FACULTY OF ENGINEERING
AND ARCHITECTURE

SE 2 : DESIGN PATTERNS

Bart Dhoedt

Academiejaar 2017-2018

DESIGN PATTERN ?

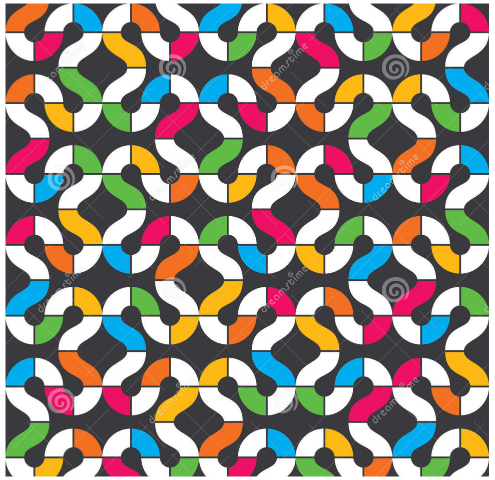
= General solution to recurring design problem

Sorts of problems

- Creation of objects/ Application configuration
- Class hierarchies
- Object interaction

Discussion of Patterns

- Name : facilitate communication about software design
- Problem description
- General Solution
- Implementation Issues (alternatives)
- Consequences (pro's vs cons)



THE GANG OF FOUR CATALOGUE

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of resp. Command Iterator Mediator Memento Observer State Strategy Visitor

CHAPTER CONTENT

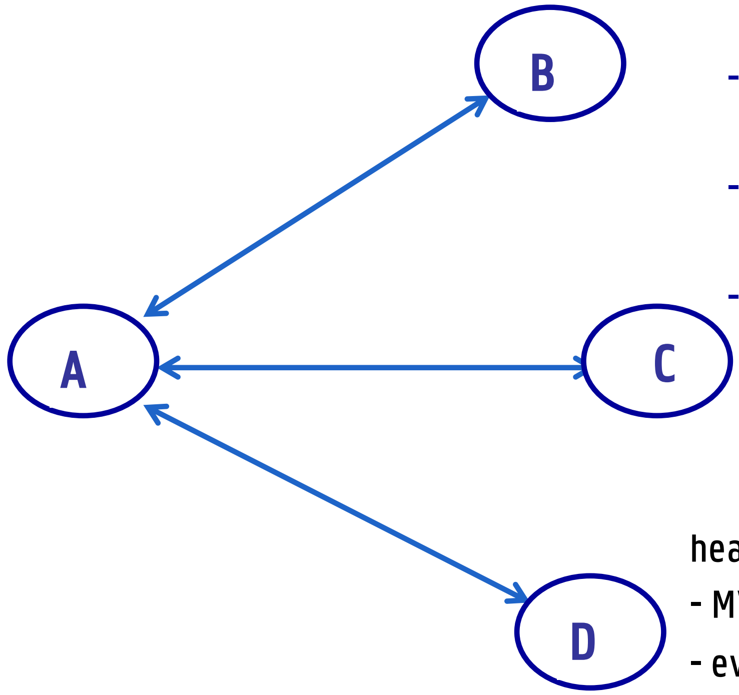
In this lecture : illustrate design patterns heavily used in frameworks (JavaFX, Android, …)

Running example : simple event broker

1. OBSERVER

OBSERVER : MOTIVATION

Organize efficiently one-to-many relations between objects

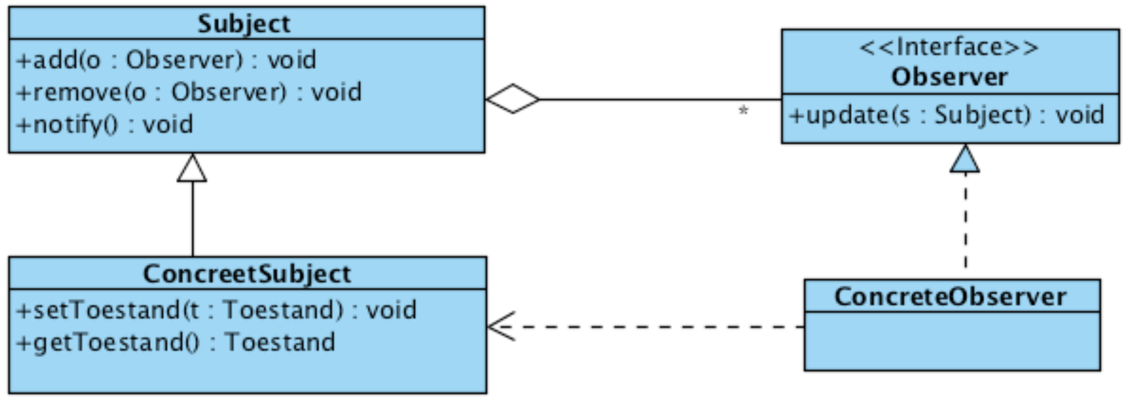


- propagate state changes of A to B, C and D
- dynamically manage collection of objects to notify
- objects to notify not known in advance

heavily used in

- MVC based systems
- event based systems (publish-subscribe)

GENERAL SOLUTION


Subject
+add(o : Observer) : void
+remove(o : Observer) : void
+notify() : void
<<Interface>>
Observer
+update(s : Subject) : void
*
ConcreetSubject
+setToestand(t : Toestand) : void
+getToestand() : Toestand
ConcreteObserver

Subject

- knows all Observers watching
- provides interface for adding/removing Observers

Observer: defines update-interface for observing objects

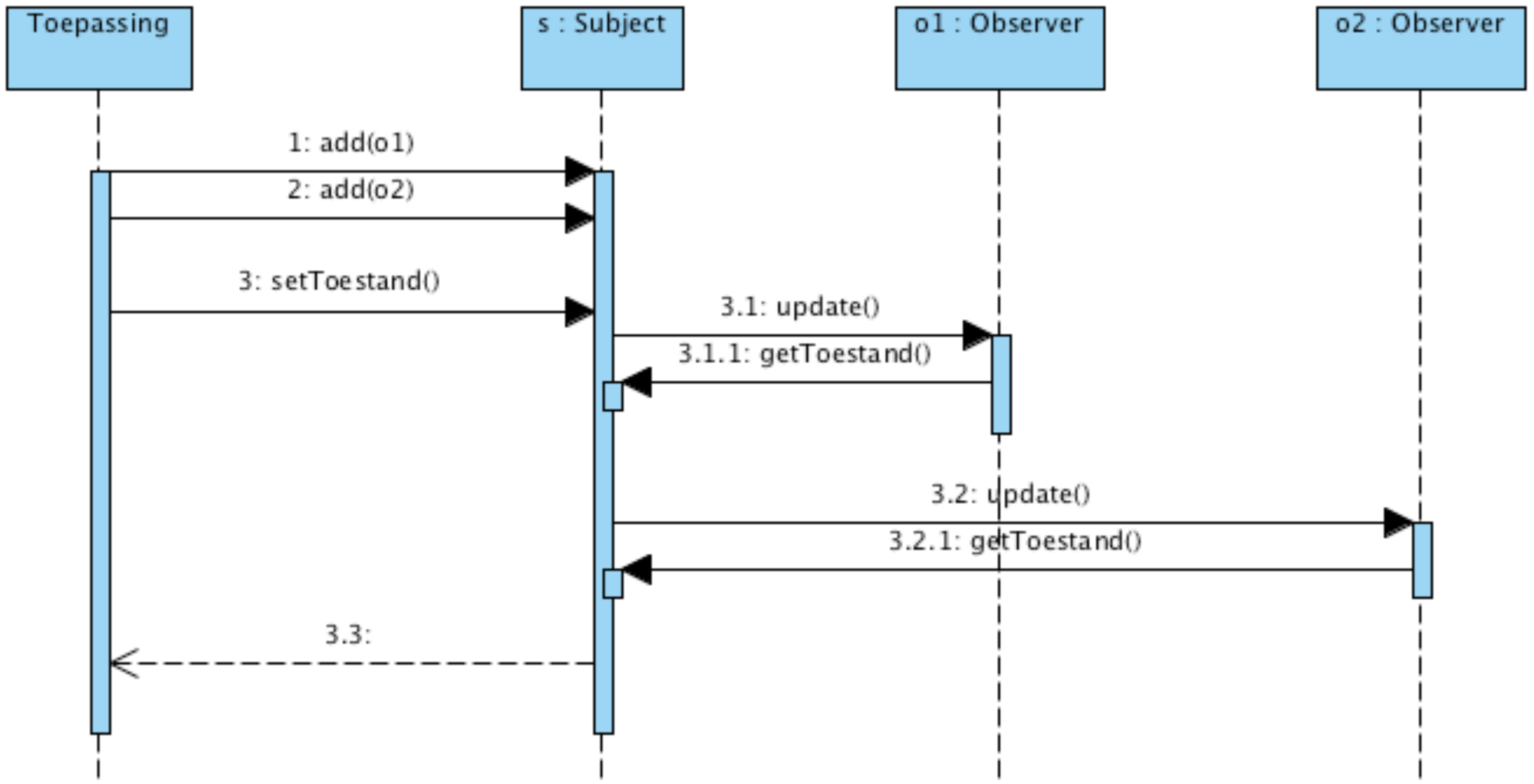
ConcreteSubject

- stores relevant state for ConcreteObservers
- notifies when relevant state changes

ConcreteObserver

- has reference to ConcreteSubject (possibly through update())
- implements Observer interface, keeps state consistent

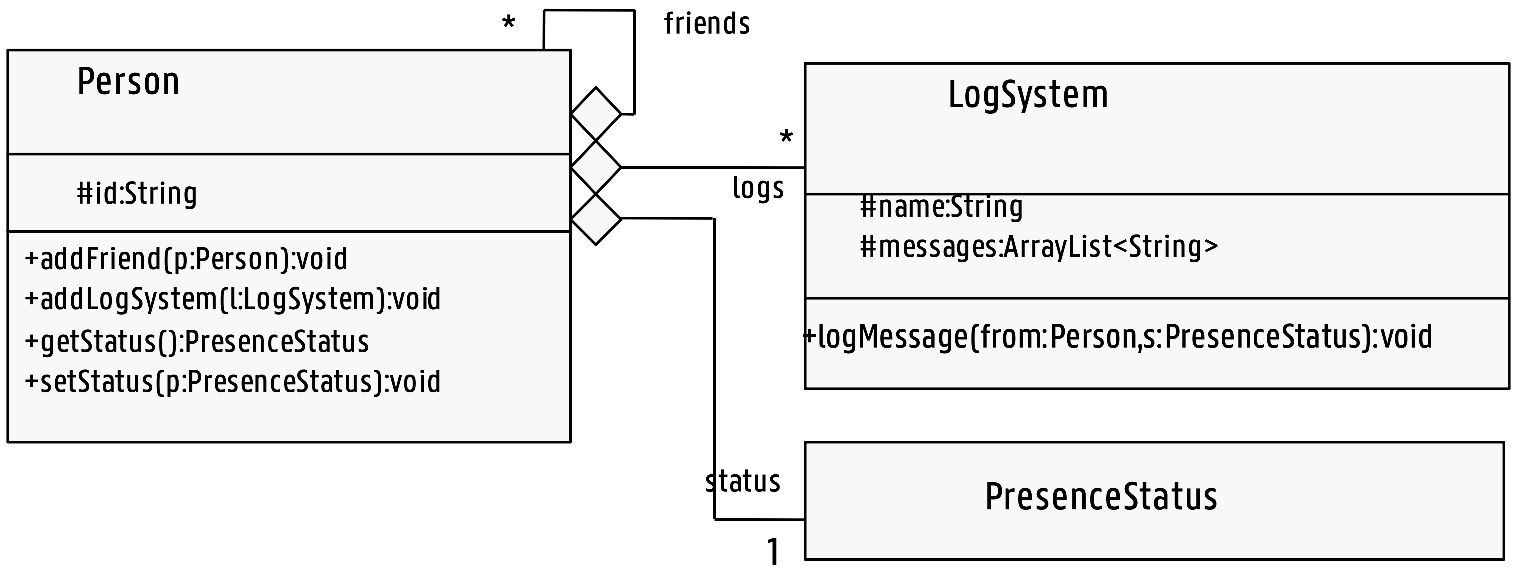
GENERAL SOLUTION



EXAMPLE : PRESENCE SERVICE

Problem

- Person has PresenceStatus
- Each Friend is notified of status change
- LogSystem keeps track of status history for each Person

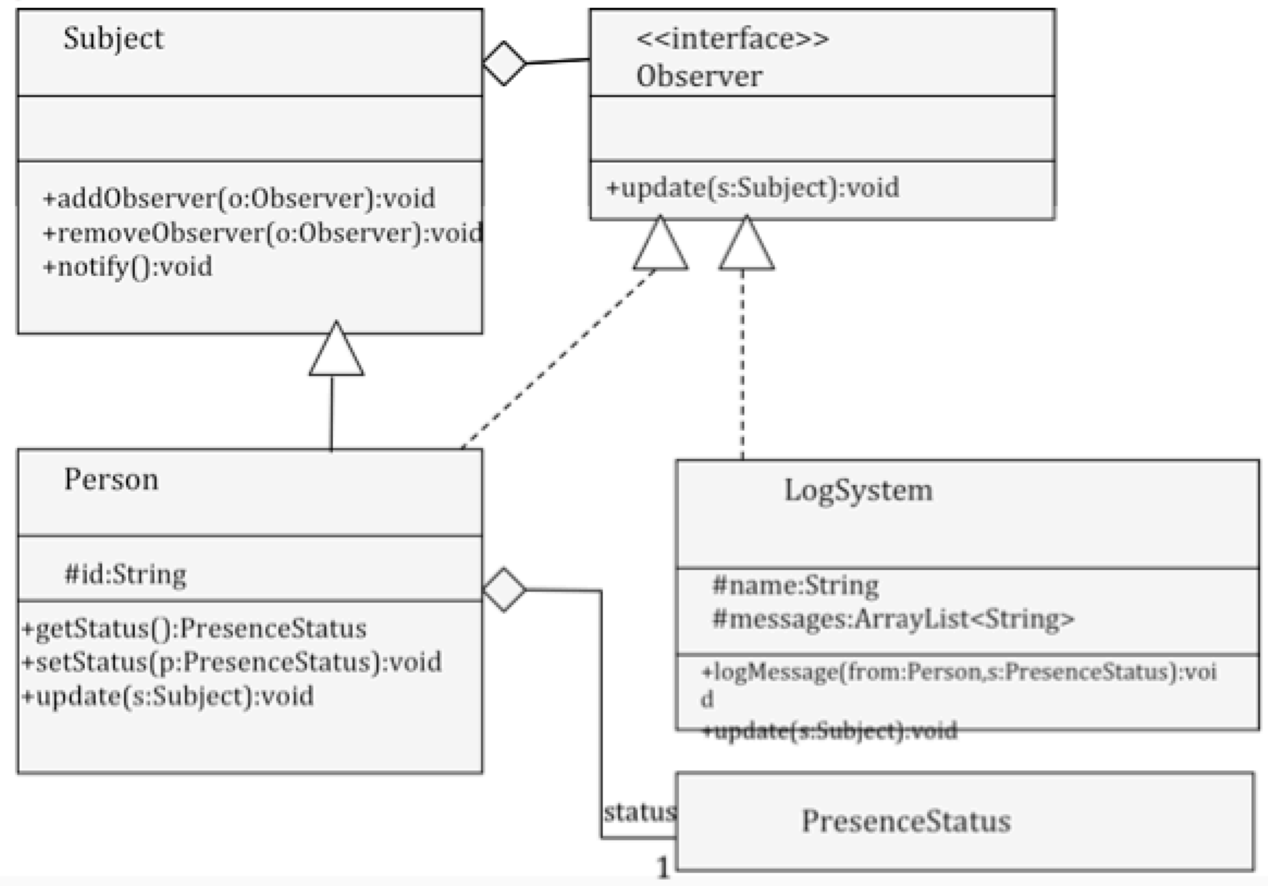

*
friends
Person
#id:String
+addFriend(p:Person):void
+addLogSystem(l:LogSystem):void
+getStatus():PresenceStatus
+setStatus(p:PresenceStatus):void
*
logs
LogSystem
#name:String
#messages:ArrayList<String>
+logMessage(from:Person,s:PresenceStatus):void
status
1
PresenceStatus

Modify to reduce coupling using Observer

EXAMPLE : PRESENCE SERVICE

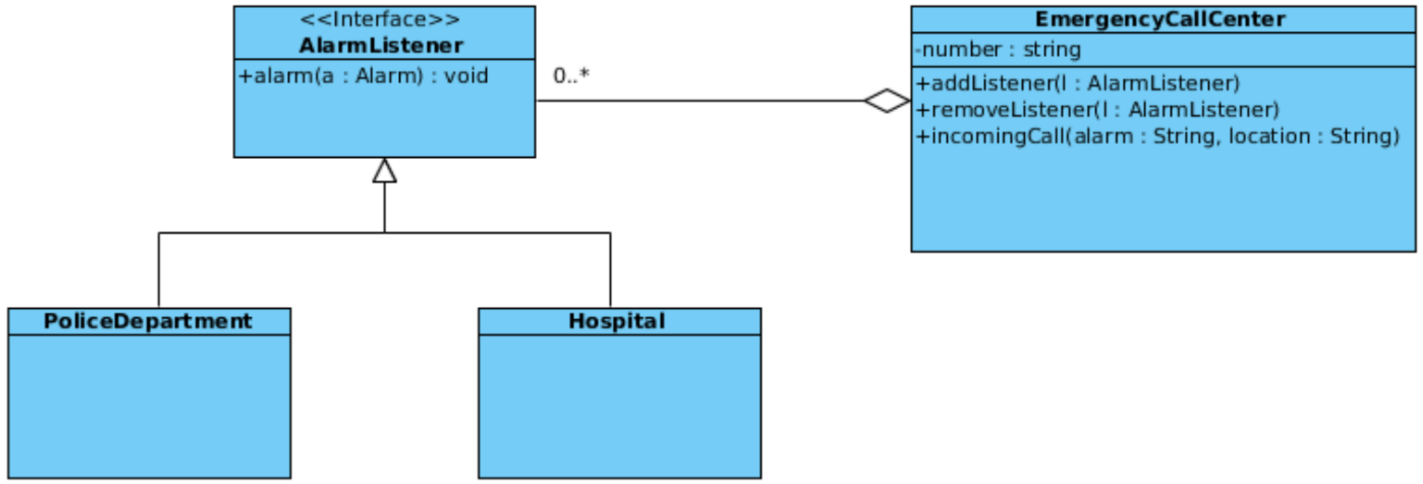
Solution

UML


Subject
+addObserver(o:Observer):void
+removeObserver(o:Observer):void
+notify():void
<<interface>>
Observer
+update(s:Subject):void
Person
#id:String
+getStatus():PresenceStatus
+setStatus(p:PresenceStatus):void
+update(s:Subject):void
LogSystem
#name:String
#messages:ArrayList<String>
+logMessage(from:Person,s:PresenceStatus):void
+update(s:Subject):void
status
1
PresenceStatus

Modify to reduce coupling using Observer

OEFENING 1 : OBSERVER



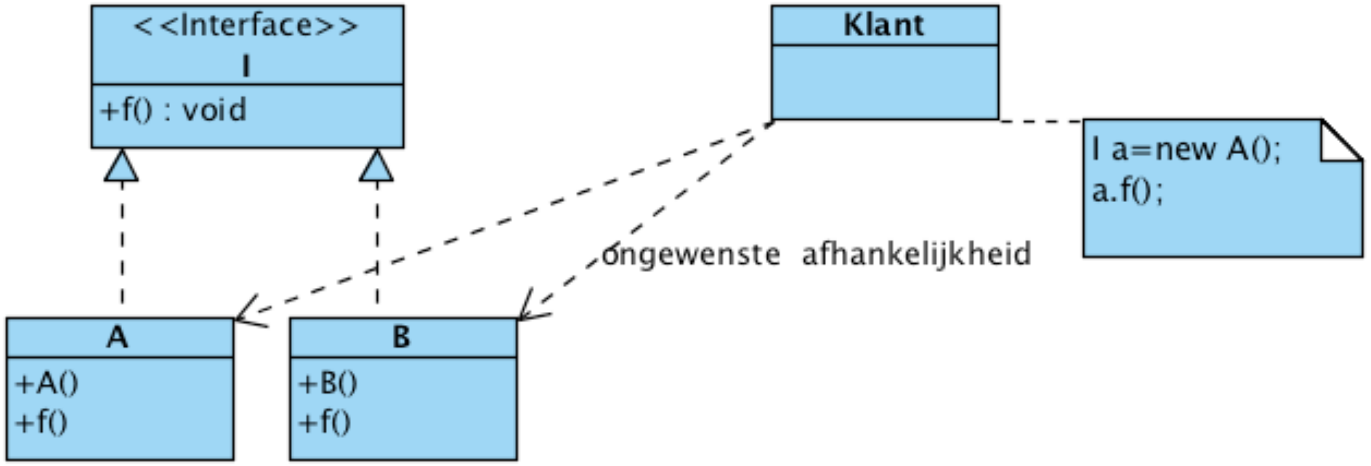
<<Interface>>
AlarmListener
+alarm(a : Alarm) : void
0..*
EmergencyCallCenter
-number : string
+addListener(l : AlarmListener)
+removeListener(l : AlarmListener)
+incomingCall(alarm : String, location : String)
PoliceDepartment
Hospital

- Download code via Minerva
- Voer alarm.Main uit en bekijk de code
- Extra ziekenhuis “AZ” luistert naar nummer 112
- Extra nummer 101, enkel PoliceDepartment als observer
- Extra oproep “burglary” in TechnologiePark
- Bijkomende klasse FireDepartment, bij alarm wordt afgedrukt :

  "Fire squad on the move to <location> for <alarmtype> "

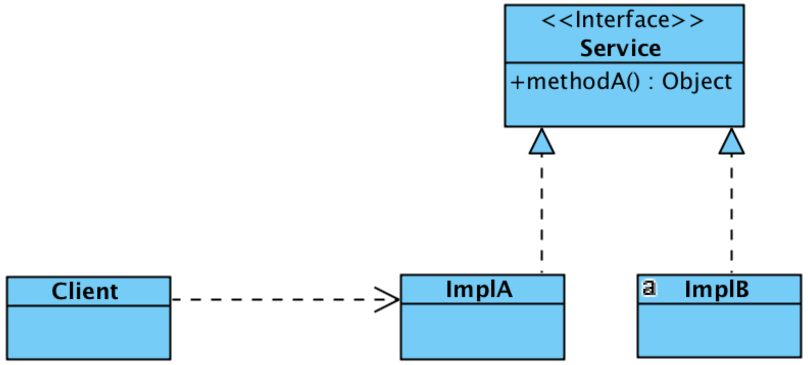
2. FACTORY

CREATIONAL PATTERNS: NO CONSTRUCTOR POLYMORPHISM



FACTORY : MOTIVATION

Configuration of application with objects
Explicit constructor calls : difficult to maintain !



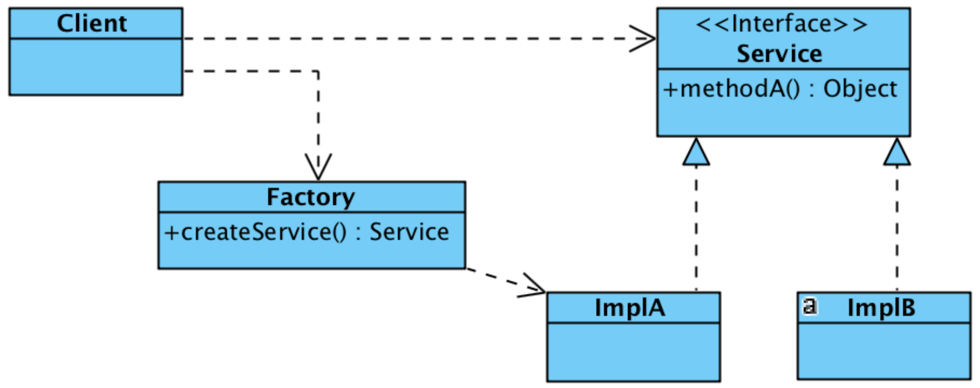
```
class Client {
        ....
        Service s1 =new ImplA();
        ....
}

class AggService {
        private Service s2 = new ImplA();
        ....
}
```

Changing ImplA -> ImplB requires lots of updates !

FACTORY : MOTIVATION

Configuration of application with objects

Explicit constructor calls : difficult to maintain !



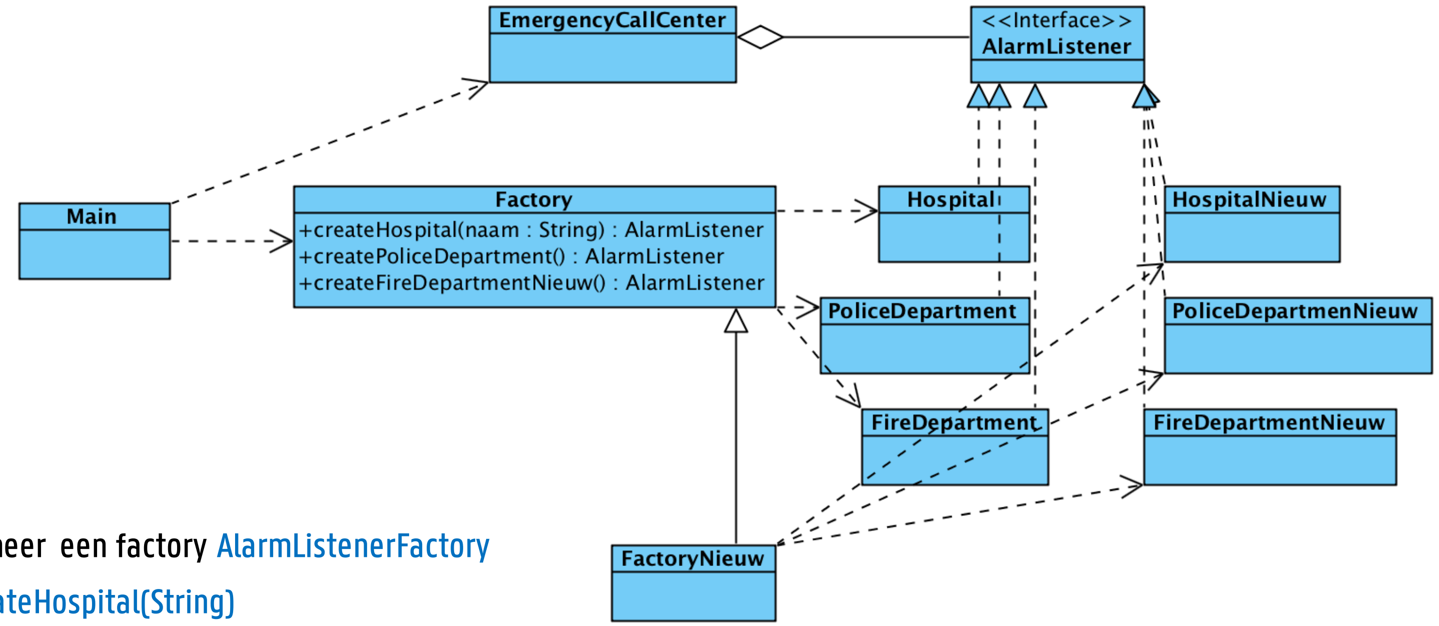
```
class Client {
        ....
        Factory f = new Factory()
        Service s1 = f.createService()
        ....
}

class  AggService  {
        private  Service  s2 = (new Factory()).createService();
       ....
}
```

Changing ImplA -> ImplB requires ONLY change in Factory-logic

Further sophistication:

- hierarchy of factories
- implement factory using Singleton pattern

OEFENING 2 : FACTORY

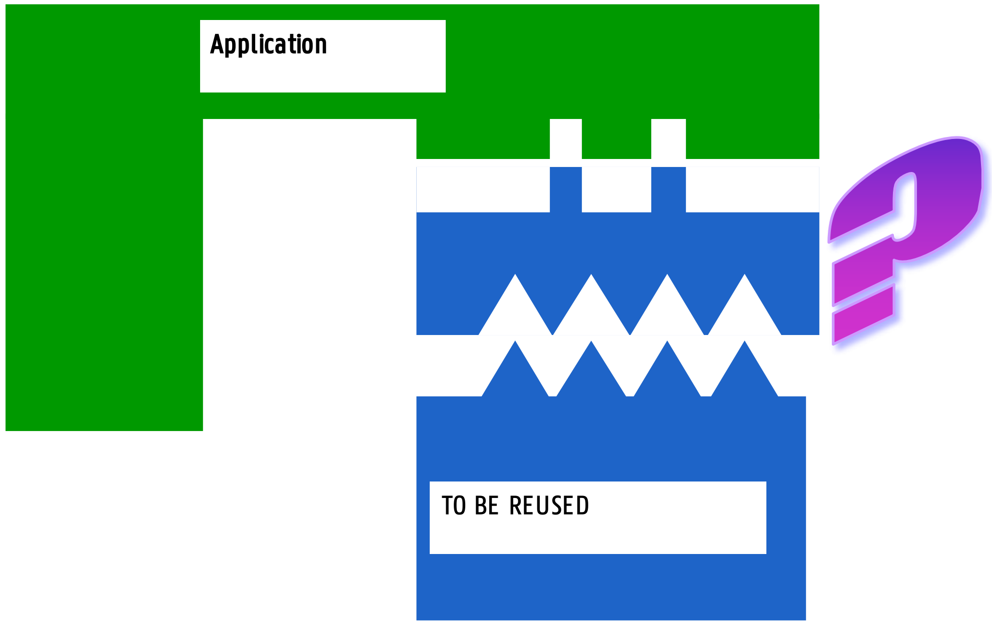


- Programmeer een factory AlarmListenerFactory
  - createHospital(String)
  - createPoliceDepartment()
  - createFireDepartment()
- Verwijder all constructoroproepen naar de klassen Hospital, PoliceDepartment, FireDepartment uit Main. Controleer!
- Maak nieuwe klassen HospitalNieuw, PoliceDepartmentNieuw en FireDepartmentNieuw
  - Overal "NIEUW:" voor de alarmboodschap
- Programmeer FactoryNieuw
- Pas Main-code aan

3. ADAPTER

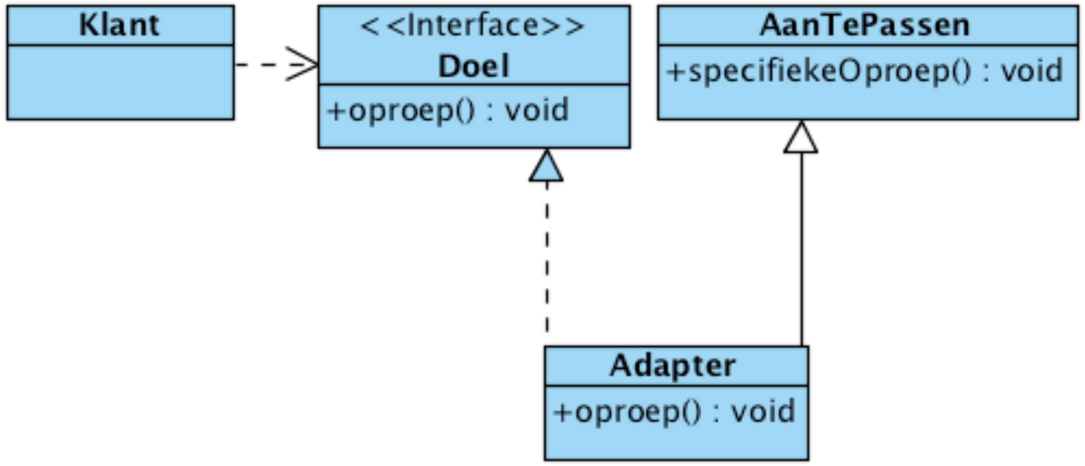
MOTIVATION

- system wants to reuse existing class
- class has WRONG interface

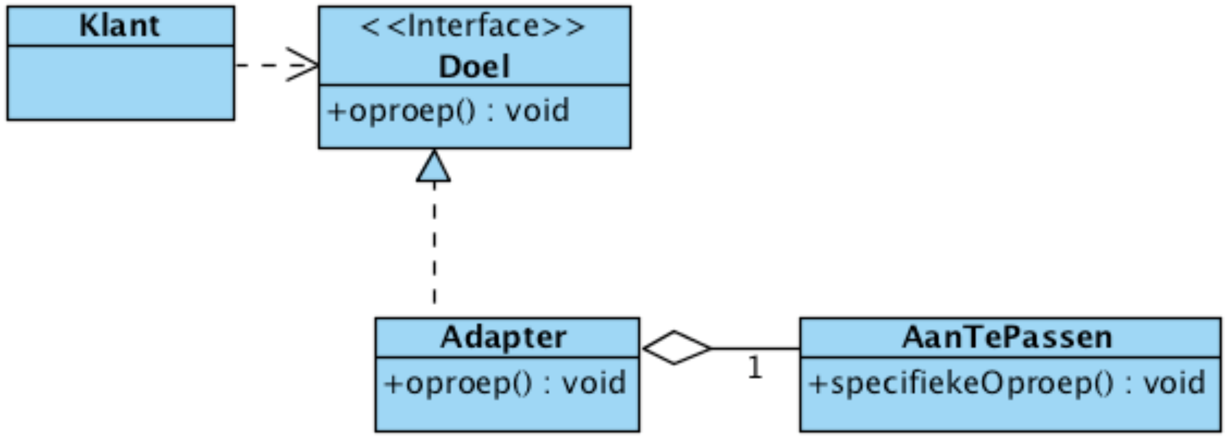


In absence of source code !

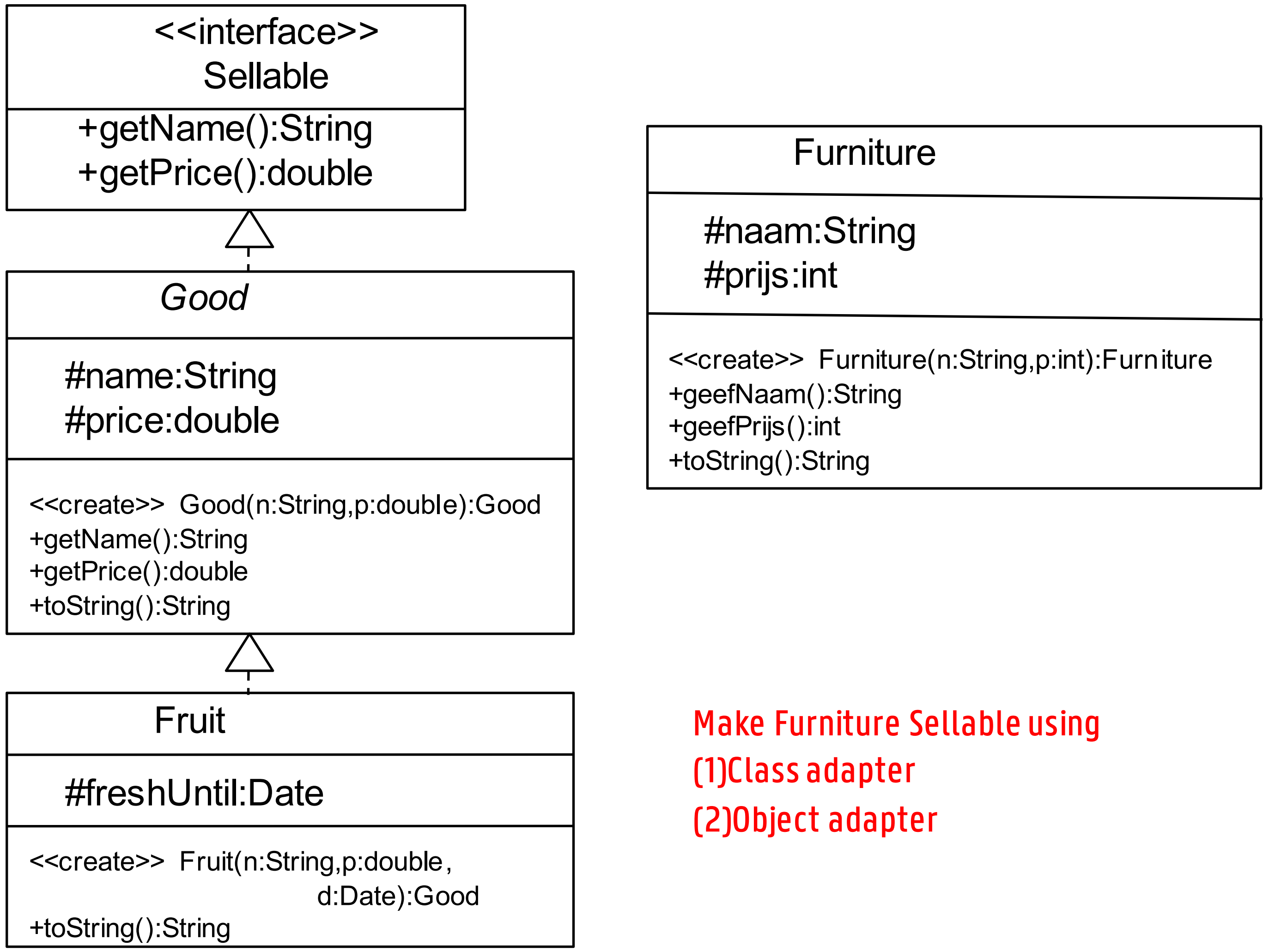
THE GENERAL SOLUTION : CLASS ADAPTER



THE GENERAL SOLUTION : OBJECT ADAPTER



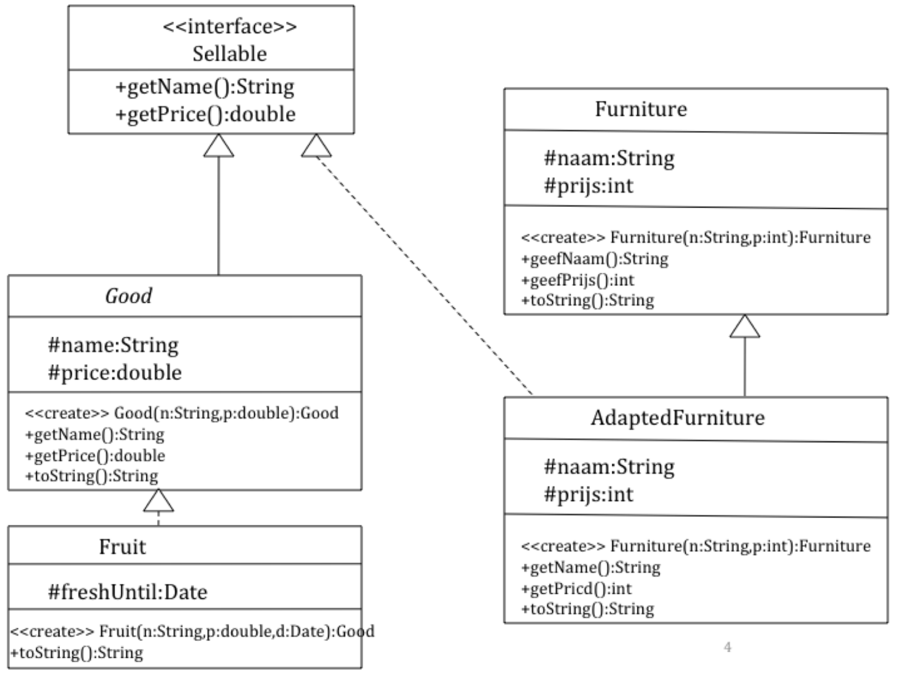
EXERCISE : SELLABLE FURNITURE



Make Furniture Sellable using
(1)Class adapter
(2)Object adapter

EXERCISE : SELLABLE FURNITURE

Solution : Class Adapter



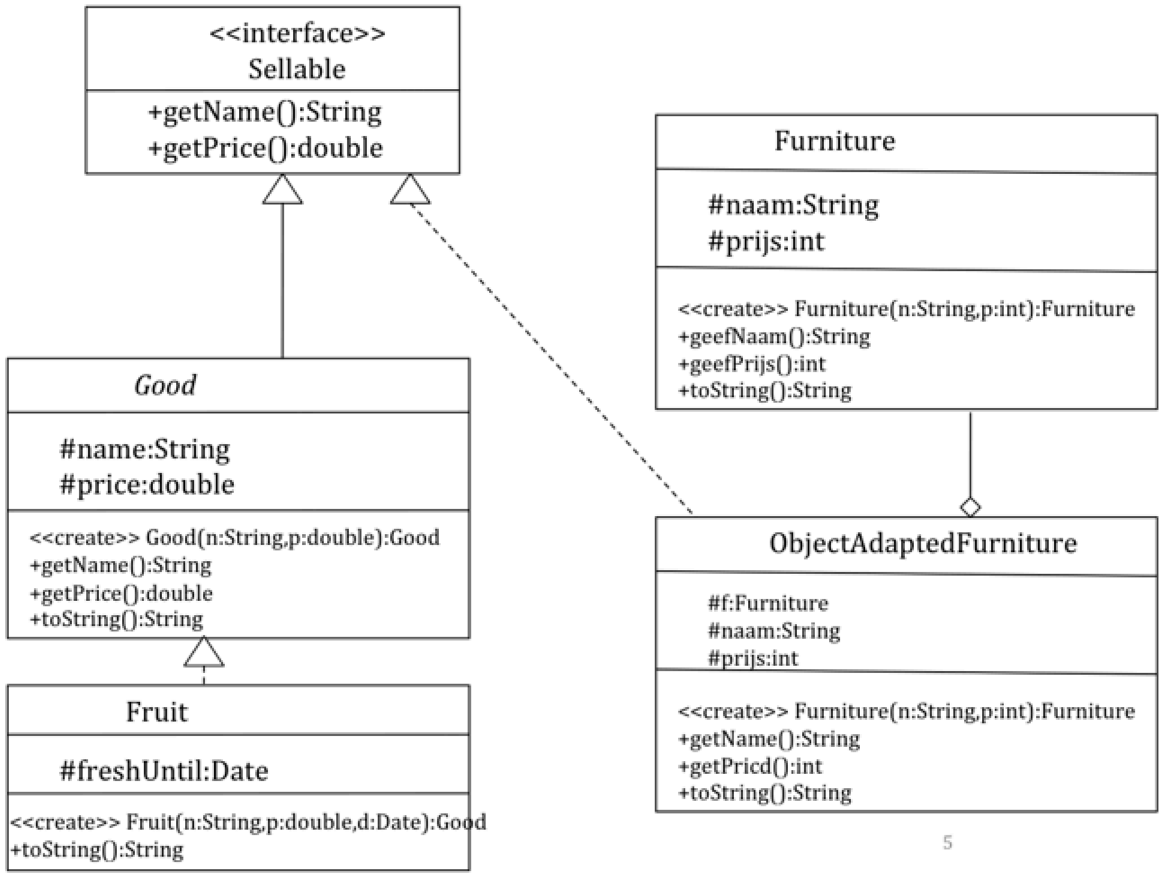
EXERCISE : SELLABLE FURNITURE

Solution : Class Adapter

```
class AdaptedFurniture extends Furniture implements Sellable {
        public AdaptedFurniture(String n,int p) {
                super(n,p);
        }
        public String getName() {
                return geefNaam();
        }
        public double getPrice() {
                return geefPrijs();
        }
        public String toString() {
                return geefNaam()+" cost : "+geefPrijs();
        }
}
```

EXERCISE : SELLABLE FURNITURE

Solution : Object Adapter



EXERCISE : SELLABLE FURNITURE

Solution : Object Adapter

```
class ObjectAdaptedFurniture implements Sellable {
        protected Furniture f;
        public ObjectAdaptedFurniture(Furniture ff){
                f=ff;
        }
        public String getName() {
                return f.geefNaam();
        }
        public double getPrice() {
                return f.geefPrijs();
        }
        public String toString() {
                return f.geefNaam()+" cost : "+f.geefPrijs();
        }
}
```

EXERCISE : SELLABLE FURNITURE

```
public class TestAdapter {
        public static void main(String[] args){
                ArrayList<Sellable> l=new ArrayList<Sellable>();
                l.add(new Fruit("Apple 1",2.5,"1/6/2009"));
                l.add(new Fruit("Greek Grapes",10.3,"15/5/2009"));
                l.add(Furniture ???);
                System.out.println(l);
                System.out.println("Total price : "+computeTotalPrice(l));
        }
        public static double computeTotalPrice(ArrayList<Sellable> l) {
                double total=0.0;
                for(Sellable i:l)
                        total+=i.getPrice();
                return total;
        }
}
```

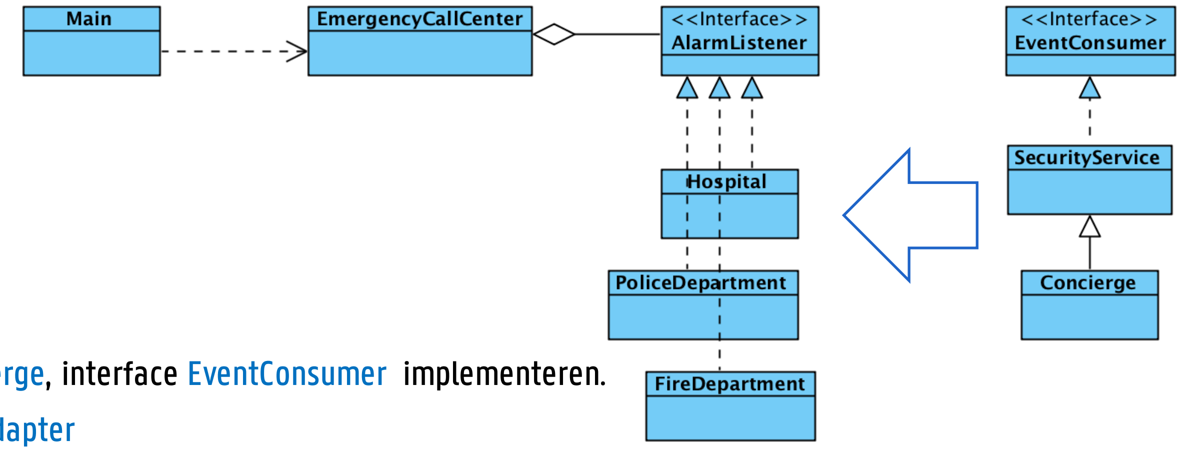
Plug in Furniture as Sellable object
using class/object adapter.

EXERCISE : SELLABLE FURNITURE

Solution

```
public class TestAdapter {
		public static void main(String[] args) {
				ArrayList<Sellable> l=new ArrayList<Sellable>();
				l.add(new Fruit("Apple 1",2.5,"1/6/2009"));
				l.add(new Fruit("Greek Grapes",10.3,"15/5/2009"));
				l.add(new AdaptedFurniture("Table",1033));
				l.add(new AdaptedFurniture("Chair",75));
				l.add(new ObjectAdaptedFurniture(new Furniture("Object Table",1033)));
				l.add(new ObjectAdaptedFurniture(new Furniture("Object Chair",75)));
				System.out.println(l);
				System.out.println("Total price : "+computeTotalPrice(l));
		}
		public static double computeTotalPrice(ArrayList<Sellable> l) {
				double total=0.0;
				for(Sellable i:l)
						total+=i.getPrice();
				return total;
		}
}
```

OEFENING 3 : ADAPTER



Gegeven klassen SecurityService, Concierge, interface EventConsumer implementeren.

- Klasseadapter SecurityServiceClassAdapter
  - Programmeer een klasseadapter SecurityServiceClassAdapter
  - Pas Main-code aan: SecurityService met naam 'Group 4' luistert naar het noodnummer '112'
- Klasseadapter ConciergeClassAdapter
  - Programmeer een klasseadapter ConciergeClassAdapter
  - Pas Main-code aan : 'Concierge met naam 'John McEnzie' luistert naar het noodnummer '112'.
- Objectadapter SecurityServiceObjectAdapter
  - Programmeer een objectadapter SecurityServiceObjectAdapter
  - Pas Main-code aan :
    - SecurityService met naam 'SecureTex' luistert naar het noodnummer '112',
    - Concierge met naam 'Peter Pauli' luistert naar het noodnummer '112'

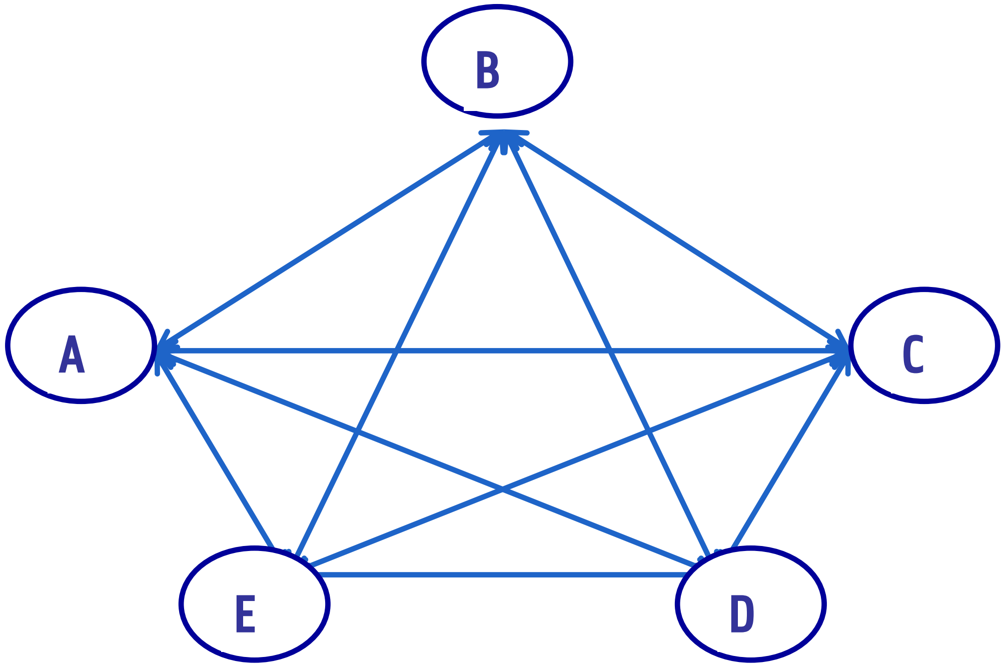
4. MEDIATOR - EVENT BROKER

MEDIATOR : MOTIVATION

Interaction between many objects

Default solution : objects contain references to each other

- Worst case : for N objects -> N*(N-1) references
- Problem : difficult to reuse object/class



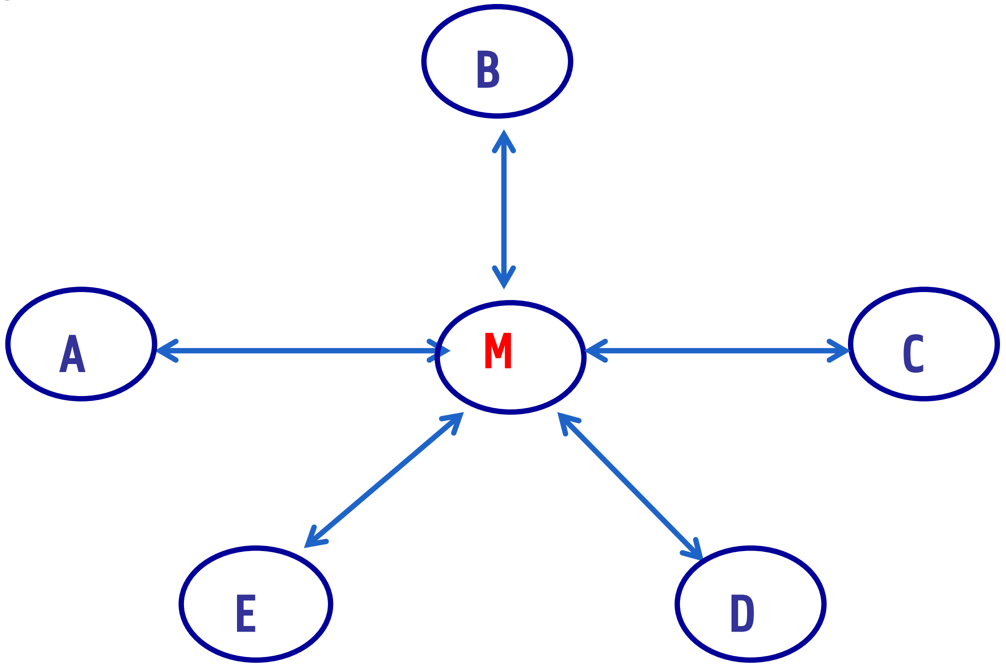
SOLUTION

Colleagues (A,B,C,D,E)

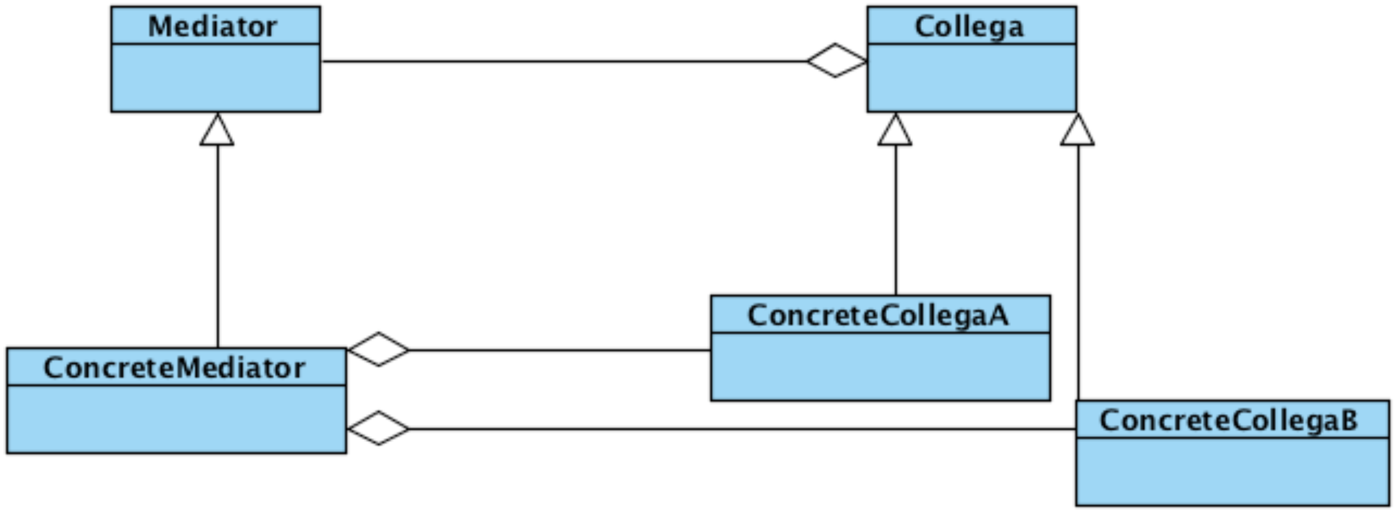
- Interacting objects
- ONLY refer to single object “Mediator”

Mediator (M)

- Knows all interacting objects
- Single point of contact
- Manages interaction



GENERAL SOLUTION



Mediator: defines interface to communicate with Colleagues

ConcreteMediator:

- coordinates Colleague objects
- knows and manages Colleague collection

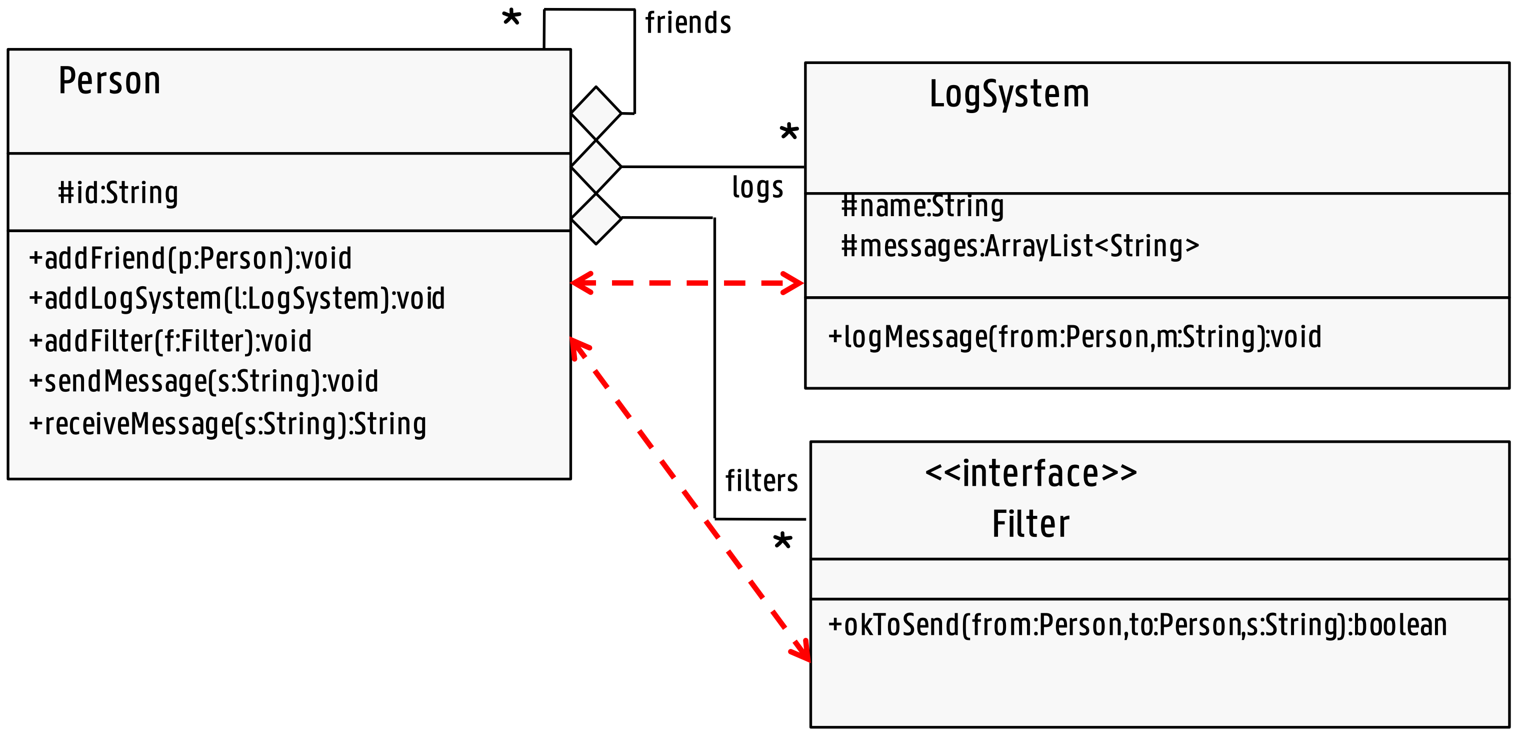
Colleague:

- knows Mediator object
- communicates through Mediator with Colleagues

EXERCISE : CHAT SERVICE

Problem

- Person can send messages to all his Friends
- Message can be filtered by a Filter object (message is only sent when all Filters agree to send the message)
- Each Person’s sent messages are logged by different LogSystems



Person
#id:String
+addFriend(p:Person):void
+addLogSystem(l:LogSystem):void
+addFilter(f:Filter):void
+sendMessage(s:String):void
+receiveMessage(s:String):String
*
friends
*
logs
LogSystem
#name:String
#messages:ArrayList<String>
+logMessage(from:Person,m:String):void
filters
*
<<interface>>
Filter
+okToSend(from:Person,to:Person,s:String):boolean

Modify to reduce coupling using Mediator

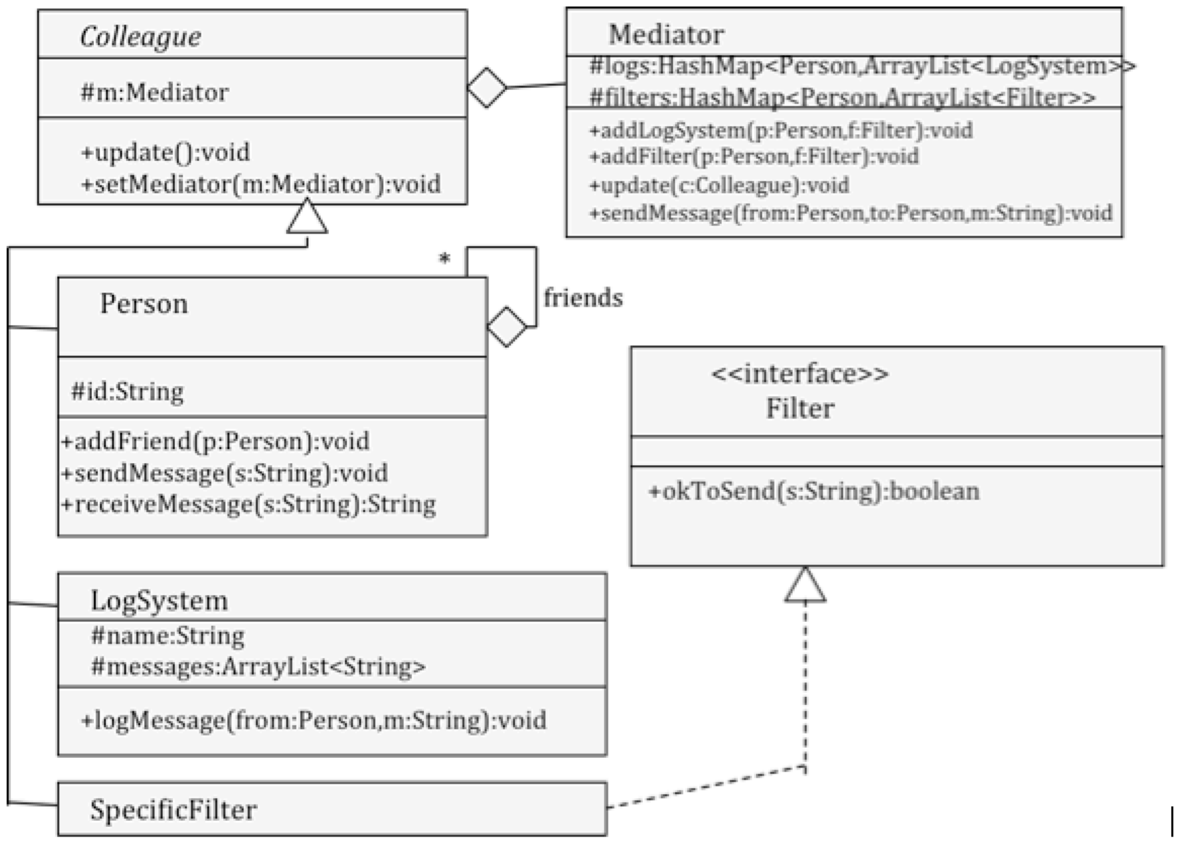
EXERCISE : CHAT SERVICE

```
public void sendMessage(String s) {
        for(Person p:friends) {
                boolean send=true;
                for(Filter f:filters) if(!(f.okToSend(this,p,s))) send=false;
                if(send) p.receiveMessage(s);
        }
        for(LogSystem l:logs)
                l.logMessage(this,s);
}
```

Modify to reduce coupling using Mediator

EXERCISE : CHAT SERVICE

Solution


Colleague
#m:Mediator
+update():void
+setMediator(m:Mediator):void
Mediator
#logs:HashMap<Person,ArrayList<LogSystem>>
#filters:HashMap<Person,ArrayList<Filter>>
+addLogSystem(p:Person,f:Filter):void
+addFilter(p:Person,f:Filter):void
+update(c:Colleague):void
+sendMessage(from:Person,to:Person,m:String):void
*
friends
Person
#id:String
+addFriend(p:Person):void
+sendMessage(s:String):void
+receiveMessage(s:String):String
<<interface>>
Filter
+okToSend(s:String):boolean
LogSystem
#name:String
#messages:ArrayList<String>
+logMessage(from:Person,m:String):void
SpecificFilter

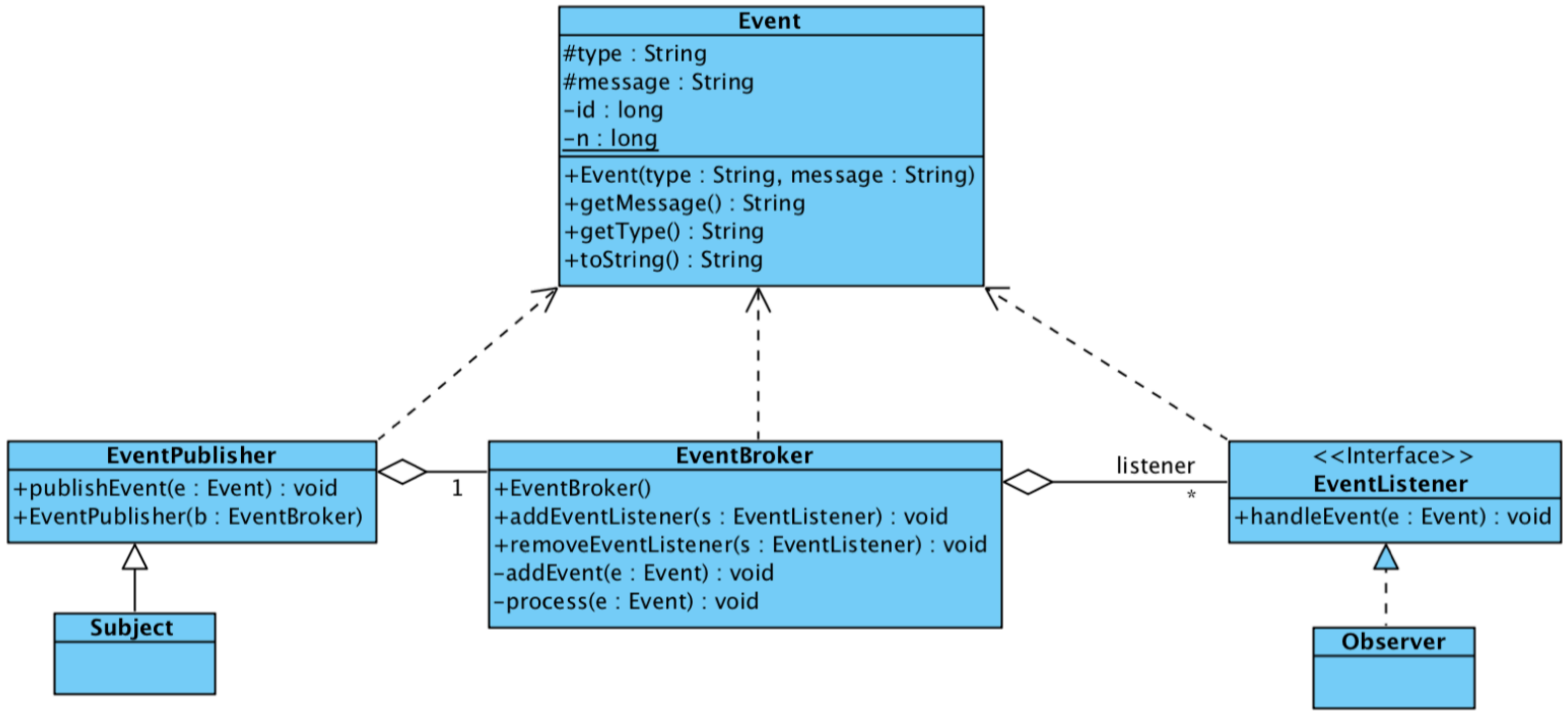
EXERCISE : CHAT SERVICE

Solution

In Mediator:

```
public void sendMessage(Person p1,Person p2,String s) {
        for(Person p:p1.getFriends()) {
                boolean send=true;
                ArrayList<Filter> fil=filters.get(p1);
                for(Filter f:fil) if(!(f.okToSend(s))) send=false;
                if(send) p1.receiveMessage(s);
        }
        ArrayList<LogSystem> log=logs.get(p1);
        for(LogSystem l:log)
                l.logMessage(""+p1+" "+s);
}
```

EVENTBROKER = MEDIATOR



Event
#type : String
#message : String
-id : long
-n : long
+Event(type : String, message : String)
+getMessage() : String
+getType() : String
+toString() : String
EventPublisher
+publishEvent(e : Event) : void
+EventPublisher(b : EventBroker)
1
EventBroker
+EventBroker()
+addEventListener(s : EventListener) : void
+removeEventListener(s : EventListener) : void
-addEvent(e : Event) : void
-process(e : Event) : void
listener
*
<<Interface>>
EventListener
+handleEvent(e : Event) : void
Subject
Observer

OEFENING 4 : MEDIATOR – EVENT BROKER

1. Programmeer alarmevent.AlarmEvent
    - erft over van Event
    - nieuw attribuut location
    - constructor met twee String-args
        - type van alarm
        - location van alarm
    - message : human-readable boodschap (bv. "ALARM! crash at Plateaustraat")
2. Programmeer alarmevent.EmergencyCallCenter
    - erft over van EventPublisher
    - implementeer incomingCall(), geeft AlarmEvents door aan EventBroker via publishEvent()
    - constructor: bijkomend argument EventBroker
3. Programmeer alarmevent.PoliceDepartment, alarmevent.Hospital en alarmevent.FireDepartment
    - implementeren EventListener
    - registreren zich bij EventBroker
    - constructor : extra argument EventBroker
4. Nieuwe alarmevent.Main-klasse : die hetzelfde gedrag test als de oorsprongkelijke Main-klasse.

OEFENING 5 : EVENT BROKER FILTER

1. Nieuwe methode in EventBroker : add/removeEventListener(String type, EventListener s) , oorspronkelijke methoden moet blijven werken !
2. Pas process() aan zodat filtering gebeurt
3. Aanpassing registraties
   - Brandweer : ‘fire’
   - Ziekenhuis : 'fire' of een 'crash'.
   - Politie : elke noodoproep

5. SINGLETON

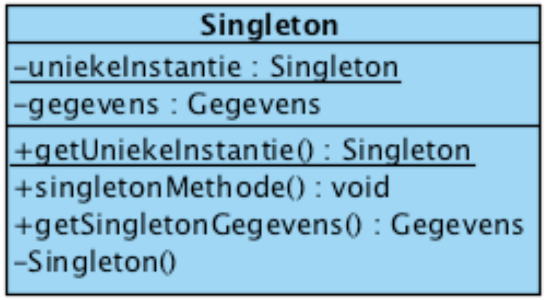
SINGLETON (CREATIONAL)

Pattern name Singleton

Problem

- how to make sure that only 1 object of a class can be instantiated ?

Solution


Singleton
-uniekeInstantie : Singleton
-gegevens : Gegevens
+getUniekeInstantie() : Singleton
+singletonMethode() : void
+getSingletonGegevens() : Gegevens
-Singleton()

- make constructor private !!!!
- ensure no default-constructor
- Singleton can be responsible for creation of unique instance

SINGLETON (CREATIONAL)

Exercise

Given class Person

```
class Person {
        protected String firstName;
        protected String lastName;
        public Person(String f, String l) {
                firstName=f;
                lastName=l;
        }
        public String toString() {
                return firstName+” “+lastName;
        }
}
```

Construct special kind of Person, Administrator. Only one Administrator is allowed in the system (Singleton).

SINGLETON (CREATIONAL)

Solution

```
class Administrator extends Person{
        protected static Administrator adm=new Administrator();
        private Administrator(){
                super("","");
        }
        public void setFirst(String f){
                firstName=f;
        }
        public void setLast(String l){
                lastName=l;
        }
        public static Administrator getAdministrator() {
                return adm;
        }
}
```

OEFENING 6 : SINGLETON

1. Pas de klasse EventBroker aan :
   - maar 1 object van de klasse mogelijk
   - unieke instantie via getEventBroker() (statisch) op te vragen
2. Constructoren van de klassen Hospital, FireDepartment, PoliceDepartment en EmergencyCallCenter 1 argument minder (geen EventBroker meer als argument)
3. Pas code van de Main-klasse aan

6. SERVICE LOCATOR (WHITEBOARD)

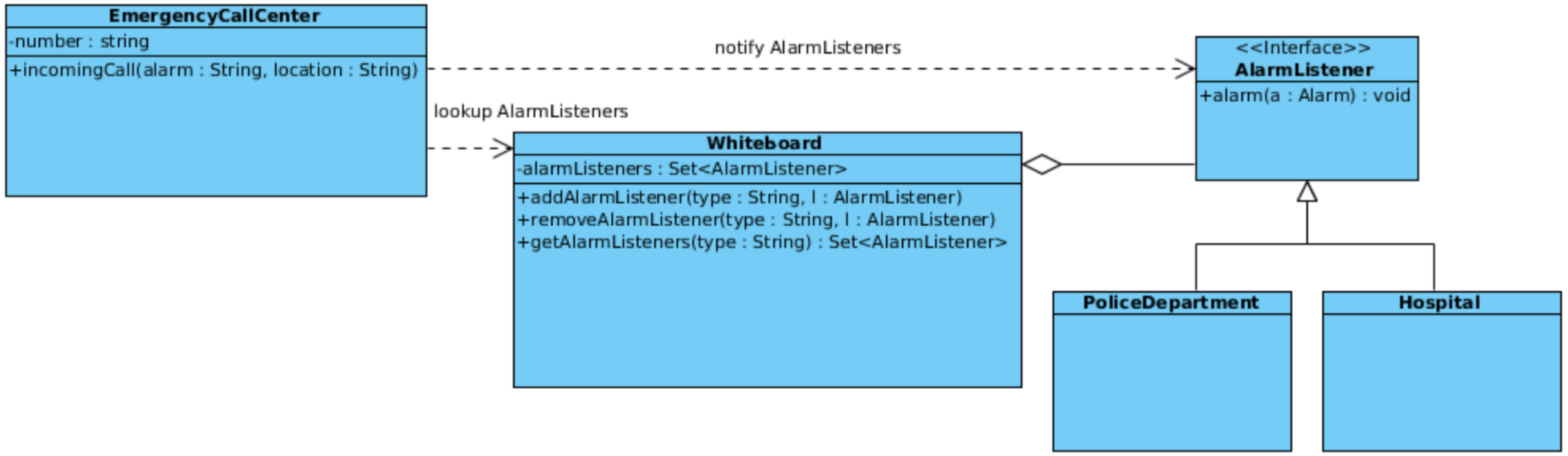
CONCEPT

Service Locator

= single point of contact in application to resolve dependencies
especially : bind implementation to interface

- often implemented as (sort of) Singleton
- acts as service repository
- static and dynamic locators

OEFENING 7 : WHITEBOARD


EmergencyCallCenter
-number : string
+incomingCall(alarm : String, location : String)
notify AlarmListeners
lookup AlarmListeners
<<Interface>>
AlarmListener
+alarm(a : Alarm) : void
Whiteboard
-alarmListeners : Set<AlarmListener>
+addAlarmListener(type : String, l : AlarmListener)
+removeAlarmListener(type : String, l : AlarmListener)
+getAlarmListeners(type : String) : Set<AlarmListener>
PoliceDepartment
Hospital

OEFENING 7 : WHITEBOARD

1. Klasse alarmwhiteboard.Whiteboard
   - Singleton
   - Mogelijkheid tot filteren op eventtype
2. Klasse alarmwhiteboard.EmergencyCallCenter
   - Vraagt alle AlarmListeners op aan Whiteboard
   - Oproep van type 'crash' of 'fire' : 1 ziekenhuis alarmeren (round robin)
   - Brandweer en Politie : idem als oefening 5, 6
3. Klasse alarmwhiteboard.Main