Design Patterns

Objectives

- Define Design
- Define the architectural, mechanistic, and detailed phases of design.
- List the aspects of a design pattern.
- List existing design patterns.
- Explain the Observer pattern.
- Draw the observer pattern structure.
- Implement an observer pattern in Java.
- Justify the need for using an observer pattern in implementing Java programs.
- Explain the concept of a container pattern.
- Construct software in Java using an implementation of a container pattern.

Exams Graded

What is design?

a noun ....

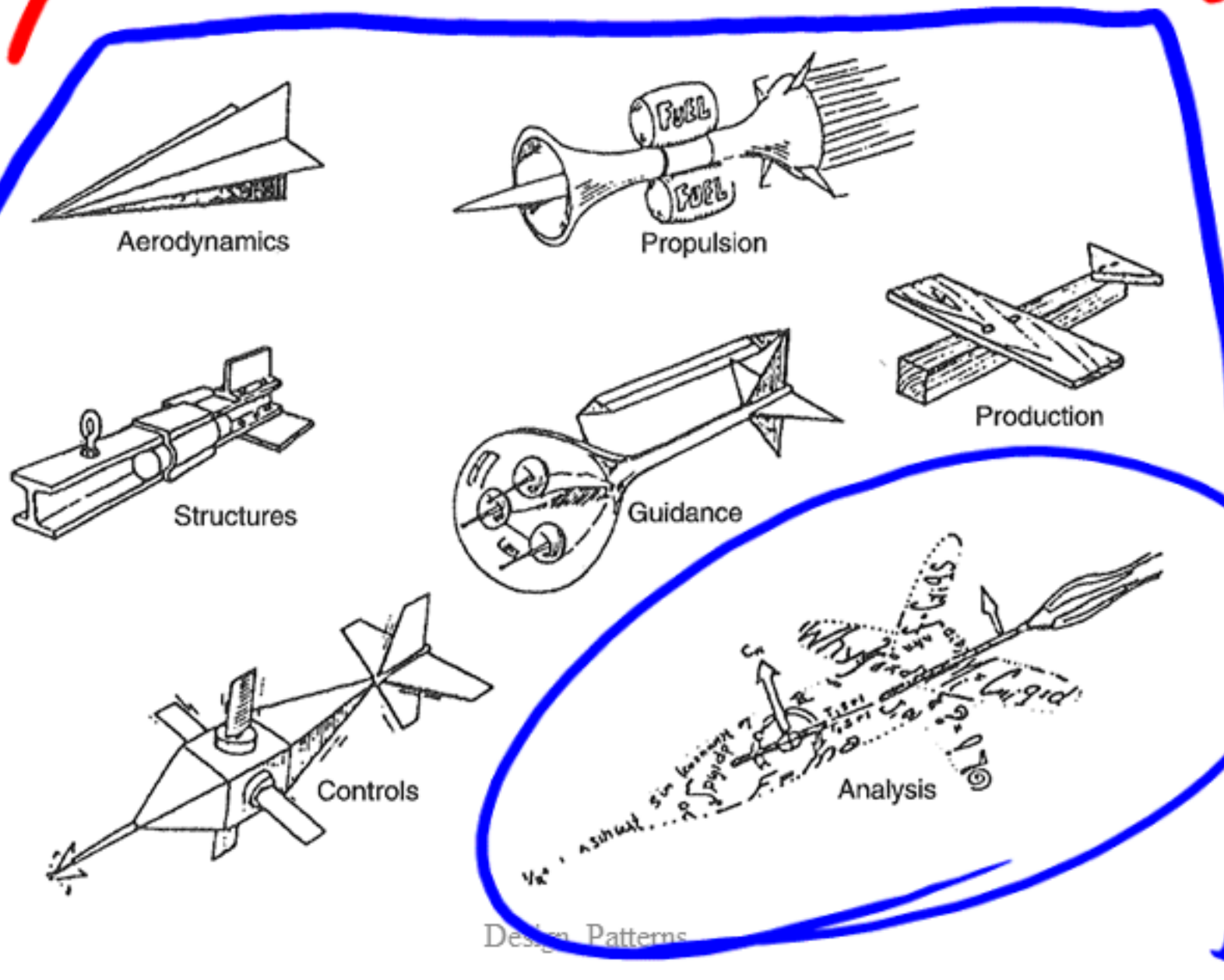
a verb ....

Something in SW engineering..

Always to go - ..

- The process of specifying a specific solution that is consistent with the analysis model
  - Optimization of model

Design



Design Phases

- Architectural Design
  - Details the largest software structures
    - Subsystems, packages, tasks
- Mechanistic Design
  - Involves the design of the interactions between classes
- Detailed Design Phases
  - Specification of the internal data structures and specific algorithms within classes

Conceptual Introduction

- You need to sample an analog signal with your micro's A/D converter.
  - Ranges from -5 to +5 volts
  - A/D runs from 0-5 volts.

Problem:

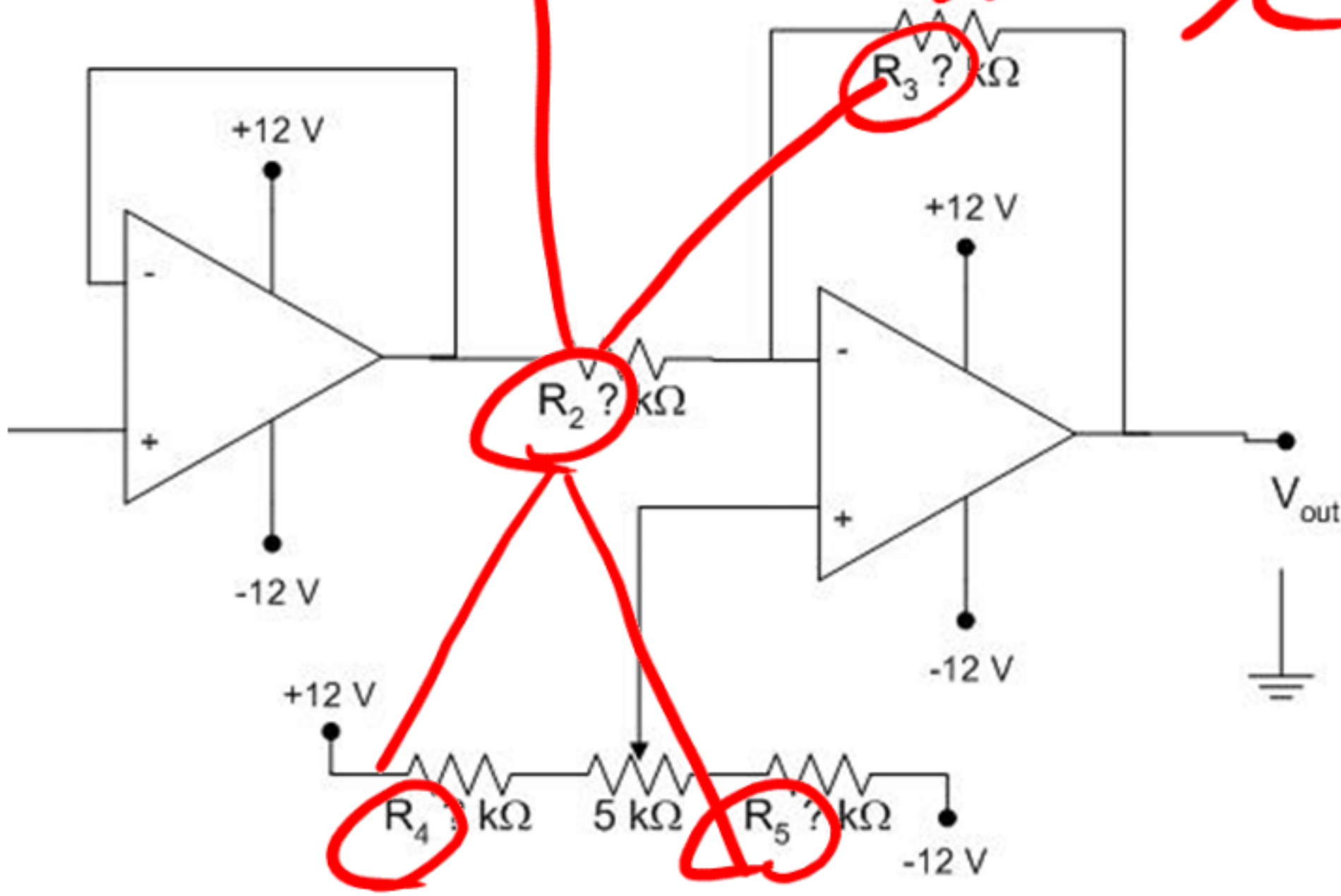
1. Attenuate the signal.

2. Remove bias from signal.

Solution

Op Amp Circuit

Remove bias and scale vs Hose



Common circuits for the problems.

Design Pattern

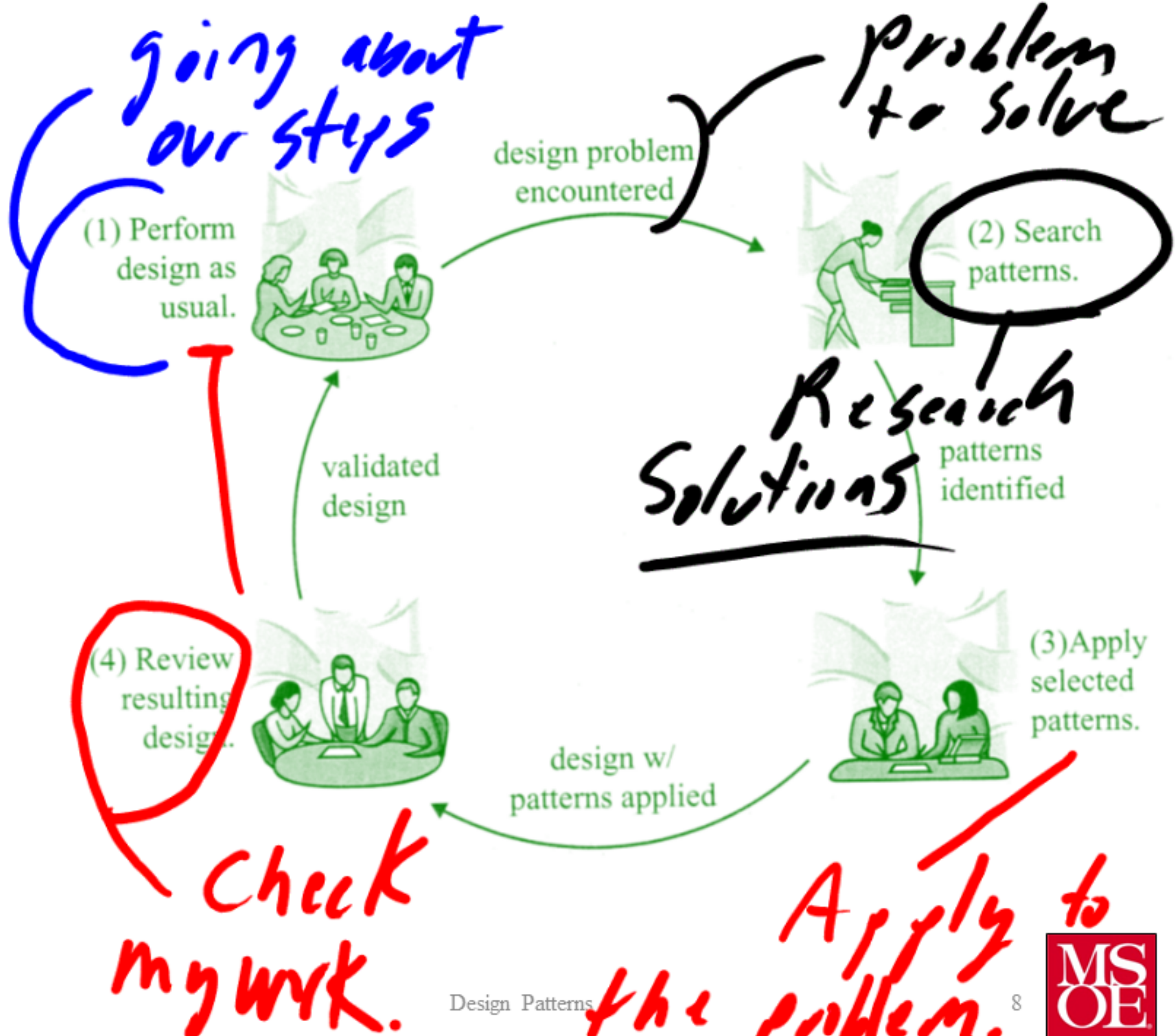
- A general reusable solution to a commonly occurring problem in engineering
  - AKA engineering cookbook

(Ways to solve the problem.

Software design Patterns

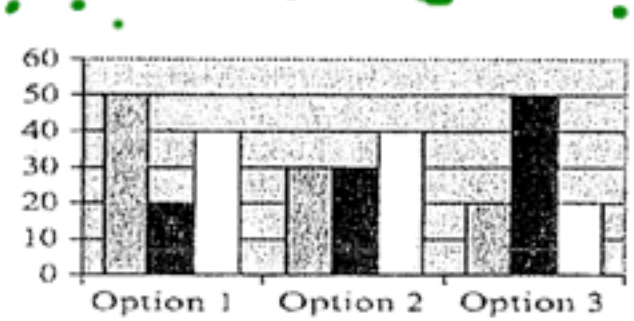
- Three aspects
  - A common problem, including a common problem context
  - A general approach to the solution
  - Consequences of using the pattern.

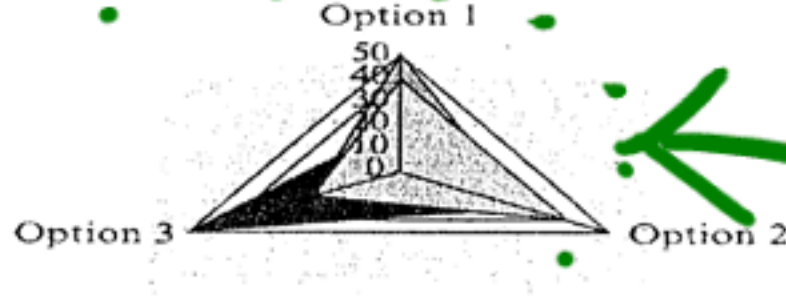
Using Design Patterns Flow



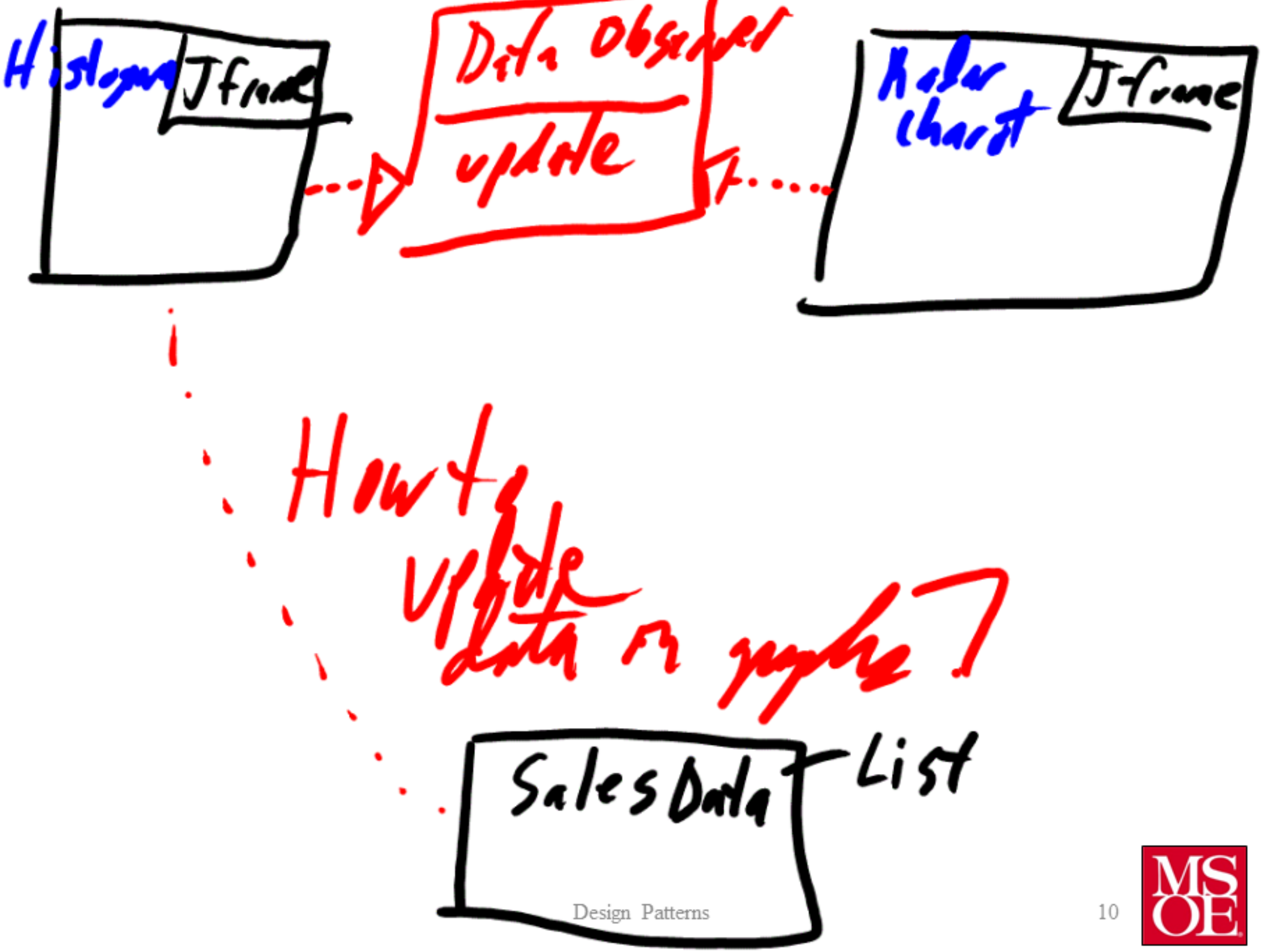
Problem

- I would like to store data representing sales data
- From the sales data, I would like to
  - Display a histogram chart of the data
  - Display a radar chart of the data.
  - How could I do this?





	Opt 1	Opt 2	Opt 3
Market Share	50	30	20
Profit Margin	20	30	50
Op. Cost	40	40	20


Histogram
JFrame
Data Observer
update
Radar chart
JFrame
How to update data or graphs?
SalesData
List

Software design Patterns

- Grouped into common areas
  - Creational Patterns
    - Deal with creating objects
    - Singleton, Abstract Factory, etc.
  - Structural Patterns
    - Common relationships between structures
    - Adapter, bridge, proxy, smart pointer pattern
  - Behavioral Patterns
    - Deal with how a system behaves
    - Roles and responsibilities
    - Command, visitor, state, observer, Reliable Transaction Pattern
  - Concurrency Patterns
    - Guarded, thread pool, etc.

Polling Versus Interrupts

- Polling
  - Periodically checking the data to see if it has changed
- Interrupts
  - "Tell me when something happens"

ISR is invoked when something changes

Friend

Unfriend

Update

Status

MSOE

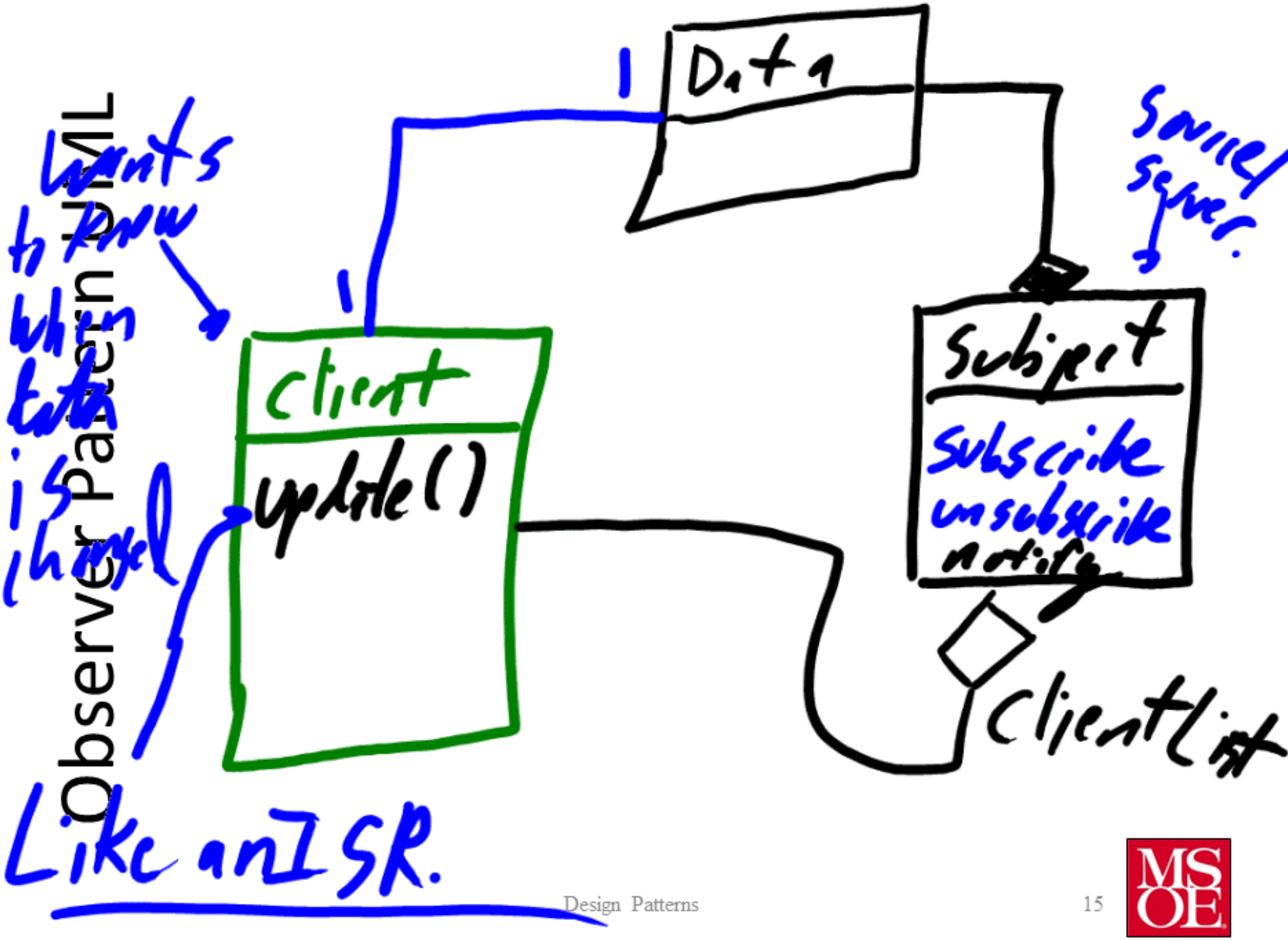
Observer Pattern

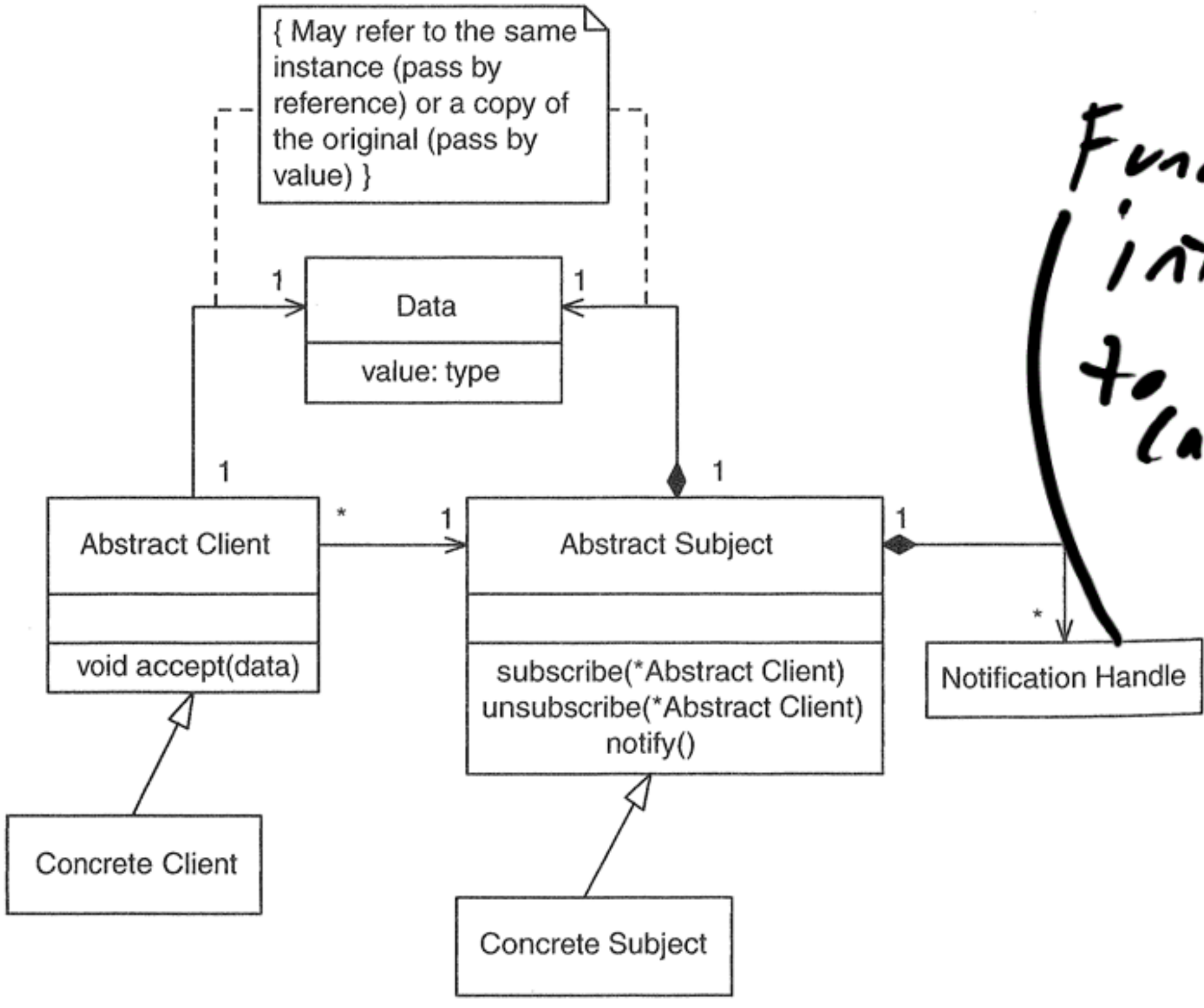
How do we do this?

- Also known as publish / subscribe
  - How to update clients in a timely manner of changes in data
- Clients subscribe to server to be notified when data changes
  - Periodically — once a day
  - On change — immediately on change
  - At most every x time period, etc. — once per hour or every 10 changes

- Interrupts are at a very low level a manifestation of an observer pattern

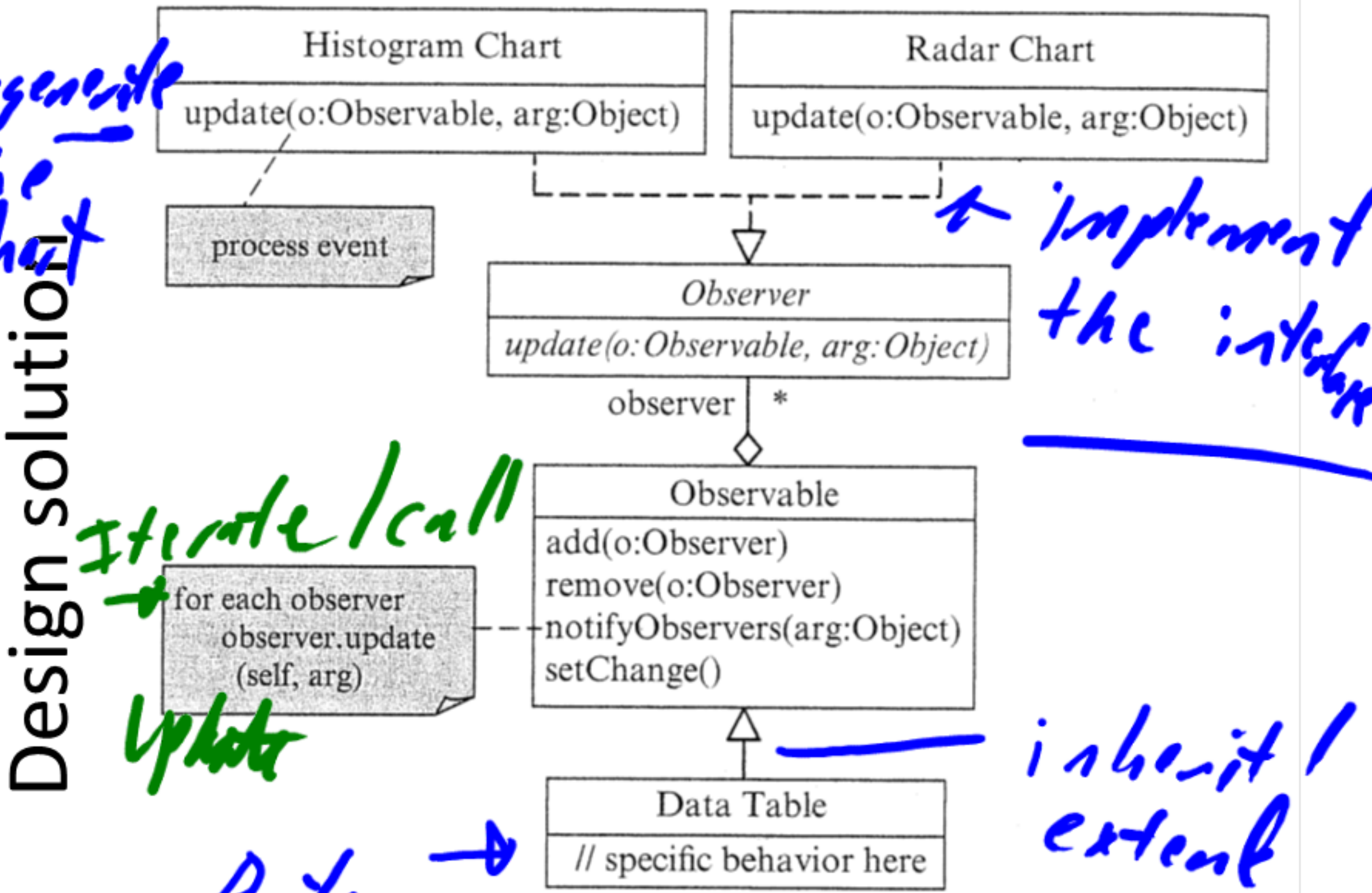
Observer Pattern UML



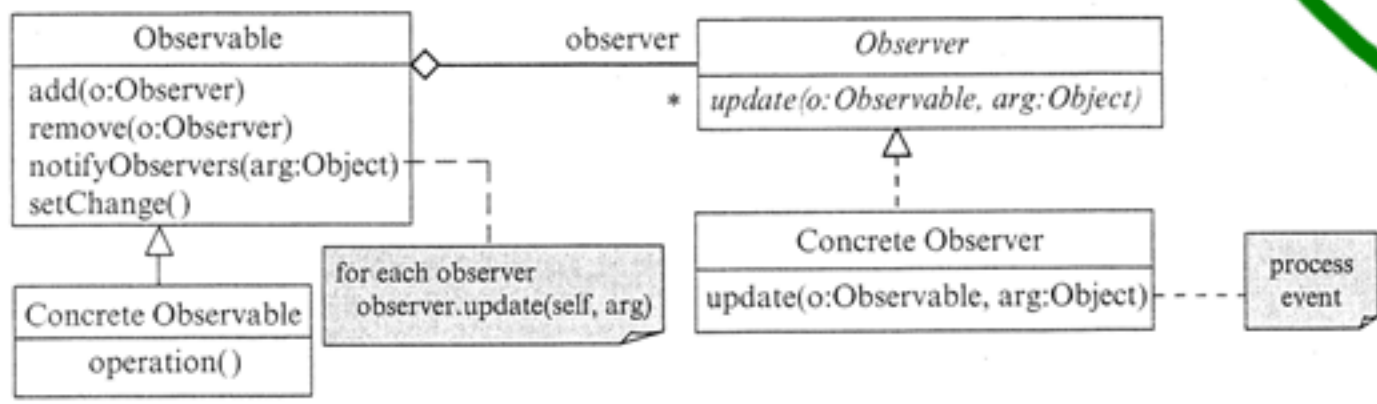
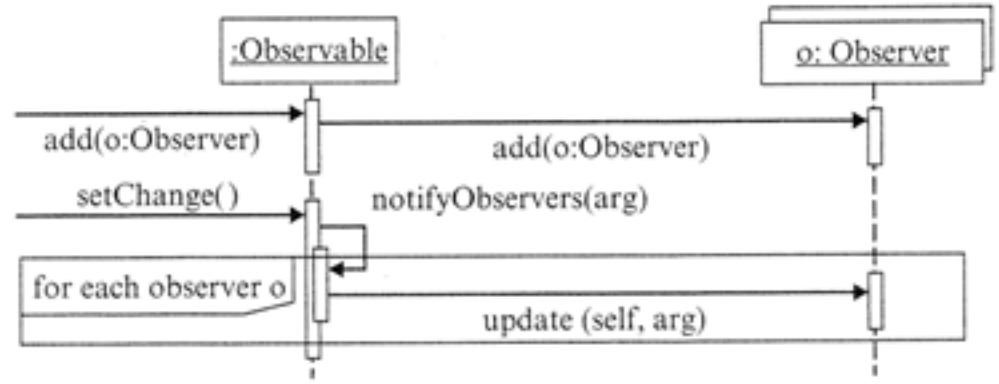


Function interface to call.

Design solution



Observer Pattern Formal Specification

Name	Observer
Type	GoF/Behavioral
Specification	
Problem	How to decouple change and related responses so that such dependencies can be added or removed freely and dynamically.
Solution	Provide a mechanism to add or remove many-to-one dependencies between objects so when one object changes states, all its dependents are notified and updated automatically.
Design	
Structural	
Behavioral	
Roles and Responsibilities	• Observable: This class defines functions for adding, removing, and notifying observers. • Concrete Observable: These classes extend the Observable class to add their own behavior such as operation(). They inherit the functions of the Observable class. • Observer: It defines an interface for all concrete observers. • Concrete Observer: These classes implement the update method to respond to the change.
Benefits	• Observers can be added to, or removed from the observer list of an observable without affecting the observable. • The Observable and the Observer classes can be reused independently. • It supports multicast and broadcast communication. The broadcast mechanism is inherited from the Observable class.
Liabilities	Unwanted concurrent update to a concrete observable may occur.
Guidelines	
Related Patterns	• Protection Proxy can be used to prevent unwanted concurrent update to a concrete observable.
Uses	• Observable and Observer are also Java APIs. The Java ActionLister API is a special case of Observer.

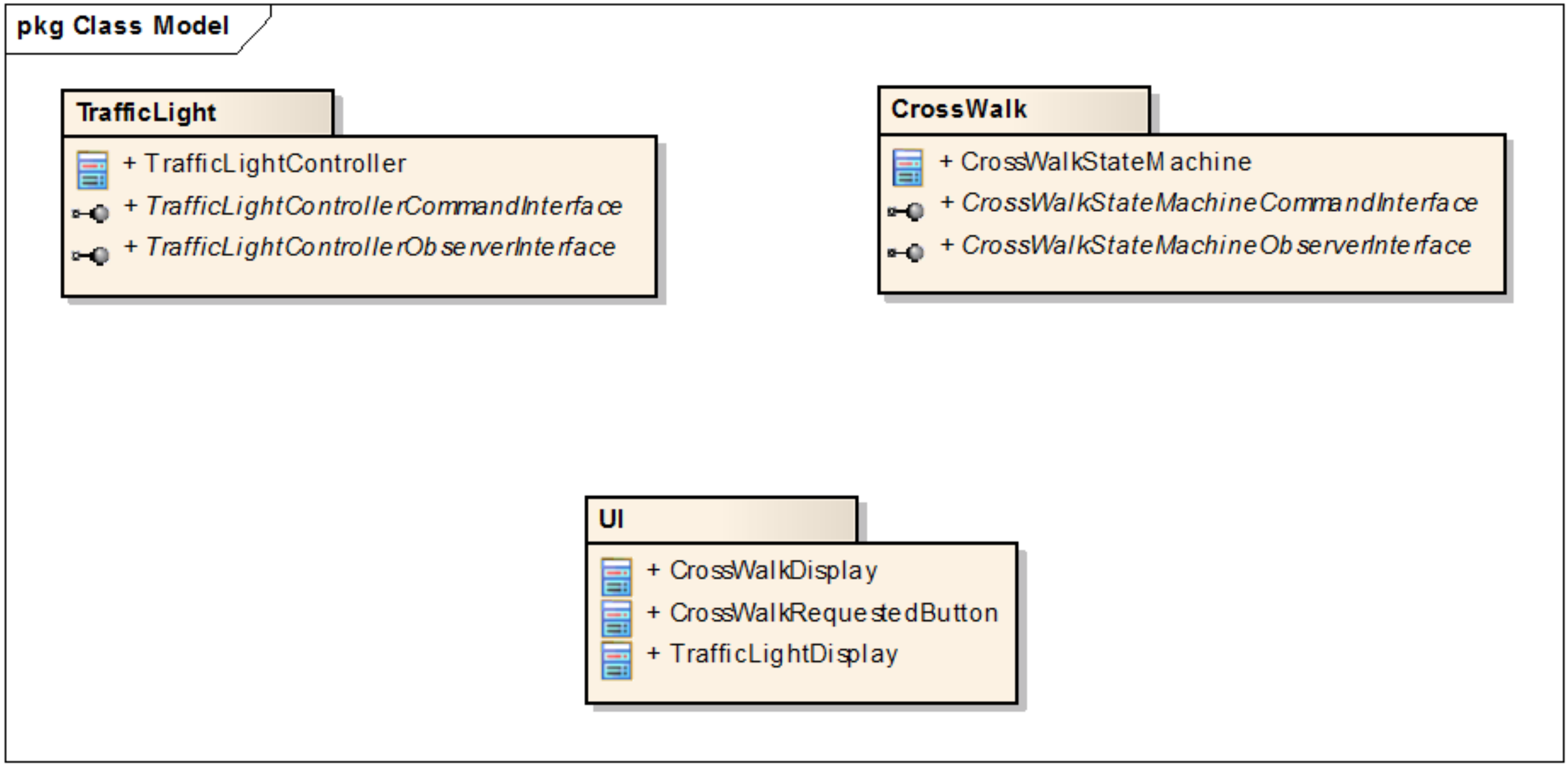
Advantages

- Decouples client – server relationships
- Allows easy modification of listeners
- Updates managed out of one place (aka the server)

Class coupling is good.

Easy to change

Easy to keep track of.


pkg Class Model
TrafficLight
+ TrafficLightController
+ TrafficLightControllerCommandInterface
+ TrafficLightControllerObserverInterface
CrossWalk
+ CrossWalkStateMachine
+ CrossWalkStateMachineCommandInterface
+ CrossWalkStateMachineObserverInterface
UI
+ CrossWalkDisplay
+ CrossWalkRequestedButton
+ TrafficLightDisplay

JPanel
ActionListener
UI::CrossWalkRequestedButton

- serialVersionUID: long = 1L {readOnly}
- cwc: CrossWalkStateMachineCommandInterface

+ CrossWalkRequestedButton()
+ actionPerformed(ActionEvent) : void
+ getCwc() : CrossWalkStateMachineCommandInterface
+ setCwc(CrossWalkStateMachineCommandInterface) : void

-cwc

«interface»
CrossWalk::CrossWalkStateMachineCommandInterface

+ RequestStreetCrossing: int = 1 {readOnly}

+ requestAction(int) : void
+ subscribe(CrossWalkStateMachineObserverInterface) : void
+ unsubscribe(CrossWalkStateMachineObserverInterface) : void

JPanel
UI::TrafficLightDisplay

- serialVersionUID: long = 1L {readOnly}
- buttons: JButton ([]) = new JButton[3]

+ TrafficLightDisplay()
+ TrafficLightDisplay(int, int)
+ AddButtons() : void
+ updateTrafficLightState(int) : void

Instantiates

JPanel
UI::CrossWalkDisplay

- serialVersionUID: long = 1L {readOnly}
- buttons: JButton ([]) = new JButton[2]

+ CrossWalkDisplay()
+ CrossWalkDisplay(int, int)
+ AddButtons() : void
+ updateCWState(int) : void

Instantiates

Main

+ main(String) : void

«interface»
TrafficLight::
TrafficLightControllerObserverInterface

+ RED_LIGHT: int = 1 {readOnly}
+ YELLOW_LIGHT: int = 2 {readOnly}
+ GREEN_LIGHT: int = 3 {readOnly}

+ updateTrafficLightState(int) : void

-observers

«interface»
CrossWalk::
CrossWalkStateMachineObserverInterface

+ ALL_OFF: int = 0 {readOnly}
+ DW_ILLUMINATED: int = 1 {readOnly}
+ DW_FLASH_ON: int = 2 {readOnly}
+ DW_FLASH_OFF: int = 3 {readOnly}
+ DW_DARKENED: int = 4 {readOnly}
+ WALK_ILLUMINATED: int = 5 {readOnly}

+ updateCWState(int) : void

-obsservers

Runnable
CrossWalk::CrossWalkStateMachine

- obsservers: List<CrossWalkStateMachineObserverInterface> = new ArrayList<C...
- tlc: TrafficLightControllerCommandInterface
- redLight: boolean = false
- crossRequested: boolean = false

+ CrossWalkStateMachine()
- updateState(int) : void
+ requestAction(int) : void
+ subscribe(CrossWalkStateMachineObserverInterface) : void
+ unsubscribe(CrossWalkStateMachineObserverInterface) : void
+ updateTrafficLightState(int) : void
+ run() : void
+ getTlc() : TrafficLightControllerCommandInterface
+ setTlc(TrafficLightControllerCommandInterface) : void

-tlc

Runnable
TrafficLight::TrafficLightController

- state: int = TrafficLightCon...
- observers: List<TrafficLightControllerObserverInterface> = new ArrayList<T...
- redLightRequested: boolean = false
- greenLightOK: boolean = true

+ requestAction(int) : void
+ TrafficLightController()
+ subscribe(TrafficLightControllerObserverInterface) : void
+ unsubscribe(TrafficLightControllerObserverInterface) : void
- UpdateObservers() : void
- changeLightToRed() : void
+ updateCWState(int) : void
+ run() : void

«interface»
TrafficLight::TrafficLightControllerCommandInterface

+ SAFE_FOR_TRAFFIC_TO_DRIVE_REQUEST: int = 2 {readOnly}

+ requestAction(int) : void
+ subscribe(TrafficLightControllerObserverInterface) : void
+ unsubscribe(TrafficLightControllerObserverInterface) : void

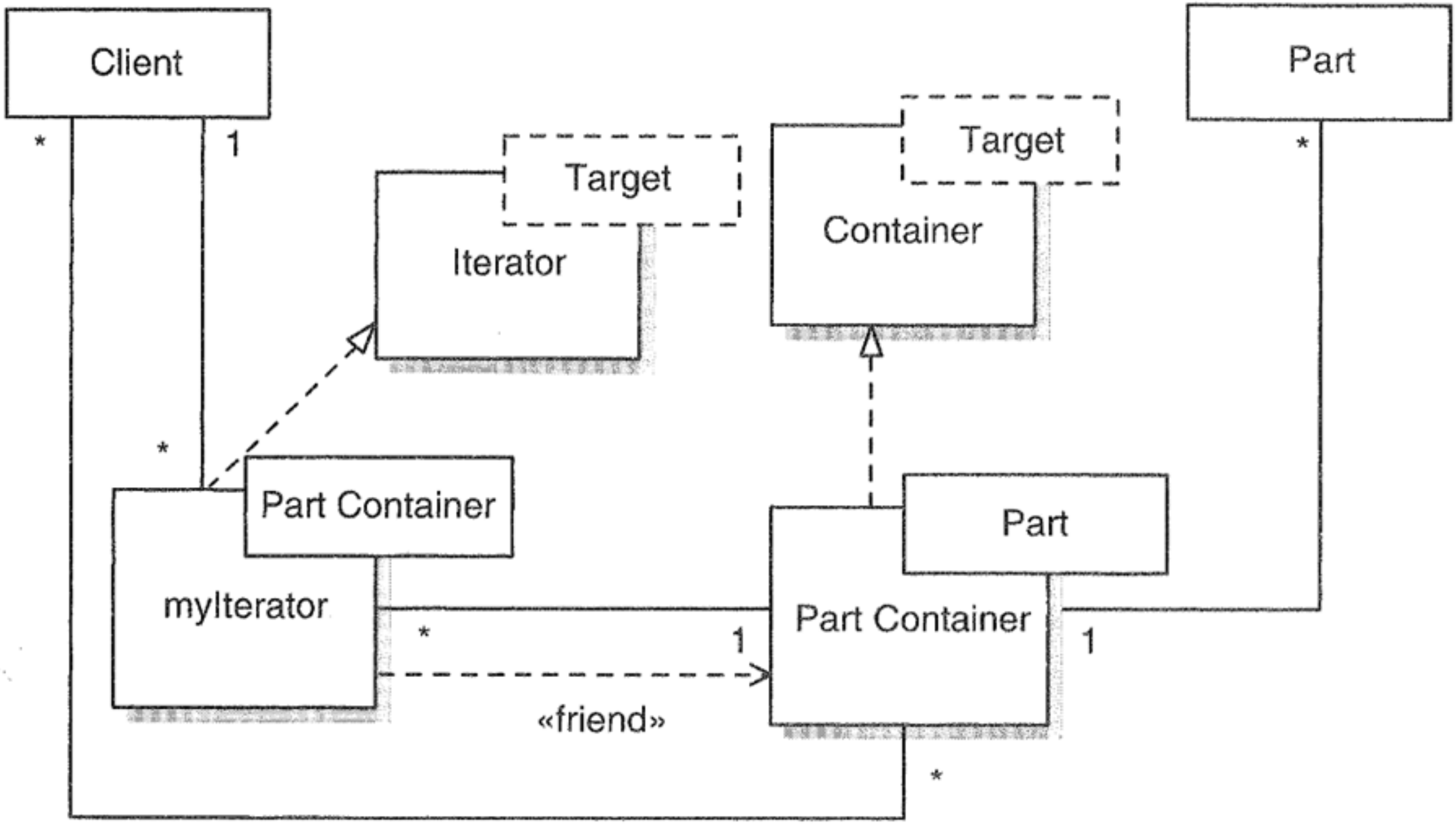
Note: Main instantiates all GUI items and all threads. Association not shown to help clarify diagram.

Lets look at some code…

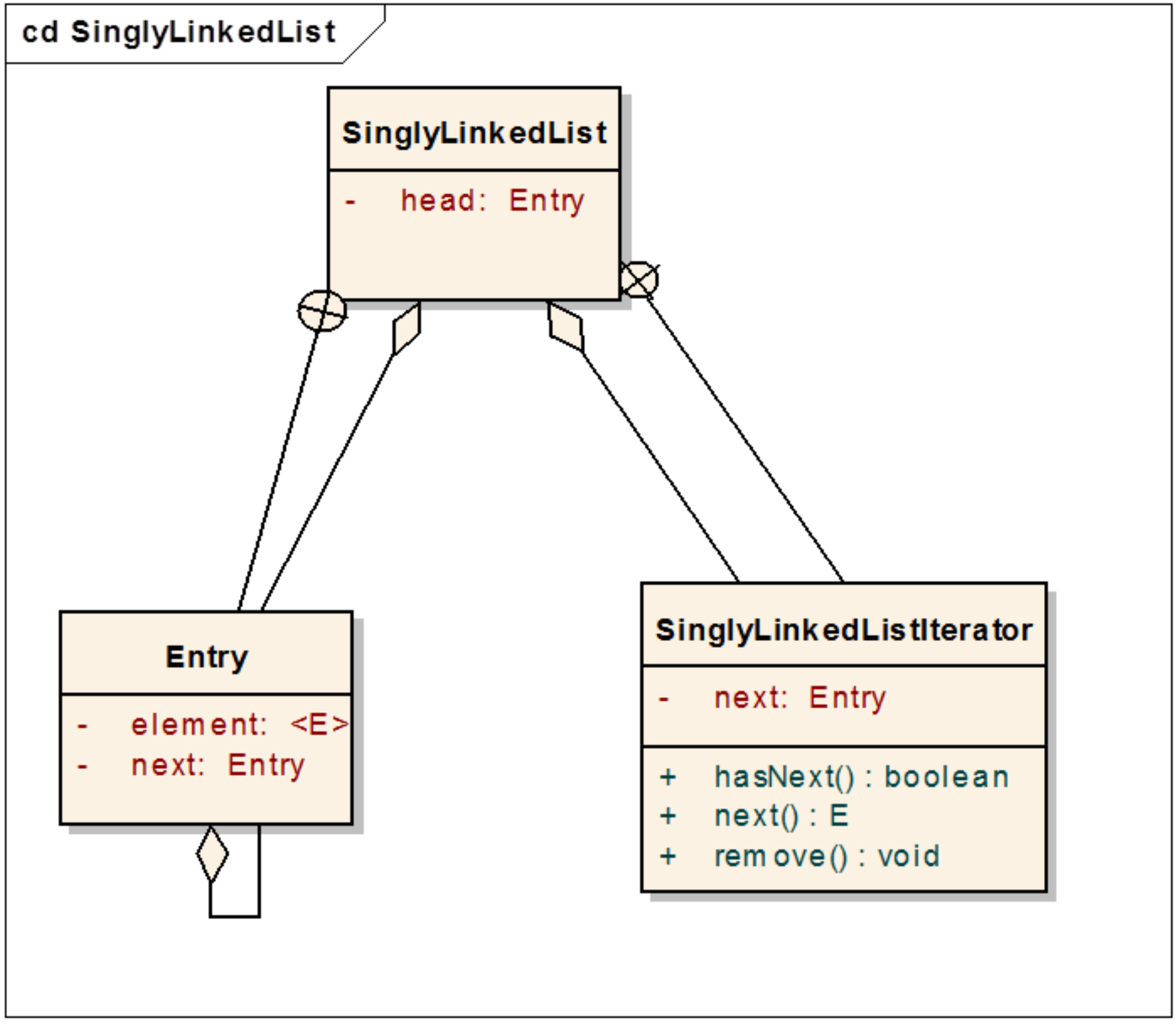
Container Pattern

- Normal associations only work for
  - (0,1)-(0,1)
  - 1-1
- Will not work for multi-valued roles

- Used for one to many associations
  - You've seen this before…


Client
*
1
Part
*
Target
Iterator
Target
Container
*
Part Container
Part
myIterator
Part Container
*
1
1
«friend»
*

Iterator


cd SinglyLinkedList
SinglyLinkedList
- head: Entry
Entry
- element: <E>
- next: Entry
SinglyLinkedListIterator
- next: Entry
+ hasNext() : boolean
+ next() : E
+ remove() : void

Container Pattern Usage