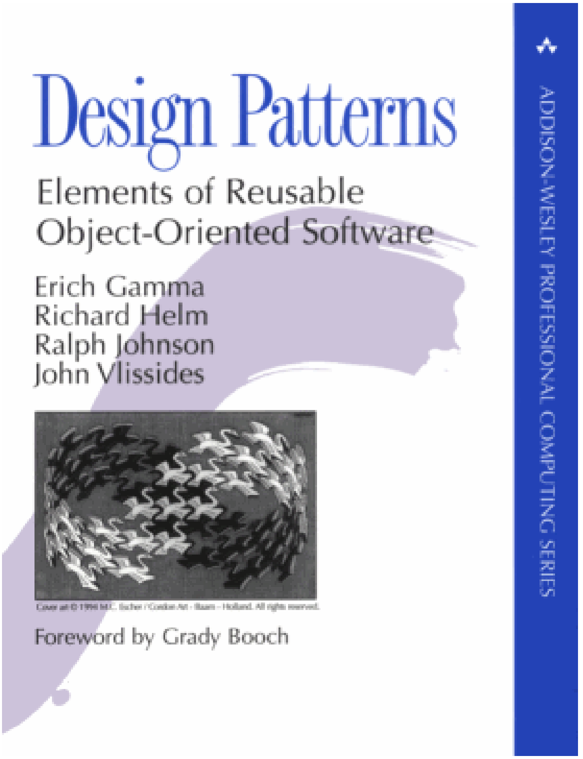
Programmeertechnieken/Programming Techniques

Part 5a
Design patterns

Koen Pelsmaekers
Campus Groep T, 2022-2023

Design patterns

The bible of Design Patterns [GoF]


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissidex, 1993.

"Think of the GOF as helping losers lose less.“ – Richard P. Gabriel

Design Pattern Concept

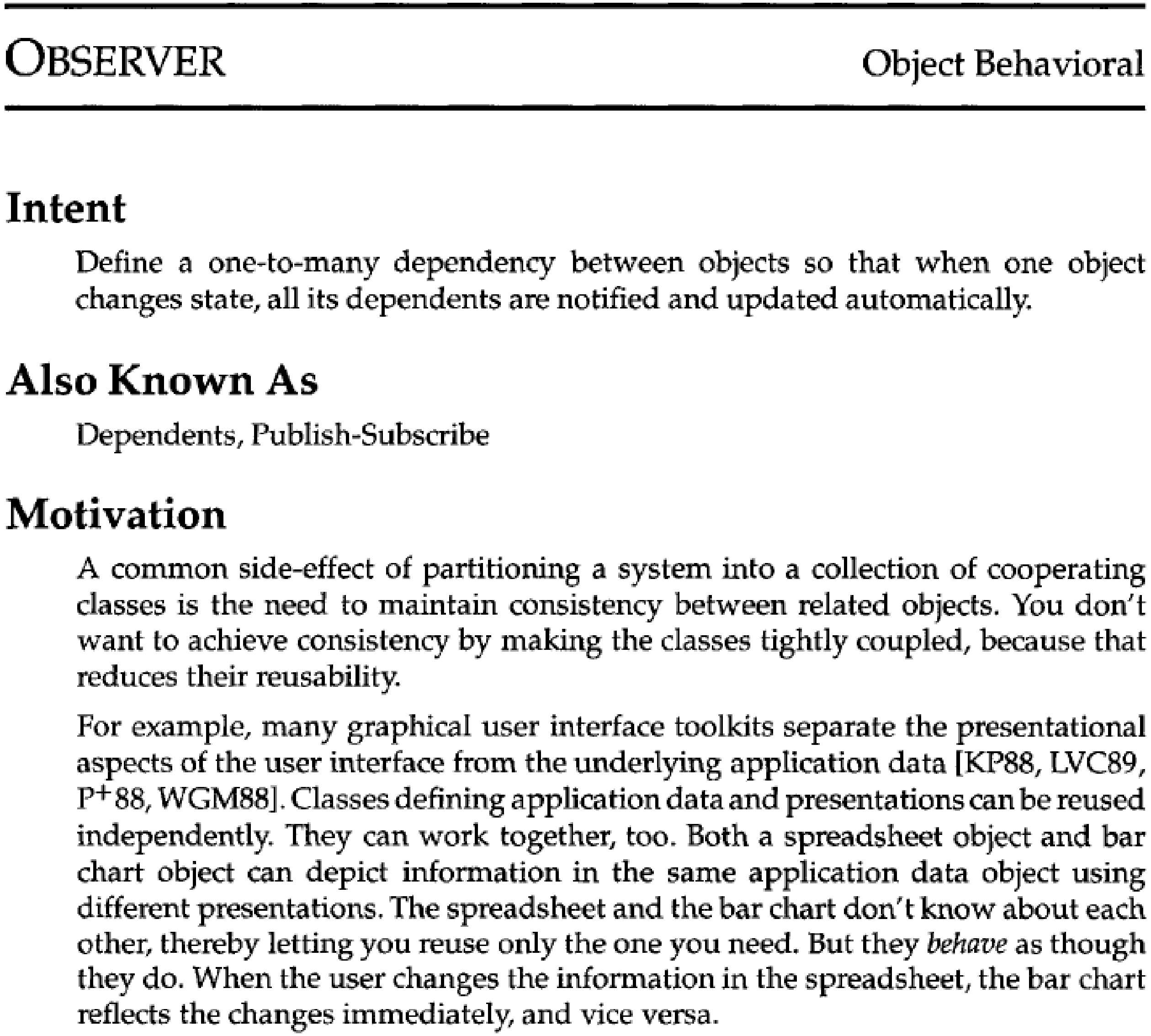
Design pattern

- Architectural patterns (Christopher Alexander)
- A general, reusable solution to a commonly occurring problem within a given context (in software design)
- Definition of relationships and interactions between classes and objects, to solve a certain kind of recurring problem
- “documented common sense”, “encapsulate what changes”, “isolate what varies”

Each pattern has

short name, brief description of the context, lengthy description of the problem, prescription for the solution

GoF



OBSERVER — Object Behavioral

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

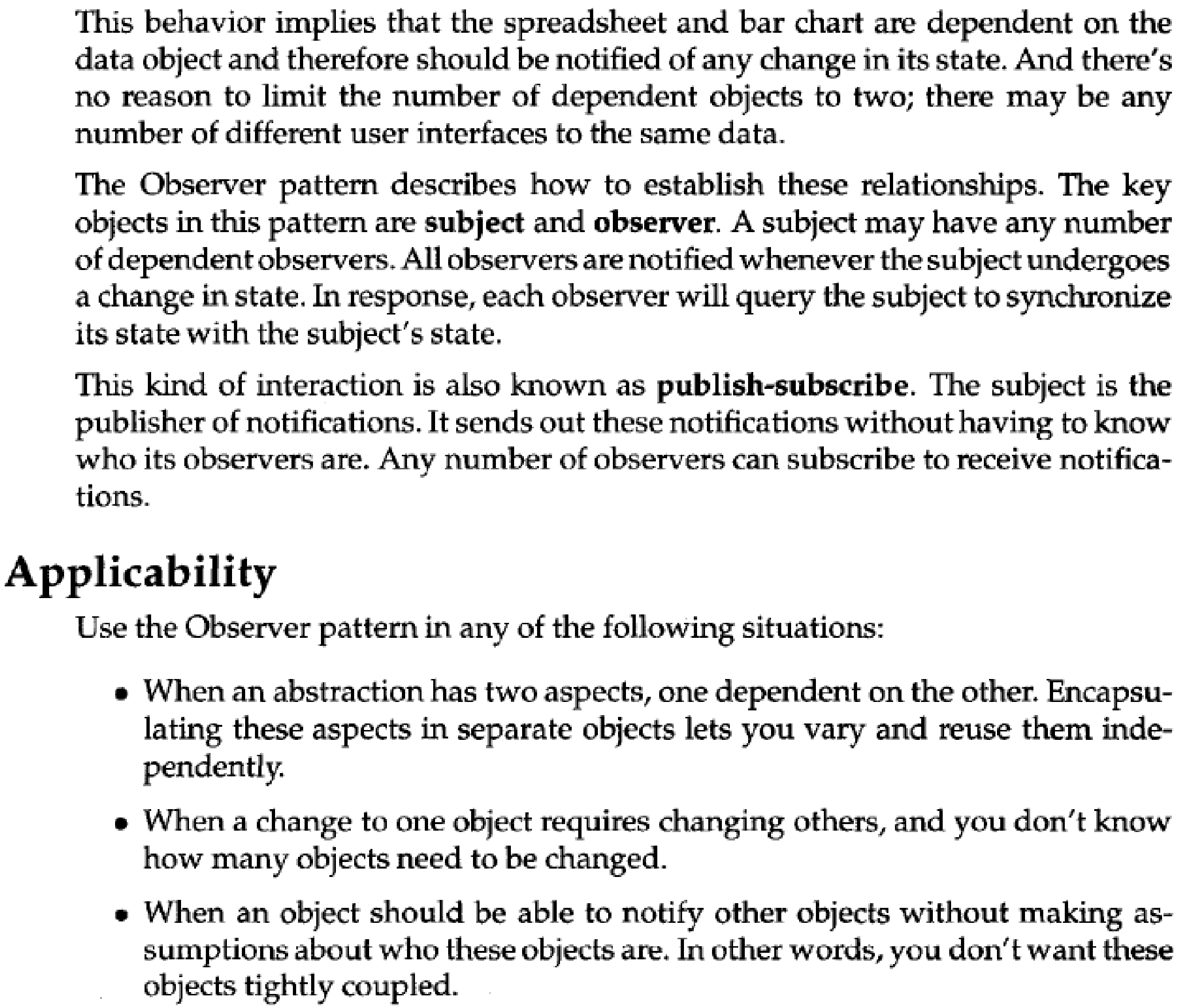
Dependents, Publish-Subscribe

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 classes tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data [KP88, LVC89, P^{+}88, WGM88]. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.

GoF



This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.

The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

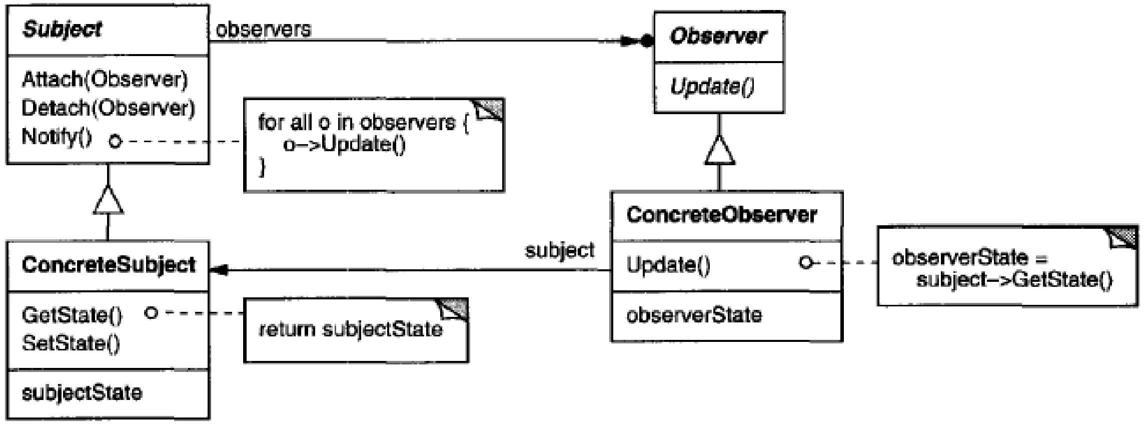
Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

GoF

Structure



```
class Subject;

class Observer {
public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

```
class Subject {
public:
    virtual ~Subject();

    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {
    _observers->Append(o);
}

void Subject::Detach (Observer* o) {
    _observers->Remove(o);
}

void Subject::Notify () {
    ListIterator<Observer*> i(_observers);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
    }
}
```

Design Pattern Categories

Based on purpose

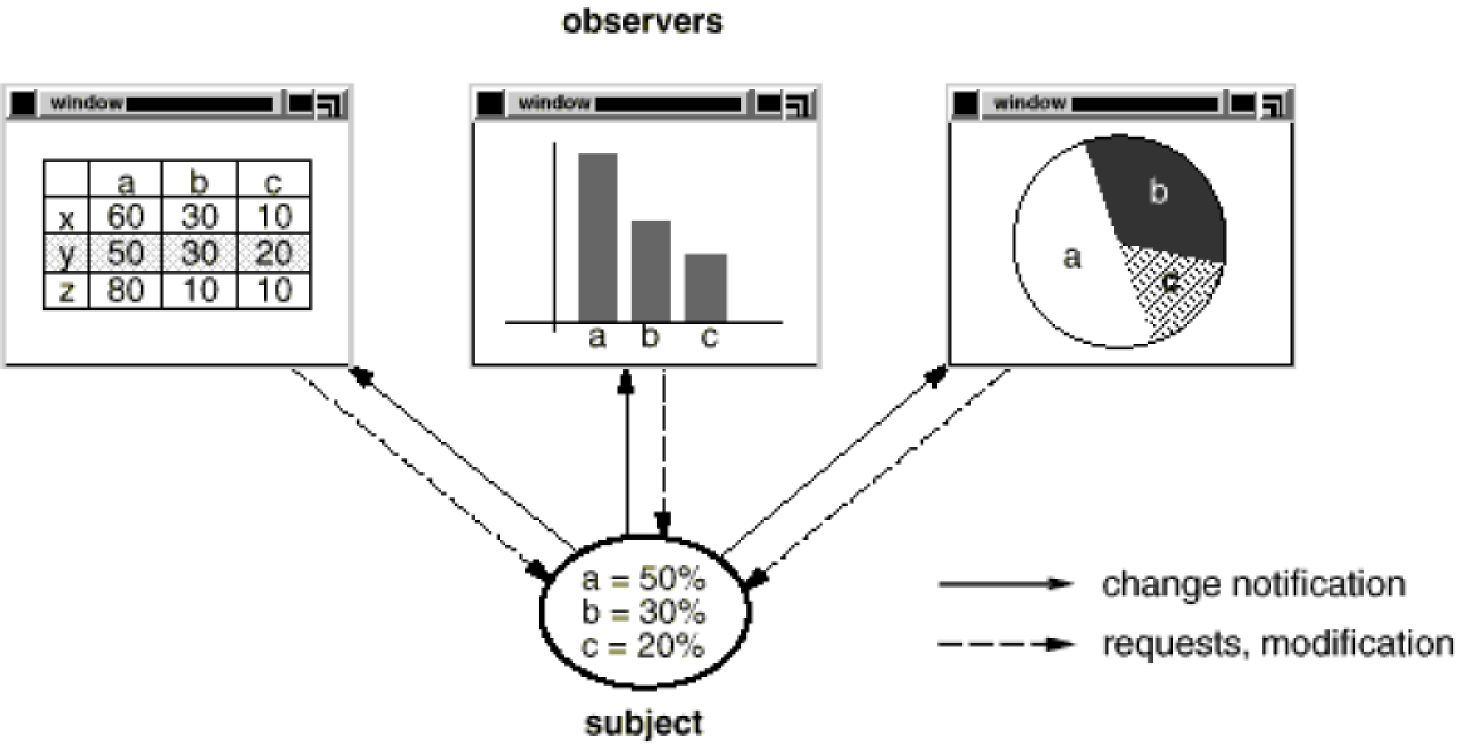
- Creational
  - defer object creation to some other class/object
  - examples: Singleton, Abstract Factory, Factory Method
- Structural
  - composing classes/objects
  - examples: Adapter, Composite, Decorator
- Behavioral
  - algotihms, flow of control, objects working together
  - examples: Observer, Strategy, Template Method

Implementation

- Interface
  - examples: Composite, Decorator, Observer, Strategy, …
- Inheritance
  - examples: Template method, …

Observer pattern

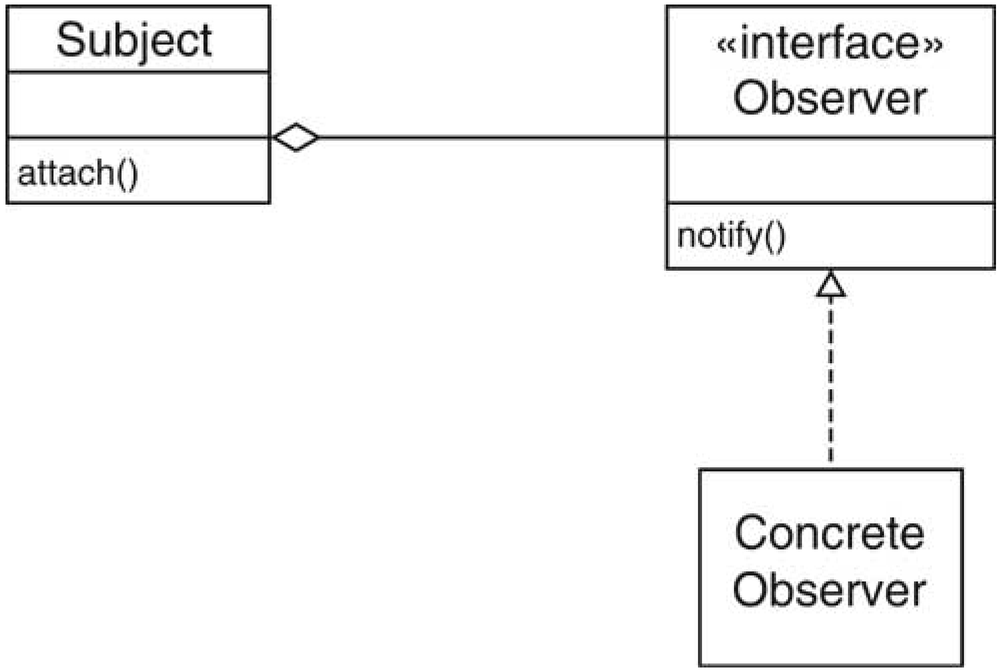
Model/View/Controller



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Observer pattern

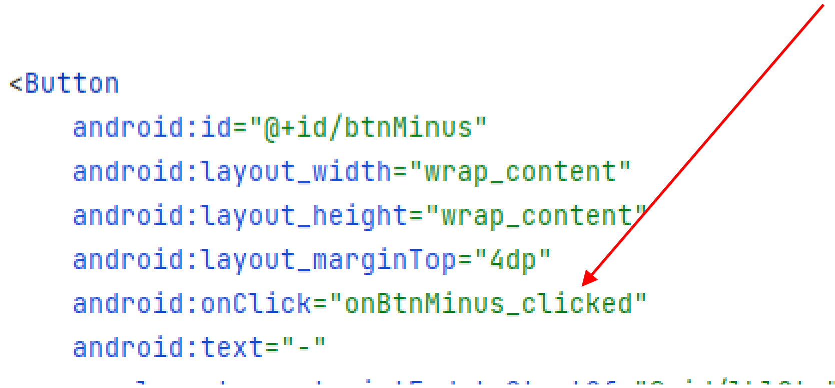
- Subject = source of events
- Observer = consumer of events
  - uses “callback” method(s)



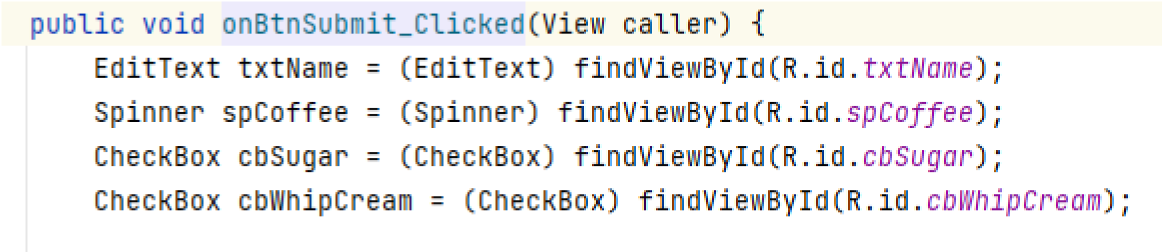
Observer pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Applicability
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don’t know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

Observer in practice



```
<Button
    android:id="@+id/btnMinus"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dp"
    android:onClick="onBtnMinus_clicked"
    android:text="-"
```



```
public void onBtnSubmit_Clicked(View caller) {
    EditText txtName = (EditText) findViewById(R.id.txtName);
    Spinner spCoffee = (Spinner) findViewById(R.id.spCoffee);
    CheckBox cbSugar = (CheckBox) findViewById(R.id.cbSugar);
    CheckBox cbWhipCream = (CheckBox) findViewById(R.id.cbWhipCream);
```

- Demo
- Examples in Java, Android API
  - ActionListener, ItemChangeListener
- Uses “callback mechanism”
- Can (sometimes) be implemented as a lambda expression (@FunctionalInterface)

Singleton Pattern

Singleton Pattern

Only 1 can exist

- Only one instance of a class exists in the JVM
- Use cases: logging, drivers, caching, pools, …
- Java API examples
  - Desktop (java.awt), Runtime (java.lang), …
- Implementation
  - private constructor, private static field + public static getter

or

  - enum (implicit or explicit private or package-private constructor) with one constant

Marker

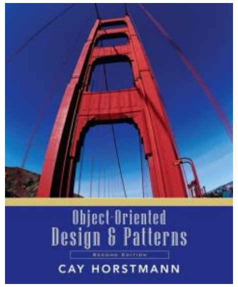
Marker (interface)

- To “mark” a class
- Implementations
    - Interface
        - Empty interface
        - Check with “instanceof” or Class class
    - Marker annotation
        - Annotation without elements
        - Special kind of interface
        - Check with Class class
- Examples: Cloneable, Serializable, Shippable (demo)

Other design patterns: examples

Design patterns (and refactoring) example

- Based on: Object-Oriented Design & Patterns, Chapter 5, Cay Horstmann
- Invoice
  - Product (description & price)
  - Bundle of products (Composite design pattern)
  - Discounted products (Decorator design pattern)
  - Format invoice (Strategy design pattern)
  - Update invoice (Observer design pattern)



decimal dot

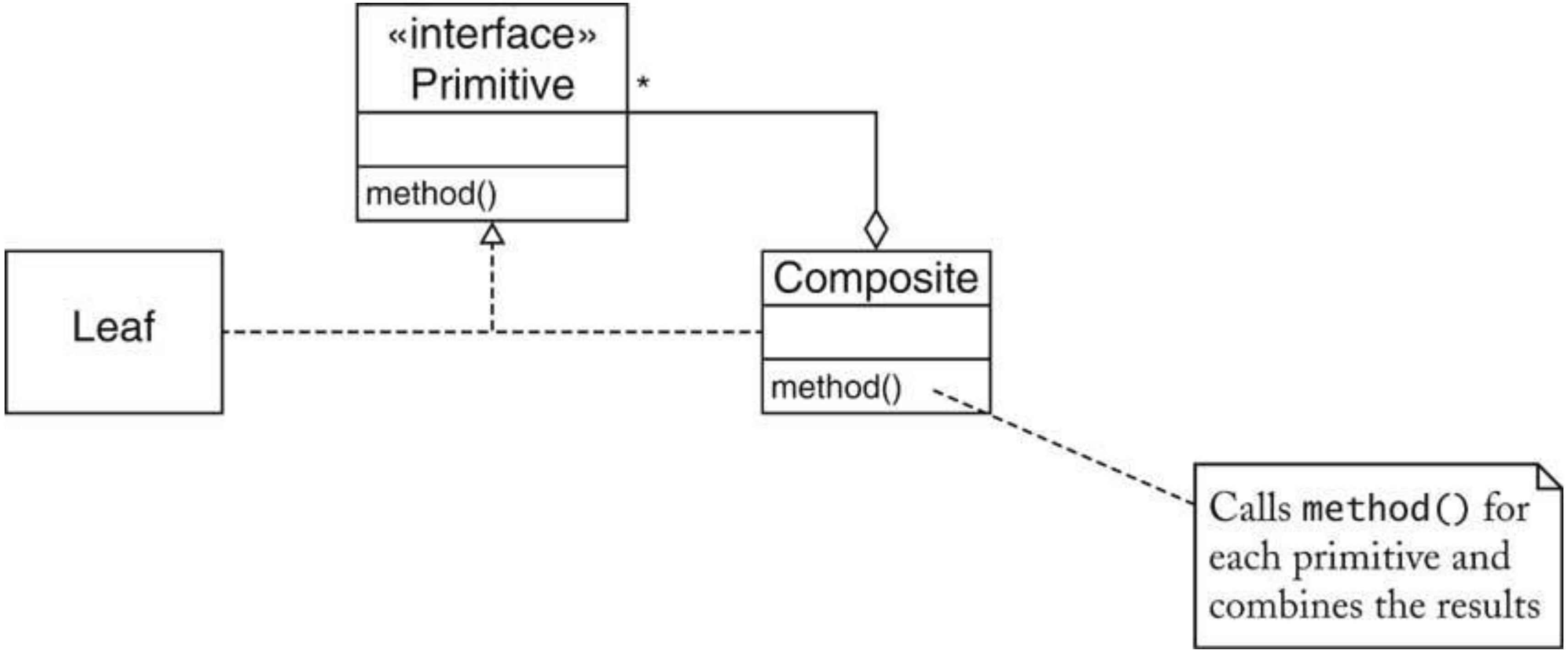
%.2f

https://horstmann.com/design_and_patterns.html

Composite pattern

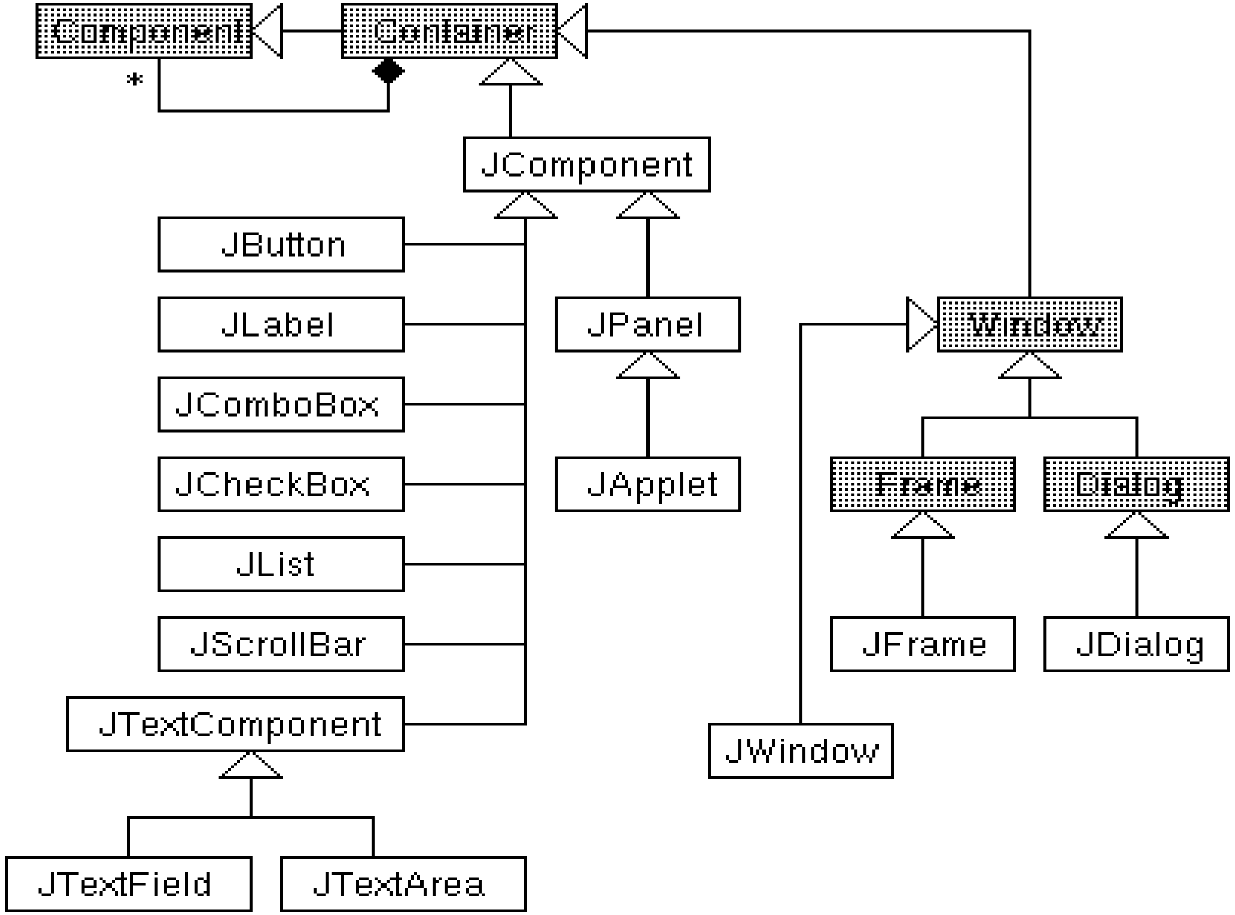
refactoring : do not change function but structure or readability

- Intent
  - Primitive objects can be combined to composite objects
  - Clients treat a composite object as a primitive object



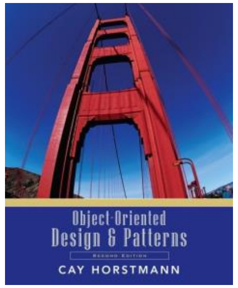
Calls method() for each primitive and combines the results

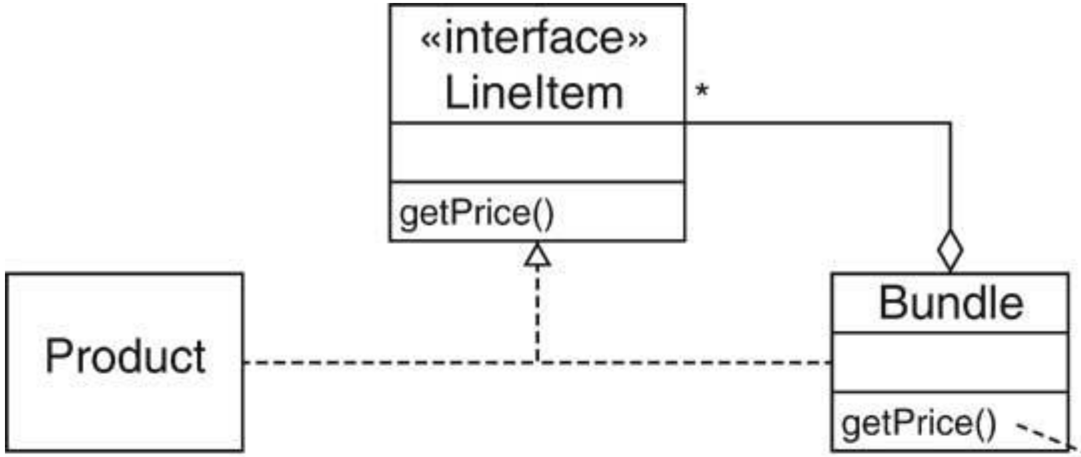
Composite pattern: example



Bundle (Composite pattern)

price sum

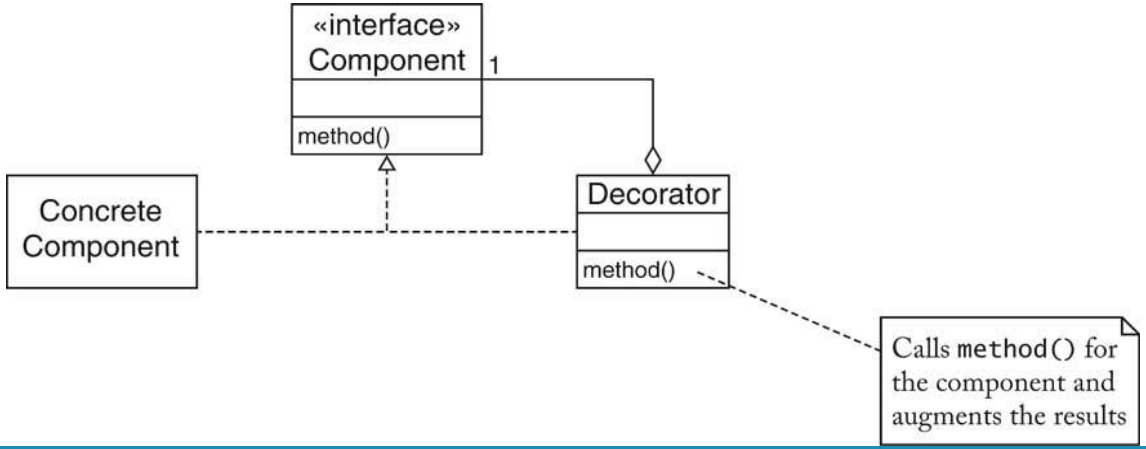




Calls getPrice() for each line item and adds the results

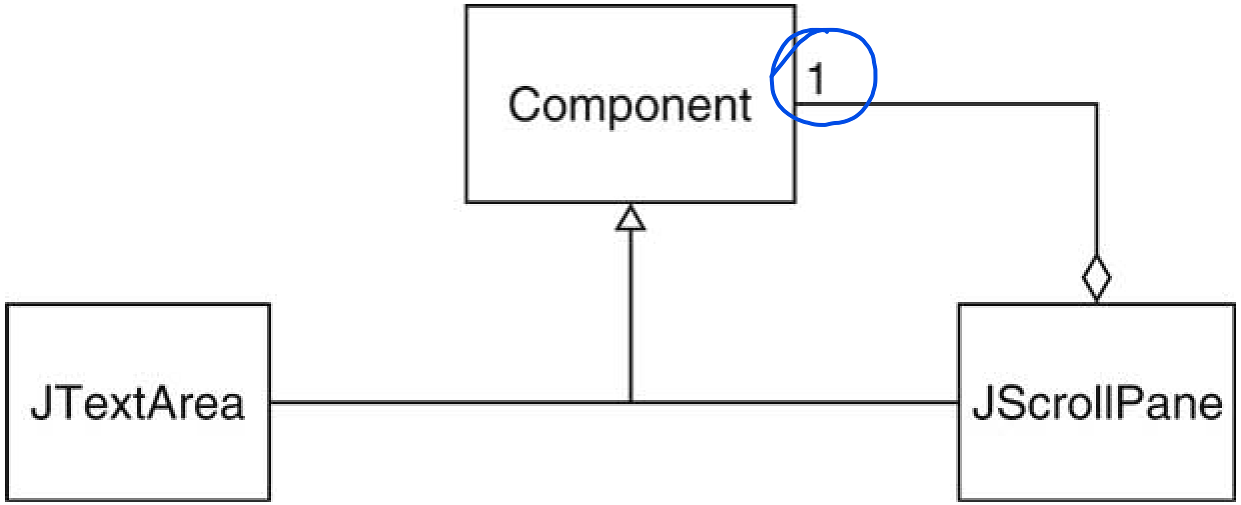
Decorator pattern

- Intent
  - Component objects can be decorated (visually or behaviorally enhanced)
  - The decorated object can be used in the same way as the undecorated object
  - The component class does not want to take on the responsibility of the decoration
  - There may be an open-ended set of possible decorations

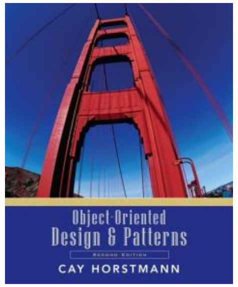


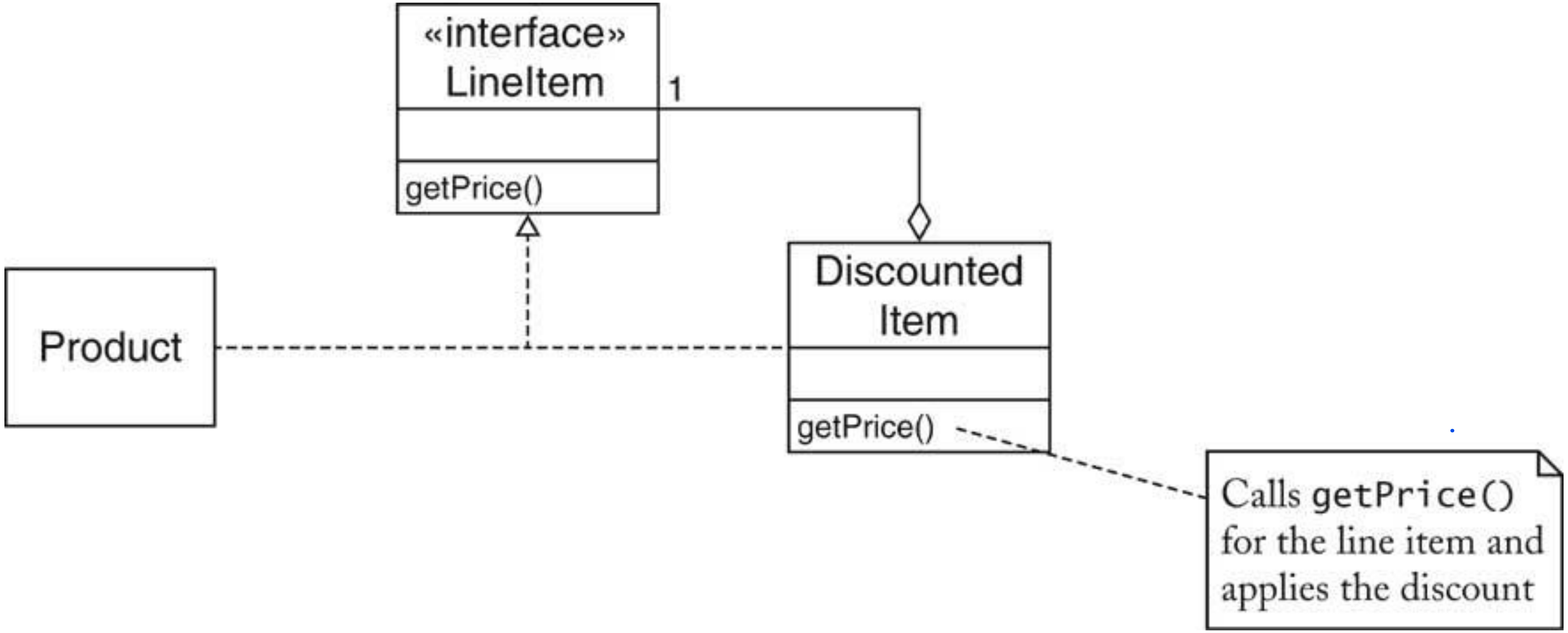
Decorator pattern: example

Very extendable



Discounted Item (Decorator pattern)

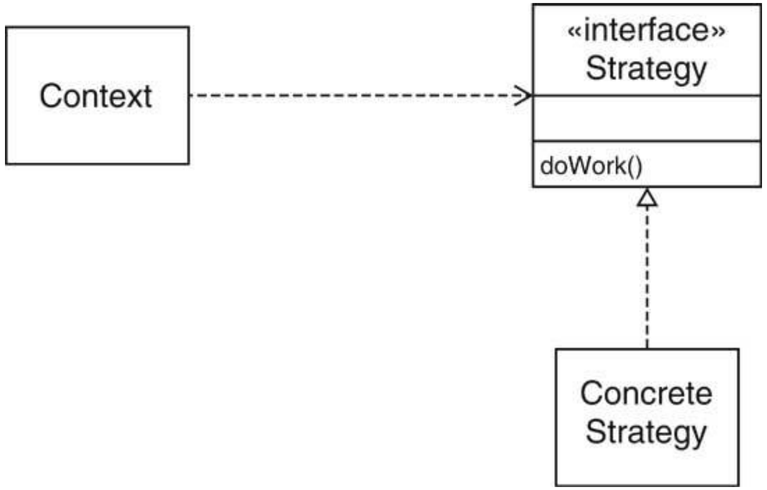




Strategy pattern

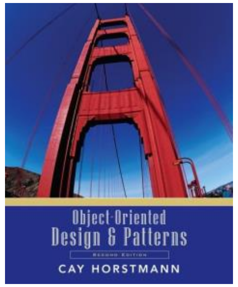
The design pattern is a common sense solution

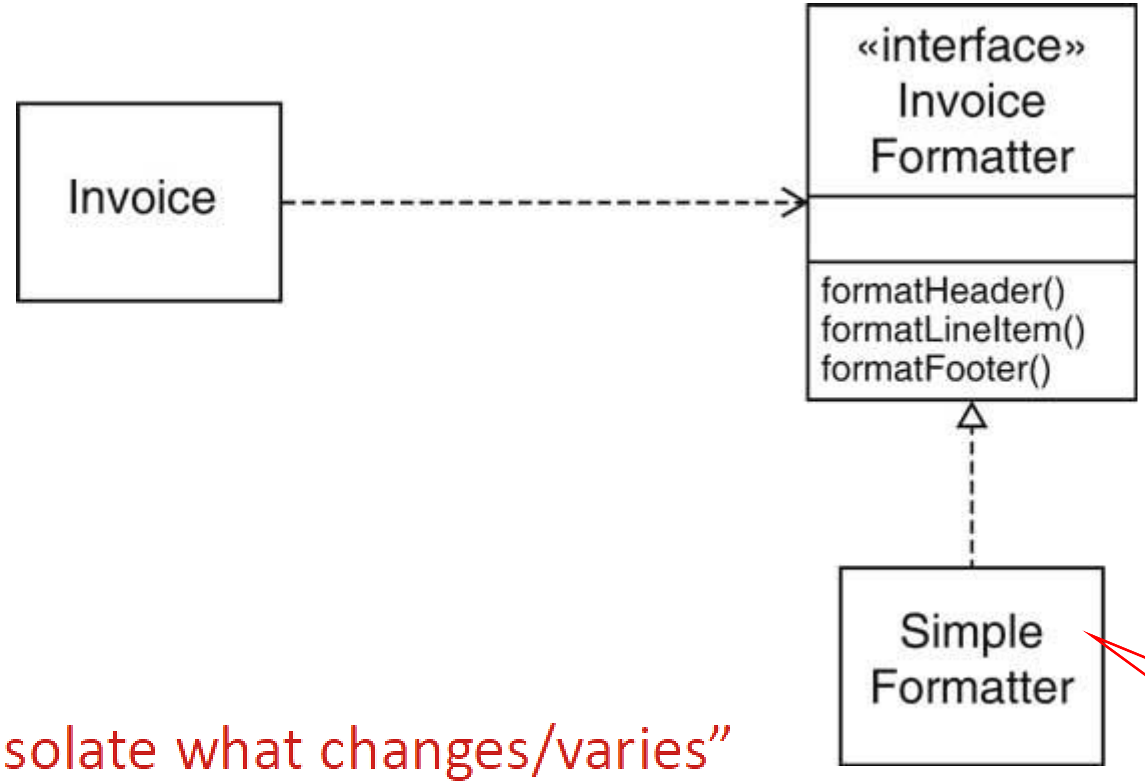
- Intent
  - A class can benefit from different variants for an algorithm
  - Clients sometimes want to replace standard algorithms with custom versions



Invoice formatting (Strategy pattern)

seperate the things that doesn't change



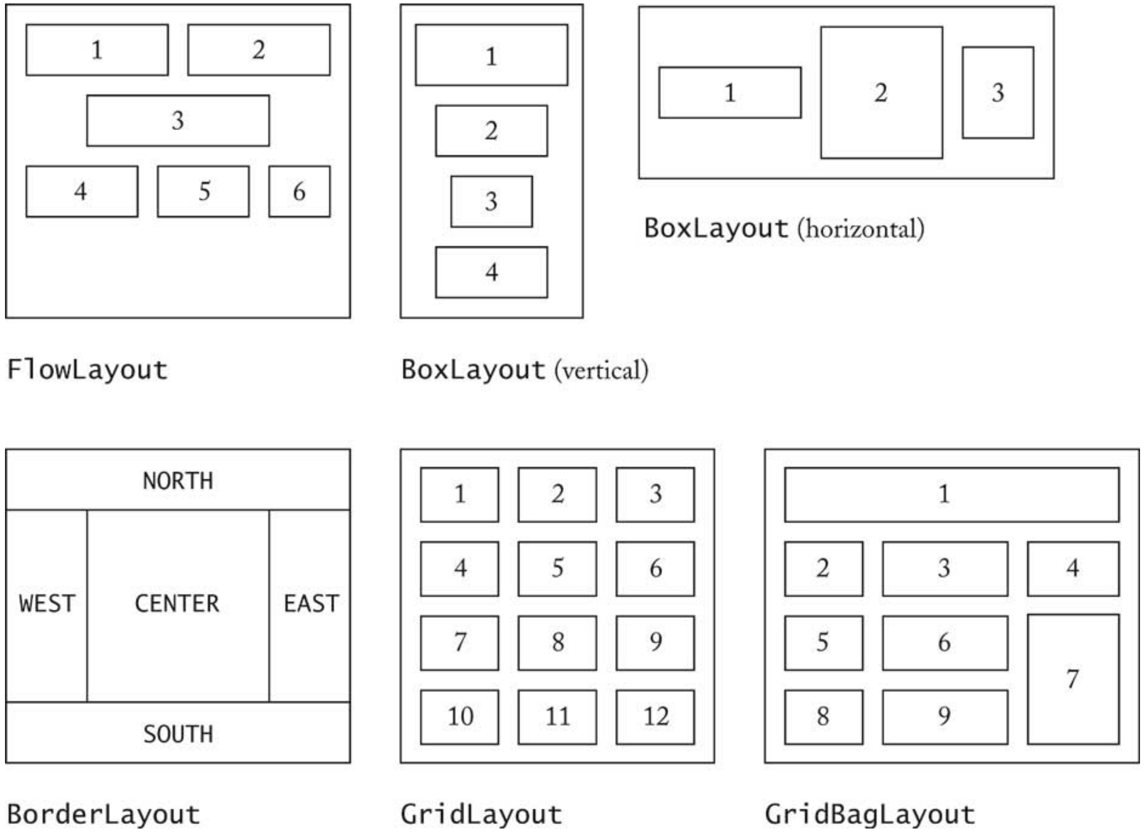


Can be replaced by other formatters

“Encapsulate/isolate what changes/varies”

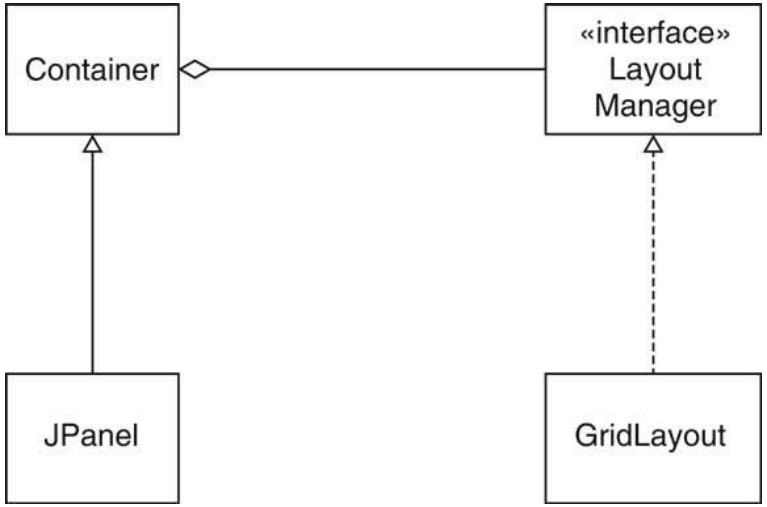
(what stays the same is isolated from what changes often)

Strategy Design Pattern: Java API LayoutManager



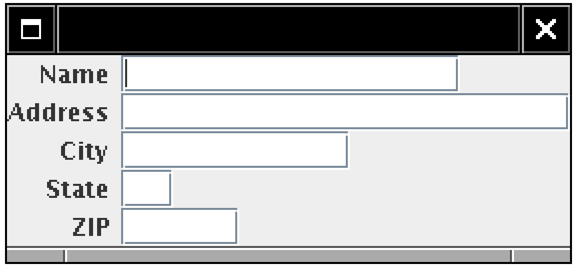
`FlowLayout` `BoxLayout` (vertical) `BoxLayout` (horizontal)

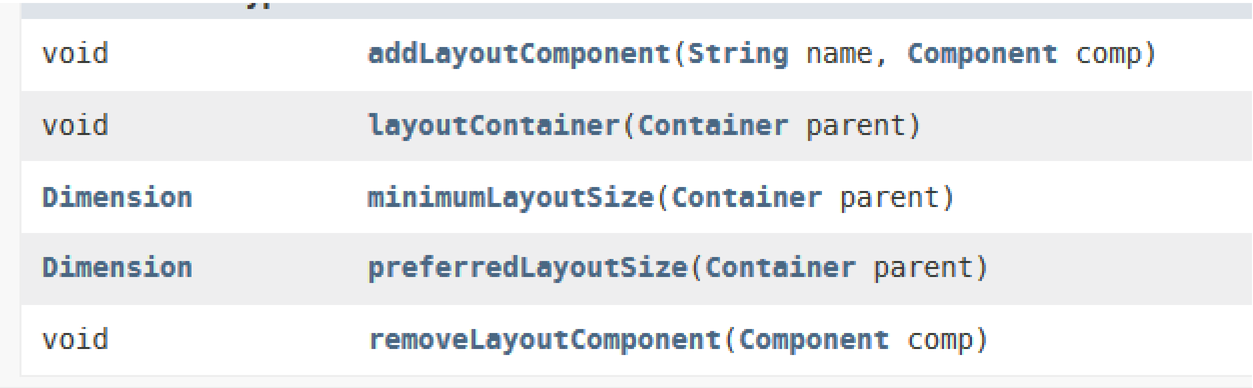
`BorderLayout` `GridLayout` `GridBagLayout`



FormLayout: new LayoutManager?

- odd-numbered components right aligned (= label)
- even-numbered components left aligned (= field)
- Implement LayoutManager Interface

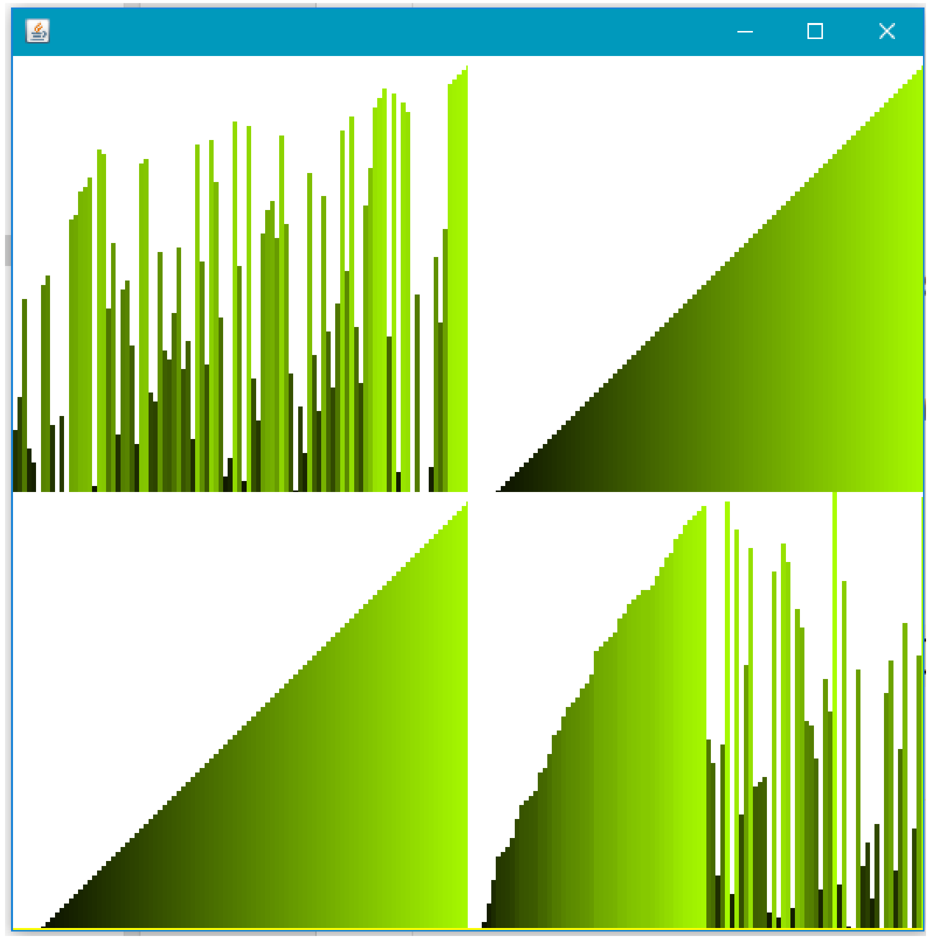




void	addLayoutComponent(String name, Component comp)
void	layoutContainer(Container parent)
Dimension	minimumLayoutSize(Container parent)
Dimension	preferredLayoutSize(Container parent)
void	removeLayoutComponent(Component comp)

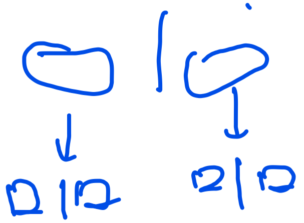
Strategy design pattern: sorting algorithms demo

- to be refactored later



quick sort

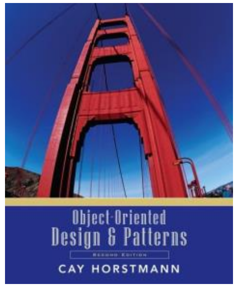
find a middle value

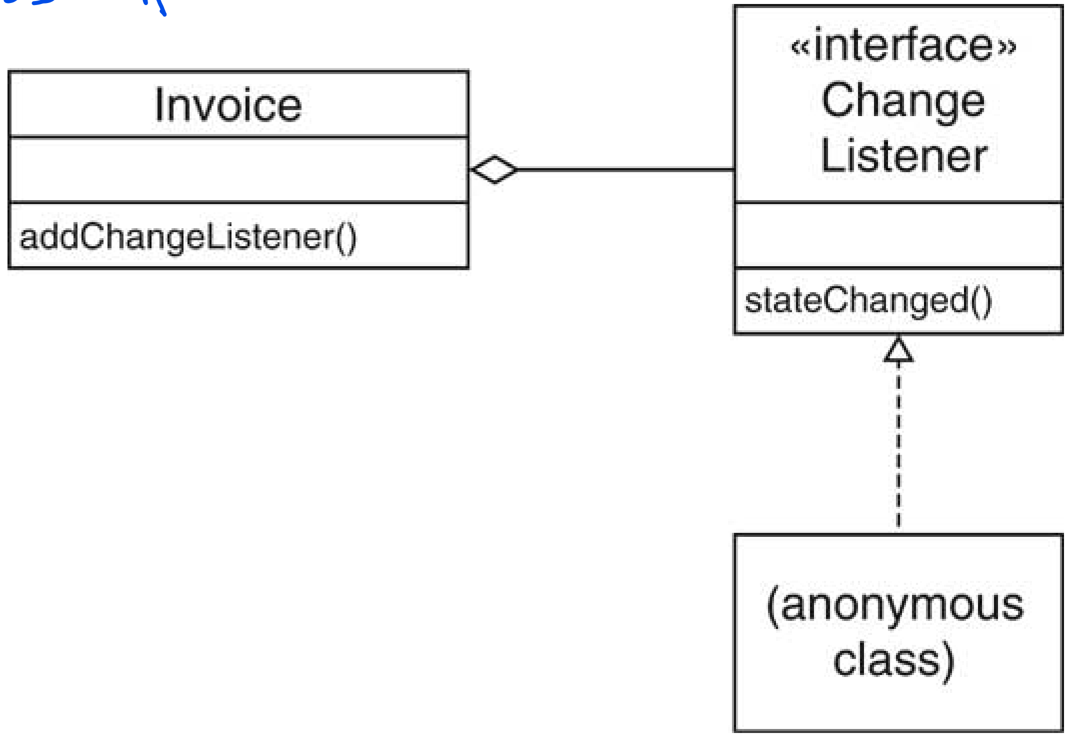


Invoice updates (Observer pattern)

string.format("in %.2f", aDouble)

%s $:




Invoice
addChangeListener()
«interface»
Change
Listener
stateChanged()
(anonymous
class)