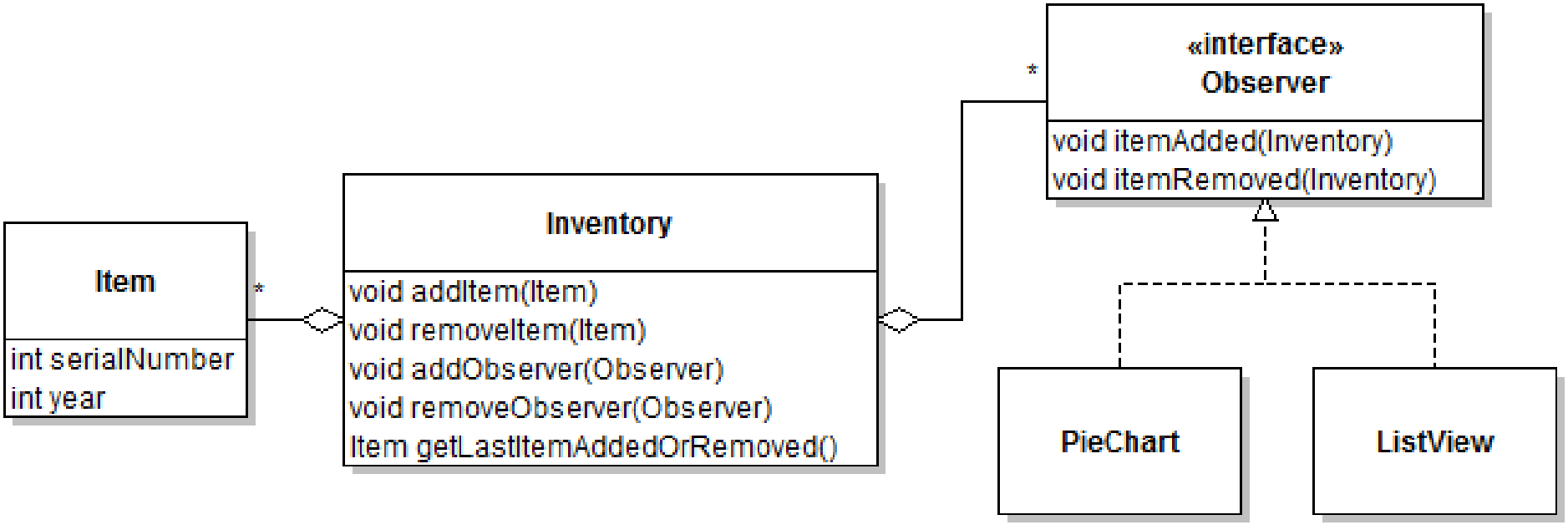
Let's evaluate these two design choices.

1. Option 1
   - It would end up being called something like `itemAddedOrRemoved`
   - The concrete observer would have to check a boolean flag to determine whether it is add or remove.
   - Therefore, it clearly has a bad smell of not being not quite right.
2. Option 2 – has none of these issues – choose this Option.

The next question is how to tell observers which item has been added or removed.

- Using the "pull" strategy, i.e.,
  - include a reference to the `Inventory` that changed as part of the callback, and
  - add a method `getLastItemAddedOrRemoved()`

Designing the Callbacks

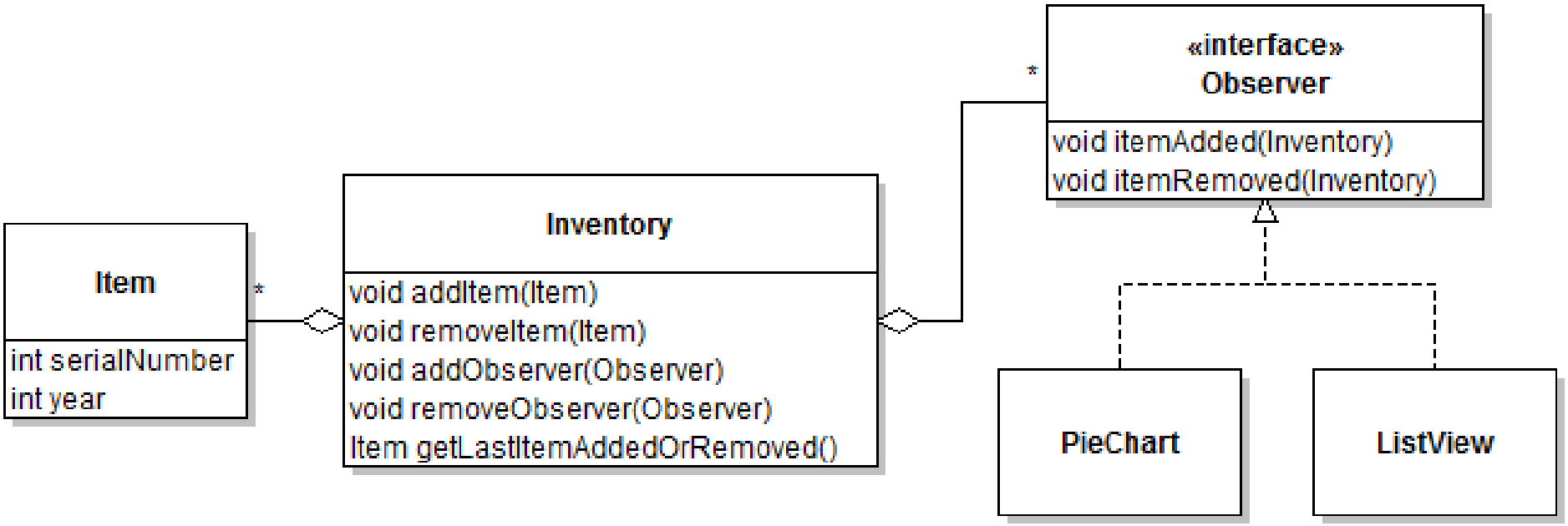


Any critiques?

Designing the Callbacks

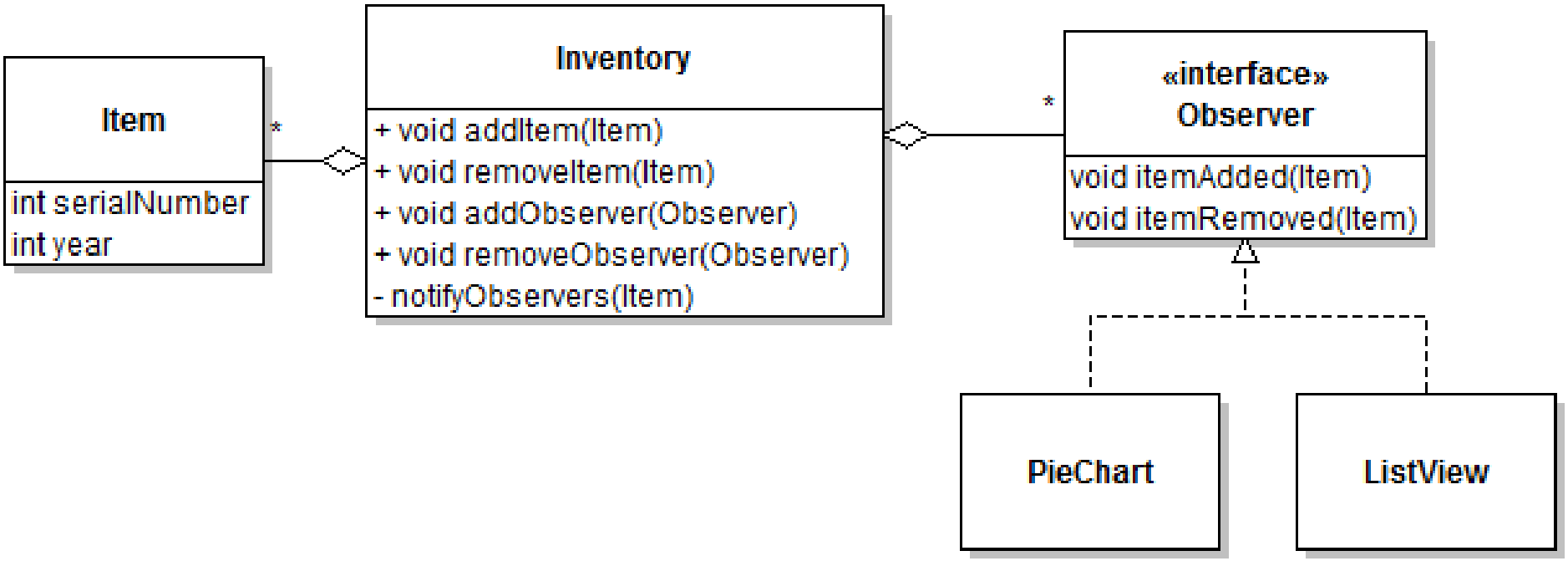
Does this look clumsy to you?

- `getLastItemAddedOrRemoved()`?
- `itemAdded(Inventory)`?
- `itemRemoved(Inventory)`?



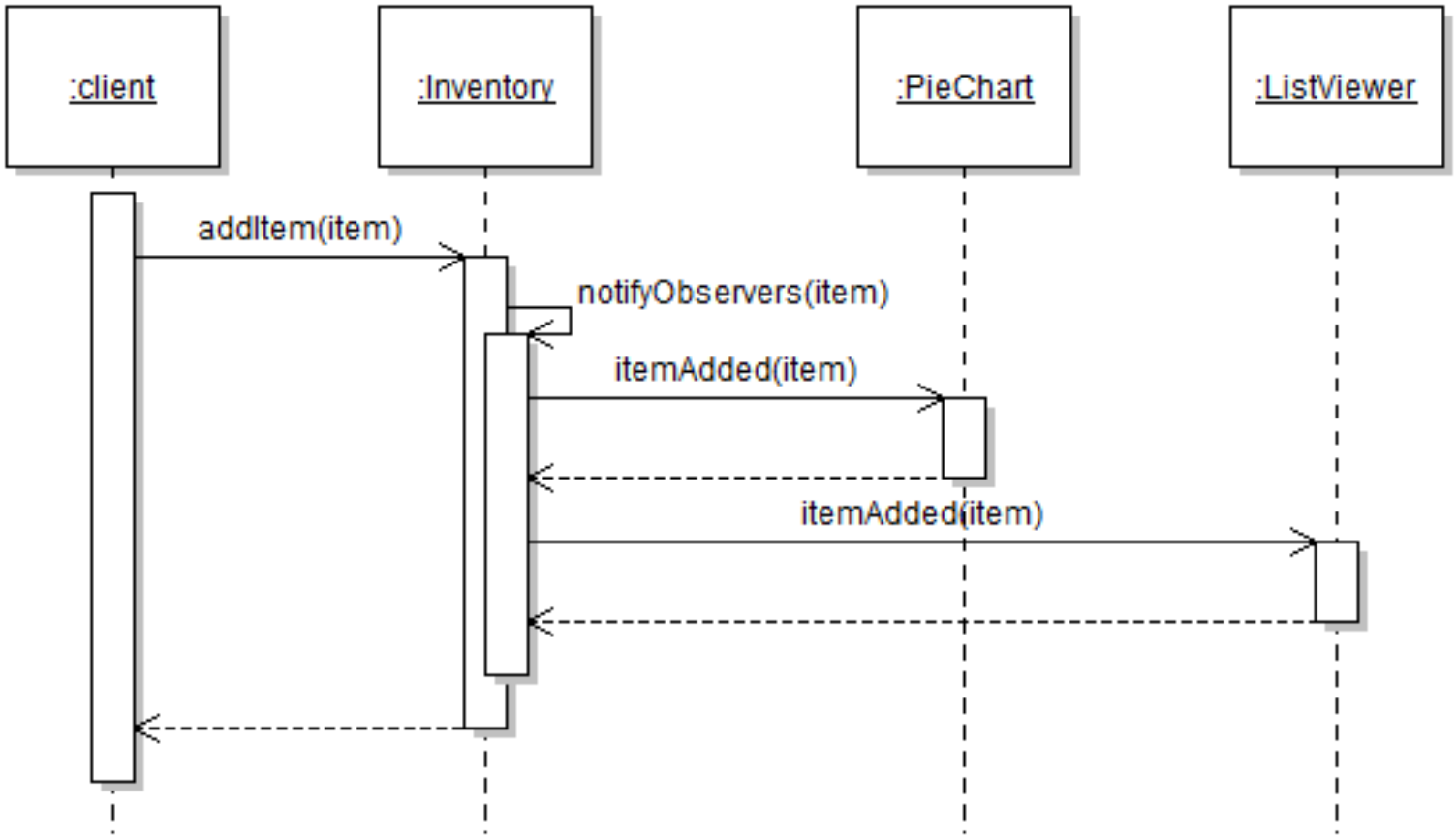
Designing the Callbacks

Get rid of `getLastItemAddedOrRemoved()` by redesigning `itemAdded()` and `itemRemoved()`



Designing the Callbacks

Assuming two observers (one of each type) are registered with the inventory, a sequence that illustrates an item being added is thus:



Summary and Questions?

- The Observer design pattern (using the Inversion of Control design strategy) is frequently used for GUI applications
- Questions?

Let's do an exercise ...

Outline

"Introduction to Software Design with Java" by Martin P. Robillard
"Engineering Software as a Service" by Armando Fox and David Patterson (2nd Edition)
"Essentials of Software Engineering" by Frank Tsui, Orlando Karam, and Barbara Bernal(4th Edition)